Christian Kreibich
Marko Jahnke (Eds.)

LNCS 6201

Detection of Intrusions and Malware, and Vulnerability Assessment

7th International Conference, DIMVA 2010
Bonn, Germany, July 2010
Proceedings

Springer

Christian Kreibich   Marko Jahnke (Eds.)

Detection of Intrusions and Malware, and Vulnerability Assessment

7th International Conference, DIMVA 2010
Bonn, Germany, July 8-9, 2010
Proceedings

Springer

Volume Editors

Christian Kreibich
International Computer Science Institute
Berkeley, USA
E-mail: christian@icir.org

Marko Jahnke
Fraunhofer FKIE
Wachtberg, Germany
E-mail: jahnke@fgan.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): C.2, K.6.5, D.4.6, E.3, H.4, K.4.4

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN	0302-9743
ISBN-10	3-642-14214-1 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-14214-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

On behalf of the Program and Steering Committees it is our pleasure to present to you the proceedings of the 7th GI International Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), documenting the work presented at the conference this year. As in the past, the conference brought together international experts from academia, industry, and government to present and discuss novel security research.

This year the 27 members of the Program Committee received 34 submissions from 18 countries. The committee, selected to represent a balanced mixture of both mature and young excellence in the field, honesty, and good judgement, carefully reviewed and evaluated all submissions at least threefold, according to scientific novelty, relevance, and technical quality. The final selection took place on March 29, 2010 at the Technische Universität München, Germany. In the end, we accepted 12 papers for publication and presentation at the conference, including two extended abstracts.

DIMVA 2010 took place at the Centre of Sciences in Bonn, Germany, on July 8 and 9. The program featured work from a wide range of topics in security, grouped into five sessions that are reflected in the chapters of the proceedings you are now reading. In addition, the conference featured three invited talks which greatly contributed to the event. We are very grateful to José Nazario (Arbor Networks), Carel van Straaten (Spamhaus), and Marc Dacier (Symantec/Eurecom) for their insightful and entertaining presentations. Thanks also to Sven Dietrich (Stevens Institute of Technology) for once again organizing and hosting the rump session, which contained a diverse range of forthcoming work.

Credit for the quality of a conference is first and foremost due to the authors of all submitted work. Their creativity and hard work was reflected in the discussions at the meeting in Munich, and we are deeply indebted to their efforts. Our sincere thanks go to the members of the Program Committee as well as the external reviewers for working the review tasks and shepherding into their busy schedules. We likewise thank the Technische Universität München for kindly providing meeting facilities. Finally, we are truly grateful for our sponsors' contributions in making DIMVA 2010 possible.

For further information about DIMVA 2010 please visit the conference website at `http://www.dimva.org/dimva2010`.

July 2010

Christian Kreibich
Marko Jahnke

Organization

Organizing Committee

Rump Session Chair	Sven Dietrich, Stevens Institute of Technology, USA
General Chair	Marko Jahnke, Fraunhofer FKIE, Germany
Program Chair	Christian Kreibich, International Computer Science Institute, USA
Sponsor Chair	Felix Leder, University of Bonn, Germany
Publicity Chair	Sebastian Schmerl, Technical University of Cottbus, Germany
Local Chair	Jens Tölle, Fraunhofer FKIE, Germany
Rump Session Chair	Sven Dietrich, Stevens Institute of Technology, USA

Program Committee

Michael Bailey	University of Michigan, USA
Herbert Bos	Vrije Universiteit Amsterdam, The Netherlands
Juan Caballero	Carnegie Mellon and UC Berkeley, USA
Hervé Debar	Télécom SudParis, France
Sven Dietrich	Stevens Institute of Technology, USA
Holger Dreger	Siemens CERT, Germany
Ulrich Flegel	SAP Research, Germany
Chris Grier	UC Berkeley, USA
Guofei Gu	Texas A&M University, USA
Thorsten Holz	Vienna University of Technology, Austria
Piotr Kijewski	NASK/CERT, Poland
Engin Kirda	Eurecom, France
Christopher Kruegel	UC Santa Barbara, USA
Pavel Laskov	University of Tübingen, Germany
Wenke Lee	Georgia Institute of Technology, USA
Corrado Leita	Symantec Research Labs, France
Kirill Levchenko	UC San Diego, USA
Michael Meier	Technical University of Dortmund, Germany
Tyler Moore	Harvard University, USA
Ludovic Mé	Supélec, France
Lexi Pimenidis	iDev GmbH, Germany
Moheeb Rajab	Google/Johns Hopkins University, USA
Sebastian Schmerl	Brandenburg University of Technology, Germany
Robin Sommer	ICSI/LBNL, USA

Henry Stern	Cisco/Ironport, USA
Diego Zamboni	HP Professional Services, Mexico

External Reviewers

Andreas Moser
Clemens Kolbitsch
Craig Williams
Debmalya Biswas
Eric Totel
Frédéric Tronel
Markus Engelberth
Matthias Neugschwandtner
Valérie Viêt Triêm Tông

Steering Committee

Chairs	Ulrich Flegel, SAP Research, Germany
	Michael Meier, Technical University of Dortmund, Germany
Members	Roland Büschkes, RWE AG, Germany
	Danilo M. Bruschi, Università degli Studi di Milano, Italy
	Hervé Debar, France Telecom R&D, France
	Bernhard Haemmerli, Acris GmbH & HSLU Lucerne, Switzerland
	Marc Heuse, Baseline Security Consulting, Germany
	Klaus Julisch, IBM Zurich Research Lab, Switzerland
	Christopher Kruegel, UC Santa Barbara, USA
	Pavel Laskov, University of Tuebingen, Germany
	Robin Sommer, ICSI/LBNL, USA
	Diego Zamboni, IBM Zurich Research Lab, Switzerland

DIMVA 2010 was organized by the Special Interest Group Security—Intrusion Detection and Response (SIDAR)— of the German Informatics Society (GI), in cooperation with Fraunhofer Institute for Communication, Information Processing and Ergonomics (FKIE) and the Communication and Distributed Systems Group at Universität Bonn, Germany.

Sponsoring Institutions

We sincerely thank our sponsors Qualys and FGA Global Advisors as well as our media partner Virus Bulletin for their support of DIMVA 2010.

QUALYS®
ON DEMAND SECURITY

Table of Contents

HookScout: Proactive Binary-Centric Hook Detection[⋆]

Heng Yin[1], Pongsin Poosankam[2,3], Steve Hanna[2], and Dawn Song[2]

[1] Syracuse University, Syracuse NY 13104
heyin@syr.edu
[2] UC Berkeley, Berkeley CA 94720
{sch,dawnsong}@cs.berkeley.edu
[3] Carnegie Mellon University, Pittsburgh PA 15213
ppoosank@cs.cmu.edu

Abstract. In order to obtain and maintain control, kernel malware usually makes persistent control flow modifications (i.e., installing hooks). To avoid detection, malware developers have started to target function pointers in kernel data structures, especially those dynamically allocated from heaps and memory pools. Function pointer modification is stealthy and the attack surface is large; thus, this type of attacks is appealing to malware developers. In this paper, we first conduct a systematic study of this problem, and show that the attack surface is vast, with over 18,000 function pointers (most of them long-lived) existing within the Windows kernel. Moreover, to demonstrate this threat is realistic for closed-source operating systems, we implement two new attacks for Windows by exploiting two function pointers individually. Then, we propose a new *proactive* hook detection technique, and develop a prototype, called *HookScout*. Our approach is *binary-centric*, and thus can generate hook detection policy without access to the OS kernel source code. Our approach is also *context-sensitive*, and thus can deal with polymorphic data structures. We evaluated HookScout with a set of rootkits which use advanced hooking techniques and show that it detects all of the stealth techniques utilized (including our new attacks). Additionally, we show that our approach is easily deployable, has wide coverage and minimal performance overhead.

⋆ This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170, by the Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Air Force Office of Scientific Research, or the Army Research Office.

1 Introduction

As malware evolves to be increasingly sophisticated and stealthy the operating system kernel has become a popular target for attacks [10]. Once the OS kernel is compromised, attackers control every aspect of the victim's system: they can implement illicit functionality directly, hide malicious user-level components, and make themselves and their components difficult to be detected and removed. To achieve these malicious goals, malware tends to make *persistent* control flow modifications, and in other words, *hooks* are installed in the victim's system. A previous study shows that the 24 out of 25 kernel rootkits in the survey make persistent control flow modifications [16].

Old-fashioned malware installs hooks by either tampering with certain kernel code regions or overwriting entries in well-known data regions. These well-known data regions include SSDT (System Service Descriptor Table), IAT (Import Address Table) and IDT (Interrupt Descriptor Table). Current hook detection tools, such as VICE [3], System Virginity Verifier [23] and IceSword [12], verify the integrity of all code regions and known data regions, and thus have successfully defeated these hooking techniques. To evade detection, malware has moved its target to previously unknown and unexplored data regions. In particular, malware overwrites function pointers in kernel data structures, which usually reside on heaps or in dynamically allocated memory pools. These kernel data structures maintain critical system states and configurations and contain important function pointers [9]. The number of function pointers in the kernel space can be large, and without in-depth knowledge of these kernel data structures, it is very difficult to locate and validate them. Therefore, this new hooking technique is ideal for attackers to install stealthy hooks.

To tackle this severe security problem, several systems have been proposed. Systems such as HookFinder [32], K-Tracer [14], and PoKeR [20] take a *postmortem* approach. These systems analyze a new kernel rootkit to extract its attack mechanism after the rootkit has caused damages and been caught. Then the extracted hooking mechanisms can be used to update the hook detection policy for guarding against similar attacks in the future. However, postmortem analysis is not an effective defense because a large number of kernel objects and function pointers exist in memory, and the number of potential locations for placing this kind of hook is enormous. This means that even when a new attack region is discovered and blocked, attackers can simply locate and exploit another data structure to achieve the same objective.

Other systems like SBCFI [16], Gibraltar [1], HookSafe [31], and SFPD [4] take a *proactive* approach. Instead of dissecting malware to figure out what regions have been used for placing hooks, these systems examine the operating system to understand where these function pointers are and how they are used, and then generate a hook detection policy. This policy can be used to traverse kernel data structures, locate function pointers in these kernel objects, and determines if they point to legitimate targets. In order to know how to traverse kernel data structures, these systems perform static source code analysis, extract type graphs, and generate traversal templates. However, in many cases, we do not have access to the source code of the operating system, such as Microsoft Windows. Therefore, the requirement of access to source code would impede third-party security practitioners to deploy these systems. Moreover, since we do not have source code of third-party device drivers and modules, hooks in these components will also be overlooked by static source code analysis.

In this paper, we take a proactive approach. We first systematically study the attack space and nature of this new hooking technique. We perform *whole-system dynamic binary analysis* to monitor kernel memory objects and keep track of function pointers propagating in the kernel space. By directly observing how the operating system is operating at the binary code level, we conduct a quantitative measurement study on this attack vector. To further demonstrate that this new threat is realistic, we implement two keyloggers by exploiting two different function pointers. Since these two attacks are new, they can successfully evade all the existing hook detection tools. To effectively defeat this threat, we propose a novel approach for proactive hook detection. We aim to derive the hook detection policy directly from the knowledge about kernel memory objects and function pointers. Compared to previous approaches, our approach is binary-centric. That is, it performs analysis directly on binary code, without assuming the access to source code. Therefore, this approach can be widely deployed on the system with closed-source OS kernel and third-party components.

To demonstrate the efficacy of our approach, we built a prototype, called *HookScout*. It consists of two subsystems: *analysis subsystem* and *detection subsystem*. The analysis subsystem performs binary code analysis on the operating system kernel and automatically generates a policy for hook detection. The detection subsystem residing on the user's machine enforces the generated policy and detects hooks in the kernel space.

In summary, this paper makes the following contributions:

– To assess the attack space of function pointer hooking technique, we conduct a systematic measurement study. It shows that the space of this new attack vector is enormous: there are around 18,000 function pointers in the Windows kernel space in total, the majority of these function pointers (90%) are long-lived, and very few (3%) ever change in their lifetime.
– To further demonstrate the severity of this problem, we identify two function pointers in the keyboard driver, and implement two new keyloggers by exploiting these two function pointers individually. These two new attacks can successfully evade the existing hook detection tools.
– We propose a binary-centric approach for generating a hook detection policy.
– We design and implement a prototype called HookScout, to demonstrate the effectiveness and efficiency of our approach.
– We evaluated HookScout with a popular closed-source operating system, Windows XP with Service Pack 2. The analysis subsystem can generate the hook detection policy within a few hours. The generated policy can achieve very high coverage (over 95%). The detection subsystem was able to detect all the rootkit samples in our sample set, including the two new synthetic attacks. We also showed that the performance overhead of this detection component is negligible.

```
typedef struct {
  int type;
  char name[512];
} OBJ_HEAD;

typedef struct {
  OBJ_HEAD head;
  LIST_ENTRY link;
  int (*open)(char *n, char *m);
  ...
} FILE_OBJ;

typedef struct {
  OBJ_HEAD head;
  LIST_ENTRY link;
  int state;
  int (*ioctl)(char *buf, int size);
  ...
} DEVICE_OBJ;
```

```
LIST_ENTRY ObjListHead;

CreateFile() {
  FILE_OBJ *f = malloc(sizeof(FILE_OBJ));
  ...
  InsertTailList(&f->link, &ObjListHead);
  ...
}

CreateDevice() {
  DEVICE_OBJ *d = malloc(sizeof(DEVICE_OBJ));
  ...
  InsertTailList(&d->link, &ObjListHead);
  ...
}
```

Fig. 1. Code snippet that illustrates a polymorphic linked list. The doubly-linked list starting with `ObjListHead` contains objects of two different types, `FILE_OBJ` and `DEVICE_OBJ`. This code snippet is a simplified example inspired by the Windows kernel hash table for organizing kernel objects.

2 Problem Statement

In this paper, we take a proactive approach to detecting function pointer hooking attacks. That is, we want to generate a hook detection policy that can be used to thoroughly locate and validate the function pointers in the kernel space. To facilitate deployment, we want to directly derive the hook detection policy from the binary code of OS kernel and device drivers. As a result, our technique can be widely used to protect closed-source operating systems like Windows and proprietary device drivers.

Furthermore, we have to cope with type polymorphism. That is, the actual type of a polymorphic data object is determined by the context under which this data object is created. Figure 1 illustrates such a case. A linked list `ObjListHead` stores objects of two different types, `FILE_OBJ` and `DEVICE_OBJ`. These two types share a common head structure `OBJ_HEAD`, while the remaining portions in these two types are different. The function `CreateFile` creates a `FILE_OBJ` object, and the function `CreateDevice` creates a `DEVICE_OBJ` object. If we are not aware of the different creation contexts of these two types of objects, we will not notice this type polymorphism, and thus will not locate and traverse the function pointers in these objects. Indeed, in the Windows kernel, many different types of kernel objects, such as files, devices, drivers, and processes, are managed in a centralized hash table [24]. These kernel objects keep important system states and function pointers. Thus, it becomes critical to traverse and verify the function pointers in these polymorphic data structures.

3 Approach Overview

At a high level, our approach consists of two subsystems: *analysis subsystem* and *detection subsystem*. The analysis subsystem performs static and dynamic binary analysis on a given distribution of an operating system, and generates a policy for hook detection. The detection subsystem is deployed on users' machines with the same distribution of the operating system installed. The detection subsystem enforces the policy generated by the analysis subsystem and actively detects hooks at runtime. Note that the system protected by the detection subsystem does not need to be the same as the one analyzed by the analysis subsystem. These two systems only need to have the same set of binary modules (including main kernel modules and common device drivers). For instance, if the analysis subsystem generates a policy for Windows XP Professional SP2, then this policy can be used for hook detection on any machines with Windows XP Professional SP2 installed. Of course, when a new kernel update is released, we need to generate a corresponding policy for it. Since our system can generate the new policy in a fully automatic manner within a few hours (as demonstrated in Sect. 5.2), we believe our approach is practical for wide deployment. In this section, we give a description of the analysis subsystem and the detection subsystem.

3.1 Analysis Subsystem

We perform *whole-system dynamic binary analysis* on the operating system for which we want to generate the hook detection policy. In other words, we run the entire installation of an operating system along with common applications, and observe how the OS kernel behaves. In particular, we are interested in the kernel's behaviors in two aspects: (1) because function pointers become the targets for installing hooks, we want to know how function pointers are created, distributed, and used; and (2) we want to monitor memory objects that are allocated either statically or dynamically. Then we can have a complete view of the kernel memory space, in terms of where memory objects are and where function pointers are located within these memory objects. Such a complete view enables us to quantitatively and qualitatively assess the space and characteristics of kernel hooking attacks, and helps us determine appropriate detection policies.

Furthermore, we want to generate the hook detection policy by inferring invariants from this complete view (or more precisely, a series of views). In particular, we need to determine the layout of each memory object, in terms of where the function pointers are located within the memory object and what properties these function pointers have (e.g. whether they change over time). This process is essentially analogous to inferring the type of a memory object.

In order to address polymorphic data structures, we propose a context-sensitive analysis technique for inferring the policy. We take into consideration the execution context where each memory object is created. We rely on the fact that memory objects created in the same execution context are of the same or compatible types. That is, these memory objects should have the same or compatible layouts.

For the example in Fig. 1, all the memory objects created in `CreateFile` are of type `FILE_OBJ`, and all the objects allocated in `CreateDevice` are of type `DEVICE_OBJ`.

By tracking function pointers and monitoring memory objects, we are able to obtain the concrete layout for each memory object at a specific moment (i.e. exactly where the function pointers exist in an object). In order to locate and validate function pointers in the future, we need to extract a generalized layout for all the memory objects that are created in the same execution context. To this end, we devise a *generalization process*, which produces a generalized layout for a given execution context by merging concrete layouts of multiple memory objects created under that context. Such a generalized layout associated with the execution context is a *context-sensitive template* in our policy. As a result, the generated policy consists of a list of context-sensitive templates.

3.2 Detection Subsystem

To enforce the generated policy, the detection subsystem needs to be context-sensitive as well. That is, the detection subsystem monitors the allocation and deallocation of memory objects, extracts the execution context when each memory object is created, and looks up the policy template corresponding to this execution context. Then according to the template associated with this memory object, the detection subsystem will periodically verify the validity of function pointers in this memory object. Continuing with the example given in Fig. 1, we would monitor memory objects created by `CreateFile` and `CreateDevice`. The creation context will be used to look up policy. Therefore, the policy template applied to the memory objects created by `CreateFile` will be different than the one applied to those created by `CreateDevice`.

4 System Design and Implementation

To demonstrate the feasibility of our approach, we design and implement a system, called HookScout. We illustrate the architecture of HookScout in Fig. 2. The analysis subsystem consists of two components: *monitor engine* and *inference engine*. The monitor engine watches the behaviors of the operating system

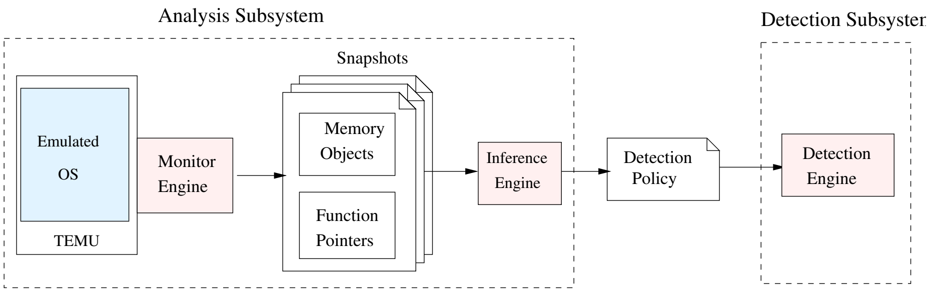


Fig. 2. Architecture of HookScout

of interest. More specifically, it monitors memory objects that are created either statically or dynamically, and keeps track of function pointer propagating in the kernel memory space. To perform this fine-grained dynamic binary analysis, we build the monitor engine on top of TEMU [33,28]. TEMU is a dynamic binary analysis platform based on an open-source whole-system emulator, QEMU [2]. During the dynamic analysis, the emulated operating system is exercised with common test cases, and the monitor engine periodically records system snapshots, including the state of memory objects and function pointers. Taking the snapshots as inputs, the *inference engine* performs context-sensitive analysis and generates the policy for hook detection. In the detection subsystem, the *detection engine*, located in the system to be protected, enforces the policy generated by the analysis subsystem and detects hook in the kernel space at runtime.

4.1 Analysis Subsystem

Monitor Engine. The monitor engine is responsible for: (1) monitoring memory objects; (2) tracking function pointers; and (3) periodically generating snapshots of the OS kernel.

Monitoring Memory Objects. The monitor engine watches memory objects that are allocated either statically or dynamically. A static memory object is a memory region statically allocated for a kernel module for storing global variables, while a dynamic memory object is allocated dynamically from heaps and memory pools. To monitor kernel memory objects, we need to have basic knowledge about kernel memory management. For Windows, we know that `MmLoadSystemImage` is used to load a kernel module. `RtlAllocateHeap` and `RtlFreeHeap` are used for heap allocation and deallocation. Additionally, `ExAllocatePoolWithTag` and `ExFreePoolWithTag` are the root APIs for allocating and freeing memory pools. We intercept these kernel functions. When a memory object is newly allocated, we extract its base address and size and keep this information in the memory object state. We maintain the information for static and dynamic memory objects in an active memory object list. When a memory object is freed, we simply remove its information from the active memory object list. Some memory objects are special and are statically allocated and pointed by system registers. For example, `IDTR` is a register pointing to a static memory region for storing interrupt descriptor table and `FS` is a segment register pointing to a static memory region for storing the current execution context in Windows. Since these special static memory regions may contain function pointers, we also monitor these objects.

For dynamically allocated memory objects, we also need to obtain the execution contexts when they are created. The execution contexts are later used by the inference engine to perform context-sensitive analysis and generate policy. We will describe how to obtain the creation context while discussing the inference engine.

Tracking Function Pointers. The monitor engine identifies where each function pointer is initialized and then keeps track of the function pointer as it propagates throughout the system.

To identify the initial function pointers, we leverage the following fact: in Windows (or other relocatable OS kernels), all the modules (including the kernel itself) are generated to be relocatable. All references with absolute addresses to the statically allocated code and data sections for each kernel module have to be placed in the relocation table (e.g., `.reloc` for PE format). In this way, if the executable loader decides to load a kernel module into a different memory region than assumed, it can go through this relocation table to update these references. Due to the fact that a function pointer refers to the absolute address of a function within a relocatable module, it must appear in the relocation table. Then to determine initial assignments of function pointers, we can check for each entry in the relocation table whether it points to a function entry. Function entries can be determined through standard static binary analysis.

```
0005ed61: mov  [ebp-50h], 00015141h
```

For example, the instruction shown above moves a constant number into a memory location. This constant's location (0005ed64h[1]) appears in the relocation table and the actual value (00015141h) of this constant points to the entry point of a function. Then we can determine that this instruction copies a function pointer into a memory location on the stack.

Moreover, an instruction may also reference a function from another module as a function pointer. In this case, this function appears in the import address table (i.e., IAT).

```
000146ae: mov  eax, ds:[00013464h] ; READ_PORT_UCHAR
000146b3: mov [0001390ch], eax
```

For example, the two instructions above moves a function READ_PORT_UCHAR defined in IAT to a global variable located at 0001390ch. Therefore, we need to check IAT for initial function pointers as well.

We developed a plugin to IDA Pro [13] to perform this static analysis. This plugin takes a kernel module as input, automatically enumerates the entries in the relocation table and import address table, identifies the function boundaries, and determines the locations of initial function pointers. By performing this analysis on all kernel modules (including device drivers), we have identified all the initial function pointers in the kernel.

Then, to keep track of function pointers propagating over the system, we perform whole-system dynamic taint analysis, as many previous systems do [7, 34, 32, 5, 6]. That is, we mark the initial function pointers as tainted, and during the execution of each instruction, if any source operand is tainted, we mark the destination operand is tainted by checking data dependency between operands. In this way, we can track which data structures and locations these function pointers are copied into. In the implementation, we make use of the taint analysis functionality in TEMU.

[1] The instruction starts at 0005ed61h. The first three bytes are used for opcode and the first operand. So this immediate operand is located at 0005ed64h.

Therefore, relying on the relocatable property of initial function pointers and dynamic taint analysis, we can identify the vast majority of function pointers (if not all) in the kernel memory space. For the OS kernels that are not relocatable (e.g., Linux), we cannot use this technique. Alternatively, we can examine the concrete value for each memory word within a memory object to see if it points to a kernel function entry.

Inference Engine. The inference engine takes the system snapshots as input, performs context-sensitive analysis, and infers a policy for hook detection.

Determining Execution Context. In general, we want to know who creates a memory object. From the binary code point of view, this information can be obtained from the call stack when the memory allocation routine is invoked. From the call stack, we obtain the return address of the memory allocation function call. Considering that the function that invokes the memory allocation routine is called by another function, we actually obtain a chain of return addresses. Therefore, we define the execution context to be a chain of return addresses and the size to be allocated. Taking into account that kernel modules can be relocated to different locations in different executions and different systems, for each return address, instead of the absolute address, we keep the relative address — the offset to the base of the module where this return address is located.

Note that the number of return addresses to be included determines the level of context sensitivity in our analysis. The more return addresses, the more context-sensitive our analysis is. For example, if function A and function B call function C, and function C allocates memory objects for A and B, the analysis with only one return address will think memory objects created in C are of the same type, which may not be true. In comparison, the analysis with two return addresses will treat memory allocated for function A and B differently. Hence, the increase of context sensitivity results in better analysis precision. However, the increase of context sensitivity also leads to more complexity in our analysis. First, it means that we need to perform more thorough test cases to cover more execution contexts. Second, it means the number of templates in the policy would increase drastically. Therefore, we need to determine an appropriate level of context sensitivity. Fortunately, as shown in Sect. 5.2, analysis with very small number (1 to 3) of return addresses can already generate high-quality policies with very high coverage.

Inferring Policy Templates. We merge the layouts of multiple dynamic memory objects with the same execution context into a generalized layout. Static memory objects are different because they are not associated with execution contexts so we uniquely identify them by their names (e.g., module names or register names). Thus, for static memory objects, we merge them according to their names.

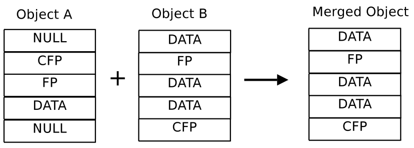
Within a memory object, we classify each field (e.g., 4-byte memory in 32-bit architecture) into one of the following types: NULL, FP, CFP, and DATA. NULL is for a field that holds a concrete value 0. FP identifies a function pointer, which we determine by checking if this field is tainted. CFP indicates a constant function pointer that has never changed its value in its lifetime. To determine a

$$\begin{tikzcd}[row sep=small]
\text{DATA} \arrow[d, dash] \\
\text{FP} \arrow[d, dash] \\
\text{CFP} \arrow[d, dash] \\
\text{NULL}
\end{tikzcd}$$

(a)

⊔	**NULL**	**CFP**	**FP**	**DATA**
NULL	NULL	CFP	FP	DATA
CFP	CFP	CFP	FP	DATA
FP	FP	FP	FP	DATA
DATA	DATA	DATA	DATA	DATA

(b)



(c)

Fig. 3. Join operator ⊔ used in merging object layouts, shown using a lattice (a), an operation table (b), and an example (c)

CFP, we check if this field is tainted in the current snapshot, and its concrete value remains unchanged in previous snapshots since this field is initialized. Thus, CFP is a subset of FP. DATA specifies a field that holds a data value, which is not tainted and does not hold a concrete value 0.

To merge a set of observed object layouts into a single generalized layout, we conservatively infer the most general type for each field, according to the ordering shown in Fig.3 (a). In the order NULL, CFP, FP, and Data, each type covers more possibilities than the earlier ones, so we generalize to the most specific type that includes all observations. This generalization corresponds to the join operator ⊔ in a simple linearly-ordered lattice. A corresponding matrix for this join operation ⊔ is also shown in Fig.3 (b). For instance, if one type is DATA and the other is a function-pointer type FP or CFP, the field might contain either a function pointer or data. To be conservative, we mark it as DATA in the generalized layout. Similarly, if a function-pointer field was sometimes constant and sometimes not constant, it is conservatively non-constant in the merged layout: CFP ⊔ FP = FP. We illustrate a concrete example how two memory objects are merged in Fig.3 (c).

As we will show in Sect.5.1, the vast majority of function pointers are constant. In other words, they never change during their whole lifetime. Thus, the generalized layouts can be directly used as a policy to detect hooks that make modifications on these constant function pointers. In the current implementation of HookScout, we employ this simple policy. This policy does not protect non-constant function pointers. We leave it as our future work to investigate more sophisticated policies for protecting non-constant function pointers. Note that so far the generated policy is a raw policy, including all templates. For the final policy to be enforced on users' machines, we only need to include the templates that contain CFP fields, which is only a small portion of all templates, as shown in Sect.5.2.

4.2 Detection Subsystem

The detection engine resides on a user's machine to detect violations of the hook detection policy generated by our analysis subsystem. We are aware that the detection engine can be implemented in at least two ways. First, it can be implemented as a kernel module inside the protected operating system. Second, it can be implemented inside a virtual machine monitor to detect attacks happening in a virtual machine. While the first approach is easy to implement and deploy, the second approach is more resilient to various attacks. In the current implementation of HookScout, we implement a proof-of-concept detection engine as a kernel module, mainly for demonstrating the effectiveness of our approach. We realize that malware is able to subvert our detection component, like any other security products sitting in the same execution environment as malware. We leave a more secure implementation as future work.

In the kernel module, we intercept the same set of kernel functions for monitoring memory objects, as those in the monitor engine. When a memory object is created, we extract its execution context and determine if there is a policy template associated with this execution context. If not, we skip this memory object. For those memory objects that are associated with policy templates, we periodically check if the constant function pointers within them hold different values than before. A different value indicates a hooking attack. When a memory object is freed, we remove it from the active object list.

As the kernel functions to be intercepted are not in the SSDT, SSDT hooking is not an option to hook these functions. Instead, we hot patch the entry of each of these functions. That is, we place a `jmp` instruction into the function entry, making the execution redirected into the detection engine. The kernel module is configured to be loaded at the earliest stage of boot time, in order to monitor the memory objects as early as possible.

Note that this periodical checking approach may still leave room for transient attacks on function pointers, depending on the frequency of checking. A more secure approach is to enforce our detection policy with HookSafe [31], where a shadow memory is maintained for protected function pointers and unauthorized changes can be detected instantly.

5 Evaluation

In the experiments, we aim to evaluate our system in the following aspects. In Sect.5.1, we quantitatively assess the attack space and characteristics of kernel-space hooking attacks. In Sect.5.2, we evaluate the analysis subsystem of HookScout, with respect to the coverage rate of the generated policy, the influence of context sensitivity to the quality of the generated policy, and performance overhead. In Sect.5.3, we evaluate the detection subsystem of HookScout, in terms of detecting real-world kernel rootkits, false alarms, and performance overhead.

Experiment Setup. Our experiments proceeded as follows. We first ran the analysis subsystem of HookScout to monitor and analyze a given operating system. To demonstrate that HookScout can work with closed-source operating systems, we chose Windows XP Professional Edition with Service Pack 2, a popular platform targeted by the majority of malware samples. During the analysis, we exercised the monitored operating system with a series of test cases that activate various

OS subsystems, including filesystem, networking, process and thread management, and so on.

It took approximately 25 minutes to boot up the Windows XP with our monitor engine and execute the test cases. Meanwhile, the monitor engine recorded system snapshots every 15 seconds. The snapshot contains the states of memory objects and function pointers. Therefore, 100 snapshots were recorded for each run. In total, we performed 3 different runs, which rendered a total of 300 snapshots. Then on these snapshots, we assessed the attack space and characteristics, and generated policy for hook detection.

We ran the analysis subsystem of HookScout on a Linux machine with a dual-core 3.0GHz CPU and 4GB RAM. We ran a Windows XP Professional SP2 disk image inside QEMU with 512MB allocated memory. We installed the detection subsystem on a machine with a 3.0GHz CPU and 4GB RAM and Windows XP Professional SP2.

5.1 Attack Space and Characteristics

By monitoring system execution and tracking function pointers in the kernel, we are able to assess the attack surface and characteristics of potential kernel hooking attacks.

First of all, we want to know how many function pointers exist in kernel space during the execution. This indicates the space of this attack vector. To explore this question, we picked the first run, and for each snapshot in that run, we counted the total number of function pointers in that snapshot[2]. Figure 4 shows the total number of function pointers over the 25-minute execution. We can see that the total number of function pointers climbs up in the first 5 minutes of system boot-up, and then fluctuates around $18,000$ during the execution of test cases. If every function pointer could be potentially exploited, the space of kernel hooking attacks is enormous. Figure 4 also shows the number of function pointers in dynamically allocated memory objects. Because these function pointers cannot be easily located and verified by traditional rootkit detection methods, they are more attractive to attackers. We can see that the number of function pointers in dynamically allocated memory objects is fairly high, around $8,000$. Therefore, there is a large attack surface for attackers to utilize in the OS kernel.

Then, we want to know how long these function pointers live in the kernel space. Since we aim to detect persistent control flow modifications, attacks would target at long-lived function pointers instead of transient ones. Therefore, we want to know how many function pointers are long-lived. We used the last snapshot in the first run as a starting point, and looked backward at each of previous snapshots. If we see a function pointer exists in one snapshot but not in the snapshot before it, we treat this snapshot as the birth time of this function pointer. Figure 5 shows the cumulative distribution function (CDF) of the function pointers' lifetime in the last snapshot of the first run. We can see that around 10% function pointers only lived less than two minutes, and approximately 90% function pointers lived longer than 17 minutes, and very few lived in between.

[2] Note that all runs had similar characteristics.

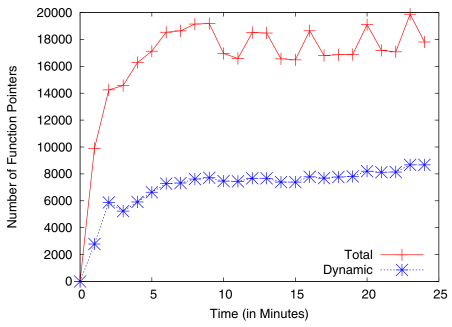


Fig. 4. Attack Space

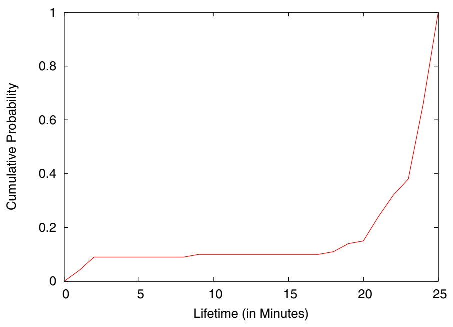


Fig. 5. Lifetime Distribution

Moreover, we want to know how frequently these function pointers change their targets during the execution. To answer this question, we examined all these snapshots, and for each of function pointers in these snapshots, checked if its concrete value was different in any of previous snapshots during its lifetime. We observe that up to 3.63% function pointers have ever changed during their lifetime. This observation indicates that a simply policy would suffice to validate the vast majority of function pointers.

Two Synthetic Keyloggers. To further assess the severity and practicality of function pointer hooking attack, we play on the attacker's side. We implemented keystroke sniffing functionality by tampering with function pointers. We performed a combination of dynamic and static binary analyses to reverse engineer a small part of kernel code related to keystroke processing. We sent some keystrokes into the emulated system and collected an execution trace for the guest kernel. Through dynamic taint analysis, we tracked how keystrokes propagate in the kernel space. In consequence, we identified several code regions that are relevant to keystroke processing. Then we statically examined these code regions using IDA Pro. It took one of the authors only a few hours to identify two function pointers (one in static memory region allocated in the keyboard driver `i8042prt.sys`, and the other in a dynamic memory region) that can be individually exploited to intercept keystrokes. To confirm that these two function pointers can be exploited indeed, we implemented two keyloggers, named `keylogger-1` and `keylogger-2`, to exploit these two function pointer respectively. We are not aware that such attacks have appeared in the literature and existing malware attacks. As shown in Sect.5.3, these two keyloggers evade the existing detection tools except HookScout. This experiment demonstrates that it is absolutely feasible for attackers to implement illicit functionalities by using this stealthy attack technique.

5.2 Policy Generation

Now we evaluate the analysis subsystem of HookScout. In particular, we are interested in how context sensitivity affects the coverage of the generated policy.

Table 1. The coverage and size of policy influenced by the level of context sensitivity

Level	Coverage		Templates	
	AVG	STDEV	Raw	Final
1	94.67%	2.97%	3518	308
2	96.10%	1.92%	4285	405
3	96.74%	1.64%	5270	511

The coverage is measured as a ratio of the number of function pointers identified by the policy to the total number of function pointers. In addition, we want to see how context sensitivity affects the size of the generated policy. To measure the coverage, we used the snapshots from the first two runs to generate policy, and then applied the generated policy to the snapshots from the third run.

We listed the experimental results in Table 1. We measured the coverage for each snapshot in the third run. In Table 1, we summarized these results by calculating the average and standard deviation of the coverage. For the size of the generated policy, we listed the number of templates in the raw policy and the number of templates in the final policy respectively. We make the following observations: (1) the generated policies can achieve very high coverage, even with 1 level of context sensitivity; (2) with an increase of context sensitivity, coverage is increased accordingly; and (3) the size of policy (i.e. the number of templates) is increased considerably with the increase of context sensitivity, but the absolute number is still fairly small. Considering that 3-level context sensitivity can achieve the highest coverage and reasonably small policy size, we chose to generate a policy with 3-level context sensitivity.

It took approximately 70 seconds to process one snapshot, and around 4 hours in total to generate a policy from 200 snapshots. Due to the fact that we only need to generate one policy for each version of OS kernel and can distribute it to all machines with the same OS kernel installed, we believe that this execution time is acceptable. Moreover, the task of policy generation can be easily partitioned and parallelized, which would increase the performance significantly.

5.3 Hook Detection

We evaluated three aspects of the detection subsystem of HookScout. First, we compiled a set of kernel rootkit samples to evaluate the effectiveness of the detection subsystem. Second, we measured its performance overhead. Third, we evaluated the occurrence of false alarms.

Detecting Kernel Hooks. We obtained a set of kernel rootkits from public resources [21, 17] and collaborative researchers. We selected the rootkit samples that are known to install kernel hooks and are able to run in our test environment. We also included the two synthetic keyloggers in the experiment to evaluate how effective the existing detection tools and HookScout are in terms of detecting new attacks. As a comparison with HookScout, we chose the following hook detection tools: IceSword [12], VICE [3], and RAIDE [19]. System

Table 2. Detection Results of Four Tools. I stands for IceSword [12]), V for VICE [3], R for RAIDE [19], and H for HookScout.

Sample Name	Hooking Region	I	V	R	H
HideProcessHookMDL [21]	SSDT	✓	✓	✓	✓
Sony Rootkit [26]	SSDT	✓	✓	✓	✓
Storm Worm [27]	SSDT	✓	✓	✓	✓
Shadow Walker [21]	IDT	?	✓	✓	✓
basic_interrupt_3 [21]	IDT	?	✓	✓	✓
TCPIRPHOOK [21]	Tcp driver object	×	✓	✓	✓
Rustock.C [22]	Fastfat driver object	×	×	✓	✓
Uay Backdoor [29]	NDIS data block	×	×	✓	✓
Keylogger-1	Kbd static data region	×	×	×	✓
Keylogger-2	Kbd dynamic data region	×	×	×	✓

Virginity Verifier [23] did not function correctly in our testing environment, so we did not include this tool in the experiment.

We listed the detection results in Table 2. We can see that all detection tools, including HookScout, are able to detect SSDT hooks, and all except IceSword are able to detect IDT hooks. IceSword displays only the content of IDT and requires manual inspection to determine if there is a hook, so we leave a "?" mark for IceSword. TCPIRPHOOK [21] and Rustock.C [22] hook function pointers in `Tcp` and `Fastfat` device driver objects respectively. IceSword does not inspect kernel objects, and thus cannot detect these hooks. While RAIDE checks both `Tcp` and `Fastfat`, VICE only checks `Fastfat` object. Uay Backdoor [29] modifies function pointers in the NDIS data structure maintained for the TCP/IP network protocol. IceSword and VICE cannot detect these hooks installed by Uay Backdoor. However, RAIDE has another special policy for checking the registered network protocol list, and thus can detect these hooks successfully. By exploiting new function pointers, our two synthetic keyloggers, keylogger-1 and keylogger-2, can evade all the detection tools in our experiment, except HookScout.

As compared to the other three detection tools, HookScout is able to detect all the samples in this set. The key difference between HookScout and the other tools is that HookScout is equipped with much more thorough detection policy, which is automatically generated by the analysis subsystem, whereas the other tools have very limited policies that are manually defined. Given the high coverage of our automatically generated policy, HookScout is substantially more difficult to evade.

It is worth noting that TCPIRPHOOK, and Rustock.C tamper with function pointers in kernel objects organized in the polymorphic hash table [24]. Even with access to the source code of Windows kernel, context-insensitive analysis approaches (such as SBCFI [16] and Gibraltar [1]) would not identify these function pointers. By contrast, with context-sensitive policy inference and hook detection, HookScout can automatically generate policy and validate these function pointers successfully. Moreover, Keylogger-1 exploits a function pointer in

Table 3. Performance Overhead of the Detection Engine

Workload	w/o HookScout	w/ HookScout 1s	w/ HookScout 5s	Slowdown 1s	Slowdown 5s
Boot OS	19.43 s	20.70s	20.43 s	6.5%	5.1%
Copy directories	7.57 s	8.09s	7.68 s	6.9%	1.5%
(De)compress files	23.84 s	24.44s	23.51 s	2.5%	-1.4%
Download a file	23.59 s	24.49s	24.42 s	3.8%	3.5%

the keyboard device driver. Without source code of this driver, source code analysis approaches [16,1,31,4] will be completely unaware of function pointers defined in this driver.

Performance Overhead. To observe how HookScout affects performance, we performed several workloads and measured their execution times with and without the detection engine installed. We also measured the performance with two different checking intervals: 1 and 5 seconds. The workloads include booting Windows, copying a directory structure, performing compression and decompression of a directory structure with `7zip`, and downloading a file with `wget`. The total size of the directory structure is 75MB. The size of the downloaded file is 100MB. Table 3 shows the execution time for each workload. Each workload is performed 7 times and the average of 5 non-minimum/maximum runs is reported. In all, the slowdown caused by HookScout is about 4.9% and 2.1% for the checking intervals of 1 second and 5 seconds respectively.

False Alarms. To evaluate the occurrence of false alarms, we installed HookScout detection engine on a healthy system (without rootkits installed), and kept it running for eight hours. Meanwhile, a user operated on this machine for his regular computing tasks. We did not observe any false alarms during this period.

6 Discussion

In this section, we discuss the feasibility of potential evasion techniques against HookScout and our countermeasures.

Exploit Uncovered Function Pointers. Attackers can perform the same analysis on function pointers, and determine which function pointers would not be located and verified by HookScout. Then they can target these function pointers to evade HookScout. First of all, our policy generation technique can achieve over 95% coverage, so we have substantially reduced the space of this attack (e.g., from 18,000 to 900). In addition, it may be impractical for attackers to take advantage of many of these uncovered function pointers, because of their mutable nature: sometimes they are function pointers and sometimes not. Furthermore, for this small number of uncovered function pointers, defenders can investigate these cases first and manually define special policies for the plausible attacks.

Exploit Uncommon Proprietary Device Drivers. QEMU has emulated a set of common hardware devices. For these common devices, HookScout can generate policy to validate functions pointers in the corresponding device drivers. For those devices that are not supported by QEMU, HookScout cannot generate policy to protect their drivers, since these drivers are not installed and activated during the analysis phase. This limitation is not specific to HookScout. No previous solutions are able to address this issue, which requires further investigation.

Attack Limited Test Cases. Our hook detection policy is derived mainly using dynamic analysis. In principle, the quality of dynamic analysis is largely determined by the completeness of test cases. Attackers could potentially exploit those function pointers that were not initialized and moved around during our analysis but will be activated in other test cases. Then HookScout would not detect these attacks. By exercising the analysis subsystem with a more complete set of test cases that exercises more kernel functionalities, this problem can be alleviated. Moreover, if attackers only target the function pointers that are only initialized and moved around in uncommon situations, it means the attacks will not become effective most of time.

Subvert or mislead HookScout. The detection engine of HookScout is implemented as a kernel module installed in user's system, and thus is subject to complete subversion just like any other security tools running in the same privilege as malware. In addition, malware may mislead HookScout by injecting fake events or changing the existing events that HookScout monitors. In particular, HookScout relies on proper functionality of kernel memory management routines and correct call stack information to monitor memory objects. More secure implementation based on virtual machine techniques can significantly raise the defense bar against this kind of attacks. For example, Payne et al. systematically discussed the challenges of secure active monitoring, proposed a series of solutions and built a framework called Lares [18].

7 Related Work

Postmortem Analysis. Several systems have been proposed to facilitate understanding of rootkit's behaviors. HookFinder [32] can automatically identify and understand how a rootkit installs hooks. K-Tracer [14] and PoKeR [20] are profiling tools for monitoring rootkit behaviors in general, including hooking behaviors, data structure manipulation and others. The better understanding of a new kernel attack can then be used to harden the security policy against similar attacks. As our study shows, the attack space for kernel function pointer hooking is vast. After a function pointer is known to be exploited, attackers may easily switch to exploit another function pointer.

Proactive Defense. The first line of defense against kernel attacks is to prevent untrusted code execution in the kernel space. Several systems leveraged the virtual machine based architecture to monitor and enforce the kernel code integrity.

Livewire [8] was the first proposal to make use of virtual machine monitor to monitor system integrity, including verifying the kernel code regions and examining specific data attacks by querying the system states. SecVisor [25], is a tiny hypervisor (i.e. virtual machine monitor) that ensures code integrity for commodity OS kernels. Patagonix [15], is another system based on hypervisor to identify covertly executed binaries. This line of defense can be circumvented by return-oriented rootkits [11], which take advantages of existing kernel code to build illicit functionalities.

The second line of defense is to enforce control flow integrity. SBCFI [16], Gibraltar [1], SFPD [4], HookMap [30], HookSafe [31], and HookScout belong to this category. These system may still catch return-oriented rootkits, as long as persistent control flow modifications are made. SBCFI [16], Gibraltar [1], and SFPD [4] perform source code analysis on the OS kernel to derive the security policy. The requirement of source code would impede their deployment on closed-source operating systems and proprietary device drivers. SBCFI and Gibraltar need manual annotations for generic pointers and perform context-insensitive analysis. Therefore, these two systems cannot deal with type polymorphism. SFPD addressed these two limitations by performing more comprehensive inter-procedural context-sensitive points-to analysis. In comparison, HookScout performs context-sensitive dynamic binary analysis. In consequence, it is able to eliminate the requirement for source code and handle type polymorphism. HookMap [30] analyzes the kernel-side execution of certain security applications to help identify potential hook sites. Compared to HookMap, HookScout performs more complete analysis by monitoring the full kernel execution and thoroughly tracking function pointers. Moreover, HookScout conducts more advanced context-sensitive type inference, so it can deal with function pointers in complex data structures and achieve significantly higher coverage than HookMap.

8 Conclusion

In this paper we targeted a class of advanced kernel attacks: function pointer hooking. We assessed the severity of this new threat. First, we conducted a quantitative measurement study to show the attack surface is vast. Second, we implemented two new keyloggers using this attack technique, showing that this threat is realistic even on closed-source operating systems like Windows. To effectively combat this threat, we presented HookScout, a proactive, context-sensitive hook detection scheme capable of detecting this stealthiest persistent control flow modifications within the Windows kernel without the need for source code. We demonstrated HookScout's ability to generate a context-sensitive policy for detecting persistent control modifications that can be used on any machine with the same version of the OS kernel installed. We evaluated our system against real world stealthy rootkits and malware and showed that we were able to detect all of them (including our synthesized keyloggers). Additionally, we showed that our approach is easily deployable, has a low overheard and most importantly, our approach is generic and capable of detecting kernel-wide function pointer changes.

References

1. Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structure invariants. In: Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008), Anaheim, California, USA (December 2008)
2. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track (April 2005)
3. Butler, J., Hoglund, G.: VICE–catch the hookers!. In: Black Hat USA (July 2004), http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf
4. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2009) (November 2009)
5. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th USENIX Security Symposium (Security 2004) (August 2004)
6. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2005) (November 2005)
7. Egele, M., Kruegel, C., Kirda, E., Yin, H., Song, D.: Dynamic Spyware Analysis. In: Proceedings of the 2007 Usenix Annual Conference (Usenix 2007) (June 2007)
8. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of Network and Distributed Systems Security Symposium (NDSS 2003) (February 2003)
9. Hoglund, G.: Kernel object hooking rootkits (KOH rootkits), http://www.rootkit.com/newsthread.php?newsid=501
10. Hultquist, S.: Rootkits: The next big enterprise threat, http://www.infoworld.com/article/07/04/30/18FErootkit_1.html
11. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th USENIX Security Symposium (July 2009)
12. IceSword, http://www.antirootkit.com/software/IceSword.htm
13. The IDA Pro Disassembler and Debugger, http://www.datarescue.com/idabase/
14. Lanzi, A., Sharif, M., Lee, W.: K-Tracer: A system for extracting kernel malware behavior. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009) (February 2009)
15. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor Support for Identifying Covertly Executing Binaries. In: Proc. 17th Usenix Security Symposium, San Jose, CA (July 2008)
16. Nick, J., Petroni, L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007) (October 2007)
17. Offensive computing, http://www.offensivecomputing.net/
18. Payne, B.D., Carbone, M., Sharif, M.I., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, Oakland 2008 (2008)

19. RAIDE, http://www.rootkit.com/vault/petersilberman/RAIDE_BETA_1.zip
20. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: EuroSys 2009 (April 2009)
21. rootkit.com, http://www.rootkit.com/
22. Rustock, C.: http://www.rootkit.com/newsread.php?newsid=879
23. Rutkowska, J.: System virginity verifier: Defining the roadmap for malware detection on windows systems. In: Hack In The Box Security Conference (September 2005), http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt
24. Schreiber, S.B.: Undocumented Windows 2000 Secrets. In: Windows 2000 Object Management, ch. 7 (2007)
25. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007 (2007)
26. Sony's DRM Rootkit: The Real Story, http://www.schneier.com/blog/archives/2005/11/sonys_drm_rootk.html
27. Storm Worm, http://news.zdnet.co.uk/security/0,1000000189,39285565,00.htm
28. TEMU: The BitBlaze dynamic analysis component, http://bitblaze.cs.berkeley.edu/temu.html
29. UAY kernel-mode backdoor, http://www.xfocus.net/tools/200602/uay_source.rar
30. Wang, Z., Jiang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)
31. Wang, Z., Jiang, X., Cui, W., Ning, P.: Mapping kernel objects to enable systematic integrity checking. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2009) (November 2009)
32. Yin, H., Liang, Z., Song, D.: HookFinder: Identifying and understanding malware hooking behaviors. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 2008) (February 2008)
33. Yin, H., Song, D.: Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley (January 2010)
34. Yin, H., Song, D., Manuel, E., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS 2007) (October 2007)

Conqueror: Tamper-Proof Code Execution on Legacy Systems

Lorenzo Martignoni[1], Roberto Paleari[2], and Danilo Bruschi[2]

[1] Università degli Studi di Udine
[2] Università degli Studi di Milano
lorenzo.martignoni@uniud.it, {roberto,bruschi}@security.dico.unimi.it

Abstract. We present Conqueror, a software-based attestation scheme for tamper-proof code execution on untrusted legacy systems. Beside providing load-time attestation of a piece of code, Conqueror also ensures run-time integrity. Conqueror constitutes a valid alternative to trusted computing platforms, for systems lacking specialized hardware for attestation. We implemented a prototype, specific for the Intel x86 architecture, and evaluated the proposed scheme. Our evaluation showed that, compared to competitors, Conqueror is resistant to static and dynamic attacks and that our scheme represents an important building block for realizing new security systems.

1 Introduction

Code attestation is the process of verifying the integrity of a piece of code executing in an untrusted system. Besides integrity verification, code attestation can also be used to execute an arbitrary piece of code in an untrusted system with the guarantee that the code is run unmodified and in an untampered execution environment. In the last years, hardware extensions, such as TPM chips [1], have been proposed for securing computations, including performing attestation. However, these extensions are not yet available on every computing device. In such a situation, pure software-based solutions are the only viable alternative.

Several software-based attestation schemes have been proposed in literature [2,3,4,5,6,7]. All these schemes are based on a challenge-response protocol involving two parties: an *untrusted system* and a *verifier*. The verifier issues a challenge for the untrusted system, where the challenge consists in computing the checksum of certain memory locations and properties of the execution environment. The checksum is computed by executing a particular *attestation routine*, or *checksum function*. Once computed, the checksum is sent back to the verifier. The verifier relies on the time to determine whether the checksum is genuine or if it could have been forged. Indeed, attestation routines are constructed such that any tampering attempt results in a noticeable increase of the execution time. Thus, a checksum received too late is a symptom of an attack.

The complexity of the attestation routine depends on the hardware characteristics of the untrusted system on which it has to be executed. Indeed, the output of the routine is guaranteed to be genuine only if it is executed in a properly configured execution environment. In complex hardware architectures, such as the ones used in personal computers, there exist several configurations of the execution environment that can be exploited by an attacker to thwart attestation. Therefore, the attestation routine must ensure, and prove to the verifier, that the execution environment in which it executes satisfies all the requirements to impede attacks. In other words, the attestation routine must attest its own code, but also the execution environment. Intuitively, the requirements for tamper-proof attestation are that the attestation routine must be executed at the highest level of privilege (i.e., at the same level of the most powerful attacker) and that its execution must be uninterruptible. Practically speaking, in a legacy system with no hardware support for virtualization, that means that the routine must execute in system mode (i.e., the privilege level of the operating system) and that all interrupts must be disabled, to prevent the attacker to regain the control of the execution at some point. Unfortunately, even if the requirements are very well defined, guaranteeing that they are satisfied in a complex execution environment where attacker and defender have the same privileges is a very challenging problem.

In this paper we present Conqueror, a software-based scheme for tamper-proof code execution on untrusted legacy systems. Conqueror provides a security primitive that allows to build applications that require the availability of a trusted computing base. Pragmatically speaking, Conqueror guarantees that an arbitrary piece of code can be executed untampered in an untrusted system, even in the presence of malicious software. Conqueror has been developed to address the limitations of Pioneer, the state-of-the-art software-based attestation solution [6]: Conqueror is immune to all attacks that are known to defeat Pioneer, and it can also be used on untrusted systems where the attacker could leverage hardware virtualization extensions to hold control of the execution environment in which the attestation routine executes. Conqueror adopts a variation of the challenge-response protocol used in traditional attestation schemes: the challenge does not consist in a seed to initialize a constant attestation routine, but instead consists in an entire routine, that is different each time, self-decrypting, and obfuscated. The intent is to make it impossible for an attacker to reverse engineer the logic of the checksum computation, and to facilitate the hiding of the sensitive operations that Conqueror needs to perform to attest that the state of the environment executing the code impedes any attack. The strength of this approach is that we are drastically increasing the time needed by an attacker to forge a checksum.

We experimentally demonstrate our claims about Conqueror's resistance to attacks. We show that even a preliminary low-level analysis of the code of Conqueror's one-time attestation routine (i.e., disassembly), which is necessary to perform any subsequent meaningful analysis for reconstructing the semantics, costs about the same time required to execute the routine. Moreover, we show that Conqueror is also resilient to dynamic attacks performed by an attacker leveraging a hardware-assisted hypervisor. Finally, to demonstrate Conqueror's potential, we present a proof-of-concept software-based primitive to launch securely a hypervisor in a running untrusted system, to segregate the system into a restricted guest. This primitive could be used in place of `skinit` [8] and `senter` [1] on untrusted systems with no hardware support for trusted computing.

2 State-of-the-Art of Attestation on Legacy Systems

This section presents Pioneer, the major Conqueror's competitor. Both systems target the same hardware architecture, but they use very different approaches. Moreover, Conqueror is resistant to attacks that are known to defeat Pioneer.

Pioneer is a software-based attestation scheme that can be used to establish a trusted computing base, called *dynamic root of trust*, on an untrusted legacy system. Pioneer is specific for Intel x86 with EM64T extensions. The code of the dynamic root of trust is guaranteed to be unmodified and to execute in a *tamper-proof execution environment*. The dynamic root of trust measures the integrity of an arbitrary executable, and then runs the executable in the trusted execution environment. The dynamic root of trust is established using a *verification function*. The verification function is an extension of a conventional checksum function and additionally includes a hash function to verify the integrity of an executable. The verification function is self-checking (i.e., it attests its own code), and it attests the execution environment.

The Pioneer verification function is composed by three components: (I) a checksum function, (II) a send function, and (III) a hash function. The checksum function is used to compute a checksum over the entire verification function and to setup the execution environment in which the other functions are guaranteed to run untampered. Since the sensitive component of Pioneer is the checksum function, we do not overview the others.

As in the majority of code-attestation schemes, in Pioneer the checksum function is known a priori and the challenge issued by the verifier consists in a seed that initializes this function. Therefore, an attacker has complete access to the checksum function and can analyze it offline to find weaknesses. The checksum function has been constructed manually to be time-optimal: no adversary function that can compute the correct checksum without introducing a noticeable overhead exists. Time-optimality is achieved using operations that prevent parallelization, that have a low variance execution time, and by executing these operations iteratively, to maximize the overhead of the attacker. Most importantly, the checksum function is responsible for initializing the execution environment and for attesting the correct initialization.

Unfortunately, since the hardware architecture for which Pioneer was developed is full of subtle details, researchers have found ways to thwart the setup of the dynamic root of trust without being noticed by the verifier. For example, it is possible to perform the entire checksum computation in user-space and to regain the control of the execution through exceptions without corrupting the checksum. Another attack consists in desynchronizing data and code pointers and to execute a modified checksum function that computes the checksum of a pristine function residing elsewhere in memory [9]. Finally, Pioneer's assumptions that the most powerful attacker operates in system mode does not hold on new commodity hardware with support for virtualization [8,10].

3 Conqueror Overview

In this section we give an overview of Conqueror, our scheme for software-based code attestation and tamper-proof code execution on untrusted legacy systems (Intel x86). Conqueror does not suffer the problems that affect the state-of-the-art attestation scheme for this class of systems.

3.1 Threat Model

Conqueror has been developed to operate in the following adversary scenario. We assume that the untrusted system has been compromised, and that the attacker operates at the highest privilege level: system mode (ring 0) if the system has no support for hardware-based virtualization, hypervisor mode if the support is available. However, we assume the adversary cannot operate in system management mode, that he cannot perform hardware-based attacks (e.g., DMA-based attacks or overclocking), and that he cannot leverage a pristine or a more powerful system to break the attestation scheme. The final assumption is that the untrusted system supports a single thread of execution (e.g., no SMP).

3.2 Conqueror Architecture and Protocol

As any other software-based code attestation scheme, Conqueror is based on a challenge-response protocol, where a verifier challenges the untrusted system. The central component of Conqueror is the Tamper-Proof Environment Bootstrapper (TPEB). As the name says, the TPEB is responsible for setting up the environment in the untrusted system for the tamper-proof execution of an arbitrary executable. Figure 1 shows the layout and the protocol of Conqueror (the numbers in the figure represent the temporal ordering of the events). The TPEB is composed by a checksum function and a send function. The checksum function computes the checksum to attest the integrity of the TPEB itself and the integrity of the executable. The send function transmits the computed checksum value to the verifier and invokes the executable. The send function is logically separated from the checksum function because it is hardware dependent (i.e., it depends on the network card installed on the untrusted system).

In Conqueror the verifier generates the checksum function on demand, such that each function differs considerably from the others. Differences are both syntactic and semantic. Moreover, functions are obfuscated using multiple obfuscation schemes. The attacker has no access to the checksum function ahead of time and cannot perform any offline analysis nor optimization [7]. In Conqueror, the newly generated checksum function is initially sent encrypted to the untrusted system. Later on, at time t_0, the verifier transmits the key for decryption. Since the verifier knows precisely in which execution environment the function must be executed and knows the hardware characteristics of the untrusted system, it can compute the expected checksum value and can estimate the amount of time that will be required by the untrusted system to decrypt, to execute the function, and to send back the result. Let $t_1 = t_0 + \Delta_t$ be the time by which the correct checksum has to be received by the verifier to be considered authentic; Δ_t is an upper bound,

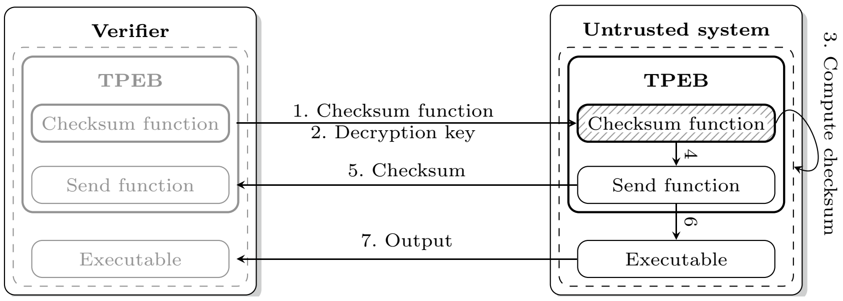


Fig. 1. Overview of Conqueror

empirically estimated, of the maximum time requested by the untrusted system to compute the checksum in the absence of an attack. If the verifier does not receive the correct checksum by t_1, then the checksum is considered forged and the execution environment not tamper-proof. In a traditional checksum function (e.g., that used in Pioneer), where the function is known a priori and can be analyzed offline, the attacker has Δ_t time to execute a malicious function to forge the checksum. In Conqueror, the attacker has Δ_t to (i) analyze the checksum function, (ii) generate a new function capable of forging the checksum, and (iii) execute the generated function. Alternatively, the attacker would have to emulate the entire execution of the checksum function. Differently from traditional checksum functions, the ones in Conqueror are generated automatically; for this reason we cannot guarantee a low collision rate nor that their implementation is optimal (in terms of execution time and in code size). Nevertheless, given the small time frame available, there is no opportunity for the attacker to reverse engineer their semantics, nor to emulate the execution, and to forge checksums in time.

Since Conqueror targets a very complex hardware architecture, particular attention has to be devoted to prevent checksum forgery, by tampering either the checksum function or the execution environment. To attest the trustworthiness of the environment, the verifier embeds in the checksum function several operations whose behavior and execution time depend on the configuration of the environment (e.g., instructions that raise exceptions when executed without enough privileges).

An attacker who tampers the execution of the checksum function will corrupt the checksum, or will incur in a time overhead that will cause the overall checksum computation to exceed the expected time Δ_t. For these reasons, Conqueror guarantees that a correct checksum, received by the verifier by t_1, is the proof that the checksum function has been executed unmodified and that the bootstrap of the tamper-proof execution environment succeeded.

4 Conqueror Implementation

Conqueror current implementation is specific for the Intel x86 architecture and so are the details of the implementation presented in this section. However, we believe the same scheme can be used, as is, on the Intel x86-64 architecture.

4.1 Tamper-Proof Environment Bootstrapper

The layout in memory of the TPEB is shown in Figure 2. The TPEB consists of the checksum function, its data, and the send function. For simplicity, the TPEB is located at a fixed address (`BASE`) and in consecutive memory pages. Moreover, the executable follows immediately the TPEB, and the overall buffer is padded to a multiple of page size (`SIZE`). We assume that the TPEB is already initialized on the untrusted system, with the exception of the checksum function. The checksum function and its data reside in a dedicated memory page (starting from `BASE`) and all unused bytes in this page are initialized randomly, to hide code and data. This page is generated on-demand by the verifier and transmitted encrypted to the untrusted system. The latter stores in memory, at the `BASE` address, the page and waits for the decryption key. Attestation begins when the verifier sends out the key. The reason for encryption is to exclude from the measurement the time required to transmit and prepare the TPEB.

4.2 Checksum Function

The checksum function is composed by a prologue, a checksum loop, and an epilogue (Figure 2). The prologue decrypts the rest of the page containing the checksum function, initializes the execution environment for the remaining of the computation, and invokes the checksum loop. The checksum loop (described in Section 4.2) computes the checksum of the memory pages containing its own code, the send function, and the executable, (i.e., from `BASE` to `BASE + SIZE`), and invokes the epilogue. The epilogue invokes the send function, which in turn invokes the executable.

The checksum function computes the checksum by combining multiple checksum gadgets. In the current implementation the checksum size is 128 bits. A *gadget* (c_i) is a small code snippet that receives in input the address of a memory location and updates the running value of the checksum, according to the content of the memory. We refer to these gadgets as *active*, since they are intentionally executed by the checksum function. The purpose of an active gadget is twofold. First, each gadget contributes to the computation of the checksum in a different way. Thus, the correct checksum can be computed only if all the gadgets are executed in the proper order and with the proper arguments. Second, certain gadgets perform additional operations to verify the trustworthiness of the execution environment and, in case the environment has been tampered, they either corrupt the checksum or introduce a time overhead. Since gadgets are scattered around the memory, differ syntactically and semantically from one checksum function to another, and are obfuscated, it becomes very difficult for the attacker to reconstruct the exact logic of the checksum function.

In addition to active gadgets, the checksum function relays on *passive gadgets* (h_j), or *handlers*, that are not invoked directly by the checksum function, but rather as the result of an unexpected event that can occur only in a tampered execution environment. If executed, passive gadgets corrupt the checksum. Passive gadgets are registered during the prologue, by replacing the Interrupt Descriptor Table (IDT)

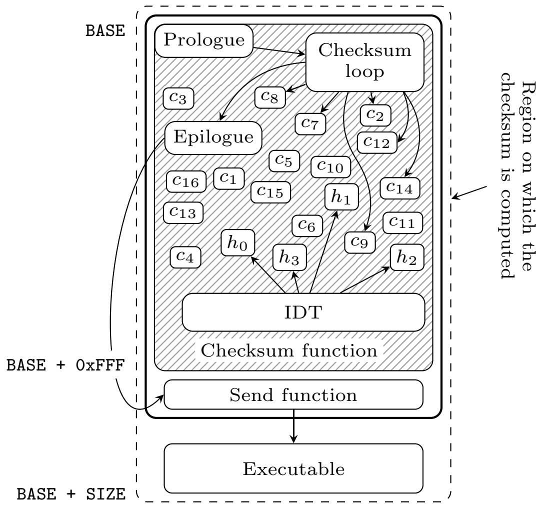

BASE
BASE + 0xFFF
BASE + SIZE
Prologue
Checksum loop
Epilogue
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16}
h_0 h_1 h_2 h_3
IDT
Checksum function
Send function
Executable
Region on which the checksum is computed

Fig. 2. Overview of the TPEB

with a new one embedded within the TPEB, and cannot be disabled by the attacker: an improper configuration of these gadgets will result in a wrong checksum.

Prologue. The prologue (Figure 3) is a small routine that decrypts the rest of the page and initializes the trusted execution environment. More precisely, the prologue disables all maskable interrupts (line 2), decrypts the rest of the page (line 4 and 5), and installs custom interrupts handlers (line 7). Custom handlers are installed by updating the address of the interrupt descriptor table (IDT). The new address is set to a location, within the memory page containing the checksum function, that holds a pre-initialized IDT (Figure 2). The mapping between interrupts and handlers (the content of the IDT) is chosen by the verifier and not known to the attacker. The handlers (h_i in Figure 2), or passive gadgets, are a special type of gadget: like normal gadgets they modify the running value of the checksum, but they terminate their execution with a special instruction to return to normal execution (i.e., `iret`). Furthermore, handlers are never invoked explicitly by the checksum loop but only in response to interrupts or exceptions.

The purpose of the prologue is twofold. First, by disabling maskable interrupts (pin-based interrupts generated by the peripherals) we inhibit the asynchronous execution of all handlers. Second, by installing custom interrupt handlers that update the checksum value, we can tell whether any interrupt or exception occurred during the computation of the checksum. If maskable interrupts are successfully disabled, no asynchronous interrupt occurs, and the checksum is not corrupted because no interrupt handler is fired. Similarly, if the checksum loop executes privileged instructions, and the checksum function is executed in system mode, no exception occurs and no exception handler corrupts the checksum. On the other hand, any attempt to execute the checksum function in user mode

```
// Disable maskable interrupts
asm("cli");
// Decrypt the remaining of the page
for (i = PROLOGUE_SIZE; i < 4096; i++)
  BASE[i] ^= KEY[i % KEY_SIZE]
// Install custom interrupt handlers
asm("lidt %0" : : "m" (IDT));
```

Fig. 3. Overview of the prologue

```
1 for (i = 0, j = 0; i < ITERATIONS; i++) {
2   x = seed(i) % (SIZE / 4);
3   do {
4     x = (x + (x*x | 5)) % (SIZE / 4);
5     checksum_gadget[j++ % GADGETS](BASE + x*4);
6   } while (x != seed(i) % (SIZE / 4));
7 }
```

Fig. 4. Overview of the checksum loop (in C for clarity)

results in an exception, in the execution of the corresponding handler, and in a corruption of the checksum value.

By positioning the IDT in the same memory page of the checksum function, we implicitly certify the content of the table. The only opportunity for the attacker is to intercept and simulate a successful update of the IDT. For example, the attacker could emulate the execution of the prologue or execute the prologue in user-space, such that the update of the IDT will raise an exception and will be intercepted. Then, the attacker could install his own malicious IDT and simulate a successful disabling of maskable interrupts. We prevent this attack by including in the checksum loop a special gadget that queries the address of the IDT and updates the running value of the checksum accordingly. Therefore, attacker's attempts to relocate the IDT will result in a corrupted checksum. Further details about the aforementioned gadget and about why its execution cannot be detected by the attacker are given in Section 4.2.

In conclusion, a correct value of the checksum, received by the verifiers within the expected time, certifies that the prologue is executed successfully, that the checksum function is executed at the maximum privilege level, and that the attacker cannot interrupt the execution using interrupts or exceptions.

Checksum Loop. The core of the checksum computation is the checksum loop shown in Figure 4. The checksum loop is composed by two nested loops. The innermost loop traverses the memory and updates the checksum according to the content of the memory, invoking a different gadget at each iteration. The memory is not traversed linearly, but instead in a pseudorandom fashion (line 4), using a T-function [11]. The T-function produces a pseudorandom permutation of all the memory locations to traverse. More precisely, the T-function returns the memory offset of the next memory location for the checksum computation. At each iteration (line 5), from the offset returned by the T-function, the checksum loop computes the absolute address of the memory location to process, and invokes a specific gadget to update the running value of the checksum (`GADGETS` represents the number of gadgets available). Clearly, without an analysis of the code, the attacker cannot predict which gadgets will process which memory locations and, even if the checksum function were weak (e.g., it suffers a high collision rate), the attacker would not have enough time to exploit the weakness. Finally, it should be noted that the execution of the checksum loop is deterministic, unless it is tampered.

The outermost loop repeats the memory traversal multiple times (`ITERATIONS` denote the number of iterations of the outermost loop). At each iteration, the T-function used in the innermost loop is initialized with a different seed (line 2). Therefore, the innermost loop is executed multiple times and at each execution the running value of the checksum is updated using a different combination of memory locations and gadgets, and the order in which the checksum is updated is also different. Since the checksum function is constructed such that any attacker's attempt to forge the correct checksum will introduce an overhead in the computation of the checksum, the outermost loop causes a constant time overhead per iteration and facilitates the detection of the attack. Details about how we select the optimal number of iterations for the outermost loop are given in Section 5.

The seeds used by the T-function to generate the addresses are also included in the memory page containing the checksum function. To avoid wasting precious bytes of the page, the vector containing the seeds is positioned at a random location within the page and is not initialized, to overlap with the existing content of the page.

Checksum Gadgets. The checksum is computed by executing a sequence of gadgets, each of which contributes to update the running value of the checksum in a different way. Certain gadgets also perform additional operations to attest the trustworthiness of the execution environment. Given that gadgets are very small in size and that an entire memory page is dedicated to the checksum function, the checksum function can rely on about a hundred different gadgets simultaneously. Gadgets are generated on demand by the verifier and change (in number, position, syntax, and semantics) from challenge to challenge.

The following paragraphs describe in details the gadgets used in the checksum function to attest the integrity of the TPEB and of the code of the executable. Figure 5 shows some sample gadgets. For clarity, the gadgets presented are not optimized and use symbolic names (in uppercase) to refer to absolute memory locations containing data: `CHKSUM` and `ADDR` refer respectively to the memory locations storing the 128-bit checksum and the address of the next word to process.

Plain checksum computation. The simplest and most frequently used gadget is responsible only for updating the running value of the checksum. Different gadgets update the checksum in different ways, by applying different arithmetical or logical operations and by modifying different bits of the checksum value. Figure 5(a) shows a sample gadget. The gadget updates the checksum by adding the result of a bitwise XOR between the current memory location (`ADDR`) and a random key (`0xa23bd430`). Note that this gadget modifies the second word of the running 128-bit checksum (`CHKSUM+4`, at line 4).

IDT attestation. During the prologue, the interrupt descriptor table is replaced with a custom table, which is provided along with the checksum function. Since the prologue is executed at the beginning of the checksum function, it is reasonable to expect the attacker to try to emulate or intercept its execution.

```
1 mov   ADDR, %eax
2 mov   (%eax), %eax
3 xor   $0xa23bd430, %eax
4 add   %eax, CHKSUM+4
```

(a)

```
1 mov   ADDR, %eax
2 mov   (%eax), %eax
3 add   %eax, CHKSUM+8
4 sidt  IDTR
5 mov   IDTR+2, %eax
6 xor   $0x6127f1, %eax
7 add   %eax, CHKSUM+8
```

(b)

```
1 mov   ADDR, %eax
2 mov   (%eax), %eax
3 xor   $0x1231d22, %eax
4 mov   %eax, %dr3
5 mov   %dr3, %ebx
6 add   %ebx, CHKSUM
```

(c)

```
1    mov   ADDR, %eax
2    mov   (%eax), %eax
3    lea   l_smc, %ebx
4    roll  $0x2, 0x1(%ebx)
5  l_smc:
6    xor   $0xdeadbeef, %eax
7    add   %eax, CHKSUM+4
```

(d)

```
 1 mov   ADDR, %eax
 2 mov   (%eax), %ebx
 3 and   $0xfffff000, %eax
 4 add   $0x2b8, %eax
 5 movb  (%eax), %cl
 6 movb  $0xc3, (%eax)
 7 call  %eax
 8 movb  %cl, (%eax)
 9 xor   $0x7b2a63ef, %ebx
10 sub   %ebx, CHKSUM+8
```

(e)

```
1 mov   ADDR, %eax
2 mov   (%eax), %ebx
3 vmlaunch
4 xor   $0x7b2a63ef, %ebx
5 sub   %ebx, CHKSUM+8
```

(f)

Fig. 5. Sample gadgets for (a) plain checksum computation, (b) IDT attestation, (c) system mode attestation, (d,e) instruction and data pointers attestation, and (f) hypervisor detection

The content of the IDT is implicitly attested by the normal checksum computation, but the address of the IDT is not. To attest that the IDT shipped with the checksum function is actually being used, the checksum function relies on a specific gadget that queries the CPU to obtain the address of the IDT and updates the checksum accordingly. Obviously, the checksum will be wrong if a different IDT is being used. The only opportunity for the attacker to force the checksum function to behave as if the requested IDT were successfully installed is to intercept the query and to manipulate its output. To query the address of the IDT, the gadget uses the `sidt` instruction. Unfortunately for the attacker, this instruction is not privileged: it does not trigger an exception when executed in user mode [12]. Consequently, the only solution for the attacker to detect the instruction is to analyze the checksum function or to emulate its execution. However, any analysis or emulation attempt will introduce a noticeable overhead in the computation of the checksum. Figure 5(b) shows a sample gadget to attest the IDT. The only difference with a plain gadget (Figure 5(a)) is the addition of the instructions to query the address of the IDT (lines 4 and 5).

System mode attestation. After the update of the IDT, the attacker cannot regain the control of the execution, because all interrupts and exceptions will be served by the handlers installed by the checksum function. Although the previously described gadget forces the attacker to install our IDT, he could still attempt to execute the entire checksum function in user mode. If no maskable interrupt occurred during the execution of the checksum function, the checksum would not get corrupted, and the attack would not be detected. However, even if we suppose that the attacker executed the checksum function in user mode and that he were able to reprogram the interrupt controller to prevent any interrupt, he would lose any opportunity to regain the control of the system after checksum computation.

To have the guarantee that the TPEB is operating in system mode, the checksum function relies on a specific class of gadgets. These gadgets use a privileged instruction to update the running value of the checksum. If the function is executed in system mode, all the instructions of the gadgets will be executed successfully. However, if the function is executed in user mode, the privileged instruction will raise an exception (because of the lack of privileges), and the exception handler we installed to handle the exception will corrupt the checksum. In some cases, the handler could also trigger an endless loop. An example of such a gadget is shown in Figure 5(c). The gadget uses the CPU register `dr3` to store an intermediate result during the computation of the new checksum value. This register can be accessed only in system mode and any access originating from user mode causes a general protection fault exception.

Instruction and data pointers attestation. The checksum function is a self-checksumming function. A common class of attacks against self-checksumming functions are *memory copy attacks*, that allow attackers to forge checksums [6]. Briefly, in a memory copy attack, the attacker modifies the instructions of the checksum function, or the execution environment, to redirect all memory reads to memory locations containing a pristine copy of the data to attest. A memory copy attack can be performed in different ways: (i) by patching the instructions of the checksum function to read from different locations, (ii) by configuring segmentation to separate the code from the data segment, and (iii) by desynchronizing the data and the instruction TLBs [9].

To prevent memory copy attacks, the checksum function uses a specific type of gadget that guarantees that reads, writes, and fetches involving the same virtual memory location refer to the same physical location. Indeed, data and instruction physical pointers equivalence is sufficient to guarantee that no memory copy attacks of type (ii) and (iii) can be performed. We intentionally do not consider the case of memory copy attacks of type (i), performed by patching or by emulating the checksum function, because of the noticeable time overhead the attacker would suffer. To validate the equivalence of data and instruction pointers we leverage a gadget based on self-modifying code [13]. The gadget updates the running value of the checksum by performing an operation that is generated dynamically by modifying the code of the checksum function in place. If no memory copy attack is being performed, the data pointer (used for both reads and writes) and the instruction pointer point to the same physical page. Thus, the memory write executed by the gadget to update its instruction modifies the physical page that is also being executed. If the attacker were performing a memory copy attack, the data and the instruction pointer would point to two different physical pages and the instruction executed to update the checksum would differ from the ones just created by the gadget. Consequently, the out-of-date instruction would corrupt the checksum.

Figure 5(d) shows a sample gadget used by Conqueror to prevent memory copy attacks. The gadget updates the checksum by adding the data read from the memory (lines 1, 2, and 7). Before the addition, the word read is XORed with an immediate (line 6). The immediate is rotated (by two bits) at each execution of the gadget, by modifying the operand of the instruction in place (line 3 and 4). In the case of a memory copy attack the checksum would not be updated correctly because the operand of the `xor` instruction would remain unmodified.

Note that, in the case of a memory copy attack of type (III), the attacker can operate on each page separately. The aforementioned gadget successfully protects against the desynchronization of data and instruction pointers that point to the page containing the checksum function, but, as is, it is ineffective at protecting other pages (containing the send function and the executable). Indeed, only instructions residing in the page containing the checksum function are executed during the checksum computation. To address this problem, we use a variation of the original gadget, that places a temporary small snippet of code (e.g., a `ret` instruction) in a random position of the input page, executes the snippet, and restores the original content of the modified locations. Figure 5(e) shows an example of this type of gadget. The gadget selects a random location in the page being attested (lines 1 to 4), saves the content of the location (line 5), replaces the content with a `ret` instruction (line 6), executes the newly generated instruction (line 7), restores the original content of the modified location (line 8), and finally updates the checksum (line 9 and 10).

Hypervisor detection. An attacker operating in hypervisor mode, on a system with hardware support for virtualization, has complete control of the operating system: he can intercept the execution of all sensitive instructions, interrupts, exceptions, and, most importantly, the hypervisor and the attacker are completely transparent to guests. Dai Zovi and Rutkowska *et al.* have clearly demonstrated what an attacker can do on systems with hardware support for virtualization [14,15]. The gadgets presented so far are effective at attesting the trustworthiness of the execution environment only if we can guarantee that no attacker can operate in hypervisor mode. Therefore, the checksum function that attests the existence of a tamper-proof execution environment on the untrusted system must be adapted to compute the correct checksum value, in the expected amount of time, only when no hypervisor is running on the system.

There is a rich ongoing debate among researchers about hypervisors detection and hiding. Although, the hardware has been specifically designed to masquerade the existence of a piece of code running in hypervisor mode, everybody has become aware that constructing a completely transparent hypervisor is fundamentally infeasible and impractical from a computational and engineering prospective [16]. Indeed, hypervisors introduce several discrepancies, especially in terms of resources and timings. Our goal is to exploit these discrepancies, in particular timing discrepancies, to detect when the execution environment could not guarantee untampered execution. The main advantage we have over attackers is that checksum validation is performed by an external party, the verifier, that has a real perception of time. We exploit this advantage by including in the checksum function special gadgets that execute instructions that unconditionally trap to the hypervisor. Similarly to exceptions, hypervisor traps cause the CPU to spend several cycles to transition from system (or user) mode to hypervisor mode, to execute the handler of the hypervisor, and to transition back to system mode. By periodically executing such instructions, we cause a noticeable time overhead when a hypervisor is running on the untrusted system.

Figure 5(f) shows a sample gadget we use to detect hypervisors. The gadget reads a word from the memory (line 1), executes a `vmlaunch` instruction (line 3), and then updates the checksum (line 4 and 5). Other instructions, such as `cpuid`, `vmread`, and `vmcall`, can be used for this purpose. The `vmlaunch` instruction is available only on CPUs with hardware support for virtualization. Furthermore, the instruction can be executed only when virtualization support has been enabled. If a hypervisor is running on the untrusted system, any attempt to execute the instruction results in a trap to the hypervisor. In any other situation the CPU refuses to execute the instruction and generates an illegal operation exception. Recall that, by installing a custom IDT, we register handlers for all exception and that these handlers modify the running value of the checksum. In particular, the handler for illegal instruction exception we install additionally updates the address of the faulty instruction for resuming the normal execution of the checksum function from the next one. That is necessary to prevent an endless loop. To do not interfere with the correct checksum computation, after the trap, the attacker has to reproduce the situation that would occur on a system without hypervisor: he has to inject an illegal instruction exception into the guest to trigger the handler registered during the prologue. If the attacker mimics exactly the behavior of the CPU in the absence of the hypervisor, the checksum is computed correctly. However, the cost of the trap, of the execution of the logic to handle the trap, of the event injection, and of the exception handling we have on a system controlled by an attacker operating in hypervisor mode is much higher than the cost of the mere exception handling that we would have on a system without hypervisor. In conclusion, the gadget takes much longer to execute in an insecure execution environment. By executing this type of gadgets multiple times during the checksum loop we have the guarantee that, if the checksum computation produces the correct return value and it does not exceed the expected computation time, the execution environment is tamper-proof.

It is worth noting that if the attacker attempted to execute the checksum function directly in hypervisor mode, he would never be able to regain the control of the execution (this is the same case of an attacker that executes the checksum function in system mode without any hypervisor).

4.3 Obfuscation

After generation, the checksum function is obfuscated using simple obfuscation techniques [17]. Particular efforts are devoted to obfuscate the checksum loop because, by analyzing the loop, the attacker could identify the position of the various gadgets. The strategy we adopt is to introduce specific gadgets for obfuscating the logic of checksum computation. More precisely, these gadgets replace some of the existing gadgets and interrupt handlers with new ones. Furthermore, we obfuscate gadgets singularly by introducing dead code, overlapping instructions, and non-trivial pointers computations.

The gadgets we used for normal checksum computation give, as a side effect, an extra advantage for the verifier over the attacker. The presence of aggressive self-modifying code prevents the attacker from using efficient code emulations techniques, such as dynamic binary translation and software-based virtualization. Indeed, self-modifying code invalidates cached translated code, and forces the emulator to analyze and translate the code again and again. We have experienced directly this problem during the development of Conqueror: self-modifying code executed in system mode caused our development system, based on VirtualBox [18], to trash.

5 Evaluation

5.1 Prototype

We implemented a prototype of Conqueror to evaluate the effectiveness of our proposed solution. The prototype is specific for untrusted 32-bit systems running Microsoft Windows XP, and it consists in a hybrid user/kernel space component, implementing the verifier protocol, and a device driver that stays resident on the untrusted system.

When the verifier wants to bootstrap a tamper-proof execution environment on the untrusted system, it generates a new checksum function and encrypts it. Checksum functions are generated by leveraging a code generation module, currently written in Python. The verifier uses a kernel component to precisely measure packets transmission and arrival times. The kernel component running on the untrusted system passively waits for challenges. When challenged, it fills the TPEB with the encrypted checksum function; when the key is received, the attestation begins. To minimize network latency, both parties intercept challenge requests and responses through a hook installed in the network driver.

To experiment the feasibility of attacks based on hardware-assisted virtualization and their cost we also implemented a minimalistic hypervisor, inspired by the Blue Pill hypervisor [15], that simply resumes normal execution after traps. Obviously, any meaningful hypervisor must be much more sophisticated than this.

5.2 Experimental Setup

For our experiments we employed three laptops with the following characteristics: Intel Core2 Duo 2.1GHz, with 4GB RAM, and a Broadcom BCM5906M network card, connected on the same 100Mbps local network. The first laptop was used as a verifier, the second one as the untrusted system, and the third one as a trusted system. Since our current implementation does not support SMP, on the laptops we used as trusted and untrusted systems we disabled the secondary core of the CPU. In our experiments, the total size of the TPEB and the executable was fixed to six 4Kb pages.

5.3 Estimating the Parameters of the Challenge

To estimate the various parameters involved in the attestation scheme, we considered two attack scenarios: a dynamic hypervisor-based attack, and a static attack aiming to reverse engineer the checksum function.

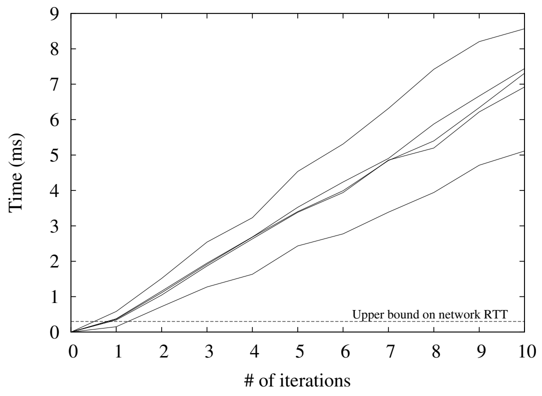


Fig. 6. Time overhead in a hypervisor-based attack

To understand how the various parameters of the challenge influenced the overall time to compute the checksum and to understand the opportunities of the attackers, we generated multiple checksum functions, varying the number and type of gadgets and the number of iterations of the checksum loop. After several experiments we decided to fix a minimum for the number of gadgets for "hypervisor detection". In each of the checksum functions we subsequently generated, at least 5% of the total of gadgets performed hypervisor detection.

To estimate the maximum checksum computation time and the network round-trip time (RTT), the verifier relies on a third-party trusted system, with the same hardware characteristics of the untrusted system. It is worth noting that checksum functions can be generated ahead of time and their execution time can be precomputed. Indeed, the running time depends only on the checksum function, on the CPU, and on the amount of data to attest. Given multiple measurements of the checksum computation time, we estimate the maximum computation time using Chebyshev's inequality, that states that for a random variable X, with mean value μ and standard deviation σ, $Pr(\mu - \sigma \leq X \leq \mu + \sigma) \geq 1 - \frac{1}{\lambda^2}$, where $\lambda \in \mathbb{R}$. In our context, X is the computation time, including the network RTT[1]. Therefore, the upper bound on checksum computation time is $\Delta_t = \mu + \lambda\sigma$, with confidence $\frac{1}{\lambda^2}$. Similarly, the minimum checksum computation time of the most powerful attacker (i.e., an attacker operating in hypervisor mode) is $\mu - \lambda\sigma$; in the calculation of the minimum computation time of the attacker we assumed the adversary to have a null network overhead.

The number of iterations of the checksum loop must be selected to force the time overhead suffered by the attacker to skyrocket. On the other hand, an excessive number of iterations would increase attacker's opportunities to reverse engineer the checksum function. The challenge is to find the best balance between the two. The approach we used was to generate multiple checksum functions, and

[1] Clearly attestation requires RTT to be minimal. The verifier can measure the RTT and wait to start the challenge if the RTT is too high.

to compare the time to compute the checksum in the trusted environment and in the environment controlled by the most powerful attacker. Figure 6 depicts the time overhead suffered by the attacker during our simulations, performed using five different checksum functions. More precisely, the figure shows the difference between the time to compute the checksum on the simulated untrusted system and on the trusted one. The simulation confirmed our hypothesis: the time overhead suffered by the attacker increases with the number of iterations of the checksum loop. According to our simulation two iterations are sufficient to detect an attack in our experimental scenario (attestation of six memory pages). However, to prevent false negatives, we doubled the number of iterations. Note that the number of iterations to detect a forgery is inversely proportional to the amount of memory to attest; thus, the number of iterations performed by the checksum loop can be tuned accordingly.

5.4 Experimental Results

Using the approaches described in the previous paragraphs we generated multiple challenges and used them to verify the effectiveness of Conqueror at detecting authentic checksum computations from forgeries. For clarity we refer to Δ_t, the upper bound of the checksum computation time estimated using Chebyshev's inequality, as the *attacker detection threshold*. In our experiments we chose $\lambda = 11$ to obtain an attacker detection rate with 99% confidence. For each challenge we estimated the attacker detection rate by challenging multiple times the trusted host. Subsequently we challenged the untrusted system twice: once the untrusted host simulated a genuine system (i.e., with no attacker), and once the host simulated the presence of the most powerful dynamic attacker (i.e., an attacker attempting to forge the checksum using a hypervisor-based attack). In all the challenges the untrusted system computed the correct checksum without exceeding the attacker detection rate. Similarly, in all the challenges the untrusted system under the control of the attacker did not compute the correct checksum in time to be considered authentic.

Figure 7 shows the details of one of the challenge we used during the experiment. The figure compares the time the untrusted system took to compute the checksum in the two aforementioned scenarios (the same challenge was repeated more than 50 times). Moreover, the figure shows the attacker detection threshold (Δ_t), and the lower bound for the most powerful attacker ($\mu_{\text{hvm}} - 11\sigma_{\text{hvm}}$). For the challenges in the figure, the average network RTT was less than 0.32ms, and the attacker detection rate was 112.44ms. Similarly, the lower bound for the computation of forged checksum was 115.56ms. The four ms difference and the very small variance between the two clearly indicate that false negatives are practically impossible. The data in the figure confirms the claim: no checksum was forged in time to be considered valid and no authentic checksum was considered forged.

The figure also compares the time requested to compute genuine checksums with the time the attacker would require to perform a preliminary static analysis (i.e., a recursive disassembly) of the checksum function. To measure to cost of the analysis, we loaded in Ida Pro [19], a widely used and well recognized

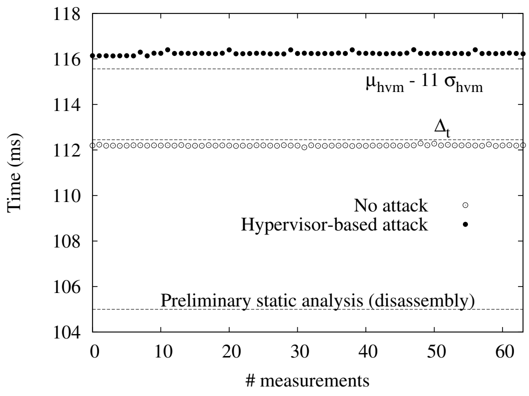


Fig. 7. Checksums computation time in different scenarios

disassembler, the checksum function and then measured the analysis time. Note that the checksum analyzed through Ida Pro was generated without employing any obfuscation technique because the disassembler would not have been able to analyze the code otherwise. The preliminary analysis took about 105ms, just four ms less than the attacker detection rate. Considering that disassembly is fundamental for any static analysis, and that any meaningful analysis to reconstruct the semantic of the checksum function costs much more, it is practically impossible for an attacker to forge a checksum without being detected.

5.5 A Real Application of Conqueror

Conqueror has been developed to build security applications that must be installed and executed on an untrusted system. All the aforementioned experiments were performed using dummy executables. Nevertheless, to demonstrate the versatility of Conqueror we have developed a special application intended to be run in the tamper-proof execution environment established by Conqueror. The application was a loader for a hypervisor. The goal was use this loader to install a *measured hypervisor* on an untrusted system [1], on-the-fly, and to segregate the untrusted system in a guest virtual machine. We successfully installed the hypervisor on our test untrusted system and then resumed the normal, but controlled, execution of the system. In conclusion, Conqueror represents a pure software alternative to the `senter` and `skinit` operations available in the Intel LaGrande [1] and AMD Pacifica [8] technologies for hypervisors secure *late launch*.

6 Discussion

Conqueror conservatively assumes that if a hypervisor is installed on the system, the hypervisor is malicious. It would be worthless to use Conqueror in a system that runs as a guest of a benign hypervisor: the dynamic root of trust could be established directly by the hypervisor.

The major limitation of Conqueror is the impossibility to bootstrap a tamper-proof environment on SMP and SMT systems. Most modern systems support symmetric flow of executions. An attacker could use the secondary computational resources to forge checksums or to regain control of the execution after attestation. Although we have not addressed the problem in detail, we would like to sketch a possible solution. The verifier can challenge the untrusted SMP (or SMT) system with multiple challenges simultaneously. More precisely, each processor is given a different checksum function to execute. To solve the challenge, the untrusted system has to compute all the checksums and send them back to the verifier, within the given time frame. Thus, the attacker is left with no spare computational resource to use.

7 Related Work

The majority of the research work on software-based attestation and verifiable code execution is specific for embedded devices and sensor networks. Most of the schemes are based on the same type of challenge and response protocol [3,4,5,6]; they have been thoroughly presented in Section 2. The strength and weaknesses of these schemes have been studied by Castelluccia *et al.* [20]. The approach used in Conqueror is instead inspired by the work done Shaneck *et al.* and by Garay *et al.* [2,7]. However, the two attestation schemes are also specific for embedded devices and not suited at all for attestation on legacy systems, the target of our work. Genuinity and Pioneer are two schemes, for environment attestation and verifiable code execution respectively, specific for legacy systems [21,6]. Unfortunately, both schemes are vulnerable to attacks. The vulnerabilities of the former have been studied by Shankar *et al.* [22]. The vulnerabilities of the latter have been introduced in Section 2.

The alternative approach to software-based attestation is hardware-based attestation. The research community spent a lot of efforts in developing hardware technology equipped with special trusted components to make hardware-based attestation practical. Examples of hardware technology with such capabilities are Cerium [23], BIND [24], Intel LaGrande Technology [1], and AMD Pacifica Technology [8]. In particular, thanks to the efforts of the Trusted Computing Group and the standardization of the TPM chip [25], Intel LaGrande and AMD Pacifica technologies are slowly becoming mainstream. They have been used as ground to develop various hardware-based attestation schemes. Examples of these schemes are the IBM Integrity measurements Architecture [26], the Open Source Loader [27], Terra [28], and Flicker [29]. Similarly to Conqueror and Pioneer, Flicker's goal is to achieve tamper-proof execution of code on untrusted systems. However, while Conqueror and Pioneer are entirely software-based solutions, Flicker leverages the TPM, available on modern commodity hardware, to accomplish the same goal. In particular Flicker relies on a feature introduced in the CPU that allows the secure late launch of virtual machine monitors.

8 Conclusions

We presented Conqueror, a software-based code attestation scheme for tamper-proof code execution on untrusted legacy systems. Conqueror allows to execute an arbitrary piece of code with the guarantee that it is run untampered, even when no specific hardware for trusted computing is available. We developed an experimental prototype of Conqueror, to evaluate its resilience against hypervisor-based attacks, the most powerful type of dynamic attack, and against attacks based on static analysis of the code. By leveraging Conqueror, we also developed a proof-of-concept pure software-based primitive to launch securely a hypervisor in a running untrusted system.

References

1. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press, Hillsboro (2009)
2. Garay, J.A., Huelsbergen, L.: Software integrity protection using timed executable agents. In: Proceedings of the 2006 ACM Symposium on Information, computer and communications security, ASIACCS (2006)
3. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: Swatt: Software-based attestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy (2004)
4. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: Scuba: Secure code update by attestation in sensor networks. In: Proceedings of the ACM Workshop on Wireless Security, WiSe (2006)
5. Seshadri, A., Luk, M., Perrig, A.: SAKE: Software attestation for key establishment in sensor networks. In: Nikoletseas, S.E., Chlebus, B.S., Johnson, D.B., Krishnamachari, B. (eds.) DCOSS 2008. LNCS, vol. 5067, pp. 372–385. Springer, Heidelberg (2008)
6. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In: Proceedings of ACM Symposium on Operating Systems Principles, SOSP (2005), `http://www.cs.cmu.edu/~arvinds/pioneer.html`
7. Shaneck, M., Mahadevan, K., Kher, V., Kim, Y.: Remote software-based attestation for wireless sensors. In: Molva, R., Tsudik, G., Westhoff, D. (eds.) ESAS 2005. LNCS, vol. 3813, pp. 27–41. Springer, Heidelberg (2005)
8. AMD, Inc.: AMD Virtualization, `http://www.amd.com/virtualization`
9. Wurster, G., van Oorschot, P.C., Somayaji, A.: A Generic Attack on Checksumming-Based Software Tamper Resistance. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy (2005)
10. Intel, Inc.: Intel Virtualization Technology, `http://www.intel.com/technology/virtualization/`
11. Klimov, A., Shamir, A.: A New Class of Invertible Mappings. In: Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems (2003)
12. Robin, J.S., Irvine, C.E.: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine monitor. In: Proceedings of the 9th USENIX Security Symposium (2000)

13. Giffin, J., Christodorescu, M., Kruger, L.: Strengthening software self-checksumming via self-modifying code. In: Proceedings of the 21st Annual Computer Security Applications Conference, ACSAC (2005)
14. Dai Zovi, D.: Hardware Virtualization Based Rootkits. Black Hat USA (2006), http://blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf
15. Rutkowska, J.: Subverting Vista Kernel For Fun And Profit. Black Hat USA, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf
16. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI) (2007)
17. Linn, C., Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In: Proceedings of the 10th ACM conference on Computer and communications security, CCS (2003)
18. Sun Microsystems, Inc.: Sun xVM VirtualBox, http://www.virtualbox.org/
19. Hex-Rays: IDA Pro., http://www.hex-rays.com/idapro/
20. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the Difficulty of Software-Based Attestation of Embedded Devices. In: Proceedings of the 16th ACM conference on Computer and Communications Security, CCS (2009)
21. Kennell, R., Jamieson, L.H.: Establishing the genuinity of remote computer systems. In: Proceedings of the 12th USENIX Security Symposium (2003)
22. Shankar, U., Chew, M., Tygar, J.: Side effects are not sufficient to authenticate software. In: Proceedings of the 13th USENIX Security Symposium (2004)
23. Chen, B., Morris, R.: Certifying Program Execution with Secure Processors. In: Proceedings of the 9th conference on Hot Topics in Operating Systems (2003)
24. Shi, E., Perrig, A., Van Doorn, L.: BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy (2005)
25. Trusted Computing Group: http://www.trustedcomputinggroup.org/
26. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: Proceedings of the 13th USENIX Security Symposium (2004)
27. Kauer, B.: OSLO: Improving the Security of Trusted Computing. In: Proceedings of 16th USENIX Security Symposium (2007)
28. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a Virtual Machine-based Platform for Trusted Computing. In: Proceedings of the nineteenth ACM symposium on Operating systems principles (2003)
29. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for tcb minimization. In: Proceedings of the ACM European Conference in Computer Systems, EuroSys (2008)

*d*Anubis – Dynamic Device Driver Analysis Based on Virtual Machine Introspection

Matthias Neugschwandtner, Christian Platzer, Paolo Milani Comparetti, and Ulrich Bayer

Secure Systems Lab, Vienna University of Technology
{mneug,cplatzer,pmilani,ulli}@seclab.tuwien.ac.at

Abstract. In the escalating arms race between malicious code and security tools designed to analyze it, detect it or mitigate its impact, malicious code running inside the operating system kernel provides an extremely powerful tool. Kernel-level code can introduce hard to detect backdoors, provide stealth by hiding files, processes or other resources and in general tamper with operating system code and data in arbitrary ways.

Under Windows, kernel-level malicious code typically takes the form of a device driver. In this work, we present *d*Anubis, a system for the real-time, dynamic analysis of malicious Windows device drivers. *d*Anubis can automatically provide a high-level, human-readable report of a driver's behavior on the system. We applied our system to a dataset of over 400 malware samples. The results of this analysis shed some light on the behavior of kernel-level malicious code that is in the wild today.

1 Introduction

Malicious code, or malware, is at the root of many security problems on the internet. Compromised computers running malware join botnets and participate in harmful activities such as spam, identity theft and distributed denial of service attacks. It is therefore no surprise that a large body of previous research has focused on collecting, detecting, analysing and mitigating the impact of malicious code.

The analysis of malicious code is an important element of current efforts to protect computer users and systems from malware. Understanding the impact of a malware sample allows to evaluate the risk it poses and helps develop detection signatures, removal tools and mitigation strategies. Because of the large number of new malware samples that appear in the wild each day, malware analysis needs to be a largely automated process.

The automatic analysis of malicious programs is complicated by the fact that malware authors can use off-the-shelf packers to make their samples extremely resistant to static code-analysis techniques. According to a recent large-scale study of current malware [1], over 40% of malware samples are packed using a known packer. Clearly, this is a lower bound to the amount of malware that is packed because malware authors may be using other, yet-unknown packers or implement their own custom solutions. While many current packers can be defeated by generic unpacking tools [2,3], packers that use emulation-based packing can currently be fully defeated only after manually reverse-engineering their emulator [4]. Furthermore, packers based on opaque constants [5], while not yet available in the wild, can generate binaries that are provably hard to analyze for any static code analyzer.

Because of these limitations, automatic malware analysis is mostly based on a dynamic approach: Malware samples are executed in an instrumented sandbox environment, and their behavior is observed and recorded. A number of dynamic malware analysis systems are currently available that can provide a human-readable report on the malware's activities [6,7]. The output of these tools can further be used to find clusters of samples with similar behavior [8,9,10], or to detect specific classes of malicious activity [11].

These systems are able to analyse the behavior of malicious code running in user-mode. The analysis of kernel-side malicious code, however, poses additional challenges. First of all, kernel-level malicious code cannot be reliably detected or analyzed unless the analysis is performed at a higher privilege level than the kernel itself. Otherwise, kernel-level malware would be able to tamper with or disable the analysis engine, in a never-ending arms race. This challenge can be overcome by using out-of-the-box, Virtual Machine Introspection techniques [12], or with more recent in-the-box monitoring techniques that leverage modern CPU features to protect the analysis engine [13]. Using such techniques, the injection and execution of code into kernel-space can be reliably detected [14,15].

Beyond detection, however, understanding the purpose and capabilities of malicious kernel code is also useful. This is challenging because, in contrast to a user-mode process, kernel code is not restricted to its own address space and to interacting with the rest of the system through a well-defined system call interface. When monitoring the behavior of a system infected by kernel-side malicious code, it is not trivial to reliably (a) attribute an observed event to the malicious code or to the benign kernel and (b) understand the high-level semantics of an observed event. In the limit, kernel-level malware could replace the entire operating system kernel with its own implementation, making understanding the differences between the behavior of a clean system and an infected one extremely challenging. In practice, malware authors prefer to perform targeted manipulations of the operating system's behavior using hooking techniques, and to make use of functions offered by the kernel rather than re-implement existing functionality. Therefore, detecting malware hooking behavior has been the focus of a significant body of recent research [16,17,18,19].

One aspect of malicious kernel code that has received less attention is device driver behavior. That is, the malware's interaction with the system's IO driver stacks, and the interface and functionality it offers to userland processes. In this work, we attempt to provide a more complete picture of the behavior of malicious kernel code. We introduce *d*Anubis, an extension to the Anubis dynamic malware analysis system[20] for the analysis of malicious Windows device drivers. *d*Anubis can automatically generate a human-readable report of the behavior of kernel malware. In addition to providing information on the use of common rootkit techniques such as call hooking, kernel patching and Direct Kernel Object Manipulation (DKOM), *d*Anubis provides information about a malicious driver's interaction with other drivers and the interface it offers to userspace. To improve the coverage of its dynamic analysis, *d*Anubis also includes a stimulation engine that attempts to trigger rootkit functionality. Running *d*Anubis on over 400 malware samples that include kernel components allows us not only to validate our tool, but also to perform the largest study of kernel-level malware to date.

In summary, our contributions are the following.

1. We present *d*Anubis, a system for the real-time dynamic analysis of malicious Windows device drivers.
2. Using *d*Anubis, we analyzed over 400 hundred samples and present the results of the first large-scale study of Windows kernel malware. These results give insight into current kernel malware and provide directions for future research.
3. *d*Anubis will be integrated into the Anubis malware analysis service, making it available to researchers and security professionals worldwide.

2 Overview

Rootkits provide malware authors with one of their most flexible and powerful tools. The term "rootkit" derives from their original purpose of maintaining root access after exploiting a system, being a "kit" of pieces of technology with the purpose to hide the attacker's presence in the system [21]. This can include hiding files, processes, registry keys and network ports that could reveal an intruder's access to the system. Early rootkits ran entirely in user space and operated by replacing system utilities such as ls, ps and netstat with versions modified to hide the activities of an unauthorized user. Later rootkits included kernel-level code, enabling the attacker to do virtually anything on the target machine, including directly tampering with control flow and data structures of the operating system. Today, the boundaries between different classes of malware have become indistinct; many techniques originally used in rootkits are now employed in other types of malware, such as bots, worms or Trojan horses. In this paper, we will use the term *rootkit* to refer to malware that uses kernel-level code to carry out its operations.

To inject malicious code into the kernel, the attacker can either use an undetected, unpatched kernel exploit, such as a buffer overflow, or – much more convenient – load and install a device driver. The latter method has the disadvantage that it depends on hijacking an administrator account. This is in practice not much of a problem since most Windows machines are operated with Administrator privileges out of convenience for the user. While Windows Vista or 7 at least require the user to confirm administrative actions such as driver loading, Windows XP provides APIs that allow loading an unsigned driver without any user interaction. As a result, a rootkit usually comes as a user-mode executable that loads a device driver, which in turn provides all the powerful functionality.

The goal of *d*Anubis is to provide a human-readable report of a device driver's behavior. Note that detection, that is, distinguishing malicious device drivers from benign ones, is outside the scope of this work. Some behavior, such as directly patching kernel code, may give a clear indication that a sample is malicious. Many types of suspicious behavior, however, may also be exhibited by benign code, especially security tools such as antivirus or personal firewall software. The reason is that these tools may attempt to "outsmart" malware by running deep inside the operating system.

*d*Anubis analyses a driver's behavior from outside the box, using a Virtual Machine Introspection (VMI) approach [12,22]. Our implementation is an extension of the Anubis malware analysis system, and is based on the Qemu [23] emulator. By instrumenting the emulator, we can monitor the execution of code in the guest OS, to observe events such as the execution of the malicious driver's code, invocation of kernel functions, or access to the guest's virtual hardware. Furthermore, by instrumenting the emulator's Memory Management Unit (MMU), we can observe the manipulation of kernel memory performed by the rootkit. *d*Anubis attempts to reconstruct the high level semantics of the observed events.

One focus of our analysis is to monitor all the "legitimate" communication channels between the rootkit and the rest of the system. That is, all channels provided by the OS for the driver to interact with the kernel, with other drivers and with user-space. This includes the invocation of kernel functions as well as the use of devices to participate in Windows I/O communication and to receive commands from user-space. Additionally, *d*Anubis can detect the use of a number of rootkit techniques such as hooking and runtime patching of kernel routines, and provide precise information on which routines are patched or hijacked. Overall, our tool can thus provide a comprehensive picture of the behavior of malicious kernel code.

3 System Implementation

A major drawback of any VMI-based approach is the loss of semantic information about the guest operating system. Instead of objects and well-defined data structures, only a heap of bytes is visible from the host system's point of view. To reconstruct the necessary information we extract all exported symbols and data structure layouts from the Windows OS as a preliminary step. During analysis, we utilize guest view casting of the virtual machine memory as proposed by Jiang et al. [22].

A further problem arises when comparing process-based dynamic analysis, as it is implemented in Anubis or comparable sandboxes, and driver-aware approaches. For userland processes, it is sufficient to watch and trace instructions belonging to the process in question, whose execution context is easily identifiable. Kernel-level code, however, can be triggered by multiple means, like interrupts or system calls, thus possibly running in the context of an arbitrary user-mode process. Therefore, we use the instruction pointer to determine whether the code being executed belongs to the malicious driver.

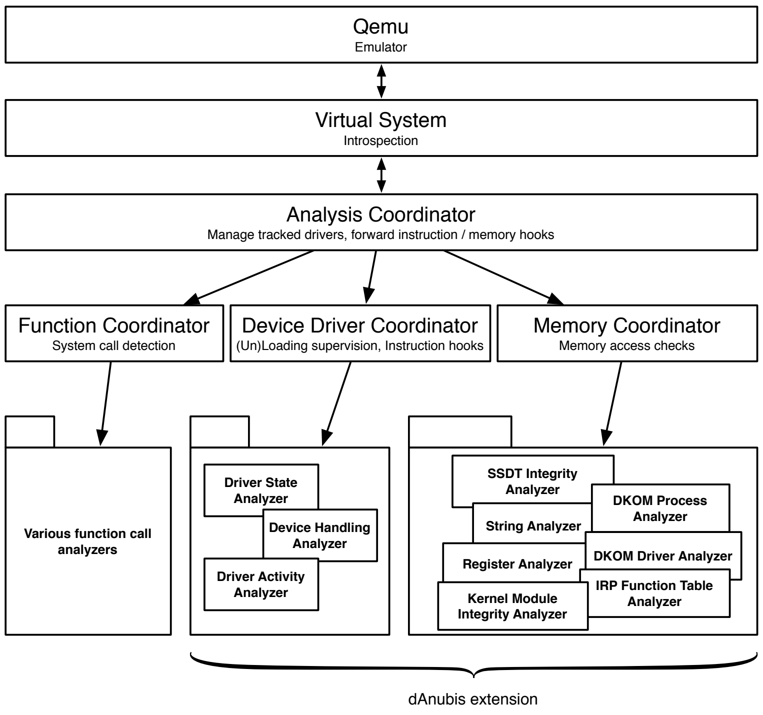


Fig. 1. Architectural overview

Our system consists of two major parts. The *Device Driver Coordinator* handles device driver-related operations while the *Memory Coordinator* is responsible for rootkit activity. Specific analysis tasks are carried out by a number of *Analyzers*. Figure 1 provides an overview of our system's architecture.

3.1 Device Driver Analysis

To analyze the behavior of device drivers, we monitor all available interfaces through which the driver can interact with the rest of the kernel, with other drivers and with userland processes [24]. The first thing to do in this respect is intercepting the low-level load- and unload mechanisms. The Windows kernel objects involved in the loading procedure provide us with import information, above all the codebase location of the driver, which allows us to track instructions belonging to the driver. Furthermore, the function addresses of the driver's entry routine (comparable to the main function of a normal program), its unload routine and its I/O dispatch routines can be gathered. Among the latter are the *major functions*, a set of well-defined functions a driver has to provide in order to participate in Windows I/O device communication. Knowing the function addresses allows us to implement a basic state supervision and relate later analysis events to the context in which they occurred.

Driver communication. The communication endpoint of a driver is the device. To receive I/O requests, a driver has to create a device and provide the aforementioned major functions to handle the requests. If requests require complex processing that can be subdivided in different parts, devices can be arranged in a stack. For example this could be the case for an encrypted file system, where the encryption is handled by the topmost driver in the stack and actual hardware access by the lowest driver. Driver stacking can be exploited for malicious purposes. For example, a rootkit may attach to the filesystem device stack to filter the results of file listings before forwarding them up the stack, while a keylogger can attach to the keyboard device to monitor all keystrokes. Therefore, we monitor whether a driver creates or attaches to a device.

Actual communication between devices and user- or kernel-mode code happens by encapsulating the request parameters in an I/O request packet (IRP) by the Windows I/O manager, which invokes the corresponding major function of the topmost driver in the stack. The IRP is then passed down and up its destined device stack. We intercept calls to attached devices, analyze these IRPs and watch for completion routines, which are invoked upon completion of a request, allowing them to filter results. Larger data amounts are not directly passed within the IRP, rather the way how it can be done – buffered I/O, direct I/O or neither of them – is specified. We also parse this information and detect strings in the data to be able to track further references to them during execution. This is accomplished using dynamic data tainting [25]. Specifically, we use techniques from [9] to detect the use of these tainted bytes in string comparison operations. This can in some cases reveal triggers for conditional behavior of the analyzed binary.

Driver activity. To get a picture of what the driver actually does when one of its functions is executed, we log calls to all exported Windows kernel functions. We expect that rootkit developers will not develop everything from scratch but make use of existing functionality. In our evaluation, we show that this assumption proves correct for most real-world samples.

We also scan the complete driver image for string occurrences and taint them to be able to log any subsequent access to such strings. As we will show in the evaluation, this simple mechanism can in practice reveal trigger conditions in the malicious code, such as the names of files and processes that are to be hidden.

3.2 Memory Analysis

Taking a look at the "standard" rootkit techniques, one similarity is obvious: they all somehow tamper with kernel memory. We achieve the goal of detecting malicious kernel memory manipulation by hooking the memory management unit (MMU) of Qemu. This allows us to detect write access independently of the instruction used so that we can put specific memory regions of the guest OS under supervision and analyse malicious changes.

Since some kernel regions are flagged read-only, rootkits often use memory descriptor lists (MDLs) to re-map a desired memory region to a new virtual address and bypass write protection. Therefore *d*Anubis also needs to analyze MDL usage.

Call table hooking. The first interesting memory region is represented by the system service dispatch table (SSDT). This table keeps a list of Windows system call handlers that can be invoked by usermode processes by issuing an interrupt or using the `sysenter` instruction. In a healthy system, the called table entry points to the beginning of the desired service routine. Malicious drivers, however, can overwrite the SSDT entry to point to arbitrary code. When called, this code typically forwards the request to the original service function, receives the response and alters it before passing it to the original caller. Again, this method provides the possibility to exclude rootkit-related information from queries like directory listings or process lists.

The same principle applies to hooks of other call tables, such as the major function dispatch table of device drivers. For incoming IRPs the Windows I/O manager normally looks up the proper handling routine in this table before invoking it. Again, the control flow can be re-routed by changing an entry in this table.

To monitor call table hooking behavior, we watch the complete memory region where a call table resides. If a manipulation occurs in one of the watched memory regions, we will know exactly which system service or major function has been hooked and monitor following calls to the hook.

DKOM. Direct kernel object manipulation (DKOM) modifies important data objects residing in Windows kernel memory. This way it can alter system behavior without changing the control flow. To hide a certain process from the process list, for example, the `EPROCESS` structure has to be altered such, that the forward and backward pointers are directed around the target entry, effectively excluding it. Although this method is more powerful and harder to detect than hooking, it has its shortcomings. It is, for instance, not feasible to mask a file from a directory listing this way.

To detect and understand DKOM activity, it is necessary to know the exact location of the kernel objects as well as their data structure and meaning. In our current implementation, DKOM detection is limited to the process and driver lists, that are frequently targeted by rootkits for the purpose of stealth.

Runtime patching. Rootkits can also affect the system by directly patching existing kernel code in memory. Usually the patch jumps to a detour containing malicious code and then back again to the original code.

To detect runtime patching, we walk through the `PsLoadedModuleList` to get the information on the codebase of the kernel modules and put them under supervision. On an integrity breach we determine exactly which kernel function has been patched by matching the patched addresses against information automatically obtained from the Windows debugging symbols.

Hardware access. In addition to manipulating kernel memory, rootkits can affect the system by directly accessing the underlying hardware. *d*Anubis monitoring of hardware access is currently limited to detecting writes to the `IA32_SYSENTER_EIP` model specific register. This register points to the system service dispatcher routine. Making this register point to malicious code places rootkit code in the execution path of all system calls.

3.3 Stimulation

Rootkit functionality often depends on external stimuli. While the entry routine may already perform some malicious activity such as hooking or patching, many types of behavior may only be performed when triggered by specific user behavior. Without the required stimuli, such as keyboard events for keyloggers or process enumeration for process-hiding rootkits, the results of dynamic analysis are bound to be incomplete.

Our goal is to improve code coverage by simulating user activity with a stimulation engine placed in the virtual machine. To this end, we implemented a stimulator that repeatedly issues a number of Windows API calls. For example, the stimulator issues the `EnumProcesses` API call, that lists all currently running processes, triggering process hiding behavior. Similarly, it issues the `RegEnumKeyEx` call, revealing register hiding. The `FindFirstFile` and `FindNextFile` API calls are used as well to reveal file hiding behavior. Note that although the directory we are querying with these calls might not contain files to be hidden, hook code will nevertheless be executed. To trigger network hiding behavior, the `GetUdpTable` and `GetTcpTable` calls are used. Furthermore, random keypresses and mouse actions are injected to simulate user input.

4 Evaluation

To evaluate our prototype, we first verified its functionality on a set of rootkits with known behavior. For this, we chose a representative suite of six well-known rootkits that employ the popular techniques described in the previous sections and can be obtained from `www.rootkit.com`. For each of the six rootkits, *d*Anubis was able to correctly identify its characteristic behavior and present it in the human-readable report. Table 1 shows which *d*Anubis components were involved in providing information on each of the rootkits.

The first sample we selected is **TCPIRPHook**. This malware modifies the address of `DEVICE_CONTROL` in the major function table of Tcpip.sys, rerouting it to a hooking function. This allows the rootkit to hide open network ports. This hooking behavior was detected by the IRP function table analyzer.

We then selected the **HideProcessMDL** rootkit as a straightforward example of process hiding by SSDT hooking. This rootkit first creates an MDL in order to gain write access to the SSDT. This is recognized by the memory coordinator, that can thus apply the mapping upon write access to the watched memory region. This enables the SSDT integrity analyzer to report the rootkit's hooking of `NtQuerySystemInformation` as soon as the hook is placed. Once the stimulator queries for the running processes, the driver state analyzer detects the call to the hooking routine, which in turn invokes the original `NtQuerySystemInformation` function. Furthermore, The string analyzer reveals that the hooking routine accesses the string "_root_" indicating the name of the process that is to be hidden.

Klog is a key-logger based on layered filter drivers. During driver initialization it creates a log file and a virtual device, which it uses to attach to the keyboard device stack. This behavior, along with name and location of the log file, is revealed by the device handling, driver activity and string analyzer. Upon stimulation of keystrokes, the device handling analyzer further detects that the driver Kbdclass is called by Klog and dynamically adds the completion routine to the state analysis and so execution of the completion routine is subsequently logged.

Migbot uses run-time patching to modify the kernel functions `SeAccessCheck` and `NtDeviceIoControlFile`. The integrity breach along with the names of the functions is immediately reported by the kernel integrity analyzer.

The **FU** rootkit uses DKOM for process and driver hiding. However, it only performs this hiding function when it receives commands from a user-mode program through device communication. To test the DKOM analyzers we manually ordered FU to hide certain processes and drivers. These manipulations along with the corresponding filenames were immediately reported by the DKOM analyzers. Furthermore, the device handling analyzer revealed the string "msdirectx.sys" in the communication with the user-mode program. This is the name of the driver we ordered FU to hide.

Finally, we tested the **sysenter** rootkit to verify that sysenter hooks are correctly recognized.

Table 1. *d*Anubis Testing results

Analyzer	TCPIRPHook	HideProcessMDL	Migbot	Klog	FU	sysenter
Device driver coordinator	√	√	√	√	√	√
Memory coordinator	-	√	-	-	-	-
Driver state analyzer	√	√	√	√	√	√
Driver activity analyzer	√	√	√	√	√	-
IRP function table analyzer	√	-	-	-	-	-
SSDT analyzer	-	√	-	-	-	-
String analyzer	-	√	-	√	√	-
Integrity analyzer	-	-	√	-	-	-
Device handling analyzer	-	-	-	√	√	-
DKOM process analyzer	-	-	-	-	√	-
DKOM driver analyzer	-	-	-	-	√	-
Register analyzer	-	-	-	-	-	√

During the analyis of these six rootkits, we also measured the impact of *d*Anubis on the performance of the Anubis sandbox. The overhead added by *d*Anubis was between 14% and 33%. These results are consistent with our goal of integrating driver analyis into a large-scale dynamic analysis framework, because the entire analysis of a malware sample can still be performed in real time, within the six minute timeslot that Anubis typically allocates to an analysis run. This is in contrast to some previous systems, such as K-Tracer [18], that need to perform a heavyweight analysis of detailed execution traces. For instance, K-Tracer needed over two hours to analyze the HideProcessMDL rootkit.

4.1 Quantitative Results

We used *d*Anubis to conduct a large-scale study of kernel malware behavior. To obtain malware samples for this study, we leveraged the analysis results of the existing Anubis system. We first considered 64733 malware samples successfully analysed by Anubis in the month of August 2009. Among those, we selected the 463 samples (0.72%) that loaded a device driver during Anubis analysis. More precisely, we selected samples that performed the `NtLoadDeviceDriver` system call. We then repeated the analysis of these samples using *d*Anubis. Note that some malware may use different mechanisms to load kernel code, such as the undocumented `NtSetSystemInformation` system call. Therefore, the actual number of rootkit samples in the dataset may have been higher than 463. While *d*Anubis is capable of correctly analysing rootkits loaded using such methods, the legacy Anubis system does not detect and log this behavior.

All samples were automatically processed by our implementation and correctly recognized as drivers. For each test run, we defined a timeout of six minutes, during which the driver had time to carry out its operations. During the entire analysis stimuli where provided by our stimulation engine. Table 2 shows which high-level activity of the samples could be observed by *d*Anubis. Three quarters of the samples performed device I/O activity. Among the typical rootkit techniques, MDL-enabled call table hooks and runtime patching seem to be very popular compared to DKOM.

Table 3 shows an overview of device-related activity. The majority of the samples – 339 – created at least one device. In 110 cases the device was actively used for communication by a user mode program: It was at least opened, as indicated by the calls to the CREATE major function. Out of these, 86 samples carried out further communication using the device control interface. In the data buffers passed along with the IRPs, meaningful communication strings could be found in 24 cases (an example is shown in Table 7). Only two samples attached to a device stack and registered completion routines. These results allow us to draw the conclusion that devices are primarily used for communication with user mode programs whereas hijacking device stacks seems to be far less popular. However, a significant amount of samples – 229 – register a device, but this device is never put to any use during the entire analysis run. The most likely explaination for

Table 2. Global analysis statistics

Driver activity	number of samples exhibiting behavior
Device driver loaded	463
Windows kernel functions used	360
Windows device I/O used	339
Strings accessed	300
Kernel code patched	76
Kernel call tables manipulated	37
MDL allocated	34
Kernel object manipulated	3

Table 3. Device analysis statistics

Device activity	number of samples
Device created	339
Driver's device accessed from user mode	110
Strings detected during communication	24
Attaches to device stack	2
Registers completion routine	2

Table 4. Hiding statistics: subject vs. technique

	SSDT hook	runtime patching	DKOM	IRP hook	Filter driver	total
Registry	5	45	0	0	0	50
File	8	2	0	0	2	12
Process	3	2	3	0	0	8
Driver	0	0	3	0	0	3
Network port	5	0	0	1	0	6

this discrepancy is that the associated executables are merely launchers for the drivers, that in turn wait for further commands to be manually issued by the human attacker. This is the case of the FU rootkit we previously discussed. This means that some of the malicious functionality of these rootkits lies dormant, waiting for activation, and is therefore not covered by the dynamic analysis. This result highlights the need for further research in rootkit analysis. Future analysis systems might be able to automatically trigger rootkits' dormant functionality, although the problem of finding trigger conditions in arbitary code cannot be solved in general [26].

Overall, only 15% percent of the samples carried out rootkit activities. Table 4 shows the amount of samples that provided stealth broken down by the techniques employed as well as the type of object being hidden. Clearly, call table hooking and runtime patching are the more widespread techniques: only three samples used DKOM for process hiding. The same samples also used DKOM to hide their device drivers from the list of kernel modules.

Of the 19 samples that employed SSDT hooking, most hooked more than one system call. Table 5 shows the most popular system calls hooked. The idea that rootkits strive to provide stealth is confirmed by the fact that the system calls to list files, registry keys and processes are clearly favored by the attackers. In addition to stealth, another common goal of rootkits is to disable antivirus protection. Samples hooking the `NtCreateProcessEx` use this to prevent the launch of anti-malware programs. IRP function table hooks were only employed by one sample, that hooked the `DEVICE_CONTROL` major function of Tcpip.sys. Rerouting the device control interface, that is the main communication access point to the driver, allows this rootkit to hide open network ports. A less subtle method of hijacking the device control interface is to directly hook the `NtDeviceIoControlFile` system call. This technique is used by five samples, also for the purpose of port hiding. None of the samples used sysenter

Table 5. Hooked system calls

System service	samples
NtQueryDirectoryFile	8
NtCreateProcessEx	8
NtDeviceIoControlFile	5
NtEnumerateKey	3
NtQuerySystemInformation	3
NtEnumerateValueKey	2
NtOpenKey	2
NtClose	1
NtCreateKey	1
NtSetInformationFile	1
NtSystemDebugControl	1
NtOpenProcess	1
NtOpenThread	1
NtCreateFile	1
NtOpenIoCompletion	1
NtSetValueKey	1
NtDeleteValueKey	1
NtMapViewOfSection	1

Table 6. Patched kernel functions

Kernel function	samples
NtQueryValueKey	42
NtSetValueKey	2
PsActiveProcessHead	2
NtEnumerateKey	1
IoCreateFile	1
NtQueryDirectoryFile	1
NtOpenKey	1
NtCreateKey	1
pIofCallDriver	1
KiFastCallEntry	1
ObReferenceObjectByHandle	1
KiDoubleFaultStack	1

hooks. The StringAnalyzer, that was mainly introduced to reveal trigger conditions, shows its full potential with SSDT hooks. For example in more than half of the cases where `NtQueryDirectoryFile` or `NtQuerySystemInformation` has been hooked, the filenames to be hidden showed up in the analysis. This also demonstrates the importance of event stimulation and the effectiveness of our stimulation engine, as the strings were mainly detected during execution of hooking routines that would not have been called without stimulation.

About ten percent of the samples used runtime patching. In these cases *d*Anubis took advantage of kernel debugging symbols to automatically identify the patched kernel functions.

Table 6 shows that, as is the case for SSDT hooks, functions that take part in processing file, process and registry key queries are among the most popular kernel functions to be manipulated. The `pIofCallDriver` pointer points to the low-level kernel code implementation that invokes the major functions of a driver. Rerouting its control flow allows a rootkit to intercept and manipulate IRPs. The `KiFastCallEntry` function is the default handler of the `sysenter` instruction. By patching this function, a rootkit inserts malicious code into the code path of every system call. In this case the automatic analysis cannot tell us what types of objects are actually being hidden by this rootkit.

4.2 Qualitative Results

To provide a clearer picture of the types of behavior that can be revealed by the analysis of a kernel malware sample using *d*Anubis, we selected three interesting samples out of the dataset discussed in the previous section. For matters of space and readability the relevant information from the reports has been condensed into tables.

In Table 7 a selection of analysis results of Sample A are shown. This rootkit hooks various system calls, among them functions suitable for file and registry key hiding. Process hiding is performed using DKOM. As the hooking functions are very similarly structured, the hook of NtEnumerateKey has been chosen as an example. After calling the original function it queries an object and its name. It then performs some string operations, which is usually necessary for filtering information. Furthermore, in the course of the driver's entry function a device is created. This device is then used for user mode communication: DEVICE_CONTROL is called from user mode several times and both a registry key and a file name could be intercepted, that the driver is presumably ordered to hide. DEVICE_CONTROL itself looks up objects according to their name or ID using ObReferenceObjectByName and PsLookupProcessByProcessId – again an indication that these objects are to be hidden.

In Table 8 selected analysis results of Sample B are shown. During driver entry, this sample creates a named device (FILEMON701) for communication with user mode. This device is then used to issue commands to the driver via `FastIoDeviceControl` to install filter drivers for sr.sys and mrxsmb.sys. To this end, two unnamed devices are created and attached to the associated device stacks.

The sr.sys driver is the Windows restore filesystem filter driver, that tracks and copies system files before changes. The mrxsmb.sys is the Windows SMB Redirector, a filesystem driver that provides access to remote folders shared over the SMB/CIFS protocol. Our stimulation engine, however, does not perform operations on network shares, nor does it modify system files. Therefore, during anlysis we only observed interception of `QUERY_VOLUME_INFORMATION` of sr.sys, that is used to query free disk space or file types. This highlights the challenge of

Table 7. Analysis report, Sample A

Driver name	syssrv	
Created devices	\Device\MyDriver	
Rootkit activity	NtOpenProcess hooked	SSDT Hook
	NtOpenThread hooked	SSDT Hook
	NtCreateFile hooked	SSDT Hook
	NtOpenIoCompletion hooked	SSDT Hook
	NtQueryDirectoryFile hooked	SSDT Hook
	NtOpenKey hooked	SSDT Hook
	NtEnumerateKey hooked	SSDT Hook
	NtEnumerateValueKey hooked	SSDT Hook
	NtSetValueKey hooked	SSDT Hook
	NtDeleteValueKey hooked	SSDT Hook
	svchost.exe hidden	DKOM process hiding
	ntoskrnl.exe: PsActiveProcessHead	Runtime patching
Invoked major functions	CREATE	called 5x from user mode
	DEVICE_CONTROL	called 5x from user mode
	CLOSE	called 5x from kernel mode
Detected strings	syssrv	in DEVICE_CONTROL IRP
	\Device\HarddiskVolume1 \WINDOWS\system32\mssrv32.exe	in DEVICE_CONTROL IRP
	SOFTWARE\Microsoft\Windows \CurrentVersion\Run\mssrv32	in DEVICE_CONTROL IRP
	\Device\%s	during entry
	MyDriver	during entry
Used kernel functions	IoCreateDevice	during entry
	KeInitializeMutex	during entry
	ObReferenceObjectByName	during DEVICE_CONTROL
	ObReferenceObjectByHandle	during DEVICE_CONTROL
	ObQueryNameString	during DEVICE_CONTROL
	KeWaitForSingleObject	during DEVICE_CONTROL
	KeReleaseMutex	during DEVICE_CONTROL
	PsLookupProcessByProcessId	during DEVICE_CONTROL
	NtEnumerateKey	during NtEnumerateKey Hook
	ObReferenceObjectByHandle	during NtEnumerateKey Hook
	ObQueryNameString	during NtEnumerateKey Hook
	wcslen, wcscpy, wcscat	during NtEnumerateKey Hook
	KeWaitForSingleObject	during NtEnumerateKey Hook
	KeReleaseMutex	during NtEnumerateKey Hook

implementing a stimulation engine that is capable of activating all hooks inserted by a rootkit. Note that generic hook-detection tecniques such as Hookfinder [17] and KTracer [18] cannot detect hooks that are never activated.

The Windows system restore functionality can be used to perform a system rollback based on the information gathered by the sr.sys driver. By attaching to sr.sys, the rootkit can prevent system restore from obtaining the necesary information on system file changes and ensure that the rootkit will not be removed by a rollback. The device name and symbols included in the rootkit's executable suggest that the publicly available sources of the Filemon tool [27] were used as a basis for this rootkit.

Table 8. Analysis report, Sample B

Driver name	FILEMON701	
Created devices	\Device\Filemon701	
	unnamed device 1	
	unnamed device 2	
Attached to devices	sr	
	MRxSmb	
Completion routine	QUERY_VOLUME_INFORMATION	for device ”sr”
Invoked I/O functions	CREATE	from user mode
	QUERY_VOLUME_INFORMATION	from kernel mode
	CLEANUP	from kernel mode
	CLOSE	from kernel mode
	READ	from kernel mode
	FastIoDeviceControl	
Used kernel functions	IoCreateDevice	during entry
	IoCreateSymbolicLink	during entry
	IoGetCurrentProcess	during entry
	ZwCreateFile	during FastIoDeviceControl
	IoCreateDevice	during FastIoDeviceControl
	IoAttachDeviceByPointer	during FastIoDeviceControl

Table 9. Processes targeted by Sample C

vsdatant.sys	watchdog.sys	zclient.exe	bcfilter.sys	bcftdi.sys
bc_hassh_f.sys	bc_ip_f.sys	bc_ngn.sys	bc_pat_f.sys	bc_prt_f.sys
bc_tdi_f.sys	filtnt.sys	sandbox.sys	mpfirewall.sys	msssrv.exe
mcshield.exe	fsbl.exe	avz.exe	avp.exe	avpm.exe
kavsvc.exe	klswd.exe	ccapp.exe	ccevtmgr.exe	ccpxysvc.exe
issvc.exe	rtvscan.exe	savscan.exe	bdss.exe	bdmcon.exe
cclaw.exe	fsav32.exe	fsm32.exe	gcasserv.exe	icmon.exe
nod32krn.exe	nod32ra.exe	pavfnsvr.exe	kav.exe	kavss.exe
inetupd.exe	livesrv.exe	iao.exe	Windows-KB890830-V1.32.exe	

Sample C performs SSDT hooking on the `NtQueryDirectoryFile` and `NtEnumerateValueKey` system calls to provide stealth. Furthermore, this sample calls the `PsSetLoadImageNotifyRoutine` to receive a callback whenever a process or driver image is loaded. The string analyzer reveals that this callback accesses a number of strings hardcoded in the rootkit image, that are shown in Table 9. These strings are clearly filenames, most of them related to antivirus software or other security tools. The most logical explanation for these observations is that the rootkit uses this technique to interfere with the loading and execution of anti-malware programs. Manual analysis confirms that the callback uses the `ZwTerminateProcess` function to kill these processes. We could not directly observe this behavior during analysis because none of the listed processes are executed in our analysis environment. This attack on security software highlights the need for a secure execution context for analysis software. The watchdog.sys file that is among those targeted by the rootkit is a driver that is also used by

CWSandbox [7], a malware analysis system that is not based on VMI but on in-the-box monitoring using a kernel driver.

5 Related Work

In this section we will discuss related research in the area of detection and analysis of malicious kernel code.

Integrity checking. In [14] the authors implement a tamper-resistant rootkit detector for Linux systems that uses VMI. To detect runtime patching, this system verifies the integrity of the kernel by hashing portions of clean memory considered critical and regularly comparing the hashes with their up-to-date counterparts. A limitation of this approach lies in the need to balance security with performance in selecting how often to perform hashing. In any case, such a system cannot guarantee that no injected code will ever be executed. To protect kernel code, an improved solution is offered by Nickle [15]. Nickle instruments the memory management unit of an emulator to redirect code fetches performed in kernel mode to a protected memory region. This way, it can detect code injected into the kernel as soon as its first instruction is executed.

Cross-view detection. Hiding an intruder's presence on a compromised system is a widespread goal of rootkits. This very behavior, however, can be exploited to detect kernel compromise. For this, cross-view detection approaches [12,28] compare system information obtained from a high-level abstract view, e.g. the Windows API, with information extracted from a lower level view, in order to reveal hiding. [14] uses a cross-view approach to detect process, kernel module and network port hiding. To detect process hiding, the authors of [29] use Antfarm [30] to determine implicit process information, without prior knowledge of the monitored guest's OS. They then perform cross-view comparison, detecting process hiding. While OS-independence is an attractive advantage of this approach, the technique employed cannot be easily extended to other types of stealth behavior. A drawback of cross-view is that it can detect the fact that something has been hidden, but cannot provide any information on how this has been done. Moreover, for information arranged in more complex data structures cross-view soon becomes impractical. For example, it can only detect file hiding if the contents of every directory on the system are compared.

Hooking detection. Hooking is another characteristic aspect of rootkit behavior that can be exploited for detection purposes. In [16], the authors present Hookmap, a system that can systematically discover possible hooking points in the execution path of system calls, enabling detection of rootkits that use these hooking points. Hookfinder [17] uses dynamic taint propagation to monitor the impact of a rootkit on the kernel. It detects a hook when a tainted value is loaded to the instruction pointer. In [18], the authors introduce k-tracer, a system that performs a sophisticated analysis of malware hooking behavior. For this, k-tracer first records an execution trace for a system call, reaching from the `sysenter` to the `sysexit` instruction. Then, an offline analysis is applied to the trace, by performing forward slicing to identify *read access* and backward slicing to reveal *manipulation* of sensitive data. This approach is however not compatible with our performance requirements for large-scale malware analysis, since k-tracer may require hours to analyze a single rootkit.

Rootkit analysis. Closely related to *d*Anubis is recent work on the dynamic analysis of rootkit behavior. In [31], the authors present rkprofiler, a system for the analysis of Windows kernel malware that is also based on VMI using Qemu. This system can reveal which system calls have had their execution paths modified to include injected code. Both [31] and [19] address the problem of understanding the semantics of rootkit modifications to dynamically allocated kernel memory. For this, they introduce techniques to recursively infer the type of an object in memory based on the type of the pointers that are used to access it, starting from the known structure of static kernel objects and function parameters.

6 Limitations

Our evaluation demonstrates that *d*Anubis can provide a substantial amount of information on malicious drivers. Nonetheless, our system suffers from a number of limitations.

Rootkit detection. To be able to analyze a rootkit's behavior, *d*Anubis must first detect the rootkit's presence in the analysed system. That is, it must be aware that extraneous code has been inserted into kernel space. For this, *d*Anubis relies on hooking system calls used for loading drivers. Therefore, we are unable to analyse rootkits injected through kernel or device driver exploits. This is a design choice, because it allows most *d*Anubis instrumentation to remain disabled until a driver is loaded, improving performance on the majority of analysed samples that *do not* load a driver. At the cost of some performance, this limitation could be addressed by integrating techniques from [15], that can reliably detect the execution of injected code. The detection of return-oriented rootkits [32], however, remains an open problem, since these rootkits do not inject any code into kernel space.

Dynamic analysis coverage. A general limitation of dynamic approaches to code analysis is that only code that is actually executed can be analyzed. In order to cover as many code paths as possible, we strive to stimulate typical rootkit functionality. Behavior that is triggered by benign user activity can be emulated to a certain extent by our stimulator. However, our large scale study has shown that many samples waits for commands to be issued from userspace through a device interface and never receive any such commands during analysis. The rootkit behavior associated with these commands is therefore not covered by our analysis. Related work in this field [19,18,31] does not specifically address this issue. Future research in rootkit analysis could attempt to design a stimulator capable of automatically issuing valid commands to malicious device drivers.

Related to the problem of coverage is the issue of detection of virtual environments and of analysis environments in general. If malware can detect our analysis environment it can thwart analyis by simply refusing to run. Unfortunately, implementing an undetectable virtual environment is infeasable in practice [33], although attackers may be reluctant to make their malware not function on widely deployed virtual environments. Defeating VM detection is largely a reactive, manual process. However, recent research [34,35] has shown that it may be possible to automatically detect previously unknown virtualization detection techniques.

Event attribution. In order to differentiate between legitimate and malicious actions, the origin of these actions has to be determined. To attribute a write access to a monitored driver we take the program counter of the instruction that carried out the manipulation and compare it with the codebase of the driver. While this technique works in practice, it can easily be fooled if the malicious driver uses a legitimate kernel function to manipulate the desired memory region. [36] introduces secure control attribution techniques based on taint tracking to tackle a similar problem in the context of (malicious) shared-memory browser extensions. Since Anubis provides tainting support, these techniques could also be adapted for integration in *d*Anubis.

7 Conclusions

The analysis of malicious code faces additional challenges when the code to be analyzed executes in kernel space. In this work, we discussed the design and implementation of *d*Anubis, a system for the dynamic analysis of Windows kernel malware. *d*Anubis can provide a comprehensive picture of a device driver's behavior and its interaction with the operating system, with other drivers and with userland processes.

We used *d*Anubis to conduct a large-scale study of kernel malware behavior that provides novel insight into current kernel-level threats. In this study, we analysed more than 400 recent rootkit samples to reveal the techniques employed to subvert the Windows kernel and, in most cases, the nefarious goals attained with these techniques. These results demonstrate that *d*Anubis can be an effective tool for security researchers and practitioners. We therefore plan to make it publicly available as part of the Anubis malware analysis service.

Acknowledgments

This work has been partially supported by the European Commission through project ICT-216026-WOMBAT funded under the 7th framework program.

References

1. Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., Kruegel, C.: Insights into current malware behavior. In: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET (2009)

2. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: 22nd Annual Computer Security Applications Conf., ACSAC (2006)
3. Kang, M.G., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables. In: ACM Workshop on Recurring malcode, WORM (2007)
4. Rolles, R.: Unpacking virtualization obfuscators. In: 3rd USENIX Workshop on Offensive Technologies, WOOT (2009)
5. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection. In: 23rd Annual Computer Security Applications Conference, ACSAC (2007)
6. Bayer, U.: Ttanalyze a tool for analyzing malware. Master's thesis, Vienna University of Technology (2005)
7. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security and Privacy 2(5) (2007)
8. Bailey, M., Oberheide, J., Andersen, J., Mao, Z., Jahanian, F., Nazario, J.: Automated Classification and Analysis of Internet Malware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 178–197. Springer, Heidelberg (2007)
9. Bayer, U., Milani Comparetti, P., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: Network and Distributed System Security Symposium, NDSS (2009)
10. Rieck, K., Holz, T., Willems, C., Duessel, P., Laskov, P.: Learning and classification of malware behavior. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 108–125. Springer, Heidelberg (2008)
11. Jacob, G., Debar, H., Filiol, E.: Malware behavioral detection by attribute-automata using abstraction from platform and language. In: Recent Advances in Intrusion Detection, RAID (2009)
12. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Network and Distributed Systems Security Symposium, NDSS (2003)
13. Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure in-vm monitoring using hardware virtualization. In: ACM conference on Computer and communications security, CCS (2009)
14. Quynh, N.A., Takefuji, Y.: Towards a tamper-resistant kernel rootkit detector. In: SAC 2007: Proceedings of the 2007 ACM symposium on Applied computing, pp. 276–283. ACM, New York (2007)
15. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
16. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 21–38. Springer, Heidelberg (2008)
17. Yin, H., Liang, Z., Song, D.: Hookfinder: Identifying and understanding malware hooking behaviors. In: Network and Distributed Systems Security Symposium, NDSS (2008)
18. Lanzi, A., Sharif, M., Lee, W.: K-tracer: A system for extracting kernel malware behavior. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (2009)
19. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: EuroSys 2009: Proceedings of the 4th ACM European conference on Computer systems, pp. 47–60. ACM, New York (2009)

20. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology 2(1), 67–77 (2006)
21. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, Reading (2005)
22. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007)
23. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the annual conference on USENIX Annual Technical Conference, p. 41. USENIX Association (2005)
24. Orwick, P., Smith, G.: Developing Drivers with the Microsoft Windows Driver Foundation. Microsoft Press, Redmond (2007)
25. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Network and Distributed Systems Security Symposium, NDSS (2005)
26. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Network and Distributed System Security, NDSS (2008)
27. Russinovich, M.: Filemon (2010), http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx
28. Beck, D., Vo, B., Verbowski, C.: Detecting stealth software with strider ghostbuster. In: Proceedings of the 2005 International Conference on Dependable Systems and Networks, pp. 368–377 (2005)
29. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Vmm-based hidden process detection and identification using lycosid. In: VEE 2008: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pp. 91–100. ACM, New York (2007)
30. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: tracking processes in a virtual machine environment. In: ATEC 2006: Proceedings of the annual conference on USENIX 2006 Annual Technical Conference (2006)
31. Xuan, C., Copeland, J., Beyah, R.: Toward revealing kernel malware behavior in virtual execution environments. In: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (2009)
32. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th USENIX Security Symposium (2009)
33. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: Vmm detection myths and realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (2007)
34. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In: USENIX Workshop on Offensive Technologies, WOOT (2009)
35. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient detection of split personalities in malware. In: Network and Distributed System Security, NDSS (2010)
36. Saxena, P., Sekar, R., Iyer, M.R., Puranik, V.: A practical technique for containment of untrusted plug-ins. Technical Report SECLAB08-01, Stony Brook University (2008)

Evaluating Bluetooth as a Medium for Botnet Command and Control

Kapil Singh, Samrit Sangal, Nehil Jain, Patrick Traynor, and Wenke Lee

School of Computer Science, Georgia Institute of Technology
{ksingh,samrit,nehjain,traynor,wenke}@cc.gatech.edu

Abstract. Malware targeting mobile phones is being studied with increasing interest by the research community. While such attention has previously focused on viruses and worms, many of which use near-field communications in order to propagate, none have investigated whether more complex malware such as botnets can effectively operate in this environment. In this paper, we investigate the challenges of constructing and maintaining mobile phone-based botnets communicating nearly exclusively via Bluetooth. Through extensive large-scale simulation based on publicly available Bluetooth traces, we demonstrate that such a malicious infrastructure is possible in many areas due to the largely repetitive nature of human daily routines. In particular, we demonstrate that command and control messages can propagate to approximately 2/3 of infected nodes within 24 hours of being issued by the botmaster. We then explore how traditional defense mechanisms can be modified to take advantage of the same information to more effectively mitigate such systems. In so doing, we demonstrate that mobile phone-based botnets are a realistic threat and that defensive strategies should be modified to consider them.

1 Introduction

Mobile phones are being increasingly tasked with sophisticated duties. From trading on financial markets and mobile banking to carrying medical records [29], these devices are beginning to be trusted with some of our most sensitive information. Unfortunately, because the majority of mobile phones lack even basic security mechanisms (e.g., memory protection), they are becoming increasingly attractive targets for malware writers. The widespread usage of such devices and the sensitivity of cellular networks to even small amounts of malicious traffic make this actuality a significant threat.

A wide range of malware targeting mobile phones has already been documented. Whether arriving via MMS [22], a downloaded executable [35] or over a Bluetooth link [14], both viruses and worms are being extensively explored in this environment. Botnets, however, have not yet been studied in depth in this setting. Representing one of the most significant threats to the Internet, mobile botnets could use compromised phones to execute regularly updated mission requests (e.g., Denial of Service, premium number dialing, password/credential theft, etc). Like their Internet-based counterparts, mobile botnets can only achieve such flexibility given the presence of a robust *command and control* (C&C) infrastructure. Unlike traditional botnets however, we argue that such an infrastructure can be successfully maintained outside of the purview of cellular providers, making detection and mitigation challenging given current strategies.

In this paper, we evaluate the potential for mobile phone-based botnets to communicate and coordinate predominantly via Bluetooth. Unlike previous work that investigates whether malware can spread over such links, we instead investigate *whether a command and control infrastructure can be maintained in an environment with almost entirely transient links.* We develop an understanding of the long term interaction of infected devices through the use of two large-scale datasets and, through simulation, demonstrate that the repetitive nature of the daily routines of human beings allows messages to be propagated to over 66% of infected nodes within a day. We then compare the impact of varying parameters including device popularity and polling interval to allow a botmaster to tradeoff the speed of propagation with their ability to remain hidden from a network provider. Finally, we conduct large-scale simulations to attempt to better model the dynamics of such botnets in a realistic setting - public transit.

In so doing, we make the following contributions:

– **Develop the first characterization of Bluetooth-based C&C for mobile devices:** Using publicly available data on mobile device interaction, we develop the first characterization of command and control operations in this setting. In particular, we show that mobile botnets are possible, but that instruction propagation latency can be significant. In exchange for such latency, botmasters are able to notably reduce the amount of traffic observable by the provider.
– **Create a new C&C architecture based on node popularity:** We develop a framework in which bots selectively communicate with the botmaster based on their popularity. In particular, only a small subset of bots with the highest degree ever speak directly to the botmaster. This mechanism helps to improve the speed of propagation without exposing all infected nodes to a network provider.
– **Develop countermeasures leveraging communication patterns:** From the information learned above, we develop patching and mitigation strategies that significantly reduce a mobile botnet's ability to defend against our countermeasures and remain hidden.

Note that traditional botnet C&C infrastructures are likely to be easily detected in these networks as providers are more likely to have a more complete global view.[1] This means that bots are unlikely to be successful in these networks unless they adopt a strategy similar to the one presented in this work.

The remainder of this paper is organized as follows: Section 2 provides an overview related work in the area of mobile malware; Section 3 discusses how mobile phones can and are likely to be infected and explores a number of possible mechanisms for command and control; Section 4 explains our data and simulator that we use to model Bluetooth-based botnets; Section 5 simulates such networks using well-known public data sets; Section 6 models a large-scale botnet in a public-transportation setting; Section 7 discusses how the above observations can be leveraged to combat such botnets; Section 8 offers concluding remarks.

[1] This is also true because associating messages with their origin given that each device authenticates to the network.

2 Related Work

Botnets [26,10] represent the major source of malicious activity on the Internet – they send spam [27], perform DDoS attacks [16] and host phishing web sites [7]. Significant attempts have been made by the research community to both categorize [10, 6] and mitigate [19,27,18,17] such threats. Unfortunately, understanding such threats in the context of cellular networks is still very limited [33,34].

The transformation of mobile devices from simple voice terminals into highly-capable, general purpose computing platforms makes the possibility of attacks originating from within the network a reality. The tremendous increase of cell phone adoption and the lack of widely implemented security mechanisms makes such platforms attractive targets to botmasters. Research has previously shown cellular infrastructure as a potential target of botnet attacks [33]; however, such botnets are composed entirely of compromised machines across the Internet. Previous work has not considered whether or not such a malicious overlay can be created and maintained exclusively on mobile phones.

Bluetooth-based malware has been extensively explored. Su et. al highlighted the presence of a diverse set of known security vulnerabilities in the Bluetooth protocol's implementation [31]. They argued that the presence of such vulnerabilities coupled with the complexity of the Bluetooth specification and its large codebase will likely lead to more complex attacks using the Bluetooth channel. Worms including Cabir [8], Mabir [9] and CommWarrior [21] have already successfully exploited this channel. Previous efforts to model the *propagation* behavior of Bluetooth-based malware have focused entirely on the analysis of Bluetooth worms. Yan et. al studied the effect of mobility on worm propagation by restricting the devices in an area with sides of length 150 meters [39,37]. The same authors later provided a comprehensive analytical model for such Bluetooth worm propagation [38]. Other studies have investigated the effect of population characteristics and device behavior on the outbreak dynamics of Bluetooth worms [28,23]. Such characteristics have been previously exploited for peer-to-peer (P2P) content distribution [20] and for studying human social behavior [12].

Mobile phone-based botnets using Bluetooth to propagate control messages bear a striking resemblance to Internet-based P2P botnets [13, 5]. In particular, even if defenders identify a subset of the bots in a botnet, communication among the remaining bots will not be disrupted. Second, in contrast to other centralized approaches (such as IRC [6]), there is no fixed endpoint from which the botmaster must transmit commands. For Bluetooth botnets, the botmaster can send messages from any Bluetooth-enabled device, and he can frequently change these source devices to evade detection. Bluetooth communication has other additional, attractive properties that benefits botnet creators. Unlike the Internet-based P2P channels, Bluetooth by principle is proximity based: this gives the defenders limited scope to observe the communication between two bot devices. Additionally, P2P botnets suffer from the problem of losing bots whenever those bots change their dynamic IP addresses. Bluetooth channels are resilient to such changes.

3 Bluetooth-Based Botnets

Our goal is to evaluate the suitability of Bluetooth as a command and control channel. In particular, we focus on the challenges facing a botmaster trying to coordinate a large number of infected devices over a transient, near-field communication channel that cannot be easily identified or blocked by cellular providers. In this section, we discuss how mobile phones can be compromised, detail our threat model and assumptions, and discuss the C&C architecture of mobile phone-based botnets.

3.1 Infecting Devices

Mobile devices have rapidly transformed from limited embedded systems to highly capable general purpose computing platforms. While such devices have long enjoyed significant diversity in hardware and operating systems, the rising popularity of smartphones and the ability to sell applications to users is leading to the establishment of standardized mobile software platforms and operating systems, such as Microsoft's Windows Mobile, Google's Android and Apple's Mobile OS X. Unfortunately, many devices are only now beginning to implement basic security mechanisms including memory protection and separation of privilege. Accordingly, such systems are expected to be increasingly targeted by malware. Malware targeting mobile devices may come from any of a number of sources. Given that 10% of cellular users downloaded games to their mobile devices at least once a month in 2007 [24] and the wide availability of free ringtones, downloadable content and executables are one of the more likely origins. Like their desktop counterparts, mobile devices are also likely to be susceptible to a range of browser exploits including drive-by downloads [25]. Finally, the presence of multiple communications interfaces makes mobile devices susceptible to malware that propagates not only through the cellular network itself [15], but also potentially through WiFi and Bluetooth [8,9,21]. Accordingly, the breadth of infection vectors exceeds that of many traditional networked systems.

3.2 Threat Model and Reasoning

Given the above, we assume that bots have already been installed on a subset of mobile phones within a network but that the C&C infrastructure remains unestablished.

We expect that powerful defenses exist to detect and disrupt botnets. In our threat model, we assume that defenders have access to malicious binaries and are able to learn the bot's entire execution behavior through forensic analysis techniques. This allows defenders to identify the command list and the algorithms used by the bot to extract commands from Bluetooth messages. Simply stated: the threat model allows defenders to know everything that is known to the bot.

Botmasters logically aim to sustain their networks for as long as possible. Stealth is therefore one of the most critical characteristics of such a botnet. The short range and ad-hoc nature of communications via Bluetooth potentially provides such an opportunity: defenders need to be *within range* of the communicating infected devices *at the time of communication*, which might not be practical considering the changing network topology of the ad-hoc network. Given that providers are not able to observe the vast majority of messages between infected devices, this means of communication appears attractive.

Reliance upon near-field communications also makes commanding bots more challenging. Specifically, a botmaster must rely on infected nodes being within range of

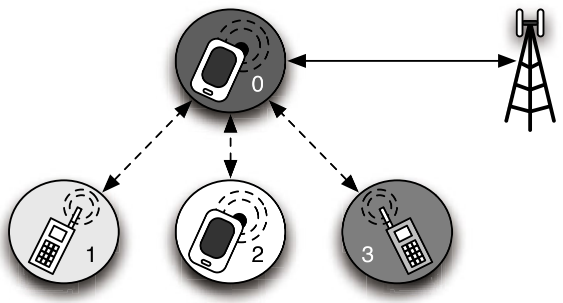


Fig. 1. Only nodes with the highest exposure to other bots (shown here by the darkness of the circle) contact the botmaster directly. All other nodes receive commands and updates when they are within Bluetooth range of an updated node.

each other on a regular basis in order to successfully propagate commands. We leverage the fact that a large portion of the population follows regular patterns of behavior. For instance, person A gets up every day at 7 am and goes to his office using the subway. In the process, he will encounter a large subset of the same people every day (who also take the train at roughly the same time everyday). He will also interact with a large subset of the same people everyday in the office. The highly regular nature of this routine provides structure in an unstructured environment. Specifically, while the botmaster may not know the exact topology of the network at any particular moment, he or she will know with some assurance the subset of devices with which a user is likely to interact with during a period of time.

3.3 Botnet Construction and Message Passing

We now explore the construction and operation of mobile phone-based botnets based on our threat model. While traditional strategies such as having all nodes use cellular data connections to reach Internet-based centralized or P2P designs are possible, large volumes of such communication are easily detectable by a provider. Accordingly, we aim to develop a mechanism that avoids such detection while overcoming a number of challenges. For instance, the botmaster must be able to learn the identities of all of the phones under his control so as to be able to accurately portray the size of the botnet when renting it. This task must be accomplished without these devices contacting the botmaster directly. Additionally, the botmaster needs to ensure that he can contact the largest possible number of infected phones within a short period of time. Unfortunately, a Bluetooth-only solution is unlikely to be sufficient in this context. In particular, an adversary would need to physically place himself near as many nodes as possible, which is more likely to be a burdensome task.

We instead propose a hybrid approach designed to maximize the speed of distribution while only minimally reducing stealth. In particular, we allow a botmaster to communicate with a *very small* number of infected nodes through cellular channels (e.g., SMS, cellular data). These nodes are selected via their relative frequency of contact with other infected devices. Specifically, as infected devices pass within range of each other, they record the identity of the other device. After reaching some threshold set by the botmaster, those nodes with a high degree of connectivity over time contact the botmaster and provide their contact log. In so doing, these devices not only provide the botmaster with knowledge of the devices under his control, but also inform him of which nodes are most likely to be able to help rapidly disseminate commands.

Figure 1 represents one such typical scenario, where the darkness of the circle around the phone reflects the popularity of that device. As we can determine from the figure, phone 0 is the most popular and therefore act the seed for communication with the botmaster. If phone 0 is disinfected, phone 3 would likely report back to the botmaster (potentially after the expiration of some long-term timeout value).

The botmaster also disseminates commands through this hierarchical structure. When a new task arises, the botmaster simply contacts the seed nodes in a particular area and provides an updated mission/payload to be distributed to other infected nodes. These nodes are most likely to be able to deliver such a payload to the largest possible number of infected nodes without requiring them to directly interact with the botmaster because of their high degree of connectivity over time.

Seed nodes logically present attractive targets to defenders. Those nodes reporting back to some centralized point are more likely to be singled out by the provider. Naïvely constructed, such a strategy would allow a provider to cripple the communications of these systems. If the very low volume of traffic is not sufficient for avoiding detection, such bots can further obfuscate their activities through a number of anonymizing techniques ranging from Tor [11] and Publius [36] to the use of a free temporary email address. Such communication could be further obscured through the use of the WiFi connection available to many smart phones. Communications from the botmaster can avoid detection by spoofing the source address of a communication from the Internet, including text messages claiming to be from within a target node's community of interest [33].

4 Experimental Setup

Given a proposed communications architecture and our threat model, we now seek to determine whether or not mobile phone-based botnets can effectively communicate using Bluetooth. In this section, we discuss a number of details related to our experimental design and testing. This infrastructure is used throughout the remainder of the paper.

4.1 Prototype Bot

Rather than running a simulation with nodes simply exchanging meaningless messages, we implemented a proof of concept mobile phone-based bot. Our prototype bot is coded in Java and deployed on the Sun Wireless Toolkit that emulates infected mobile devices. Each bot instance acts as a peer in the bot network, listening for new commands and simultaneously sending commands to other discovered bots. At the initial infection stage, the bot registers itself with a Universally Unique Identifier (UUID) in the service register present in the mobile device, thus allowing it to be discovered by other bots. It then waits for new incoming connections. A two-way Bluetooth connection is established with other bots when they come within range. As part of our protocol for information exchange among bots, the bot is updated with the latest version of the command, which also includes the updated parameters of the command.

As a proof-of-concept, we implemented a botnet command that directs bots to send an SMS to a specified mobile number without being noticed by the sender. Such a command can launch a denial-of-service attack on a targeted area [33] should enough devices participate. Because most service providers also charge for incoming messages, such attacks can also incur substantial costs for the targeted victim. These initial experiments demonstrate that Bluetooth can successfully be used to pass commands between infected nodes with relative ease.

4.2 Experimental Goals

We use our proof-of-concept bot to evaluate various parameters that directly or indirectly impact command propagation in a mobile phone-based botnet. While some of these parameters, such as the device polling interval, can be controlled by the botmaster, other parameters are influenced by the inherent characteristics of near-field communications and the human movement patterns. We enumerate these parameters and evaluate their impact using a range of simulations that model a range of settings and scenarios.

Our simulations are categorized into two experimental sets. The first set of simulations are performed based on two trace logs, one each from MIT [12] and NUS [30], that are collected using real devices carried by individuals for their day-to-day activities. By means of these experiments, we evaluate various operational parameters of the proposed botnet C&C mechanism, which are useful in determining the correctness of our hypothesis.

In our second set of experiments, we use publicly-available information to create a large scale simulation model of New York City's subway system and thereafter, use this model to demonstrate command propagation in a typical scenario of public transportation. The goal of these simulations is to expand our evaluation beyond the boundaries imposed by the limited traces, and demonstrate the viability of a large-scale mobile phone-based botnet.

In order to run the simulation, one initial node is selected and subsequently all the nodes encountered by the selected node are enumerated in time stamp order. If the period of contact shown by the traces is equal to or more than 5 minutes (this is the least time granularity for the MIT data set), we assume that the other node successfully received the command. This is a conservative assumption given that botnet commands are generally negligible in size (on the order of tens of bytes) and Bluetooth can transfer data at a rate of approximately 1Mbps. In our experiments, we examine various factors and environmental variables that have an effect on the latency of the command propagation in the proposed Bluetooth-based botnet.

We do not consider device heterogeneity in our experiments because we view infection and command propagation as two separate tasks. In particular, because devices can be infected through any number of different vectors (e.g., drive-by downloads, malicious executables, browser bugs, etc), phones of all varieties can be forced to run

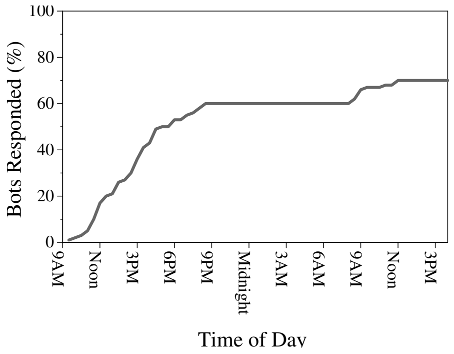


Fig. 2. Command Propagation Rate

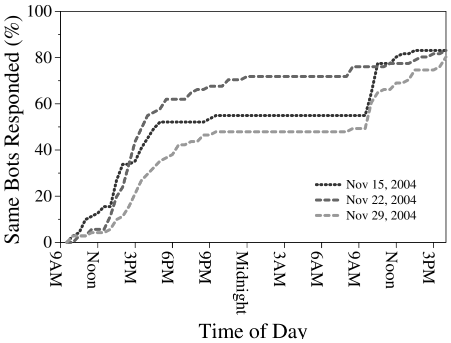


Fig. 3. Recurrent bot connections over different days

bot software. Our evaluation focuses not on how infection happens but how messages spread after infection has occurred.

5 Trace Based Simulations

We use the architecture and assumptions detailed in the previous two sections to develop simulations of mobile phone-based botnets. In this section, we use publicly available Bluetooth traces to perform a number of simulations. These experiments characterize the impact of various factors on effective botnet communication.

5.1 Description of Datasets

Reality Mining Dataset (MIT). The Reality Mining project is a collection of environmental data gathered by one hundred mobile phones over a course of six months. Polling by each of these phones was used to determine the presence and identity of other Bluetooth-capable devices in their proximity. This data was initially used to provide insight into the dynamics of the social behavior of both individuals and groups [12]. While one of the more extensive available datasets, the major limitation of this study is that the time between subsequent polling for devices is five minutes and as a result, our evaluations miss interaction of devices that were in contact for less time. The proposed message dissemination techniques are therefore likely to perform even better than the results we present. We attempt to overcome this limitation with a large-scale simulation in Section 6.

Bluetooth Dataset (NUS). We use a second collection of logs known as the the National University of Singapore (NUS) Bluetooth [30] dataset that contains traces of Bluetooth sightings by 12 devices over a period of 7 months. Each device polled for other devices every 30 seconds and recorded device identifiers of all the Bluetooth enabled devices in its range, providing significantly finer granularity of interaction than the MIT dataset. Out of the 12 experimental devices, 3 were static devices placed near lecture halls on the NUS campus and the rest were given to the faculty and students. This dataset makes it possible for us to evaluate the effect of varying polling interval on the characteristics of the Bluetooth-based botnet, and also demonstrate how mobility of the commanding device can influence the command propagation rate in the botnet.

5.2 Simulation Results

Command Propagation Rate. Figure 2 provides a single but representative view of command dissemination by the most popular node. Note that the command is rapidly dispersed during the morning hours, plateaus in the evening and the increases again slightly the next morning. Note that because of the regularity of regularity of human behavior, the increase on the second day is relatively low given that few different nodes are observed between any two given days. Our experiments demonstrate that messages are consistently delivered to greater than 2/3 of all infected nodes within 24 hours of the botmaster contacting the seed node.

We define *Probability of Delivery (PoD)* as the percentage of bots responding within a predefined time period. The desired time represents the maximum acceptable latency for a botmaster to successfully distribute a command to some portion of the nodes under his control. Given Figure 2, a realistic value for such a response time is approximately 24 hours; however, the nature of the command may offer the botmaster additional flexibility. For instance, the botmaster may not require that a spam campaign be coordinated, allowing nodes to being their job immediately after receiving the command. Alternatively, denial of service attacks are likely to require greater coordination, meaning that such attacks may need to be planned further in advance in order to be successful. When sent at optimal times (i.e, the morning), PoD values rise relatively quickly, with 40% of nodes having received commands within five hours (Figure 2). Mission and command launch time must therefore be carefully considered by the botmaster.

Note that our experiments assume the use of only a single seed node. A botmaster could potentially seed multiple nodes with commands in order to compensate for known geographic barriers or increase the speed of command dissemination. However, such a strategy must be carefully balanced against the botnet's need for stealth.

Long-Term Communications. The results thus far demonstrate that Bluetooth-based mobile botnets are capable of passing commands on a single day. However, the ability to re-establish connections across long periods of time is necessary for such a network to be worth the effort to construct. For this analysis, we take the traces from November 1, 2004 as the initial list of infected bots. We then evaluate how many of these infected nodes come in contact with a command-carrying node over subsequent days. The same node is chosen as the initial node for iterations repeated over different days. Note that the experiments were conducted over the closed set of 100 nodes in the MIT data set; other datasets show similar behavior (Section 6).

Figure 3 shows the percentage of infected nodes that repeatedly come in contact with a command-carrying node over different days. In other words, it represents the number of bot nodes that would successfully receive a botnet command. Our results show that bots tend to come in contact with a large subset of the same nodes repeatedly over

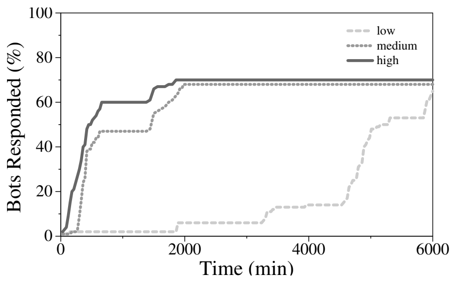


Fig. 4. Effect of the node popularity on command propagation

	30	50	70
High Popularity Node	5hrs	8hrs	13hrs
Medium Popularity Node	5.5hrs	25hrs	31hrs
Low Popularity Node	80hrs	86hrs	102hrs

Fig. 5. Propagation Times for various nodes

different days. This number is on an average about 50% within the first 8 hours of a day's schedule and goes up to more than 80% after a day.

As seen in Figure 3, the pattern does not vary significantly across different days, which shows that a command issued at a particular node will follow a generally predictable spread for that node on any given day. While we observed a maximum variance of about 18% at any particular time, eventually this variance becomes negligible with more than 80% of the same infected nodes are consistently seen on any given day.

Device Popularity. Figure 4 shows the command propagation rate for seed nodes of high, medium and low popularity. In order to determine the popularity of the nodes, we sort all nodes based on the number of contacts they have with other nodes. High and low popular nodes have the maximum and minimum contacts respectively; node with medium popularity is chosen as the median of the sorted data set. The patterns in this figure are representative of all days.

The popularity of a seed node intuitively has an effect on the time it takes for the command to propagate. However, this difference is negligible for some cases for nodes of medium and high popularity. Nodes with very low popularity take significantly longer to reach a desired PoD but eventually exhibit the same saturation. A likely explanation for this behavior is that such nodes eventually encounter more popular nodes, thereby increasing the rate of message dissemination. Accordingly, regardless of which node is selected, commands are eventually propagated. This results may not always hold true for larger datasets, which may have much more significant outliers than the MIT and NUS datasets; however, most nodes will encounter a large enough number of other nodes that message delivery can occur within a reasonable timeframe.

Figure 5 provides numerical results corresponding to the range of popular nodes. Note that while reaching 70% requires a significant amount of time in all but the most popular case, such widespread distribution may not be required. In particular, given a large number of nodes, a botmaster may only want to dedicate a subset of their botnet to a specific task. Accordingly, a botmaster may wish to select nodes of differing popularity to best meet their mission.

Polling Interval. We define *polling interval* as the time between two consecutive polls conducted by a Bluetooth device looking for other devices in its proximity. Figures 6

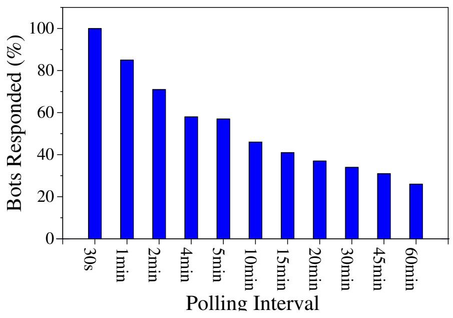


Fig. 6. Effect of varying polling intervals on static seed nodes

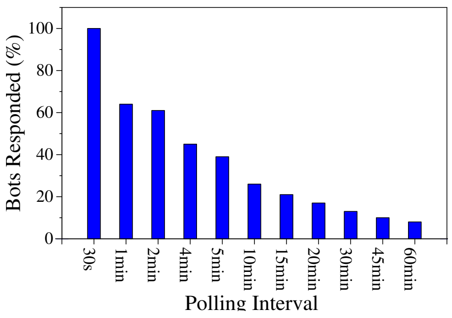


Fig. 7. Effect of varying polling intervals on mobile seed nodes

and 7 show the effect of the variation in polling interval on the PoD. The experiment is performed on the NUS data set, which has a base polling interval of 30 seconds, as the polling granularity of the MIT dataset is too coarse to offer interesting results. The results are presented with polling interval of 30 seconds as the base reference; all other results are shown relative to this case. The botmaster is able to control the polling interval at each bot. A low polling interval would diminish stealthiness as the bot may become visible to the user of the victim devices: more frequently probing by the device has an adverse effect on the battery life of the device. This observable behavior might alert the user about the presence of a malicious bot on the device, thus exposing the botnet. On the other hand, increasing the polling interval improves the stealth of the botnet, but reduces the number of devices the botmaster can spread his command to in the desired time.

We repeated the experiment both for the static nodes placed at strategic points on the NUS campus and for the mobile nodes that were free to roam around. As shown by the graphs in Figure 6 and 7, there is a clear decrease in the PoD with increasing polling interval. This result is expected because with longer polling interval, there is greater possibility of missing devices that come in the proximity of the polling device at a time between two subsequent polls when the device is not polling. Moreover, the percentage drop in the PoD is less for static nodes as compared to the mobile nodes: both static and mobile nodes show the highest PoD for the base case with polling interval of 30 seconds; for polling interval of 1 minute, this goes down to 85% for static nodes and much lower 65% for mobile nodes. One possible reasoning behind a much lower drop for the case of static nodes is that people traverse popular spots multiple times and spend relatively longer time at such locations, hence there is a stronger possibility of a successful command transfer to the corresponding devices carried by these people. However, for mobile nodes, since even the polling device is in motion, this phenomenon of repeatedly coming in contact may not occur.

The graphs also show that even with a higher polling interval, a considerable percentage of the devices can still receive the botnet command. For example, even for a polling interval as high as 60 minutes, the PoD values are still significant – 26% for static nodes and 9% for dynamic nodes.

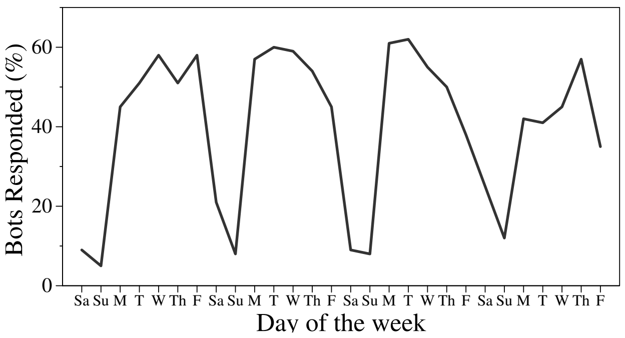


Fig. 8. Weekday Effect: Static nodes

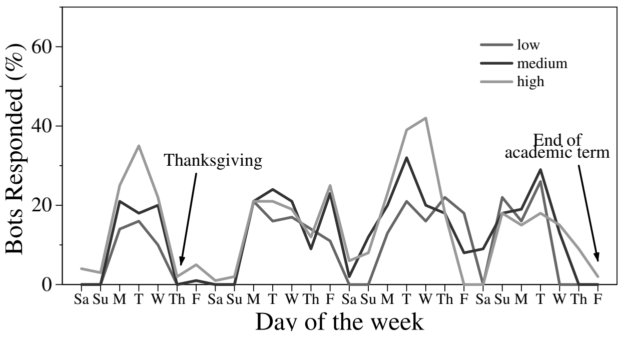


Fig. 9. Weekday Effect: Dynamic nodes

In essence, by choosing the polling interval for his bot victims and carefully placing Bluetooth devices at strategic locations, the botmaster can achieve a balance between the PoD and stealthiness for his desired use of the botnet. Additionally, the botmaster can direct his bots to use *adaptive* polling to attain such a balance: a bot would aggressively search for other devices by lower its polling interval when the bot device is in motion (possibly during morning and evening hours) and would switch to a higher polling interval at other times of the day when the device is static.

Weekday Patterns. Humans follow daily and weekly patterns that also greatly influence the command propagation rate. Figure 8 presents the number of daily encounters broken down by the day of the week when they occur. This graph confirms the intuition that more encounters occur on week days than on weekend days. We observed similar behavior for both MIT and NUS data sets, and also for both static and mobile seed nodes (Figure 8 and 9). Additionally, such behavior is independent of the popularity of the node (Figure 9).

These observations support our argument that repetitive nature of human routines can be leveraged by mobile phone-based botnets. With a better understanding of such weekday patterns of the targeted devices, the botmaster can be more effective in propagating the commands faster. For example, it would be beneficial to issue a command on a weekday rather than a weekend. Special considerations should also be made for days such as Thanksgiving and the end of an academic term (Figure 9)[2].

Each set of contacted nodes generated for different weeks for one particular node has 70% of the nodes as common in all result sets. That means around 50 nodes were contacted every time the command was issued to a particular node, out of the 70 nodes contacted each time. Out of the remaining 30%, most of the nodes were present in more than one result set but not in all. This shows that there is low weekly fluctuations and that a botmaster can control most of the botnet effectively over any given week.

6 Modeling the Public Transport System

In this section, we perform a large-scale simulation to demonstrate the viability of a mobile phone-based botnet in a larger real-world setting. Our tests demonstrate that communication in such a botnet maintaining the previously discussed characteristics even in a large-scale environment and provides the botmaster with reasonable tradeoffs between botnet response time and stealthiness.

We simulate the rush hour period on a typical weekday at New York City's Grand Central Terminal [1]. Grand Central is one of the busiest stations in the city – it not only serves the second busiest subway station in the city, but also serves the Metro North trains from upstate New York. Consequently, the subway station receives passenger traffic both from people coming to NYC using the Metro North trains and from local commuters. We use this setting to simulate bots that reside on the cellular phones of individuals traversing the station as per their daily routines.

We use publicly available data sources for our simulations to model the station and to estimate the mobility patterns of the commuters. We also use probabilistic distributions for various parameters to allow variation in the individual behavior of bots. We acknowledge that accurately predicting patterns in human movement is difficult, however, approximations can be made by carefully analyzing different sources of publicly available data and statistics. When certain statistics are not available, we make conservative estimates.

A person arriving at Grand Central Terminal will typically have the following movement pattern: he or she arrives at Grand Central either by taking a Metro North train or by entering the station through one of the entrances. He or she traverses the station to reach the desired subway platform and then waits for a random time at the platform before the train arrives. He or she boards one of the train cars and therein remains in a static position till the train reaches his desired destination.

6.1 Simulation Setup

Train Station: Grand Central Terminal is modeled as a square, with entrances and train tracks placed along its edges. The size of the station and the number of entrances used in our model are the same as those existing in the real structure [1]. We uniformly place entrances along two sides of the square. Metro North tracks (44 in total) are placed

[2] Verified against MIT's 2004 academic calendar.

along the third side, and the fourth side has the 3 subway tracks. This design is a close approximation of the architecture of the actual terminal [1].

Train Cars: We again used publicly available information about the size of New York City's transit trains to model the trains in our simulation [3]. The number of cars are fixed and identical for all trains arriving at the subway station. While boarding a train, a person chooses one car at random. For simplicity, we assume that the commands propagate from one device to another only within one car. The train arrival times are simulated according to the known subway time schedule.

Commuter Traffic Estimates: The Metro North trains at Grand Central serve approximately 125K commuters per day [1]; approximately 150K for the subway station [2]. We assume that Metro North passengers constitute about 50% of the subway commuters. We also assume that 50% of the daily commuters take the subway during rush hours. The arrival of commuters at Grand Central is distributed over the complete rush time interval (6:30AM–9:30AM) based on a gaussian distribution with mean at 8:00AM and variance of 33%.

Phone Infections: We only consider devices that have been previously infected and a currently carried by the commuters in the terminal during these simulations. In order to estimate the number of bot victims, we consider some known statistics on cellular phones usage: about 80% of commuters in New York City carry phones according to a MNRR survey [4]. We assume that approximately 30% of these phones can be (and are) infected. We believe that this is a safe assumption based on our earlier premise that mobile software platforms and operating systems are being standardized without adequate focus on security, leaving such mobile devices vulnerable to malware infections.

Command Transfer: The time required to transfer a command from one bot-infected device to another via the Bluetooth channel is the total time taken for the four stages of the protocol, namely inquiry, connection establishment, probing and command transfer. The corresponding timeouts for the four stages are set to 10.24 (from Bluetooth specification), 5.12 (from Bluetooth specification), 0.1 and 1 seconds, respectively. We model the time for all four stages as a gaussian distribution with mean set to half of the corresponding timeout value.

Simulating Human Movement: We use a modified form of Random Landmark Model [39] to simulate the movement pattern of humans. In our simulation, the initial starting position is either one of the entrances of the train station or any track of the Metro North, and the destination is set to one of the subway tracks. These starting and ending points are randomly chosen for each individual. The speed of movement is fixed at typical walking speeds of 1, 2 and 3 meters/s. After arriving at a particular train station, a person waits at a fixed point before boarding the next arriving train. Once the entry time of an individual is determined during the initial cycle of our simulation (a cycle represents a rush hour period on any given day), we use gaussian distribution to calculate the entry time for all subsequent cycles. We limit the distribution to within 10 minutes of the entry time of the initial cycle. It effectively represents a regular daily commute for an individual, who boards a train almost at the same time every day.

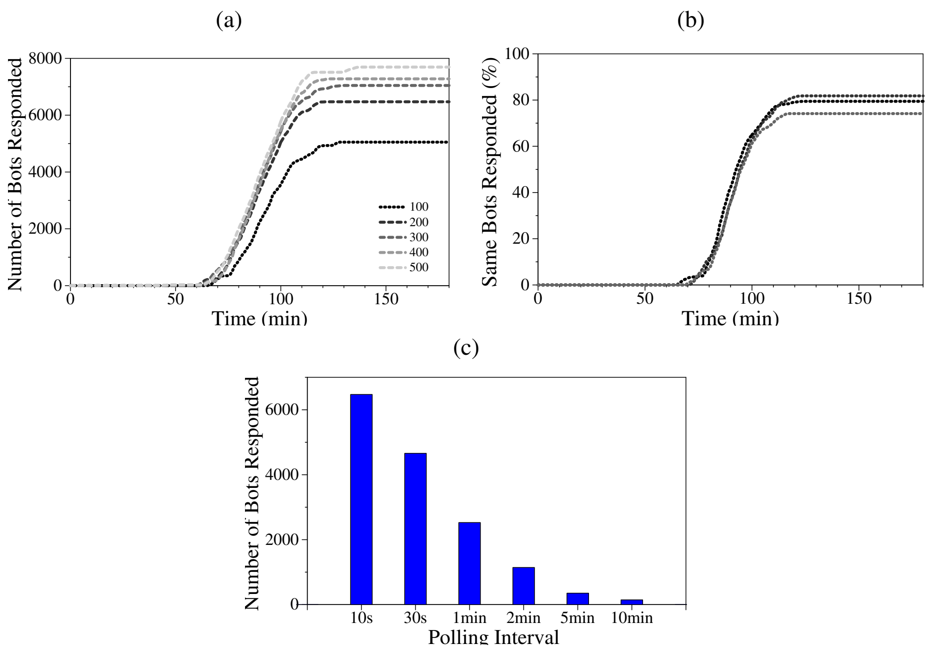


Fig. 10. Simulation results for the transport model: (a) Effect of the initial number of seeds on the number of bots receiving the command, (b) number of bot encountered repeatedly over multiple runs simulating different days, and (c) effect of varying polling interval on command propagation rate. During the rush hour commute, 30% of the total population or around 17K nodes are vulnerable. In (a), the initial seeds are chosen using the same Gaussian distribution used for the station entry time of nodes. These initial seeds are not counted in the results.

6.2 Simulation Results

Our results show that public transportation can act as an ideal environment for a botmaster to effectively pass commands to a large bot population. With a relatively small initial seed of command-carrying nodes, the botnet commands can reach a considerable number of bots (Figure 10a). Such seeds can be created by the botmaster using other alternate mechanisms, for example, by planting static nodes at popular spots near the station or by dropping commands at the nodes traveling to Grand Central in the Metro North trains. We observe that more seeds allow commands to propagate faster, although the propagation rate speedup is modest beyond a threshold. This suggests that the botmaster can achieve considerable coverage even when the number of seeding bots is low: for a initial seed of 200, the command can be transferred to about 6500 nodes within the rush hour, which forms more than 35% of the population that can be infected. Note that these numbers correspond to only nodes that receive commands at Grand Central Terminal; these nodes will further propagate the commands to other nodes when they visit their work, school, homes, etc. Such results are out of scope for this simulation.

Our transport simulation model reinforces our premise that humans follow routines in their day-to-day movements, which can be exploited for botnet C&C. Figure 10b shows that a command propagation cycle typically encounters more than 70% of the same nodes every day at rush hour. These numbers would be much higher in a more closed setting like offices where the movement of employees is typical more restricted and as a result, the Bluetooth devices come in constant contact with each other for longer periods of time.

Our results also demonstrate that by carefully specifying the polling interval for the bots, a botmaster can balance the latency experienced by the botnet and stealthiness of the bots (Figure 10c). Keeping the interval to a lower value results in higher propagation rates: for a polling interval of 10 seconds, the command propagates to about 6500 bots, which constitutes about 36% of the total vulnerable machines. With an increased polling interval, the propagation rate drops substantially with only 14% of the bots receiving command for a polling interval of 1 minute; this value drops drastically to about 1% for 5 minutes. One reasoning behind such a drastic drop is the dynamic nature of the transportation system: individuals come in contact with each other for much shorter intervals of time (as compared to the academic environment represented by the MIT dataset). Therefore, for larger polling interval, higher number of victims are missed between subsequent polls.

The transport model and simulation results reiterate and reinforce the trace-based analysis discussed in the previous section. These results show that public networks fulfill the requirements and the premises underlying the creation of a mobile phone-based botnet using Bluetooth as its communication channel. They also demonstrate that command propagation numbers would be much higher if botmasters target such large-scale public networks.

Note that our observations are necessarily conservative. A more in depth approach could model the movement of people throughout a city throughout an entire day. Such a model would demonstrate that there would be other opportunities for messages to be passed. However, realistically modelling the movements of every person in a city is extremely difficult. By focusing on transportation hubs, we are able to demonstrate that these kinds of botnets are possible and simply note that improvements to the propagation of command and control messages can be expected given other repetitive daily interactions.

7 Defensive Strategies

The modeling and simulation in the previous sections have shown that mobile phone-based botnets can plausibly use Bluetooth as a communications channel. While increasing the latency of C&C messages, this approach significantly reduces the probability that defenders can observe or disrupt these networks. The use of traditional intervention techniques is unlikely to help protect cellular providers against such activity. Careful modifications that consider the patterns of interaction discussed in this paper, however, are likely to prove highly effective.

Like the move towards heterogeneity in platform architectures and operating systems, we argue that software patching mechanisms in this space will begin to mirror the desktop world. In particular, providers will likely be able to help push critical patches to devices. Such a mechanism would provide a number of benefits. For instance, whereas the majority of phone users have never installed software updates, the provider could help them do so with minimal interaction. Such improvements could not only reduce vulnerability to infection, but also improve the services available to users.

Our propagation analysis of mobile phone-based botnet behavior provides some key insights into developing effective remediation strategies. Our results show that botnet command propagation is at its peak when infected devices have a high probability of being in proximity of each other – weekday mornings. Bots are logically less likely to encounter each other later at night and during weekends. Achieving the widest propagation of commands, even if for a time delayed event (one launched later using the loosely synchronized clock provided by the network), therefore requires the botmaster to send messages at specific times of the day. If this brief time window is missed, bots will be forced to communicate with each other over the network.

Remediation can therefore be most successful if launched when bots are least able to coordinate and defend against such efforts. In particular, evenings and weekends provide the most significant periods in which bots are unlikely to communicate via Bluetooth. Such periods also represent the ideal time periods for providers to push updates. In particular, a high density of mobile users in a single location significantly limits the rate with which such updates can be disseminated – previous work has shown the limitations of cellular resources and how attempts to communicate with a large number of users in a single cell can accidentally [32] or intentionally [33] deny service. Accordingly, by pushing patches when customers are least likely to be using their phones and bots are least likely to be able to warn each other without using the provider's network, the provider can more effectively combat such botnets.

Detection mechanisms in this space also hold interesting possibilities. Whereas desktop systems generally rely on installed scanning tools (e.g., antivirus) or network-based IDSs to identify infection, mobile phones have the potential to leverage exciting new mechanisms. In particular, devices plugged into desktops to synchronize information can also be put through a more rigorous set of tests to determine whether or not malware is present. This process can include up-to-the-minute updates from the provider, which can help the more powerful desktop-based software tailor its investigation.

Knowledge of the limitations of a mobile phone-based botnet using Bluetooth helps to mitigate the advantages near-field communications affords in these systems. When used in conjunction with the proposed mitigation infrastructure, such systems are more combatable than initially supposed.

8 Conclusion

Mobile phones are becoming increasingly able to perform critical tasks. However, such devices are also becoming increasing susceptible to infection. Whereas a number of other researchers have investigated the characteristics of such infection, this paper instead attempts to determine whether such devices can be used to support a botnet. In particular, this work tests whether or not such a botnet can effectively communicate using near-field communications to avoid traditional detection mechanisms. In this paper, we demonstrate that such a botnet is possible due to the largely repetitive mobility patterns found in human behavior. Over the course of a day, we show that commands can be consistently disseminated to over 66% of infected nodes. We then leverage such observations to develop more effective patching and mitigation strategies used to either isolate infected devices or force those whose infection is unknown to reveal themselves. While such botnets have not been observed yet in the wild, this work demonstrates that their eventual existence should be anticipated.

References

1. Grand Central Terminal, http://en.wikipedia.org/wiki/Grand_Central_Terminal (last accessed Feburary 2, 2010)
2. MTA NYC Transit: 2007 Ridership by Subway Station (2007), http://www.mta.info/nyct/facts/ridership/ridership_sub.htm (last accessed Feburary 2, 2010)
3. New York City Subway Rolling Stock, http://www.nycsubway.org/cars/index.html (last accessed Feburary 2, 2010)
4. Rail commuters still bothered by cell phone abuse, http://www.trainweb.org/ct/cellsurvey.htm (last accessed Feburary 2, 2010)
5. Sinit P2P Trojan Analysis, http://www.lurhq.com/sinit.html (last accessed Feburary 2, 2010)
6. Barford, P., Yegneswaran, V.: An Inside Look at Botnets. Advances in Information Security (2006)
7. Cooke, E., Jahanian, F., Mcpherson, D.: The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In: Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI), Cambridge, MA (June 2005)
8. F.-S. Corporation: F-Secure Computer Virus Descriptions: Cabir (December 2004), http://www.f-secure.com/v-descs/cabir.shtml (last accessed Feburary 2, 2010)
9. F.-S. Corporation: F-Secure Computer Virus Descriptions: Mabir.A (April 2005), http://www.f-secure.com/v-descs/mabir.shtml (last accessed Feburary 2, 2010)
10. Dagon, D., Gu, G., Lee, C., Lee, W.: A Taxonomy of Botnet Structures. In: Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC), Miami, FL (December 2007)
11. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The Second-Generation Onion Router. In: Proceedings of the 13th USENIX Security Symposium (SECURITY), San Diego, CA (August 2004)
12. Eagle, N., Pentland, A.: Reality Mining: Sensing Complex Social Systems. Journal of Personal and Ubiquitous Computing (2005)
13. Eckman, B.: http://lists.sans.org/pipermail/unisog/2006-April/026261.html (last accessed Feburary 2, 2010)
14. Ferrie, P., Szor, P., Stanev, R., Mouritzen, R.: Security Response: SymbOS.Cabir. Symantec Corporation (2007)
15. Fleizach, C., Liljenstam, M., Johansson, P., Voelker, G.M., Mehes, A.: Can You Infect Me Now?: Malware Propagation in Mobile Phone Networks. In: ACM Workshop on Recurring Malcode (WORM), Alexandria, Virginia, USA (November 2007)

16. Freiling, F.C., Holz, T., Wicherski, G.: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In: di Vimercati, S.d.C., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 319–335. Springer, Heidelberg (2005)
17. Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In: Proceedings of the 17^{th} USENIX Security Symposium (SECURITY), San Jose, CA (July 2008)
18. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In: Proceedings of the 16^{th} USENIX Security Symposium (SECURITY), Boston, MA (August 2007)
19. John, J.P., Moshchuk, A., Gribble, S.D., Krishnamurthy, A.: Studying Spamming Botnets Using Botlab. In: Proceedings of the 6^{th} Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA (April 2009)
20. Jung, S., Lee, U., Chang, A., Cho, D.-K., Gerla, M.: BlueTorrent: Cooperative Content Sharing for Bluetooth Users. In: Proceedings of the 5^{th} IEEE International Conference on Pervasive Computing and Communications (PerCom), White Plains, NY (March 2007)
21. Lactaotao, M.: Security Information: Virus Encyclopedia: Symbos_comwar.a: Technical Details. Trend Micro Incorporated (2005)
22. Mulliner, C., Vigna, G.: Vulnerability Analysis of MMS User Agents. In: Proceedings of the 22^{rd} Annual Computer Security Applications Conference (ACSAC), Miami Beach, FL (December 2006)
23. Nekovee, M.: Worm Epidemics in Wireless Adhoc Networks. Journal of Physics 9, 189 (2007)
24. Pettey, C.: Gartner Says Worldwide Mobile Gaming Revenue to Grow 50 Percent in 2007 (June 2007),
http://www.gartner.com/it/page.jsp?id=507467 (last accessed Feburary 2, 2010)
25. Provos, N., Mavrommatis, P., Rajab, M.A., Monrose, F.: All Your iFRAMEs Point to Us. In: Proceedings of the 17^{th} USENIX Security Symposium (SECURITY), San Jose, CA (July 2008)
26. Rajab, M., Zarfoss, J., Monrose, F., Terzis, A.: My Botnet is Bigger than Yours (Maybe, Better than Yours): Why Size Estimates Remain Challenging. In: Proceedings of the 1^{st} Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA (April 2007)
27. Ramachandran, A., Feamster, N.: Understanding the Network-Level Behavior of Spammers. In: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM), Pisa, Italy (September 2006)
28. Rhodes, C.J., Nekovee, M.: The Opportunistic Transmission of Wireless Worms between Mobile Devices (2008),
http://arxiv.org/abs/0802.2685 (last accessed Feburary 2, 2010)
29. Science Daily. Medical Records on Your Cell Phone: Computer Scientists Turn Cell Phones into Health Care Resource (2006),
http://www.sciencedaily.com/videos/2006/
0306-medical_records_on_your_cell_phone.htm
(last accessed Feburary 2, 2010)
30. Srinivasan, V., Natarajan, A., Motani, M.: CRAWDAD data set nus/bluetooth (v. 2007-09-03) (September 2007),
http://crawdad.cs.dartmouth.edu/nus/bluetooth (last accessed Feburary 2, 2010)
31. Su, J., Chan, K., Miklas, A., Po, K., Akhavan, A., Saroiu, S., Lara, E., Goel, A.: A Preliminary Investigation of Worm Infections in a Bluetooth Environment. In: ACM Workshop on Recurring Malcode (WORM), Alexandria, VA (November 2006)

32. Traynor, P.: Characterizing the Limitations of Third-Party EAS Over Cellular Text Messaging Services. 3G Americas Whitepaper (2008)
33. Traynor, P., Enck, W., McDaniel, P., Porta, T.L.: Exploiting Open Functionality in SMS-Capable Cellular Networks. Journal of Computer Security (JCS) (2008)
34. Traynor, P., Lin, M., Ongtang, M., Rao, V., Jaeger, T., La Porta, T., McDaniel, P.: On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS), Chicago, IL (November 2009)
35. Vamosi, R.: Mobile phone malware in our future (2008), http://news.cnet.com/8301-10789_3-10071982-57.html (Last accessed February 2, 2010)
36. Waldman, M., Rubin, A.D., Cranor, L.F.: Publius: A Robust, Tamper-Evident, Censorship-Resistant Web Publishing System. In: Proceedings of the 9^{th} USENIX Security Symposium (SECURITY), Denver, CO (August 2000)
37. Yan, G., Eidenbenz, S.: Bluetooth Worms: Models, Dynamics, and Defense Implications. In: Proceedings of the 22^{rd} Annual Computer Security Applications Conference (ACSAC), Miami Beach, FL (December 2006)
38. Yan, G., Eidenbenz, S.: Modeling Propagation Dynamics of Bluetooth Worms. In: Proceedings of the 27^{th} International Conference on Distributed Computing Systems (ICDCS), Toronto, Canada (June 2007)
39. Yan, G., Flores, H.D., Cuellar, L., Hengartner, N., Eidenbenz, S., Vu, V.: Bluetooth Worm Propagation: Mobility Pattern Matters! In: Proceedings of the 2^{nd} ACM Symposium on Information, Computer and Communications Security (ASIACCS), Singapore (March 2007)

Take a Deep Breath: A Stealthy, Resilient and Cost-Effective Botnet Using Skype

Antonio Nappa[1], Aristide Fattori[1], Marco Balduzzi[2]
Matteo Dell'Amico[2], and Lorenzo Cavallaro[3]

[1] DICo, Università degli Studi di Milano, Italy
{nappa,joystick}@security.dico.unimi.it
[2] Eurecom, Sophia-Antipolis, France
{marco.balduzzi,matteo.dell-amico}@eurecom.fr
[3] Faculty of Sciences, Vrije Universiteit Amsterdam, The Netherlands
sullivan@few.vu.nl

Abstract. Skype is one of the most used P2P applications on the Internet: VoIP calls, instant messaging, SMS and other features are provided at a low cost to millions of users. Although Skype is a closed source application, an API allows developers to build custom plugins which interact over the Skype network, taking advantage of its reliability and capability to easily bypass firewalls and NAT devices. Since the protocol is completely undocumented, Skype traffic is particularly hard to analyze and to reverse engineer. We propose a novel botnet model that exploits an overlay network such as Skype to build a *parasitic overlay*, making it extremely difficult to track the botmaster and disrupt the botnet without damaging legitimate Skype users. While Skype is particularly valid for this purpose due to its abundance of features and its widespread installed base, our model is generically applicable to distributed applications that employ overlay networks to send direct messages between nodes (e.g., peer-to-peer software with messaging capabilities). We are convinced that similar botnet models are likely to appear into the wild in the near future and that the threats they pose should not be underestimated. Our contribution strives to provide the tools to correctly evaluate and understand the possible evolution and deployment of this phenomenon.

1 Introduction

Botnets are a major plague of the Internet: miscreants have the possibility to hire and control armies of several thousands of infected PCs to fulfill their malicious intents. Theft of sensitive information, sending unsolicited commercial email (SPAM), launching distributed denial of service (DDoS) attacks, and scanning activities are among the most nefarious actions attributable to botnets.

The threat posed by bots and their constant and relevant position in the underground economy made them one of the most discussed topics by security experts and researchers [27,20]. A plethora of techniques have been proposed and deployed to address such a phenomenon [24,37,9,50,51,31]. Some of them aim to understand botnets' *modus operandi* [45,29,44], while others aim to detect patterns typically exhibited by bot-infected machines [24,9,50,51].

Unfortunately, despite the efforts spent by the research community in fighting botnets, they remain an omnipresent menace to the safety, confidentiality, and integrity of Internet users' data. On the one hand, bots authors, increasingly motivated by financial profits [12,45], are constantly looking for the most appealing features a botnet should have: stealthiness (i.e., non-noisy, low-pace, encrypted and distributed communications), resiliency (to nodes shutdown), and cost-effectiveness (i.e., easy to infect/spread to new machines). On the other hand, defense strategies must cope with such ever evolving malware, while being, at the same time, easy-to-deploy exhibiting a contained rate of false positives.

Skype is the *de-facto* standard when it comes to VoIP and related communications. It has a number of ancillary features that make it the ideal platform for a solid communication infrastructure. In fact, it protects the confidentiality of its users by encrypting all their communications, it is fault-tolerant by adopting a de-centralized communication infrastructure, and it is firewall- and NAT-agnostic, meaning that it generally works transparently with no particular network configuration. Therefore, despite its closed-source nature, it is not surprising how Skype has rapidly gained a huge consensus among the millions of Internet users that use it on a daily basis.

Despite some efforts tailored to understanding Skype's code and network patterns [6,17], such a closed infrastructure remains almost obscure, nowadays. Skype-generated network traffic is thus extremely difficult to filter and arduous to analyze with common network-based intrusion detection systems [3]. As an unavoidable consequence, criminals have soon realized how Skype's encrypted communications could then protect the confidentiality of their (illegal) business, hampering the activities of law enforcement agencies [1,2,4,33]. Even worse, the whole Skype infrastructure meets perfectly all the aforementioned features for a stealth, resilient, and cost-effective botnet. A plethora of Skype users can potentially become unwitting victims of a powerful skype-based botnet: the year 2009 alone counted for 443 millions of active Skype accounts, with an average number of 42.2 millions of active users per day [18]. These numbers would certainly attract cyber-criminals soon, and such network and communication characteristics would definitely make the traffic generated by a Skype-based botnet an especially difficult needle to find within the haystack of regular Skype traffic.

In this paper, we show how to take advantage of the features and protection mechanisms offered by Skype to create a botnet that is, at the same time, resilient, easy to deploy and difficult to analyze. Using existing infrastructures for botnet command and control is not a new idea, since many botnets nowadays adopt IRC servers; however, a *parasitic* peer-to-peer overlay built on top of another decentralized overlay allows for many new and interesting features[1]:

[1] We point out that Skype is not the only network that can be exploited by a parasitic overlay: any de-centralized overlay network providing direct messaging capabilities between the nodes can be a suitable target.

- it is hard to set bots and regular Skype traffic apart, as our parasitic overlay network sits on top of the regular Skype network;
- the malicious network so obtained has no bottlenecks nor single point of failure, and this is achieved by deploying an unstructured network with no hierarchical differences among the nodes;
- the lack of a hierarchical structure allows also to use any controlled node as an entry point for the botmaster;
- our parasitic overlay network tolerates the loss of bots: each node/bot contains only a small set of neighbors used for message passing and no information about the botmaster;
- dynamic transparent routing strategies are in charge of routing messages through alternative routes, should one or more bots become unavailable (e.g., shut down);
- the policy adopted for registering new nodes makes it cost-unattractive to obtain a comprehensive list of all the bots.

Simulation experiments we performed show that our model is indeed practical and guarantees a strong resilience to the botnet, ensuring that messages are delivered even when many bots are offline.

It is worth noting that exploring emergent threats is a problem with important practical consequences, as already acknowledged by the research community [49]: emerging botnets have already begun to adopt stealthy communications [45] and de-centralized network infrastructures [29,44] as a mean to become more resilient and hard to track down to keep contributing to the flourishing cyber-criminal business. Thus, a necessary first step for developing robust defenses is that to study about emergent threats. This is the motivation of our work: to show that it is easy, feasible, and practical to deploy a stealth, resilient, and cost-effective botnet. We finally conclude the paper by sketching the design and implementation of a host-based countermeasure that aims to classify Skype plugins as good or malicious by monitoring their interactions with the core application. Although our results are preliminary, they look promising to tackle such a nefarious threat at the end-host systems.

2 Skype Overview

Skype is a widely used application, which features VoIP and calls to land-line phones, audio and video conferencing, SMS and instant messaging, and more. It is organized as a hybrid peer-to-peer (P2P) network with central servers, supernodes, and ordinary clients [3].

Supernodes play an important role in the whole network. In fact, a set of supernodes is responsible for bootstrapping the network. Thus, they act as the point of entrance in the overlay infrastructure, and messages sent by a node are routed through them. They are elected by considering different criteria, such as the availability of a public IP, and the bandwidth/computational resources available on the client. Thus, every host with such features can become a supernode.

The security of Skype is so strong that Governments of some Countries have reported that criminals and mobs started using Skype for their communications in order to avoid eavesdropping by the police forces [1,4,33]. The only effective countermeasures seem to be devoted at attacking the end-hosts system [41].

On the one hand, software that comes with valid security and privacy policies creates positive sensations of trust and safeness to its users, encouraging the enlargement of the installed base. On the other hand, the presence of possible weaknesses in its architecture gives to attackers a means to take advantage of its features for purposes beyond the original design of the application. Indeed, miscreants have already started to misuse the API to deploy malware. For instance, Peskyspy[2] is a trojan that records Skype conversations and provides a backdoor to download them, while Pykspa[2] and Chatosky[2] are two worms that spread using the Skype chat. Skype malware benefit from the fact of being deployed as Skype plugins. In fact, whenever a plugin issues a command (e.g., to create a chat or to send a message), Skype behaves exactly as if the command was invoked by the real user. For example, all the Skype plugins' traffic that leave an host is automatically encrypted. Malware can take advantage of these features to hide themselves and their actions, thus becoming very hard to detect. All these features make the API appetizing to miscreants that look for new and powerful means to create and control a botnet [7,15].

Fortunately, to the best of our knowledge, Skype-based botnets are still exclusively theoretical. Nonetheless, it is interesting to ask ourselves whether such an emerging threat could potentially be a serious menace to the (Internet) society, in the near future. In the following, we show that it is indeed practical and feasible to build a cost-effective botnet by exploiting the features offered by a pre-existent overlay networks such as, in our case, Skype.

2.1 The Skype API

The Skype API allows developers to write applications using features such as sending chat or SMS messages, starting or redirecting calls, or searching for friends. Unfortunately, this API mechanism is far from being as secure as the core of Skype is [17].

A weakness of the API is that there is no control over the number of messages that a plugin is allowed to send. Basically, all the possible activities that a human user can perform through the client (calls, SMSs and chats) can be automated by a plugin, without any flooding control that would limit spamming. For instance, Pykspa, is a Skype-based malware that spreads by spamming messages with links throughout the Skype chat without any rate-limiting threshold. Furthermore, a Skype plugin can directly have access to the search routine and easily harvest many addresses of Skype-registered contacts. This information is fundamental since the access to the search routine gives the possibility to find formerly unknown peers.

Every time a third party application wants to interact with Skype, a check is performed to determine if this software is allowed to access the API. The mechanism

[2] Symantec names.

used by Skype to accomplish this control is based on white/blacklisting. There are two levels of white/blacklists: *local* and *global*. The former is stored in the Skype configuration file using a hash value: when a new plugin wants to interact with Skype, at the first execution, Skype comes up with a dialog box that prompts for the user acknowledgment. When the user selects to authorize or block the plugin, her decision is stored in the configuration file along with its hash value. This hash value is the checksum of the plugin binary and is used by Skype to check for changes in an already authorized plugin. Instead, the latter list is determined centrally by Skype authorities and then propagated to its users throughout the P2P network.

Unfortunately, there are two inherent limitations of such an approach. On the one hand, the existence of API bindings for interpreted languages, make it hard to white/blacklist a single plugin as the interpreter program is the one that interacts with the core. On the other hand, hackers were able to reverse engineer the hash function used to calculate the plugin signatures, and published their results on the Internet giving to malware authors the possibility to develop their own extensions [17]. As such a vulnerability can be fixed in the next releases of Skype, we opt for a different strategy: silently waiting for the authorization dialog to appear, and then performing a fake click to authorize the malware without user consent. We implemented this technique in our bot prototype (see Section 3.2).

3 System Description

In this Section we present our parasitic overlay, a network of non-structured peers that exchange messages over a pre-existent P2P overlay network such as, in our case, Skype.

In our botnet, messages exchanged between bots and the master flow through the network exactly as legitimate messages of the application. This makes the botnet traffic unrecognizable with respect to the legitimate Skype traffic: parasitic overlay nodes behave as ordinary peers of the underlying "host" overlay network. Fig. 1 shows how nodes communicate in the parasitic overlay: botnet nodes (in black, on the above layer, linked to the corresponding infected skype node through a dashed line) send messages to each other directly, without being aware of how routing is performed in the underlying Skype network.

3.1 Botnet Protocol

The communication between the bots and the master is protected by using an ad-hoc encryption scheme in addition to the encryption already performed by Skype. This preserves the confidentiality of the messages exchanged by the bots, even if Skype disclosed, to law enforcement authorities, the encryption key of a particular bot-engaged communication. Messages are sent through the Skype chat as common conversations between users. To accurately replicate the behavior of botnets present in the wild, we designed the architecture in order to provide unicast, multicast and broadcast communication between the master and the bots.

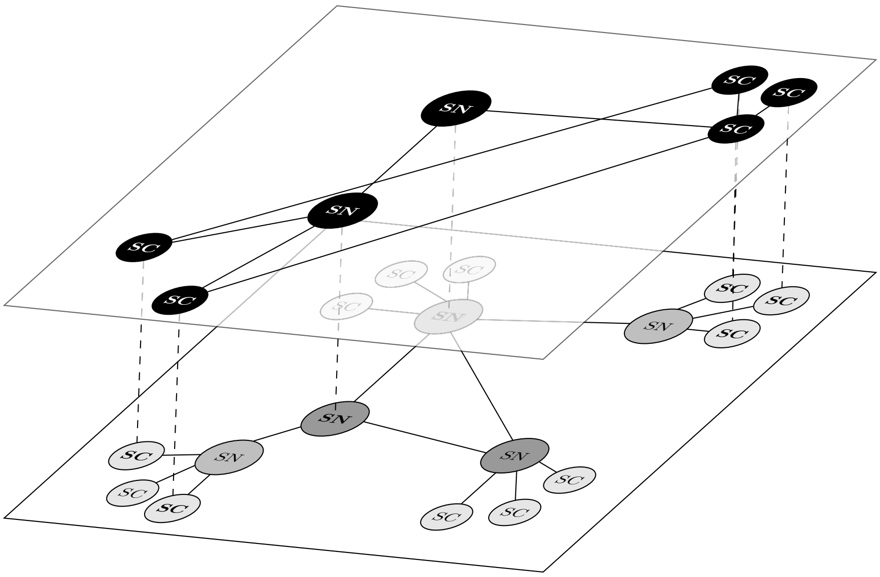


Fig. 1. The parasitic overlay network over Skype. While all the messages are routed from Skype clients (SCs) through supernodes (SNs), the parasitic overlay network makes no hierarchical difference between supernodes and regular clients.

Message Encryption. Bots can receive single commands, group commands and global commands, respectively useful for well targeted attacks, botnet rental or for updates or massive attacks. This is possible by using different encryption mechanisms between the master and the bots. Each node owns a set of symmetric keys: a *node key*, used to receive unicast messages from the master, and any number of *group keys*, to receive multicast messages. Group keys are sent to nodes via new messages from the master. All messages from nodes to the master are encrypted using the master's public key (shipped with the malware binary), while messages from the master to nodes are encrypted with the appropriate symmetric key and signed by the master. All nodes try to decrypt the messages they receive with the keys they possess. All encrypted messages are prepended by a random string to avoid that messages containing the same clear-text result in the same ciphertext.

Message Passing. The message-passing procedure broadcasts every message to all participating peers in the network, using a flooding algorithm similar to the one used in unstructured peer-to-peer networks such as Gnutella [22]: when a peer receives a new message, it forwards it to all neighbors. By doing so, no routing information to reach the botmaster is disclosed. A set of hashes of all received messages is locally kept by nodes to avoid forwarding again old messages. Algorithm 1 shows the details of a bot behavior upon reception of a message. If the received message has not been processed yet, and if the bot is able to decrypt the message, it means the new message is directed to him, thus it verifies if the master's signature is valid. If this condition holds, the bot executes the command received in the message. In the end, regardless whether the message is directed to the bot or not, it stores the hash of the message and forwards it to its neighbors. Overhead and performances of the flooding mechanism will be discussed in Section 4.

```
Input:
  – Received message $M$
  – List of neighbors $N$
  – Set of received message hashes $H$
  – List of symmetric keys (node key and group keys) $K$
Output:
  – Commands to execute $execute(C)$
  – Messages to forward $forward(F, N)$
foreach message $M$ do
    if $hash(M) \in H$ then
        $drop(M)$;
    end
    else
        if $M$ can be decrypted with a key $k \in K$ then
            $C \leftarrow decrypt(M, k)$;
            if signature of $M$ is verified then
                $execute(C)$;
            end
        end
        $add(hash(M), H)$;
        $forward(M, N)$;
    end
end
```

Algorithm 1. Message-passing algorithm

Botnet Bootstrap. When new nodes join the botnet, they bootstrap their connection by generating a node key and by connecting to a set of pre-defined *gate nodes* (GNs), shipped with the binary, that serve as temporary neighbors for the network bootstrap. Gate nodes are ordinary nodes connected to the network, and they are used to reach the botmaster via the message passing protocol described in Section 3.1. The new node announcement contains its Skype username, the newly-generated node key and, as any communications sent from nodes to the botmaster, it is encrypted with the botmaster public key. Again using the message passing protocol, the botmaster responds with a list of l nodes that will be, from that moment on, the neighbors of the new node. This message is encrypted with the node's symmetric key that was sent to the botmaster. Appropriate values for the l parameter will be discussed in Section 4.1. Since the GNs set guarantees to new infected bots the possibility of joining the network, they may appear as a particularly vulnerable point: if the GNs are excluded from the Skype network or the malware gets uninstalled from them, no new nodes can join the botnet. However, it is important to point out that these gate nodes are ordinary nodes, exactly as the other nodes in the network, and therefore the list of GNs shipped with the binary can be updated at will by the botmaster if the GNs are unreachable. Moreover, the gate nodes do not have any special routing information, and therefore they will not disclose any information about the identity of the botmaster even if they fall under the control of an authority and are inspected.

Bootstrap Fail-Over. In the unlikely case the GNs are not available, because they have been dismissed already, the bot cannot receive any bootstrap list from the master. As a fallback measure, the bot issues a Skype search based on a criterion generated from a seed S that is common to all bots. This seed is obtained by an external source that is completely independent and easily accessible by every bot, e.g., by following a strategy similar to the Twitter-seeded domain-flux adopted recently by the Torpig botnet [45]. The master registers one or more Skype users with usernames generated starting from S and sets in their public fields, e.g. the status message, the list of the active GNs that the new bots have to use for their bootstrap phase. As the approach relies on dynamic and daily updated external sources, it seems unfeasible, for a defense mechanism, to predict and shut all the soon-to-be-registered users off in a timely, effective, and cost-effective manner. Moreover, this fallback measure does not expose more information about the parasitic overlay than the normal bootstrap phase.

3.2 Implementation

The proof-of-concept bot has been entirely developed in Python, exploiting the capabilities of the `Skype4Py` library. The library is cross-platform and the bot developed on Linux runs also on Windows and Mac OS X operating systems.

As discussed earlier, Skype comes with an access control mechanism that prevents to load a plugin without an explicit user acknowledgment. At the first execution of a plugin, Skype prompts the user with an authorization request that blocks the program execution until an answer is given. Subsequently, Skype calculates a signature for the plugin and updates its configuration file. There are basically two ways to circumvent this Skype-enforced access control mechanism. One would require to reverse the underlying hashing algorithm, while the other would require to mimic users' interactions. The first approach, described in [17], suffers from updates of the hashing algorithm that would require to be understood and reversed again. On the other hand, the second approach is more generic and is the one we have thus implemented.

From a malware author's perspective, such an access control poses a problem to the automated registration of malicious applications and can decrease the success rate of an infection. To cope with this issue, we integrated in our bot an X11 tool [42] that, simulating keyboard and mouse inputs, automatically validates our plugin registration. We have instructed the prototype to listen and to react immediately at the Skype authorization dialog, authorizing the plugin in a concealed fashion. Our implementation bypasses the Skype security protection mechanism and allows an automated and hidden registration of the bot. The same approach is practical on Microsoft Windows through the use of standard libraries (e.g., `FindWindow()`, `GetCurPos()`, and `MouseEvent()` functions [36]), without the need of external tools.

4 Experiments

We performed two different sets of experiments on our botnet prototype. In Section 4.1, we describe a simulation of our message-passing algorithm that aims at evaluating the trade-offs between the overhead due to the message-passing algorithm and the percentage of bots that are reached by every message. In Section 4.2, we show the empirical observations made while deploying and running our bot prototypes on real systems.

4.1 Network Traffic Simulation

To test the effectiveness of message delivery and the overhead it involves, we wrote a custom simulator that emulates the dynamics of message spreading of Algorithm 1. Since our message-passing protocol floods messages to all nodes, relying on cryptography to ensure that only the intended recipients can decrypt it, we measure the effectiveness of the algorithm with two quantities: coverage, that is the percentage of nodes that are reached by a message sent from a given starting node, and the overhead, expressed as the ration between number of messages sent in the whole system and the number of nodes in it. A perfect algorithm would have a coverage of 100% (all bots are reached) and an overhead of 1 (each peer receives exactly one message).

In our simulator, we generate a network topology with n nodes and l links per node. We start with a completely connected topology of l nodes, and then we iteratively add new nodes connecting them with a random subset of l pre-existing ones, until we reach the desired size of n nodes. Before simulating the propagation of each message, we consider each node and randomly shut it off or leave it on according to a uniform probability a, representing the average peer online availability. Table 1 gives an overview of the simulation parameters.

If we consider our network with the tools of graph theory, it is important to evaluate the size of connected components in the subgraph that we obtain by considering only online nodes. In fact, when a message gets propagated starting from a given node, it will reach all nodes that belong to the same connected component. For Erdös-Rényi (ER) random graphs, a key value is the number of edges in the network, which in our case–considering the probability that nodes are online–corresponds to the value of $l \cdot a \cdot \overline{n}$, where $\overline{n} \simeq n \cdot a$ is the number of online nodes. In an ER graph [8], for a large number of nodes $\overline{n}$, a giant component (i.e., a connected component having size proportional to $\overline{n}$) appears when

Table 1. Parameters for the network traffic simulation

Parameter	Description	Default
n	number of nodes	10,000
l	links added by each new peer	100
m	number of messages sent in the simulation	100
a	probability that a node is online	0.05

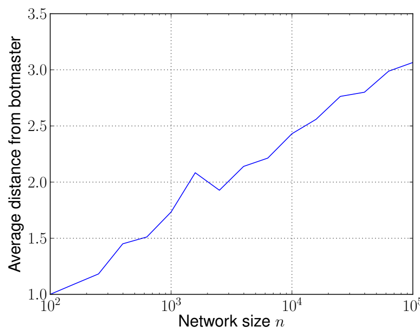
number of edges is greater than $0.5\overline{n}$, and the whole graph becomes connected with high probability when this value reaches $(\ln \overline{n}/2)\overline{n}$.

While in our case, due to the way we build the network, we do not have a perfect ER graph, we experimentally observe in Table 2 a similar behavior with respect to coverage, and a clear trade-off between network coverage and overhead due to sending redundant messages. In particular, a choice of $l = 40$, entailing $l \cdot a \cdot \overline{n} = 2\overline{n}$ reaches around 90% of the nodes imposing a cost of 1.88 messages per node, while a value of $l = 92$ resulting in $l \cdot a \cdot \overline{n} \simeq (\ln \overline{n}/2)\overline{n}$ reaches 99.84% of the nodes that are online, but it involves sending 4.57 messages per node.

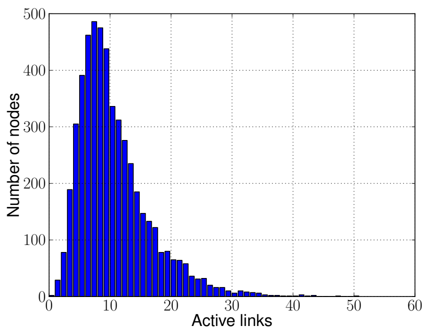
Figure 2(a) shows the number of hops separating, on average, each node from the botmaster with growing network size n. This number of steps, and therefore latencies in message delivery, have a slow logarithmic growth with respect to the network size. Instead, Figure 2(b) shows the number of online neighbors per node ($n = 100,000$ in this case). Nodes that joined the network earlier are more

Table 2. Coverage and overhead for various values of the number of links l

Links per node (l)	Number of edges ($l \cdot a \cdot \overline{n}$)	Coverage	Overhead
10	$0.5 \cdot \overline{n}$	8.69%	0.08
20	$1 \cdot \overline{n}$	55.23%	0.71
40	$2 \cdot \overline{n}$	90.29%	1.88
60	$3 \cdot \overline{n}$	96.76%	2.88
92	$4.60 \cdot \overline{n} \, (\simeq (\ln \overline{n}/2)\overline{n})$	99.84%	4.57



(a) Number of hops.



(b) Number of online neighbors.

Fig. 2. Informations on network topology

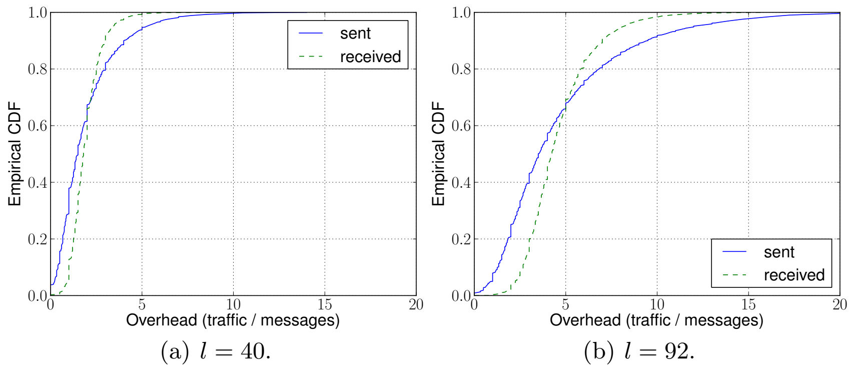


Fig. 3. Number of sent and received messages per node

likely to have more connections, since they had more opportunities of getting chosen as new neighbors of nodes that just joined the network.

Another important issue is load balancing between peers: in Figure 3, we analyze the empirical CDF with the average number of sent/received messages per node. We observe that there is no strong deviation from the average for received messages, while this phenomenon is more strongly perceivable for sent messages. We attribute this to the fact that the older nodes, being more well-connected and closer to the "center" of the network, receive their messages generally earlier than their neighbors and they are thus more often responsible for propagating them.

4.2 Bot Deployment

We tested our Skype botnet infrastructure on a testbed network of several hosts that we "infected" with our bot prototype. We simulate the infection by injecting the bot execution code in the start-up scripts of the infected machine's users. In a real scenario, attackers can infect their victims in several different ways, e.g. by setting-up a drive-by-download site that exploits the visitors' browser vulnerabilities or by sending SPAM email embedding the malicious code.

We deployed our bot on 37 hosts, geographically distributed between France and Italy. In one of them we included the code to support the botmaster operations, such as listing the botnet's bots, managing the network, and sending commands.

In a first experiment, we confirmed the capacity of the bot to discover an active Skype session and to silently register itself as a trusted plugin without explicit user authorization, as outlined in Section 3.2. Then, once given to each bot a single gate node (chosen among the botnet nodes), we verified that all bots correctly joined the botnet, by registering with $l = 2$ neighbors nodes. By doing so, we validated the implementation of the botnet bootstrap protocol formulated in Section 3.1. Table 3 details the timings of bootstrapping steps; bots were able to complete their bootstrapping procedure, on average, within 12 seconds.

Table 3. Bootstrap timing

Bootstrap action	**Time (s)**
Attaching to Skype	1.55
Contacting the Botmaster	4.40
Linking to the Network	6.03
Total	11.98

In a second experiment, we booted every bot and made the botnet run for about 14 hours. We used an ad-hoc *fuzzer* to instruct commands to the bots at random time intervals, registering 1,373 total issued orders. The average time for the master to obtain an answer from a bot that executed a command ranged from 5.25 to 15.75 seconds. We noticed that the variability in this measure depends mainly on the network topology of our parasitic overlay: older nodes, which are usually placed closer to the botmaster, receive their messages first regardless of Internet-level proximity with their neighbors.

With this botnet topology, the bots' answers were reaching the botmaster with an average hop count of 3.62 (see Figure 4). This count is higher than the numbers presented in Figure 2(a), due to a lower value $l = 2$ chosen in this setting. However, only a slow logarithmic growth of this value (and of message delivery latencies) is to be expected with the growth of the network size.

To estimate the amount of traffic generated from our overlay botnet to keep the botmaster concealed, we picked two random nodes, where we measured and classified the number of incoming and outgoing messages. The number of duplicates between all received messages ranged between 74% and 83%. This added cost in network traffic is the price to pay in order to obtain the resilience properties described in Section 4.1.

Finally, to evaluate the resources consumed by our prototype, we installed the *HotSanic* analysis tool [38] on all bots to monitor network, CPU, and memory used by the prototype. Our bandwidth consumption was below 1KB/s even if during the bootstrap phase it was possible to notice some peaks around 6KB/s caused by the messages employed for bootstrapping. We can easily assume that the bandwidth consumption is very low. We did not observe noticeable variations on the use of CPU and memory.

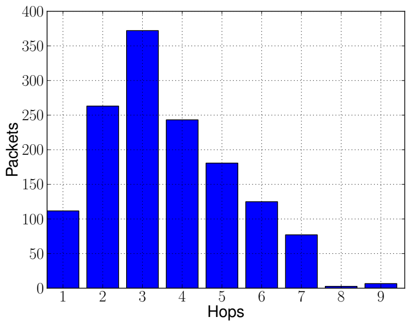


Fig. 4. Number of hops

5 Security Analysis

In this Section, we discuss possible attacks and countermeasures against our botnet model, focusing on each different part of the botnet lifecycle. In the following we refer to “attacker” as a security analyst that is trying to compromise the botnet model. We assume that the attacker is able to reverse engineer the malware and collect traffic dumps of known bots.

First, it is extremely difficult to obtain information about the network topology by observing the traffic sent and received by bots: all traffic undergoes two levels of encryption (one provided by Skype, the other by our scheme); the botmaster can manage the network using any infected node as entry point and change it at will; the routing behavior adopted by the botmaster’s node is exactly the same as any other node. To make the job of the attacker even more difficult, nodes can be instructed to add random delays to message forwarding and to randomly generate “garbage” messages that will be delivered to the whole network even if they cannot be decrypted by any peer.

Second, an attacker that takes control of an infected machine gains access to the list of neighbors’ Skype usernames and to the messages addressed from the master to that bot. This data can be used to detect what the botnet, or part of it, is currently up to (e.g. which SPAM campaigns it is currently perpetrating), but not very much can be said about the overall botnet infrastructure. Nodes can change Skype identifiers if the botmaster instructs them to do so, making information about neighbors short-lived. The botmaster is still very difficult to track since, when seen from a neighbor’s point of view, it is not distinguishable from any other infected host.

Third, if an attacker can successfully reverse-engineer the malware, she is able to discover the hard-coded GNs and to collect the announce sent during bootstrap. With this information, she can perpetrate a replay attack on the botnet. This attack is done by repeatedly delivering announce messages to progressively gather neighbor nodes received during the bootstrap phase. In order to mitigate this attack, it is possible to limit the number of neighbor nodes sent to new bots within any defined temporal window.

Finally, since in our model messages are flooded to the whole network, an attacker can try to overburden the botnet nodes by sending a large number of meaningless messages to the network; to mitigate this effect, we propose to adopt a rate-limiting approach on incoming messages [28]: in this way, only a given maximum number of messages from each neighbor is routed to the rest of the network; thus, only the closest neighbors of each attacker will be likely to suffer from the attack.

6 A Host-Based Skype Malware Detector

We have so far discussed how a hideous Skype plugin can infect a victim to make it part of a botnet. In the following, we describe an approach that allows for a deep analysis of Skype plugins, possibly leading to the detection of the evil ones.

As mentioned earlier, Skype's network traffic is encrypted with strong cryptography algorithms and its binary is protected by anti-debugging and obfuscation techniques [3,17]. While these functionalities create a very good protection system for common users, they also constitute a limit as it is almost impossible for an external entity to investigate Skype's internals and its network traffic. Our idea leverages the fact that, while all the network traffic is encrypted, the messages exchanged on the API communication channel established between Skype and a plugin, as seen in Section 2.1, are completely in clear text. It is therefore possible to analyze the actions performed by a plugin before they are delivered to the Skype core to infer a model that describe the plugin's behaviors at best.

We propose a behavior-based analysis of the *command protocol layer* (CPL) of the Skype API, for the purpose of detecting whether an application is performing malicious actions through Skype. The set of Skype's API commands is quite small and therefore behavior-based analysis can be very effective when applied to the CPL.

The command protocol layer API is based on messaging between Skype and a plugin. This messaging is performed by leveraging the system's standard functions. In a first setup phase, known as *attach* phase, a plugin establish a channel between itself and Skype, known as the *communication layer*. Messages are then exchanged over this layer between the two applications, using a plain-text command protocol.

To perform our analysis, we hijack the attach phase. The hijacking is done through a two-part system: the first component is a Skype plugin of its own, known as *proxy*, while the second one is `WUSSTrace`, a Windows user-space system call tracer [34]. When the proxy receives messages from the target plugin, it simulates the corresponding Skype behavior and replies, thus establishing a communication layer between itself and the plugin. From now on, the proxy acts as a relay and forwards commands sent over the channel from the target plugin to Skype, and viceversa. The plugin is unaware of not being communicating directly with Skype and it consequently behaves normally.

The proxy component includes an analysis engine and several models of malicious behaviors that we created observing the API calls issued by existing Skype malware. By matching the behavior of the attached plugins with the malicious models at our disposal, it is possible to give an preliminary evaluation of the plugin behavior.

Our behavior-based approach is similar to the ones proposed by other malware detection and analysis systems [10,35]. The main difference lies in the fact that we apply it to a different (and higher) level, i.e. the API CPL. By keeping our analysis at the higher CPL level, we avoid all the fine-grained details that other techniques must cope with. These details include, for example, syscall analysis and system API analysis, that are often greatly complex and costly. At the same time, we are able to extract the behavioral semantic of a Skype plugin, exactly as similar techniques.

The first set of results we obtained shows a high rate of false-positive during analyses performed on a certain temporal window. We plan to overcome this limitation by appropriately throttling the temporal window size and inserting an API message rate limiting. This limit should be tailored on the number of interesting API actions performed by plugin in a certain amount of time. We have defined a small set of "interesting" actions and we plan, as a future work, to refine this set by observing the behavior of benign and malign plugins. Through these observations and experiments, we also plan to better refine the temporal window parameter used so far. Although the classification technique is still in an early development stage we are convinced that our approach is a good starting point.

7 Related Work

The botnet phenomenon has quickly become a major security concern for Internet users. As such, it has rapidly gained popularity among the mass media and the attention of the research community interested in understanding, analyzing, and detecting bot-infected machines [29,44,45]. Even worse, such a threat is nowadays exacerbated by the fact that malware authors are willing to give up their fame for an economic profit [12,45]: a reason that motivates by itself miscreants, more than ever, to constantly work towards stealth, high-resilient, and low-cost botnets.

Storm [29] and Waledac [44] are probably the two most famous P2P-based botnets present in the wild. Although these botnets are hard to track down due to their decentralized infrastructure, researchers have shown how it is possible to infiltrate them, disrupt their communications, and acquire detailed information about the botnets' *modus operandi* (e.g., spreading/infection mechanisms, type of threats, corpus of infected hosts).

To overcome such drawbacks, Starnberger *et al.* presented Overbot in [43]. Overbot uses an existing P2P protocol, Kademlia, to provide a stealth command and control channel while guaranteeing the anonymity of the participating nodes and the integrity of the messages exchanged among them. Overbot is the closest work to ours; although we share a similar underlying decentralized structure, there are a number of salient properties that set the two approaches apart. In our approach, the communication bootstrap of a node starts by sending messages using a set of pre-defined nodes–gate nodes (GNs)–that are shipped with the malware. Gate nodes are ordinary bot-infected nodes, and, as such, they perform message routing exactly as any other node. They are just used during the initial bootstrap phase every time a new infected node wants to join the network, but, after that, they do not need to continuously receive communication. On the other hand, sensor nodes in Overbot are a resident part of the botnet: an observer may perform statistical analysis and inferences on the traffic that such nodes generate and receive. Furthermore, Overbot's sensor nodes are equipped with the private key of the botmaster. This means that once a node is compromised, it becomes possible to decrypt *all* the traffic sent to the botmaster. On the other hand, a compromised node in our approach exposes only its symmetric key, which gives the chance to disclose the traffic sent only by that node.

Research in the analysis and detection of bot-infected machines has been quite prolific in the past years [14,21,39,5,23,25,26,24,31,51,46,50,9].

Original signature-based systems, focused on the detection of syntactic artifacts of the malware, are vulnerable to obfuscation techniques and have thus been superseded by approaches that aim to characterize the behavior of a malicious samples on end-user systems [32,35,52,19,10]. These approaches are usually effective in analyzing, detecting and describing malware behaviors. Unfortunately, the ability of the malware to mimic legitimate process behaviors may trick such systems to produce too many false alarms. Moreover, the computational resources required to perform the analysis are non-negligible, and, even worse, users are required to install the analysis platform on their machines. Therefore, it is desirable to have complementary solutions that monitor network events to spot malware-infected machines.

From a different perspective, research in the detection of bots based on the analysis of network events has proceeded by following two main directions. One line of research revolves around the concept of *vertical correlation*. Basically, network events and traffic are inspected, looking for typical evidence of bot infections (such as scanning) or command and control communications [25,23,5]. Unfortunately, some of these techniques are tailored to a specific botnet structure [23,5], while others rely on the presence of a specific bot-infection lifecycle [25]. Others are not immune to encoding or encryption schema as they require to analyze the packets' payload [50], and others again are sensible to perturbation in the network or timing features of the observed connections [9].

The second line of botnet detection research, instead, focuses mainly on *horizontal correlation*, where network events are correlated to identify cases in which two or more hosts are involved in similar, malicious communication. Interesting approaches are represented by BotSniffer [26], BotMiner [24], TAMD [51], and the work proposed in [46]. Except for [46], which detects IRC-based botnets by analyzing aggregated flows, the main strength of the these systems is that they are independent on the underlying botnet structure, and thus, they have shown to be effective in detecting pull-, push-, and P2P-based botnets. On the other hand, correlating actions performed by different hosts requires that at least two hosts in the monitored network are infected by the same bot. As a consequence, these techniques cannot detect single bot-infected hosts. This is a significant limitation, especially when considering the trend toward smaller botnets [14].

Even worse, state-of-the-art techniques [24] are generally triggered upon the observation of malicious and noisy behavioral patterns, where scan, SPAM, and DDoS activities are probably the most representative actions. Unfortunately, in their quest to an ever increasing illegal financial gain [20,27] and to avoid being easily detected by making *much ado for nothing*, bots engage in low-pace, legitimate-resembling activities [45]. Spotting such communications becomes then a very hard task, which, in the end, hampers the detection of the infected machines.

The parasitic overlay network presented in this paper has all the features required to thwart the current state-of-the-art botnet detection approaches. Message encryption hampers the creation of content-based network signatures, while unknown routing strategies make it difficult to track down IP addresses. In addition, Skype itself makes the network highly resilient to failure and provide a massive user corpus, which gives the chance to rely on a non-negligible number of bots. It is worth noting that speculations on using Skype as a vehicle to build a powerful botnet infrastructure have been around for a while [7,15,30,47,48,13]. Fortunately, to the best of our knowledge, such rumors have never evolved into a full-fledged Skype-based botnet in the wild. We have nonetheless shown that such a botnet can be easily designed and implemented. In addition, our simulation and deployment experiments have shown that building a stealthy, resilient and low-cost botnet is indeed possible and practical. Research in botnet detection must thus be refined to deal with the threats posed by such advanced malicious networks that are likely to appear in the near future.

8 Conclusion

In this paper we have described that the design and implementation of a stealth, resilient, and cost-effective botnet based on a general overlay network, such as the one offered by Skype, is not a chimera. It is indeed a practical and realistic threat that may emerge in the near future. The unstructured *parasitic overlay* network proposed, effectively propagates messages leaving to each node only a limited knowledge of the whole network topology, making the botmaster difficult to track down and making the network difficult to map.

In the *parasitic overlay*, messages are flooded through the network to avoid propagating information about how to reach the botmaster, relying on cryptography to ensure that the messages can be only be read by the intended recipients. In future work, taking inspiration from routing strategies in anonymous peer-to-peer networks [40,11,16], we intend to explore more efficient routing strategies for messages, making sure that the information given to each node makes it still difficult to track down the botmaster.

Since we believe that the menace posed by the model of botnet presented in this paper will soon emerge, our future works will focus also on the improvement of the host-based detection technique we briefly outlined.

References

1. Adnkronos International. Italy: Govt probes suspected mafia use of Skype (February 2009), http://www.adnkronos.com/AKI/English/Security/?id=3.0.3031811578
2. Anderson, N.: Is Skype a haven for criminals? (February 2006), http://arstechnica.com/old/content/2006/02/6206.ars
3. Baset, S., Schulzrinne, H.: An analysis of the Skype peer-to-peer internet telephony protocol. In: CoRR (2004)
4. BBC. Italy police warn of Skype threat (February 2009), http://news.bbc.co.uk/2/hi/europe/7890443.stm

5. Binkley, J.R.: An algorithm for anomaly-based botnet detection. In: SRUTI 2006 (2006)
6. Biondi, P., Desclaux, F.: Silver Needle in the Skype (March 2006)
7. Blancher, C.: Fire in the Skype–Skype powered botnets (October 2006), http://sid.rstack.org/pres/0606_Recon_Skype_Botnet.pdf
8. Bollobás, B.: Random Graphs. Cambridge University Press, Cambridge (January 2001)
9. Cavallaro, L., Kruegel, C., Vigna, G.: Mining the network behavior of bots. Tech. Rep. 2009-12, Department of Computer Science, University of California at Santa Barbara (UCSB), CA, USA (July 2009)
10. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy, Oakland 2005 (2005)
11. Ciaccio, G.: Improving sender anonymity in a structured overlay with imprecise routing. LNCS. Springer, Heidelberg (2006)
12. CNET News. Hacking for dollars (July 2005), http://news.cnet.com/Hacking-for-dollars/2100-7349_3-5772238.html
13. CNET News. Skype could provide botnet controls (January 2006), http://news.cnet.com/2100-7349_3-6031306.html
14. Cooke, E., Jahanian, F., McPherson, D.: The zombie roundup: understanding, detecting, and disrupting botnets. In: SRUTI 2005: Proceedings of the Workshop on Steps to Reducing Unwanted Traffic on the Internet (2005)
15. Danchev, D.: Skype to control botnets?! (January 2006), http://ddanchev.blogspot.com/2006/01/skype-to-control-botnets.html
16. Dell'Amico, M.: Mapping small worlds. In: IEEE P2P 2007 (2007)
17. Desclaux, F., Kortchinsky, K.: Vanilla Skype part 2 (June 2006)
18. Ebay. Ebay, Paypak, Skype 2009, Q1 financial report (2009), http://ebayinkblog.com/wp-content/uploads/2009/04/ebay-q1-09-earnings-release.pdf
19. Egele, M., Kruegel, C., Kirda, E., Yin, H.: Dynamic Spyware Analysis. In: Proceedings of the 2007 Usenix Annual Conference, Usenix 2007 (2007)
20. Franklin, J., Paxson, V., Perrig, A., Savage, S.: An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In: CCS 2007: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007)
21. Freiling, F.C., Holz, T., Wicherski, G.: Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In: Proceedings of 10 th European Symposium on Research in Computer Security, ESORICS (2005)
22. Gnutella Development Forum. Gnutella protocol specification, http://wiki.limewire.org/index.php?title=GDF
23. Goebel, J., Holz, T.: Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. In: HotBots 2007: Proceedings of the First Workshop on Hot Topics in Understanding Botnets (2007)
24. Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In: Proceedings of the 17th USENIX Security Symposium (2008)
25. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In: Proceedings of the 16th USENIX Security Symposium (2007)
26. Gu, G., Zhang, J., Lee, W.: BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium, NDSS 2008 (2008)

27. Gutmann, P.: The Commercial Malware Industry. In: Proceedings of the DEFCON conference (2007)
28. He, Q., Ammar, M.: Congestion control and message loss in Gnutella networks. In: Proceedings of SPIE (2003)
29. Holz, T., Steiner, M., Dahl, F., Biersack, E., Freiling, F.: Measurements and Mitigation of Peer-to-Peer-based Botnets:A Case study on Storm Worm. In: USENIX Workshop on Large Scale Exploits and Emerging Threats (2008)
30. IT World: Making a PBX 'botnet' out of Skype or Google Voice? (April 2009), http://www.itworld.com/internet/66280/making-pbx-botnet-out-skype-or-google-voice
31. Karasaridis, A., Rexroad, B., Hoeflin, D.: Wide-scale Botnet Detection and Characterization. In: HotBots 2007: Proceedings of the First Workshop on Hot Topics in Understanding Botnets (2007)
32. Lanzi, A., Sharif, M., Lee, W.: K-Tracer: A System for Extracting Kernel Malware Behavior. In: The 16th Annual Network and Distributed System Security Symposium, NDSS 2009 (2009)
33. Leiden, J.: Anti-mafia cops want Skype tapping (Feburary 2009), http://www.theregister.co.uk/2009/02/24/eurojust_voip_wiretap_probe/
34. Martignoni, L., Paleari, R.: WUSSTrace - a user-space syscall tracer for Microsoft Windows, http://security.dico.unimi.it/projects.shtml
35. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A Layered Architecture for Detecting Malicious Behaviors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 78–97. Springer, Heidelberg (2008)
36. Microsoft. MSDN Library on developing Windows User Interfaces, http://msdn.microsoft.com/en-us/library/ms632587.VS.85.aspx
37. Passerini, E., Paleari, R., Martignoni, L., Bruschi, D.: FLuXOR: Detecting and Monitoring Fast-Flux Service Networks. LNCS. Springer, Heidelberg (2008)
38. Pissny, B.: HotSanic, HTML overview to System and Network Information Center (July 2004), http://hotsanic.sourceforge.net
39. Rajab, M.A., Zarfoss, J., Monrose, F., Terzis, A.: A Multifaceted Approach to Understanding the Botnet Phenomenon. In: IMC 2006: Proceedings of the 6th ACM SIGCOMM on Internet measurement (2006)
40. Sandberg, O.: Distributed routing in small-world networks. In: ALENEX 2006 (2006)
41. Schneier, B.: Bavarian government wants to intercept Skype calls, http://www.schneier.com/blog/archives/2008/02/bavarian_govern.html
42. Sissel, J.: xdotool, http://www.semicomplete.com/projects/xdotool/
43. Starnberger, G., Kruegel, C., Kirda, E.: Overbot - A botnet protocol based on Kademlia. In: Proceedings of the International on Security and Privacy in Communication Networks, SecureComm., Istambul, Turkey (2008)
44. Stock, B., Goebel, J., Engelberth, M., Freiling, F., Holz, T.: Walowdac - Analysis of a Peer-to-Peer Botnet. In: European Conference on Computer Network Defense (EC2ND) (November 2009)
45. Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., Vigna, G.: Your Botnet is My Botnet: Analysis of a Botnet Takeover. In: Proceedings of the 16th ACM conference on Computer and Communications Security, CCS 2009 (2009)

46. Strayer, W.T., Walsh, R., Livadas, C., Lapsley, D.: Detecting botnets with tight command and control. In: Proceedings of the 31st IEEE Conference on Local Computer Networks (2006)
47. TechWorld. Cambridge prof. warns of Skype botnet threat. VoIP traffic can cover a multitude of sins (January 2006), http://news.techworld.com/security/5232/cambridge-prof-warns-of-skype-botnet-threat/
48. TechWorld. How bad is the Skype botnet threat? Skype's sneakiness leads to a security risk (January 2006), http://features.techworld.com/security/2199/how-bad-is-the-skype-botnet-threat/
49. EU Forward. Forward: Managing Emerging Threats in ICT Infrastructures (2008), http://www.ict-forward.eu
50. Wurzinger, P., Bilge, L., Holz, T., Goebel, J., Kruegel, C., Kirda, E.: Automatically Generating Models for Botnet Detection. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 232–249. Springer, Heidelberg (2009)
51. Yen, T.-F., Reiter, M.K.: Traffic Aggregation for Malware Detection. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 207–227. Springer, Heidelberg (2008)
52. Yin, H., Song, D., Egele, D.M., Kruegel, C., Kirda, E.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In: CCS 2007: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007)

Covertly Probing Underground Economy Marketplaces

Hanno Fallmann, Gilbert Wondracek, and Christian Platzer

Vienna University of Technology
Secure Systems Lab
{fallmann,gilbert,cplatzer}@iseclab.org

Abstract. Cyber-criminals around the world are using Internet-based communication channels to establish trade relationships and complete fraudulent transactions. Furthermore, they control and operate publicly accessible information channels that serve as marketplaces for the underground economy. In this work, we present a novel system for automatically monitoring these channels and their participants. Our approach is focused on creating a stealthy system, which allows it to stay largely undetected by both marketplace operators and participants. We implemented a prototype that is capable of monitoring IRC (Internet Relay Chat) and web forum marketplaces, and successfully performed an experimental evaluation over a period of 11 months. In our experimental evaluation we present the findings about the captured underground information channels and their characteristics.

1 Introduction

In recent years, there has been a significant rise in dubious or even outright criminal activity performed via the Internet [1,2]. For example, cyber-criminals conduct credit card fraud, trade compromised user accounts, or openly sell stolen banking credentials online. To communicate with each other and to coordinate themselves, cyber-criminals make use of online communication channels, such as chat rooms, instant messaging, e-mail or web forums. In particular, media like IRC (Internet Relay Chat) chatrooms or Internet forums are frequently used as *underground marketplaces*, virtual places where goods and services that are related to cyber-crime are being offered and traded. These marketplaces appear to be popular among criminals, as they are easily accessible, highly frequented and typically offer a high degree of anonymity to their participants. Clearly, the ability to monitor such underground economy marketplaces would allow researchers and law enforcement to gain new insights into the internals of the existing underground economy and to more efficiently predict or counter cyber-crime. The main contributions presented in this work are the following:

1. We present a novel system for covertly and automatically identifying and monitoring a large number of underground marketplaces simultaneously.

2. We performed an experimental evaluation of our implementation to prove the usability of the developed system.
3. Based on a dataset which spans a period of approximately one year, we present a comprehensive overview on currently established communication methods and channels within the underground community.

In the following section, we give an overview of existing work related to automatically monitoring the underground economy.

2 Related Work

Observing the underground economy is not a new topic, and several related studies have been previously published.

For example, in their study on the underground economy, Thomas and Martin [3] examine IRC based marketplaces. However, as the authors focus on a high-level analysis of the underground economy's structure and actors, they collected relatively little data from these marketplaces.

A more extensive study was conducted by Franklin et al. [1]. In their work, the authors present findings on the underground economy that they derived from the message data of an IRC channel. While a significant amount of data was collected, the scope of the marketplace observation is limited to a single data source.

Interestingly, Herley and Florencio recently published work [4] that claims that the underground economy trading places are classic examples of lemon markets [5], i.e. prices in the underground economy do not necessarily reflect the quality of the offered goods. Furthermore, the authors claim that IRC is mainly used by lesser-skilled cyber-criminals.

In a study [6] by the security company Symantec, both IRC and web forums were covered. The authors claim to have collected 44 million messages over one year in IRC and web forums. Unfortunately, no details on the methods and techniques used for collecting the data are given.

A different approach to underground marketplace monitoring is presented by Holz et al. [2]. Instead of observing the marketplaces directly, the authors analyze data that they have extracted from “dropzones”, i.e. places where criminals collect stolen user data.

In the work of Zhuge et al. [7], aspects related to the Chinese underground market are described. The authors' focus deals primarily with the impact of malicious websites. To estimate the volume of criminal activity, they crawl a single black market forum and only one business platform.

3 Underground Marketplaces

To be able to determine the design criteria for an efficient system for monitoring the underground economy, it is necessary to first understand the characteristics of underground marketplaces. While, in theory, it is possible to use arbitrary communication channels (such as e-mail lists) as underground marketplaces, our real-world observations indicate that only two specific types are widely used by cyber-criminals, IRC rooms and web forums.

3.1 IRC Rooms

IRC (Internet Relay Chat) is a well-known, popular, text-based chat protocol that is specified in an RFC document [8]. However, in order that IRC networks can develop new features as well as protect their users from spammers and automated malicious users, they extend the original RFC specification and create their own protocol rules. Unfortunately, these additions complicate an automated aggregation of messages for our monitoring system since our probes are automated as well. Furthermore, underground related IRC channels (i.e. chat rooms) are actively policed by the channel operator to get rid of unwanted participants, e.g. *rippers* who are fraudulent traders and who scam vendors and buyers alike. In order to solve these challenges, we mimic human behavior and aim at creating as little annoyance as possible to chat participants, while preventing our system from causing excessive resource usage for server operators.

3.2 Web Forums

Web-based forums are the second dominant medium used for underground marketplaces. They are often based on popular off-the-shelf software (e.g., phpBB [9], vBulletin [10]) that is commonly used for benign forums and message boards. In contrast to IRC rooms, forums exhibit a more restricted access policy. Typically, users who wish to participate have to first create user accounts and authenticate themselves via credentials (nickname and password), before they can write or sometimes even read messages. Communication is structured in forum *threads*, which represent a communication topic and consist of a list of messages posted by users, i.e. each new message in a thread is attached to the end of the list.

4 System Design

The overall aim of our system is to observe a large number of underground economy related communication channels. To this end, we deploy a number of sensors for distinct types of communication media. Furthermore, we aim at a system that can be easily extended (i.e., adding sensors should require little effort). In the scope of this document, a *probe* is a software agent within our system that executes surveillance tasks on a specific type of communication medium and that is managed by a *probe pool*. Moreover, the probes are able to collaborate on a given task in a coordinated manner. To this end, probes have the ability to communicate with the main system. For example, a probe can notify the main system if it is unable to continue its task, thus activating a replacement probe. Additionally, our system can incorporate many different network interfaces, making it more stealthy and flexible.

4.1 IRC Sensor

Our general aim in observing IRC networks is to covertly detect underground trading channels and to retrieve a maximum amount of information from them. To this end, it is necessary to observe these channels for longer periods of time (i.e. at least several days), while capturing all public messages during this time frame. In practice, this is a non-trivial task, as it requires our system to be resilient against being intentionally blocked or banned from communication channels by administrators. At the same time, our system should be able to collect user data from individual participants within a channel, while appearing as a genuine user itself.

Information Gathering Methods. Besides recording messages from IRC channels, the IRC sensor probes have the ability to retrieve information about users directly. To this end we developed several methods that differ in the returned information and their "visibility", i.e. some can be regarded as common IRC operations while others might appear more suspicious to a channel operator.

For example, by sending an IRC `whois` request we retrieve, amongst other values, information about the "real" name (designated by the user), the IP address, the channels the user has currently joined, as well as the information if the user has IRC operator privileges. Additionally, we can make use of protocols built on top of the IRC infrastructure (e.g. CTCP (Client To Client Protocol) [11] and DCC (Direct Client to Client) [12] protocol) to learn more about the adversaries and the IRC clients they are using. Moreover, if we collect the IP address of a user, we can apply tools like the geolocation database GeoIP [13] (to pinpoint the IP address to a geographical location), or Nmap [14] (to acquire specific information about the computer of the adversary).

Observation Strategies. To control the behavior of individual probes, the system assigns an *observation strategy* to each probe instance. Each strategy determines which information gathering techniques are used by the probes, and how "aggressive" they are in pursuing their goals.

Chain Strategy. The chain strategy aims at automatically extending the original observation scope (defined by the probe's initial list of channels) by finding additional, interesting channels. To this end, probes following the chain strategy periodically request the list of joined channels from each user in the currently observed channels by sending IRC `whois` requests. For each newly found channel, the size of the intersection set with users in the already observed channels is computed. The channel with the largest intersection set is regarded as the most "popular" channel, and will be added to the probe's target list.

Swap Strategy. We found that IRC users who are too passive, i.e. who do not participate in conversation at all, are frequently removed from IRC channels (e.g. to prevent resource waste or to get rid of "zombie" peers who did not log out properly). To prevent this from happening to our observation probes, the *swap strategy* adds additional probes to observed channels after a certain amount of time. Before the original probe leaves the channel, it waits for a random, intentional overlap time to obscure the "swap".

Chat Strategy. A considerably large proportion of underground economy channel messages are from announcement-bots that advertisers use to draw attention to their offers and requests. Typically, users are asked to send a private message to learn about details of these business offers. As soon as the strategy encounters this type of message, the promoter will be directly engaged using the A.I.M.L. [15] chat system. This chat system locates proper responses to incoming messages using a library consisting of entries written in a XML dialect called Artificial Intelligence Markup Language.

Sensor Strategy. The purpose of this strategy is to cover channels with names and topics that match denoted patterns. To this end, the strategy dispatches an IRC `list` command to retrieve the channels of the network and assigns one probe for each matched channel.

Combinations. Various compositions of strategies can be constructed with different grades of observation behavior, ranging from *passive* to *aggressive*. For example, the combination of a chain strategy with a swap strategy with an aggressive observation attitude using `whois` and DCC requests to concentrate on users, leads to a more adaptive strategy that rotates the probes between channels and expands the observation coverage.

Supervising Information Accumulation - The Right Strategy for the Right Job. To probe underground communication channels in an IRC network they must be discovered first. For each network a network supervisor is initiated that starts the sensor strategy with include and exclude patterns specifically designed for the purpose of recognizing fraudulent trade channels. Our aim is to quickly find these channels in the beginning and then expand the observation with additional methods. If, for example, a credit card trading channel is masked as a sports channel, it will be initially ignored with this method, but will be discovered by another technique.

As soon as a new channel has entered our observation scope a channel supervisor is loaded. Since the intention is not to annoy innocent users and cause needless traffic, the channel supervisor starts the surveillance in a passive manner. After a designated time, all the messages belonging to the channel are automatically assessed on the relation to underground economy context. An SVM (Support Vector Machine) [16], with a training set tailored to recognize fraudulent content, makes the classification possible. Based on the affinity of a channel to underground economy, the channel supervisor adapts its observation strategies and mechanisms.

Besides using the *pattern matching* approach to broaden the observation scope, the channel supervisor applies the chain strategy on fraudulent channels to observe other *popular channels* of the current users. Additionally, if a designated quota of maximum channels is not reached, *random channels* are being joined and dismissed as soon as the SVM reasoner classified them as being benign to make room for new random channels.

4.2 Web Forum Sensor

According to Guo et al. [17], web forums exhibit a number of characteristics that make it non-trivial to extract structured data with standard web crawlers. The major problem is that the same content of a forum can have a multitude of URIs addressing it. One reason, is the dynamic nature of a web forum. For example, two requests, that differ in the URI, can lead the forum engine into generating the same content. Another reason is the existence of "noisy links", i.e. URIs that contain functions like ordering posts or searching content. This problem can lead a standard web crawler to be redirected in a circular way (so-called *spider-traps*). A further intricacy we faced is the diversity of different forum engines and versions that require a general solution. Taking these challenges into consideration, we decided to use the approach described by Yang et al. [18].

Crawling Underground Economy Forums. Unlike benign forums (i.e. non-underground), most underground economy related forums employ some sort of additional authentication measures or counter-measures that prevent automatic crawling. The following list highlights the most frequently employed mechanisms, and how we address them in our implementation.

1. The content is only viewable for registered users. Since we enhanced the crawler with login functionality, we only have to manually register users to the forum one time.
2. Reputation-based trust systems that allow only users who gained a high enough status to view the content. An example are *escrow services*, i.e. forum administrators charge a fee to verify the integrity and quality of trade offers and monetary transactions before any goods are exchanged. We are not able to automatically gain these privileges for a user, but if the content seems to be valuable for analysis, this can be done manually.
3. Individual users can only view a certain amount of pages per time unit, sometimes coupled with a limitation of recovered pages per network address. The solution to evade these restriction of viewable pages is similar to the IRC swap strategy: Each registered probe has a dedicated IP address assigned. During the crawl of the forum the probes are being changed (logoff old probe, login new probe) after a random amount of acquired pages.

5 Experimental Evaluation

We started an observation of underground economy marketplaces in March 2009 and collected data for the following 11 months.

5.1 Coverage and Proportion of IRC Networks

To find interesting IRC networks for our experiment, we initially extracted known server addresses from the server list of mIRC [19], a popular IRC client, and from a website [20] that is dedicated to finding IRC networks. Additionally, we complemented this list with servers that we manually extracted from announcements in underground economy IRC channels.

In our crawling experiments, we examined a total of 26,207 distinct IRC channels from 291 networks. Of these, 2,677 channels contained chat messages and 4.7% of them have been recognized to be related to underground economy content. A chat message is a public channel message that excludes server notifications such as join (i.e. entering a channel), part (i.e. leaving a channel) or kick (i.e. expel a user from a channel). The content identification has been done with an SVM (Support Vector Machine) [16]. The categorization results were manually checked and we found zero false positive recognized channels and 67 false negative recognized channels. However, the SVM module is exchangeable and an improved version can be effortless integrated. We found 23,530 channels to contain no chat messages.

In Sect. 4.1 we have described three different basic methods to provide a reasonable coverage of fraudulent channels in the search space. The *pattern matching* approach retrieves the obvious channels and has the biggest hit rate with nearly 58% of all covered fraudulent channels. The chain strategy provides all the *popular channels* we missed and constitutes with a scope of over 40% together with the pattern matching approach, 98% of all exposed channels. Only two channels have been detected with the joining of *random channels*. However, this method allowed us to exclude over 22,000 channels not being used for criminal activities.

5.2 IRC Observation Results

In 291 IRC Networks 495,939 distinct user names have been accumulated. Using 14,526 probes we gathered 15 GB of data for which the statistic is listed in Table 1. *Kicks* denotes the number of received expulsion over all users where as *Kicks of Probes* only takes our observing users into account and *Distinct Kicks of Probes* counts manifold expulsions of a probe in a channel as one. *Channel Bans* consists of the number of distinct channels we were banned from. Comparing the total value of the kick rate with the rate that affected only our probes, it is clearly visible that our strategies significantly reduced the potential of being expelled from a channel. Because traders advertise their goods with a high message rate, they are responsible for a big fraction of all the messages we collected. We can clearly confirm the presence of these traders and the extension of their actions in IRC.

5.3 Web Forum Observation Results

First of all, before the observation can start, we have to locate addresses of web forums in which illicit trade is taking place. To this end, we initially gather URIs via keywords entered in web search engines. After underground economy related forums have been crawled, it is possible to derive forum addresses from them. Additionally the sensor system provides a global search on all communication

Table 1. Statistics of the IRC observation regarding the user behavior

	Malignant	Per Channel	Benign	Per Channel
Channels	126	-	2,551	-
Chat Messages	43,148,421	342,447.79	2,950,208	1,156.49
Joins	550,685	4,370.52	562,002	220.31
Parts	100,354	796.46	169,670	66.51
Kicks	25,298	200.78	2,792	1.09
Kicks of probes	79	0.63	1,996	0.78
Distinct Kicks of probes	42	0.33	1,105	0.43
Channel Bans	26	0.21	681	0.27

channels: if a forum address is posted in an IRC channel, an observation can be started immediately on it and vice versa. To test the web forum sensor, we crawled eleven different forums multiple times. All in all we gathered from 11 web forums over 127 GB of pages that contain over one million forum posts.

5.4 Classification and Analysis of Web Forums Related to Underground Economy

Web forums are being used differently by miscreants: Advertisers use spamming tools on mostly innocent forums to promote messages similar to those on IRC channels. Some forums are only used to exchange knowledge, to provide tutorials for beginners or to find new contacts. Other forums include trading sections as well, including auctions of stolen goods.

Table 2 shows examples for different usages of such forums are listed. A low value of *Posts per User* is an indication for a high proportion of different users with a low post count. This can be, either due to the fact that a forum is fairly new, or if it is being abused by spammers. Depending on the vigilance of the forum admins, these newly created users and posts have a short life-time. Since

Table 2. Examples of different forums and how they are being used in the underground economy. (The letter *d* stand for discussion of underground economy, *t* for the trading of goods, and *s* for a spammed forum).

Forum	Boards	Threads	Posts	Users	Posts/User	1 Post/Thread	d	t	s
blackhatpalace.com	22	424	1,323	160	8.27	49.92%	✓		
forum.rorta.net	21	2,643	53,731	843	63.74	5.60%	✓		
www.carders.cc	67	6,290	65,312	2,411	27.09	16.96%	✓	✓	
www.hack-info.ru	47	27,221	207,020	9,891	20.93	34.80%	✓	✓	
www.clicks.ws	26	9,463	19,975	2,681	7.45	76.71%			✓
www.hotsurfs.com	62	39,297	61,287	2,161	28.36	88.78%			✓
www.talk-hyip.com	9	3,884	5,966	2,166	2.75	94.00%			✓
www.wifi-forum.com	22	109,221	610,658	32,084	19.03	59.00%			✓

the majority of the spamming tools create a new thread and post one spam message into it, the percentage of threads with only one post can additionally be used to figure out the current state of the forum regarding spam. The forum `www.talk-hyip.com` with an average rate of 2.75 posts per user and a proportion of 94% of threads containing only one post, is an example of a lost battle against spammers.

The time span of the forums, from the date of the first post till the date of the last post, reflects on the forum type as well. In our data, the lifetime of fraudulent forums with trading and discussion sections is shorter when compared to spammed forums. To get the data of underground economy forums with an active community we visited some of the sites more than once, to find out how many new posts have been committed. Occasionally it happened that a site was offline, either because they were completely shut down, or because they were just not reachable for a few months.

6 Conclusion

In this work, we presented a novel system for automatically monitoring adversary information channels. For example, in the domain of the online underground economy, researchers who study online crime or law enforcement agencies have a vital interest in acquiring data from related sources, such as underground marketplaces or chatrooms used by criminals. To the best of our knowledge, our system is the first to include specific features to monitor information channels related to the underground economy. Furthermore, our system can mimic (human) user behavior to remain stealthy, i.e. avoid being detected by administrators.

For an experimental evaluation we have implemented a prototype that can observe IRC channels and web forums, the most widely spread information media used by cyber criminals. During a period of 11 months, our system has managed to collect a dataset of more than 43 million chat messages and approximately one million forum entries from underground sources without experiencing any problems. This demonstrates that our system can be effectively used in a real-world setting to acquire vital information on cybercrime, which can be used for investigations or research in the area.

Acknowledgements

This work has been supported by the Austrian Research Promotion Agency (FFG) under grant 820854. We also thank our shepherd Kirill Levchenko and the anonymous reviewers for their valuable insights and comments.

References

1. Franklin, J., Paxson, V., Savage, S., Perrig, A.: An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In: ACM Conference on Computer and Communications Security (CCS), November 2007. ACM, New York (2007)

2. Holz, T., Engelberth, M., Freiling, F.C.: Learning More about the Underground Economy: A Case-Study of Keyloggers and Dropzones. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 1–18. Springer, Heidelberg (2009)
3. Thomas, R., Martin, J.: The Underground Economy: Priceless. In: USENIX; LOGIN (2006)
4. Herley, C., Florencio, D.: Nobody Sells Gold for the Price of Silver: Dishonesty, Uncertainty and the Underground Economy. Technical report, Microsoft Research (2009)
5. Akerlof, G.A.: The Market for "Lemons": Quality Uncertainty and the Market Mechanism. The Quarterly Journal of Economics (3) (1970)
6. Symantec: Symantec Report on the Underground Economy (2008), `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_underground_economy_report_11-2008-14525717.en-us.pdf`
7. Zhuge, J., Holz, T., Song, C., Guo, J., Han, X., Zou, W.: Studying Malicious Websites and the Underground Economy on the Chinese Web. Technical report (2008)
8. Oikarinen, J., Reed, D.: RFC 1459: Internet Relay Chat Protocol. Technical report (May 1993)
9. Online: phpBB, `http://www.phpbb.com/` (accessed: April 2010)
10. Online: vBulletin, `http://www.vbulletin.com/` (accessed: April 2010)
11. Zeuge, K., Rollo, T., Mesander, B.: Client To Client Protocol (CTCP), `http://www.irchelp.org/irchelp/rfc/ctcpspec.html`
12. Zeuge, K., Rollo, T., Mesander, B.: Direct Client Connection (DCC), `http://www.irchelp.org/irchelp/rfc/dccspec.html`
13. Online: GeoIP, `http://www.maxmind.com/` (accessed: April 2010)
14. Online: Network Tool Nmap, `http://nmap.org/` (accessed: April 2010)
15. Wallace, R.: The Elements of AIML Style. Technical report, ALICE A.I. Foundation (2003)
16. Joachims, T.: Text Categorization with Support Vector Machines: Learning with Many Relevant Features. In: European Conference on Machine Learning (ECML), pp. 137–142. Springer, Berlin (1998)
17. Guo, Y., Li, K., Zhang, K., Zhang, G.: Board Forum Crawling: A Web Crawling Method for Web Forum. In: WI 2006: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence, Washington, DC, USA, pp. 745–748. IEEE Computer Society, Los Alamitos (2006)
18. Yang, J.M., Cai, R., Wang, Y., Zhu, J., Zhang, L., Ma, W.Y.: Incorporating site-level knowledge to extract structured data from web forums. In: WWW 2009: Proceedings of the 18th international conference on World wide web, pp. 181–190. ACM, New York (2009)
19. Online: mIRC server list, `http://www.mirc.com/servers.ini` (accessed: April 2010)
20. Online: IRC netsplit, `http://irc.netsplit.de/` (accessed: April 2010)

Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners

Adam Doupé, Marco Cova, and Giovanni Vigna

University of California, Santa Barbara
{adoupe,marco,vigna}@cs.ucsb.edu

Abstract. Black-box web vulnerability scanners are a class of tools that can be used to identify security issues in web applications. These tools are often marketed as "point-and-click pentesting" tools that automatically evaluate the security of web applications with little or no human support. These tools access a web application in the same way users do, and, therefore, have the advantage of being independent of the particular technology used to implement the web application. However, these tools need to be able to access and test the application's various components, which are often hidden behind forms, JavaScript-generated links, and Flash applications.

This paper presents an evaluation of eleven black-box web vulnerability scanners, both commercial and open-source. The evaluation composes different types of vulnerabilities with different challenges to the crawling capabilities of the tools. These tests are integrated in a realistic web application. The results of the evaluation show that crawling is a task that is as critical and challenging to the overall ability to detect vulnerabilities as the vulnerability detection techniques themselves, and that many classes of vulnerabilities are completely overlooked by these tools, and thus research is required to improve the automated detection of these flaws.

1 Introduction

Web application vulnerabilities, such as cross-site scripting and SQL injection, are one of the most pressing security problems on the Internet today. In fact, web application vulnerabilities are widespread, accounting for the majority of the vulnerabilities reported in the Common Vulnerabilities and Exposures database [4]; they are frequent targets of automated attacks [20]; and, if exploited successfully, they enable serious attacks, such as data breaches [9] and drive-by-download attacks [17]. In this scenario, security testing of web applications is clearly essential.

A common approach to the security testing of web applications consists of using *black-box web vulnerability scanners*. These are tools that crawl a web application to enumerate all the reachable pages and the associated input vectors (e.g., HTML form fields and HTTP GET parameters), generate specially-crafted input values that are submitted to the application, and observe the application's behavior (e.g., its HTTP responses) to determine if a vulnerability has been triggered.

Web application scanners have gained popularity, due to their independence from the specific web application's technology, ease of use, and high level of automation.

(In fact, web application scanners are often marketed as "point-and-click" pentesting tools.) In the past few years, they have also become a requirement in several standards, most notably, in the Payment Card Industry Data Security Standard [15].

Nevertheless, web application scanners have limitations. Primarily, as most testing tools, they provide no guarantee of soundness. Indeed, in the last few years, several reports have shown that state-of-the-art web application scanners fail to detect a significant number of vulnerabilities in test applications [21,22,24,16,1]. These reports are valuable, as they warn against the naive use of web application scanners (and the false sense of security that derives from it), enable more informed buying decisions, and prompt to rethink security compliance standards.

However, knowing that web application scanners miss vulnerabilities (or that, conversely, they may raise false alerts) is only part of the question. Understanding *why* these tools have poor detection performance is critical to gain insights into how current tools work and to identify open problems that require further research. More concretely, we seek to determine the root causes of the errors that web application scanners make, by considering all the phases of their testing cycle, from crawling, to input selection, to response analysis. For example, some of the questions that we want to answer are: Do web application scanners correctly handle JavaScript code? Can they detect vulnerabilities that are "deep" in the application (e.g., that are reachable only after correctly submitting complex forms)? Can they precisely keep track of the state of the application?

To do this, we built a realistic web application, called WackoPicko, and used it to evaluate eleven web application scanners on their ability to crawl complex web applications and to identify the associated vulnerabilities. More precisely, the WackoPicko application uses features that are commonly found in modern web applications and that make their crawling difficult, such as complex HTML forms, extensive JavaScript and Flash code, and dynamically-created pages. Furthermore, we introduced in the application's source code a number of vulnerabilities that are representative of the bugs commonly found in real-world applications. The eleven web application scanners that we tested include both commercial and open-source tools. We evaluated each of them under three different configuration settings, corresponding to increasing levels of manual intervention. We then analyzed the results produced by the tools in order to understand how the tools work, how effective they are, and what makes them fail. The ultimate goal of this effort is to identify which tasks are the most challenging for black-box vulnerability scanners and may require novel approaches to be tackled successfully.

The main contributions of this paper are the following:

- We performed the most extensive and thorough evaluation of black-box web application vulnerability scanners so far.
- We identify a number of challenges that scanners need to overcome to successfully test modern web applications both in terms of crawling and attack analysis capabilities.
- We describe the design of a testing web site for web application scanners that composes crawling challenges with vulnerability instances. This site has been made available to the public and can be used by other researchers in the field.
- We analyze in detail *why* the web application vulnerability scanners succeed or fail and we identify areas that need further research.

2 Background

Before discussing the design of our tests, it is useful to briefly discuss the vulnerabilities that web application scanners try to identify and to present an abstract model of a typical scanner.

2.1 Web Application Vulnerabilities

Web applications contain a mix of traditional flaws (e.g., ineffective authentication and authorization mechanisms) and web-specific vulnerabilities (e.g., using user-provided inputs in SQL queries without proper sanitization). Here, we will briefly describe some of the most common vulnerabilities in web applications (for further details, the interested reader can refer to the OWASP Top 10 List, which tracks the most critical vulnerabilities in web applications [13]):

- **Cross-Site Scripting (XSS):** XSS vulnerabilities allow an attacker to execute malicious JavaScript code as if the application sent that code to the user. This is the first most serious vulnerability of the OWASP Top 10 List, and WackoPicko includes five different XSS vulnerabilities, both reflected and stored.
- **SQL Injection:** SQL injection vulnerabilities allow one to manipulate, create or execute arbitrary SQL queries. This is the second most serious vulnerability on the OWASP Top 10 List, and the WackoPicko web application contains both a reflected and a stored SQL injection vulnerability.
- **Code Injection:** Code injection vulnerabilities allow an attacker to execute arbitrary commands or execute arbitrary code. This is the third most serious vulnerability on the OWASP Top 10 List, and WackoPicko includes both a command line injection and a file inclusion vulnerability (which might result in the execution of code).
- **Broken Access Controls:** A web application with broken access controls fails to properly define or enforce access to some of its resources. This is the tenth most serious vulnerability on the OWASP Top 10 List, and WackoPicko has an instance of this kind of vulnerability.

2.2 Web Application Scanners

In abstract, web application scanners can be seen as consisting of three main modules: a *crawler* module, an *attacker* module, and an *analysis* module. The crawling component is seeded with a set of URLs, retrieves the corresponding pages, and follows links and redirects to identify all the reachable pages in the application. In addition, the crawler identifies all the input points to the application, such as the parameters of GET requests, the input fields of HTML forms, and the controls that allow one to upload files.

The attacker module analyzes the URLs discovered by the crawler and the corresponding input points. Then, for each input and for each vulnerability type for which the web application vulnerability scanner tests, the attacker module generates values that are likely to trigger a vulnerability. For example, the attacker module would attempt to inject JavaScript code when testing for XSS vulnerabilities, or strings that have a special meaning in the SQL language, such as ticks and SQL operators, when testing for SQL injection vulnerabilities. Input values are usually generated using heuristics or using predefined values, such as those contained in one of the many available XSS and SQL injection cheat-sheets [18,19].

The analysis module analyzes the pages returned by the web application in response to the attacks launched by the attacker module to detect possible vulnerabilities and to provide feedback to the other modules. For example, if the page returned in response to input testing for SQL injection contains a database error message, the analysis module may infer the existence of a SQL injection vulnerability.

3 The WackoPicko Web Site

A preliminary step for assessing web application scanners consists of choosing a web application to be tested. We have three requirements for such an application: it must have clearly defined vulnerabilities (to assess the scanner's detection performance), it must be easily customizable (to add crawling challenges and experiment with different types of vulnerabilities), and it must be representative of the web applications in use today (in terms of functionality and of technologies used).

We found that existing applications did not satisfy our requirements. Applications that deliberately contain vulnerabilities, such as HacmeBank [5] and WebGoat [11], are often designed to be educational tools rather than realistic testbeds for scanners. Others, such as SiteGenerator [10], are well-known, and certain scanners may be optimized to perform well on them. An alternative then is to use an older version of an open-source application that has known vulnerabilities. In this case, however, we would not be able to control and test the crawling capabilities of the scanners, and there would be no way to establish a false negative rate.

Therefore, we decided to create our own test application, called WackoPicko. It is important to note that WackoPicko is a realistic, fully functional web application. As opposed to a simple test application that contains just vulnerabilities, WackoPicko tests the scanners under realistic conditions. To test the scanners' support for client-side JavaScript code, we also used the open source Web Input Vector Extractor Teaser (WIVET). WIVET is a synthetic benchmark that measures how well a crawler is able to discover and follow links in a variety of formats, such as JavaScript, Flash, and form submissions.

3.1 Design

WackoPicko is a photo sharing and photo-purchasing site. A typical user of WackoPicko is able to upload photos, browse other user's photos, comment on photos, and purchase the rights to a high-quality version of a photo.

Authentication. WackoPicko provides personalized content to registered users. Despite recent efforts for a unified login across web sites [14], most web applications require a user to create an account in order to utilize the services offered. Thus, WackoPicko has a user registration system. Once a user has created an account, he/she can log in to access WackoPicko's restricted features.

Upload Pictures. When a photo is uploaded to WackoPicko by a registered user, other users can comment on it, as well as purchase the right to a high-quality version.

Comment On Pictures. Once a picture is uploaded into WackoPicko, all registered users can comment on the photo by filling out a form. Once created, the comment is displayed, along with the picture, with all the other comments associated with the picture.

Purchase Pictures. A registered user on WackoPicko can purchase the high-quality version of a picture. The purchase follows a multi-step process in which a shopping cart is filled with the items to be purchased, similar to the process used in e-commerce sites. After pictures are added to the cart, the total price of the cart is reviewed, discount coupons may be applied, and the order is placed. Once the pictures are purchased, the user is provided with links to the high-quality version of the pictures.

Search. To enable users to easily search for various pictures, WackoPicko provides a search toolbar at the top of every page. The search functionality utilizes the tag field that was filled out when the picture was uploaded. After a query is issued, the user is presented with a list of all the pictures that have tags that match the query.

Guestbook. A guestbook page provides a way to receive feedback from all visitors to the WackoPicko web site. The form used to submit feedback contains a "name" field and a "comment" field.

Admin Area. WackoPicko has a special area for administrators only, which has a different login mechanism than regular users. Administrators can perform special actions, such as deleting user accounts, or changing the tags of a picture.

3.2 Vulnerabilities

The WackoPicko web site contains sixteen vulnerabilities that are representative of vulnerabilities found in the wild, as reported by the OWASP Top 10 Project [13]. In the following we provide a brief description of each vulnerability.

3.2.1 Publicly Accessible Vulnerabilities

A number of vulnerabilities in WackoPicko can be exploited without first logging into the web site.

Reflected XSS3. There is a XSS vulnerability on the search page, which is accessible without having to log into the application. In fact, the query parameter is not sanitized before being echoed to the user. The presence of the vulnerability can be tested by setting the query parameter to `<script>alert('xss')</script>`. When this string is reflected to the user, it will cause the browser to display an alert message. (Of course, an attacker would leverage the vulnerability to perform some malicious activity rather than alerting the victim.)

Stored XSS. There is a stored XSS vulnerability in the guestbook page. The `comment` field is not properly escaped, and therefore, an attacker can exploit this vulnerability by creating a comment containing JavaScript code. Whenever a user visits the guestbook page, the attack will be triggered and the (possibly malicious) JavaScript code executed.

Session ID. The session information associated with administrative accounts is handled differently than the information associated with the sessions of normal users. The functionality associated with normal users uses PHP's session handling capabilities, which is assumed to be free of any session-related vulnerabilities (e.g., session fixation, easily-guessable session IDs). However the admin section uses a custom session cookie to keep track of sessions. The value used in the cookie is a non-random value that is incremented when a new session is created. Therefore, an attacker can easily guess the session id and access the application with administrative rights.

Weak password. The administrative account page has an easily-guessable username and password combination: admin/admin.

Reflected SQL Injection. WackoPicko contains a reflected SQL injection in the `username` field of the login form. By introducing a tick into the `username` field it is possible to perform arbitrary queries in the database and obtain, for example, the usernames and passwords of all the users in the system.

Command Line Injection. WackoPicko provides a simple service that checks to see if a user's password can be found in the dictionary. The `password` parameter of the form used to request the check is used without sanitization in the shell command: `grep ^<password>$ /etc/dictionaries-common/words`. This can be exploited by providing as the password value a dollar sign (to close grep's regular expression), followed by a semicolon (to terminate the grep command), followed by extra commands.

File Inclusion. The admin interface is accessed through a main page, called *index.php*. The index page acts as a portal; any value that is passed as its `page` parameter will be concatenated with the string ".php", and then the resulting PHP script will be run. For instance, the URL for the admin login page is `/admin/index.php?page=login`. On the server side, *index.php* will execute *login.php* which displays the form. This design is inherently flawed, because it introduces a file inclusion vulnerability. An attacker can exploit this vulnerability and execute remote PHP code by supplying, for example, `http://hacker/blah.php%00` as the `page` parameter to *index.php*. The `%00` at the end of the string causes PHP to ignore the ".php" that is appended to the page parameter. Thus *index.php* will download and execute the code at `http://hacker/blah.php`.

Unauthorized File Exposure. In addition to executing remote code, the file inclusion vulnerability can also be exploited to expose local files. Passing `/etc/passwd%00` as the "page" GET parameter to *index.php* of the admin section will cause the contents of the `/etc/passwd` file to be disclosed.

Reflected XSS Behind JavaScript. On WackoPicko's home page there is a form that checks if a file is in the proper format for WackoPicko to process. This form has two parameters, a file parameter and a name parameter. Upon a successful upload, the name is echoed back to the user unsanitized, and therefore, this represents a reflected vulnerability. However, the form is dynamically generated using JavaScript, and the target of the form is dynamically created by concatenating strings. This prevents a crawler from using simple pattern matching to discover the URL used by the form.

Parameter Manipulation. The WackoPicko home page provides a link to a sample profile page. The link uses the "userid" GET parameter to view the sample user (who has id of 1). An attacker can manipulate this variable to view any profile page without having a valid user account.

3.2.2 Vulnerabilities Requiring Authentication

A second class of vulnerabilities in WackoPicko can be exploited only after logging into the web site.

Stored SQL Injection. When users create an account, they are asked to supply their first name. This supplied value is then used unsanitized on a page that shows other users who have a similar first name. An attacker can exploit this vulnerability by creating a user with the name "' ; DROP users;#" then visiting the similar users page.

Directory Traversal. When uploading a picture, WackoPicko copies the file uploaded by the user to a subdirectory of the *upload* directory. The name of the subdirectory is the user-supplied tag of the uploaded picture. A malicious user can manipulate the tag parameter to perform a directory traversal attack. More precisely, by pre-pending "`../../`" to the tag parameter the attacker can reference files outside the upload directory and overwrite them.

Multi-Step Stored XSS. Similar to the stored XSS attack that exists on the guestbook, comments on pictures are susceptible to a stored XSS attack. However, this vulnerability is more difficult to exploit because the user must be logged in and must confirm the preview of the comment before the attack is actually triggered.

Forceful Browsing. One of the central ideas behind WackoPicko is the ability of users to purchase the rights to high-quality versions of pictures. However, the access to the links to the high-quality version of the picture is not checked, and an attacker who acquires the URL of a high-quality picture can access it without creating an account, thus bypassing the authentication logic.

Logic Flaw. The coupon system suffers from a logic flaw, as a coupon can be applied multiple times to the same order reducing the final price of an order to zero.

Reflected XSS Behind Flash. On the user's home page there is a Flash form that asks the user for his/her favorite color. The resulting page is vulnerable to a reflected XSS attack, where the "value" parameter is echoed back to the user without being sanitized.

3.3 Crawling Challenges

Crawling is arguably the most important part of a web application vulnerability scanner; if the scanner's attack engine is poor, it *might* miss a vulnerability, but if its crawling engine is poor and cannot reach the vulnerability, then it will *surely* miss the vulnerability. Because of the critical nature of crawling, we have included several types of crawling challenges in WackoPicko, some of which hide vulnerabilities.

HTML Parsing. Malformed HTML makes it difficult for web application scanners to crawl web sites. For instance, a crawler must be able to navigate HTML frames and be able to upload a file. Even though these tasks are straightforward for a human user with a regular browser, they represent a challenge for crawlers.

Multi-Step Process. Even though most web sites are built on top of the stateless HTTP protocol, a variety of techniques are utilized to introduce state into web applications. In order to properly analyze a web site, web application vulnerability scanners must be able to understand the state-based transactions that take place. In WackoPicko, there are several state-based interactions.

Table 1. Characteristics of the scanners evaluated

Name	Version Used	License	Type	Price
Acunetix	6.1 Build 20090318	Commercial	Standalone	$4,995-$6,350
AppScan	7.8.0.0 iFix001 Build: 570 Security Rules Version 647	Commercial	Standalone	$17,550-$32,500
Burp	1.2	Commercial	Proxy	£125 ($190.82)
Grendel-Scan	1.0	GPLv3	Standalone	N/A
Hailstorm	5.7 Build 3926	Commercial	Standalone	$10,000
Milescan	1.4	Commercial	Proxy	$495-$1,495
N-Stalker	2009 - Build 7.0.0.207	Commercial	Standalone	$899-$6,299
NTOSpider	3.2.067	Commercial	Standalone	$10,000
Paros	3.2.13	Clarified Artistic License	Proxy	N/A
w3af	1.0-rc2	GPLv2	Standalone	N/A
Webinspect	7.7.869.0	Commercial	Standalone	$6,000-$30,000

Infinite Web Site. It is often the case that some dynamically-generated content will create a very large (possibly infinite) crawling space. For example, WackoPicko has the ability to display a daily calendar. Each page of the calendar displays the agenda for a given day and links to the page for the following day. A crawler that naively followed the links in the WackoPicko's calendar would end up trying to visit an infinite sequence of pages, all generated dynamically by the same component.

Authentication. One feature that is common to most web sites is an authentication mechanism. Because this is so prevalent, scanners must properly handle authentication, possibly by creating accounts, logging in with valid credentials, and recognizing actions that log the crawler out. WackoPicko includes a registration and login system to test the scanner's crawlers ability to handle the authentication process correctly.

Client-side Code. Being able to parse and understand client-side technologies presents a major challenge for web application vulnerability scanners. WackoPicko includes vulnerabilities behind a JavaScript-created form, as well as behind a Flash application.

Link Extraction. We also tested the scanners on WIVET, an open-source benchmark for web link extractors [12]. WIVET contains 54 tests and assigns a final score to a crawler based on the percent of tests that it passes. The tests require scanners to analyze simple links, multi-page forms, links in comments and JavaScript actions on a variety of HTML elements. There are also AJAX-based tests as well as Flash-based tests. In our tests, we used WIVET version number 129.

4 Experimental Evaluation

We tested 11 web application scanners by running them on our WackoPicko web site. The tested scanners included 8 proprietary tools and 3 open source programs. Their cost ranges from free to tens of thousands of dollars. We used evaluation versions of each software, however they were fully functional. A summary of the characteristics of the scanners we evaluated is given in Table 1.

We ran the WackoPicko web application on a typical LAMP machine, with Apache 2.2.9, PHP 5.2.6, and MySQL 5.0.67. We enabled the `allow_url_fopen` PHP option and disabled the `allow_url_include` and `magic_quotes` options. We ran

Table 2. Detection results. For each scanner, the simplest configuration that detected a vulnerability is given. Empty cells indicate no detection in any mode.

Name	Reflected XSS	Stored XSS	Reflected SQL Injection	Command-line Injection	File Inclusion	File Exposure	XSS via JavaScript	XSS via Flash
Acunetix	INITIAL	INITIAL	INITIAL		INITIAL	INITIAL	INITIAL	
AppScan	INITIAL	INITIAL	INITIAL		INITIAL	INITIAL		
Burp	INITIAL	MANUAL	INITIAL	INITIAL		INITIAL		MANUAL
Grendel-Scan	MANUAL		CONFIG					
Hailstorm	INITIAL	CONFIG	CONFIG					MANUAL
Milescan	INITIAL	MANUAL	CONFIG					
N-Stalker	INITIAL	MANUAL	MANUAL			INITIAL	INITIAL	MANUAL
NTOSpider	INITIAL	INITIAL	INITIAL					
Paros	INITIAL	INITIAL	CONFIG					MANUAL
w3af	INITIAL	MANUAL	INITIAL		INITIAL			MANUAL
Webinspect	INITIAL	INITIAL	INITIAL		INITIAL		INITIAL	MANUAL

the scanners on a machine with a Pentium 4 3.6GHz CPU, 1024 MB of RAM, and Microsoft Windows XP, Service Pack 2.

4.1 Setup

The WackoPicko server used in testing the web vulnerability scanners was run in a virtual machine, so that before each test run the server could be put in an identical initial state. This state included ten regular users, nine pictures, and five administrator users.

Each scanner was run in three different configuration modes against WackoPicko, with each configuration requiring more setup on the part of the user. In all configuration styles, the default values for configuration parameters were used, and when choices were required, sensible values were chosen. In the INITIAL configuration mode, the scanner was directed to the initial page of WackoPicko and told to scan for all vulnerabilities. In the CONFIG setup, the scanner was given a valid username/password combination or login macro before scanning. MANUAL configuration required the most work on the part of the user; each scanner was put into a "proxy" mode and then the user browsed to each vulnerable page accessible without credentials; then, the user logged in and visited each vulnerability that required a login. Additionally a picture was uploaded, the rights to a high-quality version of a picture were purchased, and a coupon was applied to the order. The scanner was then asked to scan the WackoPicko web site.

4.2 Detection Results

The results of running the scanners against the WackoPicko site are shown in Table 2 and, graphically, in Figure 1. The values in the table correspond to the simplest configuration that discovered the vulnerability. An empty cell indicates that the given scanner did not discover the vulnerability in any mode. The table only reports the vulnerabilities that were detected by at least one scanner. Further analysis of why the scanners missed certain vulnerabilities is contained in Sections 4.3 and 4.4.

The running time of the scanners is shown in Figure 3. These times range from 74 seconds for the fastest tool (Burp) to 6 hours (N-Stalker). The majority of the scanners completed the scan within a half hour, which is acceptable for an automated tool.

4.2.1 False Negatives

One of the benefits of developing the WackoPicko web application to test the scanners is the ability for us to measure the false negatives of the scanners. An ideal scanner would be able to detect all vulnerabilities. In fact, we had a group composed of students with average security skills analyze WackoPicko. The students found all vulnerabilities except for the forceful browsing vulnerability. The automated scanners did not do as well; there were a number of vulnerabilities that were not detected by any scanner. These vulnerabilities are discussed hereinafter.

Session ID. No scanner was able to detect the session ID vulnerability on the admin login page. The vulnerability was not detected because the scanners were not given a valid username/password combination for the admin interface. This is consistent with what would happen when scanning a typical application, as the administration interface would include powerful functionality that the scanner should not invoke, like view, create, edit or delete sensitive user data. The session ID was only set on a successful login, which is why this vulnerability was not detected by any scanner.

Weak Password. Even though the scanners were not given a valid username/password combination for the administrator web site, an administrator account with the combination of admin/admin was present on the system. NTOSpider was the only scanner that successfully logged in with the admin/admin combination. However, it did not report it as an error, which suggests that it was unable to detect that the login was successful, even though the response that was returned for this request was different from every other login attempt.

Parameter Manipulation. The parameter manipulation vulnerability was not discovered by any scanner. There were two causes for this: first, only three of the scanners (AppScan, NTOSpider, and w3af) input a different number than the default value "1" to the `userid` parameter. Of the three, only NTOSpider used a value that successfully manipulated the `userid` parameter. The other reason was that in order to successfully detect a parameter manipulation vulnerability, the scanner needs to determine which pages require a valid username/password to access and which ones do not and it is clear that none of the scanners make this determination.

Stored SQL Injection. The stored SQL injection was also not discovered by any scanners, due to the fact that a scanner must create an account to discover the stored SQL injection. The reasons for this are discussed in more detail in Section 4.4.4.

Directory Traversal. The directory traversal vulnerability was also not discovered by any of the scanners. This failure is caused by the scanners being unable to upload a picture. We discuss this issue in Section 4.4.2, when we analyze how each of the scanners behaved when they had to upload a picture.

Multi-Step Stored XSS. The stored XSS vulnerability that required a confirmation step was also missed by every scanner. In Section 4.4.5, we analyze how many of the scanners were able to successfully create a comment on a picture.

Forceful Browsing. No scanner found the forceful browsing vulnerability, which is not surprising since it is an application-specific vulnerability. These vulnerabilities are difficult to identify without access to the source code of the application [2].

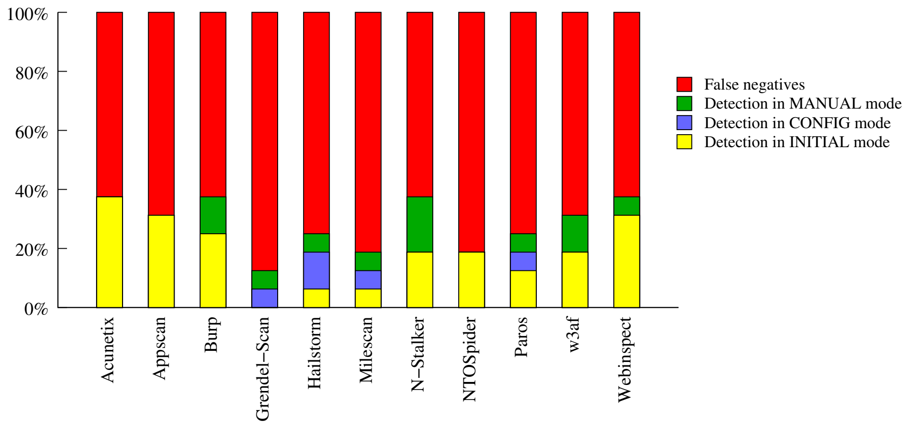


Fig. 1. Detection performance (true positives and false negatives) of the evaluated scanners

Table 3. False positives

Name	INITIAL	CONFIG	MANUAL
Acunetix	1	7	4
AppScan	11	20	26
Burp	1	2	6
Grendel-Scan	15	16	16
Hailstorm	3	11	3
Milescan	0	0	0
N-Stalker	5	0	0
NTOSpider	3	1	3
Paros	1	1	1
w3af	1	1	9
Webinspect	215	317	297

Logic Flaw. Another vulnerability that none of the scanners uncovered was the logic flaw that existed in the coupon management functionality. Also in this case, some domain knowledge about the application is needed to find the vulnerability.

4.2.2 False Positives

The total number of false positives for each of the scanning configurations are show in Table 3. The number of false positives that each scanner generates is an important metric, because the greater the number of false positives, the less useful the tool is to the end user, who has to figure out which of the vulnerabilities reported are actual flaws and which are spurious results of the analysis.

The majority of the false positives across all scanners were due to a supposed "Server Path Disclosure." This is an information leakage vulnerability where the server leaks the paths of local files, which might give an attacker hints about the structure of the file system.

An analysis of the results identified two main reasons why these false positives were generated. The first is that while testing the application for file traversal or file injection vulnerabilities, some of the scanners passed parameters with values of file names, which, on some pages (e.g., the guestbook page), caused the file name to be

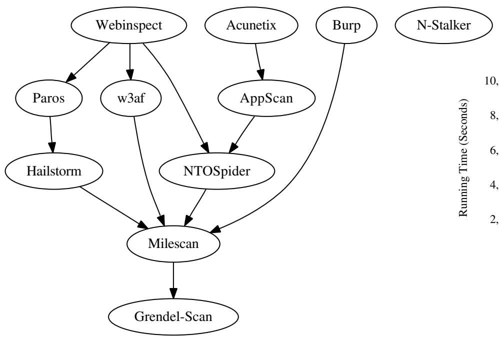


Fig. 2. Dominates graph

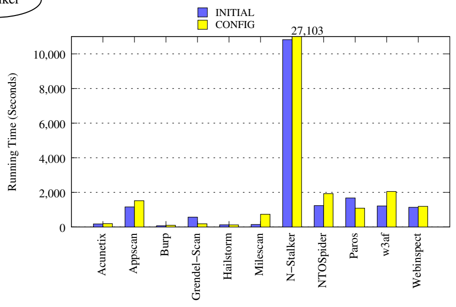


Fig. 3. Scanner Running Times

included in that page's contents. When the scanner then tested the page for a Server Path Disclosure, it found the injected values in the page content, and generated a Server Path Disclosure vulnerability report. The other reason for the generation of false positives is that WackoPicko uses absolute paths in the `href` attribute of anchors (e.g., `/users/home.php`), which the scanner mistook for the disclosure of paths in the local system. Webinspect generated false positives because of both the above reasons, which explains the large amount of false positives produced by the tool.

Some scanners reported genuine false positives: Hailstorm reported a false XSS vulnerability and two false PHP code injection vulnerabilities, NTOSpider reported three false XSS vulnerabilities and w3af reported a false PHP `eval()` input injection vulnerability.

4.2.3 Measuring and Comparing Detection Capabilities

Comparing the scanners using a single benchmark like WackoPicko does not represent an exhaustive evaluation. However, we believe that the results provide insights about the current state of black-box web application vulnerability scanners.

One possible way of comparing the results of the scanners is arranging them in a lattice. This lattice is ordered on the basis of *strict dominance*. Scanner A *strictly dominates* Scanner B if and only if for every vulnerability discovered by Scanner B, Scanner A discovered that vulnerability with the same configuration level or simpler, and Scanner A either discovered a vulnerability that Scanner B did not discover or Scanner A discovered a vulnerability that Scanner B discovered, but with a simpler configuration. *Strictly dominates* has the property that any assignment of scores to vulnerabilities must preserve the *strictly dominates* relationship.

Figure 2 shows the *strictly dominates* graph for the scanners, where a directed edge from Scanner A to Scanner B means that Scanner A *strictly dominates* Scanner B. Because *strictly dominates* is transitive, if one scanner *strictly dominates* another it also *strictly dominates* all the scanners that the dominated scanner dominates, therefore, all redundant edges are not included. Figure 2 is organized so that the scanners in the top level are those that are not *strictly dominated* by any scanners. Those in the second level are strictly dominated by only one scanner and so on, until the last level, which contains those scanners that strictly dominate no other scanner.

Table 4. Vulnerability scores

Name	Detection	INITIAL Reachability	CONFIG Reachability	MANUAL Reachability
XSS Reflected	1	0	0	0
XSS Stored	2	0	0	0
SessionID	4	0	0	0
SQL Injection Reflected	1	0	0	0
Commandline Injection	4	0	0	0
File Inclusion	3	0	0	0
File Exposure	3	0	0	0
XSS Reflected behind JavaScript	1	3	3	0
Parameter Manipulation	8	0	0	0
Weak password	3	0	0	0
SQL Injection Stored Login	7	7	3	3
Directory Traversal Login	8	8	6	4
XSS Stored Login	2	8	7	6
Forceful Browsing Login	8	7	6	3
Logic Flaws - Coupon	9	9	8	6
XSS Reflected behind flash	1	9	7	1

Table 5. Final ranking

Name	Score
Acunetix	14
Webinspect	13
Burp	13
N-Stalker	13
AppScan	10
w3af	9
Paros	6
Hailstorm	6
NTOSpider	4
Milescan	4
Grendel-Scan	3

Some interesting observations arise from Figure 2. N-Stalker does not strictly dominate any scanner and no scanner strictly dominates it. This is due to the unique combination of vulnerabilities that N-Stalker discovered and missed. Burp is also interesting due to the fact that it only dominates two scanners but no scanner dominates Burp because it was the only scanner to discover the command-line injection vulnerability.

While Figure 2 is interesting, it does not give a way to compare two scanners where one does not strictly dominate the other. In order to compare the scanners, we assigned scores to each vulnerability present in WackoPicko. The scores are displayed in Table 4. The "Detection" score column in Table 4 is how many points a scanner is awarded based on how difficult it is for an automated tool to detect the existence of the vulnerability. In addition to the "Detection" score, each vulnerability is assigned a "Reachability" score, which indicates how difficult the vulnerability is to reach (i.e., it reflects the difficulty of crawling to the page that contains the vulnerability). There are three "Reachability" scores for each vulnerability, corresponding to how difficult it is for a scanner to reach the vulnerability when run in INITIAL, CONFIG, or MANUAL mode. Of course, these vulnerability scores are subjective and depend on the specific characteristics of our WackoPicko application. However, their values try to estimate the crawling and detection difficulty of each vulnerability in this context.

The final score for each scanner is calculated by adding up the "Detection" score for each vulnerability the scanner detected and the "Reachability" score for the configuration (INITIAL, CONFIG and MANUAL) used when running the scanner. In the case of a tie, the scanners were ranked by how many vulnerabilities were discovered in INITIAL mode, which was enough to break all ties. Table 5 shows the final ranking of the scanners.

4.3 Attack and Analysis Capabilities

Analyzing how each scanner attempted to detect vulnerabilities gives us insight into how these programs work and illuminates areas for further research. First, the scanner would crawl the site looking for injection points, typically in the form of GET or POST parameters. Once the scanner identifies all the inputs on a page, it then attempts to inject values for each parameter and observes the response. When a page has more than one input, each parameter is injected in turn, and generally no two parameters are injected in the same request. However, scanners differ in what they supply as values of the non-injected parameters: some have a default value like `1234` or `Peter Wiener`, while others leave the fields blank. This has an impact on the results of the scanner, for example the WackoPicko guestbook requires that both the `name` and `comment` fields are present before making a comment, and thus the strategy employed by each scanner can affect the effectiveness of the vulnerability scanning process.

When detecting XSS attacks, most scanners employed similar techniques, some with a more sophisticated attempt to evade possible filters than others. One particularly effective strategy employed was to first input random data with various combinations of dangerous characters, such as `/ ,",',<, and >`, and then, if one of these combinations was found unchanged in the response, to attempt the injection of the full range of XSS attacks. This technique speeds up the analysis significantly, because the full XSS attack is not attempted against every input vector. Differently, some of the scanners took an exhaustive approach, attempting the full gamut of attacks on every combination of inputs.

When attempting a XSS attack, the thorough scanners would inject the typical `<script> alert('xss') </script>` as well as a whole range of XSS attack strings, such as JavaScript in a tag with the `onmouseover` attribute, in an `img`, `div` or `meta` tag, or `iframe`. Other scanners attempted to evade filters by using a different JavaScript function other than `alert`, or by using a different casing of `script`, such as `ScRiPt`.

Unlike with XSS, scanners could not perform an easy test to exclude a parameter from thorough testing for other Unsanitized Input vulnerabilities because the results of a successful exploit might not be readily evident in the response. This is true for the command-line injection on the WackoPicko site, because the output of the injectable command was not used in the response. Burp, the only scanner that was able to successfully detect the command line injection vulnerability, did so by injecting `` `ping -c 100 localhost` `` and noticing that the response time for the page was much slower than when nothing was injected.

This pattern of measuring the difference in response times was also seen in detecting SQL injections. In addition to injecting something with a SQL control character, such as tick or quote and seeing if an error is generated, the scanners also used a time-delay SQL injection, inputting `waitfor delay '0:0:20'` and seeing if the execution was delayed. This is a variation of the technique of using time-delay SQL injection to extract database information from a blind SQL vulnerability.

When testing for File Exposure, the scanners were typically the same; however one aspect caused them to miss the WackoPicko vulnerability. Each scanner that was looking for this vulnerability input the name of a file that they knew existed on the system, such as `/etc/passwd` on UNIX-like systems or `C:\boot.ini` for Windows. The scanners then looked for known strings in the response. The difficulty in exploiting the WackoPicko file exposure was including the null-terminating character (`%00`) at the

Table 6. Number of accesses to vulnerable web pages in INITIAL, CONFIG, and MANUAL modes

Name	Reflected XSS			Stored XSS			Reflected SQL Injection			Command-line Injection			File Inclusion / File Exposure / Weak password			XSS Reflected - JavaScript		
	INITIAL	CONFIG	MANUAL															
Acunetix	496	638	498	613	779	724	544	709	546	495	637	497	198	244	200	670	860	671
AppScan	581	575	817	381	352	492	274	933	628	189	191	288	267	258	430	0	0	442
Burp	256	256	207	192	192	262	68	222	221	68	68	200	125	316	320	0	0	178
Grendel-Scan	0	0	44	1	1	3	14	34	44	1	1	3	2	2	5	0	0	2
Hailstorm	232	229	233	10	205	209	45	224	231	180	160	162	8	204	216	153	147	148
Milescan	104	0	208	50	0	170	75	272	1237	0	0	131	80	0	246	0	0	163
N-Stalker	1738	1162	2689	2484	2100	3475	2764	1022	2110	2005	1894	1987	1437	2063	1824	1409	1292	1335
NTOSpider	856	679	692	252	370	370	184	5	5	105	9	9	243	614	614	11	13	13
Paros	68	68	58	126	126	110	151	299	97	28	28	72	146	146	185	0	0	56
w3af	157	157	563	259	257	464	1377	1411	2634	140	142	253	263	262	470	0	0	34
Webinspect	108	108	105	631	631	630	297	403	346	164	164	164	239	237	234	909	909	0
Name	Parameter Manipulation			Directory Traversal			Logic Flaw			Forceful Browsing			XSS Reflected behind flash					
Acunetix	2	0	2	35	1149	37	0	0	5	0	0	206	1	34	458			
AppScan	221	210	222	80	70	941	0	0	329	0	0	71	0	0	243			
Burp	192	194	124	68	68	394	0	0	314	0	0	151	0	0	125			
Grendel-Scan	3	3	6	1	1	3	0	0	6	0	0	1	0	0	3			
Hailstorm	3	143	146	336	329	344	131	132	5	102	102	105	0	0	143			
Milescan	105	0	103	8	0	163	0	0	1	0	0	60	0	0	68			
N-Stalker	1291	1270	1302	22	2079	4704	0	0	3	0	0	2	0	0	1315			
NTOSpider	107	115	115	11	572	572	0	11	11	0	0	0	0	11	11			
Paros	72	72	72	14	14	0	0	0	114	0	0	70	0	0	60			
w3af	128	128	124	31	30	783	0	0	235	0	0	270	0	0	119			
Webinspect	102	102	102	29	29	690	0	8	3	0	118	82	0	0	97			

end of the string, which caused PHP to ignore anything added by the application after the `/etc/passwd` part. The results show that only 4 scanners successfully discovered this vulnerability.

The remote code execution vulnerability in WackoPicko is similar to the file exposure vulnerability. However, instead of injecting known files, the scanners injected known web site addresses. This was typically from a domain the scanner's developers owned, and thus when successfully exploited, the injected page appeared instead of the regular page. The same difficulty in a successful exploitation existed in the File Exposure vulnerability, so a scanner had to add `%00` after the injected web site. Only 3 scanners were able to successfully identify this vulnerability.

4.4 Crawling Capabilities

The number of URLs requested and accessed varies considerably among scanners, depending on the capability and strategies implemented in the crawler and attack components. Table 6 shows the number of times each scanner made a `POST` or `GET` request to a vulnerable URL when the scanners were run in INITIAL, CONFIG, and MANUAL mode. For instance, from Table 6 we can see that Hailstorm was able to access many of the vulnerable pages that required a valid username/password when run in INITIAL mode. It can also be seen that N-Stalker takes a shotgun-like approach to scanning; it has over 1,000 accesses for each vulnerable URL, while in contrast Grendel-Scan never had over 50 accesses to a vulnerable URL.

In the following, we discuss the main challenges that the crawler components of the web application scanners under test faced.

4.4.1 HTML

The results for the stored XSS attack reveal some interesting characteristics of the analysis performed by the various scanners. For instance, Burp, Grendel-Scan, Hailstorm, Milescan, N-Stalker, and w3af were unable to discover the stored XSS vulnerability in INITIAL configuration mode. Burp and N-Stalker failed because of defective HTML parsing. Neither of the scanners correctly interpreted the `<textarea>` tag as an input to the HTML form. This was evident because both scanners only sent the `name` parameter when attempting to leave a comment on the guestbook. When run in MANUAL mode, however, the scanners discovered the vulnerability, because the user provided values for all these fields. Grendel-Scan and Milescan missed the stored XSS vulnerability for the same reason: they did not attempt a `POST` request unless the user used the proxy to make the request.

Hailstorm did not try to inject any values to the guestbook when in INITIAL mode, and, instead, used `testval` as the `name` parameter and `Default text` as the `comment` parameter. One explanation for this could be that Hailstorm was run in the default "turbo" mode, which Cenzic claims catches 95% of vulnerabilities, and chose not to fuzz the form to improve speed.

Finally, w3af missed the stored XSS vulnerability due to leaving one parameter blank while attempting to inject the other parameter. It was unable to create a guestbook entry, because both parameters are required.

4.4.2 Uploading a Picture

Being able to upload a picture is critical to discover the Directory Traversal vulnerability, as a properly crafted `tag` parameter can overwrite any file the web server can access. It was very difficult for the scanners to successfully upload a file: no scanner was able to upload a picture in INITIAL and CONFIG modes, and only AppScan and Webinspect were able to upload a picture after being showed how to do it in MANUAL configuration, with AppScan and Webinspect uploading 324 and 166 pictures respectively. Interestingly, Hailstorm, N-Stalker and NTOSpider never successfully uploaded a picture, even in MANUAL configuration. This surprising result is due to poor proxies or poor in-application browsers. For instance, Hailstorm includes an embedded Mozilla browser for the user to browse the site when they want to do so manually, and after repeated attempts the embedded browser was never able to upload a file. The other scanners that failed, N-Stalker and NTOSpider, had faulty HTTP proxies that did not know how to properly forward the file uploaded, thus the request never completed successfully.

4.4.3 Client-Side Code

The results of the WIVET tests are shown in Figure 4. Analyzing the WIVET results gives a very good idea of the JavaScript capabilities of each scanner. Of all the 54 WIVET tests, 24 required actually executing or understand JavaScript code; that is, the test could not be passed simply by using a regular expression to extract the links on the page. Webinspect was the only scanner able to complete all of the dynamic JavaScript challenges. Of the rest of the scanners, Acunetix and NTOSpider only missed one of the dynamic JavaScript tests. Even though Hailstorm missed 12 of the dynamic JavaScript tests, we believe that this is because of a bug in the JavaScript analysis engine and not a general limitation of the tool. In fact, Hailstorm was able to correctly handle JavaScript on the `onmouseup` and `onclick` parametrized functions. These tests were on parametrized `onmouseout` and `onmousedown` functions, but since Hailstorm was able to correctly handle the `onmouseup` and `onclick` parametrized functions, this can be considered a bug in Hailstorm's JavaScript parsing. From this, it can also be concluded that AppScan, Grendel-Scan, Milescan, and w3af perform no dynamic JavaScript parsing. Thus, Webinspect, Acunetix, NTOSpider, and Hailstorm can be claimed to have the best JavaScript parsing. The fact that N-Stalker found the reflected XSS vulnerability behind a JavaScript form in WackoPicko suggests that it can execute JavaScript, however it failed the WIVET benchmark so we cannot evaluate the extent of the parsing performed.

In looking at the WIVET results, there was one benchmark that no scanner was able to reach, which was behind a Flash application. The application had a link on a button's `onclick` event, however this link was dynamically created at run time. This failure shows that none of the current scanners processes Flash content with the same level of sophistication as JavaScript. This conclusion is supported by none of the scanners discovering the XSS vulnerability behind a Flash application in WackoPicko when in INITIAL or CONFIG mode.

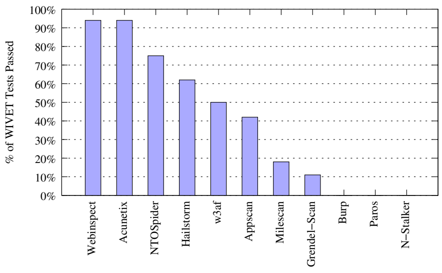


Fig. 4. WIVET results

4.4.4 Authentication

Table 7 shows the attempts that were made to create an account on the WackoPicko site. The Name column is the name of the scanner, "Successful" is the number of accounts successfully created, and "Error" is the number of account creation attempts that were unsuccessful. Note that Table 7 reports the results of the scanners when run in INITIAL mode only, because the results for the other configurations were almost identical.

Table 7. Account creation

Name	Successful	Error
Acunetix	0	431
AppScan	1	297
Burp	0	0
Grendel-Scan	0	0
Hailstorm	107	276
Milescan	0	0
N-Stalker	74	1389
NTOSpider	74	330
Paros	0	176
w3af	0	538
Webinspect	127	267

Table 7 shows the capability of the scanners to handle user registration functionality. As can be seen from Table 7, only five of the scanners were able to successfully create an account. Of these, Hailstorm was the only one to leverage this ability to visit vulnerable URLs that required a login in its INITIAL run.

Creating an account is important in discovering the stored SQL injection that no scanner successfully detected. It is fairly telling that even though five scanners were able to create an account, none of them detected the vulnerability. It is entirely possible that none of the scanners actively searched for stored SQL injections, which is much harder to detect than stored XSS injections.

In addition to being critically important to the WackoPicko benchmark, being able to create an account is an important skill for a scanner to have when analyzing any web site, especially if that scanner wishes to be a point-and-click web application vulnerability scanner.

4.4.5 Multi-step Processes

In the WackoPicko web site there is a vulnerability that is triggered by going through a multi-step process. This vulnerability is the stored XSS on pictures, which requires an attacker to confirm a comment posting for the attack to be successful. Hailstorm and NTOSpider were the only scanners to successfully create a comment on the INITIAL run (creating 25 and 1 comment, respectively). This is important for two reasons: first, to be able to create a comment in the INITIAL run, the scanner had to create an account and log in with that account, which is consistent with Table 7. Also, all 25 of the comments successfully created by Hailstorm only contained the text `Default text`, which means that Hailstorm was not able to create a comment that exploited the vulnerability.

All scanners were able to create a comment when run in MANUAL configuration, since they were shown by the user how to carry out this task. However, only AppScan, Hailstorm, NTOSpider, and Webinspect (creating 6, 21, 7, and 2 comments respectively) were able to create a comment that was different than the one provided by the user. Of these scanners only Webinspect was able to create a comment that exploited the vulnerability, `<iFrAmE sRc=hTtP://xSrFtEsT .sPi/> </iFrAmE>`, however Webinspect failed to report this vulnerability. One plausible explanation for not detecting would be the scanners' XSS strategy discussed in Section 4.3. While testing the `text` parameter for a vulnerability, most of the scanners realized that it was properly escaped on the preview page, and thus stopped trying to inject XSS attacks. This would explain the directory traversal attack comment that AppScan successfully created and why Hailstorm did not attempt any injection. This is an example where the performance optimization of the vulnerability analysis can lead to false negatives.

4.4.6 Infinite Web Sites

One of the scanners attempted to visit all of the pages of the infinite calendar. When running Grendel-Scan, the calendar portion of WackoPicko had to be removed because the scanner ran out of memory attempting to access every page. Acunetix, Burp, N-Stalker and w3af had the largest accesses (474, 691, 1780 and 3094 respectively), due to their attempts to exploit the calendar page. The other scanners used less accesses (between 27 and 243) because they were able to determine that no error was present.

5 Lessons Learned

We found that the crawling of modern web applications can be a serious challenge for today's web vulnerability scanners. A first class of problems we encountered consisted of implementation errors and the lack of support for commonly-used technologies. For example, handling of multimedia data (image uploads) exposed bugs in certain proxy-based scanners, which prevented the tools from delivering attacks to the application under test. Incomplete or incorrect HTML parsers caused scanners to ignore input vectors that would have exposed vulnerabilities. The lack of support for JavaScript (and Flash) prevented tools from reaching vulnerable pages altogether. *Support for well-known, pervasive technology should be improved.*

The second class of problems that hindered crawling is related to the design of modern web applications. In particular, applications with complex forms and aggressive checking of input values can effectively block a scanner, preventing it from crawling the pages "deep" in the web site structure. Handling this problem could be done, for example, by using heuristics to identify acceptable inputs or by reverse engineering the input filters. Furthermore, the behavior of an application can be wildly different depending on its internal "state," i.e., the values of internal variables that are not explicitly exposed to the scanner. The classic example of application state is whether the current user is logged in or not. A scanner that does not correctly model and track the state of an application (e.g., it does not realize that it has been automatically logged out) will fail to crawl all relevant parts of the application. *More sophisticated algorithms are needed to perform "deep" crawling and track the state of the application under test.*

Current scanners fail to detect (or even check for) application-specific (or "logic") vulnerabilities. Unfortunately, as applications become more complex, this type of vulnerabilities will also become more prevalent. *More research is warranted to automate the detection of application logic vulnerabilities.*

In conclusion, far from being point-and-click tools to be used by anybody, web application black-box security scanners require a sophisticated understanding of the application under test and of the limitations of the tool, in order to be effective.

6 Related Work

Our work is related to two main areas of research: the design of web applications for assessing vulnerability analysis tools and the evaluation of web scanners.

Designing test web applications. Vulnerable test applications are required to assess web vulnerability scanners. Unfortunately, no standard test suite is currently available or accepted by the industry. HacmeBank [5] and WebGoat [11] are two well-known, publicly-available, vulnerable web applications, but their design is focused more on teaching web application security rather than testing automated scanners. SiteGenerator [10] is a tool to generate sites with certain characteristics (e.g., classes of vulnerabilities) according to its input configuration. While SiteGenerator is useful to automatically produce different vulnerable sites, we found it easier to manually introduce in WackoPicko the vulnerabilities with the characteristics that we wanted to test.

Evaluating web vulnerability scanners. There exists a growing body of literature on the evaluation of web vulnerability scanners. For example, Suto compared three scanners against three different applications and used code coverage, among other metrics, as a measure of the effectiveness of each scanner [21]. In a recent follow-up study, Suto [22] assessed seven scanners and compared their detection capabilities and the time required to run them. Wiegenstein et al. ran five unnamed scanners against a custom benchmark [24]. Unfortunately, the authors do not discuss in detail the reasons for detections or spidering failures. In their survey of web security assessment tools, Curphey and Araujo reported that black-box scanners perform poorly [3]. Peine examined in depth the functionality and user interfaces of seven scanners (three commercial) that were run against WebGoat and one real-world application [16]. Kals et al. developed a new web vulnerability scanner and tested it on about 25,000 live web pages [7]. Since no ground truth is available for these sites, the authors cannot discuss false negative rate or failures of their tool. More recently, AnantaSec released an evaluation of three scanners against 13 real-world applications, three web applications provided by the scanners vendors, and a series of JavaScript tests [1]. While this experiment assesses a large number of real-world applications, only a limited number of scanners are tested and no explanation is given for the results. In addition, Vieira et al. tested four web scanners on 300 web services [23]. They also report high rates of false positives and false negatives.

In comparison, our work, to the best of our knowledge, performs the largest evaluation of web application scanners in terms of the number of tested tools (eleven, both commercial and open-source), and the class of vulnerabilities analyzed. In addition, we discuss the effectiveness of different configurations and levels of manual intervention, and examine in detail the reasons for a scanner's success or failure.

Furthermore, we provide a discussion of challenges (i.e., critical limitations) of current web vulnerability scanners. While some of these problem areas were discussed before [6,8], we provide quantitative evidence that these issues are actually limiting the performance of today's tools. We believe that this discussion will provide useful insight into how to improve state-of-the-art of black-box web vulnerability scanners.

7 Conclusions

This paper presented the evaluation of eleven black-box web vulnerability scanners. The results of the evaluation clearly show that the ability to crawl a web application and reach "deep" into the application's resources is as important as the ability to detect the vulnerabilities themselves.

It is also clear that although techniques to detect certain kinds of vulnerabilities are well-established and seem to work reliably, there are whole classes of vulnerabilities that are not well-understood and cannot be detected by the state-of-the-art scanners. We found that eight out of sixteen vulnerabilities were not detected by *any* of the scanners.

We have also found areas that require further research so that web application vulnerability scanners can improve their detection of vulnerabilities. Deep crawling is vital to discover all vulnerabilities in an application. Improved reverse engineering is necessary to keep track of the state of the application, which can enable automated detection of complex vulnerabilities.

Finally, we found that there is no strong correlation between cost of the scanner and functionality provided as some of the free or very cost-effective scanners performed as well as scanners that cost thousands of dollars.

Acknowledgments

This work has been supported by the National Science Foundation, under grants CCR-0524853, CCR-0716095, CCR-0831408, CNS-0845559 and CNS-0905537, and by the ONR under grant N000140911042.

References

1. AnantaSec: Web Vulnerability Scanners Evaluation (January 2009), http://anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html

2. Balzarotti, D., Cova, M., Felmetsger, V., Vigna, G.: Multi-module Vulnerability Analysis of Web-based Applications. In: Proceedings of the ACM conference on Computer and Communications Security (CCS), pp. 25–35 (2007)
3. Curphey, M., Araujo, R.: Web Application Security Assessment Tools. IEEE Security and Privacy 4(4), 32–41 (2006)
4. CVE: Common Vulnerabilities and Exposures, http://www.cve.mitre.org
5. Foundstone: Hacme Bank v2.0 (May 2006), http://www.foundstone.com/us/resources/proddesc/hacmebank.htm
6. Grossman, J.: Challenges of Automated Web Application Scanning. In: BlackHat Windows Security Conference (2004)
7. Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: SecuBat: A Web Vulnerability Scanner. In: Proceedings of the International World Wide Web Conference (2006)
8. McAllister, S., Kruegel, C., Kirda, E.: Leveraging User Interactions for In-Depth Testing of Web Applications. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection (2008)
9. Open Security Foundation: OSF DataLossDB: Data Loss News, Statistics, and Research, http://datalossdb.org/
10. Open Web Application Security Project (OWASP): OWASP SiteGenerator, http://www.owasp.org/index.php/OWASP_SiteGenerator
11. Open Web Application Security Project (OWASP): OWASP WebGoat Project, http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project
12. Open Web Application Security Project (OWASP): Web Input Vector Extractor Teaser, http://code.google.com/p/wivet/
13. Open Web Application Security Project (OWASP): OWASP Top Ten Project (2010), http://www.owasp.org/index.php/Top_10
14. OpenID Foundation: OpenID, http://openid.net/
15. PCI Security Standards Council: PCI DDS Requirements and Security Assessment Procedures, v1.2 (October 2008)
16. Peine, H.: Security Test Tools for Web Applications. Tech. Rep. 048.06, Fraunhofer IESE (January 2006)
17. Provos, N., Mavrommatis, P., Rajab, M., Monrose, F.: All Your iFRAMEs Point to Us. In: Proceedings of the USENIX Security Symposium, pp. 1–16 (2008)
18. RSnake: Sql injection cheat sheet, http://ha.ckers.org/sqlinjection/
19. RSnake: XSS (Cross Site Scripting) Cheat Sheet, http://ha.ckers.org/xss.html
20. Small, S., Mason, J., Monrose, F., Provos, N., Stubblefield, A.: To Catch a Predator: A Natural Language Approach for Eliciting Malicious Payloads. In: Proceedings of the USENIX Security Symposium (2008)
21. Suto, L.: Analyzing the Effectiveness and Coverage of Web Application Security Scanners (October 2007) (case Study)
22. Suto, L.: Analyzing the Accuracy and Time Costs of Web Application Security Scanners (Feburary 2010)
23. Vieira, M., Antunes, N., Madeira, H.: Using Web Security Scanners to Detect Vulnerabilities in Web Services. In: Proceedings of the Conference on Dependable Systems and Networks (2009)
24. Wiegenstein, A., Weidemann, F., Schumacher, M., Schinzel, S.: Web Application Vulnerability Scanners—a Benchmark. Tech. rep., Virtual Forge GmbH (October 2006)

Organizing Large Scale Hacking Competitions

Nicholas Childers, Bryce Boe, Lorenzo Cavallaro, Ludovico Cavedon, Marco Cova, Manuel Egele, and Giovanni Vigna

Security Group
Department of Computer Science
University of California, Santa Barbara
{voltaire,bboe,sullivan,cavedon,marco,manuel,vigna}@cs.ucsb.edu

Abstract. Computer security competitions and challenges are a way to foster innovation and educate students in a highly-motivating setting. In recent years, a number of different security competitions and challenges were carried out, each with different characteristics, configurations, and goals. From 2003 to 2007, we carried out a number of live security exercises involving dozens of universities from around the world. These exercises were designed as "traditional" Capture The Flag competitions, where teams both attacked and defended a virtualized host, which provided several vulnerable services. In 2008 and 2009, we introduced two completely new types of competition: a security "treasure hunt" and a botnet-inspired competition. These two competitions, to date, represent the largest live security exercises ever attempted and involved hundreds of students across the globe. In this paper, we describe these two new competition designs, the challenges overcome, and the lessons learned, with the goal of providing useful guidelines to other educators who want to pursue the organization of similar events.

1 Introduction

Computer security has become a major aspect of our everyday experience with the Internet. To some degree, every user of the Internet is aware of the potential harm that can come from its use. Therefore, it is unsurprising to see that computer security education has also improved substantially in the past decade, in terms of both the number of university undergraduate and graduate level courses offered and the type of educational tools used to teach security concepts. This increase in the importance of computer security is also reflected by the offerings of the job market. For example, at www.computermajors.com it is stated that while entry-level salaries for computer science professionals specializing in web development start at around $75,000, "those who specialize in computer and online security can earn up to $93,000." [2]

An important aspect of computer security education is hands-on experience. Despite the importance of foundational security classes that focus on more abstract concepts in security, such as cryptography and information theory models, Internet security issues often require substantial hands-on training in order to be understood and mastered. Thus, it is important to improve security training by providing novel approaches, which complement the existing, more traditional educational tools normally used in graduate and undergraduate courses on computer security.

A class of these tools is represented by *security competitions.* In these competitions, a number of teams (or individuals) compete against each other in some security-related challenge. As an educational tool, these competitions have both advantages and disadvantages. A notable advantage (and the main reason why these events are organized) is that competition motivates students to go beyond the normal "call of duty" and explore original approaches, sometimes requiring the development of novel tools. Another advantage is that students usually operate against a determined opponent while under strict time constraints and with limited resources thus mimicking a more realistic situation than one can reproduce using paper-and-pencil *Gedanken* experiments. Unfortunately, security competitions have one major disadvantage: they usually require a large amount of resources to design, develop, and run [10,11].

Designing security competitions is hard, as they need to be at the right level of difficulty with respect to the target audience. If a competition is too difficult, the participants become frustrated; if a competition is too easy, the participants are not challenged and will lose interest. Ideally, a competition will provide a variety of challenges of differing difficulties such that all participants of various skill levels are both challenged by the tasks and gratified by success. An additional design challenge is how to evaluate the participants and score their actions. Ideally, a scoring system is fair, relevant, and automated allowing the best participant to be clearly identified beyond any reasonable doubt. However, scoring systems must be hard to reverse-engineer and cheat, as it would be ironic to have a security competition requiring "adversarial" approaches and "oblique" reasoning that utilizes a scoring system reliant on the "good behavior" of the participants.

Developing security competitions is time consuming since a specific competition can seldom be reused. For example, consider a competition where the flaws of vulnerable applications have to be identified by the participants. Once a competition ends, it is likely that the vulnerabilities discovered will be discussed in blogs and "walk-through" pages.[1] After the disclosure of vulnerability details, these services cannot be reused and new ones must be developed.

Running a security competition is challenging, because the competition's execution environment is often hostile and thus difficult to monitor and control. Therefore, it is of paramount importance to have mechanisms and policies allowing for the containment of the participants and for the easy diagnosis of possible problems. In addition, security competitions are often limited in time, thus unexpected execution problems might make the competition unplayable, wasting weeks of preparation.

Annually, since 2003, we organized an international, wide-area security competition involving dozens of teams throughout the world. The goal of these live exercises was to test the participant's security skills in a time-constrained setting.

[1] See for example, the walk-through for the 2008 DefCon qualification challenge at `http://nopsr.us/ctf2008qual`

Our competitions were designed as educational tools, and were open to educational institutions only.

From 2003 to 2007, each edition of the competition was structured as a Capture The Flag hacking challenge, called the iCTF (further described in Section 2), where remote teams connected to a VPN and competed independently against each other, leveraging both attack and defense techniques. In this "traditional" design, borrowed from the DefCon Capture The Flag competition (CTF), teams received identical copies of a virtualized host containing a number of vulnerable services. Each team's goal was to keep the services running uncompromised while breaking the security of other teams' services. Subsequent editions of the competition grew in the number of teams participating and in the sophistication of the exploitable services. The design, however, remained substantially unchanged.

In both 2008 and 2009, we introduced completely new designs for the competition. In 2008, we created a security "treasure hunt" where 39 teams from around the world had to compromise the security of 39 dedicated, identical target networks within a limited time frame. In 2009, 56 teams participated in a competition where each team had to compromise the browsers of thousands of simulated users, compromise the users' banking accounts, and finally make the users' computers part of a botnet.

In addition to this paper's content, we provide scoring software, virtual machines and network traces from each of our previous iCTFs.[2] The software and virtual machines are useful for the creation similar competitions and the network traces are useful for researchers working on intrusion detection and correlation techniques.

This paper describes these new competition designs, how they were implemented and executed, and what lessons were learned from running them. By providing a detailed discussion of the issues and challenges overcome, as well as the mistakes made, we hope to provide guidance to other educators in the security field who want to pursue the organization of similar competitions. In summary, the contributions of this paper are the following:

- We present two novel designs for large-scale live security exercises. To the best of our knowledge, these are the largest educational security exercises carried out to date. Their design, implementation, and execution required a substantial research and engineering effort.
- We discuss the lessons learned from running these competitions. Given the amount of work necessary to organize and the current lack of documentation and analysis of such events, we think this paper provides a valuable contribution.

This paper is structured as follows. In Section 2, we present background information on both security competitions in general, and on the iCTF in particular. Section 3 presents the design of the 2008 competition and its characteristics. Section 4 describes the competition held in 2009. In Section 5, we describe the

[2] All files can be obtained at `http://ictf.cs.ucsb.edu/`

lessons learned in designing, implementing, and running these competitions. Finally, Section 6 briefly concludes.

2 Background and History

The best-known online security challenge is the DefCon CTF, which is held annually at the DefCon convention. At DefCon 2009, eight teams received an identical copy of a virtualized system containing a number of vulnerable services. Each team ran their virtual machine on a virtual private network, with the goal of maintaining uptime on and securing a set of services throughout the contest whilst concurrently compromising the other teams' services. Compromising and securing services required teams to leverage their knowledge of vulnerability detection. Compromising another team's service allowed teams to "capture the flag" thus accumulating attack points, whereas securing services allowed teams to retain flags and acquire defensive points. Several of the former DefCon CTFs followed a similar design [3].

Despite DefCon's nonspecific focus on security education, it inspired several editions of the UCSB International Capture The Flag (iCTF). One of the major differences between UCSB's iCTF and DefCon's CTF is that the iCTF involves educational institutions spread across the world, whereas the DefCon CTF allows only locally-connected teams. DefCon therefore requires the physical co-location of the contestants thus constraining participation to a limited number of teams. By not requiring contestants to be physically present, the UCSB iCTF additionally allows dozens of remotely located teams to connect to the competition network.

The UCSB iCTF editions from 2003 to 2007 were similar to the DefCon CTF in that the participants had to protect and attack a virtualized system containing a number of vulnerable services. A scoring system actively checked the state of these services, ensuring their availability. In addition, the scoring system periodically set short identification tokens, termed "flags" for each team.. Teams competed by securing their system and breaking into their competitors' systems to discover flags. The successful compromise of another team's service was demonstrated through the submission of a flag to the scoring system. Teams periodically earned defensive points for each service that retained its flags during the particular period. The team with the most points at the end of the competition won. These UCSB iCTFs in turn inspired other educational hacking competitions, such as CIPHER [6] and RuCTF [4].

Recently, a different type of competition has received a significant amount of attention. In the Pwn2Own hacking challenge [7] participants try to compromise the security of various up-to-date computer devices such as laptops, and smart phones. Whoever successfully compromises a device, wins the device itself as a prize. This competition is solely focused on attack, does not have an educational focus, and does not allow any real interaction amongst the participants who attack a single target in parallel.[3]

[3] Note that this type of design is very similar to early editions of the DefCon CTF, where participants competed in breaking into a shared target system.

Another interesting competition is the Cyber Defense Exercise (CDX) [1,5,8], in which a number of military schools compete in protecting their networks from external attackers. This competition differs from the UCSB iCTF in a number of ways. First, the competition's sole focus is defense. Second, the competition is scored in person by human evaluators who observe the activity of the participants, and score them according to their ability to react to attacks. This evaluation method is subjective and requires a human judge for each team thus yielding it impractical in a large-scale online security competition.

In both 2008 and 2009, we introduced two new designs, which, to the best of our knowledge, were never previously implemented in large-scale educational security exercises. These new designs are the focus of the remainder of the paper.

3 The 2008 iCTF — A Security "Treasure Hunt" Scenario

"It is the early morning and someone is frantically knocking at your door. Shockingly, this person turns out to be a high-profile counter-terrorist agent from a popular television series demanding that you help him. It comes to light that an evil organization known only as 'Softerror.com' has set up an explosive device set to detonate in seven hours. Only you and your small group of elite hackers have the necessary skills to infiltrate the Softerror network, find the bomb, and disarm it before it is too late."

This introductory scenario was given to the teams participating in the 2008 UCSB iCTF, which ran Friday, December 5, from 9am to 5pm, PST. 39 teams, averaging 15 students each, competed from educational institutions spread across several continents. Unlike former iCTFs where each team setup a vulnerable virtual server, in the 2008 iCTF we created 39 identical, yet independent copies of a small network with the topology depicted in Figure 1. The network was allegedly run by a terrorist organization called Softerror.com and was composed of four hosts, each of which belonged to a separate subnetwork and had to be compromised in a specific sequence. The final host contained a virtual bomb that had to be defused to complete the competition. A more detailed description of the four hosts comprising the Softerror.com network is included in Section 3.1.

The creation of 39 replicated networks, required a completely different software and hardware infrastructure from what was used in previous iCTFs. A total of 160 virtual machines needed to be hosted for each of the teams' networks and our test network. Furthermore, machines had to be isolated so that the teams could not interfere with each other. In addition, the network had to be set up in such a way that one network was only made available following the compromise of a previous network. This task was accomplished using a complex routing system that created the illusion of a separate dedicated network for each team. The details of the network setup are described in Section 3.2.

In addition to the new network topology, another significant difference between this competition and classic CTFs was that the teams did not directly attack each other. Instead, this competition modeled a more "real world" situation where the teams had to penetrate a remote network. Given only a single IP

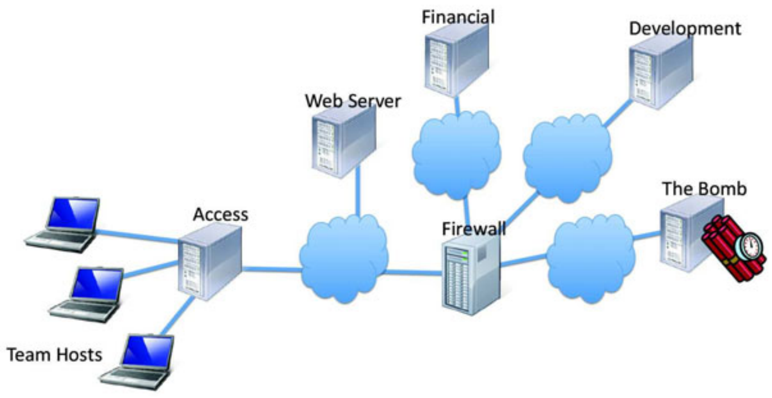


Fig. 1. The 2008 iCTF Network Topology

address, the teams needed to map out the network, discern what services were running, and then exploit these services to gain access to the various servers. Then, they could use the compromised host as a starting point to penetrate deeper into the network and repeat the process, discovering new machines and crafting the corresponding exploits.

Although this competition could have been presented as a "race" where the winner was the team to deactivate the bomb the quickest, we wanted to score the competition at a finer granularity, allowing us to easily monitor the progress of each team. Therefore, we decided to assign a score to each service and award points when the service was compromised. The teams were instructed to submit a report about each compromised service through a web site. The reports were manually examined and the points awarded by a judge. Although we had wanted to automate this process, it was decided that we wanted to know how each team broke into each server. Thus, because teams did not have to directly attack each other, and could make progress immediately after compromising a service, independent of the judge, we predicted to have sufficient time to manually judge each exploit. Unfortunately, as described in Section 3.3, this manual process had its share of problems.

Further differentiating from former CTFs, this competition had the notion of a "mole," which was a fictitious character who provided valuable clues. In order to gain access to clues, teams spent points earned from compromising services. Thus each team was faced with the choice to use points to potentially speed up service compromise. At the end of the event, the overall winner was the team who had the most points and defused the bomb.

3.1 Vulnerable Applications

The competition was divided into three major stages. The stages were arranged by increasing difficulty. The first stage was a web server with relatively simple vulnerabilities, which successfully compromising led to two possible paths: a team could attack either the financial server or the development server. Exploiting either of these servers, opened access to the final stage, the bomb challenge. The network partitioning was accomplished by setting up a firewall within the second stage machines that blocked traffic to the bomb server. Thus, in order to gain access to the final stage, the teams had to gain root access on one of the second stage machines and disable the firewall.

The Web Server. The web server was the first stage of the "Softerror network" on which a simple PHP-based web site ran. Successful compromise required each team to discover a command injection vulnerability. While a simple code inspection of the site would reveal the vulnerability, teams were not given the source code. However, another vulnerability was introduced that allowed the reading of an arbitrary file. This vulnerability could be discovered by inspecting the normal operation of the site and noticing that a user-generated link actually contained a base64 encoded file system path. By exploiting this vulnerability, a team could then download the source PHP code and discover the command injection vulnerability.

The Financial Server. The Financial server hosted the Softerror "financial" information. This server was set up as a series of four stages that had to be completed in order. The primary focus of the financial stages was passwords and password management. The first stage required breaking a weak password given its hash, which could be easily accomplished by performing a web search of the hash. The other three stages contained various examples of poor password management. Stage 2 was a service with weak authentication, stage 3 stored the passwords in a plain-text file that was readable from a vulnerable application, and stage 4 contained an SQL injection vulnerability that could be used to reveal the final password.

The Development Server. The development server challenge required binary reversal. The service itself was accessible on a remote port and contained a format string vulnerability allowing arbitrary code execution. In reality, remote exploits can be very complicated to successfully exploit, as they often require detailed understanding of the target machine's underlying memory layout. To simplify matters, the application leaked a significant amount of information, thus easing the process of developing an exploit. Our intent was to make exploiting this vulnerability approximately as hard as breaking into the financial server.

The Bomb Host. The bomb was implemented as an Elf x86 "firmware" binary that could be downloaded, modified, and then uploaded back to the server. The bomb was also a binary reversing challenge, as no source code was made available. In order to successfully complete this challenge, a disarm function of the library had to be written by the teams. After writing, uploading, and successfully calling the function, the firmware deactivated the bomb thus completing the competition for the team.

3.2 Infrastructure

The most prominent difference between the 2008 iCTF compared to the classic CTF is the network topology. Normally, each competitor is responsible for providing hardware to host the vulnerable virtual server. With this competition, a motivating goal was to mimic a real-world remote network penetration, including reconnaissance and multi-step attacks. To facilitate our goal, we decided to host all the machines ourselves, guaranteeing a level playing field and a network topology that we directly controlled. Even though the network given to each team consisted of only four machines, the number of teams required us to host 160 virtual machines.

It was readily apparent that we did not have enough physical hardware to support the required number of virtual machines if we were to use "heavy weight" virtualization platforms such as VMware or Xen. Although there are many reasons why we may have wanted to use these suites, the amount of hardware resources they require per guest made them unsuitable. In order to meet our virtual machine requirements within our hardware budget, we chose OpenVZ, a modified Linux kernel that provides a mechanism to elegantly solve this issue.

The advantage of using OpenVZ is that it allows for a very lightweight style of virtualization in a Unix environment. Instead of virtualizing an entire hardware stack, OpenVZ is a kernel modification that creates a sort of "super" root user on the host machine. This super user can then spawn guest machines that have their own operating environment, including the notion of a regular root user. This concept of kernel-level virtualization has existed for quite some time, especially on the BSD family of operating systems, where it is known as a "jail." One downside to this kernel-based approach is that the range of guest operating systems is very limited. Only Unix-like operating systems with a similarly patched kernel can be used as guests. The benefit of taking this approach is that the kernel itself is now shared among all the guests thus making resource sharing much simpler and very efficient. Once we decided on using OpenVZ, scaling our limited resources to the required number of machines turned out to be fairly simple.

Our operating system of choice was Ubuntu 8.10 with the aforementioned OpenVZ modified kernel. Each guest was created using the default OpenVZ Ubuntu template. We had six physical machines based on a Core2 quad core CPU with 4 Gigabytes of RAM and 1 Terabyte of disk space each. With some basic testing, it was determined that we could host a little over 40 guests on each individual machine using the default configuration settings. Although OpenVZ has a plethora of parameters that can be tweaked and adjusted to provide even greater scalability, we hit our target number with the default settings. Of those machines, four of them were selected to be our server "stacks." That is, on one machine we brought up 40 OpenVZ guests responsible for running identical copies of the "web server" service. Likewise, one machine was dedicated to each of the other three servers. Although this setup has merit based just on its simplicity, the primary reason for hosting identical services on the same physical machine is that it is trivial to write directly to the file system of OpenVZ guests from the host OS. In addition, OpenVZ supports executing commands from the host OS as a root user on the guest. With these two features, keeping all identical guests on the same machine greatly simplified management, as updating configuration files and sharing resources is made vastly simpler.

With the configuration of individual machines done, it was decided that they should be arranged around a single "firewall" host. This host had eight physical Ethernet ports, making it ideal to act as both a central location to record traffic and also help enforce routing rules. One requirement was that each team should see an isolated view of their network. To accomplish this, a basic iptables-based firewall was used to restrict traffic based on the IP address and virtual Ethernet device of the OpenVZ guest. That is, traffic coming from a virtual Ethernet device associated with one team would only be forwarded to the IP ranges also associated with that team. Furthermore, these firewall rules ran on the host machine, which was inaccessible from the guest machines. This setup gave us the unique ability to have both mandatory iptables rules to enforce game rules and iptables rules on the guest itself to simulate an internal firewall that could be modified. Thus we could ensure the order in which each stage had to be broken. Moreover, even if teams were to spoof traffic, they could not interfere with each other.

3.3 Overview of the Live Exercise

The competition took place on December 5th, 2008, from 9am to 5pm, PST. Even though we did not disclose the nature of this competition until the day of the event, we were able to get the network up and running beforehand by distributing a test virtual machine to test network connectivity, much like we would have done had we run a more traditional CTF. Although we had not stress-tested our OpenVZ-based network against the oncoming onslaught of connections, the network was stable. The largest connectivity issues came from a few teams that accidentally wrecked one of their OpenVZ guests. We had explicitly set it up so that all traffic had to go through the initial web server and the teams had to be very careful not to sever this connection. Even though we were explicit about being careful due to this very problem, we did attempt to help teams that had not heeded this warning. However, we also had to exercise great care while helping as a misconfigured firewall or an improper command could easily have taken the entire competition offline.

Another issue we overlooked was the name of our fictitious terrorist group. Clearly, the name Sof*terror* was supposed to invoke the idea of an evil software group doing evil software things. We had not considered that there might actually be a Soft*error* group that would supply friendly services such as consulting and development to their entirely benign customer base. Unfortunately, we did not think to check for similar names until after the competition began. Thus, when we received an email from a team asking if they were supposed to attack the real domain www.softerror.com, we were quite surprised. However, since the traffic of the whole competition was confined to a VPN with non-routable IP addresses, no actual attacks were carried out against external targets.

Initially we had hoped to augment the process of scoring exploits through the use of automated tools. Unfortunately, these tools were untested by the time the competition started and issues quickly developed. We were then faced with the decision of trying to hot fix them as the competition progressed or take a manual approach. With the competition running along, it was decided that we should take the latter approach and that the teams would have to submit their exploits by email to be manually judged. When decided, the number of emails started to grow very rapidly as teams raced to submit their exploits. Given that we had about ten people involved as organizers, we were quickly overwhelmed with the torrent of emails and the competition suffered because of it. While we attempted to maintain fairness by giving emails that detailed successful breaks for a particular level to the same judge, the response time could be fairly long, which prompted teams to send additional emails, exacerbating the problem. Moreover, trying to work through all the issues related to scoring took us away from dealing with other issues that came up, such as the occasional connection issues mentioned above.

At the end of the competition, with minutes to spare, Team ENOFLAG managed to upload a modified version of the bomb firmware that successfully deactivated their bomb. Embroiled with the controversy surrounding our scoring procedures, we were at least relived that we could unequivocally decide the winner of the 2008 iCTF.

4 The 2009 iCTF — A Botnet Attack Scenario

The theme for the 2009 iCTF was "Know Thy Enemy!" For this competition, we decided to mimic the world of malware and design a competition that incorporated many features unique to the physiology of modern botnets. In this iCTF edition, each team played the role of an evil botmaster, competing against other botmasters for the control of a large number of simulated users. The 2009 iCTF was the largest security competition to date, with 56 teams representing more than 800 participants.

Scripts simulated users that were to be compromised and controlled by the participants. Each simulated user followed a cyclical pattern: First the user visited a bank (called *Robabank*, a pun on the real Rabobank) using a browser, and logged in using her credentials. The bank set a cookie in the user's browser to authenticate further requests. Then the user visited a news site (called *PayPerNews*) and randomly extracted a word from the content of the news. Teams could publish news on this site by paying with money from their bank account.

The word chosen by the user was then submitted to a search engine called *Goollable*. Goollable routinely crawled each team's editable webpage. One of the links returned by the search engine was chosen by the simulated user using a Pareto distribution that gave higher probability to the top links. The user directed the browser to this page, controlled by one of the teams, and was possibly compromised by a drive-by-download attack. Finally, the user returned to the bank web site and checked the balance of her account. Even though the user script was identical for all 1,024 users, the users browsed with different browsers each having multiple versions with unique vulnerabilities.

The goal of the participants was to lure a user to a web site under the participant's control, perform a drive-by-download attack against the user's browser, and take complete control of the user. Once the user was compromised, a team had to do two things: i) transfer the money from the user's Robabank account to their own account thus accumulating "money points," and ii) establish a connection from the user to a remote host, called the *Mothership*, on which to send information identifying the compromising team. A team gained "botnet points" by performing this action. This last step was introduced to generate traffic patterns resembling the interaction of bots with Command-and-Control (C&C) hosts in real botnets.

Solving side challenges offered teams a third way to gather points. Challenges varied in type (e.g., binary reversing, trivia, forensics) and difficulty. Teams were awarded "leetness points" for solving a challenge.[4]

At the end of the game, the final score was determined by calculating the percentage of each team with respect to each type of points and computing a weighted sum of the percentages, where botnet points had a weight that was twice the weight of leetness points and money points. More precisely, given the maximum money point value across all teams, M, the maximum botnet point value, B, the maximum leetness point value, L, and the score in each of these categories for a specific team, m, b, and l, the total score for a team was computed as $25m/M + 50b/B + 25l/L$. Note that during the game, teams could exchange leetness points and botnet points into money points using the *Madoffunds* web site. The exchange rates varied dynamically throughout the competition.

This rather complex system of inter-operating components was a central aspect of the 2009 iCTF. That is, instead of just concentrating on single services or single aspects of the game, the participants were forced to understand the *system as a whole*. Even though this aspect generated some frustration with the participants, who were used to the straightforward designs of previous competitions, the purpose of the complexity was to educate the students on understanding security as a property of complex systems and not just as a property of single components.

We expected the teams to first solve a few challenges in order to gain leetness points. These points would then be converted into money points using the *Madoffunds* site and used to pay for the publishing of news on the *PayPerNews* web site. At the same time, a team had to set up a web page with content "related" to the published news. The idea was that a user would eventually choose a term in a news item published by a team, whose web site would score "high" in association with that term. This scheme is similar to the Search Engine Optimization (SEO) techniques used by Internet criminals to deliver drive-by-download attacks. Once the user was lured to visit the team's web site, the team had to fingerprint the user's browser and deliver an attack that would allow the team to take control of the user. The first team to compromise a user could transfer all the user's money to their account. Additionally, all teams that compromised a user could gather botnet points by setting up a bot that connected to the *Mothership* host.

The *Robabank*, *Madoffunds*, *PayPerNews*, *Goollable*, and *Mothership* sites had no (intended) vulnerabilities. The only vulnerable software components were the

[4] A discussion of some iCTF09 challenges can be found at `http://www.cs.ucsb.edu/~bboe/r/ictf09`

browsers used by the simulated users. In the following, we provide more details about the search engine behavior and the browsers whose vulnerabilities had to be exploited.

4.1 The Crawler and Search Engine

A crucial goal of each team was driving the vulnerable browsers to their web server. In order to do so, teams needed to perform SEO to boost their search results for desired keywords thus driving traffic to their web server. Each team was allowed to host a single web page accessible by the root path. A sequential web crawler visited the teams' web pages once a minute in random order. In order to be indexed, web servers needed to respond to requests within one second, and responses over 10KB were ignored.

Once a page was crawled, keywords were extracted from *title*, *h1* and *p* HTML tags, and a base score was assigned to each keyword based on the number of times the particular keyword appeared in the text relative to the total number of keywords. For instance, in the text, "the quick brown fox jumps over the lazy ground hog" there are a total of ten keywords with *the* appearing twice thus having a density of 0.2. All other keywords have a density of 0.1.

To prevent teams from using naïve techniques, such as having a page with only a single keyword or alternatively containing every word in the dictionary, only keywords with densities between 0.01 and 0.03 were assigned base scores. However, keywords appearing in either the *title* or *h1* tags were guaranteed a base score of at least 0.008. In addition to the base score, a bonus multiplier was applied to keywords appearing in the *title* or *h1* tags according to a linearly decreasing function that favored sooner appearing keywords. For example, in the title "iCTF 2009 was Super Awesome!" the keyword *iCTF* would receive a 0.3 fraction of the *title* multiplier and the remaining words would respectively receive a 0.25, 0.2, 0.15, and 0.1 fraction of the multiplier.

On the other end of this system was the *Goollable* search site. *Goollable* was accessible both by the simulated users and to each team through a standard HTML interface. The simulated users performed single keyword searches to determine which team's web server to visit and the teams accessed Goollable to see their relative search ranking for particular keywords. When designing this system, it was our hope that teams would reverse-engineer the scoring function in attempt to achieve the maximum possible score for desired keywords. Our hope, however, fell short and we decided to release the crawler and search engine source code midway through the competition.[5]

4.2 The Vulnerable Browsers

The overall goal of a team was to compromise as many users as possible. Users had to be lured to a web site under the control of the attacker that would

[5] The crawler and search engine source code is available at `http://www.cs.ucsb.edu/~bboe/public/ictf09/search_engine.tar.gz`

deliver a drive-by-download attack, in a way similar to what happens with attack "campaigns" in the wild. In the following, we describe the characteristics of the various browsers that were used by the simulated users in the competition.

Operla. As the name suggests, *Operla* was a browser written in Perl that relies on `libwwwperl` to perform the necessary HTTP communication. Operla supports a minimal form of the HTML `object` tag, and introduces a so-called *Remote-Cookie-Store*. Three versions of Operla were deployed incrementally during the competition each of which contained unique vulnerabilities.

The first version of the Operla implemented a *Remote-Cookie-Store*. This feature was designed to upload a copy of the users' cookies to a remote location. While the attacker could choose an arbitrary URL to upload to, Operla would perform this action only if an associated security header contained the correct password. Operla stored an MD5-sum of the password, thus by determining the plain text password associated with the MD5-sum, an attacker could trigger the upload of the cookies. Since the MD5-sum was stored without a salt, searching for this value on the web was enough to retrieve the required password.

The second version of Operla contained a vulnerability that mimics the real-world vulnerability of the Sina DLoader ActiveX component [9]. This component allowed an attacker to download and install an arbitrary file from the Internet on the victims' computers. Operla, by incorrectly validating the parameters for HTML object tags, suffered from a similar vulnerability.

The final version of Operla contained a remote code execution vulnerability. To exploit the vulnerability an attacker had to perform the following steps. First, two HTTP headers needed to be sent back to the browser. One contained the code that should be executed upon successful exploitation and the other contained an arbitrary URL. To this response, Operla created a challenge string consisting of ten random characters and transmitted them in a request to the arbitrary URL. Second, the attacker needed to respond to the request with a JPEG image consisting of a visual representation of the challenge string and containing the MD5-sum of the challenge string in the image's EXIF header. If the image was configured properly Operla executed the attacker-provided code.

Jecko. Jecko was a vulnerable browser written in the Java language. During the competition, we provided the teams with three versions of Jecko, each containing a different vulnerability. All versions were distributed in bytecode format, which was trivial to convert to source code by using a Java decompiler, such as JAD.

The first vulnerability was a command injection in the code that handles the HTML `applet` tag. When Jecko parses an `applet` tag, it retrieves the code base specified by the `code` attribute, saves it on the local disk, and executes it by spawning a system shell, which, in turn, invokes the Java interpreter using a restrictive security policy. In addition, Jecko allows pages to specify a custom `mx` attribute to set the maximum memory available to the "applet" program. The value, however, is used unsanitized in the shell invocation. Students had to identify the command injection vulnerability in Jecko's source code, and attack it by injecting arbitrary shell commands, which would then be executed with the privileges of the user running the browser.

The second vulnerability consisted of exposing untrusted pages to insecure plugins. In Jecko, plugins are implemented as C libraries that are loaded by the Jecko through the JNI framework. Pages can instantiate plugins by using the `object` tag. Jecko was provided with two plugins, one of which exposed a function that allows pages to download the resource at a given URL and execute it. This vulnerability mimics several real-world vulnerabilities, such as CVE-2008-2463. Exploiting this vulnerability required the attacker to understand Jecko's plugin mechanism and reverse-engineer the provided binary plugins.

Finally, the last vulnerability consisted of an authentication flaw in Jecko's upgrade system. When Jecko parses a page containing the custom `XBUGPROTECTION` tag, it assumes it is visiting a site that provides updates for the browser. Then, the URL specified by this tag should contain a serialized Java object that specifies the commands required for the updates. These commands are encrypted with a key transmitted as part of a custom HTTP header. The attacker had to reverse-engineer this upgrade mechanism and discover that the update commands are not signed. To launch an actual exploit, teams had to create serialized versions of the update object and configure their pages to respond with the appropriate custom HTTP header.

Erbrawser. Erbrawser was a browser written in the Erlang programming language and was composed of a main module that implemented the user interface, performed queries via the standard Erlang `http` library, and parsed the HTML responses via the *mochiweb* toolkit. The vulnerability for this browser was designed to be simple to detect and exploit. The main challenge was becoming familiar with this functional language and producing the correct code to inject into Erbrawser. Erbrawser was deployed in two versions containing a similar vulnerability, although the second version of the browser was more difficult to exploit. The source code for both browsers was given to the teams thus making it somewhat simple to discover the exploit in the first version.

Erbrawser introduced the `<script type="text/erlangscript">` tag that executed its content in the Erlang interpreter without sanitization or sandboxing. This feature contained a number of vulnerabilities that allowed the execution of arbitrary third-party code inside a user process. The first version of Erbrawser allowed the execution of any Erlang commands, thus the easiest way to exploit was to execute shell commands through `os::cmd()`. The second version of Erbrawser forbid the execution of `os::cmd()` and similar functions. Nonetheless, having direct access to Erlang's interpreter gave the attacker the ability to use the browser as a bot by spawning an Erlang thread inside the browser. This thread could then continuously read and submit flags to the *Mothership*.

Pyrefox. Pyrefox was a browser written in Python. The browser had two versions each with a vulnerability. The first version of Pyrefox used the *path* attribute of a cookie to determine the name of the file in which to store the cookie's contents. Therefore, one could use a path traversal attack to overwrite or create any file accessible to the browser's user. The second version of Pyrefox fixed that vulnerability but introduced a code injection vulnerability. This vulnerability was associated with the fictitious scripting language, called JavaScrapt, supported by the Pyrefox browser. The JavaScrapt language allowed a modification to the page by specifying an XML Path-like expression to a new string. However, the path and the string were passed unsanitized to an `eval()` statement, allowing for the execution of arbitrary Python code.

Crefox. Crefox was a browser written in the C programming language requiring `libcurl` and `htmlcxx` for HTTP communications and HTML parsing respectively. Crefox came in three versions, all with vulnerabilities that eventually allowed an attacker to execute arbitrary code supplied as part of the pages retrieved by the browser. We voluntarily leaked all the versions of Crefox source code throughout the competition to allow the teams to focus on exploiting the vulnerabilties rather than reversing the binaries.

The first version of Crefox had a NULL pointer de-reference vulnerability in addition to a `mmap` call that mapped the downloaded HTML page at the virtual memory address 0. Thus, triggering the vulnerable code path required the attacker to serve a page starting with the special string `USESAFEPRINTFUNCTIONA`, followed by the code the attacker wanted to execute. When the special string appeared, Crefox attempted to call a nonexistent function thus executing the attacker's code. The second version of the browser fixed the previous vulnerability, but introduced a format string vulnerability triggered when the downloaded page was about to be printed. To make this vulnerability difficult to detect, the `printf` function name was encoded and subsequently retrieved at run-time.

Finally, the third version of Crefox fixed the previous vulnerability, but introduced two new ones, namely a plain stack-based buffer overflow, and a non-control data buffer overflow leading to a command injection. Ironically, the former was not intended at all, but was the result of a typo that switched the destination buffer of a string copy routine from a global variable to one residing on the stack. To trigger the command injection vulnerability, the attacker had to embed a new HTML `ictf` tag with a `code` attribute containing a specific pattern followed by the shell command to inject.

4.3 Overview of the Live Exercise

The 2009 iCTF took place on December 4, 2009 between 8 am and 5 pm, PST. The participating teams connected to the competition VPN during the preceding week. The teams received an encrypted archive that contained a presentation describing the complex setup of this competition. The presentation was supposed to include audio that described the various steps of the competition, but unfortunately the audio was not included, due to technical problems. This generated some confusion in the initial phase of the game. We eventually gave teams the audio portion of the instructions and the game could start.

The first problem we ran into was the poor performance of the entire system. Various components of the infrastructure were only lightly tested and under the full weight of hundreds of participants, they slowed to a crawl. We traced this issue to the fact that most components were not multi-threaded and therefore they could not respond to the volume of requests being made. We solved the issue by adding some standard threading code and while this operation easy to do, it still required some time, causing further delays to the game.

In addition to the scoring infrastructure woes, we also had to deal with problems with the simulated users. The aforementioned threading issues affected the simulated user scripts causing them to misbehave and not visit teams' websites as often as they should have. There were also some browser specific issues, for example the Java based Jecko browser would periodically run out of memory causing our simulated user processes to terminate unexpectedly. This issue went mostly unresolved and as a consequence there were not many opportunities for teams to exploit this particular browser.

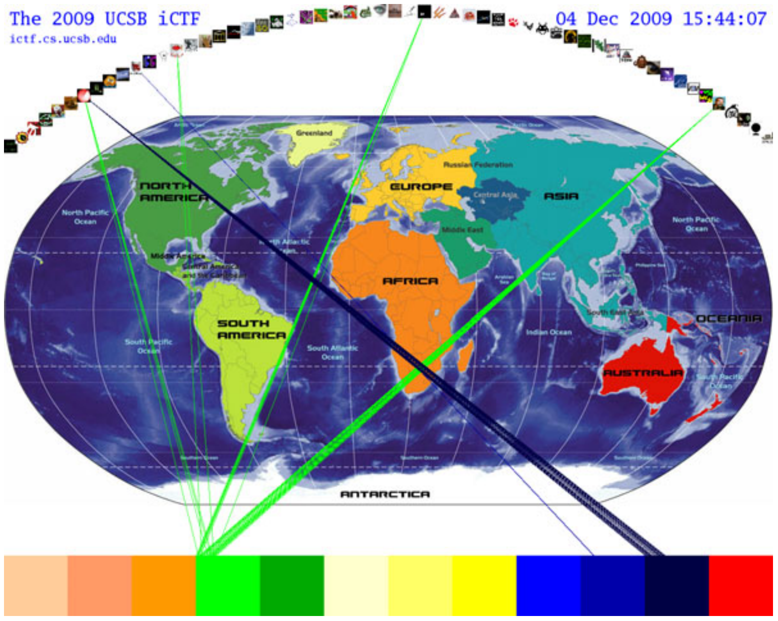


Fig. 2. The 2009 iCTF Scoreboard. Each pixel on the bottom of the screen represents one of the 1,024 users, each colored according to the browser they used. A line from a user to a team indicates that the user was "owned" by the team.

While we were sorting out these issues, a team managed to discover a flaw in the banking system. The goal of the banking system was to have people withdraw money from our simulated users and add it to their own accounts. One enterprising team discovered a flaw in our validation routines, in that we allowed for negative amounts. In effect, this allowed them to drain the accounts of other teams by simply making a transfer request from their own account to the victims account, with a negative amount. They were gracious enough to alert us and let us patch the flaw, instead of wrecking havoc throughout the scoring system.

Even though the competition had a slow beginning, the teams started to figure out how to exploit the competition's complex system of interconnected components and eventually users were compromised and made part of a botnet. Figure 2 shows the graphic format that was used, during the competition, to show which users were "owned" by which team.

By the end of the competition, the team CInsects managed to capture a significant portion of the simulated users that visited their site and took first place.

4.4 Feedback

After the 2009 competition was completed, a poll was given out to the point of contact for each team. Each person was asked to rank various aspects of this competition. Of the 56 participants we received 35 responses.

The most basic question we wanted to answer was in regards to the size of the competition. Every year we see more and more teams participating, however we did not ask about how many individuals each team had. This year we wanted to get a more accurate number on how many people were participating. From our respondents we learned that, on average, each team was composed of 15 players. Although it is unfortunate that we cannot give an exact number because we did not get feedback from every team, we can provide a very reasonable estimate that this competition involved well over 800 individuals from all over the world, supporting our claim of running the largest security competitions.

Table 1. Botnet Gameplay Feedback

Category	No Response	Awful	Satisfactory	Awesome
Playability	2	4	23	6
Score System	1	8	20	6
Novelty	1	2	12	20
Network Setup	1	4	16	14
Challenges	1	4	15	15

From these responses as outlined in Table 1 and the feedback received, it is clear that while most teams enjoyed the competition there were numerous aspects that needed improvement. The most common complaint was that the mechanics of this competition were not entirely clear at the start and it took teams a significant amount of time just to understand what they were being asked to accomplish. However, the overwhelming majority of teams agreed that it was a novel competition and echoed our sentiments in that creating a new competition leveled the playing field. It is also interesting to note that not many people complained about the network connectivity, even though we were having issues with the simulated users connecting to teams early on in the competition. We suspect that our network issues may have been masked by the fact that the teams were taking a significant amount of time to understand the competition.

Table 2. Botnet Team Feedback

Category	Yes	No
First Time Playing	12	23
Developed Tools	18	17
Gained Skill	33	2
Educational	33	2

Table 3. Network Trace Characteristics

Year	Duration (hh:mm)	Data Size (MB)	Packets (10^6)	Data Rate (bytes/s)
2003	3:19	2,215	6.96	188,941
2004	0:54	258	2.78	121,680
2005	7:27	12,239	30.14	1,427,060
2007	6:45	37,495	92.57	2,065,115
2008	41:29	5,631	34.11	60,179
2009	17:51	13,052	40.58	550,408

By the time the teams were actually ready to receive traffic, we had resolved the network issues.

One of the primary goals for running these security competitions is that we want participants to go beyond the normal course work and explore new and original ways to accomplish these tasks. In order to check if this goal was being met, we asked how many teams had developed tools specifically for this competition. As displayed in Table 2, half of our respondents stated they had worked on tools specifically for this competition, giving us empirical evidence that these competitions are working as intended. Furthermore, we also collected evidence supporting the continuing popularity of these competitions, as more than two thirds of our respondents were veterans who had played in previous CTFs.

5 Lessons Learned

Throughout the seven years (and eight editions) of the iCTF we learned (the hard way) a number of lessons.

An important lesson that we learned very early is that the scoring system is the most critical component of the competition. A scoring system must be *fair* and *objective*. Given the size of the competitions we ran, this means that it also needs to be *automated*, that is, it cannot rely on human input. Even though we learned this lesson many years ago, our scoring system failed catastrophically during the 2008 competition. In this case, the lack of testing forced us to switch to manual evaluation, with very unsatisfactory results. Fortunately, because of its design, the competition had a clear winner, which was determined in an objective way. However, we cannot say the same about the ranking of the rest of the teams.

A second lesson learned is that the critical events should be *repeatable*. That is, all the events that cause a change in the score of a team should be logged, so that if a bug is found in the scoring mechanisms, it is possible to handle the failure and restore the correct scores of all teams. The 2009 scoring system did not include a manual component, but suffered from a number of glitches, mostly due to erroneous database programming. Fortunately, all transactions were logged and, therefore, it was possible to troubleshoot the problems and restore the correct scores.

A third lesson learned about the scoring procedure is that a scoring system should be *easily understood* by everyone involved. This helps because, on one hand, the participants will understand what is strategically important and what is not, and, in addition, they can identify errors in the scoring process and help the organizers fix them. On the other hand, if the system is well-understood by the organizers, it is easy for them to fix problems on-the-fly, which is often necessary, given the time-critical nature of the competition.

As previously mentioned, our scoring system, as well as network traces can be found at `http://ictf.cs.ucsb.edu/`. Table 3 summarizes the network traces captured for the past iCTFs.

While these lessons described above have been learned throughout all the editions of the iCTF, there are several lessons that are specific to the 2008 and 2009 editions.

A first lesson is that attack-only competitions like the 2008 and 2009 iCTFs are valuable. When the iCTF was first conceived, it was to be the final test for students finishing a graduate-level computer security course. The goal was to provide a one-of-a-kind hands-on learning experience for teams of novice security experts. Since then, several annual CTFs have emerged in addition to the iCTF, and there are quite a few teams that regularly participate in these competitions. As such, these teams have gained significant experience and are quite organized. Even though the goal of the competition is to foster the development of novel tools and approaches, the fact that some teams come very prepared can make the competition too hard for newer, inexperienced teams. By having an attack-only competition, where the teams compete side-by-side and not directly against each other, it is possible to shield newcomers from "veteran" teams. Of course, a major drawback of attack-only competitions is that the defense skills of the students are not tested.

New participants were also aided by the new nature of the competition. In fact, by changing radically the style of the competition, we managed to somewhat level the playing field. Although the winning teams were still experienced groups, in both the 2008 and 2009 editions, teams of first-time competitors placed quite high in the ranking. This was possible because we intentionally did not disclose in advance to the teams the nature of these new competitions. In both cases, many "veteran" teams expected a standard CTF and were surprised to learn that this was not the case. Of course, it is hard to keep surprising teams, as designing new competitions requires a substantial amount of work. However, it is arguable that this type of competition is inherently easier for novice teams to participate in.

Another important lesson is that too much novelty can hurt the overall competition. Although the 2008 competition was a radical departure from the traditional iCTF, the task was fairly straightforward: break into a network. With the 2009 iCTF, the competition structure was much more complicated. Not only did teams have to reverse-engineer the browser software they also had to perform Search Engine Optimization to get users to visit their sites. Moreover, once they understood how to capture users, to score points they had to figure out how the banking system worked, as well as how the botnet *Mothership* could be used to generate more points. In total, there were three different kinds of points, with a fairly complex relationship between them, which many participants found sometimes confusing.

A final thought about these kind of competitions is: *are they worth the effort?* Preparing all the editions of the iCTF and, in particular, the two completely new iCTF structures of 2008 and 2009 took an enormous amount of time and resources. Therefore, it is understandable to wonder what are the benefits. We think that the fact that after the iCTF was introduced many other similar events started surfacing shows that there is interest and perceived value for these events. We really do think that competition fosters group work and creative thinking, as witnessed by the feedback we gathered for the 2009 iCTF, and we think that live exercises are a useful tool to support the security education of students. Also, these types of competitions help the organizing team, because they provide visibility and publicity to their institution. For example, this year a number of the applications to the graduate program of the Department of Computer Science at UCSB mentioned the iCTF.

6 Conclusions

Security competitions and challenges are a powerful educational tool to teach hands-on security. However, the design, implementation, and execution of complex, large-scale competitions require a substantial amount of effort. In this paper we described two novel designs that were implemented as large-scale security live educational exercises. These exercises involved several hundred students from dozens of educational institutions spread across the world. The information that we provided about the software/hardware infrastructure supporting the competitions, as well as the lessons learned in running these events can be useful for educators who want to create similar competitions.

Acknowledgements

This work has been supported by the National Science Foundation, under grants CCR-0524853, CCR-0716753, CCR-0820907, and by the ARO under grant W911NF-09-1-0553.

References

1. Augustine, T., Dodge, R.: Cyber Defense Exercise: Meeting Learning Objectives thru Competition. In: Proceedings of the Colloquium for Information Systems Security Education, CISSE (2006)
2. ComputerMajors.com: Computer Science Degrees: Starting Salaries (June 2009), http://www.computermajors.com/starting-salaries-for-computer-science-grads

3. Cowan, C., Arnold, S., Beattie, S., Wright, C., Viega, J.: Defcon Capture the Flag: defending vulnerable code from intense attack. In: Proceedings of the DARPA Information Survivability Conference and Exposition (April 2003)
4. Group, T.H.: The ructf challenge (2009), http://www.ructf.org
5. Mullins, B., Lacey, T., Mills, R., Trechter, J., Bass, S.: How the Cyber Defense Exercise Shaped an Information-Assurance Curriculum. IEEE Security & Privacy 5(5) (2007)
6. Pimenidis, L.: Cipher: capture the flag (2008), http://www.cipher-ctf.org/
7. Pwn2own 2009 at cansecwest (March 2009), http://dvlabs.tippingpoint.com/blog/2009/02/25/pwn2own-2009
8. Schepens, W., Ragsdale, D., Surdu, J.: The Cyber Defense Exercise: An Evaluation of the Effectiveness of Information Assurance Education. Black Hat Federal (2003)
9. SecurityFocus: Sina DLoader Class ActiveX Control 'DonwloadAndInstall' Method Arbitrary File Download Vulnerability, http://www.securityfocus.com/bid/30223/info
10. Vigna, G.: Teaching Hands-On Network Security: Testbeds and Live Exercises. Journal of Information Warfare 3(2), 8–25 (2003)
11. Vigna, G.: Teaching Network Security Through Live Exercises. In: Irvine, C., Armstrong, H. (eds.) Proceedings of the Third Annual World Conference on Information Security Education (WISE 3), June 2003, pp. 3–18. Kluwer Academic Publishers, Monterey (2003)

An Online Adaptive Approach to Alert Correlation

Hanli Ren, Natalia Stakhanova, and Ali A. Ghorbani

Information Security Center of eXcellence
University of New Brunswick
Fredericton, New Brunswick, Canada
{e8vwe,natalia,ghorbani}@unb.ca

Abstract. The current intrusion detection systems (IDSs) generate a tremendous number of intrusion alerts. In practice, managing and analyzing this large number of low-level alerts is one of the most challenging tasks for a system administrator. In this context alert correlation techniques aiming to provide a succinct and high-level view of attacks gained a lot of interest. Although, a variety of methods were proposed, the majority of them address the alert correlation in the off-line setting. In this work, we focus on the online approach to alert correlation. Specifically, we propose a fully automated adaptive approach for online correlation of intrusion alerts in two stages. In the first online stage, we employ a Bayesian network to automatically extract information about the constraints and causal relationships among alerts. Based on the extracted information, we reconstruct attack scenarios on-the-fly providing network administrator with the current network view and predicting the next potential steps of the attacker. Our approach is illustrated using both the well known DARPA 2000 data set and the live traffic data collected from a Honeynet network.

Keywords: alert correlation, Bayesian network.

1 Introduction

The rapid increase in the number, sophistication and impact of computer attacks makes the computer systems unpredictable and unreliable, emphasizing the importance of intrusion detection technology. Intrusion detection systems (IDS) are generally designed to provide a system administrator with sufficient information to handle intrusion incidents. In practice, with the ever increasing capacity of networks this often translates to a large number of low-level alerts produced by an IDS. A tremendous volume of alerts coupled with their low quality makes it challenging for a system administrator to handle intrusions in timely manner.

In this context, the alert correlation techniques aiming to consolidate relevant IDS alerts in a concise high-level format gained a special interest. Several alert correlation techniques have been proposed in the recent past including approaches based on feature similarity analysis [1], attack rules and scenarios [2,3,4,5,6,7] and analysis of alert statistics [8,9,10,11]. Generally, these techniques follow one of two directions: they either rely on expert knowledge or infer relationships among alerts using statistical or machine learning analysis. Although the expert knowledge based approaches appear to produce accurate results, they are generally limited in their ability to integrate novel alerts. Moreover, accurately defining all possible relationships among existing alerts can be prohibitively tedious and time-consuming, and thus not always feasible. The inference approaches on the other hand allow to accommodate novel alerts through automatic alert analysis. However, they might not fully discover the causal relationships between related alerts. Given the complementary nature of the expert knowledge-based and inference approaches, it is highly desirable to combine their strengths, while avoiding their weaknesses.

In this paper, we present a method for automatic correlation of intrusion detection alerts that brings together the strengths of expert knowledge-based and inference approaches. Instead of relying on user expertise, we propose to analyze the casual relationships among alerts using a Bayesian network. Through this analysis coupled with network configuration information and expert knowledge we automatically extract the constraints and alert relationships that characterize attack steps. The extracted information is essentially attributed to various attacks and thus can be employed to piece together an attack strategy in the online setting.

To facilitate real-time intrusion analysis, we further develop a technique for the adaptive online alert correlation. To allow the correlation procedure to automatically adjust to the new previously unseen (in the offline setting) behavior, the approach monitors the alerts behavior to reflect any significant changes that might potentially influence the causal relationships among alerts. This online correlation strategy not only provides a picture of the current intrusive activity on the network, but also predicts a potential next step of an attacker.

Although the online component is specifically developed for runtime correlation of alerts, both components can be applied in the offline setting.

The contributions of this paper can be summarized as follows:

– *A Bayesian correlation feature selection model* that allows to automatically retrieve causal relationships and relevant features among alerts without expert or domain knowledge. The proposed feature selection method explicitly shows the relationships among alerts and provides reasoning behind these relations.
– *An adaptive method for online attack scenario construction* that allows a user to extract attack patterns in real time. The proposed method provides a dynamic adaptation of the correlation procedure to the temporal changes in alerts' behavior. This allows to address previously unseen alerts, and consequently, discover new attack steps.
– *Implementation* of the proposed approach that allows a user to generate attack scenarios from a large amount of raw alerts on-the-fly.

The reminder of the paper is organized as follows. A brief overview of related work is given in Section 2. Section 3 provides an overview of the proposed approach for alert correlation. Section 4 describes the detailed design of the alter correlation feature selection algorithm based on Bayesian causality theory. Section 5 describes the design of online alert correlation. Experimental results are given in Section 6. Section 7 concludes the paper with our future work.

2 Related Work

In recent years, a variety of alert correlation techniques have been proposed aiming to reduce the overwhelming number of alerts and to provide a global and condensed view of the network security status. Broadly speaking, these approaches can be divided into several groups: *alert aggregation techniques* [1,12,13] that cluster similar alerts; the methods focused on *detection accuracy improvement* [14,15,16] that aim to improve the accuracy of intrusion detection often through filtering of false positive and low-interest alerts; the methods for *alert prioritization* [17,18,19] that focus on adjusting priority of alerts based on their severity; and *alert causality* analysis. Since our work employs the alert causality analysis, we will primarily focus on the related work in this area.

Research on causality analysis trends can be considered within the three categories: *scenario-based correlation*, *rule-based correlation* and *statistical correlation*. In general, the studies in the first two categories rely on expert knowledge to find related alerts, while the approaches in the last category aim to infer logical relationships among alerts using statistical or machine learning analysis.

Scenario-based correlation methods [5,6,7,20] find relationships among alerts based on the known attack scenarios. The goal of alert correlation in this case is to find a sequence of alerts that match pre-specified scenarios. Attack scenarios can be specified using an attack language (e.g., SRARL [6], LAMDBA [5], ADeLe [7]) or learned using machine learning techniques [20]. One of the major downsides of these approaches is the necessity to develop all attack scenarios in advance, which is not only a time-consuming and error-prone process, but also requires a considerable expert knowledge.

Rule-based correlation approaches [2,3,4] are based on the observation that the majority of alerts are related, i.e., they either represent the early stages of an attack or intermediate steps of more advanced attack behavior. Thus, analyzing alerts based on the predefined rules containing prerequisites and consequences of attack steps is sufficient to identify related alerts. Similar to the scenario-based correlation methods, these approaches require specific attack knowledge. Although they can explicitly show the logical relationship between the alerts, they cannot handle novel attacks since their prerequisites and consequences are undefined.

As opposed to rule-based methods, statistical approaches [8,9,10] statistically analyze relationships among alerts based on their co-occurrence within certain time period, and thus, are generally independent of prior domain knowledge. As such, Qin et al. [8,21] presented a Bayesian correlation engine to discover the strong statistical dependency among alerts. Based on the assumption that alerts are causally related if there exists a strong statistical dependency among them, Qin analyzes statistical patterns among the aggregated alerts, i.e. *hyper alerts*. The degree of relevance of alerts is evaluated by calculating the conditional probability

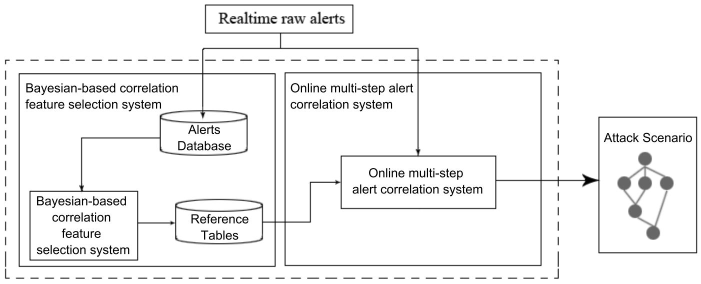

Realtime raw alerts
Bayesian-based correlation feature selection system
Alerts Database
Bayesian-based correlation feature selection system
Reference Tables
Online multi-step alert correlation system
Online multi-step alert correlation system
Attack Scenario

Fig. 1. Overview of the alert correlation system

among each pair of hyper alerts. The approach builds the attack scenarios by evaluating the causal relationship between each pair of hyper alerts. Since the number of all possible combinations of hyper alerts can be extremely large, a straightforward application of this approach in the online setting becomes infeasible.

Motivated by this idea, we propose to generate attack strategies on-the-fly. Specifically, rather than evaluating statistical patterns among the hyper alerts, we focus on extracting the constraints and causal relationship between each pair of alert types (even if the number of raw alerts is large, the number of alert types is usually limited to a small number). We take an advantage of this strategy in the online attack scenarios construction stage by analyzing only the most relevant features.

There have been several works [9,10] specifically focused on the methods for estimating correlation probability among alerts using Multilayer Perceptron (MLP) [22], Support Vector Machines (SVM) [23] and frequent structure mining technique [10]. Unlike scenario-based and rule-based methods, statistical approaches do not require expert knowledge and are capable of representing unknown attacks; they however cannot explicitly show the causal relationship among alerts. Requiring heavy statistical analysis, these techniques are generally time consuming and thus not applicable in the online setting.

In summary, all the above approaches focus on the offline alert correlation. In our work, we attempt to address this issue by performing alert correlation in two stages. Unlike scenario-based and rule-based methods, we use statistical analysis to automatically extract prerequisites and consequences of attack steps. Since the statistical analysis is a time-consuming process, we perform it in an offline mode. Then, based on the extracted information, the online component then connects related alerts and constructs attack scenarios on-the fly. Contrary to the statistical methods, our approach can explicitly show the causal relationships between alerts.

3 Overview

The alert correlation aims to consolidate IDS alerts based on their causal relationships. These relationships can be discovered relatively easy if attack strategy, prerequisites and consequences are known in advance. In practice, manual generation of such attack information requires expert knowledge and experience, and thus is not only time-consuming, but also error-prone. An alternative approach to realize these relationships is through the automatic analysis of raw alerts. However, the main challenge that arises in this context is to extract sufficient number of constraints and conditions pertinent to the considered attack strategy in order to accurately characterize the instances of this attack in the future. In this work, we adopt the latter approach and attempt to extract casual alert relationships automatically. To achieve this goal we propose a two-component correlation model. The two components of our model, namely, offline Bayesian correlation feature selection component, and online multi-step alert correlation components, are shown in Figure 1.

In the offline component, we aim to extract relevant alert information that can be later employed in the online alert correlation. First, we aggregate alerts that belong to the same attack step. Based on the Bayesian causality we then analyze the relevance of alerts representing different attack steps. Finally, we extract the features that define relevancy of attack steps. As the result of the offline correlation, the system produces reference tables (namely, the correlation and relevance tables) that contain information necessary to identify causal relationships among alerts on-the-fly. The online alert correlation component processes the raw low-level alerts to extract attack scenarios based on the attack information provided by the reference tables. To dynamically adjust correlation process to the current alerts' behavior that might incorporate previously unseen alerts, the online module monitors the changes in the alerts' behavior. These temporal changes are automatically accounted for in the attack scenario construction.

Note that the offline component primary allows to speed up the online correlation process. Thus when necessary both components can be applied offline for analysis of historical data.

4 Bayesian Correlation Feature Selection

Figure 2 shows the first step of the proposed alert correlation framework, which is the offline analysis of low-level alerts. This analysis aims to produce sufficient information for the following automatic real-time attack reconstruction.

In the proposed framework, this analysis is performed in two steps: preprocessing and feature extraction. In the preprocessing step, the aim is to reduce redundancy by bringing alerts into a standard format of a hyper alert [8]. Generally, intrusion detection sensors produce a number of alerts for each suspicious event. Most of these alerts are repetitive and provide the same information about an event. This information can be concisely represented through hyper alerts. A hyper alert is essentially a set of low-level alerts aggregated based on the values of their attributes and clustered into a group.

In the next step, we infer the causal relationships between the hyper alerts through a Bayesian probability analysis. Based on the correlation probability, we extract hyper alert attributes that significantly influence the degree of relevance

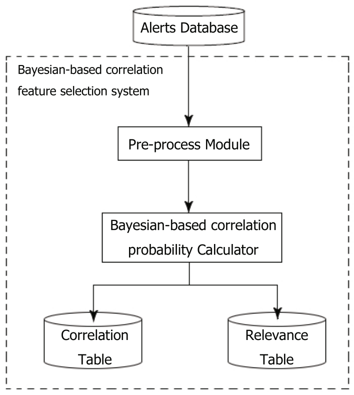

Alerts Database
Bayesian-based correlation feature selection system
Pre-process Module
Bayesian-based correlation probability Calculator
Correlation Table
Relevance Table

Fig. 2. Overview of the offline alert correlation component

of two hyper alerts. Analyzing relevance of the extracted features allows to reason about the relationships between hyper alerts and consequently between alert types. One of the requirements in this context is to ensure that the inferred relationships reflect specific properties of the network environment. This information may downgrade the relevance of alerts not applicable in a given environment or emphasize critical relationships. For example, by introducing the network asset group information, the relevancy of alerts whose target IP addresses belong to the same asset group is emphasized, even though their exact destination IP address are different. The network specific properties are incorporated through the generalization hierarchies built using expert knowledge.

4.1 Alert Preprocessing

The preprocessing step follows a straightforward strategy to compress information represented in low-level alerts. For the causal relationships analysis we adopt a hyper alert format discussed by [8].

A *low-level alert* a_i is an alert produced by intrusion detection sensor (e.g. IDS) with a set of alert attributes, i.e, features such as destination IP address, port number etc., denoted by $f_1 \dots f_j$. The value that a feature f_j assumes in alert a_i is denoted by $a_i[f_j]$. The range of possible values, i.e., domain of feature f_j is denoted by $dom(f_j)$.

Given a set of low-level alerts $a_1 \dots a_n$, a *hyper alert* A_i is a group of low-level alerts a_k, $1 \leq k \leq n$ with the same features' values (except timestamp), i.e, for each $a_k, a_l \in A_i$, $a_k[f_j] = a_l[f_j]$. The hyper alerts, as well as low-level alerts can be also distinguished based on the *type* of alert that denotes a certain attack class/step. To differentiate between the types of alerts, we refer to low-level alerts using small letters and to hyper alerts using capital letters. As such, low-level alerts a_i, b_i and hyper alerts A_i, B_i indicate alerts of type a and b, respectively. Then, $A = [A_1, A_2, A_3, \cdots, A_m]$ denotes a group of all hyper alerts of type a

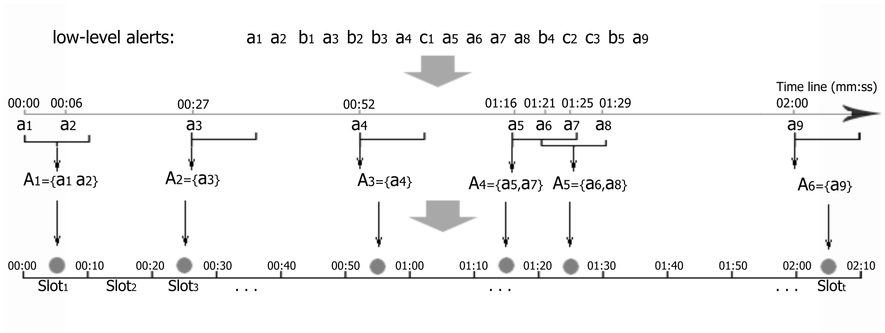


Fig. 3. The pre-processing procedure

and $dom(A, f_j) \subset dom(f_j)$ denotes the range of values that feature f_i assumes in alerts $a \in A$.

To form a set of hyper alerts, a stream of low-level alerts is initially broken into time windows using a sliding window approach. The alerts within each window is clustered based on the alert type and then merged into hyper alerts based on the alert feature values. The example of preprocessing procedure for alerts of type a is illustrated in Figure 3. For example, the alerts a_1, a_2 fall into the same time window and happened to have the same feature values, thus they are merged into a hyper alert A_1. Similarly, a_5, a_6, a_7 occurred in the same time window. The feature values of alert a_5 match alert a_7, but do not match alert a_6, thus only a_5 and a_7 are merged into a hyper alert A_4. In the next time window, we consider alerts not previously merged into hyper alerts, i.e., a_6 and a_8. These two alerts have the same feature values, so we merge them into a hyper alert A_5. The resulting stream of hyper alerts is also broken into time windows, $Slot_1, Slot_2, \cdots, Slot_t$. The result of preprocessing stage is the groups of hyper alerts of different types $A, B, \ldots, Z$.

4.2 Feature Selection

Evaluating relationships between intrusion detection alerts essentially means analyzing alert attributes, i.e. features. Since not all attributes equally contribute to the relationship between two alerts, it is desirable to identify the attributes that are necessary to be analyzed. Thus, given a set of hyper alerts, the goal is to extract alert features that are the main contributors to the relationships between intrusion detection alerts.

We perform this process in three steps as follows:

1. *Feature construction:* derive additional features based on the basic alert attributes such as IP address, port, and protocol.
2. *Correlation probability calculation:* estimate the correlation probability of alerts and determine the alert features that contribute the most to this probability.
3. *Construction of correlation and relevance tables.*

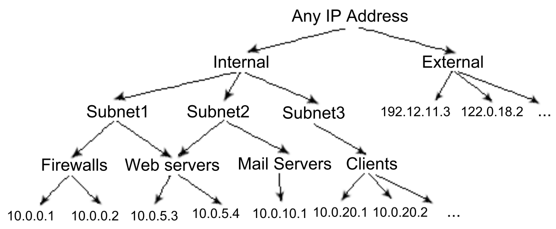


Fig. 4. An example of a generalization hierarchy of IP address

Step 1: Feature construction. The IDS alert features capture intrinsic alert properties, such as the IP address of an alert, its port, protocol information, etc. While the values of these features are the same for low-level alerts grouped in one hyper alert (except time stamp), their values vary among the hyper alerts of the same type. At the same time feature values of hyper alerts share common patterns that allow to describe the hyper alert type. For example, a set of hyper alerts representing an attack on a subnet, although has a different destination IP addresses, shares the same subnet address.

Extracting the basic alert attributes may not be sufficient to fully discover these patterns. We, thus, derive additional features from the available attributes, called *extended features*. To derive extended features we follow the idea of *generalization hierarchy* introduced by [13]. The generalization hierarchy is a directed acyclic graph that spawns elements of an alert attribute domain. For example, Figure 4 shows a sample generalization hierarchy for IP addresses. We employ the following generalization hierarchies:

– *Alert IP address hierarchy* includes source and destination IP addresses and is generalized into asset groups (e.g. firewalls, mail servers), subnets and network domain (e.g. internal, external network).
– *Alert port number hierarchy* includes both source and destination ports and is generalized according to a service assigned to a port (e.g., DNS, FTP, HTTP) and into privileged and non-privileged ports.

Generally, generating generalization hierarchy is network-specific and thus requires expert knowledge.

Step 2: Correlation probability calculation. The probability inference engine is based on Bayesian network [24], one of the most widely used models for understanding the causal relationships among a large number of variables. A Bayesian network is essentially a graphical model that represents probabilistic relationships among all variables.

A Bayesian network model consists of: (1) a network structure that describes analyzed variables via a directed acyclic graph (DAG) and (2) a set of probabilities associated with each variable and presented in conditional probability tables (CPTs). Together, these components describe causal or dependent relationships among variables and the strengths of these relationships [24]. Figure 5 shows a

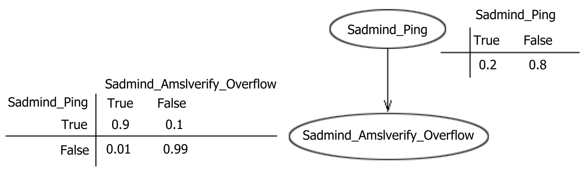

Sadmind_Ping
Sadmind_Amslverify_Overflow
Sadmind_Ping: True 0.2, False 0.8
Sadmind_Amslverify_Overflow | Sadmind_Ping True: True 0.9, False 0.1
Sadmind_Ping False: True 0.01, False 0.99

Fig. 5. An example of a causal network

snapshot of an alert Bayesian network. The occurrence of children alerts is primarily influenced by the state of their parents. In this context, the children alerts can be viewed as a direct cause of parent alerts, i.e., a consequent attack step. To evaluate the probability of child alert occurrence given its parents' state a conditional probability $P(child|parent)$ has to be computed. This step is known as probabilistic inference and can be assessed as follows:

$$P(child = c|parent = p) = \frac{P(child = c \wedge parent = p)}{P(parent = p)} \qquad (1)$$

Propagating the probabilistic inference calculation from parents to children allows to infer dependencies among alerts and estimate the strengths of their relationships.

Our interest in Bayesian inference model is not limited to relationships between alert types. To be able to assess the alert relevance in the online setting, we need to reduce the amount of features analyzed. Thus, we employ Bayesian model to determine the influence of the individual features on the causal relationships among alerts.

The pseudocode for estimating Bayesian correlation probability among alert types is given in Algorithm 1. Given a pair of hyper alert groups $\langle A, B\rangle$, the procedure aims to analyze the causal relationship between the hyper alert of type a and b, and specifically, the influence of feature f_j on this relationship. The procedure returns the probability of the occurrence of type b alerts given that type a alerts happened, denoted by $P(B|A[f_j] = dom(B, f_j))$. This process requires calculation of three components:

– the prior probability of alerts of type b, $P(B)$, which essentially indicates a probability of type b alerts happening (Line 2). Note that prior probabilities of alert types can be extracted during alert preprocessing step.
– the probability of occurrence of alerts of type a with a specific value, $P(A[f_j] = dom(B, f_j))$ (Lines 4-7).
– the probability of occurrence of type b alerts with the given feature value, $P(B \wedge A[f_j] = dom(B, f_j))$ (Lines 9-14). Intuitively, it is clear that the relationship between alerts of two types have to be analyzed through some temporal constraints. We employ an *expirePeriod* forcing the algorithm to consider only alerts following type a alert within this specified time period. Theoretically, a large time window allows us to find the relationships between alerts of slowly developing attacks. However, it also increases the analysis

Algorithm 1. Causal Relationship Analysis procedure

```
1: function $P(B|A[f_j])$ relAnalysis ($< A, B >, f_j$)
2:   calculate $P(B)$;
3:   $AF \leftarrow \emptyset$;
4:   for each $A_i \in A$ do
5:     If $A_i[f_j] \in dom(B, f_j)$, add $A_i$ into $AF$;
6:   end for
7:   calculate $P(A[f_j] = dom(B, f_j))$;
8:   $ABF \leftarrow \emptyset$;
9:   for each $A_i \in AF$ do
10:      $w$ ←number of time windows covered by the $expirePeriod$ ;
11:      $TW_{A_i} \leftarrow$ time window of $A_i$ ;
12:      $TW \leftarrow \{TW_{A_i}, TW_{A_i+1}, \cdots, TW_{A_i+w}\}$;
13:      Within $TW$, if $\exists B_{ii} \in B$ s.t. $B_{ii}[f_j] = A_i[f_j]$, then add $A_i$ into $ABF$;
14:   end for
15:   calculate $P(B \wedge A[f_j] = dom(B, f_j))$;
16:   $P(B|A[f_j] = dom(B, f_j)) \leftarrow \frac{P(B \wedge A[f_j]=dom(B,f_j))}{P(A[f_j]=dom(B,f_j))}$ ;
17:   return $P(B)$,$P(B|A[f_j] = dom(B, f_j))$;
18: end function
```

Algorithm 2. FeatureSubsetSelection procedure

```
1: function $F_{subset}$ featureSelection($< A, B >$,$F$)
2:   $F_{subset} \leftarrow \emptyset$;
3:   for each $f_j \in F$ do
4:     if $P(B|A[f_j] = dom(B, f_j)) > t$ then
5:       add $f_j$ into $F_{subset}$;
6:     end if
7:   end for
8:   $n \leftarrow$ number of elements in $F_{subset}$;
9:   $k \leftarrow 2$;
10:  while $k \leq n$ do
11:    $G \leftarrow$ all $k$ size combinations from $F_{subset}$;
12:    $tempSet \leftarrow \emptyset$;
13:    for each $g_i \in G$ do
14:      if $P(B|A[G] = dom(B, G)) > t$ then
15:        add $\forall f_j \in g_i$ into $tempSet$;
16:      end if
17:    end for
18:    if $tempSet \neq \emptyset$ then
19:      $F_{subset} \leftarrow tempSet$;
20:      $n \leftarrow$ number of elements in $F_{subset}$;
21:      $k \leftarrow k + 1$;
22:    end if
23:  end while
24:  return $F_{subset}$;
25: end function
```

time. Thus, the *expirePeriod* size selection should be depend on the system performance.

The main advantage of the casual relationship procedure (Algorithm 1) is that it requires no knowledge of attack scenarios or constraints. Applied to the stream of alerts, the algorithm can distinguish influential and irrelevant features for proceeding alerts; it cannot, however, identify the degree of this influence and determine whether combinations of certain features can be associated with the more significant influence. This type of analysis is performed by Algorithm 2.

Based on the relevancy strength and a predefined threshold t, we can distinguish four kinds of features:

– If $P(B|A[f_j] = dom(B, f_j)) = P(B)$, then f_j is an *irrelevant feature*. In other words, feature f_j does not influence the occurrence probability of alerts of type b.
– If $P(B|A[f_j] = dom(B, f_j)) < P(B)$, then f_j is *a relevant feature with negative influence*. The presence of certain values of feature f_j in alerts of type a decreases the occurrence probability of type b alerts.
– If $P(B) < P(B|A[f_j] = dom(B, f_j)) < t$, then f_j is *a relevant feature with positive influence*. The presence of certain values of feature f_j in alerts of type a slightly increases the occurrence probability of type b alerts.
– If $P(B|A[f_j] = dom(B, f_j)) > t$, then f_j is *a relevant feature with critical influence*. The presence of certain values of feature f_j in alerts of type a significantly increases the occurrence probability of type b alerts.

To analyze the significance of subsets of features, only relevant features with critical influence are analyzed. Algorithm 2 follows a greedy approach by analyzing all possible combinations of features (Lines 10-23). Starting with pairs of

Table 1. An example of Correlation Table

Alert Type Pair	**Correlation probability**	**Relevant Features/constrains**
$<T_1, T_2>$	70%	f_2,f_4,f_6
$<T_1, T_3>$	65%	f_1,f_3,f_4,f_6
...	...	...
$<T_i, T_n>$	80%	$f_2 = f_4$

Table 2. An example of Relevance Table

Alert type	**Occurrence Probability of** T_i **Alerts**	**Relevant Alert Types**	
		Strongly relevant	**Weakly relevant**
T_1	5%	T_2, T_3, T_5, T_6	T_4, T_7, T_8, T_9
...	...	...	...
T_n	1%	T_7	T_1, T_2, T_8, T_9

features, the procedure randomly adds a feature to each subset whose probability exceeds the specified threshold t (Lines 14-16).

Step 3: Construction of correlation and relevance tables. Once the correlation probabilities are evaluated, we construct reference tables, specifically, *correlation* and *relevance* tables that allow to hypothesize about causal relationships among alerts. The correlation table contains for all alert type pairs: the correlation probability, the relevant features that significantly influence this probability and the constrains that characterize the relationship of this pair. We denote $T = [T_1, T_2, ...T_z]$ as a set of alert types. Table 1 gives an example of the correlation table.

As oppose to the correlation table, the relevance table contains information per alert type. An example of the relevance table given in Table 2 shows that each alert type is associated with occurrence probability and the sets of weakly or strongly relevant alert types.

5 Online Alert Correlation

One of the challenges in applying alert correlation in practice is the ability of the system to extract attack strategies on-the-fly. This is mainly due to the amount of information necessary to process and draw a meaningful conclusions about the relationships among alerts. In Section 4 we introduced the Bayesian correlation engine that performs this analysis in the offline setting and outputs probability information of alert types and the corresponding relevant features. The goal of the online component is to identify "causally" related alerts and construct attack scenarios on-the-fly based on the relationships and constraints identified in the *Correlation Table.*

The online alert correlation component is given in Figure 6 and consists of the two primary modules: an *alert correlation module* responsible for identifying alert causality, and an *attack scenario module* that produces an attack graph based on the pairs of causally-related alerts.

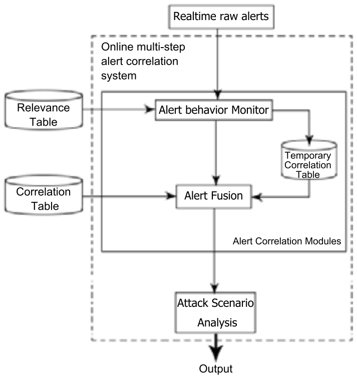


Fig. 6. Overview of online multi-step alert correlation system

Algorithm 3. Online alert correlation procedure

```
1: function AttackList onlineCorrelation ($Table_{corr}$, $Table_{rel}$, $t$)
2:   $TempTable_{corr} \leftarrow$ OccurProbCheck($Table_{rel}$);
3:   $RecordedAlerts \leftarrow \emptyset$;
4:   $AttackList \leftarrow \emptyset$;
5:   for each incoming alert $b$ do
6:     $Type_B \leftarrow$ Type of $b$;
7:     for each alert $a \in RecordedAlerts$ do
8:       $Type_A \leftarrow$ Type of $a$;
9:       $TypePair \leftarrow < Type_A, Type_B >$ ;
10:      if $\exists$ $TypePair$ in $TempTable_{corr}$ then
11:        $p \leftarrow getCorProb(TypePair, TempTable_{corr})$;
12:      else
13:        $p \leftarrow getCorProb(TypePair, Table_{corr})$;
14:      end if
15:      if $p > t$ then
16:        $F \leftarrow getRelFeatures(TypePair, Table_{corr})$;
17:        if $a$ and $b$ have the same value of all $F$ then
18:          if $a \in attack$:$\exists attack \in AttackList$ then
19:            Add $b$ into $attack$;
20:          else
21:            Create a new $attack$;
22:            Add $a$ and $b$ into $attack$;
23:            Add $attack$ into $AttackList$;
24:          end if
25:        end if
26:      end if
27:    end for
28:    Add $b$ into $RecordedAlerts$;
29:  end for
30:  return $AttackList$;
31: end function
```

Algorithm 4. Occurrence probability check procedure

```
1: function $Table_{rel}$ OccurProbCheck ($TempTable_{corr}$)
2:   $alertsList \leftarrow$ all alerts happened in the last hour
3:   for each alert type $t$ do
4:     $P_1 \leftarrow$ occurrence probability of all type $t$ alerts in $alertsList$
5:     $P_2 \leftarrow$ occurrence probability of type $t$ alerts in $Table_{rel}$
6:     if $P_1 - P_2 > 100\%$ then
7:       $weaklyRel \leftarrow getWeaklyRelType(t, Table_{rel})$;
8:       for each alert type $wt \in weaklyRel$ do
9:         $recomputeCorProb(\langle t, wt \rangle)$;
10:        update $TempTable_{corr}$;
11:      end for
12:    end if
13:    if $P_2 - P_1 > 100\%$ then
14:      $stronglyRel \leftarrow getStronglyRelType(t, Table_{rel})$;
15:      for each alert type $st \in stronglyRel$ do
16:        $recomputeCorProb(\langle t, st \rangle)$;
17:        update $TempTable_{corr}$;
18:      end for
19:    end if
20:  end for
21:  return $TempTable_{corr}$;
22: end function
```

Adaptive alert correlation module. To discover "causally" related alerts, the correlation module relies on the reference tables. The reference tables maintain stable alert information that represents alerts' behavior in the past. As we see in practice, the majority of attacks have established patterns. Thus, it is reasonable to assume that if an alert was linked to certain attack steps in the past, it is likely to be related to those steps in the future. Following this intuition, the online correlation module analyzes alert pairs with a high correlation probability based on the information provided by the correlation table.

While this strategy works well for the known patterns frequently seen in attacks, it does not allow to discover new attack steps or to incorporate in attack scenario alerts with less obvious relationships (e.g., due to their low presence in the data during the offline analysis). In order to account for these alerts in the online step, the online correlation module monitors the alerts' behavior, specifically the changes in the occurrence probability of alerts. Any sudden and significant change in the frequency of known alerts or appearance of new ones may indicate potential change in the "strength" of relationships of the corresponding alert pairs. These temporal changes to relationships, i.e, correlation probability of alerts are maintained in the *Temporal correlation table*, which can be viewed as a snapshot of alerts behavior at a given period of time. Since this table has a temporal nature, it only serves as an intermediate step between scheduled runs of the offline feature selection process.

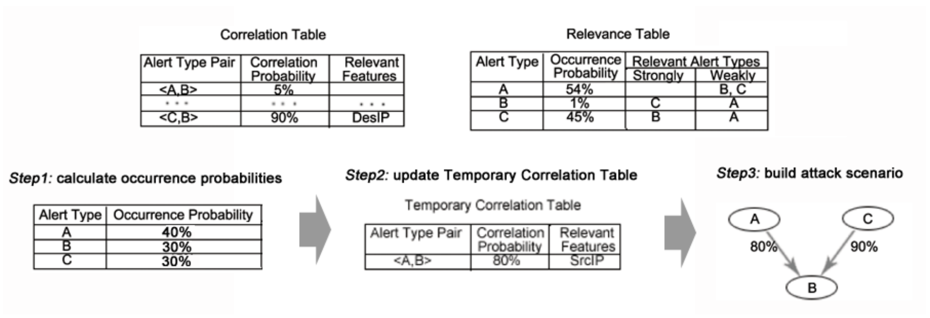
The main advantage of such approach is that it allows to discover new alerts' relationships without any domain or expert knowledge and incorporate them into the attack scenario on-the-fly.

The *Online alert correlation* procedure function in Algorithm 3 presents the pseudocode for online correlation component. It takes as arguments a correlation table $Table_{corr}$, a relevant table $Table_{rel}$, a threshold t and an alert stream *stream*. The function returns the *AttackList*, a list of attacks where each attack is given as a set of correlated alerts.

The online correlation of alerts is performed in two steps: first, alerts' occurrence probability is analyzed to determine if any deviation in alerts' behavior has occurred. Second, the causal relationships among alerts are determined.

Step 1 - Alert behavior analysis: a temporary correlation table is maintained to monitor the significant and sudden changes in the alerts' behavior. First, we calculate the occurrence probability of each alert type in the last period of time (e.g. last one hour) (Line 4). Then we compare this probability with the one stored in the Relevance Table. If the occurrence probability of T_i alerts suddenly increases, the correlation probability of the alert type pairs grouped by T_i and its weakly relevant alert types is re-calculated. In case the produced result does not match the Correlation Table, the corresponding information is logged in the Temporary Correlation Table (Lines 6-12). On the other hand, if the occurrence probability of T_i alerts suddenly decreases, the correlation probability of the alert type pairs grouped by T_i and the corresponding strongly relevant alert types is re-calculated (Lines 13-19).

Step 2 - Alert fusion: before hyper alerts correlation probability can be computed, we pair the alerts that have shown strong connection in the past according to both the Correlation Table and the Temporary Correlation Table. When fusing two alerts, the alert type pair information is queried from the Temporary Correlation Table first, if no record found, the original Correlation Table is used. In practice, there is a number of alerts that do not have clear relationships to


Correlation Table

Alert Type Pair	Correlation Probability	Relevant Features
<A,B>	5%	
...	...	...
<C,B>	90%	DesIP

Relevance Table

Alert Type	Occurrence Probability	Relevant Alert Types	
		Strongly	Weakly
A	54%		B, C
B	1%	C	A
C	45%	B	A

Step1: calculate occurrence probabilities

Alert Type	Occurrence Probability
A	40%
B	30%
C	30%

Step2: update Temporary Correlation Table

Temporary Correlation Table

Alert Type Pair	Correlation Probability	Relevant Features
<A,B>	80%	SrcIP

Fig. 7. An example of Correlation Process

other alerts. Although these alerts are preserved in the Correlation Table, their contribution to attack scenario is insignificant and even misleading. Thus, we apply a probability threshold that allows to navigate to the alert pairs that have the strongest relationships, i.e, are more likely to represent a meaningful step in an attack. Thus, given a probability threshold t, two alerts A_i and A_j are linked together, if $CorProb\langle A_i.A_j\rangle > t$, that is if the correlation probability of $\langle A_i.A_j\rangle$ exceeds the threshold t.

Illustrative example. Let a_1, b_1, c_1, a_2, c_2, a_3, b_2, b_3, a_4, c_3 be the latest alert steam to be analyzed by the online component in the example in Figure 7. Assume that the provided Correlation and Relevance tables have been built in the offline component. First, the occurrence probabilities of each alert type observed in the incoming stream are calculated and compared with the contents of the Relevance table (*Step*1). Let us assume that the probability of alerts of type b suddenly increased ($P(B) = 30\%$ compared to the previously recorded $P(B) = 1\%$). This increase first of all influences the correlation probability between the alerts of type b and the weakly relevant alert types. The strongly relevant types are not considered as the probability increased and thus their relevancy cannot become weak. In this case, the correlation probability of the pair $\langle A, B\rangle$ is re-calculated and the result is recorded in the *Temporary Correlation Table* as shown in *Step*3. In *Step*4, the attack scenario is built based on information contained in both the *Correlation Table* and the *Temporary Correlation Table.*

Attack scenario analysis. An attack scenario is generated based on the pairs of causally related alerts. Figure 8 shows a simple example of an attack graph. As shown in the attack graph, the $Port\_Scan, Buffer\_Overflow, FTP\_User$ alerts have already been grouped. Moreover, even though FTP_Pass alert was not reported yet, the online component is able to predict it based on the known causal relationships of $Buffer\_Overflow$ alert (i.e., FTP_Pass attack has the same source IP and destination IP address with the $Buffer\_Overflow$ attack).

Alert Type Pair	Correlation Probability	Selected Features
<Port_Scan, Buffer_Overflow>	75%	DesIP,DesPort
<Buffer_Overflow, FTP_User>	90%	SrcIP,DesIP
<Buffer_Overflow. FTP_Pass>	90%	SrcIP,DesIP

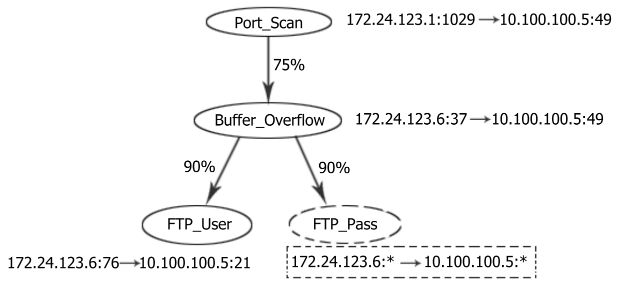


Fig. 8. Attack Graph Example

6 Experimental Results

To evaluate the effectiveness of the proposed alert correlation approach, we have implemented the algorithms described in Sections 4 and 5 and performed a series of experiments focusing on the following issues: (1) the ability of the proposed approach to select relevant features, (2) to construct accurate attack scenarios, (3) the ability to discover new attack steps, and (4) performance efficiency of the approach.

Feature selection. For our experiments we employed 2000 DARPA/Lincoln Lab offline evaluation data [25], in particular LLDOS 1.0 scenario which includes a distributed Denial-of-Service (DDoS) attack. In this scenario, the attacker first scans the network to determine which hosts are "up", then uses the "ping" option of the *sadmind* exploit program to determine which of the discovered hosts are running the *sadmind* service. Eventually, the attacker launches the *sadmind* Remote-to-Root exploit in order to compromise the vulnerable machines, and uses telnet, *rcp* and *rsh* to install a DDoS program in the compromised machines.

The low level alerts for a given data set were generated using the signature-based Snort IDS [25] by replaying "Inside-tcpdump" data. Among 15 different alert types produced by Snort, 5 alert types are directly related to the LLODS1.0 scenario. The correlation probability and the selected features of these 5 alert types are shown in Table 3.

As the results show, alert types generally exhibit specific attack patterns. For example, *Sadmind_Ping* alerts usually share the same source IP address and destination subnet, which means the attacker probes several target machines in a subnet from one source to detect hosts running *Sadmind* service. On the other hand, *Mstream_Zombie* alerts usually share the same destination port, which means the attacker issues same attacks (attacks against the same port) on various targets from different sources.

Table 3. Correlation Table

Alert Type Pair	**Correlation probability**	**Relevant Features/constrains**
<Sadmind_Ping, Sadmind_Ping>	0.96	SrcIP, DesSubnet
<Sadmind_Ping, Sadmind_Overflow>	1.0	SrcIP,DesIP,DesPort
<Sadmind_Ping, Admind>	1.0	srcIP,desIP,SrcPort,DesPort
<Sadmind_Ping, Rsh>	1.0	SrcIP,DesIP
<Sadmind_Ping, Mstream_Zombie>	1.0	DesIP of Sadmind_Ping equals scrIP of Mstream_Zombie
<Sadmind_Overflow, Sadmind_Overflow>	0.86	SrcIP,DesSubnet,DesPort
<Sadmind_Overflow, Sadmind_Ping>	0.0	
<Sadmind_Overflow, Admind>	1.0	srcIP,desIP,SrcPort,DesPort
<Sadmind_Overflow, Rsh>	0.86	SrcIP,DesIP
<Sadmind_Overflow, Mstream_Zombie>	1.0	DesIP of Sadmind_Overflow equals scrIP of Mstream_Zombie
<Admind,Admind>	0.81	SrcIP,DesSubnet,DesPort
<Admind, Sadmind_Ping>	0.13	SrcIP,DesIP
<Admind, Sadmind_Overflow>	0.87	SrcIP,DesIP,DesPort
<Admind, Rsh>	0.75	SrcIP,DesIP
<Admind, Mstream_Zombie>	1.0	DesIP of Admind equals scrIP of Mstream_Zombie
<Rsh, Rsh>	0.81	SrcSubnet,DesSubnet,DesPort
<Rsh, Sadmind_Ping>	0.0	
<Rsh, Sadmind_Overflow>	0.0	
<Rsh, Admind>	0.0	
<Rsh, Mstream_Zombie>	1.0	DesIP of Rsh equals scrIP of Mstream_Zombie
<Mstream_Zombie, Mstream_Zombie>	0.79	DesPort
<Mstream_Zombie, Sadmind_Ping>	0	
<Mstream_Zombie, Sadmind_Overflow>	0	
<Mstream_Zombie, Admind>	0	
<Mstream_Zombie, Rsh>	0	

The results in Table 3 also show the causal relationships between different alert types. Take $\langle Sadmind\_Ping, Sadmind\_Overflow\rangle$ as an example, *Sadmind_Overflow* alerts usually happen after *Sadmind_Ping*, and they share the same source IP, target IP and port, which means after using *Sadmind_Ping* to probe several targets running *Sadmind* service, the attacker issues *Sadmind_Overflow* attacks on the same targets. While for $\langle Rsh, Mstream\_Zombie\rangle$, the destination IP of Rsh alerts usually the same as the source IP address of *Mastream_Zombie* alerts, which means after the attacker compromises a target machine by *Rsh* attacks, *Mastream_Zombie* attacks will be launched against the final victim from the target machine.

Accuracy. To evaluate the accuracy of our system, we use two criteria: *True positive correlated* and *False positive correlated* rates.

True positive correlated (TPC) rate indicates the percentage of the correctly correlated alert type pairs ($True\_Correlated\_Pairs$) among all the alert type pairs that have causal relationships ($Related\_Pairs$).

$$TPC = \frac{number\ of\ True\_Correlated\_Pairs}{number\ of\ Related\_Pairs}$$

False positive correlated (FPC) rate indicates the percentage of the incorrectly correlated alert type pairs ($False\_Correlated\_Pairs$) among all the correlated alert type pairs ($Correlated\_Pairs$).

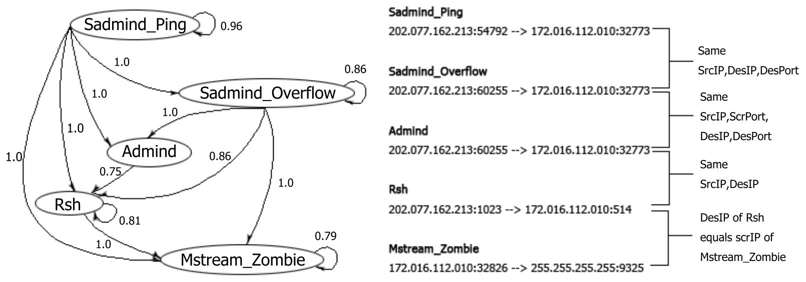
$$FPC = \frac{number\ of\ False\_Correlated\_Pairs}{number\ of\ Correlated\_Pairs}$$

Among all those 225 possible alert type pairs generated from the 15 types of alerts reported by Snort, 63 alert type pairs are labeled as causally related based on the attack scenario description given by DARPA dataset. If we set the correlation threshold to 50%, there are 70 alert type pairs correlated by our system. Among them, 9 pairs are falsely correlated, so the TPC rate of our approach on DARPA dataset is 96.8%, and the FPC rate is 12.9%. All the 9 pairs are generated among four alert types: *FTP_User, FTP_Pass, Email_Almail_Overflow and Email_Debug.* The close analysis of these pairs reveals that incorrect correlation of these alerts happens due to the high occurrence probability numbers (4% 10% compared to the probability for other types of less than 1%). This mainly due to the fact that most of these alerts share the same source or destination IP addresses as the number of different IP addresses used in DARPA experiment is very small.

Attack scenario construction. Figure 9 shows the complete attack scenario extracted from Table 3 and a group of alerts involved in this attack. A node in the attack scenario graph indicates an alert type (attack step), the edges in the graph are associated with the corresponding correlation probabilities. The rest of correlated alert type pairs, which are not shown in Table 3, also produce several connected attack graphs. But, there is no connection between these graphs and the DDoS attack scenario.

New attack steps discovery. In order to evaluate our method's ability to adapt to the temporal changes, we employed the live network traffic collected by the netForensics Honeynet team [26]. The provided logs spawn over 7 days of network traffic that triggered overall 15602 Snort alerts.

By scanning the traffic of the first day, Snort generated 1508 alerts belonging to 27 different alert types. This produced 729 alert pairs. Based on the available description and expert knowledge, out of these 729 pairs, 198 were labeled as causally related.



```
Sadmind_Ping
202.077.162.213:54792 --> 172.016.112.010:32773

Sadmind_Overflow
202.077.162.213:60255 --> 172.016.112.010:32773

Admind
202.077.162.213:60255 --> 172.016.112.010:32773

Rsh
202.077.162.213:1023 --> 172.016.112.010:514

Mstream_Zombie
172.016.112.010:32826 --> 255.255.255.255:9325
```

Fig. 9. Attack scenario

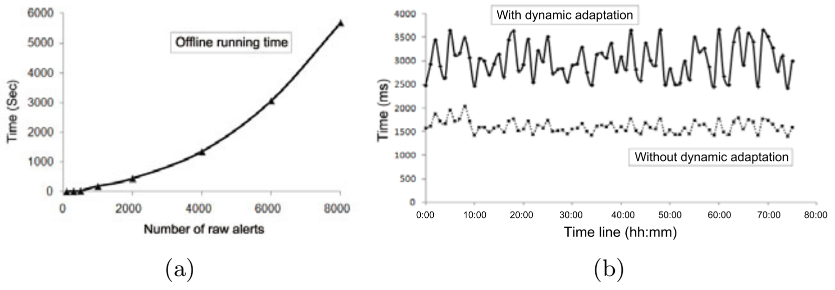


Fig. 10. Performance of the offline and online components

Applying our Bayesian offline analysis to the alerts generated by Snort, the $CorrelationTable$ shows 226 correlated alert type pairs, among them 191 pairs are correctly correlated and 36 pairs are incorrectly correlated. Thus, the TPC rate is 96.5%, and the FPC rate is 15.9%. Most of these false positive correlations are caused by the false alarms produced by Snort IDS. Specifically, Snort reported a large number of alerts of types: *ICMP Destination Unreachable, MS-SQL Worm* and *WEB-MISC WebDAV*. Due to the high frequency of these alerts and the similarity of their IP addresses, the occurrence probability of these alert types was high which resulted in their incorrect correlation.

By scanning the first hour traffic of the second day, Snort generated 221 new alerts. Among them, 2 new alert types were found with 82 alerts all together. This requires the recalculation of correlation probability of these two new alert types with the existing ones. The TPC rate without such recalculation is 93.2%, while after recalculation the TPC rate increases to 96.1%. Moreover, since the occurrence probability of some types of alerts suddenly decreases, 4 alert type pairs appear to be no longer correlated. Thus, the FPC rate decreases to 14%.

Performance. In this experiment, we focused on the evaluation of performance of both offline and online components. The experiments were run on Intel(R) Core(TM)2 with CPU of 2.4GHz. For the evaluation we employed the Honeynet traffic. The offline component was trained based on the first 8000 alerts generated by Snort and the online correlation was performed on the other half of alerts (7602 alerts) which constitutes around 80 hours. The results are presented in Figure 10. The online component was run in two modes with the dynamic adaptation to the current alert behavior and without. For this experiment, the online correlation module was configured to perform incremental correlation on one hour basis, i.e., the attack scenarios were continuously updated during one hour period, after this period expired new graphs were started. As Figure 10 (b) shows, the correlation process with the dynamic adaptation on average takes 3015 ms to process alerts triggered during one hour (on average 96 alerts). This is a reasonable processing time requirement that we consider suitable for an offline as well as an online analysis.

7 Conclusion and Future Work

In this paper, we presented a novel approach for adaptive online alert correlation. The approach incorporates two components: the offline module that is responsible for retrieving relevant attack information from the previously observed alerts based on Bayesian causality mechanism; and the online component that is based on the extracted information correlates raw alerts and constructs attack scenarios online. In addition to historic alert information, the proposed approach is able to dynamically adjust to the current alert behavior and reflect it in the correlation process. The advantages of the proposed correlation method were examined using DARPA 2000 data sets and live Honeynet data. In the future, we plan to deploy the proposed approach in the real world environment, which might bring new insights into our approach advantages.

Acknowledgements

This work was funded by the Atlantic Canada Opportunity Agency (ACOA) through the Atlantic Innovation Fund (AIF) and NSERC through grant RGPIN-2277441 to Dr. Ali A. Ghorbani.

References

1. Valdes, A., Skinner, K.: Probabilistic alert correlation. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, pp. 54–68. Springer, Heidelberg (2001)
2. Ning, P., Cui, Y., Reeves, D.S.: Constructing attacks scenarios through correlation of intrusion alerts. In: Proceedings of the 9th ACM conference on Computer and communications security, pp. 245–254 (2002)
3. Cheung, S., Lindqvist, U., Fong, M.: Modeling multistep cyber attacks for scenario recognition. In: DARPA Information Survivability Conference and Exposition, vol. 1, pp. 284–292 (2003)
4. Cuppens, F., Miege, A.: Alert correlation in a cooperative intrusion detection framework. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, pp. 202–215 (2002)
5. Cuppens, F., Ortalo, R.: A language to model a database for detection of attacks. In: Debar, H., Mé, L., Wu, S.F. (eds.) RAID 2000. LNCS, vol. 1907, pp. 197–216. Springer, Heidelberg (2000)
6. Eckmann, S.T., Vigna, G., Kemmerer, R.A.: Statl: An attack language for state-based intrusion detection. Journal of Computer Security 10, 71–103 (2002)
7. Totel, E., Vivinis, B., Mé, L.: A language driven IDS for event and alert correlation. In: SEC, pp. 209–224 (2004)
8. Qin, X.: A probabilistic-based framework for INFOSEC alert correlation. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection, vol. 2820, pp. 73–93 (2003)
9. Zhu, B., Ghorbani, A.A.: Alert correlation for extracting attack strategies. International Journal of Network Security, 244–258 (2006)
10. Sadoddin, R., Ghorbani, A.A.: An incremental frequent structure mining framework for real-time alert correlation. Computers and Security 28, 153–173 (2009)

11. Zhang, S., Li, J., Chen, X., Fan, L.: Building network attack graph for alert causal correlation. Computers and Security 27, 188–196 (2008)
12. Maggia, F., Matteuccia, M., Zanero, S.: Reducing false positives in anomaly detectors through fuzzy alert aggregation. Information Fusion 10, 300–311 (2009)
13. Julisch, K.: Clustering intrusion detection alarms to support root cause analysis. ACM Transactions on Information and System Security 6, 443–471 (2002)
14. Pietraszek, T.: Using adaptive alert classification to reduce false positives in intrusion detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 102–124. Springer, Heidelberg (2004)
15. Manganaris, S., Christensen, M., Zerkle, D., Hermiz, K.: A data mining analysis of rtid alarms. Computer Networks: The International Journal of Computer and Telecommunications Networking 34, 571–577 (2000)
16. Viinikka, J., Debar, H., Mé, L.: Processing intrusion detection alert aggregates with time series modeling. Information Fusion 10, 312–324 (2009)
17. Morin, B., Mé, L., Debar, H., Ducasse, M.: M2d2: A formal data model for IDS alert correlation. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 115–137. Springer, Heidelberg (2002)
18. Morin, B., Mé, L., Debar, H., Ducasse, M.: A logic-based model to support alert correlation in intrusion detection. Information Fusion 10, 285–299 (2009)
19. Porras, P.A., Fong, M.W., Valdes, A.: A mission-impact-based approach to infosec alarm correlation. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection, pp. 95–114 (2002)
20. Dain, O.M., Cunningham, R.K.: Fusing a heterogeneous alert stream into scenarios. In: Proceeding of the 2001 ACM Workshop on Data Mining for Security Applications, pp. 1–13 (2001)
21. Qin, X., Lee, W.: Discovering novel attack strategies from INFOSEC alerts. In: Proceedings of the 9th European Symposium on Research in Computer Security, Sophia Antipolis, pp. 439–456 (2004)
22. Haykin, S.: Neural networks: A comprehensive foundation, 2nd edn. (1998)
23. Cristianini, N., Taylor, J.S.: An introduction to support vector machines and other kernel-based learning methods (2000)
24. Heckerman, D.: A tutorial on learning with bayesian networks. Technical Report MSR-TR-95-06, Microsoft Research (1995)
25. Laboratory, M.L.: 2000 darpa intrusion detection scenario specific datasets (2000)
26. netForensics Honeynet team: Honeynet traffic logs, http://old.honeynet.org/scans/scan34/

KIDS – Keyed Intrusion Detection System

Sasa Mrdovic and Branislava Drazenovic

University of Sarajevo
Faculty of Electrical Engineering
Zmaja od Bosne bb
71000 Sarajevo
Bosnia and Herzegovina
{sasa.mrdovic,branislava.drazenovic}@etf.unsa.ba

Abstract. Since most current network attacks happen at the application layer, analysis of packet payload is necessary for their detection. Unfortunately malicious packets may be crafted to mimic normal payload, and so avoid detection if the anomaly detection method is known. This paper proposes keyed packet payload anomaly detection NIDS. Model of normal payload is key dependent. Key is different for each implementation of the method and is kept secret. Therefore model of normal payload is secret although detection method is public. This prevents mimicry attacks. Payload is partitioned into words. Words are defined by delimiters. Set of delimiters plays a role of a key. Proposed design is implemented and tested. Testing with HTTP traffic confirmed the same detection capabilities for different keys.

Keywords: Network intrusion detection, anomaly detection, word model, Kerckhoffs' principle, keyed IDS.

1 Introduction

Methods of intrusion detection made a huge advance from the time of their onset in 1980s. But, during this period attacks evolved from Morris to Conflicker worm. For each method of attack, the method of defense must be changed by modifying the detection algorithm. Once the detection algorithm is known, a new attack is created, and this race may continue forever. This paper is a try to hinder the creation of mimicry attacks based on the information of the detection method.

Network intrusion detection system (NIDS), based on anomaly detection is subject of this paper. Its role in the defense system is to detect network attacks that do not have signature yet. An anomaly based NIDS is based on a model of the normal behavior of the protected network segment. The deviation of the incoming traffic from model of normal behavior is considered as an attack. Since NIDS analyze network packets, the model is built from packet elements. Currently, most frequent attacks are at the application level [1]. To detect application level attack, the network packet payload analysis is obligatory.

Regardless of data used to model normal system behavior, anomaly based IDSs are susceptible to mimicry attacks. Simply said, a model of normal behavior is a set of allowed parameter values spans. An atypical behavior is detected when a value of one or more parameters is outside of its normal span. When parameters and their normal values are known, it is possible to create malicious attack packets having the same model as normal payload. Mimicry attack packets look as normal ones, and that prevents their identifications as unusual.

This paper offers a new approach to detection that could prevent mimicry attacks. It combines and further develops two previous works by the same authors [2,3]. The main idea is to use basic cryptography principle: the system security is not in secrecy of its design but in secrecy of a key. The intrusion detection is based on the analyses of the full packet payload and should be able to detect attacks on application level. Model of normal traffic and detection of deviation from normal is based on division of payload into "words". Words are consecutive payload bytes separated by a set of delimiters, selected byte values. Each implementation of method uses its own set of delimiters, a key, which results in its own model of the normal network traffic. Attackers cannot generate mimicry attacks if they do not know the key. A network intrusion detection method employing this principle is developed and tested.

The paper is organized as follows. Related work is addressed in section 2. Section 3 explains a new implementation of the intrusion detection method and how a key can be used. Results of testing are presented in section 4. Conclusion and discussion as well as directions for future research work are in section 5.

2 Related Work

Papers published in recent years deal with various aspects of payload analysis. Papers [4,5,6] advocate a partial payload analysis. Modern tools for packet manipulation could create attack inserted in payload in diverse forms. The form could be fine-tuned to become unrecognizable as an attack by above detection methods.

The consideration of a single application protocol makes detection easier. Number of papers analyze HTTP requests only [7,8,9,10]. This approach showed good results.

Another research direction seeks exploit code in the packet payload [11,12,13,14]. All these papers make assumptions on how attack code might look and based on those hypotheses analyze packet payload. With current rate of creation of new attacks and mutations of existing ones it is questionable if such approach can keep pace. Paper [15] states that it does not pay to try to model all versions of polymorphic sequences of payload bytes. It suggests that signature based detection is limited and that detection of anomaly from normal is more promising.

The technique proposed in this paper is close to methods that use payload division or byte grouping. PAYL, approach advocated in [16] and [17] uses single byte frequencies to make the model. Reported results are excellent, but since a single byte model is too simple, it is easy to create attack packets that fit the model of normal.

In Anagram detector [18] a model of normal traffic is constructed using fixed length payload byte sequences, named n-grams. It uses simple formula that is fast to calculate and gives excellent detection results. On the other hand, it is extremely sensitive to attacks in training data. A single attack in these data will make all of its n-grams part of normal model and make that and similar attacks absolutely undetectable.

Division of payload into byte sequences that do not have fixed length is another approach. This approach needs delimiters, byte values that divide payload into parts. Important result of [19] is generation of delimiter sets for a variety of protocols. Proposed delimiters for HTTP are used in [20]. Authors compared results for words, payload byte sequences separated by delimiters, model with n-grams models, and showed that they are comparable in accuracy of detection, but have a much smaller computational load.

First real mimicry attacks were described relatively recently [21,22]. Ideas on how to avoid detection by anomaly based NIDS are even newer. Based on suggestions published in [23], papers [24,25] show how to create attacks that current NIDS cannot detect. Paper on Anagram [18] is the only one that partly addresses mimicry issue.

Detection based on multi-byte payload analysis that prevents mimicry attacks is an approach that will be presented next.

3 Proposed Detection Method

Proposed detection method continues work of other researchers mentioned above. The idea is to combine payload partitioning into words from [20] with an improvement of simple anomaly score calculation in [18]. This should provide a simple, easily storable model, and a fast detection. A new part of the model data on transition from one word to another. The most important extension of the model is use of key.

3.1 Model Building Principles

The model of the normal packet payload depends on the way of partitioning payload into words. Term “word” has the same meaning as in written humane language: sequence of symbols between two delimiters. In human written language delimiters are spaces and punctuation marks. In the payload symbols that separate words are some predetermined bytes. The selection of delimiters is not unique or obvious as in written human language, and might depend on the application level protocol used.

Since each set of delimiters produces a unique set of normal payload words, and accordingly a unique model it may be used as a key. Each implementation should use an individual set of delimiters. The important question is how a particular set of delimiters affects the detection capability of an IDS. This problem will be addressed in the next section by investigating relationship between keys detection capabilities.

3.1.1 Learning and Detection

The model of normal behavior is built during the training phase. Normal, attack free, payloads is partitioned into words. Words in normal traffic with their frequencies constitute the first part of the model. A malicious payload will have word frequency distribution significantly different from the normal payload.

The other part of the model is word transition frequency distribution. Probability of appearance of a word in a language usually depends on the previous word. Assumption is that payload words should behave in the same way. A malicious payload should show a word order different than in a normal payload. During the training phase appearances of any pair of words are counted and stored.

In the detection phase a new packet payload that was not used in learning, is analyzed and assigned two scores, named word score Sw and transition score St.

The word score is calculated using the following formula:

$$S_w = \frac{1}{k}\sum_{i=1}^{k}\frac{1}{n(w_i)} \tag{1}$$

In this formula k is the number of words in a payload and $n(w_i)$ is the number of appearances of the word w_i in the learned model. For the words that were not seen in the normal traffic thus having $n(w_i) = 0$, the corresponding term in the sum is set to two instead of infinity. This value is twice the value for the word that was seen only once during training, that has $n(w_i) = 1$. In this way words that did not appear in training payloads will have the highest contribution to anomaly score, but that contribution will not be such to make contribution from rare words completely insignificant.

The formula used to calculate the transition score follows:

$$S_t = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{n(t_i)} \tag{2}$$

In this formula m is the number of transitions in a payload and $n(t_i)$ is the number of appearances of the transition t_i in learned model. Value of m is one less than k in (1). Similarly to the formula for the word score, transitions not seen during training will have $n(t_i) = 0$. Again, term $1/n(t_i)$ is set to two instead of infinity. The same reasoning can be applied to formula (2) as for formula (1).

For a faster calculation of scores by avoiding division, of the inverse values of $n(w_i)$ and of $n(t_i)$ are calculated and stored before the detection phase.

Scores based on formulas (1) and (2) are used to obtain total score S. To keep the number of false positive detections low, both word score and transition score must be high to have a high total score. For this reason the total score is found as their product.

$$S = S_w * S_t \tag{3}$$

4 Testing

Although proposed detection method should work with any protocol, HTTP is selected for testing, since it is probably the most widely used application protocol nowadays. Standard TCP HTTP port 80 is generally open on most firewalls. Increasing number of Web applications increases number of vulnerabilities. According to SANS institute statistics Web application vulnerabilities represent almost half of all vulnerabilities discovered in 2007 [26]. The open port and application vulnerabilities attract attackers. Web based attacks make majority of all attacks [27]. HTTP protection seems to be the most needed and any improvements would be welcome.

The major issue in IDS testing is evaluation dataset. A very active recent thread on Security Focus IDS related mailing list showed how a good evaluation dataset is badly needed. The current problems in HTTP testing methods are pointed out in [28]. The best known, and once most widely used data sets for IDS testing, were created by DARPA/MIT Lincoln Laboratories in 1998 [29] and 1999 [30]. There are two main reasons why DARPA data sets are not adequate for current HTTP IDS. Both traffic and attacks in those data sets are obsolete. There are only four web attacks in the data sets. Recent HTTP anomaly detection papers mostly use their own data sets [7,8,9,10,17,18]. Those that do use DARPA data sets, also use their own [16,20].

In this paper EE department of the Sarajevo University traffic was used for testing. Real traffic with outside world was recorded for 12 days in November 2007. Traffic was recorded on the inner side of department external router / firewall. Traffic was first cleaned from attacks using fully tuned Snort, signature based NIDS, with latest signatures combined with manual inspection. The cleaned traffic therefore may have only few attacks. It is a question if it is possible at all to get real traffic that is guaranteed to be attack free [31]. For detection purposes recorded traffic was intentionally combined with attacks generated using Metasploit framework. Attack details are provided later.

4.1 Initial Set of Delimiters

The testing had two parts. The aim of the first part was to assess the ability of proposed score to detect anomalous payload using the delimiter set generating maximal number of meaningful words in HTTP and HTML protocols. The influence of key size and content on the learning process, the normal behavior model and detection was the task in the second part.

Initial key of 20 delimiters taken from [20] were:

CR LF TAB SPACE , . : / \ & ? = ( ) [ ] " ; < >

A set of delimiter symbols may result in any length of a words and any total number of words. In order to keep the number of words down, and thus get a smaller model, allowed word length was limited to the range from 3 to 16. Words shorter than 3 bytes are ignored, since they hardly could be an important part of an attack. Also, a word always ends after 16 consecutive bytes, and a new word begins.

4.1.1 Learning and Detection

The recorded traffic, cleaned from attacks, was used to build a model of normal payload. All incoming traffic packet payloads were scanned and partitioned into words, Words, word pairs, and their respective frequencies were stored in a hash table. Number of learned words leveled after the 96 hours of learning. At this point it was concluded that there would be no significant increase in number of words in by continuing the training. Testing should confirm if the conclusion was correct. Total number of learned normal words was about 33 000.

Proposed model consists of normal word frequencies and word transition frequencies. A 33 000 x 33 000 matrix would be needed to store all the word pairs and their frequencies. The distribution of word frequencies showed that a small number of words appeared very frequently, while a huge number appeared only several times. This word distribution was used to create a reduced size transition matrix, instead of full size but sparse matrix. First, the set of words that appear more than ten times in traffic was found. Then, a set of pairs of these words was selected to enter reduced size matrix. The resulting matrix size was 80 times smaller than the full matrix. All the other word pairs were considered as rare, and they were assigned a high partial anomaly score. A question remained would a similar word distribution hold for a different set of delimiters.

To test the ability to detect attacks, packet payload was scored using formula (3). The test checked how detection method scores real attacks. For this purpose 17 HTTP attacks were created using Metasploit. Attacks with related vulnerabilities, CVE references and used attack payloads are given in Table 1. It should be pointed out that the attacks that were used mostly exploit vulnerabilities in Web servers. Attacks that target Web applications, like SQL injection, XSS or similar should be further tested.

Table 1. Attacks with related vulnerabilities and used payloads

No.	Vulnerability	CVE	Payload
1	Apache Chunked-Encoding	2002-0392	adduser
2	Apache Chunked-Encoding	2002-0392	meterpreter-reverse_tcp
3	Apache Chunked-Encoding	2002-0392	shell-reverse_http
4	Apache mod_jk overflow	2007-0774	adduser
5	Apache mod_jk overflow	2007-0774	shell-reverse_tcp
6	Apache mod_rewrite	2006-3747	shell-bind_tcp
7	Apache mod_rewrite	2006-3747	shell-reverse_tcp
8	Apache mod_rewrite	2006-3747	vncinject-reverse_http
9	Apache mod_rewrite	2006-3747	vncinject-reverse_tcp
10	IIS 5.0 IDQ Path Overflow	2001-0500	shell-reverse_http
11	IIS 5.0 IDQ Path Overflow	2001-0500	shell-reverse_tcp
12	IIS ISAPI w3who.dll	2004-1134	exec
13	IIS ISAPI w3who.dll	2004-1134	shell-reverse_tcp
14	Oracle 9i XDB HTTP PASS	2003-0727	shell-reverse_tcp
15	Xitami If_Mod_Since	2007-5067	shell-reverse_tcp
16	HP OpenView Network Node Manager CGI Buffer Overflow	2007-6204	shell-reverse_tcp
17	BSD Mercantec SoftCart CGI Overflow	2004-2221	shell-reverse_tcp

Six days of traffic that were not used during learning were combined with attacks from Table 1. Results of the test are presented on ROC curve in Fig. 1. Threshold for an anomalous score was varied in the range from 0.2 to 2.0. False positive rate scale goes from 0 to 0.005 to provide enough detail in the part of the picture where ROC changes. Results are equal or better then the results reported by other researchers. Real comparison is difficult due to reasons explained in [28].

4.2 Arbitrary Sets of Delimiters

Since a set of delimiters is a key, each implementation should use an individual set of delimiters. A set of delimiters not optimized for the highest percentage of meaningful words, might negatively affect the detection capability. In addition, number of words might increase substantially. Also, word distribution might change, preventing usage of reduced matrix for transition storage. To test above issues various 120 keys were created. Key sizes were 15, 20, 25, 30, and 30 different sets were made for each size. Delimiters were chosen using function “rand” to generate a number between 0 and 255. For each of these keys, the initial set tests were repeated.

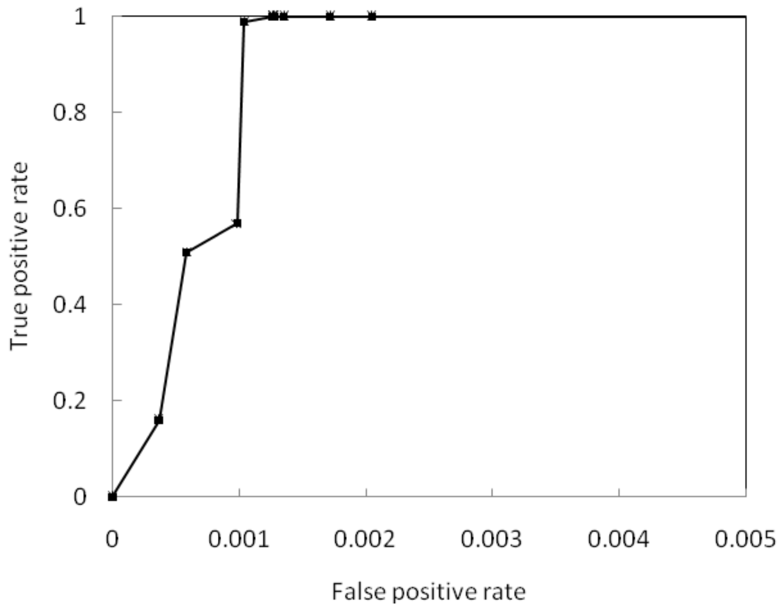


Fig. 1. ROC curve for initial set of delimiters

4.2.1 Results

For every set of delimiters, number of learned words leveled after 96 hours of learning, as was the case with initial set of delimiters. For those sets, total number of learned words was between 40000 and 50000, for 20% to 50% increase. The increase was expected, but it is not huge. It did not have any important adverse effect on model building or storage. Word distribution has not changed for any of the sets. This is important since the distribution was basis for the reduction of the model size.

Curves in Fig 2 present min, average and max true positive rate for fixed false positive rate points among all 30 delimiter sets for each of four set sizes. Although results vary, there are some keys that are as effective as the initial one. Before deploying a key for practical operation, experiments could be performed to find a good one. Effort to break the method even for small set of good keys is still high enough to prevent key guessing and practical attacks.

Since random delimiters can be used, new models of normal payload can be constantly built. While using one model to detect attacks, normal traffic is tokenized to words with a new set of delimiters. After system has trained enough a new model should be tested. If it is good, it can be used for detection right away or as needed. This enables easy change of keys in regular intervals or on request. Also, the model is now dynamic and can always represent current normal payloads.

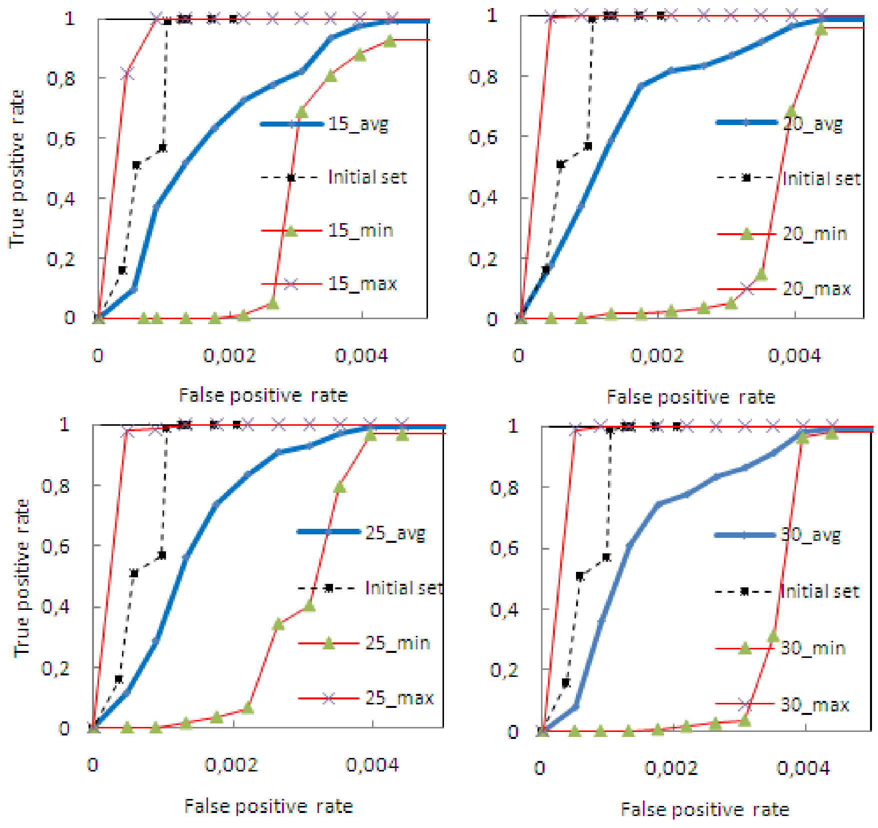


Fig. 2. Average, min and max ROC curves for random sets of 15,20, 25 and 30 delimiters

5 Conclusion

Keyed intrusion detection system is an implementation of the open design principle. Detection method is public but each implementation uses different secret key. Model of normal behavior is key dependent. Creation of attacks that fit the model is difficult, if not impossible, without the knowledge of the key.

Detection method analyses network packet payloads. Payload analysis enables detection of application level attacks. Payloads are tokenized to words. Model of normal payload is built on frequency distribution of words and transitions between them. Word boundaries are defined by set of delimiters, selected byte values. Set of delimiters plays a role of the key. Different sets of delimiters result in different words and different model.

Method is not protocol dependent. It was tested with HTTP traffic, but should work with all text based application level protocols. This should be tested and it is planned for future work. Future work will further check set of delimiters selection. It has a role of a key and there might be some weak keys.

Keyed IDS might be built with basic detection method different from the one proposed in this paper. It is a novel idea and there might be some better implementations.

References

1. Rash, M., Orebaugh, A.D., Clark, G., Pinkard, B., Babbin, J.: Intrusion Prevention and Active Response: Deploying Network and Host IPS, Syngress (2005)
2. Mrdovic, S., Perunicic, B.: Kerckhoffs' Principle for Intrusion Detection. In: The 13th International Telecommunications Network Strategy and Planning Symposium, Networks 2008, pp. 1–14 (2008)
3. Mrdovic, S., Perunicic, B.: NIDS Based on Payload Word Frequencies and Anomaly of Transitions. In: Third International Conference on Digital Information Management, ICDIM 2008, pp. 334–339 (2008)
4. Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., Zhou, S.: Specification-based anomaly detection: a new approach for detecting network intrusions, pp. 265–274. ACM, Washington (2002)
5. Mahoney, M.V.: Network traffic anomaly detection based on packet bytes, pp. 346–350. ACM, Melbourne (2003)
6. Zanero, S., Savaresi, S.M.: Unsupervised learning techniques for an intrusion detection system, pp. 412–419. ACM, Nicosia (2004)
7. Kruegel, C., Vigna, G.: Anomaly detection of web-based attacks, pp. 251–261. ACM, Washington D.C (2003)
8. Kruegel, C., Vigna, G., Robertson, W.: A multi-model approach to the detection of web-based attacks. Computer Networks 48, 717–738 (2005)
9. Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.A.: Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks (2006)
10. Ingham, K., Somayaji, A., Burge, J., Forrest, S.: Learning DFA representations of HTTP for protecting web applications. Computer Networks 51, 1239–1255 (2007)
11. Akritidis, P., Markatos, E.P., Polychronakis, M., Anagnostakis, K.: Stride: Polymorphic sled detection through instruction sequence analysis (2005)
12. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic Worm Detection Using Structural Information of Executables (2005)
13. Wang, X., Pan, C., Liu, P., Zhu, S.: SigFree: a signature-free buffer overflow attack blocker, p. 16. USENIX Association, Vancouver (2006)
14. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based Detection of Non-self-contained Polymorphic Shellcode
15. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the infeasibility of modeling polymorphic shellcode, pp. 541–551. ACM, Alexandria (2007)
16. Wang, K., Stolfo, S.J.: Anomalous Payload-Based Network Intrusion Detection (2004)
17. Wang, K., Cretu, G., Stolfo, S.J.: Anomalous Payload-Based Worm Detection and Signature Generation (2005)
18. Wang, K., Parekh, J., Stolfo, S.: Anagram: A Content Anomaly Detector Resistant to Mimicry Attack, pp. 226–248 (2006)
19. Vargiya, R., Chan, P.: Boundary Detection in Tokenizing Network Application Payload for Anomaly Detection (2003)
20. Rieck, K., Laskov, P.: Language models for detection of unknown attacks in network traffic. Journal in Computer Virology 2, 243–256 (2007)
21. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems, pp. 255–264. ACM, Washington (2002)
22. Tan, K., Killourhy, K., Maxion, R.: Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, pp. 54–73. Springer, Heidelberg (2002)

23. Kolesnikov, O., Lee, W.: Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic, College of Computing, Georgia Tech. (2005)
24. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks, p. 17. USENIX Association, Vancouver (2006)
25. Fogla, P., Lee, W.: Evading network anomaly detection systems: formal reasoning and practical techniques, pp. 59–68. ACM, Alexandria (2006)
26. SANS Institute, SANS Top-20 2007, Security Risks, Annual Update (2007)
27. Internet Security Threat Report, Symantec Corporation (2008)
28. Ingham, K., Inoue, H.: Comparing Anomaly Detection Techniques for HTTP. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 42–62. Springer, Heidelberg (2007)
29. Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R., Zissman, M.: Evaluating intrusion detection systems: the 1998 DARPA off-line intrusion detection evaluation, vol. 2, pp. 12–26 (2000)
30. Richard, L., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA off-line intrusion detection evaluation. Computer Networks 34, 579–595 (2000)
31. Gates, C., Taylor, C.: Challenging the anomaly detection paradigm: a provocative discussion, pp. 21–29. ACM, Germany (2006)

Modeling and Containment of Search Worms Targeting Web Applications

Jingyu Hua and Kouichi Sakurai

Department of Informatics, Kyushu University,
Fukuoka 812-0053, Japan
{huajingyu,sakurai}@itslab.csce.kyushu-u.ac.jp

Abstract. Many web applications leak sensitive pages (we name them *eigenpages*) that can disclose their vulnerabilities. As a result, some worms like *Santy* locate their targets by searching specific eigenpages in search engines with well-crafted keywords. Such worms are so called *search worms*. In this paper, we focus on the modeling and containment of these search worms. We first study the influence of the eigenpage distribution on their spreading by introducing two propagation models: U-Model assuming eigenpages uniformly distributed on servers and PL-Model assuming the distribution follows a power law. We show that the uniform distribution maximizes the spreading speed of the search worm. Then we study the influence of the page ranking and introduce another propagation model: PR-Model. In this model, search results are ranked based on their PageRank values and the relative importance of their resident servers. Finally, we propose a containment system for search worms based on honey-page insertion: a small number of fake pages which will induce visitors to pre-established honeypots are randomly inserted into search results, and then infectious can be detected and reported to search engines when their malicious scans hit honeypots. We study the relationship between the containment effectiveness and the honey-page insert rate with our propagation models and find that the Santy worm can be almost completely stopped at its early age by inserting no more than 2 honey pages in every 100 search results, which is extremely effective.

1 Introduction

1.1 Background and Motivation

Due to many reasons, a lot of worms target web servers. There are many ways for a worm to find vulnerable servers: they can simply scan the whole IP address space randomly as the Code Red Worm did [1], or seek help from Botnets, who might hold hitlists of vulnerable servers. However, we should not leave out another important resource: search engines, which are regarded as largest warehouses of web contents. Because of security negligence, web servers may expose sensitive pages disclosing their vulnerabilities to the public. Thereby, once these pages are crawled by search engines, worms can accurately locate their targets by searching these pages in search engines with carefully-crafted keywords. We name these pages eigenpages. They might be server pages with default titles, error pages generated by software, etc.

Actually, several worms [2, 3] using search engines to spread have appeared in the past few years. They are called search worms. Among them, Santy [2], which is created in Perl exploiting vulnerability in phpBB bulletin system, is the most famous one. Since URIs of pages of these vulnerable systems usually contain the phrase *"viewtopic.php"*, this worm searches its potential prey on search engines (Google, Yahoo, etc.) with the key words like: *allinurl: "viewtopic.php" "topic=12345"*, where the random number *12345* is used to increase the diversity of search results. According to statistics [4], this worm spread extremely fast: it was released on Dec 20th, 2004 and as many as 40,000 sites had been infected two days later when Google put defenses in place. Fortunately, although Santy worm infects Web sites, it does not infect computers used to view those sites. Otherwise, its destructive power would be unprecedented.

Since search worms bring a great security threat on the internet, it is of great importance to carefully characterize these search worms and develop efficient containment strategies for them. In this paper, we right focus on the modeling and containment of Santy-like search worms targeting web applications.

1.2 Related Work

To our best knowledge, although the first version of Santy was found as early as in 2004 [2], there is not so much research work on search worms by now, especially on the modeling of their propagation.

Most previous modeling works are focused on those traditional worms adopting random scanning strategy within the whole IP address space. The basis of these worms' modeling is the classical epidemic model [5]. It made a homogeneous assumption that an infected host has the equal probability to infect any vulnerable host. Denoted by $I(t)$ the number of infected hosts at time t; V the total number of vulnerable hosts. The epidemic model is:

$$\frac{dI(t)}{dt} = \beta I(t)[V - I(t)] \tag{1}$$

where β is called the *pairwise rate of infection.* Based on this model, Staniford et al. [6] presented a "Random Constant Spread" (RCS) model to characterize the propagation of uniform scan worms:

$$\frac{dI(t)}{dt} = \frac{\delta}{N} I(t)[V - I(t)] \tag{2}$$

where δ is the scan rate and N is the size of the scanning space. Later, RCS was extended by many researchers to analyze real worms so as to guide the designing of containment system for these worms [7, 8, 9, 10].

The concept of search worm was first introduced by Provos et al. [11]. In their work, they described the basic propagating steps of search worms and analyzed the measurements taken from MyDoom.O and Santy outbreaks. They argued that signature-based protections were ill-suitable for search worm. Thus, they proposed an algorithm that prevented worm propagation based on analyzing the result set: During indexing, pages belonging to vulnerable servers or contain potential infection targets were tagged. Then if search results including a large fraction of tagged pages were detected, they inferred that the query was due to a search worm and returned only results that have not been tagged. The limitation of this containment mechanism is that it's hard to tag all the vulnerable servers. After all, servers currently thought secure may also contain unknown exploitable security holes.

Johnny [12] named malicious queries for vulnerable servers *Google Dorks*. He is maintaining a Google Hacking Database (GHD) to collect emerging Google Dorks. By now, 1423 entries have been recoded. Search engines can make use of this database to perform query filtering, as a result, search worms using known Google Dorks can be blocked. However, for search worms using zero-day Google Dorks, this signature-based mechanism is helpless.

Based on the GHD, Riden et al. [13] constructed a Google Hacker Honeypot to provide reconnaissance against attackers that use search engines as a hacking tool against web resources. They emulated a vulnerable web application which contains honey pages that can be searched by Google Dorks in the GHD. Then, hackers might be induced to this honeypot by search engine and their malicious activities could be recorded for a further study. We think, if cooperating with search engine, such honeypot can be extended to perform search worm containment. Our containment system proposed in this paper is right such an extension.

1.3 Challenge Issues

For the modeling, since search worms spread in different ways with traditional worms, there are some unique properties that may affect their propagation. For example, since scans of these worms are counted on eigenpages returned by search engine, their propagation may have a solid relation with the distribution of eigenpages: servers containing more eigenpages are more likely to be scanned. In addition, due to the ranking inherent in the returned results, a search worm may encounter many result collisions across subsequent queries and popular sites attract more infection attempts, which also affect its propagation performance. Therefore, we have to construct propagation models for search worms by taking these factors into consideration.

For the containment, since search engine is the SPOF (single point of failure) of search worms, the easiest way is to detect and stop malicious queries made by search worms directly on the search engine. However, instead of limits we described in the last section, query filtering technologies such as those based on result set analyzing or GHD may also bring great side effects to the normal using of search engine. In our opinion, a query is good or evil is not determined by the query itself but by what the launcher does to search results. Actually, evil queries can be also used for good. For example, we can make use of Google Dorks to discover security holes on our own web site. As search engines simply deny these queries, they also give up many useful capabilities. Therefore, we need a more advanced mechanism to detect and contain search worms accurately and effectively.

1.4 Our Contributions

In this paper, we study the above two challenge issues and make the following contributions in brief:

- We present a virtual search worm abstracted from Santy and detail its propagation steps.
- We study the effects of eigenpage distribution on the propagation of the search worm. For this purpose, we introduce two propagation models: U-Model assuming eigenpages uniformly distributed on servers and PL-Model assuming eigenpage distribution follows a power law. We show that the uniform distribution maximizes the spreading speed of the search worm.
- We study the effects of page ranking on the propagation of the search worm. For this purpose, we introduce another model: PR-Model, where search results are ranked based on their PageRank values and the relative importance of their resident servers. Simulation shows the PR-Model spreads much slower than the U-Model and PL-Model.
- Eventually, we introduce a containment system for the search worm. In this system, search engine randomly insert honey pages among search results. These pages are fake ones and pointing to preestablished honeypots. No user can visit them except those induced by search engines. Then, infectious can be detected when their malicious scans hitting honeypots. Detected infectious will be reported to search engine and further queries from them are denied. Since infectious are blocked based on detecting malicious scans against search results, limits of using query filtering can be well overcome. We analyze the effectiveness of this containment system under the help of our propagation models and solve two core problems:
  (1) What value should the honey-page insert rate take in order to contain the prevalence rate of a search worm to a specific requirement?
  (2) Does such value always exist for arbitrary containment requirements?
  Simulations show that the Santy worm can be almost completely stopped at its early age by inserting no more than 2 honey pages in every 100 search results. This brings negligible side-effects to the normal using of search engines.

1.5 Page Organization

The remainder of the paper is structured as follows. Section 2 describes a typical search worm designed by us. Section 3 presents two propagation models: U-Model and PL-Model for this search worm. They make different assumptions with the distribution of the eigenpages. Section 4 explores the impacts of page ranking to the propagation of the search worm. Section 5 presents a containment system for the search worm based on our models. Section 6 summarizes our contributions, provides some discussions and directions for future work.

2 A Virtual Search Worm

In this section, we present a virtual search worm that we will analyze in the following sections. It is abstracted from the Santy Worm but stronger. Although it is not real, we can still disclose many basic characteristics of search worms by analyzing it.

Suppose this worm targets an unknown flaw of a server application, which exposes some eigenpages to the search engine. These eigenpages contain some special keywords in specific locations. Thus, vulnerable servers can be pinpointed by searching these keywords in the search engine. Let N and V denote the total number of suspicious servers containing eigenpages and the total number of vulnerable servers really suffer the flaw, respectively. Obviously, $N \geq V$. We assume one server can be only infected once. Then, once a vulnerable server is infected by the worm, it will repeat the following infection cycle to propagate itself:

(1) Search keywords "*special-keywords and random-keywords*" in a search engine. *special keywords* are used to make sure all the search results are eigenpages that belong to the N suspicious servers. *random-keywords* are used to increase the diversity of search results. Here, we make an attacker-favorable assumption that because of using the *random-keywords*, the search engine randomly selects m eigenpages as search results for a specific query. We say this assumption is hacker-favorable because it will maximum the diversity of search results, which is expected by the hacker.

(2) Select δ pages from the search results to scan their host servers. The select mechanism will be covered later.

(3) Once a vulnerable server is found, exploit its flaw and add it to a botnet. Then, it automatically downloads the worm code and begins this infection cycle on itself, too.

For convenience, we provide a list of the variables used in this paper in Table 1. Now let's consider the propagation modeling of the above search worm.

Table 1. Notations

Symbol	Explanation
N	The count of servers containing eigenpages
V	The count of vulnerable servers
δ	The number of scans made by each infected server in each round
I	The count of infected servers
P	The total count of eigenpages
ε	Density of vulnerable servers, $\varepsilon = \frac{V}{N}$.

3 Modeling the Propagation of the Search Worm

3.1 Effects of Eigenpage Distribution

Since the scans launched by the search worm are totally counted on the eigenpages returned by the search engine, we intuitively feel that those servers containing more eigenpages are more likely to be scanned. In other words, the propagation of the search worm may have a solid relationship with the distribution of the eigenpages. In this section, we look into this issue by proposing two propagation models with different assumptions on the distribution of eigenpages.

Before that, we make another attacker-favorable assumption that in the second step of the infection cycle, an infected server randomly selects δ ($\delta \leq m$) pages among the m search results to scan. As we known, as search engines rank their results, pages on popular servers are more likely to appear in front compared with others. Then, if a search worm selects δ top-ranking search results to exploit as the Santy does, they may encounter many scan collisions. As a result, the propagation slows down. But with the random select assumption, these effects caused by page ranking no longer exist. Therefore, we say this assumption is attacker-favorable. Actually, with the two attacker-favorable assumptions, the scan strategy can be simplified as randomly selecting δ pages among the whole space of eigenpages.

The first model we introduce is named U-Model, which assumes that eigenpages are uniformly distributed on suspectable servers, that is $p_1 = p_2 = \cdots = p_N = p$, where p_i is the count of eigenpages on the i-th suspicious server. This condition is not rare. For instance, if default pages generated by a vulnerable server application during its installing phase are used as eigenpages, they are really uniformly distributed.

Suppose that infected servers take one time tick to complete the whole infection cycle and no newly infected servers can begin to propagate themselves before the end of this cycle. This assumption might not be realistic, but it can simplify the model without significantly affecting the results. Denote by $I(t)$ the total number of infected servers by the end of the time tick t. Thus, $\delta I(t-1)$ is the total number of scans made by the infected servers during the t-th time tick. Then, under the two attacker-favorable assumptions, the probability that a specific suspicious server is hit by at least one scan during this period is $1-(1-\frac{p}{Np})^{\delta I(t-1)} = 1-(1-\frac{1}{N})^{\delta I(t-1)}$. Since the number of remained vulnerable servers by the time tick $t-1$ is $V-I(t-1)$, the total number of infected servers after the i-th time tick is

$$I(t) = I(t-1) + [V - I(t-1)][1-(1-\frac{1}{N})^{\delta I(t-1)}] \tag{3}$$

When $\frac{1}{N} \ll 1$,

$$I(t) = I(t-1) + [V - I(t-1)]\frac{\delta I(t-1)}{N} \tag{4}$$

which is the same as the RCS model defined by (2). Thus, we find that when eigenpages are uniformly distributed on servers, under the two attacker-favorable assumptions, the propagation of the search worm is equivalent to the worm adopting uniform scanning strategy. But, now, the scanning space has been reduced from the whole IP address space to servers exposing eigenpages. So the propagation speed can be greatly improved.

Although uniform distribution assumption is reasonable under some circumstances, it's impossible in most time. Just consider the Santy worm, page population varies among different phpBB bulletin systems. Huberman et al. [14] pointed out that the distribution of web pages among sites follows a power law. They analyzed pages of sites crawled by Alexa and Infoseek, which covered 259,794 and 525,882 sites, respectively. Both data sets displayed a power law and appeared as a straight line on a log-log plot. Thereby, let's consider the propagation model of the search worm when the eigenpage distribution follows the power law.

According to the definition of the power law, the probability that the number of eigenpages p on a suspicious server is greater than x is

$$prob(p > x) = (\frac{p_{min}}{x})^{\sigma} \tag{5}$$

where p_{min} is the minimum value of p and $\sigma \geq 1$ is the exponent parameter. In order to simplify the problem, we divide servers into k groups and assume servers belonging to the same group have the same number of eigenpages. Denote by p_i the count of eigenpages on each server belonging to the i-th group. We assume $p_i = p_1 \times 10^{i-1}$ and the number of servers in the i-th group is

$$a_i = N \cdot [prob(p > p_i) - prob(p > p_{i+1})] = N \cdot [(\frac{p_{min}}{p_i})^{\sigma} - (\frac{p_{min}}{p_{i+1}})^{\sigma}] \tag{6}$$

If we keep the two attacker-favorable assumptions, the number of infected servers in the i-th group after the t-th time tick is

$$I(t,i) = I(t-1,i) + [V_i - I(t-1,i)][1 - (1 - \frac{p_i}{P})^{\delta I(t-1)}]$$

where $P = \sum_{i=1}^{k} a_i p_i$ denotes the total number of eigenpages and V_i denotes the vulnerable servers in the i-th group. We assume the density of vulnerable servers is the same for each group. Thereby, we derive a new spreading model named PL-Model for the search worm:

$$\begin{cases} I(t,i) = I(t-1,i) + [V_i - I(t-1,i)][1 - (1 - \frac{p_i}{P})^{\delta I(t-1)}] \\ I(t) = \sum_{i=1}^{k} I(t,i) \end{cases} \tag{7}$$

Using (7), we do some simulations to study the characteristics of the PL-Model. We are mainly focused on the effects of the power law exponent. Fig.1(a) shows the propagation of the PL-Model with different power law exponents. Fig.1(b) shows the time ticks the PL-Model used to infect 95% of vulnerable servers with different power law exponents. We can find that as the power law exponent σ increases, it takes the worm less time to spread. In addition, compared with the U-Model, the PL-Model spreads slower. Actually, we have the following theorem:

Proposition 1. *Among different distributions of eigenpages, the uniform distribution optimizes the performance of search worms.*

Proof. Assume $s_1, s_2, \cdots, s_n$ are n uninfected vulnerable servers by the t-th time tick. Denote by $p_1, p_2, \cdots, p_n$ the numbers of eigenpages on them. Then, the next time tick will have $\triangle = \sum_{i=1}^{n}[1-(1-\frac{p_i}{P})^{\delta I(t)}]$ newly infected servers. Let $r_i = \frac{p_i}{P}$, we have:

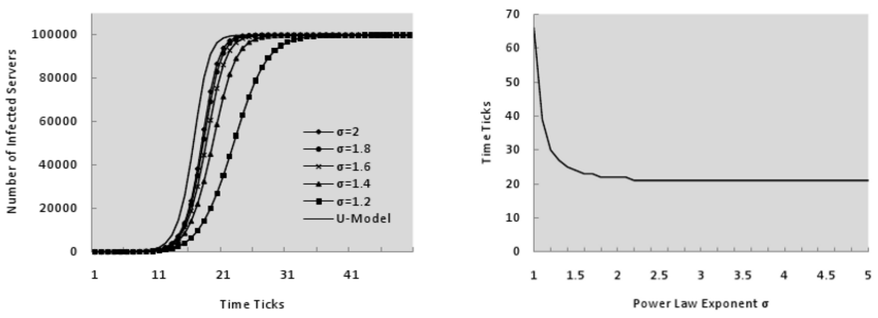
$$\begin{cases} \triangle = n - \sum_{i=1}^{n}(1-r_i)^{\delta I(t)}; \\ \sum_{i=1}^{n} r_i = 1, \forall i, 1 > r_i > 0; \end{cases}$$

According to the mean value inequality, under the condition $(1 - r_i) > 0$, $\sum_{i=1}^{n}(1 - r_i)^{\delta I(t)} \geq n \cdot (\frac{\sum_{i=1}^{n}(1-r_i)}{n})^{\delta I(t)}$ with equality holding if and only if $1 - r_1 = 1 - r_2 = \cdots = 1 - r_n$. Since $\sum_{i=1}^{n} r_i = 1$, we get

$$\triangle <= n - n \cdot (\frac{n-1}{n})^{\delta I(t)} \tag{8}$$

The maximum value is obtained if and only if $r_1 = r_2 = \cdots = r_n$, which means the eigenpages are uniformly distributed on the servers. Thereby, we obtain the conclusion.

Now, based on this theorem, above simulation results are easy to understand: As σ increases, the power law distribution becomes closer to the uniform distribution, where the propagation reaches the highest speed.



(a) Propagation of the search worm with different power law exponents

(b) Time ticks used to infect 95% of vulnerable servers with different power law exponents

Fig. 1. Effects of the power law exponent (All cases are for $N = 10^7$, $V = 10^5$, $\delta = 100$ and $p_{min} = 10$)

3.2 Effects of Page Ranking

With the second attacker-favorable assumption, the effects of page ranking can be eliminated. However, in most time, this assumption is impossible. Usually, for security, users are only allowed to access a limited number of top-ranking results for their queries in search engines (e.g., in Google, only the first 1000 search results are accessible for a specific query). Thus, search worm like Santy has no choice but to scan the δ top-ranking pages of each query. As a result, they will encounter many result collisions across subsequent queries which affect their propagation performance greatly. In this section, we consider this issue.

In brief, the ranking of a page is determined by two factors: the keyword relevance and the page importance. In this paper, we just consider the later. By now, PageRank, which was developed by Google, is the most famous algorithm used to measure the importance of pages. The PageRank value of a particular page is roughly based upon the quantity of inbound links as well as the PageRank values of the pages providing the links. Pandurangan et al. [15] and Litvak et al. [16] claimed that the distribution of PageRank values follows the power law and the exponent is about 1.1 for cumulative plots. Google simplified PageRank value into $0-10$ on a logarithmic scale. The base is a secret and estimated to be between 5 and 10. We use 6. We also assume the PageRank value is an integer and actual values from 6^d to 6^{d+1} are all simplified to d. Then, the probability that the PageRank value of a specific page equals to d $(0 \le d \le 10)$ is

$$prob(PR = d) = \begin{cases} (\frac{1}{6^d})^\alpha - (\frac{1}{6^{d+1}})^\alpha & 0 \le d < 10 \\ (\frac{1}{6^{10}})^\alpha & d = 10 \end{cases} \tag{9}$$

When PageRank values of two pages are the same, we assume their rankings are determined by the relative importance of their resident sites. Since search engines prefer larger sites, we simply assume a server containing more pages are more important.

Algorithm 1. Calculate δ_i: expected number of scans falling in the i-th group among the total δ scans launched by an infected server during an infection cycle

```
1: $total \leftarrow 0$, $pr \leftarrow 10$, $i \leftarrow k$
2: while $pr \ge 0$ do
3:   while $i \ge 1$ do
4:     $temp = m \times \frac{a_i p_i}{P} \times Prob(PR = pr)$ /* Expected number of search results with
       a PageRank of $pr$ and belonging to the $i$-th group */
5:     if $total + temp > \delta$ then
6:       $\delta_i$ += $\delta - total$
7:       return
8:     else
9:       $\delta_i$ += $temp$
10:      $total$ += $temp$
11:      $i - -$
12:    end if
13:  end while
14:  $pr - -$
15:  $i \leftarrow k$
16: end while
```

Now, let's derive a new propagation model: PR-Model, for the search worm. In this model, for a specific query, the search engine still randomly selects m pages as the result set, but these pages are ranked and only the top δ ones are scanned. This model makes the same assumption with the PL-Model that eigenpages's distribution follows a power law. At the same time, we also divide servers into groups and assume servers belonging to the i-th group have the same number of eigenpages: $p_i = p_1 \times 10^{i-1}$. To simplify the problem, a server containing more eigenpages are considered larger. Thereby, servers belonging to the j-th group are more important than servers belonging to i-th group when $j > i$. Let δ_i denote the expected number of scans hit servers of the i-th group among the total δ scans launched by an infected serverin an infection cycle. δ_i can be computed by Algorithm.1. Then, we can derive the PR-Model as follows,

$$\begin{cases} I(t,i) = I(t-1,i) + [V_i - I(t-1,i)][1 - (1 - \frac{1}{a_i})^{\delta_i I(t-1)}] \\ I(t) = \sum_{i=1}^{k} I(t,i) \end{cases} \tag{10}$$

We simulate this model with $m = 10000$ and $\delta = 100$, and compare it with the PL-Model described in the last section. The results are present in Figure 2. We can find that the propagation is slowed greatly.

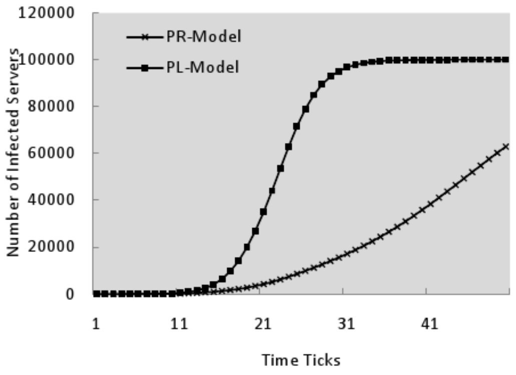


Fig. 2. Comparison between the PR-Model and the PL-Model (All cases are for $N = 10^7$, $V = 10^5$, $\sigma = 1.2$, $\delta = 100$, $p_{min} = 10$, starting on a single machine)

4 Containment of the Search Worm

One of the goals of modeling the spread of worms is to be able to detect and contain them. In this section, we present a concept containment system based on honey-page insertion and use our propagation models to investigate its effectiveness.

4.1 Basic Idea

Some security systems detect traditional worms by monitoring unused IP addresses. This is because normal hosts rarely visit unused IP addresses, but once they are infected by worms employing uniform scanning strategy, they may scan those unused IP addresses. We extend this idea to detect the search worm. We make search engines randomly insert honey pages pointing to honeypots into search results. These pages are fake ones, i.e., no user could request them except those induced by search engines. Then, if an infected server selects a honey page into its target vector in an infection cycle, it will exploit a honeypot and be identified. Of course, normal users may also click links of honey pages by chance and visit honeypots. Therefore, honeypots must have the intelligence to distinguish between legal accesses and evil accesses. This is a hot topic in the research area of honeypots but beyond the scope of this paper. We simply assume our honeypots can always make correct judgement. When infected servers are detected, honeypots can report them to search engines who will deny their further queries to contain the worm propagation. Let's investigate the possibility of using this idea to build a containment system for the search worm.

Suppose μ $(0 \leq \mu < 1)$ is the insert rate of honey pages, which means if the search engine returns m results for a specific query, the expected number of honey pages among them is μm. In other words, for a specific result, the probability that it is a honey page is μ. Then, if an infected server scans δ of the N results, it hits a honey page with the probability: $\beta = 1 - (1-\mu)^\delta$. When a scan from an infected server hits a honey page, we say it is observed. Let D_t denote the number of observed infected servers by the then end of the time tick t. Then, at the time tick $t+1$, the expected number of newly observed infected servers is $\beta \cdot [I(t) - D(t)]$. Therefore,

$$D(t+1) = D(t) + [I(t) - D(t)][1 - (1-\mu)^\delta] \qquad (11)$$

Based on (11) we can compare the evolution of observed infected servers with the evolution of actual infected servers. Fig.3 shows an example in which we simulate the U-Model using parameters of Santy under different honey page insert rates. Since there were approximately 6 million web sites using phpBB system and among them about 4,0000 were infected during the Santy breakout [4], we simply assume $N = 6 \times 10^6$ and $V = 4 \times 10^4$. We can find that, in this case, the curve where $\mu = 0.01$ is very close to the real worm propagation using the U-Model. This means an infected server can be induced to scan honeypots in a very short time after it is infected even if the insert rate is very small. We obtain similar results when we simulate the PL-Model and PR-Model. Hence, it's quite possible for us to construct a containment system to detect and stop search worm by inserting a small number of honey pages in search results without greatly affecting the normal using of search engine.

4.2 Methodology

First we set up some honeypots in the internet. URIs of honey pages inserted into search results will induce visitors to these servers. They reply to connection requests as a normal server so as to deceive the machine at the other end. Then,

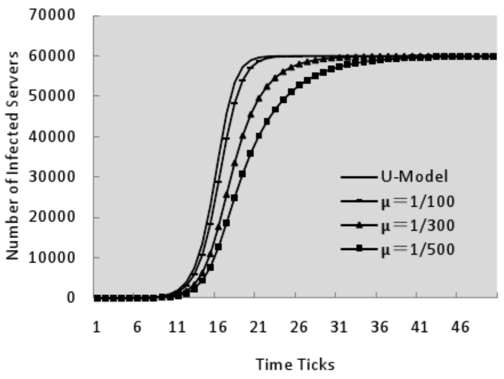


Fig. 3. Comparison between observed infected servers and actual infected servers in U-Model for different honey-page insert rates (All cases are for $N = 6 \times 10^6$, $V = 4 \times 10^4$, $\delta = 100$ and starting on a single machine)

if the access behavior of a visitor is considered malicious by a honeypot, its IP address will be reported to the search engine. Once the search engine receives such a report, the corresponding IP address is added into the blacklist and further queries from it are denied in a certain period of time. Thus, the infected servers can be divided into two categories: active ones and dead ones. Active ones are those haven't been put into the blacklist and still have the ability to propagate, while dead ones are those have been put into the blacklist and lose the ability to propagate. Since our detection method is relied on the real activities made by infectious against search results (honey pages), it can be much more accurate compared with those mechanisms based on query filtering.

For this simple containment system, one question needs to be answered: what value the honey-page insert rate should take to contain the final prevalence rate of a search worm to a specific level. Here, the final prevalence rate is the ration of vulnerable servers that have ever been infected before the worm finally dies out.

Since the U-Model spreads fastest, we mainly focus on it. With the above containment strategy applied, the U-Model becomes:

$$\begin{cases} A(t) = A(t-1) + [V - A(t-1) - D(t-1)]\frac{\delta A(t-1)}{N} - \beta A(t-1) \\ D(t) = D(t-1) + \beta A(t-1) \end{cases} \tag{12}$$

where A(t), D(t) denote the number of active infected servers and the number of dead infected servers by the t-th time tick, respectively. We call this model contained U-Model. Similarly, contained PL-Model and contained PR-Model can be also easily defined like this. We omit them here. Obviously, this is a discrete time model. Let's extend it to a continuous time model as follows:

$$\begin{cases} \frac{dA(t)}{dt} = \alpha V(t)A(t) - \beta A(t) \\ \frac{dD(t)}{dt} = \beta A(t) \\ \frac{dV(t)}{dt} = -\alpha V(t)A(t) \\ V = A(t) + D(t) + V(t) \\ \alpha = \frac{\delta}{N} \\ \beta = 1 - (1-\mu)^{\delta} \end{cases} \tag{13}$$

where $V(t)$ denotes the remained vulnerable servers at time t. This is equivalent to the classic Kermack-McKendrick Model [5], in which the worm will die out eventually, i.e., $A(t_{final}) = 0$. Let's consider the behavior of $A(t)$ in terms of $V(t)$ by dividing the first equation with the third equation:

$$\frac{dA(t)}{dV(t)} = -1 + \rho V(t)^{-1};$$

where $\rho = \beta/\alpha$. Hence, we obtain that:

$$A(t) = -V(t) + \rho \ln V(t) + const$$

Let's assume $V(0) \approx V$, $A(0) \approx 0$: the search worm starts with very few infected servers, then easy to derive that $const = V - \rho \ln V$. Thereby,

$$A(t) = -V(t) + \rho \ln V(t) + V - \rho \ln V. \tag{14}$$

Based on (13), we can answer the question mentioned earlier: in order to contain the final prevalence rate of a search worm to γ $(0 \leq \gamma \leq 1)$, i.e., $V(t_{final}) = (1-\gamma)V$ when $A(t_{final}) = 0$, ρ has to satisfy the following condition:

$$\rho = \frac{-\gamma V}{\ln(1-\gamma)} \tag{15}$$

Hence,

$$\mu = 1 - [1 + \frac{\gamma \varepsilon \delta}{\ln(1-\gamma)}]^{1/\delta} \tag{16}$$

where $\varepsilon = \frac{V}{N}$ is the density of vulnerable servers. It's easy to find that μ increases as γ decreases. Thereby, given a containment requirement, we can use (16) to compute the bottom boundary for the honey-page insert rate. For example, consider the Santy worm with few initial infected servers ($V = 4 \times 10^4$, $N = 6 \times 10^6$, $\delta = 100$). If its final prevalence rate is required to be contained below 1%, the honey-page insert rate should be larger than 0.011. For the PL-Model and PR-Model, since they spread slower than the U-Model, when the honey-page insert rate is set to μ according to (16), their final prevalence rates will be lower than γ.

We simulate the U-Model, PL-Model and PR-Model under the protection of our containment system using the parameters of the Santy worm, respectively. Fig.4 shows the final prevalence rates of these models for different values of μ with 1 initial infectious. As we can see from this figure, as μ increases, the final prevalence rate decreases. With the same μ, the final prevalence rates of the PL-Model and the PR-Model are lower than the U-Model. In addition, when $\mu = 0.011$, the final prevalence rates of three models are all contained below 1%. This means that the Santy worm can be almost completely prevented if search engines randomly insert no more than 2 pages in every 100 search results. This brings negligible side-effects to the normal using of search engines. Thereby, our containment system is extremely effective.

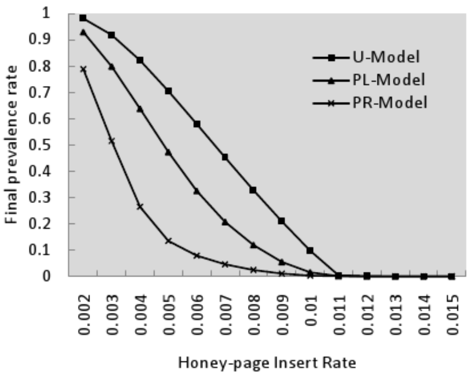


Fig. 4. Containment effects with different honey-page insert rates (All cases are for $N = 6 \times 10^6$, $V = 4 \times 10^4$, $\delta = 100$, starting on a single machine. The power law exponent in the PL-Model is 1.6. The power law exponent for the PageRank is 1.1.).

4.3 Discussion

In the above, we give the mathematic way to compute the honey-page insert rate for a specific containment requirement. However, we have to answer another question: does μ always exist for any given prevalence rate γ? Or else, is there a containment limit for our containment system? In this section, we answer this question.

According to (15), we can easily find that γ decreases as μ increases. Thereby, when μ takes its maximum value 1, γ will reach its minimum value γ_{min}. Obviously, If $0 < \gamma_{min} < 1$, μ does not exist for a prevalence rate $\gamma < \gamma_{min}$, i.e., the prevalence rate can be contained at best to γ_{min}. But if $\gamma_{min} \leq 0$, for any given γ $(0 < \gamma \leq 1)$, μ always exists. Let $\mu = 1$, and substitute it into (15), we have:

$$-\lambda\gamma_{min} = \ln(1 - \gamma_{min}) \tag{17}$$

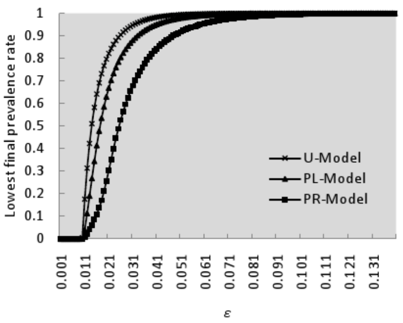
where $\lambda = \varepsilon\delta$. γ_{min} is just one solution of this equation. In order to simplify things, we introduce a new variable $x = 1 - \gamma$, then we have to solve the new equation:

$$\lambda(x - 1) = \ln x$$

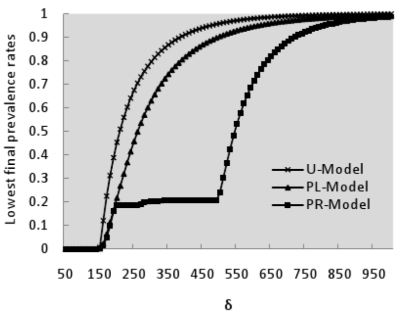
By considering the graphs of the two functions: $f(x) = \lambda(x-1)$ and $g(x) = \ln x$, we can get a conclusion that if $\lambda > 1$, $0 < \gamma_{min} < 1$, else if $0 < \lambda \leq 1$, $\gamma_{min} \leq 0$. Thereby, we obtain a theorem:

Proposition 2. *For our containment system based on honey-page insertion, if a search worm satisfies the condition that $\varepsilon\delta \leq 1$, its final prevalence rate γ ($0 < \gamma \leq 1$) can be contained to any requirement by using a specific honey-page insert rate computed by (16), otherwise, the final prevalence rate can be only guaranteed contained to γ_{min}, which is a solution of (17) in the interval* $(0, 1]$.

However, we have to notice that 1 is just a theoretical maximum value for μ. It's not practical. Because an insert rate equal to 1 means that results returned by search engines are all honey pages, which is ridiculous. Therefore, with the premise that our honey-page insertion should not greatly affect the function of search engines, we reduce the maximum value of μ to 0.1. After all, one honey page in every 10 search results is much more acceptable. With this honey-page insert rate, we simulate the three contained models again to investigate the containment limits of our containment system. Fig.5 presents the results. We still use the real data of the Santy worm, but in Fig.5 (a) we vary the ε and in Fig.5 (b) we vary the δ. We can find that, when $\varepsilon < 0.011$ and $\delta < 150$, there are no limits for our containment system to contain the Santy worm, otherwise, the limits do exist. In addition, the curve for the PR-Model in Fig.5 (b) looks very strange that the containment limit varies little when the δ increases from 200 to 500. According to our analysis, this is because in the PR-Model we divide suspicious servers into groups and newly added scans between that interval are all falling into groups whose infection rate has reached 100% since $\delta = 200$. As a result, they contribute nothing to increasing the final infection rate of the search worm. Actually, this is another refection of the effects of page ranking: popular (larger) servers attract more infection attempts.



(a) ε varies ($\delta = 100$)



(b) δ varies ($\varepsilon = \frac{4\times10^4}{6\times10^6}$)

Fig. 5. Containment limits of our containment system (All cases are for $N = 6 \times 10^6$, $\delta = 100$, starting on a single machine,the power law $\sigma = 1.6$.)

5 Conclusion and Future Work

In this paper, we have studied the modeling and containment of search worms. We first present a virtual search worm abstracted from the Santy worm.

And then, we study two factors that greatly affect the propagation of this worm: eigenpage distribution and page ranking. To study the influence of eigenpage distribution, we propose two propagation models: U-Model assuming eigenpages are uniformly distributed on suspicious servers and PL-Model assuming the distribution follows a power law. We find and prove that, the uniform distribution maximizes the spreading speed of the search worm. To study the influence of page ranking, we propose the PR-Model. In this model, we assume the ranking of an eigenpage is determined by its PageRank value, which also follows a power law and the relative importance of its resident host. Simulation shows the PR-Model spreads much slower than the U-Model and PL-Model. Eventually, We propose a containment system for the search worm by randomly inserting honey pages among search results. Then infectious can be detected and blocked by monitoring and analyzing access behaviors in the honeypots pointed to by honey pages. We analyze this containment system under the help of our propagation models and solve two core questions: (1) How to calculate the honey-page insert rate for a given containment requirement? (2) Does this value always exist for an arbitrary requirement? From our calculation and simulation, by inserting no more than 2 honey pages in every 100 search results, the Santy worm can be almost completely stopped at its early age. Thereby, our containment system is really effective.

Although we make some contributions, there are still a variety of challenge issues that require further investigations. Firstly, search worms bring significant loads to search engines, as a result, their propagations will be constrained by the throughput of search engines. But we haven't considered this factor in our propagation models by now. Secondly, search worm may validate the truth of search results by checking the URL formats or analyzing their page contents. Then we have to develop effective mechanisms to disguise honey pages as true ones both in URL and contents. Since we just propose a concept containment system in this paper, these technique details are not covered. We plan to study these issues in the next step.

Acknowledgement

The first author of this research is supported by the governmental scholarship from China Scholarship Council. This research is also partially supported by JAPAN SCIENCE AND TECHNOLOGY AGENCY (JST), Strategic Japanese-Indian Cooperative Programme on Multidisciplinary Research Field, which combines Information and Communications Technology with Other Fields, entitled "Analysis of Cryptographic Algorithms and Evaluation on Enhancing Network Security Based on Mathematical Science."

References

[1] Zou, C.C., Gong, W., Towsley, D.: Code Red Worm Propagation Modeling and Analysis. In: 9th ACM Symposium on Computer and Communication Security (CCS 2002), pp. 138–147. ACM Press, Washington (2002)

[2] Hyppone, M., et al.: F-Secure Virus Descriptions: Santy (2004), http://www.f-secure.com/v-descs/santy_a.shtml

[3] Sophos.: Sophos Virus Analysis: W32/MyDoom-O (2004), http://www.sophos.com/security/analyses/w32mydoomo.html

[4] Kotadia, M.: Google squashes Santy worm (2004), http://news.cnet.com/Google-squashes-Santy-worm/2100-7349_3-5500265.html

[5] Daley, D.J., Gani, J.: Epidemic Modeling: An Introduction. Cambridge University Press, Cambridge (1999)

[6] Staniford, S., Paxson, V., Weaver, N.: How to Own the Internet in Your Spare Time. In: The 11th USENIX Security Symposium, pp. 149–167. USENIX Association, California (2002)

[7] Chen, Z., Gao, L., Kwiat, K.: Modeling the Spread of Active Worms. In: 2003 IEEE INFOCOMM, pp. 1890–1900. IEEE Press, San Francisco (2003)

[8] Zou, C.C., Gong, W., Towsley, D.: Worm propagation modeling and analysis under dynamic quarantine defense. In: 2003 ACM workshop on Rapid malcode, pp. 51–60. Acm Press, Washington (2003)

[9] Zou, C.C., Gong, W., Towsley, D., Lixin, G.: The monitoring and early detection of internet worms. IEEE Transaction on Networking (TON) 13(5), 961–974 (2005)

[10] Sellke, S.H., Shroff, N.B., Bagchi, S.: Modeling and Automated Containment of Worms. IEEE Transactions on Dependable and Secure Computing (TDSC) 5(2), 71–86 (2008)

[11] Provos, N., McClain, J., Wang, K.: Search Worms. In: WORM 2006, pp. 1–8. ACM Press, Virginia (2006)

[12] Johhny.: Google Hacking Database (2009), http://www.hackersforcharity.org/ghdb/

[13] Riden, J., McGeehan, R., Engert, B., Mueter, M.: Know your Enemy: Web Application Threats (2008), http://www.honeynet.org/papers/webapp/

[14] Huberman, B.A., Adamic, L.A.: Growth dynamics of the world wide web. Nature 401(6749), 131 (1999)

[15] Pandurangan, G., Raghavan, P., Upfal, E.: Using PageRank to characterize Web structure. Internet Math. 3(1), 1–20 (2006)

[16] Litvak, N., Scheinhardt, W.R.W., Volkovich, Y.: In-degree and PageRank: Why do they follow similar power laws? Internet Math. 4(2-3), 175–198 (2007)

HProxy: Client-Side Detection of SSL Stripping Attacks

Nick Nikiforakis, Yves Younan, and Wouter Joosen

IBBT-DistriNet
Katholieke Universiteit Leuven
Celestijnenlaan 200A B3001
Leuven, Belgium
{nick.nikiforakis,yves.younan,wouter.joosen}@cs.kuleuven.be

Abstract. In today's world wide web hundreds of thousands of companies use SSL to protect their customers' transactions from potential eavesdroppers. Recently, a new attack against the common usage of SSL surfaced, SSL stripping. The attack is based on the fact that users almost never request secure pages explicitly but rather rely on the servers, to redirect them to the appropriate secure version of a particular website. An attacker, after becoming man-in-the-middle can suppress such messages and provide the user with "stripped" versions of the requested website forcing him to communicate over an insecure channel. In this paper, we analyze the ways that SSL stripping can be used by attackers and present a countermeasure against such attacks. We leverage the browser's history to create a security profile for each visited website. Each profile contains information about the exact use of SSL at each website and all future connections to that site are validated against it. We show that SSL stripping attacks can be prevented with acceptable overhead and without support from web servers or trusted third parties.

Keywords: MITM Detection, SSL Stripping, Browser Security.

1 Introduction

In 1994 Netscape Communications released the first complete Secure Sockets Library (SSL) which allowed applications to exchange messages securely over the Internet [20]. This library uses cryptographic algorithms to encrypt and decrypt messages in order to prevent the logging and tampering of these messages by potential eavesdroppers. Today SSL is considered a requirement for companies who handle sensitive user data, such as bank account credentials and credit card numbers. According to a study by Netcraft[16], in January of 2009 the number of valid SSL certificates on the Internet reached one million, recording an average growth of 18,000 certificates per month. Due to its widespread usage, attackers have developed several attacks, mainly focusing in the forging of invalid SSL certificates and hoping that users will accept them.

Recently however a new attack has surfaced [13]. This technique is not based on any specific programming error but rather on the whole architecture and usage of secure webpages. It is based on the observation that most users never explicitly request SSL protected websites, in the sense that they never type the `https` prefix in their browsers. The transition from cleartext pages to encrypted ones is done usually either through web server redirects, secure links, or secure target links of HTML forms. If an attacker can launch a man-in-the-middle (MITM) attack, he can suppress all such transitions by "stripping" these transitional links from the cleartext HTTP protocol or HTML webpages before forwarding these messages/webpages to the unsuspecting client. Due to stripping of all SSL information, all data that would originally be encrypted are now sent as cleartext by the user's browser providing the attacker with sensitive data such as user credentials to email accounts, bank accounts and credit card numbers used in online transactions.

In this paper, we explore the idea of using the browser's history as a detection mechanism. We design a client-side proxy which creates a unique profile for each secure website visited by the user. This profile contains information about the specific use of SSL in that website. Using this profile and a set of detection rules, our system can identify when the page has been maliciously altered by a MITM and block the connection with the attacker while notifying the user of an attacker's presence on the network. Our approach does not require server-side cooperation and it does not rely on third-party services.

The main contributions of this paper are:

– Analysis and extension of a new class of web attacks
– Development of a generic detection ruleset for potential attack vectors
– Implementation of a client-side proxy which protects end-users from such attacks

The rest of this paper is structured as follows. In Section 2, we describe how SSL stripping attacks work followed by the reasons which make these attacks widely effective in Section 3. In Section 4 we present the architecture and workings of HProxy. We discuss some difficulties and how we overcame them in Section 5. In Section 6 we present the evaluation of our approach followed by some implementation details in Section 7. Section 8 discusses the related work and we conclude in Section 9.

2 Anatomy of SSL Stripping Attacks

Once an attacker becomes MITM on a network, he can modify HTTP messages and HTML elements in order to trick the user's browser into establishing unencrypted connections. In the following two scenarios we present two successful attacks based on redirect suppression and target form re-writing. The first attack exploits HTTP protocol messages and the second attack rewrites parts of a cleartext HTML webpage.

2.1 Redirect Suppression

1. The attacker launches a successful MITM attack against a wireless network becoming the network's gateway. From this point on, all requests and

responses from any host on the wireless network are inspected and potentially modified by him.

2. An unsuspecting user from this wireless network uses his browser and types in the URL bar, `mybank.com`. The browser crafts the appropriate HTTP message and forwards the message to the network's gateway.
3. The attacker inspects the message and realizes that the user is about to start a transaction with `mybank.com`. He forwards the message to MyBank's webserver.
4. `mybank.com` protects their entire website using SSL thus, the webserver responds with a 301 (Moved Message) to `https://www.mybank.com`.
5. The attacker intercepts the move message, and instead of forwarding it to the user, he establishes a secure connection with MyBank and after decrypting the resulting HTML, he forwards that to the user.
6. The user's browser receives cleartext HTML, as a response to his request and renders it. What the user now sees is an unencrypted version of MyBank's login page. The only thing that is missing is a subtle lock icon, which would be otherwise located somewhere on the browser window.
7. From this point on, all user-data are transmitted as cleartext to the attacker, where he tunnels them through his own encrypted connection. This results in completely functional but unencrypted web sessions.

2.2 Target form Re-writing

This attack is quite similar to the redirect suppression attack except for a significant detail. Target form re-writing is an attack against websites which operate mainly over HTTP and they only protect parts of their webpages, such as a login form and any subsequent pages for logged-in users. The way this is constructed in HTML is that while the main page is transfered over HTTP, the target URL of a specific form has an HTTPS prefix. When the user clicks the "submit" button, the browser recognizes the secure protocol and attempts to establish an SSL connection with the target web server. This is disastrous for an attacker because, even though he controls all local network connections, he has no realistic way of presenting a valid SSL certificate for the secure handshake of the requested web server. The attacker thus, will have to present a self-signed certificate resulting in multiple warnings which the user must accept before proceeding with the connection. In order to avoid this pitfall, the attacker strips all secure form links and replaces them with cleartext versions. So, a form with a target of `https://www.example.com/login.php` becomes `http://www.example.com/login.php` (note the missing `S` from the protocol). The browser has no way of knowing that the original link had a secure target and thus sends the user's credentials over an unencrypted channel. In the same way as before, the attacker uses these credentials in his own valid SSL connection and later forwards to the user the resulting HTML page.

3 Effectiveness of the Attack

In this section we would like to stress the severity of the SSL attacks described in Section 2. We argue that the two main reasons which make SSL stripping such an effective attack are: a) the wide applicability of it in modern networks and b) the way that feedback works on browser software.

3.1 Applicability

When eavesdropping attacks were first introduced, they targeted hubbed networks since hubs transmit all packets to all connected hosts, leaving each host to choose the packets that are addressed for itself and disregard the rest. The attacker simply configured his network card to read all packets (promiscuous mode) and had immediate access to all the information coming in and out of the hubbed network. Once hubs started being replaced by switches, this attack was no longer feasible since switches forwarded packets only to the hosts that were intended to receive them (using their MAC addresses as a filter). Attackers had to resort to helper techniques (such as ARP flooding, which filled-up the switch's memory forcing it to start transmitting everything to everyone to keep the network functioning) in order for their eavesdropping attacks to be effective [15].

Today however, due to the widespread use of wireless network connections, attackers have access to hundreds of thousands of wireless networks ranging from home and hotel networks to airport and business networks. Wireless networks are by definition hubbed networks since the transport medium is "air". Even secure wireless networks (WEP/WPA2) are susceptible to MITM attacks as long as the attacker can find the encryption key (trivial for WEP [25] not so trivial for WPA2).

The ramifications become even greater when we consider that wireless networks are not restricted to laptops anymore due to the market penetration of hand held devices which use them to connect to the Internet. More and more people use these kind of devices to perform sensitive operations from public wireless networks without suspecting that a potential attacker could be eavesdropping their transactions.

3.2 Software Feedback

The second main reason that makes this attack effective is that it doesn't produce negative feedback. Computer users have been unconsciously trained for years that the absence of warning messages and popups means that all operations were successful and nothing unexpected happened. This holds true also for security critical operations where users trust that a webpage is secure as long as the browser remains "silent".

In the scenario where an attacker tries to present to a web browser a self-signed, expired or otherwise illegal certificate, the browser presents a number of dialogues to the user which inform him of the problems and advise him not to proceed with his request. Modern browsers (such as Firefox) have the user click many times on a number of different dialogues before allowing him to proceed. Many users, understand that it is best to trust their browser's warnings, especially if they are working from an unfamiliar network (such as a hotel network), even if they end up not doing so [22].

In the SSL stripping attack however, the browser is never presented with any illegal SSL certificates since the attacker strips the whole SSL connection before it reaches the victim. With no warning dialogues, the user has little to no visual cues that something has gone wrong. In the case of SSL-only websites (websites that operate solely under the HTTPS protocol) the only visual cue that such an attack generates is the absence of lock icon somewhere on the browser's window (something that the attacker can compensate for by changing the .favico icon of the website to a padlock). In partly-protected websites, where the attacker strips the SSL protocol from links and login forms, there are no visual cues and the only way for a user to spot the attack is to manually inspect the source code and identify the parts that have been changed.

4 Automatic Detection of SSL Stripping

In this section we describe our approach that automatically detects the existence of a MITM attacker conducting an SSL stripping attack on a network. The main strength of MITM attacks is the fact that the attacker has complete control of all data coming in and going out of a network. Any client-side technique trying to detect an attacker's presence must never rely solely on data received by the current network connection.

4.1 Core Functionality

Our approach is based on browser history. The observation that lead to this work is that while a MITM attacker has at some point in time, complete control of all traffic on a network, he did not always have this control. We assume that users mainly use secure networks, such as WPA2-protected wireless networks or properly configured switched networks and use insecure networks only circumstantially. Regular browsing of SSL-enabled websites from these secure locations can provide us with enough data to create a profile of what is expected in a particular webpage and what is not.

Our client-side detection tool, History Proxy (HProxy), is trained with the requests and responses of websites that the user regularly visits and builds a profile for each one. It is important to point out that HProxy creates a profile based on the security characteristics of a website and not based on the website's content, enabling it to operate correctly on static as well as most dynamic websites.

HProxy uses the profile of a website, the current browser request and response along with a detection ruleset to identify when a page is maliciously modified by a MITM conducting an SSL stripping attack. The detection ruleset is straightforward and will be explained in detail in Section 4.3.

4.2 Architecture of HProxy

The architecture of HProxy comprises of the detection ruleset and a number of components which utilize and enforce it - Fig. 1. The main components are: a webpage analyzer, which analyzes and identifies the requests initiated from the browser along with the server responses, a MITM Identifier which checks requests and responses against the detection ruleset to decide whether a page is safe or not and lastly a taint module which tries to prevent the leakage of private information even if the MITM-identifier incorrectly tags a page as safe.

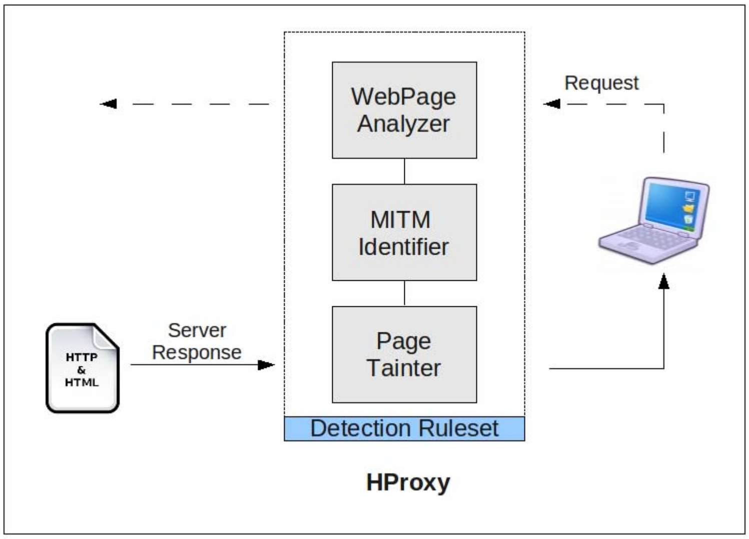


Fig. 1. Architecture of HProxy

Webpage Analyzer. The webpage analyzer is the component responsible of identifying all the critical parts of a webpage. The critical parts of a webpage are the parts that a MITM attacker can insert or alter in order to steal credentials from the end users and are the following:

- JavaScript blocks
- HTTP forms and their targets
- `Iframe` tags
- HTTP Moved messages

The Webpage Analyzer identifies all of the above data structures, along with their attributes and records them in the page's current profile. If a particular page is visited for the first time then this current profile is registered in the profile database, effectively becoming the page's original profile, and the page is forwarded to the user. If not, then the current profile will be checked against the page's original profile by the MITM Identifier. Why these structures are dangerous will be described in detail in Section 4.3.

MITM Identifier. The MITM Identifier component encapsulates almost all the detecting capabilities of HProxy (except of the taint component which will be discussed later). It uses the page's current profile as created by the Webpage Analyzer against the page's original profile. In order to make a decision whether a page is altered by an attacker or not, the MITM Identifier utilizes the detection ruleset of HProxy. This ruleset consists of rules for every sensitive data structure that was previously mentioned. Each rule contains the dangerous modifications that can appear in each page, using the page's original profile as a base. Any modifications detected by the Webpage Analyzer that are identifiable by this ruleset are considered a sign of an SSL stripping attack and thus the page is not forwarded to the user.

PageTainter. Even though we have strived to create a ruleset which will be able to detect all malicious modifications we deliberately decided to allow content changes when we cannot decisively classify them as an attack. In order to compensate for these potentially false negatives, HProxy contains a module called PageTainter. The purpose of PageTainter is to enable HProxy to stop in time the leakage of private user data, even when the MITM Identifier module wrongly tags a malicious page as "safe". For HProxy to stop the leakage of private data, it must first be able to identify what private data is. In order to do this, PageTainter modifies each webpage that contains a secure login form (identifiable by the password-type HTML element) and adds a JavaScript routine which sends the password from it to HProxy once the user types it in. This password is recorded in HProxy in a domain,password tuple[1]. In addition to that, it taints all forms with an extra hidden field which contains location information so that we can later identify which page initiated a GET or a POST request. For each request that initiates from the browser, the PageTainter module, using the hidden domain field checks for the presence of the stored password in the outgoing data. If the page is legitimate, the domain's password will never appear in the HTTP data because it is exchanged only over SSL. A detection of it signifies the fact that an attacker's successful modification passed through our MITM Identifier and is now sending out the password. In this case, HProxy does not allow the connection to be established and informs the user of the attack. To make sure that an attacker will not obfuscate the password beyond recognition by the PageTainter, our detection ruleset has very strict JavaScript rules which will be explained in the next section.

4.3 Detection Ruleset

Using the description of SSL-stripping attacks as a base, we studied and recorded all possible HTML and HTTP elements that could be misused by a MITM attacker. This study resulted in a set of pragmatic rules which essentially describe dangerous transitions from the original webpage (as recorded by HProxy) to all future instances of it. A transition can be either an addition of one or more

[1] HProxy runs on the same physical host as the browser(s) that it protects thus there are no privacy issues with the stored passwords.

HTML/HTTP elements by the attacker to the original webpage or the modification of existing ones.

The detection ruleset consists of dangerous modifications for every class of sensitive data structures. Each page that comes from the network is checked against each class of rules before it is handed back to the user. In the rest of this section we present the rules for each class of sensitive structures.

HTTP Moved Messages. The HTTP protocol has a variety of protocol messages of which the "moved" messages can be misused in an SSL stripping attack since their suppression can lead to unencrypted sessions (as shown in the example attack in Section 2.1). The main rule for this class of messages states that, if the original page profile contains a move message from an `HTTP` to an `HTTPS` page, then any other behavior is potentially dangerous. Given an original request of `HTTP GET` for `domain_a` and an original response stored in the profile database of `MOVED` to `HTTPS domain_a/page_a`, we list all the possible modifications and whether they are allowed by our ruleset, in the following table.

Current Response	**Modification**	**Allowed?**
MOVED HTTPS domain_a/page_a	None	Yes
MOVED HTTPS domain_a/page_b	Changed page	Yes
MOVED HTTP domain_a/page_a	Non-SSL protocol	No
MOVED HTTP domain_b/page_a	Changed domain	No
MOVED HTTPS domain_b/page_a	Changed domain	No
OK `<html>....</html>`	HTML instead of MOVED	No

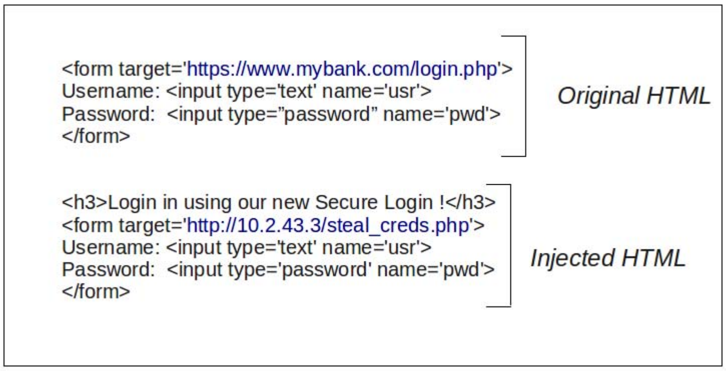
This ruleset derives from the observation that the developers of a website may decide to create new webpages or rename existing ones, but they will not suddenly stop providing HTTPS nor export their secure service to another domain. For websites that operate entirely using SSL, this is the only class of rules that will be applied to them as they will operate securely over HTTPS once the `MOVE` message has been correctly processed.

The rest of the ruleset is there to protect websites that are partly protected by SSL. Such websites use SSL only for their login forms and possibly for the subsequent pages that result after a successful login. The transition from unprotected to protected pages (within the same website) is done usually through a HTTPS form target or through a HTTPS link.

JavaScript. JavaScript is a powerful, flexible and descriptive language that is legitimately used in almost all modern websites to make the user experience better and to offload servers of common tasks that can be executed on the client-side. All these features of JavaScript, including the fact that it is enabled by default in all major browsers make it an ideal target for attackers. Attackers can and have been using JavaScript for a multitude of attacks ranging from Cross-site Scripting [11] to Heap Spraying attacks [19]. For the purpose of stealing credentials, JavaScript can be used to read parts of the webpage (such as a typed-in username and password) and send it out to the attacker.

JavaScript can be categorized as either inline or external. Inline JavaScript, is written inline an HTML webpage, e.g. `<html><script>...</script> </html>`. External JavaScript, is written in separate files, present on a webserver that are being included in an HTML page using a special tag, e.g. `<html><script src="http://domain1/js_file.js"> </html>`. Unfortunately for users, both categories of JavaScript can be misused by a MITM. If an attacker adds inline JavaScript in a webpage before forwarding it to the user, the browser has no easy way of discerning which JavaScript parts were legitimately present in the original page and which were later added by the attacker. Also, the attacker can reply to a legitimate external JavaScript request with malicious code since he already has full control over the network and can thus masquerade himself as the webserver.

Because of the nature of JavaScript, HProxy has no realistic way of discerning between original and "added" JavaScript except through the use of whitelisting. The first time that a page which contains an HTTPS form is visited all JavaScript code (internal and external) is identified and recorded in the page's profile. If in a future request of that specific webpage, new or modified JavaScript is identified then the page is tagged as unsafe and it is not forwarded to the user. HProxy's initial whitelisting mechanism involved string comparisons of JavaScript blocks between page loads of the same website. Unfortunately though, the practice of simple whitelisting can lead to false positives. A way around these false positives is through the use of a JavaScript preprocessor. This preprocessor can distinguish between the JavaScript parts that have been legitimately changed by the web server and the parts which have been added or modified by an attacker. We expand HProxy to include such a preprocessor and we explore this notion in detail later on, in Section 5.



```
<form target='https://www.mybank.com/login.php'>
Username: <input type='text' name='usr'>
Password:  <input type="password" name='pwd'>
</form>
```

Original HTML

```
<h3>Login in using our new Secure Login !</h3>
<form target='http://10.2.43.3/steal_creds.php'>
Username: <input type='text' name='usr'>
Password:  <input type='password' name='pwd'>
</form>
```

Injected HTML

Fig. 2. Example of an injected HTML form by a MITM attacker

Iframe Tags. can be as dangerous as JavaScript. An attacker can add extra `iframe` tags in order to overlay fake login forms over the real ones [7] or reply with malicious content to legitimate `iframe` requests. Our detection ruleset for `iframe` tags states that no such tags are allowed in pages where an SSL login form is present. The only time an `iframe` tag is allowed is when the original profile of a website states that the login form itself is coded inside the `iframe`.

HTTP Forms. can be altered by a MITM attacker so as to prevent the user's browser from establishing an encrypted session with a web server, as was demonstrated in Section 2.2. Additionally, extra forms can also be used by an attacker as a way of stealing private information. The set of rules for this class of sensitive data structures is similar to the HTTP Move class ruleset. The previously mentioned Webpage analyzer, records every form, target and protocol for each page that an SSL login form is identified. The ruleset contains the dangerous form modifications that could leak private user credentials. The main rules are applied on the following characteristics:

– **Absence of forms** - The profile for each website maintains information about the number of forms in each page, whether they are login forms and which forms have secure target URLs. Once a missing form is detected, HProxy reads the profile to see the type of the missing form. If the missing form was a secure login form then HProxy tags this as an attack and drops the request. If the missing form was a plain HTTP form (such as a `Search` form) then HProxy allows the page to proceed.
– **New forms** - New forms can be introduced in a webpage either by web designers (who wish to add functionality to a specific page) or by an attacker who tries to lure the user into typing his credentials in the wrong form - Fig 2. If the new form is not a login form then it is an allowed deviation from the page's profile. If the new form is a login-form it is only allowed if the target of the form is secure and in the same domain as the original SSL login form of the page. Even so, there is a chance that a MITM can convince a user to submit his credentials through a non-login form. In these cases, PageTainter will identify the user's password in outgoing data and drop the request before it reaches the attacker.
– **Modified forms** - In this case, an attacker can modify a secure form into an insecure form. Based on the same observation from HTTP moved messages, HProxy does not allow a modified form to be forwarded to the browser if it detects: `(a)` a security downgrade in a login form (the original had an HTTPS target whereas the current one has an HTTP target); or `(b)` a domain change in the target URL.

4.4 Redirect Suppression Revisited

In Section 2.1 we presented one of the most common SSL stripping attacks against browsers, namely redirect suppression. The MITM suppressed the HTTP `Moved` messages and provided the user with an unencrypted version of an originally encrypted website. In this section we repeat the attack but this time, the user is running the HProxy tool. Steps 1-5 are the same with the earlier example but are repeated here for the sake of completeness.

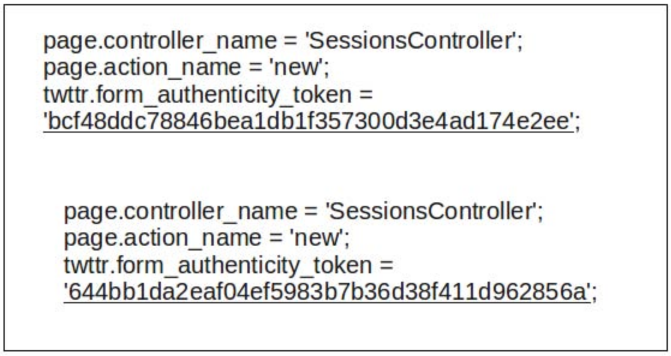
1. The attacker launches a successful MITM attack against a wireless network becoming the network's gateway. From this point on, all requests and responses from any host on the wireless network are inspected and potentially modified by him.
2. An unsuspecting user from this wireless network uses his browser and types in the URL bar, `mybank.com`. The browser crafts the appropriate HTTP message and forwards the message to the network's gateway.
3. The attacker inspects the message and realizes that the user is about to start a transaction with `mybank.com`. He forwards the message to MyBank's webserver.
4. `mybank.com` protects their entire website using SSL thus, the webserver responds with a 301 (Moved Message) to `https://www.mybank.com`.
5. The attacker intercepts the move message, and instead of forwarding it to the user, he establishes a secure connection with MyBank and after decrypting the resulting HTML, he forwards that to the user.
6. HProxy receives the response from the "server" and inspects it. HProxy's trained profile for MyBank states that `mybank.com` is an SSL protected website and when the user requests the website using HTTP, the server redirects him to the HTTPS version of it. This time however HProxy identifies the response as cleartext HTML which is not acceptable according to its detection ruleset.
7. HProxy drops the request and notifies the user about the presence of a MITM on the local network along with specific details.

5 Discussion

By analyzing the JavaScript code generated by the top visited websites (as reported by Alexa [24]) we discovered that the dynamic nature of today's Internet doesn't stop in dynamically generated HTML. Many top websites provide different JavaScript code blocks each time they are visited, even when the visits are seconds apart. This means that a simple whitelisting of JavaScript based on string comparison would result in enough false positives to render HProxy unusable. In this section we discuss two techniques that can greatly reduce these false positives: JavaScript preprocessing and Signed JavaScript. The final version of HProxy includes a JavaScript Preprocessor while Signed JavaScript can be used in the future to completely eliminate false positives. We also describe a different way of identifying a MITM by inspecting client requests and the potential problems of that approach.

5.1 JavaScript Preprocessing

Most of the JavaScript blocks, even the ones that constantly change, follow a specific structure that can be tracked along page loads. By comparing internal and external JavaScript along two consecutive page loads of a specific webpage, we can discover the static and the dynamic parts of that code. E.g., The JavaScript



```
page.controller_name = 'SessionsController';
page.action_name = 'new';
twttr.form_authenticity_token =
'bcf48ddc78846bea1db1f357300d3e4ad174e2ee';


  page.controller_name = 'SessionsController';
  page.action_name = 'new';
  twttr.form_authenticity_token =
  '644bb1da2eaf04ef5983b7b36d38f411d962856a';
```

Fig. 3. Portion of the JavaScript code present in two consecutive page loads of the login page of Twitter. The underlined part is the part that changes with each page load.

code in two consecutive loads of Twitter's login page differs only in the contents of a specific variable - Fig. 3.

We leverage this re-occurring structure to design a JavaScript preprocessor that greatly reduces false positives. When a website is visited for the first time through HProxy, the Webpage Analyzer (Section 4.2) makes a duplicate request and compares the JavaScript blocks from the original response and the duplicate one. If the blocks are different it then creates a template of the parts that didn't change and records the place and length of the dynamic parts. This information is stored in the Web pages profile and all future visits of that website will be validated against this template. This enables us, to discern between normal dynamic behavior of a website and JavaScript that was maliciously added by a MITM in order to steal the user's credentials. Although a JavaScript preprocessing that would work on an interpretation level would possibly be able to produce zero false positives we believe that the overhead of such an approach would be prohibitively high and thus we did not research that direction.

5.2 Signed JavaScript

Signed JavaScript (SJS) is JavaScript that has been signed by the web server using a valid certificate such as the one used in HTTPS communications. SJS can provide among other features (such as access to restricted JavaScript functions) the guarantee that the script the browser parses has not been modified since it was sent by the Web server [17]. This integrity assurance can be used by HProxy to whitelist unconditionally all JavaScript code blocks that are signed. The downside of this technique is that it requires both server and client-side support[2].

[2] At the time of this writing, only Mozilla Firefox appears to support SJS.

5.3 Inspecting Client Requests vs. Server Responses

It is evident that trying to secure JavaScript at the client-side can be a tedious and error-prone process. A different approach of detecting a MITM which may at first appear more appealing is to analyze the client-side requests for anomalous behavior rather than the server-side responses to client-side requests. In such a case, the resulting system would inspect the requests (both secure and insecure) of the browser and compare them to the requests done in the past. A security downgrade of a request, (e.g. the browser is currently trying to communicate to website `X` using an unencrypted channel whereas it always used to communicate over a secure channel), would be a sign of a MITM operating on the network and the request would be dropped. In such a system, JavaScript whitelisting would not be an issue since HProxy would only inspect the outgoing requests, regardless of their origin (HTML or JavaScript).

While this approach looks promising it produces more problems than it solves since it has no good way of discerning the nature of new outgoing requests. Consider the scenario where an attacker adds a JavaScript routine which copies the password from the correct form, encrypts it and sends it out using an AJAX request to a new domain. The system would not be able to find a previous outgoing request to match the current request by, and would have to either drop the request (also dropping legitimate new requests - false positives) or let it pass (false negatives). Also, in partly SSL-protected pages, where the client communicates with the same website using both encrypted and unencrypted channels, the MITM could force the browser to send private information over the wrong channel which would again result in leaking credentials.

For these reasons, we decided that a combination of inspecting server responses, preprocessing JavaScript and tracking private data (through the PageTainter - 4.2) would be more effective than inspecting client requests and thus we did not implement such a system.

6 Evaluation

In this section we provide a security evaluation, the number of false positives and the performance overhead of our approach.

6.1 Security Evaluation

HProxy can protect the end-user against the attacks described in [13] as well as a number of new techniques that could be used to steal user credentials in the context of SSL stripping attacks. It can protect the user from credential stealing through redirect suppression, insecure forms, JavaScript methods and injected `iframe` tags.

In order to test the actual effectiveness of our prototype we created a network setup with two clients and a wireless Access Point(AP) with Internet connection. One client was the legitimate user and the other one the MITM, both running the latest version of Ubuntu Linux. From the MITM machine we enabled IP forwarding and we used the `arpspoof` (part of the `dsniff` suite [6]) to position ourselves between the victim machine and the AP. We then run `sslstrip` [21], a tool which strips the SSL links from incoming traffic, creates SSL tunnels with the legitimate websites and captures sensitive data typed by the user. We started browsing the web from the victim machine and we observed that pages which normally are protected through SSL (like GMail and Paypal) were now appearing over HTTP, without any browser warnings whatsoever. Any data typed in fields of those pages were successfully eavesdropped by the MITM host.

We reset the experiment, enabled HProxy and started browsing the web. We browsed through a number of common websites so that HProxy could create a profile for each one of them. We then repeated the procedure of becoming MITM and run `sslstrip`. Through the victim client, we started visiting all the previously "stripped" websites. This time however, HProxy detected all malicious changes done by `sslstrip` and warned the user of the presence of a MITM attacker on the network.

6.2 False Positives

A false positive, is an alert that an Intrusion Detection System (IDS) issues when it detects an attack, that in reality did not happen. When HProxy parses a page, it can occasionally reach to the conclusion that the page was modified by an attacker even if the page was legitimately modified by the web server. These false conclusions can confuse the user as well as undermine his trust of the tool. Most of HProxy's false positives can be generated by its JavaScript rules, as explained in section 4.3.

In order to make these occasions as rare as possible we decided to monitor JavaScript blocks only in pages that contain (or originally contained) secure login forms. This decision does not undermine the overall security of HProxy since in the context of SSL Stripping attacks, JavaScript can only be used to steal credentials as they are typed-in by the user in a secure form. In addition to that, we developed a JavaScript Preprocessor, as explained in Section 5.1 which generates a template of each website and a list of expected JavaScript changes.

To measure the amount of false-positives, we compiled a list of 100 websites that contain login pages and we programmed Firefox using `ChickenFoot` [4] to automatically visit them three consecutive times. Firefox's incoming and outgoing traffic was inspected by HProxy which in turn decided whether the page was secure or not. The first time the page was visited, HProxy created a profile for it, which it used for the next two times. Due to our lab secure network settings, any attack reported by HProxy was a false positive.

In Fig. 4 we present the ratio of HProxy's false-positives using three methods of whitelisting JavaScript. The first method that we used is simply gathering all the JavaScript blocks of a webpage and computing their MD5 checksum. If the JavaScript blocks between two page loads differ, then their checksums will also be different. In the second method, we use the JavaScript preprocessor with a strict template, where the changes detected by the preprocessor must be in the precise place and of precise length as the ones originally recorded. Finally we use the

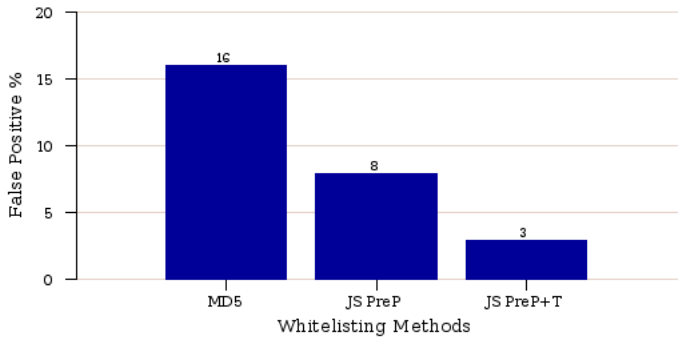


Fig. 4. False-positive ratio of HProxy using three different methods of whitelisting JavaScript

same preprocessor but this time we include a "tolerance factor" of 10 characters, where the position and length of changes may vary up to 10 characters (less that 1% of the total length of JavaScript code for most websites).

Using the last method as the whitelisting method of choice, HProxy can handle almost all JavaScript changes successfully. The false-positives are created by webpages which produce JavaScript blocks of different length each time that they are visited. The websites that contain such pages are always the same and can thus be added to a list of unprotected pages.

6.3 Performance

To measure the performance overhead of our HProxy prototype, we used a list of the top 500 global websites [24] and we programmed Firefox to visit them ten times each while measuring how much time each page needed to fully load. In

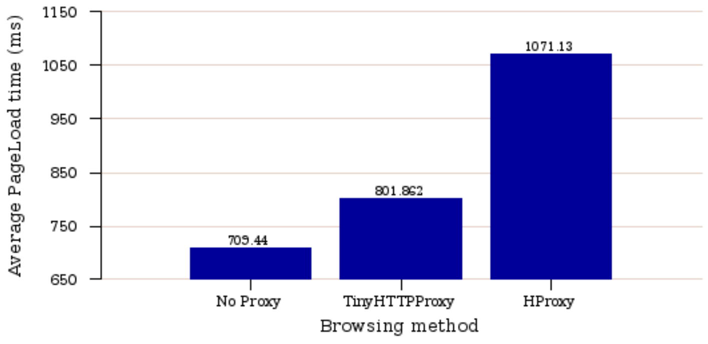


Fig. 5. Average load time of the top 500 websites of the Internet when accessed locally without a proxy, with a simple forwarding proxy(TinyHTTPProxy) and with HProxy

order to avoid network inconsistencies we downloaded a copy of each website and browse them locally using a web server that we setup on the same machine that Firefox was running. All caching mechanisms of Firefox were disabled and we were clearing the Linux memory cache between experiments. We repeated the experiment three times and in Fig. 5 we present the average load time of Firefox when it run: `(a)` without a proxy `(b)` using a proxy that just forwarded requests to and from Firefox and `(c)` using HProxy. Hproxy shows an overhead of 33% when compared with the forwarding proxy and 51% when compared with Firefox directly accessing the web pages. While this overhead appears substantial, it is important to remember that even the 51% overhead is actually an overhead of 0.41 seconds of time. Firefox starts rendering received content, long before each page fully loads. This means that the user can start "consuming" the content of each page without having to wait for all objects to be downloaded. Given this behavior, we believe that the added delay of HProxy is only minutely, if at all, perceived by the user during normal web browsing.

7 Implementation

We implemented a prototype version of HProxy using Python. We used an already implemented Python proxy, TinyHTTPProxy [10] and we built on top of it to add the various detection mechanisms that were described in earlier sections. We chose to implement HProxy as a stand-alone application and not as a browser plugin because we wanted to test parts of its functionality (such as the AJAX functions emmited by the PageTainter module) with multiple browsers. HProxy runs on the same physical machine as the browser(s) that it protects. A proxy running on a different machine could potentially be used by multiple users to improve caching but that would allow a MITM to impersonate HProxy and steal user credentials. The Webpage Analyzer and the PageTainter modules use the BeautifulSoup HTML parser [2] to recognize forms, JavaScript and `iframe` tags. For the HTTP `Moved` messages we wrote our own parser using regular expressions.

The reason why we chose Python instead of another programming language is because Python's features make it ideal for fast prototyping. We believe however, that if HProxy gets re-implemented using a compiled language or if it becomes part of a browser (as an extension or as part of the browser's code) the overhead of its use will be much lower than the one we measured in Section 6.3.

8 Related Work

To the best of our knowledge, this paper is the first academic countermeasure which is specifically geared towards SSL stripping attacks. Previous studies mainly focus on the detection of a MITM attacker especially on wireless networks. While a number of these studies detect a wider range of attacks than our approach, it is important to point out that most of them require either specific hardware or knowledge of the network that surpasses the average user's session.

This effectively means that unless the techniques are employed before-hand by the administrators of the network they can be of little to no use to the connecting clients. On the other hand HProxy is a client-side tool which protects users from SSL stripping attacks without requiring any support from the wireless network infrastructure.

A number of studies use the information already existing in the 802.11 protocol to identify attackers that try to impersonate legitimate wireless nodes by changing their MAC address. The authors of [9,26] use the sequence number field of MAC frames as a heuristic for detecting nodes who try to mimic existing MAC addresses. The sequence number is incremented by the node every time that a frame is sent. They create an intrusion detection system which identifies attackers by monitoring invalid, duplicate or dis-proportionally large sequence numbers. Martinez et al. [14] suggest the use of a dedicated passive Wireless Intrusion Detection System (WIDS) which identifies attackers by logging and measuring the time interval between beacon frames. Beacon frames that were broadcasted before the expiration of the last beacon frame (as announced by the AP) are a sign of an impersonation attack. In the same manner, Laroche et. al [12] present a WIDS which uses information such as sequence numbers and fragment numbers, to identify layer-2 attacks. Genetic algorithms are executed against these datasets in an effort to identify impersonating nodes. Unfortunately, their IDS requires training on labeled data sets making it impractical for fast fluctuating wireless networks such as the ones deployed in hotels and airports where wireless nodes are constantly added and removed.

Other researchers have focused more on the physical characteristics of wireless networks and how they relate to intrusion detection. Chen et. al [3] as well as Sheng et al. [18] use the Received Signal Strength (RSS) of a wireless access point as a way to differentiate between the legitimate access point(s) and an attacker masquerading as one. In both studies, multiple passive gathering devices are used to record the RSS and the data gathered is analyzed using cluster algorithms and Gaussian models. Similarly Suski et al. [23] use special wireless hardware monitors to create and monitor an "RF Fingerprint" based on the inherent emission features of each wireless node. While the detection rates of such studies are quite high, unfortunately their approaches are inherently tied to a significant increase in setup costs (in time, hardware or both) making them unattractive for everyday deployment environments.

Moving up to the top layer of the OSI model, several studies have shown that security systems lack usability and that users accept dialogues and warnings without really understanding the security implications of their actions [1,5,8,22]. Xia et al. [27] try to combat MITM attacks by developing a system which tries to give as much information to the user as possible when invalid certificates are encountered or when a password is about to be transmitted over an unencrypted connection. Due to the nature of SSL stripping attacks, the attacker does not have to present an invalid certificate in order to successfully eavesdrop the user, thus the part of their approach that deals with invalid certificates is ineffective against it. The part that deals with the un-encrypted transmission of a password can be of some use but can be easily circumvented using JavaScript or `iframe` tags as shown in Section 4.3.

9 Conclusion

Hundreds of thousands of websites rely on SSL daily to protect their customers' traffic from eavesdroppers. Recently though, a new kind of attack against the usage of the SSL protocol surfaced: SSL stripping. The power of such an attack is mainly due the fact that it produces no negative feedback, something that users have been unconsciously trained to search for as an indicator of a page's "insecurity".

In this paper we demonstrated that SSL stripping attacks are a realistic threat and presented a countermeasure that protects against them. This countermeasure, called HProxy, leverages the browser's history to create security profiles for each website. These profiles contain information about the use of SSL and every future load of that website is validated against that profile. Our prototype implementation of HProxy accurately detected all SSL stripping attacks with very few false positives. Our evaluation of HProxy showed that it can be used with acceptable overhead and without requiring server side support or trusted third parties to secure users against this type of attack.

Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U. Leuven.

References

1. Almuhimedi, H., Bhan, A., Mohindra, D., Sunshine, J.: Toward Web Browsers that Make or Break Trust. In: Symposium Of Usable Privacy and Security (SOUPS) (2008)
2. BeautifulSoup Parser, `http://www.crummy.com/software/BeautifulSoup/`
3. Chen, Y., Trappe, W., Martin, R.P.: Detecting and Localizing Wireless Spoofing Attacks. In: Proceedings of the Fourth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (IEEE SECON 2007), San Diego, CA, USA (2007)
4. Chickenfoot for Firefox: Rewrite the Web, `http://groups.csail.mit.edu/uid/chickenfoot/faq.html`
5. Dhamija, R., Tygar, J.D., Hearst, M.: Why phishing works. In: CHI 2006: Proceedings of the SIGCHI conference on Human Factors in computing systems, pp. 581–590. ACM, New York (2006)
6. dsniff, `http://monkey.org/~dugsong/dsniff/`
7. Egele, M., Balduzzi, M., Kirda, E., Balzarotti, D., Kruegel, C.: A Solution for the Automated Detection of Clickjacking Attacks. In: Proceedings of ASIACCS, Beijing, China (April 2010)

8. Friedman, B., Hurley, D., Howe, D.C., Felten, E., Nissenbaum, H.: Users' conceptions of web security: a comparative study. In: CHI 2002 extended abstracts on Human factors in computing systems, pp. 746–747. ACM, New York (2002)
9. Guo, F., Chiueh, T.-c.: Sequence number-based MAC address spoof detection. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 309–329. Springer, Heidelberg (2006)
10. Hisao, S.: Tiny HTTP Proxy in Python, http://www.okisoft.co.jp/esc/python/proxy/
11. Klein, A.: Cross Site Scripting Explained, Sanctum White Paper (2002)
12. LaRoche, P., Nur Zincir-Heywood, A.: Genetic Programming Based WiFi Data Link Layer Attack Detection. In: CNSR 2006: Proceedings of the 4th Annual Communication Networks and Services Research Conference, Washington, DC, USA, pp. 285–292. IEEE Computer Society, Los Alamitos (2006)
13. Marlinspike, M.: New Tricks for Defeating SSL in Practice. In: Proceedings of BlackHat 2009, DC (2009)
14. Martínez, A., Zurutuza, U., Uribeetxeberria, R., Fernández, M., Lizarraga, J., Serna, A., naki Vélez, I.: Beacon Frame Spoofing Attack Detection in IEEE 802.11 Networks. In: ARES 2008: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security, Washington, DC, USA, pp. 520–525. IEEE Computer Society, Los Alamitos (2008)
15. Nachreiner, C.: Anatomy of an ARP Poisoning Attack, http://www.watchguard.com/infocenter/editorial/135324.asp
16. NetCraft. One Million SSL Sites on the Web, http://news.netcraft.com/archives/2009/02/01/one_million_ssl_sites_on_the_web.html
17. Ruderman, J.: JavaScript Security: Signed Scripts, http://www.mozilla.org/projects/security/components/signed-scripts.html
18. Sheng, Y., Tan, K., Chen, G., Kotz, D., Campbell, A.: Detecting 802.11 MAC Layer Spoofing Using Received Signal Strength. In: Proceedings of INFOCOM 2008, pp. 1768–1776 (2008)
19. Sotirov, A.: Heap Feng Shui in Javascript. In: Proceedings of BlackHat Europe 2007 (2007)
20. The SSL Protocol, http://www.webstart.com/jed/papers/HRM/references/ssl.html
21. Moxie Marlinspike's sslstrip, http://www.thoughtcrime.org/software/sslstrip/
22. Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., Cranor, L.F.: Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In: Proceedings of Usenix Security (2009)
23. Suski, W.C., Temple, M.A., Mendenhall, M.J., Mills, R.F.: Using Spectral Fingerprints to Improve Wireless Network Security. In: IEEE Global Telecommunications Conference, IEEE GLOBECOM 2008, 30-December 4, pp. 1–5 (2008)
24. Alexa Top 500 Global Sites, http://www.alexa.com/topsites
25. Walker, J.R., Submission Page Jesse Walker, Intel Corporation: Unsafe at any key size; An analysis of the WEP encapsulation (2000)
26. Wright, J.: Detecting Wireless LAN MAC Address Spoofing (2003)
27. Xia, H., Brustoloni, J.C.: Hardening Web browsers against man-in-the-middle and eavesdropping attacks. In: WWW 2005: Proceedings of the 14th international conference on World Wide Web, pp. 489–498. ACM, New York (2005)

Author Index