J Comput Virol (2011) 7:121–131
DOI 10.1007/s11416-010-0142-4

ORIGINAL PAPER

Improving antivirus accuracy with hypervisor assisted analysis

Daniel Quist · Lorie Liebrock · Joshua Neil

Received: 14 December 2009 / Accepted: 11 March 2010 / Published online: 6 April 2010

Abstract Modern malware protection systems bring an especially difficult problem to antivirus scanners. Simple obfuscation methods can diminish the effectiveness of a scanner significantly, often times rendering them completely ineffective. This paper outlines the usage of a hypervisor based deobfuscation engine that greatly improves the effectiveness of existing scanning engines. We have modified the Ether malware analysis framework to add the following features to deobfuscation: section and header rebuilding and automated kernel virtual address descriptor import rebuilding. Using these repair mechanisms we have shown as high as 45% improvement in the effectiveness of antivirus scanning engines.

1 Introduction

The modern scourge of malware is greatly exacerbated by the effective protections that are placed around the malicious files. These protections, referred to as packers or software armoring, regularly confound antivirus detection software. In the latest AV Comparatives proactive retrospective tests, it was found that detection was substantially less than ideal [2]. The result for the best vendor was 69%, while the lowest was 14%, for detection of new viruses. The average detection rate for new threats was only 42%. In private testing, Damballa, Inc. [9] showed that overall detection rates were very similar. A similar study has shown that while the overall number of samples is increasing in a super-linear manner, the families or underlying code bases of the malware grow in a much slower manner [11].

The explanation of this increase can be attributed partially to the malware author's easy access to defense systems. With these systems in hand, the antivirus sample can be easily modified to subvert detection [8]. A study by Gutmann [15] showed that malware authors are selling copies of their tools that are guaranteed to subvert antivirus for a certain number of signature updates. The customer can choose the number of updates, which are sold in a subscription manner. In practice, these modifications are fairly trivial to implement but extremely complex to detect correctly.

The obfuscation problem is difficult to address. A proof showing that the obfuscation problem is Turing complete has been shown [27]. This tells us that there is no way to generically tell when a program is completely unpacked. This work gets around this problem by using a heuristic to determine tell when a program is likely to be unpacked. It was shown that there exists a set of modifications to make the detection of any virus in the domain of NP-Complete problems [6]. This creates problems for antivirus vendors trying to detect new and virulent strains among existing families.

It is with this motivation that we began our research in covert analysis and monitoring systems. If we can subvert the protection mechanisms and bypass the defenses of the malware, we can increase the effectiveness of existing detection software dramatically. We do not try to definitively detect the completion of the unpacking process, but apply a further set of heuristics to better determine the contents of the original executable. Dinaburg et al. [10] showed that it is possible to monitor a program without detection using Ether. This technique opened the door to a range of analysis techniques. Paleari et al. [23] reinforced

D. Quist (✉) · L. Liebrock
New Mexico Tech, Socorro, USA
e-mail: dannyquist@gmail.com

D. Quist · J. Neil
Los Alamos National Laboratory, Los Alamos, USA

J. Neil
University of New Mexico, Albuquerque, USA

the need to avoid detection by implementing instruction fuzzing for emulated CPUs. This supports identification of a large set of instructions that can be used to determine the type of emulator being used, as well as the version being used. An emulator will have a difficult time generating a bug-for-bug equivalent set of instructions. The advantage of running on raw hardware is that these variations can be ignored.

The generalized method discussed by Josse for unpacking and repairing an executable requires a number of steps [18]:

1. Find the original entry point
2. Perform consistency checks within the section headers
3. Rebuild the import address table (IAT)
4. Rebuild the relocation table

Using the view of a completely unmodified yet still instrumented system allows for interesting new analysis. First, it requires that any introspection into the system observe the computer's running state at the time of infection. To implement this, we read and parse portions of the Windows kernel data structures through the Ether API. This provides several improvements over existing analysis systems: Not making modification to the running system either in the userspace or kernel space of the system makes detection much more difficult. Second, many modifications used to subvert analysis are specifically targeted at the internal monitoring features of Windows. Using the kernel objects to infer characteristics improves the reliability. At some point the program's deceptions have to allow it to actually execute. Finally a level of safety can be assured using a complete VM based solution.

This paper makes several contributions: we demonstrate the methods necessary to repair dump files through a hypervisor controlled system. We then provide new original entry point detection techniques as they are applied to hypervisors. Finally, we contribute a new method of harvesting the imported APIs by reading the virtual address descriptors from the Windows kernel data structures. Each of these techniques expands on those established in [10,18,22,27] by altering the techniques to work inside of a fully hardware virtualized environment. Our primary goals are to improve detection of viruses through traditional antivirus software and improve manual reverse engineering.

The paper is organized as follows. We begin with a discussion of related work in Sect. 2. Section 3 discusses the mechanisms of hypervisor analysis used. Continued in that section are the methods of repairing memory dumps, accurately detecting the original entrypoint, and rebuilding the import tables using the kernel virtual address descriptors. Next, we discuss our testing methodology in Sect. 4 and the results in Sect. 5. We end with the conclusion and an overview of future work.

2 Related work

Automated unpacking is a well tread topic. PolyUnpack monitors each instruction as it is traced through the modified QEMU section [27]. Each instruction is decoded until a data section is reached. Renovo takes a different tactic of using a fully emulated environment to monitor execution [19]. Memory writes are tracked in such a manner as to keep a collection of "dirty" memory regions. At the granularity of every basic block, Renovo checks to see if a memory write has occurred. Upon execution inside of this memory region the executable is extracted and written to disk. Ether uses a method similar to PolyUnpack but leverages to the Intel hypervisor hardware to implement a covert monitoring system [10]. It uses a method similar to Renovo, but works at a memory block level. In Ether, once execution is seen in one of these areas, it generates a candidate dump of the executable from memory without modifying any of the components.

OllyBonE uses a method similar to that of Sparks and Butler [28] to detect memory writes [29]. OllyBonE's detections occur inside of OllyDbg and require manual interaction. Saffron operates similarly to OllyBonE but removes the debugger component and adds a dynamic translation component [25]. This allows Saffron to resolve addresses at the instruction granularity as opposed to the page boundary in OllyBonE. Josse [18] uses the QEMU system to implement granular monitoring of executables to enable manual human analysis. Detection of the OEP by Josse uses a binary differencing method to detect execution that differs in memory versus that on disk. If the two instructions do not match, this is marked as the OEP of the suspected program. Import reconstruction consists of monitoring the execution of the program and watching for known API calls.

When validating each of the previously mentioned techniques, most did not address the proper repairing of executables. None of them corrected or attempted to fix the OEP detection, other than to provide a large set of possible candidates. It is with this motivation that we attempt to address these problems.

3 Hypervisor based analysis

Traditionally malware analysis has occurred using off-the-shelf components such as VMWare or Microsoft's Virtual PC. These virtual machine products use a trap-and-emulate technique for maintaining the separation between guest and host operating systems. This provides a fast management interface, but suffers from detectability. The primary reason for detectability using this method is the Intel architecture's inability to virtualize correctly [26]. Ferrie [12] described techniques in which it was possible to detect all of the popular virtualization products. The root of all these problems is that each of these virtualization systems lack a perfect bug-for-bug emulation of the Intel CPU instruction set. Finding a way to maintain undetectability vastly improves the ability of malware to detect it is being instrumented [13].

The advent of modern hardware based virtualization has traditionally been used as a method of optimizing the use of hardware for servers. While the performance improvements of these hardware platforms have been challenged [5], the ability of Ether to hide itself has been significantly improved [10]. Our work uses the Ether framework. Ether extends the functionality of the Xen hypervisor system by adding the ability to monitor application events at a per-instruction level. Ether provides several methods to monitor a running executable: instruction tracing, API tracing, and a generic unpacker detector. In testing, we have found that most malware is unable to detect the presence of the hardware virtualizations.

No virtualization system is completely undetectable. In testing we find that viruses do not currently detect the presence of Ether. The Intel hardware virtualization system (VMX) was created to allow the Intel CPU to manage multiple operating systems on a single CPU. The seminal work by [26] showed that the original implementation of the Intel IA-32 architecture did not support virtualization. There have been many ways to get around this limitation. For example, QEMU has been shown to be a very popular choice for code instrumentation. However, each virtualization system has been shown to be detectable [12]. Hardware virtualization has been shown to be detectable by analyzing covert channels [20]. While each of these systems can be detected, we prefer Ether for a few reasons. First, Ether is an open project with source code readily available. Second, assembly code is executed on an actual Intel CPU, instead of being emulated. Many of the single-instruction virtualization detection methods are effective because virtualization systems don't emulate the Intel architecture bug for bug [24]. Finally, we prefer Ether as it is a supported hardware method for evading detection. As new techniques are added, it is possible to subvert them by adding more handling code. This creates additional overhead, but still preserves the system integrity. The underlying operating system was a very good design decision by the Ether developers.

After reducing the detectability of the monitoring system, the next step is to repair the executable to improve analysis. We add portable executable (PE) repair functionality to the unpacker. The first result is fewer false-negatives from commercial antivirus scanners. Second, common reverse engineering tools such as IDA Pro are able to load and parse the file for manual analysis.

3.1 Repairing portable executables

The portable executable (PE) format is the binary format used by Microsoft's Windows platforms. Normally this contains

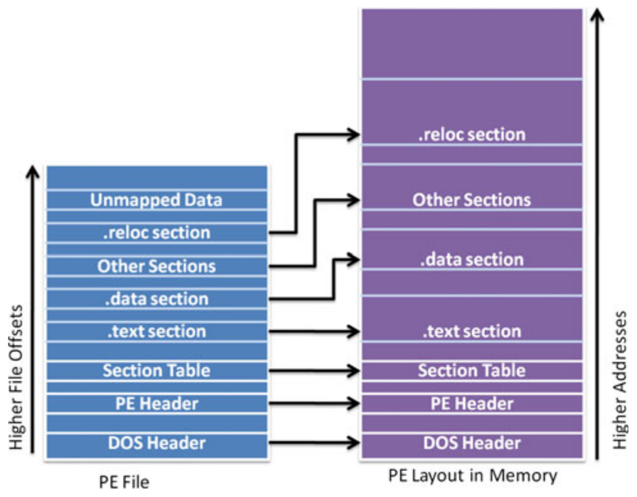

Fig. 1 PE File and Memory Layout Differences

the information that is needed to run a program. It contains the imported APIs, the memory usage requirements, executable code, and data for the executable. Some of the earliest obfuscations for software abused the optimizations of the Windows loader's handling of the PE format. The method used to dump the executable is to take the starting address pointed to by the ImageBase field in the optional header. This copying process does not produce the original pre-obfuscated version but rather the in memory view of the file. This allows for manual reverse engineering and analysis.

In order to prepare an executable for analysis, a couple of steps must be performed. First, there is a difference between the memory layout of a PE on disk versus in memory. During execution the program will need temporary memory storage for its stack. This is defined at runtime and does not have actual data prior to load time. To accommodate this and to save space in the actual executable, Windows allows for a post-load execution space to be created without including it in the file. Figure 1 illustrates this size difference. These spaces show up as sections that have a section size of an arbitrary amount and a raw size of zero. Memory in a loaded executable is also aligned on cache boundaries to increase performance. This has the overall effect of creating a different view of the file in memory and on disk.

We fix the executable using the following steps: first, iterate over all the section headers. At each section header, the size of data on disk is set to be the size of the data in memory. Second, the virtual pointers are updated to point to the area in memory where they would exist. Since the Windows loader places all of the information in the correct place at runtime using a relative virtual address, the dumped data will match the memory. The side effect of this technique is that the dumped executable is much larger than the original. This is mitigated by the fact that the data can now be analyzed. The final step is to fix the original entry point.

```
Input: Set of all executed Instructions and Operands
Output: Set of all candidate OEPs
$OEPs = \{\}$;
for each Executed Instruction $M_i$ and modified address $M_w$ do
    if $Addr[M_i] > 0$ then
        $OEPs$ += $M_i$;
    end
    else
        if $M_w \neq null$ then
            Addr[$M_w$]+=1;
        end
    end
end
return OEPs;
```

Fig. 2 Generic unpacking algorithm

Table 1 Number of candidate OEPS by packer using the algorithm in Figure 2

Packer	Detected OEPs
Armadillo	1
Petite	1
UPX	1
UPX Scrambler	1
Aspack	2
FSG	2
Pecompact	2
PEPack	12
VmProtect	12
Asprotect	15
Themida	33
Yoda	43
PEX	133
MEW	1018

3.2 Original entry point detection

The original entry point (OEP) is the address of code of the program prior to obfuscation by a packer. This address is critically important as it gives a starting point for all of the analysis of the original code. With an incorrect original entry point, IDA Pro will not be able to find and parse all of the executable instructions within a program. When a packer is applied to a program the entry point for the program is replaced with the packer's address. Finding the OEP can be difficult to impossible, based upon the packers obfuscation methods.

A standard algorithm exists and is used by various automatic unpackers to find the original entry point [10,19,25,27]. The basis of this algorithm is to keep a hash table of all addresses where a memory write has occurred. When an execution is detected in one of these address spaces, this is marked as a candidate OEP. This algorithm is detailed in Fig. 2.

While this algorithm will identify candidate OEPs, it can produce a large number of unpacked files. The number varies depending on the unpacking algorithm. Table 1 shows the number of candidate OEPs generated by various packers. The primary indicator of how well the algorithm will perform is based upon how many times the self-modified code is executed. In packers such as MEW or FSG, the code is re-executed many times before finally executing the original entry point code. One solution to this problem is to analyze each file individually based on the number of candidate OEPs. This can be tedious if performed manually, and due to the expansive nature of some packers, it can generate an exhorbitant amount of data. We chose to modify this unpacking process to limit the total number of candidate dump files.

3.2.1 Stack based OEP detection

The modification we make to our algorithm is to take the very last candidate OEP generated and perform a stack based analysis. Since candidate OEPs are generated in execution order, the last one will have a function stack. The stack inside a running executable contains the return address, arguments, and local variables for a running program. In a normally compiled executable, this creates a trail of crumbs that will lead to the original executable. We start by looking at the current value of the base pointer *ebp* and load the offset where the return address exists. To verify the return address is correct, we simply check to see if the address is within the executable's address space. Analyzing the return address should yield a call instruction if a standard program call frame exists. Upon finding this return address we continue the process until there is a non-valid memory address. A non-valid memory address will either be an address outside of the executable's loaded memory addresses, or a bogus value (such as null). The very last address in this chain is very close to the beginning of the original OEP. To find the first address we disassemble each instruction previous to the current one until we get a junk instruction. The last valid instruction is then marked as the new OEP. This method has been used as a debugging technique [4] as well as a reverse engineering tool for security researchers [7]. Our use of this technique is to make an educated guess about the top of the call stack and the beginning of the code for each call.

The method is effective for a couple of reasons. First, most packers implement unpacking code in a method that either abuses or does not use the standard stack framing system.

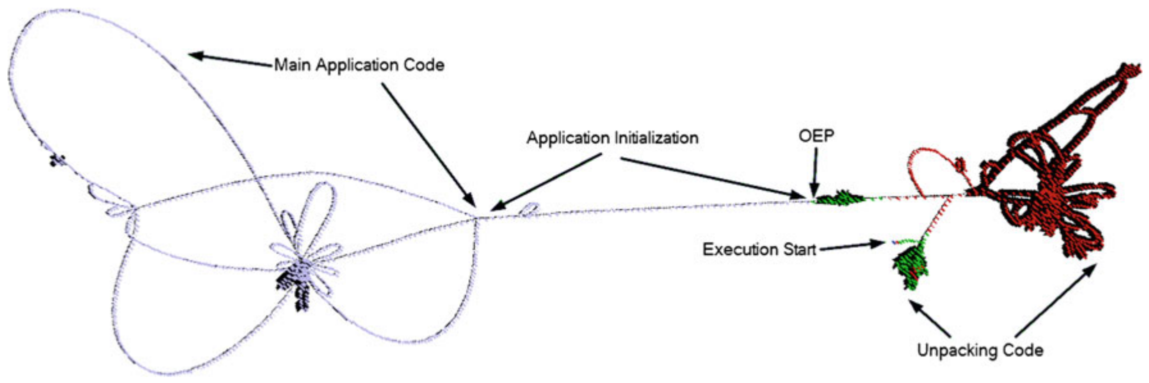

Fig. 3 Visualization of the Netbull Virus Protected with the Mew Packer

Second, the packers usually do not share the same section as the original code. This is illustrated in Fig. 3 where sections are distinguished by a transition between large numbers of nodes of the same color. Each vertex of the code represents a basic block of execution of the program. Basic blocks consist of all assembly operations that are contained between two adjacent branching operations.

3.2.2 Visualization of unpacking

The colors represent the areas, of memory the basic blocks represent. Red indicates execution in a section with high entropy. Most packers and obfuscators are able to compress an executable such that it has a uniform distribution of data. Areas of high entropy inside of the original executable indicate where the program has transitioned to the unpacked portions of the executable. Green is execution into non-existent code sections; if the executed instruction is non-existent in the on-disk executable, this indicates that the code is generated dynamically or is self-modifying. These data areas commonly allocated are dynamically allocated in heap space, such as that returned by malloc. Light purple shows execution where a section exists on disk, but not in the run-time executable. This is most often found when data is allocated in the PE section headers, but not used until runtime. Neon green shows instructions that differ between the in-memory and on-disk executables. This is another sign that points to execution of self-modifying code.

Until the executable runs in the light purple section, there is no clear stack structure. The algorithm from Fig. 2 would trigger a memory dump when the program reaches the green areas. The final area, the light purple section, is the final unpacked version of the code. This is where an actual call-frame system will be setup and represents the area with the original entry point. Figure 4 highlights these areas with a zoomed view of the unpacking loop from Fig. 3.

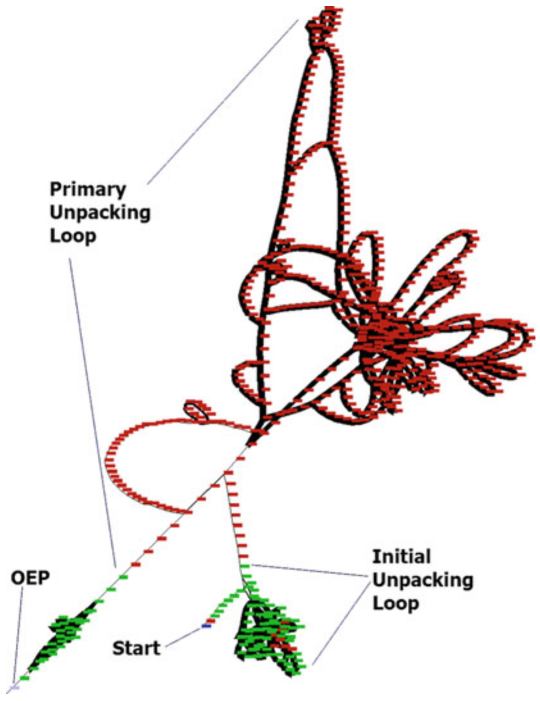

Fig. 4 Zoomed view of the packer loops from figure 3

3.2.3 Verification

To verify the effectiveness of this technique, we took an unmodified, unprotected copy of the Netbull and made note of the entry point: virtual address 0x0040bb08. We then packed the sample with the UPX, Aspack, Asprotect, Mew, Pex, Yoda, PEPack, VmProtect, FSG, Themida, and Armadillo packers. To verify that the packers worked correctly we ran the files and verified that the virus executed and performed its usual tasks. We then ran an instruction trace on each packed file as well as the unpacking process. To validate the OEP detection we simply verified that the entry point detected by the unpacking code matches the address we recorded previously. Of the fourteen packers we tested all OEPs were found correctly.

3.3 Virtual address descriptor import rebuilding

When final unpacking is completed and a program is being extracted from memory the next step is to rebuild the imports. Several tools exist to perform this task inside of a running Windows systems [14,21,27]. These implementations are designed to use the native Windows API. However, this method is flawed, as it requires tools to run in the guest operating system. Many viruses can actually detect the presence of these analysis tools and modify their behavior. A completely separate system is needed to extract the information through the hypervisor.

Our contribution builds upon the techniques discussed by Josse [18]. Specifically we adapt it for use inside of Ether's hypervisor environment. Analysis and modification of kernel data structures for malware analysis is not new. Josse discussed parsing kernel structures for recovering imported data. We add a new system based on parsing the internal Windows Virtual Address Descriptors (VAD). This allows for the resolution of API calls to the actual DLL. This is discussed in Sect. 3.3.3.

The generalized process to rebuild the imports is as follows.

1. Identify the API calls in the executable code.
2. Find the location of the import address table.
3. Determine the dynamic link library (DLL) of each imported API.
4. Add the newly reconstructed import address table to the executable.

It was necessary to alter steps 1 through 3 in order to function inside of a hypervisor environment. Our contribution provides a technique to extract the relative data from the guest via the hypervisor.

3.3.1 Identification of API calls in executable code

Identification of API calls is performed in two steps. First we monitor all execution and log the address of the program counter (PC) in order to track execution of the program. When we see an execution that is outside of the executable's address space, we check for execution in an imported module. The first address after this transition is the first address in the imported module API. The module API is parsed and then translated into a format needed for the Import Address Table.

Our technique has the advantage of finding APIs that are obfuscated by the packer. Other import rebuilders do not resolve this indirection and will miss hidden or obfuscated API calls. This technique suffers from the problem of not being able to find unreferenced or unexecuted API calls. To address this deficiency we use a method from OllyDump and PolyUnpack to look for any indirect call or jmp instructions [14,27]. Scanning through all of the executable code sections, we can scan for the two byte code representation of these indirect calls. These indirect calls will first point to the loaded module's import address table (IAT) and then finally point outside the base executable code address space to the DLL. The IAT will contain a listing of all of the modules and APIs imported by the program at runtime. It is an ordered table that is populated with the destination API addresses inside of an imported dynamic link library (DLL).

3.3.2 Finding the import address table

The IAT is a critical piece of the puzzle to rebuild the imports. The table consists of a listing of addresses imported by each DLL and contains a listing of virtual addresses for each of the destination DLLs. These are the actual locations in the mapped DLL space that can be used to derive the character name for each function. The derivation of the actual names is performed by the inspection of the kernel virtual address descriptors (VADs) for each process.

Finding the beginning of the IAT is necessary to rebuild the complete set of imports. Each time an API is found using the method in Section 3.3.1, we traverse the list of imports in reverse until we find a null DWORD indicating the beginning of the list. This address is then compared to the lowest previously found candidate for the IAT. If the value is lower than the previous one, it is replaced. This address is used to populate an internal data structure containing the ordered listing of DLLs and the APIs imported by them for use in sect. 3.3.4

3.3.3 DLL API identification

Previous methods use the internal Windows APIs to resolve an address to a particular DLL. Invoking this API is not possible from the hypervisor so we must replace it. Our contribution adds the technique of parsing the Windows kernel memory management data structure in order to recover the DLL and API information for rebuilding.

Viewing and validating the import information through the process VADs is important for several reasons. First it provides a runtime view of the data that is loaded for the process. Many times an executable will obfuscate the locations of this information in the process memory space. Since the VADs are used to populate the page tables and page directories, this information will contain the correct information for a running process. Using the kernel data structures also lets us target the appropriate areas to parse DLL information from. Given that PE files have the same representation on disk as they do in memory, our parsing follows a similar format.

The structure of a virtual address descriptor is a binary tree of the addresses. Every time an address is noted or logged we walk the VAD tree to find the appropriate descriptor. Figure 5

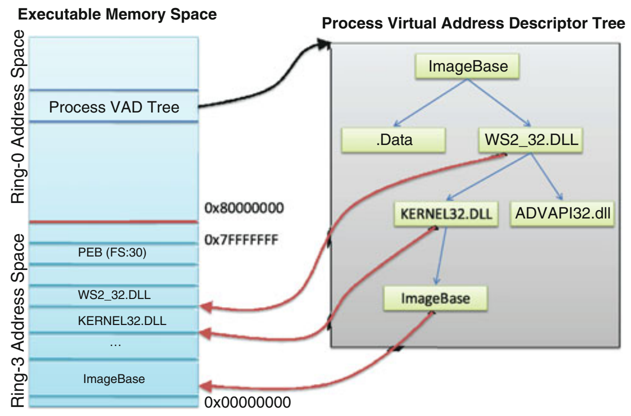

Fig. 5 Relationship of the VAD with process memory

illustrates the layout and relationship to the process memory. This descriptor contains a module path name which is either the executable or an imported DLL. The DLL then contains an export table which is used to extract the name of the API.

To recover the particular VAD for a process, we parse the kernel process execution block. An addition to Ether allows for logging of this important area of kernel memory on each instruction execution in the modified environment. The VAD is not modified during this logging process. When the unpacking process is complete (discussed in Sect. 3.2), this virtual address table is used to discover each of the DLL's and API's belonging to the process.

To recover the API from the running process, Ether's read_from_guest API is called to extract the information. Each DLL contains an export listing of all the APIs it provides. DLLs will be mapped into the program's running process space by virtue of the import process. Since DLLs are PE files and the PE file maps itself into memory, we can extract and record each of the API entries. To do this, we first take the address of the indirect call and verify it is outside the image space of the running process (this is computed by adding the ImageBase with the SizeOfImage variables). Once we have the image base, we traverse the VAD tree looking for the range of memory that matches the running API address. From here we match the calculated offset of the file with the DLL's export table to find the name of exported function. If a text name for the executable is not present in the DLL, we use the ordinal value described in the Microsoft PE specification [1]. This is noted and inserted into a linked list of imported functions in the executable.

3.3.4 Rebuilding the newly constructed IAT

To repair the import table we add another section to the executable. This section contains the new import directory for the running program. We take our linked list of APIs, as well as the observed import address table, and use that to build a named list of imports. These are written out of the executable to disk during the completion of the process dumping.

Import obfuscation sometimes varies from the traditional methods. One example is the PE scrambler by Nick Harbour [17]. The obfuscation used in this technique creates a central dispatch point by which to hide the destination information. To address this type of attack, we use a second process monitoring method to rebuild the import tables. In the course of the execution of our program, we watch for executions outside of the executable's image base. If this data is tracked to memory that is mapped by a DLL, we can say that this is an API call. From here we build up an observed IAT containing the log of all the calls seen. Tools in the class of PEScrambler modify the base code to point to the API dispatcher. We simply change the instructions back to their original call structure using the IAT entry.

The combination of these techniques provides a robust method to reconstruct most import obfuscation techniques. Manual reverse engineering often begins with an analysis of the API calls. Malicious software will need to invoke these routines in order to affect change on the infected system. Having this information pre-built and reassembled allows for quicker analysis. Antivirus detection is also increased as import data is used to improve detection heuristics.

4 Testing and analysis

Our testing methodology consists of three strategies. First, we test using a collection of Linux virus scanners for samples that are detected by exactly one vendor. Second, we perform the same analysis on freshly collected samples. Finally, we take the samples from the two previous tests, wait 2 weeks, and test them against the latest updates from 40 antivirus scanners.

To begin testing, we took four antivirus programs which would run in the Linux operating system. We updated all of their signatures to the latest versions, and updated the software to the latest revisions. We then scanned our entire repository of 500,000 samples to randomly extract 1,000 samples with poor detection results. We define a sample as poorly detected if it is only found by exactly one antivirus scanner. The second test was to take a whole day's worth of freshly collected malware. These samples were collected via honeypots, web harvesting systems, and from malware trading networks. Finally to test the overall effectiveness, we allowed 2 weeks to pass and submitted all of the samples (consisting of 1,693 samples) to VirusTotal [3], a website which aggregates results from 40 virus scanners. This test allowed us to see what the detection improvement was after each of the vendors had been given a chance to develop new signatures. Each of the submitted samples were then executed to verify they showed malicious behavior.

To verify a file was malicious, we looked for three distinct characteristics. First, each of the samples was analyzed to show successful execution. This means that there were no observed crashes or unrecoverable problems with the sample. The second test was to validate that the viruses were modifying the state of the running system in some manner. To determine this state modification we took a baseline of the system before and after execution. We looked for added registry keys, modified files, and additions to startup systems. Finally, we verify a process is running correctly by looking for system calls made by the application. To make the determination that a sample is malicious, we look for any detection by any antivirus software. These tests validate, as much as possible, that the program runs and exhibits behavior consistent with that of a generic execution of a malicious program.

Our verification method reduced the original 1,000 samples down to 697 and resulted in 1,195 total samples for a single day of collection.

In order to implement the test, we set up a Windows XP Service Pack 2 virtual machine image to run inside of the Xen/Ether environment. To automate the unpacking process we enable file sharing, create a new account with administrative privileges, and start the virtual machine normally. The system is configured with a static IP address with no gateway in order to limit the network traffic to that generated in a locally confined network. Furthermore we isolate the Ether system behind a switch with no external network access. To speed up the start-up of the virtual machines, we create a ramdisk containing the original clean disk image. To analyze the file we then take the following steps:

1. Copy the cached clean virtual machine image to the runtime location
2. Start the VM inside the Xen system
3. Copy the executable to a writable network share on the system
4. Start Ether monitoring of the program
5. Execute the program using the Winexe program[16]
6. Wait 5 min to allow the program to run

The 5 min wait time was chosen as a good trade-off in efficiency of unpacking and thoroughness of the unpacking system. While there are samples that will not be correctly analyzed in this timeframe, our testing has shown 90% will be unpacked.

After this process has ended, the results will be a set of copies of the executable from the running program memory. Each of these files are scanned by the antivirus scanners. We define improvement as the difference in detection rates before and after unpacking.

Finally we omit the names of the antivirus programs in all tests. Since we are looking for improvement in antivirus scanner performance, the individual scanner is not the primary concern. Given the sensitive nature of calling out antivirus vendors, we defer to other sources (such as AV Comparatives [2]) for a much more thorough job ranking of individual scanning effectiveness and performance. Such rankings are not the intended goal of this paper.

5 Results and analysis

The results from the scanning process were very positive. Out of each of our tests outlined in Sect. 4, we showed improvement across all scanners. The selection of samples that were detected by only one scanner showed the most improvement. The highest improvement of all the scanners was 45.23%. The average improvement in this area was 19.86% and the lowest improvement was 0.68%. The newer samples from a single day of freshly collected malware showed improved scanning as well. The average improvement across all scanners was 7.37%, with the highest being 12.54% and the lowest at 1.70%. Finally the delayed scanning with 40 scanners showed a lower, but still improved testing result. The highest improvement in this test was 11.5%. The average improvement was 1.37%, and the lowest improvement was 0%.

The total detection rate of all the pre-analyzed malware samples was 66.78%. After the hypervisor analysis and repair the average was 67.73% detection. Each of the samples in our single-day collection was detected by one or more scanners.

The highest value improvement was shown in the same day tests. Such samples are threatening networks on a daily basis. The total detection average of all the vendors in this area is about 67%. The single day improvement average of 7.37% shows that more threats can be successfully identified.

After 2 weeks, the scanner developers have had time to develop new signatures for the threats. It is difficult for any vendor to be able to detect all threats. The worst scanner showed an 11.50% improvement. This is important to note as our tool allows better coverage of the new viruses invading systems.

Figure 6 shows the total detection rates for all the samples submitted to VirusTotal, a mass scanning website, after a 2 week delay. The detection rates from Fig. 7 show the relative improvement after our unpacking process has been applied. A higher score indicates that the our processing improves the results. A low score can indicate a couple of outcomes. First, the signatures for a particular scanner do not include an adequate detection mechanism for a sample of malware. To get an idea of whether this occurs, we can analyze the improvement graph (Fig. 7) and the total detection graph (Fig. 6). Scanners J, U, Y, AF, and AH all have low improvement scores. When looking at the total detection rates, each of these scanners score well below the 50% mark. It is reasonable to conclude that these scanners do not have signatures for these particular malware samples. Second, each of the

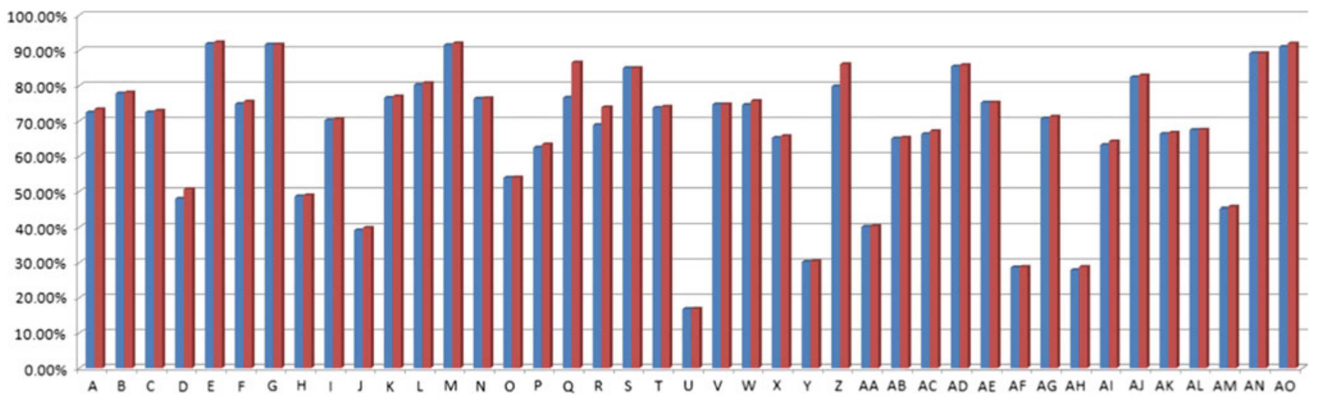

Fig. 6 Total detection rates after 2 weeks submitted to VirusTotal. Blue is the detection rate before unpacking and red is the detection rate after. The results include 40 scanners

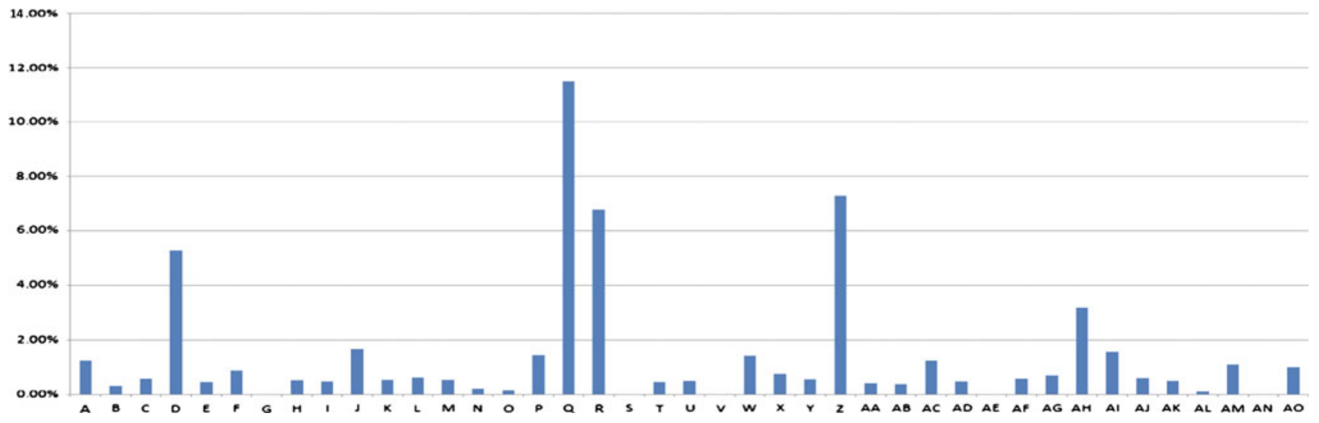

Fig. 7 Percentage improvement over all samples scanned by Virustotal after 2 week period. These results include 40 scanners

scanners has had 2 weeks to develop signatures for the samples we provided. These viruses were collected from well-known sources that share with members of the antivirus community. Thus 2 weeks allows for thorough development of both signatures and improvements to unpacking engines. Our unpacking mechanism shows improvements in just 5 min. Our technique exploits the fact that newer malware families do not appear as often as new obfuscation techniques. We leverage our improved unpacking technique to modify the malware into a form that is detected by antivirus signatures.

Figure 8 shows another way of looking at the data. In this graph the top bars denote the maximum improvement for each of the tests. The bottom shows the minimum improvement. 50% of the data is contained in the colored boxes. The space between the top of the box and the maximum mark is where the upper quartile is contained. Similarly the space between the bottom of the box and the minimum is where the bottom quartile is located. The dark line inside each of these boxes indicates the median improvement. The best improvement was among samples detected by at most one scanner, denoted P in the graph. S shows the testing performed on a single day, while VT shows the average results from the 2 week delayed VirusTotal test. Each of these indicate an average improvement. Overall each of the tests show the improvement is weighted towards the lower values.

In each of the tests, improvement was shown. The highest rate of success was seen in the poorly detected samples test. The samples in this test were not necessarily current; some existed for several months before this test was performed. Antivirus companies place higher value on tracking current threats as they are the ones with the greatest effect on their customers. Older samples do not tend to get as much attention as the newer threats. The vendors will have a signature, but may not have a method of deobfuscating each of the packers. Removing these protections will allow the existing signatures to match the samples more reliably.

6 Future work

Several improvements can be made upon this work. First, we would like to address the performance considerations of the unpacking process. Currently our method is to let the sample

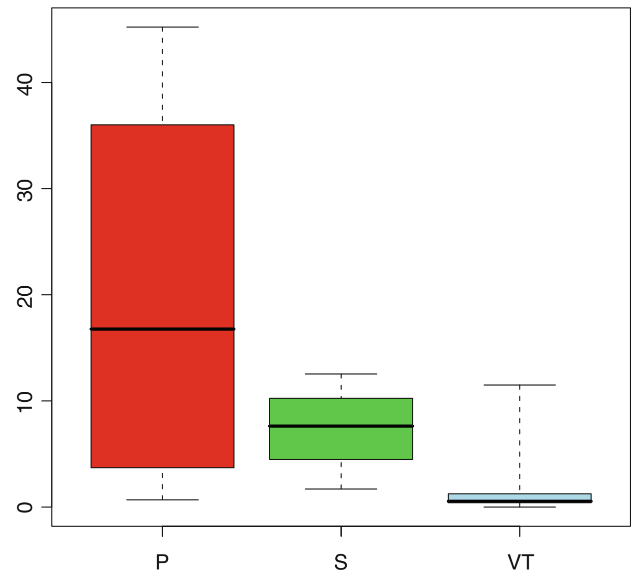

Fig. 8 The *y* axis shows the percentage improvement for all unpacking methods. The *x* axis contains labels for each of the test types. P denotes detection improvement for poorly detected samples (only one scanner detection) with Linux AV scanners, S is samples from a single day with Linux AV scanners, and all samples submitted to Virustotal after 2 weeks are listed under VT

execute for 5 min. We would like to explore other methods for determining whether a sample has been unpacked. Integration of the visualization and manual reverse engineering would allow for better control of the unpacking process. Finally, we would like to integrate Ether with several existing debugging frameworks such as OllyDebug, WinDBG, and IDA Pro's built-in debugger.

7 Conclusion

We have applied a new unpacking method for improving malware detection to hypervisor based analysis. First, we discussed the section and file repairs. Second, we showed a technique for recovering the original entry point using data harvested from dynamic analysis. Finally, we present a new method for rebuilding import information, while preserving the isolation of the infected host. These improvements were then measured against three different scenarios: A poorly detected set of malware, fresh malware collected on a single day, and total detection from 40 different virus scanners after 2 weeks. For each of these tests, our tool decreased the effectiveness of obfuscations used by malware authors. Thus we provide an improved defensive mechanism to aid in the protection of network resources.

Acknowledgments This work was partially funded by NSF Scholarship for Service grant DUE-0313885. The authors would like to thank Curt Hash, Mike Fisk, Peter Silberman, Alex Kent, Alex Brugh, and Scott Miller for their help with this research.

References

1. Microsoft portable executable and common object file format specification. Specification Document, March 2008. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
2. Antivirus Comparatives - proactive/retrospective test (on demand detection of virus/malware). Online Report, November 2009. http://www.av-comparatives.org/comparativesreviews/main-tests
3. Hispasec Systems, Virustotal: Free online virus and malware scan. Company Webpage, November 2009. http://www.virustotal.com/
4. Manually walking a stack. Webpage, November 2009. http://msdn.microsoft.com/en-us/library/cc267826.aspx
5. Adams, K., Agesen, O.: A comparison of software and hardware techniques for x86 virtualization. In: ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 2–13. ACM, New York, NY, USA (2006)
6. Borello, J.-M., Mé, L.: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**, 211–220 (2008)
7. Cachaalany, E.: An attempt to reconstruct the call stack. Hex-Rays Blog, September 2009. http://hexblog.com/2009/09/an_attempt_to_reconstruct_the.html
8. Christodorescu, M., Jha, S.: Testing malware detectors. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), pp. 34–44. ACM Press, Boston, MA, USA (2004)
9. Damballa. Risk calculator. Company Webpage, November 2009. http://www.damballa.com/overview/risk.php
10. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2008)
11. Eckelberry, A.: The growth of malware. Blog Post (Jan. 2008). http://sunbeltblog.blogspot.com/2008/01/growth-of-malware.html
12. Ferrie, P.: Attacks on virtual machine emulators. Symantec Advanced Threat Research Whitepapers (2007)
13. Ferrie, P.: Anti-unpacker tricks - part one. Virus Bulletin (2008)
14. Gigapede, Ollydump 2.21. Webpage (2009)
15. Gutmann, P.: The commercial malware industry. In: Defcon 15, Las Vegas, NV (2007)
16. Hajda, A.: Winexe. Online Download, November 2009. http://eol.ovh.org/winexe/
17. Harbour, N.: Advanced software armoring and polymorphic kung-fu. In: Defcon 16, Las Vegas, NV (2008)
18. Josse, S.: Secure and advanced unpacking using computer emulation. J. Comput. Virol. (3), 221–236 (2007)
19. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM) (2007)
20. Lauradoux, C.: Detecting virtual rootkits with cover channels. In: Proceedings of the 17th EICAR Conference, Laval, France, EICAR (2008)
21. MackT, Import reconstructor 1.7, March 2008, http://www.woodmann.com/collaborative/tools/index.php/ImpREC
22. Martignoni, L., Christorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Proceedings of the 2007 Computer Security Applications Conference, pp. 431–441. Miami Beach, FL, USA (2007)
23. Martignoni, L., Paleari, R., Roglia, G.F., Bruschi, D.: Testing CPU emulators. In: Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA), pp. 261–272. ACM, Chicago, IL, USA (2009)
24. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A fistful of red-pills: how to automatically generate procedures to detect

CPU emulators. In: Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT), ACM, Montreal, Canada (2009)
25. Quist, D., Smith, V.: Covert debugging: Circumventing software armoring. In: Blackhat USA, Las Vegas, NV (2007)
26. Robin, J.S., Irvine, C.E.: Analysis of the intel pentiums ability to support a secure virtual machine monitor. In: Proceedings of the 9th USENIX Security Symposium, Denver, CO (2000)
27. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In: ACSAC, pp. 289–300 (2006)
28. Sparks, S., Butler, J.: Raising the bar for windows rootkit detection. Phrack, **11**(63) (2005)
29. Stewart, J.: Ollybone: Semi-automatic unpacking on ia-32. In: Defcon 14, Las Vegas, NV (2006)