Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen Based Honeypots

Corrado Leita[1], Marc Dacier[1], and Frederic Massicotte[2]

[1] Institut Eurecom, Sophia Antipolis, France
{leita, dacier}@eurecom.fr
[2] Communications Research Centre, Ottawa, Canada
fmassico@crc.ca

Abstract. Spitzner proposed to classify honeypots into low, medium and high interaction ones. Several instances of low interaction exist, such as honeyd, as well as high interaction, such as GenII. Medium interaction systems have recently received increased attention. ScriptGen and Role-Player, for instance, are as talkative as a high interaction system while limiting the associated risks. In this paper, we do build upon the work we have proposed on ScriptGen to automatically create honeyd scripts able to interact with attack tools without relying on any a-priori knowledge of the protocols involved. The main contributions of this paper are threefold. First, we propose a solution to detect and handle so-called intra-protocol dependencies. Second, we do the same for inter-protocols dependencies. Last but not least, we show how, by modifying our initial refinement analysis, we can, on the fly, generate new scripts as new attacks, i.e. 0-day, show up. As few as 50 samples of attacks, i.e. less than one per platform we have currently deployed in the world, is enough to produce a script that can then automatically enrich all these platforms.

1 Introduction

Honeypots are powerful systems for information gathering and learning. L.Spitzner in [1] has defined a honeypot as "a resource whose value is being in attacked or compromised. This means, that a honeypot is expected to get probed, attacked and potentially exploited. Honeypots do not fix anything. They provide us with additional, valuable information". In [1] honeypots are classified according to the degree an attacker can interact with the operating system.

In high interaction honeypots, the attacker interacts with real operating systems usually deployed through virtual emulators. This ensures a very reliable source of information, but also brings some major drawbacks. High interaction honeypots are real hosts and therefore can be compromised: the maintenance cost and the risk involved in them is high. Also, the amount of resources required to deploy such honeypots is usually substantial.

In low interaction honeypots such as honeyd [2], the attacker interacts with simple programs that pretend to behave as a real operating system through very simple approaches. Honeyd uses a set of scripts to implement responders to the most common services. Given a request, these scripts try to produce a response that mimics the behavior of the emulated server. This approach has two major drawbacks. On the one hand, the manual generation of these scripts is a tedious and sometimes impossible task due to the unavailability of protocol specifications. On the other hand, they are often not able to correctly handle complex protocols, limiting the length of the conversation that the honeypot is able to carry on with the client. Since many exploits deliver the malicious payload only after an exchange of several packets with the server, low interaction honeypots are often not able to carry on the conversation long enough to discriminate between different types of activities. For instance, in our experience within the Leurre.com project [3,4,5,6,7,8], due to the lack of emulation scripts we have been able to observe only the first request of many interesting activities such as the spread of the Blaster worm [9]. But since Blaster sends the exploit in the second request of its dialog on port 135, we have never been able to observe such a payload. Therefore it becomes very difficult to distinguish Blaster's activity from other activities targeting the same port using solely the payload as a discriminating factor.

The lack of emulation scripts led us to investigate the feasibility of automatically generating emulators starting from samples of protocol interaction using the ScriptGen framework [10]. We showed how it was possible to take advantage of the statistical diversity of a large number of training samples to rebuild a partial notion of semantics. This can be done in a completely protocol-independent way: no assumption is made on the protocol behavior, nor on its semantics. Our first results showed how ScriptGen had been able to successfully carry on a small segment of conversation with the clients, proving the validity of the method but also showing the need to improve emulation.

In this paper we take a big step forward, showing how it is possible to dramatically increase the emulation quality by coupling the seminal work presented in [10] with a number of novel contributions. Specifically, this paper presents i) an innovative algorithm to infer dependencies in the content of protocol messages (*intra-protocol dependencies*) without requiring the knowledge of protocol semantics; ii) a new algorithm to generate relations in the interaction of multiple TCP sessions (*inter-protocols dependencies*); iii) a proxying algorithm that allows a ScriptGen honeypot to automatically build a training set to refine its knowledge of the protocol reacting to the detection of new activities.

This paper is organized as follows: section 2 gives an overview on the current state of the art in the field; section 3 introduces the main concepts and contributions of this paper; section 4 gives an in-depth description of the novel contributions to the ScriptGen framework; section 5 shows the experimental validation performed on the new ScriptGen emulators; section 6 concludes the paper.

2 State of the Art

The contributions of this paper put their roots in a seminal work presented in [10]. ScriptGen is a method that aims at building protocol emulators in a completely automated and protocol-independent way. This is possible through an algorithm detailed in [10] called *region analysis*. Region analysis uses bioinformatics algorithms [11] as primitives to rebuild protocol semantics and to raise the training data to a higher level of abstraction. This is done in a completely protocol-independent fashion: no assumption is made on the protocol semantics or on the protocol behavior. This allows us to build emulators for protocols whose specification is not available or partially unknown. In [10] we validated the approach, and we identified a number of limitations that were preventing ScriptGen emulators from correctly carrying on complete conversations with a client.

Shortly after our initial publication, Cui et al. presented the results of a similar approach, named RolePlayer [12], carried out in parallel to ours. These authors have the same goals in mind but have imposed different constraints on themselves. RolePlayer uses as input two cleaned and well-chosen *scripts*. These scripts are training samples of the conversation that must be emulated. As ScriptGen does, RolePlayer uses bioinformatics algorithms to align bytes and delimit fields inside the protocol byte stream. RolePlayer gives semantic value to the various fields using additional information (IP addresses, host names used in the conversation) and a simple "cookbook" of rules to give an interpretation to the various fields. This "cookbook" is a set of heuristics deduced from observations made on various known protocols.

The RolePlayer approach offers a very elegant solution but it is worth noting that it is orthogonal to ScriptGen's philosophy and shows a number of limitations. First of all, the usage of only two scripts in the alignment phase requires carefully chosen samples in order to avoid false deductions. This process can be easily done by a human operator, but an automatic preparation of the training set does not appear straightforward. Furthermore, it appears that the design of well behaved samples precludes the usage of this technique for online creation of scripts as we propose to do it in section 4.3. To accomplish the same purpose, ScriptGen performs the analysis on a statistically significant number of samples. ScriptGen exploits the statistical diversity of the samples to minimize false deductions without requiring any sort of human intervention. As we will show in this paper, this property is extremely interesting when implementing automated learning of new activities. In fact, we will show in this paper how ScriptGen is able to react to 0-day attacks, exploiting its characteristics to learn the behavior of the new activity. It does so by building in a completely automated fashion a new training set and using it to refine its knowledge of the protocol. For this to be possible, no human intervention must be necessary; the process must be totally automated. ScriptGen, being completely automated and protocol-agnostic, fulfills these requirements. As opposed to that, the additional manual input required by RolePlayer to generate the emulators is a severe limitation with respect to this objective. Also, RolePlayer takes advantage of a set of heuristics that are deduced from the knowledge of existing protocols. Even if these heuristics might be valid for a certain number of protocols, they restrict the generality of the method itself by taking into consideration only the number of well-known protocols for which these assumptions hold. Finally, RolePlayer as

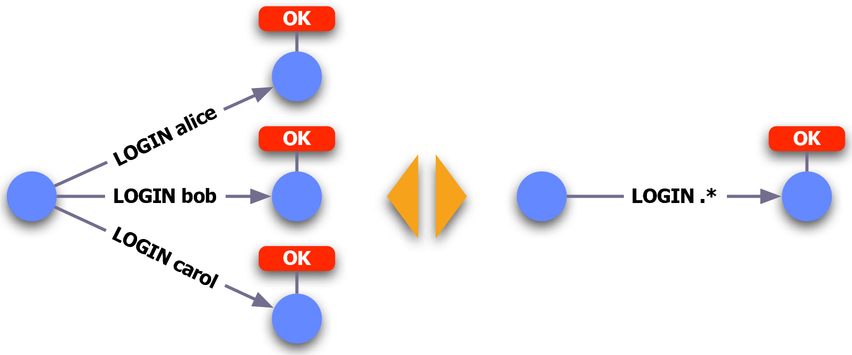

Fig. 1. Simple example of semantic abstraction

described in [12] seems to be able to replay only a single *script* at a time. It does not offer a structure to handle in parallel different protocol functional flows. A ScriptGen emulator instead is able to map different activities to different paths of the internal protocol state machine.

A completely different approach is instead followed in the context of the *mwcollect* project [13,14], that has recently merged with the *nepenthes* project. These tools use a set of vulnerability modules to attract bots, analyze their shell code and use download modules to fetch the malware code from the attacking bot. Currently, the vulnerability modules are manually handled and specific to each known exploit, but a future integration of the ScriptGen approach with these tools might lead to very interesting results.

3 Related Work and Novel Contributions

The work shown in this paper builds upon the work introduced in [10]. The ScriptGen approach allows building protocol emulators in a protocol-independent way: no assumption is made on protocol behavior, nor on its semantics. The approach uses a set of training conversations between an attacker and a real server to build a state machine representing the protocol language from an application level point of view. Each state is labeled with the corresponding server answer; each transition is labeled with client requests. When the emulator receives a request from the client that matches the label of one of the outgoing transitions from the current state, it moves to the corresponding future state and uses its label to reply to the client. Since we are not assuming any knowledge of protocol semantics, the client requests are seen as simple byte streams and they are therefore too specific: the generated state machines would be unnecessarily large and not able to handle any kind of variation from the data seen during training. For this reason we introduced the region analysis algorithm, detailed in [10]. This algorithm is able to take advantage of the statistical diversity of the samples to identify the variable and fixed parts of the protocol stream, using

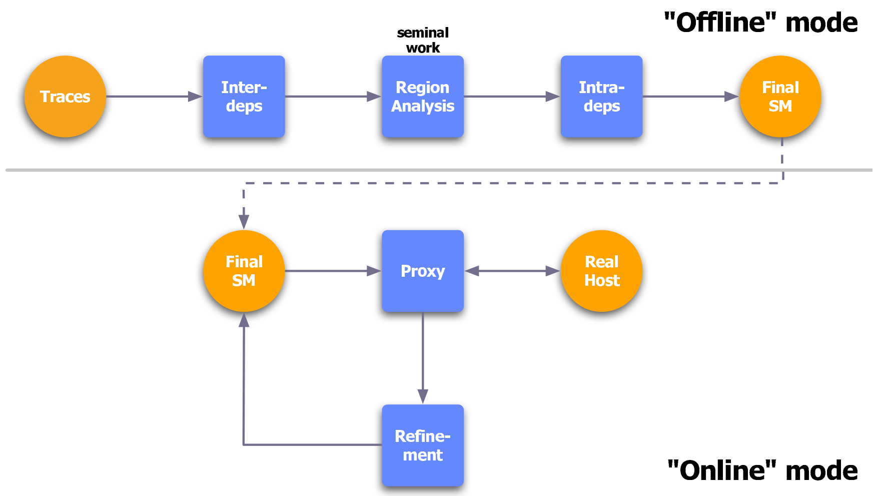

Fig. 2. The ScriptGen framework

bioinformatic algorithms. Using clustering and refinement techniques, the algorithm aggregates the outgoing edges and produces as output a semantic-aware representation of their value. The protocol stream is thus transformed in a sequence of mutating regions (groups of mutating bytes with no semantic value) and fixed regions (groups of bytes whose content is considered as discriminating from a semantic point of view). Figure 1 shows an example of the semantic abstraction introduced by region analysis: the algorithm is able to infer from the statistical diversity of the samples part of the underlying protocol structure, distinguishing the "LOGIN" command from the username. The LOGIN command will generate a fixed region and will be considered as discriminating in determining the protocol functional behavior. The username instead will generate a mutating region and the content of the field will not be considered as semantically discriminating.

We showed in [10] a preliminary validation of the method, that was able to exchange a limited number of packets with several attacking sources. While these first tests showed the validity of the method, they also underlined a number of limitations and the need for additional enrichments to the initial work. This led to the ScriptGen framework presented in this paper and represented in Figure 2. This paper introduces a set of novel algorithms aimed at circumventing the limitations identified in [10] and demonstrates how to exploit the potentials of this approach. These can be summarized as follows:

1. **Support for intra-protocol dependencies.** In many protocols, one of the two peers involved in the conversation chooses a cookie value to be put in the message. For instance, in NetBIOS Session Service the client chooses a 16 bit transaction ID: for the server answer to be accepted, it must use the same value in the corresponding protocol field.

2. **Support for inter-protocol dependencies.** In many different cases the state of the emulation goes further than the single TCP session. For instance, successfully running a buffer overflow attack on a certain port might open a remote shell on a previously closed port. If that port has been open since the beginning, the exploit might refuse to run. Also, multiple TCP sessions may be interleaved (such as in FTP) generating dependencies between them.
3. **Proxying and automated learning.** The stateful approach and the structure of the state machine itself allows an extremely precise detection of new activities. Every time that a request is received and no outgoing edge from the current state matches with it, an alert can be triggered. Taking advantage of a proxying algorithm to carry on the conversation with the client, it is possible to build a training set to automatically refine the existing state machine, thus reacting in a very precise way to 0-days attacks.

4 Dependencies and Proxies

4.1 Intra-protocol Dependencies

Examining the conversation between a source and a server we can identify two different types of dependencies. We can observe dependencies in the content of a TCP session (intra-protocol dependencies), such as the cookie field mentioned before, and dependencies between different TCP sessions (inter-protocols dependencies). This first section focuses on the former.

In order to carry on a successful conversation with the client, it is important to correctly handle cookie fields, that is protocol fields of mutating content whose value must recur in both client requests and server answers. Two different situations can be identified:

1. The client sets the cookie in its request, and the value must be reused in the server answer. In this case the emulator must be able to retrieve the value from the incoming request and copy it, or a derived value from it (e.g. the value incremented by 1) in the generated answer.
2. The server sets the cookie in its answer, and the client must reuse the same value for the following requests to be accepted. From the emulator point of view, this does not generate any issue. The server label will contain a valid answer extracted from a training file, using a certain value for the field. The corresponding field in the client requests will be classified as mutating. This leads only to two approximations: the emulator will always use the same value, and it will accept as correct any value used by the client without discarding the wrong ones. These approximations might be exploited by a malicious user to fingerprint a ScriptGen honeypot, but can still be considered as acceptable when dealing with attack tools.

We will further focus here on the first scenario, that is the most challenging since it requires the emulator to identify the cookie fields and establish content dependencies between the client requests and the following answers.

In order to identify these dependencies, it is necessary to correlate the content of client requests with the content of the following server answers. By using many training conversations, we are able to reliably identify dependencies by taking advantage of statistical diversity. Using a reduced number of samples, in fact, makes it difficult to reliably deduce this kind of relationship. For instance, the value of a mutating field in the client request might incidentally match the content of the data payload sent back by the server in a following message of the conversation. Using a large amount of samples drastically reduces the probability of false deductions.

The algorithm presented in this paper to handle content dependencies is composed of two separate steps: *link generation* and *consolidation.*

During link generation, the algorithm takes into consideration each request contained in the training set, enriched by the output of region analysis, and correlates it with all the following server answers contained in the corresponding training conversation. The algorithm takes into consideration all the bytes in the request that are not covered by a *significant* fixed region. A significant fixed region is defined as a region whose content always has a unique match in the client request. Many regions are not big enough to be considered as significant when considered alone. Having a single match inside the client request, significant fixed regions can therefore be used as *markers* to define relative positions inside the client request.

For instance, representing a fixed region as *F("content")* and a mutating region as M(), we consider the following output of the region analysis:

F("LOGIN:")+M()+F("TIME:")+M()+F(":")+M()

that matches, for instance, the following client request:

"LOGIN: bob TIME: 12:13"

The fixed region *F("LOGIN:")* will be considered as significant. But the fixed region *F(":")* will have multiple matches inside the client request and will not be used as marker.

For all those bytes that are not covered by these markers, the algorithm correlates each byte with the server answers using a correlation function. In the most simple case, the correlation function returns 1 if the bytes match, and returns 0 if the bytes differ. For each encountered match, the algorithm tries to maximize the number of consecutive correlated bytes starting from a minimum of two. For instance, we consider these two simple training conversations:

1. R1: "*Hi, my ID is 147 what time is it?*"
   A1: "*Welcome 147, time is 14:05*"
2. R1: "*Hi, my ID is 134 what time is it?*"
   A1: "*Welcome 134, time is 14:18*"

In this case, region analysis will enrich the request generating two significant fixed regions: "Hi, my ID is " (F1) and " what time is it?" (F2). For each training conversation, the link generation algorithm will search for correlations between the remaining bytes of the request and the following answers, producing *links*. A link is a logical object that provides in a dynamic way the content to be put in a certain position of the server answer. Different kinds of links might be introduced in the future. For the time being, when a content match is found in a server answer, the matching content is replaced with the output of a *matching link*. A matching link is defined by the tuple $L = (Rq, S, R, Os, Ot)$

- Rq: The client request the link is referring to
- S: The starting marker
- R: The trailing marker
- Os: An offset with respect to the starting marker
- Ot: An offset with respect to the trailing marker

To better understand the meaning of these characteristics, we can refer back to the previous example. For the first training conversation, the link generation would define two links in the server answer: "Welcome L_1, time is L_2:05". We will have

$$L_1 = (R1, F1, F2, 0, 0)$$

$$L_2 = (R1, F1, F2, 0, -1)$$

Instead, for the second training conversation the resulting server answer will be: "Welcome L_3, time is 14:18". L_3 will be identified by the tuple:

$$L_3 = (R1, F1, F2, 0, 0)$$

From this example it is clear that link generation parses each conversation independently, making optimistic guesses on the dependencies. Link generation therefore generates many guesses on the content dependencies. Some of them (links $L_1$and L_3) might be correct, others (such as link L_2) might be coincidental matches between the request content and the payload of the answer.

The second step of the analysis consists of taking advantage of the statistical variability to consolidate these guesses, filtering out the coincidental matches and taking into consideration only the real content dependencies. This step is therefore called *consolidation*.

The input to this step is a set of proposals for a certain server answer generated by the previous link generation. The algorithm takes into consideration each byte and compares the content of each proposal for that byte. This content can be either a link or the value of the answer in the original training file. The most recurring content is put in the consolidated answer, while the other ones are discarded. All the proposals having a content for that byte differing from the chosen one will not be taken into consideration any more for the remaining bytes. Referring back to the previous example, the output of the consolidation phase for answer A1 will be: "Welcome $L_{1=3}$, time is 14:05".

Figure 3 represents the consolidation behavior in a very pessimistic case. In this case, the number of misleading links is as high as the number of proposals.

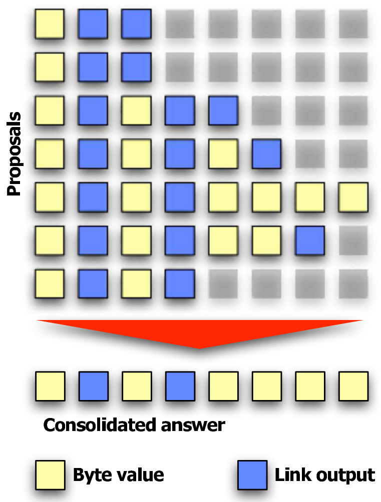

Fig. 3. Consolidation

The algorithm is such that the consolidated answer will always be equal at least to one of the proposals. Also, increasing the number of training samples will increase the number of proposals, therefore increasing the robustness to misleading links. The number of valid proposals at the end of the algorithm can be considered as the confidence level for the validity of the consolidated answer.

During emulation, the link information is used to transform the referenced content of the requests and provide the content for the server answers. Using the significant fixed regions as markers, and offsets to specify relative positions, it is possible to correctly retrieve variable length values.

What has been stated herein with reference to simple equality relations can be extended to other types of relations, such as incrementing counters, by simply defining different types of links.

4.2 Inter-protocols Dependencies

In order to handle dependencies among, for instance, different TCP sessions, it is necessary to re-define the notion of state in the ScriptGen model. In [10], we bound the emulation state to a single TCP session. Each TCP connection was associated with a different state, and any event or side-effect outside that binding was not taken into consideration. In order to allow dependency handling among different sessions, the definition of state must be widened. For this reason utilize the concept of conversation: a conversation is defined as the whole amount of data that has been exchanged between a single attacking source and the attacked server in the training data. The attacking source is identified by its IP address and a timestamp, in order to take into consideration dynamic IP allocation. The same IP address, when coming back after a period of time greater than 24 hours, will be considered as a different source. A conversation therefore consists of all

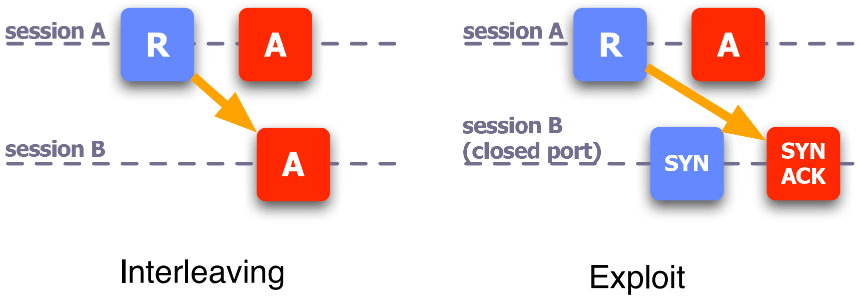

Fig. 4. Inter-protocols dependencies

the activities performed by an attacker towards the vulnerable host, and might be composed of several TCP sessions and several exchanges of UDP messages.

Considering each conversation as the domain for the inter-protocols analysis, we identified two different types of session dependency:

- **Session interleaving:** some protocols spread the interaction between the client and the server on multiple connections to different ports. For instance FTP separates the control connection from the data connection. Messages seen on one session initiate activities on the other one: an FTP *recv* command on the control connection will cause traffic to be generated on the data connection.
- **Exploits:** when a vulnerable service is attacked by a malicious client, the client might succeed in exploiting a buffer overflow attack on the victim over a certain port and open a previously closed port. We will see that this kind of dependency is extremely important: section 5 will show how the incorrect handling of this kind of dependencies can influence the conversation with the client.

From a practical point of view, the two dependencies are illustrated in Figure 4. It is interesting to notice how, in both cases, a client request in a given TCP session modifies the server state triggering events that are outside the scope of the connection itself. In the case of session interleaving, the request triggers a server message on a different connection; in the case of the exploit dependency, the request opens a previously closed port. It is important to understand that these are just two examples of external state modifications that can be caused by client requests. Referring to the cause of the exploit, another common behavior observed in buffer overflow attacks consists of actively fetching malware from an external location. This specific case is extremely interesting and is subject of ongoing research.

A session interleaving dependency is triggered by the following conditions: i) more than one session is open (e.g. A and B) ii) after a client request in session A, and eventually an answer from the server for that session, the first encountered packet is an answer from the server in session B. This means in fact that the request on session A has influenced the state of session B, triggering a message from the server.

Knowing the list of commonly open ports for the emulated server, the exploit session dependency is triggered if the following conditions hold: i) session A is bound to a known open port (e.g. port 139 on a Windows 2000 host) ii) session B is targeting a closed port (e.g. port 4444 on the same Windows 2000 host) iii) an outgoing TCP SYN/ACK is sent by the server from session B after having received a request in session A. The TCP SYN/ACK means in fact that the port, previously known to be closed, is now open.

Once dependencies are identified, ScriptGen emulates causality through a simple signalling method between different state machines. During this emulation, the emulator allocates different broadcast buses for signals, one for each source, as shown in Figure 5. When the emulator reaches a state that the dependency analysis has identified as the trigger for a session dependency, a signal is sent on the bus for that source. The other state machines will be notified of the signal and will eventually react to it. With respect to session interleaving, the given request will generate a signal that will trigger a transition on the state machine associated with session B. The transition to the new state will therefore generate a new server message, that will be sent back to the client emulating the correct behavior. In the case of the exploit session dependency, the state machine associated to the closed port will start accepting connections on that port only after having received the signal corresponding to the client request. This will allow the correct emulation of the expected server behavior.

4.3 Proxying and Incremental Refinement

One of the main contributions of this paper consists in being able to react to new activities, triggering new alerts and being able to refine the existing state machine. To do so, we refine the existing region analysis algorithm in order to support incremental refinements. Then we introduce a novel proxying algorithm that allows ScriptGen to rely on a real host to build its training set.

In [10] we started from a too specific state machine and then we used Region Analysis to move to a higher level of abstraction, aggregating the existing states and generating transitions based on regions. There was no clear separation between raw data, not parsed yet because of the lack of enough samples to generate macroclusters, and data whose semantics had already been rebuilt. For this reason, in the new incremental algorithm that we propose, we split the analysis into two distinct phases described in Sections 4.3 and 4.3. Section 4.3 will describe more in depth the new proxying algorithm.

Update phase. Given an existing (eventually empty) state machine, each incoming flow is *attached* to it. Starting from the root, we use the sequence of requests in the incoming flow to traverse the existing edges of the state machine, choosing the future state according to a *matching function* defined later.

While traversing the state machine, the server labels are updated on the various nodes with the eventually empty server answers found in the training conversation. If for a certain state no outgoing edge matches the client request, the remaining training conversation is attached to the state's *bucket*. The bucket is

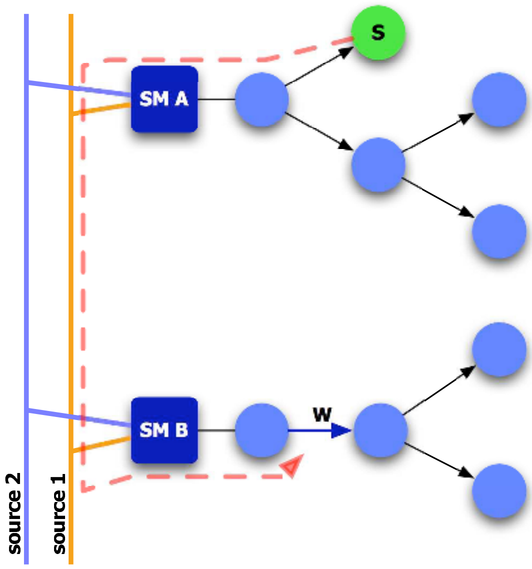

Fig. 5. Signalling

simply a container for new raw data that will be used in the following phase to perform the refinement.

The notion of bucket allows a distinction between new unprocessed data and the already consolidated transitions, solving the issue mentioned before. Also, the update phase is applied indifferently during the training phase and during the emulation. The only difference between the two cases is when encountering an unmatched request to be put in the bucket. While in the training phase the rest of the conversation is known, during emulation since the future state is unknown it is not possible to continue the dialog with the source. The proxying algorithm will allow ScriptGen to rely on a real host to continue the conversation, using the proxied conversation to build training samples to perform the refinement.

One of the most critical aspects in the update phase consists in the choice of the matching function. At first, our choice had been to try to be as robust as possible to new activities or to imprecise state machines generated from an insufficient number of samples. So we accepted imperfect matches, that is requests whose content did not completely match with the output of region analysis. But this leads to two major drawbacks. First of all, tolerating imperfect matches between the incoming request and the known transitions might lead to the choice of a wrong transition generating a completely wrong answer, corrupting the conversation. Also, distinguishing imperfect matches from new activities becomes impossible. For this reason we moved to a much more conservative choice, consisting of requiring an exact match of all the fixed regions, transforming the output of the region analysis in a regular expression. If multiple transitions match the same incoming request, the most refined one is chosen: that is, the transition

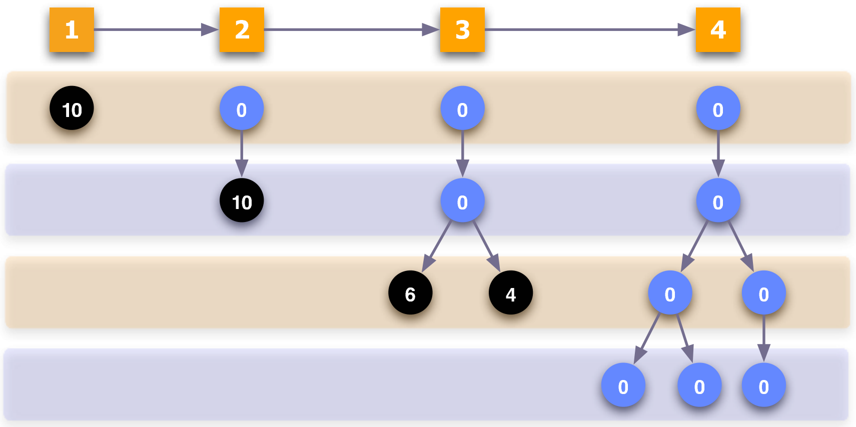

Fig. 6. Iterative refinement

having the maximum number of matching fixed bytes. This is necessary to correctly handle microclustering, in which recurring values of the mutating parts are transformed into fixed regions. The previous policy will give preference to the refined microcluster, having more fixed bytes, rather than the generic transition.

Refinement phase. During the refinement phase, ScriptGen inspects state buckets in order to search for possible refinements. If a bucket is not empty, ScriptGen runs the region analysis algorithm over the unmatched requests present in the bucket. If the number of samples is enough to generate macroclusters of sufficient size, one or more transitions are generated refining the existing ones. After having generated all the possible edges, the update phase will be repeated on the refined state machine.

Figure 6 shows an simple example of iterative refinement. For each state in the diagram, the label corresponds to the number of training conversations in the bucket. A training file consisting of 10 training flows is used to update an empty state machine. Since the state machine is empty, none of the initial client requests contained in the samples will match an existing transition. Thus all the samples will be put in the initial bucket, as shown in figure at step 1. The refinement phase will then pick the training samples contained in the bucket, and apply the region analysis algorithm to the samples. Region analysis generates a different transition for each set of sample client requests believed to have a different semantic meaning. In this first step, a single transition is generated. After the refinement phase, the update phase is then triggered and the training flows are matched with the newly created transition. Since the state machine is still incomplete, the training samples do not find a match in the following state, and are thus stored in the corresponding bucket for the next refinement iteration (step 2). The process repeats until the refinement phase is not able to generate other transitions: this happens at step 4, in which the sample flows do not contain any further interaction between the attacking source and the server (client closes connection after having sent 3 requests to the server). The state machine is then complete.

Proxying algorithm. The previous sections showed how, through the concept of buckets and the separation between update and refinement phase, we are able to handle in a uniform way the training performed with real sample conversations and the interaction with real sources during emulation. As mentioned before, there is still a major difference between these two cases. While during training the whole conversation is already known, this is not true during emulation. In the second case in fact, when receiving a request for which no matching transition exists, we do not have a way to make the client continue its conversation with the server. However we need the continuation of the conversation to train ScriptGen to refine the state machine.

To acquire this information, when encountering unmatched requests we tunnel the client conversation to another host able to handle it, such as a high interaction honeypot. Focusing for conciseness only on the TCP case, the proxying algorithm works as follows:

- Every source initiates a certain number of connections with the ScriptGen honeypot. The emulator keeps track of all of them, buffering all the received requests.
- When receiving an unmatched request from host H_i at time t_u, the emulator triggers the proxy initialization. At time t_u the source will have a certain number of open connections ${C^O}_1 ... {C^O}_p$.
- The emulator will search for an available host in its pool of servers and will allocate it to the source. It will then initialize it, replaying all the buffered requests received from host H_i before time t_u. If the initialization is successful, it will end up with p open connections between the emulator and the allocated server ${P^O}_1 ... {P^O}_p$. For each of them, the ScriptGen emulator will setup an application-level association $C_i^O \leftrightarrow P_i^O$. Every message received from one of the two ends,will be forwarded to the other end. The message content will be stored, building the training sequence to be used in the refinement.
- Every incoming connection from the same source after the proxy initialization at time t_u will directly generate an application-level association with the allocated server, and the application level payloads will be used to update and eventually refine the existing state machines.
- After a certain time of inactivity, the source will expire and the allocated server will be freed. The emulator will use the retrieved conversations to run the update and refinement phase.

This algorithm allows the emulator to promptly react in a completely automated way to requests that the state machine is not able to parse. Through proxying, the emulator is able to build its own training set and use this training set to update its protocol knowledge. Assuming that the state machine represents the whole set of known activities going on in a certain network, this algorithm offers valuable properties. It allows us to go much further than just sending alerts for new activities. We can automatically build a training set and use it to infer semantics. This output can therefore be used to automatically generate signatures for the newly observed activities.

5 Testing

In order to retrieve significant information about the real quality of the emulation, we have run a set of experiments to evaluate ScriptGen's behavior when dealing with a real client.

To perform our tests, we took advantage of the flexibility of the controlled virtual network presented in [15]. This network provides a secure environment to run completely automated attack scenarios. Thanks to a huge database of attack scripts and virtual machine configurations, this setup allows an extreme flexibility and can be considered as the ideal testbed for our emulators. A discussion of the exhaustive test of ScriptGen behavior using all the available attack scripts is left as a future work. For the scope of this paper we want to provide an in-depth analysis of ScriptGen's behavior in a single interesting case.

Among all the used exploits, for the sake of conciseness we chose to focus in this paper on a specific vulnerability exploited by a Metasploit Project[1] module. The vulnerability is the Microsoft Windows LSASS Remote Overflow [16] (used by the Sasser worm). This vulnerability exploits a validation failure on the LSARPC named pipe and, through a specially crafted packet, allows an attacker to execute arbitrary code on the attacked host. In the specific implementation of the exploit at our disposal, the attack consists of 41 requests and 40 answers on a single TCP connection targeting port 139. This is therefore a clear example of "long" activity whose analysis would greatly benefit from the increased verbosity offered by ScriptGen. We chose this exploit for several reasons:

- This exploit opens a shell on port 4444 on the attacked host. Also, the exploit checks if the port is open before starting the attack: if the port is already open, it does not proceed further. This is a clear case in which session dependencies are needed in order to emulate the correct behavior. If the port is always open, the honeypot will never observe the attack on port 139 and will instead observe only a connection attempt on port 4444. If the port is always closed, the attack will always fail. Using dependencies, we are able to send a signal only when the last state of the attack path is reached. The state machine for port 4444 waits for that signal before opening the port.
- This exploit targets the NetBIOS Session Service. Its protocol semantic is rather complex, and offers many examples of content dependencies. If the content dependencies are not handled correctly, the client aborts the connection after the second answer from the honeypot as shown in [10]. This shows the importance of this kind of dependencies, that greatly influences the length of the conversation.

[1] `http://www.metasploit.org`

Table 1. Attack output

```
[*] Starting Bind Handler.
[*] Detected a Windows 2000 target ()
[*] Sending 32 DCE request fragments...
[*] Sending the final DCE fragment
[*] Exiting Bind Handler.
```

Due to the complexity of the NetBIOS Session Service protocol, this case is representative of the upper bounds of the complexity that might be faced in protocol emulation.

5.1 Emulation Quality

In order to assess the emulation quality of the produced emulators, we have used our virtual network infrastructure to generate a training sample consisting of 100 samples of the attack against a real Windows 2000 target. After every run of the attack, the target was reverted to its initial state and the experiment was repeated. In order to maximize variability (with special attention to timestamps) the various runs of the attack have been spaced in time by an interval of 5 minutes. All the interaction was collected in a tcpdump file, and was then used to automatically train ScriptGen and produce two state machines: one for TCP port 139, the other for TCP port 4444.

Analysis of the state machine. Before analyzing the behavior of the emulator in a network test, it might be interesting to inspect the content of the state machine generated by ScriptGen. As expected, the state machine is a sequence of 42 states. There is only one leaf, and therefore a single path: all the states, except for the last one, have exactly one child. Thanks to the consolidation algorithm, there is always a unique server answer bound to each state. Also, ScriptGen has correctly identified session dependencies, associating a signal to the last state of the state machine for port 139. When that state is reached, the signal triggers a transition for port 4444, opening it.

Looking at these client requests more in depth, we can see that after an initial session request (whose content is always the same) ScriptGen generates more complex sequences of fixed and mutating regions. More specifically, we can notice that most requests share two mutating regions of size 2 respectively at bytes 30-31 and 34-35. Looking at the protocol specification, these fields correspond to the *processID* and the *multiplexID* of the SMB header. These two fields are chosen by the client and must be repeated in the following answers given by the server. Inspecting the server answers, we can indeed note that content dependency handling has correctly generated the correct links to handle those dependencies.

Experimental evaluation. We deployed a ScriptGen based host in our testing virtual network, and we ran the attack script against the honeypot.

The emulator handled perfectly all the content and session dependencies, traversing the whole path of the state machine.

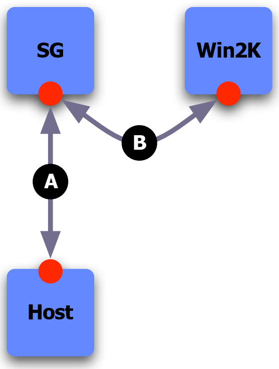

Fig. 7. Test scenario

The output of the attack script is indistinguishable from the one of a successful attack against a real host (table 1), proving the quality of the emulation.

It is important to notice that this is a complete validation of the region analysis approach. It started from a rich training set, without *any* kind of additional information, and successfully handled a conversation with same structure, but with partially different content (different process IDs, different timestamps).

5.2 Reaction to Unknown Activities

In this section we want to experiment with ScriptGen's capability to react to new activities and to automatically refine existing state machines retrieving training information through proxying. We know from the previous experimental results that, given a sufficient number of training samples, ScriptGen is able to carry on a complete conversation with a client. Here we want to inspect the ability of the emulator to produce its own training set to refine the state machine, and its ability to reliably identify new activities.

The experiments have been run in a very simple test scenario, shown in Figure 7. The attack is run against a ScriptGen honeypot, that is allowed to rely on a Windows 2000 virtual machine using the proxying algorithm described in Section 4.3.

Learning. The first aspect that we want to inspect is ScriptGen's ability to reliably refine the state machine. Given a certain activity, initially unknown, ScriptGen should take advantage of proxying to build its own training set and refine the state machine. After the refinement, ScriptGen must be able to correctly handle the activity, without contacting the proxy any more.

For refinement to be reliable, the training set must be diverse enough to allow a correct inference of its semantics. If the training set is not diverse enough, coincidental matches of mutable values may lead to wrong deductions on their nature. If this happens, following instances of the same activity may not match the generated transitions. This may generate erroneous alerts for new activities (false positives). Therefore the *refinement condition*, that triggers the refinement

Table 2. Experimental results

N	# false alerts	# critical requests
3	3	3
5	3	3
10	0	0
20	2	1
50	0	0

of the state machine when the samples are considered to be diverse enough, becomes critical.

In this first scenario, the ScriptGen honeypot has been deployed with an empty state machine for port 139. We used different refinement conditions, and then ran 100 times the same activity (the same exploit used to study the emulation quality). Since the different runs of the activities are spaced in time by approximately 10 seconds, we considered as a good measure of diversity (also from a temporal point of view) different thresholds on the number of available training samples. When the number of samples retrieved through proxying is equal to N, ScriptGen refines the state machine and then continues emulation. Running the experiment in the same conditions using different values of N and then inspecting the resulting state machines will give a measure of the sensitivity of ScriptGen to the lack of diversity of the samples.

Table 2 shows the relevant characteristics of the generated state machines. If the training sample is not diverse enough, ScriptGen will generate false alerts. That is, after the first refinement of the state machine the emulator will not be able to correctly match successive requests, interpreting them as a new activity. The number of false alerts is therefore connected to the quality of the training samples. We expect a decreasing number of false alerts when increasing the value of N. After each alert ScriptGen will again use proxying to collect a training sample, and refine the existing state machine with one or more functional paths.

It is also important to understand whether or not these unmatched requests are observed at a critical point of the state machine: there might be a particularly complex request for which region analysis is not able to generate a reliable transition. For this reason we count the number of nodes that triggered an unmatching transition, which therefore corresponds to the number of critical requests in the protocol state machine.

Figure 8 gives a visual explanation of the two concepts previously explained. While the first case can be considered as a symptom of a general lack of variability, the second case is probably due to a more specific problem in a given request. Referring to Table 2, we can map the first case to low values of N (N=3,N=5) while we can find an example of the second case for N=20.

A first striking result is the fact that in all cases, ScriptGen has been able to generate a complete state machine at the first time the refinement condition has been triggered. But since some of the protocol variability is linked to time-dependent fields (timestamps, and as we will see process IDs), the produced refinements incorporate false deductions that lead to unmatched requests after some time.

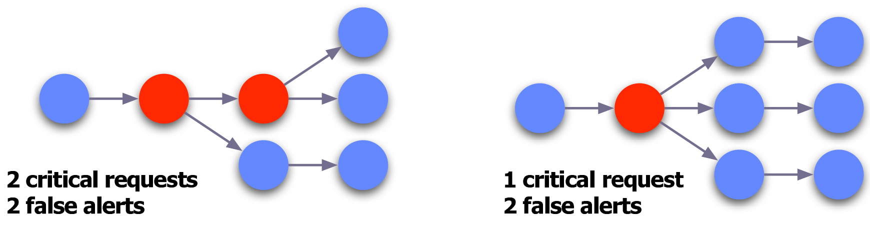

Fig. 8. Different refinement cases

When N is equal to 20 we can experience a rather strange artifact. For the second request in the conversation, 2 false alerts are generated. Inspecting the transitions, we can notice that the artifact is due to the last byte of the process ID: it is considered as a fixed region. Since this value is stored following the little-endian convention in the NetBIOS protocol, it actually corresponds to the high part of the process identifier of the attacking client. Since process identifiers are often assigned sequentially, and since the attacking host was not reverted to initial conditions during the experiment, this is not surprising. It is a clear case in which the lack of variability of the samples leads ScriptGen to make wrong assumptions. Only with N equal to 50 we have enough variability to correctly classify the byte as part of a mutating region. In the case N equal to 10, the problem was not raised only by a fortunate sequence of 100 process IDs having all the same high part.

It is important to understand that some of the lack of diversity that we encountered in this experiment is due to specific artifacts of the chosen scenario. We are running every attack instance from the same host in an iterative way. This means that the process ID in the SMB header, usually appearing as a random field, here has incremental values. In a real attack scenario in which the honeypot is contacted by many different hosts, the diversity of this field would be greatly increased and so probably the number of samples required to generate reliable refinements would decrease. However, ScriptGen has been able to correctly generate a reliable refinement of an initially empty state machine using a training set of 50 conversations automatically generated through proxying. This validates the ability of ScriptGen to learn new activities.

Triggering new activities. After having shown how ScriptGen is able to produce refinements to the state machine, we need to investigate its capability to reliably detect new activities. The previous section investigated the ability to generate reliable refinements, and therefore ScriptGen's ability of not generating false positives. Here we want to investigate ScriptGen's ability to reliably detect new activities, and therefore false negatives. To do so, we deployed a new ScriptGen honeypot, in the same configuration as shown in Figure 7, but already instructed with the state machine generated in the previous example for a value of N equal to 50. Then we run against this honeypot a new activity on the same port, namely the Microsoft PnP MS05-039 Overflow [17]. We followed the same pattern used in the previous experiment: 100 runs spaced in time choosing a triggering threshold equal to 50. The attack followed the first path for the first 5 requests, and only at that point triggered an unmatched request. Using just 50 samples of interaction, ScriptGen has been able to correctly refine the state machine adding a single path to the existing one. The refined state machine correctly handled all the 50 successive runs of the attack.

This is an extremely important result. First of all, it shows how ScriptGen-based honeypots are able to reliably identify new activities. Also, since the two activities are identifiable only after the exchange of 5 couples of request/answer, it validates the importance and the power of the ScriptGen approach with respect to the current state of the art in honeypot technology.

6 Conclusion

In this paper, we have shown the feasibility of using a completely protocol-unaware approach to build scripts to emulate the behavior of servers under attack. As opposed to the approach considered by the authors of the RolePlayer system, we have deliberately refused to take advantage of any heuristic to recognize important fields in the arguments received from the clients or sent by the servers. Instead, by using several instances of the same attack, we can automatically retrieve the fields which have some importance from a semantic point of view and are important to let the conversation between the client and server continue. More specifically, we have shown that two distinct types of dependency are important to take into account. We have named them, respectively, intra-protocol and inter-protocol dependencies. We have proposed new algorithms to handle them efficiently. We have also shown that this newly created mechanism can be further enhanced to create new scripts online as new attacks are appearing by, temporarily, proxying the requests and responses between the attackers and a real server. Experimental results obtained with our approach are very good and demonstrate the potential inherent in the large-scale deployment of honeynets such as our Leurre.com project [3,4,5,6,7,8]. The ScriptGen approach would in fact allow us to collect an even richer data set than the one we have accumulated so far.

References

1. Spitzner, L.: Honeypots: Tracking Hackers. Addison-Welsey, Boston (2002)
2. Provos, N.: A virtual honeypot framework. In: Proceedings of the 12th USENIX Security Symposium. (2004) 1–14
3. Dacier, M., Pouget, F., Debar, H.: Attack processes found on the internet. In: NATO Symposium IST-041/RSY-013, Toulouse, France (2004)
4. Dacier, M., Pouget, F., Debar, H.: Honeypots, a practical mean to validate malicious fault assumptions. In: Proceedings of the 10th Pacific Ream Dependable Computing Conference (PRDC04), Tahiti (2004)
5. Dacier, M., Pouget, F., Debar, H.: Honeypot-based forensics. In: Proceedings of AusCERT Asia Pacific Information Technology Security Conference 2004, Brisbane, Australia (2004)

6. Dacier, M., Pouget, F., Debar, H.: Towards a better understanding of internet threats to enhance survivability. In: Proceedings of the International Infrastructure Survivability Workshop 2004 (IISW'04), Lisbonne, Portugal (2004)
7. Dacier, M., Pouget, F., Debar, H.: Leurre.com: On the advantages of deploying a large scale distributed honeypot platform. In: Proceedings of the E-Crime and Computer Conference 2005 (ECCE'05), Monaco (2005)
8. Dacier, M., Pouget, F., Debar, H.: Honeynets: foundations for the development of early warning information systems. In Kowalik, J., Gorski, J., Sachenko, A., eds.: Proceedings of the Cyberspace Security and Defense: Research Issues. (2005)
9. CERT: Cert advisory ca-2003-20 w32/blaster worm (2003)
10. Leita, C., Mermoud, K., Dacier, M.: Scriptgen: an automated script generation tool for honeyd. In: Proceedings of the 21st Annual Computer Security Applications Conference. (2005)
11. Needleman, S., Wunsch, C.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol. 48(3):443-53 (1970)
12. Cui, W., Vern, P., Weaver, N., Katz, R.H.: Protocol-independent adaptive replay of application dialog. In: The 13th Annual Network and Distributed System Security Symposium (NDSS). (2006)
13. Freiling, F.C., Holz, T., Wicherski, G.: Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In: Lecture Notes in Computer Science, Springer-Verlag GmbH (2005) 319–335
14. The Honeynet Project: Know your enemy: Tracking botnets. Know Your Enemy Whitepapers (2005)
15. Massicotte, F., Couture, M., De Montigny-Leboeuf, A.: Using a vmware network infrastructure to collect traffic traces for intrusion detection evaluation. In: Proceedings of the 21st Annual Computer Security Applications Conference. (2005)
16. OSVDB: Microsoft windows lsass remote overflow, http://www.osvdb.org/5248 (2006)
17. OSVDB: Microsoft pnp ms05-039 overflow, http://www.osvdb.org/18605 (2005)