ScriptGen: an automated script generation tool for honeyd

Corrado Leita, Ken Mermoud, Marc Dacier
Institut Eurecom
Sophia Antipolis, France
{leita,mermoud,dacier}@eurecom.fr

Abstract

Honeyd [14] is a popular tool developed by Niels Provos that offers a simple way to emulate services offered by several machines on a single PC. It is a so called low interaction honeypot. Responses to incoming requests are generated thanks to ad-hoc scripts that need to be written by hand. As a result, few scripts exist, especially for services handling proprietary protocols. In this paper, we propose a method to alleviate these problems by automatically generating new scripts. We explain the method and describe its limitations. We analyze the quality of the generated scripts thanks to two different methods. On the one hand, we have launched known attacks against a machine running our scripts; on the other hand, we have deployed that machine on the Internet, next to a high interaction honeypot during two months. For those attackers that have targeted both machines, we can verify if our scripts have, or not, been able to fool them. We also discuss the various tuning parameters of the algorithm that can be set to either increase the quality of the script or, at the contrary, to reduce its complexity.

1 Introduction

Honeypots have recently received a lot of attention in the research community. They can be used for several purposes, ranging from the capture of zero-day attacks to the long term gathering of data. Honeyd [14] is one of the simplest and most popular solutions. It has been extensively used, for instance, in the Leurre'com project where dozens of similar platforms have been deployed in the world [4, 5, 6, 7, 8, 9]. Unfortunately, Honeyd is based on specific scripts that are required to emulate the various services listening to remote requests. Writing these scripts is a tedious and sometimes impossible task, especially for proprietary protocols for which no documentation exists. As a result, there are not so many existing honeyd scripts. This makes the fingerprinting of honeyd platforms rather simple and they do not provide as much information as they could. Had they more services offered, we would learn more about the attackers. Our approach aims at generating these scripts automatically, without having to know anything neither about the daemon implementing the service, nor about the protocol.

In the general case, this is probably impossible to do but we have a much more modest goal. We want to provide good answers to requests sent to a honeypot by attack tools. This dramatically simplifies the problem in the sense that the requests we need to answer to are generated by deterministic automata, the exploits. They represent a very limited subset of the total possible input space in terms of protocol data units. They also typically exercise a very limited number of execution paths in the execution tree of the services we want to emulate.

Keeping this very specific application domain in mind, we have developed a three steps approach to generate our scripts:

1. We put a real machine on the Internet, as a honeypot, and we record all traffic to and from that machine in a tcpdump file. If the machine gets compromised, we stop the experiment and clean it (i.e. reinstall it).

2. We analyze thanks to various techniques the sequences of message exchanges between clients and servers. We derive from this analysis a state machine that represents the observed requests and replies. We have one state machine per listening port.

3. We derive from that state machine a honeyd script that is able to recognize incoming packets and provide a suitable answer.

Of course, as we will see in the paper, such an approach can only offer an approximation of the real services. However, for those interested in studying the attacks thanks to honeypots, the more packets they can exchange with the attackers, the more information they have at their disposal to identify the attack. Therefore, for that specific application domain, we believe that the ability to automatically generate scripts for all classical services that are targeted by the attackers constitutes a major improvement to existing low interaction honeypots such as honeyd.

To present our method, the paper is structured as follows. Section 2 presents the method as well as the various algorithms designed and implemented to generate the scripts. Section 3 offers a discussion of the expected quality of the simulation with respect to the price we are ready to pay in terms of complexity of the script. Section 4 provides the results of experiments run during two months to validate the method. Finally, Section 5 concludes the paper.

2 ScriptGen

2.1 Overview

ScriptGen can be described by four functional modules, represented in figure 1:

- **Message Sequence factory**. This module is responsible for extracting messages exchanged between a client and a server from the tcpdump file. A notion of sequence can be given for different protocols (e.g. UDP, or IP-only based protocols); here we focus on TCP-based protocols. This module reconstructs TCP streams, correctly handling retransmissions and reordering.
- **State Machine Builder.** These messages are used as building blocks to build a state machine. At this point, it can lead to the generation of a very large, redundant and highly inefficient state machine. It is usually required to control the complexity growth by defining thresholds that limit the number of outgoing edges of each state. In such case, clearly, the execution of the script may reach a state where it may not be able to reproduce perfectly the behavior of the real server.
- **State Machine Simplifier.** This is the core of ScriptGen. This module is responsible for analyzing the "raw" state machine and for introducing some sort of semantics. This is achieved thanks to two distinct algorithms interacting with each other. The first one, the PI algorithm, is taken from [3] and described in Subsection 2.4.1. The second one is a novel contribution of this paper. We call it the *Region Analysis algorithm* and we explain it in Subsection 2.4.2. As a result, we obtain a much simpler state machine where incoming messages are not recognized as simple sequences of bytes but instead as sequences of typed regions that must fulfill certain properties.
- **Script Generator.** This last module is responsible for creating a honeyd-compatible script from the simplified state machine.

2.2 Message Sequence Factory

A message sequence is an ordered list of messages. A message is seen as a piece of the interaction between the client and the server. More formally, a message is defined as the longest consecutive set of bytes going in the same direction (e.g. client to server or vice versa). A TCP session can be decomposed into a list of messages. That list represents the observed dialog between the client and the server. The length of a sequence is defined as the number of messages sent either by the client or the server.

Many solutions have been deployed to efficiently reassemble TCP packets. For instance, responders like iSink [15] are built as a Click kernel module [11] in order to use a fast flow reassembler, while another possible solution is used in [13] and consists in directly using an existent IDS. We did not use any of those solutions and decided to implement our own to easily customize it to our needs.

2.2.1 Rebuilding TCP sequences

One of the first design problems is to define an optimized algorithm to parse the tcpdump file and rebuild the conversation between clients and server, correctly handling duplicated and out-of-order packets. We have to take into account the fact that the client may not respect the classical TCP state machine in order, for instance, to implement IDS evasion techniques. Therefore, ScriptGen rebuilds TCP flows on the basis of the following assumptions:

- A packet is interesting only if it's carrying a payload. Every pure-ACK packet, for instance, is ignored by ScriptGen. Retransmissions are also ignored. Only packets containing a TCP payload are considered.
- A TCP session starts with the first SYN packet. For every new SYN packet, ScriptGen allocates all the data structures necessary to handle the new flow.
- A TCP session ends with the first FIN/RST packet encountered. When one of the two communicating parties decides to end the conversation, the conversation is considered finished.
- The TCP sequence number is used as an index of an array where we store the payload. This enables ScriptGen to handle out of order packets as well as retransmissions. In that last case, the very first packet is accepted. Following ones are discarded.

We acknowledge the fact that these assumptions may cause trouble in the general case. For instance, packet checksum is not computed and therefore transmission errors are not detected; also, incorrect sequence numbers in the IP header may lead to the allocation of huge amounts of memory.

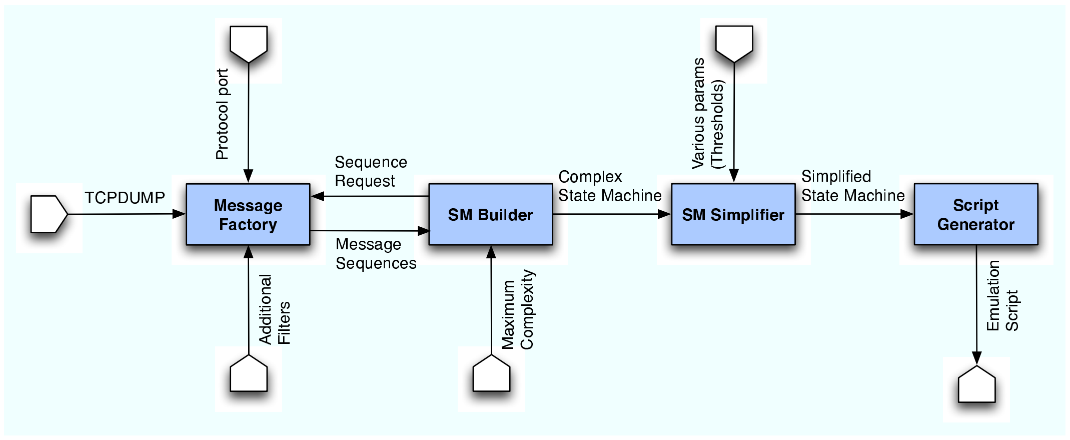

Figure 1. General structure

Nevertheless, based on our experience with several months of data, they appear to be satisfactory for our specific needs.

2.3 State Machine Builder

The State Machine Builder creates a complex State Machine from the message sequences generated by the Message Sequence Factory. The state machine is created in an iterative way, by adding all observed message sequences one by one.

The State Machine is composed of edges and states. For a given state, the outgoing edges represent the possible transitions towards the next future state. Each edge is labeled with a message representing the client request which will trigger that transition, while each state is labeled with a message representing the answer that the server will send back to the client when entering it. Every edge label has also a weight. The weight represents the frequency with which samples have traversed that specific transition.

It is worth pointing out that if the answer provided by a server is a function not only of the past exchanges but also of some external factor, such as, for instance, the time of the day, then a given state could have more than one label. In other words, a given exchange of messages may lead to two, or more, different answers from the server. Therefore, server labels are maintained in arrays. The frequency of each label is also kept. The most frequent one is the default choice when the script has to generate a reply.

In order to avoid overly complex State Machines, two thresholds are defined: the maximum fan-out of one state and the maximum number of states. The maximum fan-out is the maximum allowed number of outgoing edges from one state.

Figure 2 shows a simple example of State Machine. The first state is labeled with the server message S0. Of course,

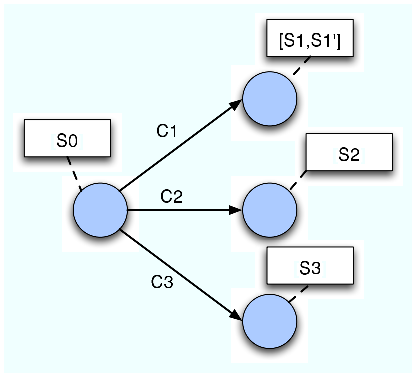

Figure 2. Simple example of State Machine

if the protocol is not sending a welcome message when the connection is opened, then this message will be empty. There are three outgoing edges representing three different client messages: C1, C2 and C3. Each of the edges is connected to a state having a label containing one or more server messages.

2.4 State Machine Simplifier

The previous algorithm creates a basic state machine without any notion of protocol semantics. This state machine is specific to the sample tcpdump file from which it has been generated and lacks generality: it is not able to handle anything that has not already been seen. In the next steps, we simplify and generalize this state machine.

A simple Instant Messaging protocol, whose sample messages are shown in table 1, is given in order to better understand the problem. Once connected to a server, we observe the client sending 12 messages. Each of them is

```
1. GET MSG FROM <bob>
2. SEND MSG TO <john> DATA: "Hi!"
3. GET MSG FROM <marty>
4. SEND MSG TO <ken> DATA: "I'm coming"
5. GET MSG FROM <corrado>
6. GET MSG FROM <liz>
7. SEND MSG TO <bill> DATA: "Be patient"
8. GET MSG FROM <robert>
9. SEND MSG TO <diego> DATA: "Sorry"
10.SEND MSG TO <miki> DATA: "It's beautiful"
11.SEND MSG TO <dan> DATA: "See you"
12.GET MSG FROM <rei>
```

Table 1. Simple IM protocol

represented in the initial state machine by an edge with a specific label, coming out of the initial state. In this case, the number of outgoing edges from the root node is proportional to the number of usernames and messages sent in the system, which is certainly not good. The State Machine is then too specific, and will not be able for instance to handle a new user that was not present in the sample file. There is a need for abstraction in order to generate from this list of transition labels some more generic patterns.

This problem is due to the fact that we ignore the semantics of the messages. We should, in fact, have only two edges leaving the initial state. One would be labeled "GET MSG FROM <username>" and the other one "SEND MSG TO <username> DATA". As we aim at deriving scripts automatically, without trying to understand the protocol, we need to find a technique that is able to retrieve that notion of semantics for us. This is where the simplification module comes into play. It is based on two distinct notions, macroclustering and microclustering, explained here below.

2.4.1 The basics

In the macroclustering phase, we run a breadth-first visit of the initial state machine and gradually collapse together states whose edges are considered to be semantically similar. Finding "semantically similar messages" implies that we are, somehow, able to infer the semantics of the exchanged messages. This is a problem partially addressed by the Protocol Informatics Project (PI) [3]. They have proposed a clever approach to reverse engineer protocols thanks to novel pattern extraction algorithms initially developed for processing DNA sequences and proteins.

PI is supposed to facilitate manual analysis of protocols. We have used it slightly differently to automatically recognize semantically equivalent messages and, from there, simplify the state machine as explained before.

PI offers a fast algorithm to perform multiple alignment on a set of protocol samples. Applied to the outgoing edges of each node, PI is able to identify the major classes of messages (distinguishing in the example in table 1 GET messages from SEND messages) and align messages of each class using simple heuristics. The result of the PI alignment for the GET cluster is shown in table 2. ScriptGen uses PI output as a building block inside a more complex algorithm called Region Analysis.

2.4.2 Region Analysis

Figure 3 shows the relationship between PI and the whole Region Analysis process. PI aligns the sequences and produces a first clustering proposal (macroclustering). Then, we have defined a new algorithm called Region Analysis that takes advantage of PI output to produce what we call microclusters.

Looking at the aligned sequences produced by PI on a byte per byte basis (see table 2), we can compute for each aligned byte:

- its most frequent type of data (binary, ASCII, zero-value, ...)
- its most frequent value
- the mutation rate (that is, the variability) of the values
- the presence of gaps in that byte (we have seen samples where that byte was not defined).

On this basis a region is defined as a sequence of bytes which i) have the same type, ii) have similar mutation rates, iii) contain the same kind of data and iv) have, or not, gaps. A region can be seen as a piece of the message which has some homogeneous characteristics and, therefore carries probably the same kind of semantic information (e.g. a variable, an atomic command, white spaces, etc..)

Macroclustering builds clusters using a definition of distance which simply counts the amount of different bytes between two aligned sequences. However, sometimes a single bit difference, e.g. in a bitmask, can be something important to identify. Therefore, to complement that first approach, microclustering computes another distance thanks to the concept of region-wide mutation rate, that is the variability of the value assumed by the region for each sequence. Focusing on each region, microclustering assumes that if some values are coming frequently, they probably carry with them some sort of semantic information. In the example in figure 4, we see that macroclustering cannot make any distinction between an HTTP GET which is retrieving an image file and one that is retrieving an HTML file. Indeed, the distance between those two sequences is not significant enough to put them into different clusters. However, when looking at each region, microclustering searches for frequent values and creates new microclusters using them. Microclustering introduces an interesting property in the Region Analysis simplification algorithm: frequently used functional parts

0012	x47	x45	x54		x4d	x53	x47		x46	x52	x4f	x4d		x3c	x77	x71	x72	x61	x66	___	___	___	x3e
0001	x47	x45	x54		x4d	x53	x47		x46	x52	x4f	x4d		x3c	___	x75	x73	x65	x72	___	___	x61	x3e
0005	x47	x45	x54		x4d	x53	x47		x46	x52	x4f	x4d		x3c	___	x64	x73	x61	x66	___	___	x61	x3e
0006	x47	x45	x54		x4d	x53	x47		x46	x52	x4f	x4d		x3c	___	___	___	x68	x66	x67	x68	x66	x3e
0003	x47	x45	x54		x4d	x53	x47		x46	x52	x4f	x4d		x3c	___	___	___	x61	x62	x63	___	___	x3e
0008	x47	x45	x54		x4d	x53	x47		x46	x52	x4f	x4d		x3c	x65	x71	x74	x73	x64	x67	___	___	x3e
DT	AAA	AAA	AAA	SSS	AAA	AAA	AAA	SSS	AAA	AAA	AAA	AAA	SSS	AAA	GGG	AAA	AAA	AAA	AAA	AAA	GGG	AAA	AAA
MT	000	000	000	000	000	000	000	000	000	000	000	000	000	000	050	066	066	066	066	050	033	050	000
ASCII	G	E	T	_	M	S	G	_	F	R	O	M	_	<	?	?	?	?	?	?	?	?	>

Table 2. Result of PI alignment

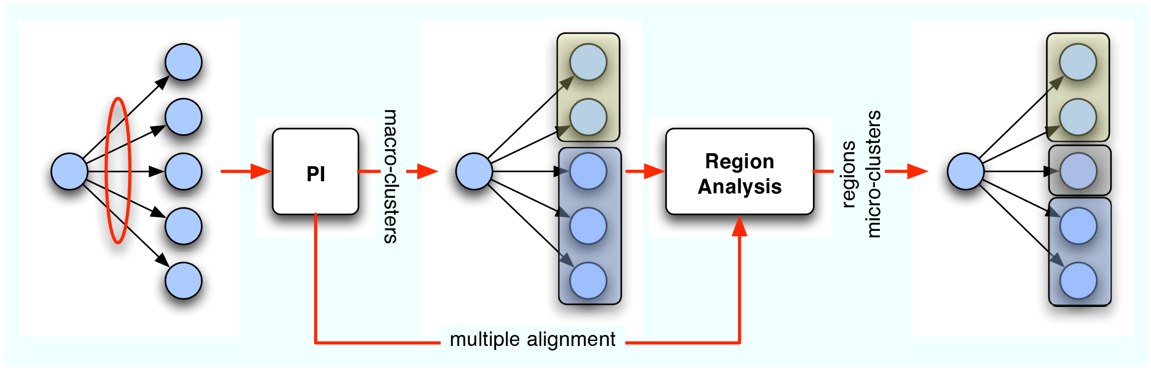

Figure 3. Region analysis sequence of operations

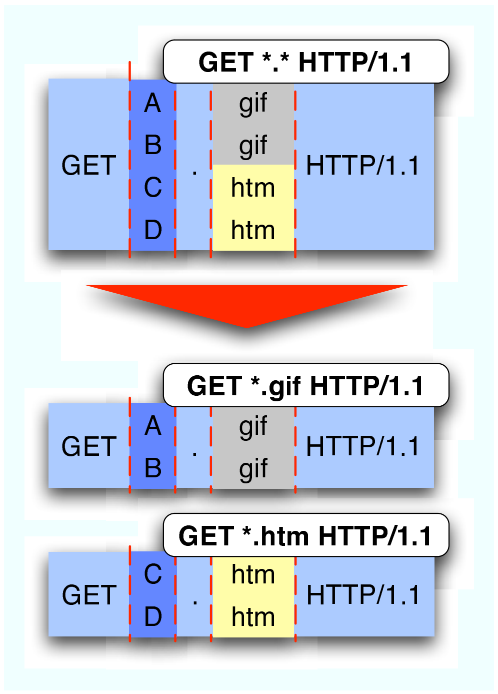

Figure 4. Example of microclustering

of the protocol (for which there is a high number of samples in the sequences) have higher probability of being put in a separate microcluster. So the emulation will be more refined for the most common functional parts.

Thus, the Region Analysis process achieves the targeted goal: it is able to add generality to a complex and specific state machine and it identifies the regions carrying a semantic value, that is, regions to be taken into consideration when determining the future state during simulation.

2.4.3 Dependency handling

Some of the identified regions play a specific role in the exchange of messages. They are those changing elements sent by a client that it expects to see sent back by the server. Session IDs are one such example. It is thus important to search for dependencies between regions of subsequent messages.

ScriptGen handles this kind of dependency between client messages and the future server answers by taking into consideration what it calls Random Regions. Random Regions are regions having a very high mutation rate, near to 100%. The dependency handler looks at the value of these regions for each sample sequence, and tries to find the same value in the corresponding server answer. If there is a match that is common to all the sample sequences traversing that specific transition, then the dependency handler stores the link that will be used during the emulation of the state machine.

Other types of dependencies come to mind that could be searched for. One can think of counters that are increased, or sequence numbers to which other values are added, etc. Not all dependencies can be automatically found and identifying precisely which ones can or cannot be handled by a similar approach is left for future research.

2.5 Script Generator

Once the State Machine is simplified, it is necessary to store it in such a way that it can be simply and efficiently emulated by a Python script usable by Honeyd. In our first prototype, this has been implemented as a simple service script: a new instance of the interpreter is created for each incoming client connection. For each state, the emulator prints the label (if present) and then fetches the client response, using the Region Analysis output to decide the future state. This is clearly not satisfactory from a performance point of view for a heavy loaded honeypot but is good enough as a proof of concept to validate the method.

It is important to choose a good algorithm to match incoming messages with all possible transitions. Reusing the same alignment algorithm is not feasible since it requires a huge amount of resources, much more than what is usually available in deployed honeypots. A simpler technique had to be found. The Script Generator focuses on the regions that Region Analysis has identified as important from a semantic point of view. They are called Fixed Regions as opposed to the Mutating Regions which are likely to contain changing parameters. The emulator implements a regular expression matching algorithm where only the Fixed Regions are considered with variable intervals of unknown characters between them. In case of a match, the emulator calculates a similarity score between the incoming message and the found message. The transition with the highest score is then chosen to reach a new state.

3 Expected Quality of the results

3.1 Known limitations

The approach described in the previous Section presents a series of problems and limitations which we are well aware of. The emulated protocol won't behave exactly like the original protocol since some approximations have been introduced. It is possible to distinguish between two types of limitations: deviations due to the approximation introduced by simplification, and those that are implicit in the approach followed.

3.1.1 Deviations due to approximation

ScriptGen has been designed to be completely automated, without any possibility of helping the simplification with additional information about the protocol behavior. This can be seen as an advantage, since it allows the emulation of any protocol of any kind without any knowledge about it. But this may also lead to challenges in performing a simplification which does not degrade too much the emulation quality.

First of all, the quality of the emulation depends heavily on the value of the thresholds chosen in the whole process. These thresholds influence the growth of the state machine (maximum fan-out) and its complexity (maximum number of nodes), and also the strength of the simplification (clustering thresholds). It may be difficult to find a good combination of these parameters, and it may be interesting to study their impact in order to find out if it is possible to identify combinations that can be considered good in all cases, or if the optimal value is protocol-dependent. This question will be addressed in some more depth in section 3.2.

Also, the sample tcpdump file has a huge impact on the final result. In fact, the characteristics of the protocol are uniquely deduced from this sample file. On the one hand, the knowledge of the protocol that can be used by the emulator is limited to the sample's content: the emulator will not be able to handle an activity that was not enclosed in it. On the other hand, the simplification uses the frequency of the various activities seen in it to define the various functional parts, identifying the semantically important regions. Only good interaction samples, using all the functionality of the protocol and presenting good diversity will lead to good results. To better understand this concept, an example may be useful. Thinking back to the simple protocol exposed in table 1 on page 4, a bad sample file may contain only the following sequences:

```
GET MSG FROM <user1>
GET MSG FROM <user2>
```

Those two sequences, when parsed by Region Analysis, would lead to identifying a transition in the state machine triggered by all the sequences matching the following regular expression:

```
GET MSG FROM <user?>
```

Of course this would be wrong, and a new user logging as "dave" would not trigger the transition, probably forcing the emulation to end prematurely. This is just a simple example, but highlights the importance that a good sampling has in the generation of the state machine.

3.1.2 Implicit limitations

ScriptGen implicitly introduces some limits that prevent it from being able to emulate all possible protocols. The first limit can be found in the fact that the notion of state in ScriptGen is local to the TCP session or to the UDP request/reply tuple. This leads to the fact that handling causality between different conversations (e.g. FTP control/data sessions) is not possible. Also, if the server response depends on a definition of state which is out of the scope of a TCP session (for instance, time of the day) ScriptGen won't be able to handle it and so the results of the simulation may not be precise.

Also, ScriptGen is obviously not able to cope with encrypted channels: it is able to replay only payloads that have been already seen in the tcpdump file. Cryptography offers mechanisms, like challenge-based authentication, especially intended to prevent a malicious user from re-authenticating by replaying the same payload, as ScriptGen would do.

3.2 Similarity tests

It can be interesting to study the impact of the simplification parameters on the quality of the emulation, defining as quality the similarity between the behavior of the emulated server and the behavior of the real one. To investigate this concept, we have developed a simple tool. This tool is able to parse a sample tcpdump file containing pieces of interaction between clients and a real server, and to replay these sequences to benchmark the emulated server. The idea is quite simple: we want to send the client messages to the emulated server, and to compare its answers with those contained in the tcpdump file itself using byte-oriented pairwise similarity. We understand that the next version of TCPopera offers the same replay feature [10]. TCPopera is a TCP traffic manipulation tool that extends the capabilities of TCPreplay [2] to reproduce realistic traffic. We will most likely use it in the future to conduct more tests but it was unfortunately not yet publicly available when we needed it.

During this test two parameters have been taken into consideration:

- Macroclustering threshold (W): this threshold controls the impact of macroclustering, defining the minimum distance between two sequences that should lead to put them into two different clusters. If W is small, Region Analysis will create many small clusters, leading to a complex and detailed state machine. If W is big, then Region Analysis will create a few big clusters that will eventually be split a second time by microclustering.
- Microclustering threshold (w): this threshold controls the way microclustering behaves, defining the frequency of the value of a region that should lead to the creation of a separate microcluster representing only that value. So when w is set to small values, microclustering will split macroclusters into many microclusters. On the other hand, if w is set to high values microclustering will not have any effect.

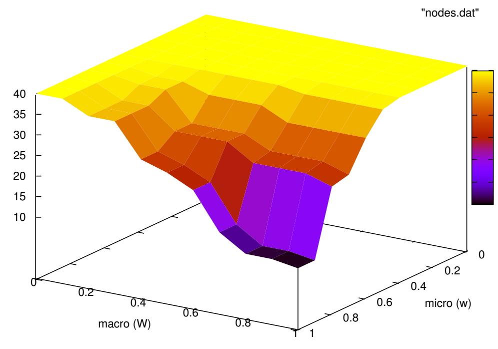

Figure 5. Number of nodes

3.2.1 The testbed

The test environment consists of one honeyd virtual host, emulating the NETBIOS Session Service (port TCP/139), and a client host running the sequences against the emulated server. The state machine has been built based on a sample interaction taken from a VMware honeypot from the Leurre'com project [4, 5, 6, 7, 8, 9]. The test has been run in the ideal case, in which the sequences used to evaluate the emulation are the same than those used to build its State Machine. The emulator, in this case, will never face unknown activities.

3.2.2 Results

The results are shown in figure 5 and 6. Figure 5 shows the variation of the number of nodes of the simplified state machine as a function of the microclustering and macroclustering thresholds. As it is possible to see, the shape in figure 5 is quite regular and, as expected, leads to the highest number of nodes in the following two cases:

- W=0, meaning that every sequence belongs to a different macrocluster
- w=0, meaning that for every macrocluster, every sequence is assigned to a different microcluster

It is clear that in those two cases the number of nodes cannot be reduced by Region Analysis. The highest simplification will correspond to the point in which w=1 and W=1, since

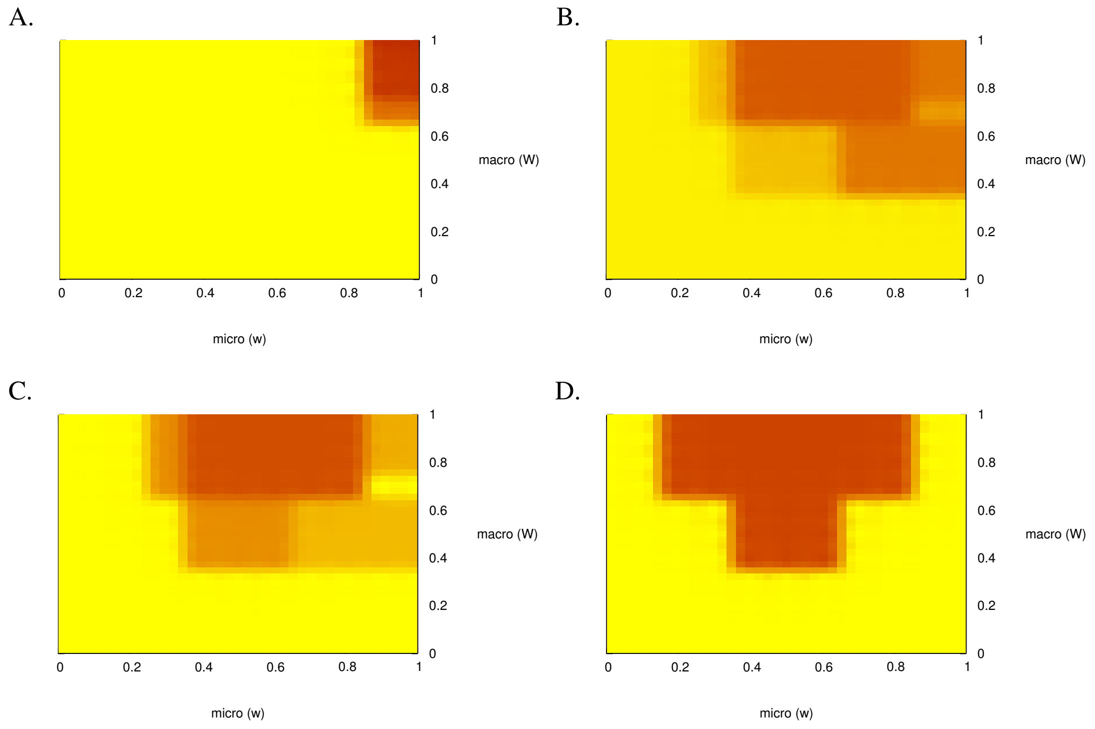

Figure 6. Score for some sequences

in that case sequences are grouped into big macroclusters, and microclustering effect is null.

Figure 6 represents the behavior of the similarity score for four different sequences as a function of the microclustering and macroclustering thresholds. The similarity score is an index of the pairwise similarity between the sequences generated by the emulated server and those generated by the real one. Lighter colors correspond to higher scores, darker colors to lower ones. It needs a bit more interpretation since the results are not straightforward.

Diagrams A and B correspond to two sequences behaving in the expected way, with a similarity score which follows the amount of simplification. Going towards the point (1,1) the score degrades: it is interesting to see that the score does not decrease gradually as the number of nodes of the state machine, but one can observe abrupt changes for some distinct values of W and/or w. This can be explained with the execution of a new collapse in the simplification algorithm on the state machine path followed by the sequence, that leads to putting into the same cluster transitions that belong to different functional parts.

Diagrams C and D show two sequences having a more peculiar behavior: with big macroclusters, average values for the microclustering threshold lead to worse values than when microclustering is disabled (w=1). This means that microclustering can lead in some situations to incorrect results, worse than the results it would be possible to obtain by disabling it. A possible explanation for this phenomenon can be found by thinking back to the interaction between microclustering and multiple alignment. Multiple alignment's objective is to maximize the similarities between the bytes of the different sequences by inserting gaps in order to align them. In a certain way, multiple alignment's final objective is to maximize the impact of microclustering by focusing exactly on those similarities and by assigning a semantic value to them. This approach is theoretically good, assuming a set of samples of infinite size. Indeed, in that case, mutating regions, without any semantic value, will be interpreted as such and correctly handled by microclustering. If the set of samples has finite size, Region Analysis and more specifically microclustering are not able to distinguish between occasional similarities between bytes and similarities due to a real semantic value of that specific protocol region. In the testbed the tcpdump sample is kept as small as possible for simplicity and for computational reasons, and so this can lead to insufficient sample data for microclustering to work correctly.

4 Experimental validation

The most interesting question that was left unanswered so far is if the quality of the ScriptGen-based honeypots is good enough to fool attack tools in exchanging with them more messages and hence more information about themselves than with a normal honeyd platform without any script. Similarity scores are useless to answer that question since they are not able to give clues on the real behavior of the emulation: there is no way of knowing whether the deviations observed in those tests can be considered relevant enough to stop the conversation with the client or not.

In order to address this point, a real ScriptGen honeypot has been deployed on the Internet in March and April 2005. The following ports were being emulated using honeyd [14] scripts:

- TCP Port 80 (HTTP), emulated by ScriptGen
- TCP Port 135 (DCE endpoint resolution), only opened
- UDP Port 137 (NetBios Name Service), emulated by ScriptGen
- TCP Port 139 (NetBios Session Service), emulated by ScriptGen

The sample tcpdump file has been built using samples of interaction extracted from high interaction honeypots data provided by the Leurre'com project [4, 5, 6, 7, 8, 9]. That file contained requests from 1107 different clients observed in a 5 months period against a VMware honeypot.

4.1 NetBios scanners

A first attempt to evaluate the performance of the ScriptGen emulation has consisted in trying to run different well known NetBios scanners[1] against our automatically generated script. All the scanners have correctly identified the host using the NetBios Name Service, correctly fetching the informations from the emulated host exactly like from a real one. So the ScriptGen honeypot is able to behave correctly in these simple scenarios, being equivalent to the emulation given by a high interaction honeypot running VMware.

4.2 Comparison with a high interaction honeypot

To have a better and more complete evaluation of the ScriptGen-based honeypot, it has been deployed on the Internet and put on the same subnet with the VMware high interaction honeypot. It has thus been possible to compare the reactions of the two different honeypot implementations placed in an homogeneous environment.

In [12], authors conclude that IP addresses can be considered free of the so called "DHCP artifacts" for periods shorter than 24 hours. This means that on such periods, each IP address can be uniquely bound to a single attacking client, even if it is dynamically assigned (as it often happens when clients are connecting to the Internet through ISPs). So analyzing the two months of collected data day by day, it has been possible to search for clients performing their activities on both honeypots, finding IP addresses common to both environments.

[1]"SoftPerfect Network Scanner" (http://www.softperfect.com); "Advanced IP Scanner" (http://www.radmin.com); "Angry IP Scanner" (http://www.angryziber.com)

Clients seen by ScriptGen honeypot	333
Clients seen by VMware honeypot	325
Clients seen by both	45
Clients seen by both performing non-null activity	30

Table 3. Results of the comparison

VM length	SG length	Sim.Vector
16	4	[C:1.0 S:1.0 C:1.0 S:0.8]
18	4	[C:1.0 S:1.0 C:1.0 S:0.8]
12	4	[C:1.0 S:1.0 C:1.0 S:0.8]
16	4	[C:1.0 S:1.0 C:1.0 S:0.8]

Table 4. Activity comparison

Table 3 shows the results that have been extracted from the observation of the tcpdump files for the two environments during the two months. For the sake of conciseness, we focus only on the NetBios TCP port 139 for which we had the most complex ScriptGen state machine. Both environments see the same amount of attacks (around 300) but only 45 attackers have hit both of them.

In table 4 it is possible to see the behavior of a single sample client attacking both environments. The first two columns show the length of the observed sequences generated by the client, that is the number of client and server messages the sequence is composed of. It is possible to see that the ScriptGen environment is performing in quite a peculiar way, generating conversations always of length 4 in the dialog with the client. This behavior can be observed in the conversations with many different clients, since 1992 sequences (67% of the total) generated by the ScriptGen honeypot always have exactly that length. It may be interesting to compare the messages sent by the ScriptGen honeypot with those sent by the VMware environment to better understand the reasons for this phenomenon. This is done in the third column of table 4 using Similarity Vectors. Similarity Vectors are arrays of similarity scores for the various messages in the sequence: in this case each message of the SG sequence is compared with each message of the VM one. The table shows that the first client request, the corresponding answer and then the second client request in the two environments are identical. There is instead a discordance in the answer that the two environments are giving to the second client request (80% similarity score): this discordance may lead the client to refuse the ScriptGen answer and to prematurely close the TCP session. Another fact that seems to confirm this hypothesis is that while each client usually performs each activity only once on the VMware environment, there are multiple subsequent attempts of running the activity on the ScriptGen honeypot.

We have analyzed the conversation with a protocol analyzer [1] and found out that the reason for ScriptGen's emulation failure has to be found in a preliminary and still incomplete implementation of the Dependency Handling (see section 2.4.3 on page 5). SMB Header in fact contains a field called Process ID, which uniquely identifies the consumer process within a virtual connection. The server answer must contain the same Process ID than the request to be considered valid and to be accepted by the client. Since the Dependency Handling support is still in a preliminary stage inside ScriptGen, it was inactive during the emulation and so the emulator has used in the answer the Process ID that had been seen in the tcpdump samples, without modifying it in order to satisfy the dependency: as a consequence, the answer was naturally rejected by the client who subsequently then closes the conversation.

We have fixed the problem and we have replayed the messages observed in the VMware environment against this new script, and the new results have highlighted the importance of the dependency handling feature. Several correct messages have then been exchanged; however, the emulator does provide erroneous replies a few steps before the end. This can be due to the fact that the observed client activity was not present in the initial tcpdump sample file. In fact, the emulator is able to handle correctly only behaviors that have been previously seen in the sample data, and may not be able to handle new kinds of activities that may be generated by a new tool or worm that was not yet spread in the period during which the sample has been built. This is confirmed by the fact that after a certain point in the conversation the certainty level with which the emulator is internally choosing the future state drastically drops to less than 50%. Even if this level is not acceptable, ScriptGen emulator chooses to carry on the conversation in any case, since there may still be a remote possibility of sending a good answer.

ScriptGen emulators, as already mentioned, may not always handle correctly newly encountered activities. In the context of the Leurre'com project [4, 5, 6, 7, 8, 9] we are more interested in gathering long term statistical data over attacks, therefore detecting promptly new activities is not one of our main objectives. Nevertheless low certainty levels may be interpreted as a signal of this situation, and therefore they may trigger appropriate reactions. An incremental version of the Region Analysis algorithm, allowing to refine an existing state machine with new samples, is still a work in progress.

4.3 Region Analysis validation

In Section 2.4.2 on page 4 the Region Analysis is posited as being able to identify the semantically important parts of the protocol, in order to generate some sort of pattern matching able to discriminate between incoming client requests and correctly choose the future state. Here it may be interesting to see the impact of Region Analysis on a real case example, analyzing the label of one of the transitions of the NetBios Session Service state machine (TCP port 139) used in the ScriptGen honeypot. The analyzed packet is a Negotiate Protocol Request, sent by the client in the initial opening steps of a conversation on that port. To understand the meaning of the various fields, a protocol analyzer has been used [1]. The fields that Region Analysis considers as important for discriminating between the various incoming packets are shown below, together with the values considered as discriminating for the transition:

NetBIOS Session Service

- Message type: Session Message
- Flags: 0x00
- (Message length is correctly not considered as discriminating)

SMB Header

- Server component: SMB
- SMB Command: Negotiate Protocol
- NT Status: STATUS_SUCCESS
- Flags: 0x18 (characterize the type of query)
- (Flags2 are considered as mutating, and include request for Unicode Strings and other stuff)
- Signature: all zeroes
- Reserver: all zeroes
- (Process ID is correctly considered as mutating)
- (UserID is mutating)
- (MultiplexID is mutating)

Negotiate Protocol Request

- (WordCount is mutating)
- (ByteCount is mutating)
- (The list of dialects is considered as mutating)

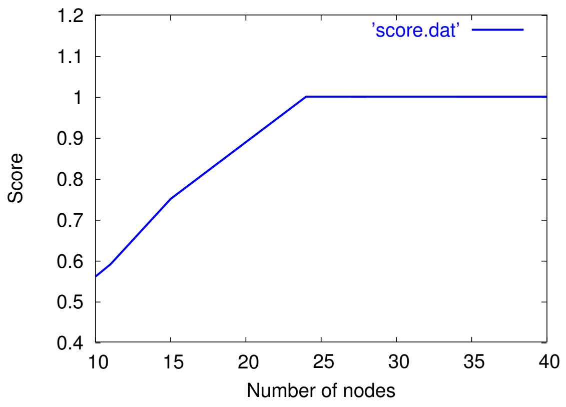

Figure 7. Similarity score as a function of complexity

So Region Analysis is correctly identifying the fields that do really have a semantic meaning (as, for instance, the name of the SMB Command) and is able to distinguish them from those that don't carry any semantic value (as the Process ID). The original state machine would not have been able to correctly handle a packet having a different Process ID from the one used in the sample, since it did not have any way of knowing the semantic value of that difference. After Region Analysis, the emulator is able to know that the Process ID field does not carry any semantic information for the choice of the future state, and correctly ignores such differences.

This highlights the importance of the macroclusterings and microclusterings and that a trivial bytewise pattern matching approach is certainly not an appropriate way to handle this issue.

4.4 Relationship between complexity and emulation quality

It has been shown in the previous Section that the ScriptGen honeypot has been able to carry on a conversation for a certain number of steps. The question is if there is any sort of control over the number of steps, in order to be able to carry on the conversation with the client as long as needed.

Elaborating the results obtained in 3.2 for a given sequence in order to correlate the number of nodes of the state machine with the obtained similarity score, it is possible to obtain the curve shown in figure 7. From this curve it is possible to obtain a direct relationship between the complexity of the state machine and the similarity score, that is the average of the values in the vectors seen in the previous section. The similarity score is equal to 1.0 if and only if the conversation generated by the emulated server is identical in content and length to the conversation that would be generated by a real server. If the conversation is shorter, the score will be lower. So a higher similarity score corresponds to a higher number of steps in the conversation between client and server.

It is then possible to state that there is a proportion between the complexity of the state machine and the quality of the emulation, in a trade-off between simulation quality and use of resources. The more resources are available for the script, the better the evaluation of the emulated service.

5 Conclusions

ScriptGen is a tool able to create honeyd-compatible emulators for any protocol using a given sample of interaction. No assumptions are made about the nature of the protocol, so it can be applied to a very wide range of different protocols without any *a priori* knowledge of their behavior.

When tested with the SMB protocol, ScriptGen has shown a certain sensitivity to the choice of the simplification parameters and to the sample tcpdump file showing the theoretical interaction. In any case it has generated emulators that were good enough to carry on a valid conversation with a client for a certain number of steps. This number depends on the complexity of the built emulator. The greater that complexity, the better the results in fooling the attack tools.

Preliminary results with this approach are extremely encouraging. They open a new avenue for research and also provide some practical results that will enable honeyd users to dramatically improve the emulation capabilities offered by this architecture. There is certainly room for improvement: dependency handling has to be enhanced and more validation needs to be carried out. Also, dynamic evolution of the state machine when new activities are encountered may represent a new interesting research topic. However, as it is, the method brings novel and important contributions on the table, namely i) enrichment of the PI algorithm thanks to the region analysis algorithm, ii) a novel method to automatically create honeyd scripts for a very large class of services and protocols, even if proprietary, iii) a sound validation methodology which provides convincing arguments in favor of the usefulness of the method.

References

[1] Ethereal, the world's most popular protocol analyzer. `http://www.ethereal.com` accessed at 03/09/2005.

[2] The tcpreplay official homepage. `tcpreplay.sourceforge.net` accessed at 03/09/2005.

[3] M. A. Beddoe. Network protocol analysis using bioinformatics algorithms. *`http://www.insidiae.com/PI` accessed at 03/09/2005*, 2005.

[4] M. Dacier, F. Pouget, and H. Debar. Attack processes found on the internet. In *NATO Symposium IST-041/RSY-013*, Toulouse, France, April 2004.

[5] M. Dacier, F. Pouget, and H. Debar. Honeypot-based forensics. In *Proceedings of AusCERT Asia Pacific Information Technology Security Conference 2004*, Brisbane, Australia, May 2004.

[6] M. Dacier, F. Pouget, and H. Debar. Honeypots, a practical mean to validate malicious fault assumptions. In *Proceedings of the 10th Pacific Ream Dependable Computing Conference (PRDC04)*, Tahiti, February 2004.

[7] M. Dacier, F. Pouget, and H. Debar. Towards a better understanding of internet threats to enhance survivability. In *Proceedings of the International Infrastructure Survivability Workshop 2004 (IISW'04)*, Lisbonne, Portugal, December 2004.

[8] M. Dacier, F. Pouget, and H. Debar. Honeynets: foundations for the development of early warning information systems. In J. Kowalik, J. Gorski, and A. Sachenko, editors, *Proceedings of the Cyberspace Security and Defense: Research Issues*, 2005.

[9] M. Dacier, F. Pouget, and H. Debar. 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, March 2005.

[10] G. H. Hong and S. F. Wu. On interactive internet traffic replay. In *8th Symposium on Recent Advanced Intrusion Detection (RAID)*, LNCS, Seattle, September 2005. Springer.

[11] E. Kohler. *The Click modular router*. PhD thesis, MIT, November 2000.

[12] D. Moore, C. Shannon, and J. Brown. Code red: a case study on the spread and victims of an internet worm. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, November 2002.

[13] Pang, Yegneswaran, Barford, Paxson, and Peterson. Characteristics of background radiation. In *Proceedings of the 4th ACM SIGCOMM conference on the Internet Measurement*, 2004.

[14] N. Provos. A virtual honeypot framework. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, August 2004.

[15] V. Yegneswaran, P. Barford, and D. Plonka. On the design and use of internet sinks for network abuse monitoring. In *Proceedings of the Network and Distributed Security Symposium (NDSS)*, February 2004.