Institut Eurécom
Department of Corporate Communications
2229, route des Crêtes
B.P. 193
06904 Sophia-Antipolis
FRANCE

Research Report RR-07-187

SGNET: a distributed infrastructure to handle zero-day exploits

3/02/2007

Corrado Leita, Marc Dacier

Tel : +33 (0)4 93 00 82 17
Fax : +33 (0)4 93 00 82 00
Email : {corrado.leita, marc.dacier}@eurecom.fr

[1]Institut Eurécom's research is partially supported by its industrial members: BMW Group Research & Technology – BMW Group Company, Bouygues Télécom, France Télécom, Cisco Systems, Hitachi Europe, SFR, Sharp, ST Microelectronics, Swisscom, Thales. This research was also supported by the RESIST European Network of Excellence, contract number 026764, and the French National project ACES, contract number ANR05RNRT00103.

Abstract

This work builds upon the Leurré.com infrastructure and the Scriptgen technology. Leurré.com is a worldwide distributed setup of low interaction honeypots whereas Scriptgen is a new class of honeypot: a medium interaction one. In this paper, we see how Scriptgen can be enriched thanks to the Argos and Nepenthes open source software in order to build a distributed system able to collect rich information about ongoing attacks and to collect malware, even for zero-day attacks, without facing the same liability and complexity issues encountered by classical high interaction honeypots. The design is precisely exposed as well as its implementation. Experimental results are offered that highlight the validity of the proposed solution.

1 Introduction

The US-CERT published in the early 2006 a security bulletin, summarizing the vulnerabilities being identified between January 2005 and December 2005. In the whole year, 5198 vulnerabilities, hitting different operating systems and applications, were reported. This is a frightening number. However, how many of them are used in practice remains unknown. Most security tools require an in-depth knowledge about each attack and the exploited vulnerability in order to provide protection against it. It is, therefore, of prime importance to start this work by looking at the most prevalent ones. The collection of trends information and the characterization of the different attacks becomes thus extremely valuable. A considerable contribution in this direction consists in the Leurré.com project[1] [13, 17, 14, 15, 18, 16]. The project used the concept of honeypot, introduced by L.Spitzner in [28], to build a worldwide observatory of attack threats and study their trends across the whole IP space over long periods of time. SGNET is a distributed honeypot framework meant to supersede the current Leurré.com infrastructure. We will show how SGNET allows to obtain very rich information about the observed attacks. SGNET enables us to collect their associated malware, even in the case of zero-day attack.

The most widely spread Internet attacks are the so-called "code injection attacks". Their final objective consists in forcing the execution of an executable code on a victim machine exploiting a vulnerable network service. Crandall et al. introduced in [9] the epsilon-gamma-pi model, to describe the content of a code-injection attack as being made of three parts:

- **Exploit (ϵ).** A set of network bytes being mapped onto data which is used for conditional control flow decisions. This consists in the set of client requests that the attacker needs to perform to lead the vulnerable service to the control flow hijacking step.
- **Bogus control data (γ).** A set of network bytes being mapped onto control data which hijacks the control flow trace and redirects it to someplace else.
- **Payload (π).** A set of network bytes to which the attacker redirects the vulnerable application control flow through the usage of ϵ and π. The payload is also commonly known as shellcode.

However, the final objective of an attack is not the code injection itself: the length of the payload is usually limited to some hundreds of bytes, or even less. It is difficult to code in this limited amount of space complex behaviors. Instead, it is used to force the victim to download from a remote location a larger amount of data: the malware. The espilon-gamma-pi model can be extended to include this dimension. We call it the *epsilon-gamma-pi-mu* model where μ stands for the malware downloaded.

[1] www.leurrecom.org

In order to retrieve precise information about a code-injection attack, all the four components of the epsilon-gamma-pi-mu model must be observed. We present in this paper SGNET, a novel honeypot framework able to emulate and observe the whole attack trace according to this model. The SGNET takes advantage of the exploit emulation capabilities of the ScriptGen approach [21, 22] and couples them with the program flow hijack detection capabilities of Argos [25] and with the shellcode emulation and malware download capabilities of Nepenthes [3]. We will show in this paper how we have been able to dynamically combine these entities to obtain a honeypot system able to observe all the four dimensions of the epsilon-gamma-pi-mu space. The contributions of this paper are manifold: 1) we present an easy to deploy honeypot setup for medium interaction, and we perform a usability study through a testing deployment on the Internet; 2) we improve the Nepenthes honeypots providing a generic vulnerability module able to incrementally handle new exploits such as zero-days, and able to detect new variants of the payload π; 3) we provide an experimental validation of the ScriptGen approach, showing that the ScriptGen technique can learn new exploits in a completely unsupervised way.

The paper is structured as follows: Section 2 recalls the principles of the ScriptGen technology. Section 3 provides a review of the related works in the field. Section 4 gives a detailed overview of the functional structure of the SGNET. Section 5 describes the SGNET implementation. Section 6 presents experimental return on experience. Section 7 concludes the paper.

2 Introduction to ScriptGen

A honeypot is a network host whose value resides in being compromised by attackers. Bailey et al. in [4] classify them according to their breadth and depth. The breadth of a honeypot system is defined as its ability to detect threats across geographical boundaries. The depth of a honeypot system represents the level of interaction with the attacking client. Solutions such as honeyd [26] allow to easily increase the breadth, but offer a very shallow depth. Instead, running real OSs inside virtualization environments such as VMware [32] provides a very profound depth but at a high cost in terms of resources and maintenance, thus preventing from achieving big breadths.

Scriptgen is an approach that aims at being a “high depth” one without compromising too much the breadth, i.e. one that can be easily deployed in the Leurré.com project to obtain large breadth and depth.

ScriptGen builds protocol emulators in a completely automated and protocol-agnostic fashion. The basic idea underneath the ScriptGen approach consists in learning the protocol behavior starting from samples of protocol interaction between an attacking client and a real host running the service. These samples are used to represent the protocol language under the form of a Finite State Machine. Each Finite State Machine modelizes the interaction between an attacking tool and a honeypot on a given protocol port. These FSMs can then be used to emulate

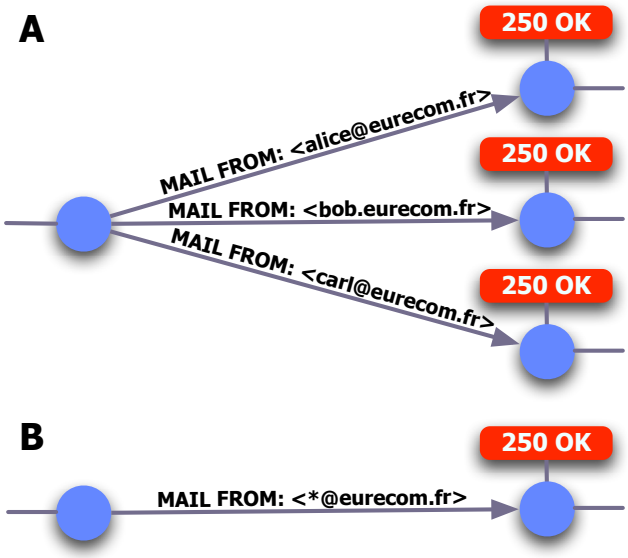

Figure 1: ScriptGen FSM

the server behavior mimicking its network interaction. ScriptGen FSMs are trees composed of states and transitions, with an optional label associated to them. For TCP, for instance, the scope of a FSM corresponds to a TCP session: the root state corresponds to the establishment of the connection. Each leaf corresponds to the termination of the connection. Excerpts of several ongoing SMTP connections are represented in such an FSM in Figure 1 A. Emulating a TCP session corresponds to the traversal of the FSM from its root to one of the leaves. Transition labels are matched with incoming client requests to choose the future state. State labels are used by the emulator as answers to be sent back to the attacking client.

When the sample protocol streams are seen as streams of unstructured bytes, the resulting FSM is too specific to correctly handle future samples of protocol interaction. For instance, if we were using the FSM represented in Figure 1 A, we would not be able to find the correct future state for a mail sent by user *dave@eurecom.fr* since we have never seen him before. A semantic abstraction is required. The Region Analysis Algorithm [22] takes advantage of the statistical variability of the samples to identify regions in the protocol stream carrying a strong semantic meaning. It takes advantage of bioinformatics alignment algorithms [23] to identify portions of the protocol streams whose value is never changing across the samples. The algorithm assumes that these portions (fixed regions) carry a strong semantic meaning, and thus are used to perform pattern matching on the incoming client requests. This allows to build, from a set of specific samples, a generic representation, with a partially rebuilt notion of semantics, as can be seen in Figure 1 B. We showed in [21] that the ScriptGen approach allows to emulate

the exploit phase for protocols as complex as NetBIOS.

The strengths of the ScriptGen approach are manifold:

- Its protocol agnostic nature allows to build in an *unsupervised* way FSMs for virtually every protocol, as long as their payload is not encrypted.
- Its ability to detect deviations from the current knowledge, as explained in [21], enables it to detect zero-days.
- Its proxying algorithm enables it to react to new activities. When facing a new activity such as a zero-day, ScriptGen will not find a path inside the FSM associated to the protocol. Thus, ScriptGen will not be able to provide an answer to the new client request received by the attacker. We showed in [21] how it is possible to take advantage of a real machine, replaying against it all the traffic received from the attacker and then act as a proxy between the attacker and the host. This allows the ScriptGen honeypot to handle correctly the conversation with the attacking machine, and more importantly it allows it to collect a new sample conversation to be used to refine incrementally the current protocol knowledge represented within the FSM.

With respect to the epsilon-gamma-pi-mu model introduced in Section 1, we can say that the ScriptGen approach aims at getting the ϵ part, leading the attacker into sending the following γ and π. Since ScriptGen focuses only on the first phase, it is unable to observe the last stage of the attack, the malware μ. An attack can be characterized as a tuple $(\epsilon, \gamma, \pi, \mu)$. Years ago Internet malicious activity was dominated by the spread of worms. In that case, it was possible to identify a correlation between the observed exploit, the corresponding injected payload and the uploaded malware (the self-replicating worm itself). Thanks to the correlation between the 4 paramaters, retrieving information about a subset of them was enough to characterize and uniquely identify the attack. This situation is changing. Taking advantage of the many freely available tools such as Metasploit [29], even unexperienced users can easily generate shellcodes with personalized behavior. This allows them to generate new combinations along all the four dimensions, weakening the correlation between them.

3 State of the art

The idea of performing automated exploit emulation by means of alignment algorithms has been considered in parallel by two different teams. This led to ScriptGen, on the one hand, and RolePlayer [10] on the other. The two approaches aim at rebuilding protocol semantics, with some important differences. Scriptgen claims to be protocol agnostic and takes advantage of the diversity of a large number of samples to build its FSMs. RolePlayer uses only two samples of protocol interaction. Since this does not allow to exploit the statistical diversity, additional information needs to be provided in order to avoid errors in identifying the fixed

and mutating portions of the protocol streams. This has an impact also on the ability to refine the protocol knowledge in a completely unsupervised way, since this additional information must be provided.

An interesting application of the RolePlayer approach can be found in a technical report available on the web [11]. In order to better address the comparison with the SGNET, we postpone its analysis to Section 5.4.

ScriptGen proved to be an extremely interesting method to emulate the exploit phase during the interaction with an attacker, being able to incrementally learn zero-day attacks. However, it lacks the capabilities to handle the injected code itself. A first attempt to solve this problem was made in [21] with the concept of *inter-protocol dependencies*.

We saw that the scope of the ScriptGen state model normally corresponds to a single TCP session. Inter-protocol dependencies define dependency relations between different TCP sessions. For instance, looking at attack samples, ScriptGen is able to identify those cases in which a TCP session is established by the attacker on a "normally closed" port after a successful client request on another TCP session. ScriptGen is thus able to learn this dependency, and reproduce this behavior through emulation. Although these heuristics represent a first step towards the code injection emulation, they present a number of shortcomings.

First of all, in order to modelize the dependencies it is required to obtain samples of the whole attack trace. Thus, to modelize the activity we need to allow an attacker to run a complete attack against a vulnerable host. This can raise a number of security concerns and raises the maintenance cost.

A second problem is more at a conceptual level. The whole attack trace can be considered as a complex function that, taking as input a network behavior and a host configuration, produces as output another network behavior (opening a bind shell, downloading a malware from a URL, ...). Inter-protocol dependencies are a set of heuristics that allow, given a certain input, to memorize the corresponding output. Instead of learning the output of the function for some given inputs, a better approach would consist in approximating the function for ranges of values. This is what the SGNET aims at, detecting code injections (detecting γ) and emulating an approximation of their behavior (emulating π) to download malware (retrieving μ).

Many approaches exist to identify code injections. We can mainly identify two families: approaches that aim at identifying an executable payload π inside a network stream taking advantage of its characteristics, and approaches that monitor a vulnerable host to detect hijacks of the control flow.

Several approaches aim at reliably detecting code injections by the observation of the network interaction between an attacker and a victim. Some of them aim at recognizing peculiar characteristics of the payload: for instance, detecting the presence of *sledges* before the executable payload [30, 2]. Some aim at detecting the presence of executable code by checking the correctness of its control or data flow. This approach can be used either to detect samples of polymorphic worms [20] (the malware μ) or to detect executable payloads π and thus buffer overflow

attempts [7, 33]. Others aim at detecting decryption routines for polymorphic shellcodes emulating their execution [24]. All these methods focus on the detection of the code injection regardless on the host configuration: the payload is detected independently from the real success or failure of the attack on the target host. This is an advantage in certain contexts, but since our objective consists in characterizing an attack we do want to know whether the attack succeeds or not on the target host. These methods could be applied in a successive phase to analyze the amount of failed attacks against a given host, but this is outside the scope of this paper and thus will not be taken into consideration here.

It becomes then interesting to detect the effect of code injection by monitoring the behavior of the target host. Among the various approaches belonging to this family, we recall Argos [25], Minos [9] and Vigilante [8]. All these approaches share a similar basic concept that is memory tainting. Keeping track of the memory locations whose content derives from packets coming from the network, they are able to detect the moment in which this data is used in an *illegal* way. All these approaches require though to execute a whole operating system together with the vulnerable applications in order to detect injections. This has two shortcomings: first of all, they are expensive in terms of resources, and thus they can achieve rather limited breadths. Secondly, in terms of epsilon-gamma-pi-mu model these solutions are limited to the first three dimensions of the attack. Even if able to detect the flow control hijack γ, the execution of the payload π must be prevented to avoid severe security concerns. This paper will show how we have been able to take advantage of the code injection capabilities of Argos, and address these shortcomings at the same time.

An interesting approach aiming at capturing and emulating the shellcode is Nepenthes [3]. Nepenthes is a honeypot with a specific objective: to download malware from attacking sources. Nepenthes is thus able to handle and observe all the four phases of the epsilon-gamma-pi-mu model. Nepenthes has proved to be of significant importance in botnet tracking studies such as in [27]. Nepenthes although suffers from two restrictions: the limited vision on the exploits ϵ and the limited vision on the payloads π. Its architecture is nicely structured into three layers:

- Vulnerability modules: Nepenthes allows the development of plugins that emulate the network conversation for specific exploits. These plugins contain information about the protocol semantics in order to retrieve the injected payload (when present) from the protocol stream.

- Shellcode detection: a signature-based engine recognizes patterns in the payload and eventually unpacks its content. An intermediate optional step consists in binding to a given port a shell emulator that receives commands from the attacker. The final output of this stage is the URL of the malicious file to be downloaded.

- Download modules: a set of plugins corresponding to different PUSH- and

PULL-based download protocols allow to collect the malware and submit it to different kinds of locations (filesystems, databases, ...)

The approach is mainly knowledge-based. It relies on some in-depth knowledge of each specific exploit and takes advantage of a set of signatures to recognize the shellcode. It is thus "blind" to any attack whose behavior falls out of the current knowledge. We will show in Section 6.3 how SGNET, going beyond the limitations of the Nepenthes approach, enables it to capture malware that it would have missed otherwise.

4 SGNET and the epsilon-gamma-pi-mu model

When facing an attacker, the SGNET activity can be separated into different parts, corresponding to the basic phases of a network attack. SGNET must emulate the network conversation with the attacker during the exploit phase (the *epsilon*). Then, it needs to detect whether the network conversation is hijacking the application control flow (the *gamma*). In case of code injection, it needs to identify the injected payload (the *pi*). Finally, it must emulate the payload behavior in order to retrieve the malware (the *mu*). This Section will show how SGNET distributes these phases to three different functional entities: *sensor*, *sample factory* and *shellcode handler*.

4.1 Epsilon: Emulating the exploit

In order to emulate the exploit phase, the SGNET needs to emulate network protocols and allow thus interaction with the attacking clients. As already shown in [22], if the emulated server does not provide a correct answer to the attacking client request, the client may abandon the conversation before sending the real code injection attack. It is thus important to provide a sufficient quality in emulation of the exploit to drive the attacker into sending the code injection. Also, coherently with what we observed in our experience with the Leurré.com project, the malicious activity is not uniformly spread over the IP space. In order to achieve the ability to observe these diversities, the SGNET must thus be able to spread its service emulation capabilities along the IP space.

The protocol emulation is delegated to the SGNET sensor. A sensor is a host bound to a set of one or more IPs in the network. Each IP can be bound to a different profile, which determines the emulated configuration and thus the service ports open to the attackers. The service emulation is delegated to the ScriptGen approach. This allows the sensors to provide a sufficient quality of emulation, enough to capture attacks, without requiring considerable amounts of resources, and gaining all the advantages already introduced in Section 2.

While handling a newly encountered attack activity, the sensor needs at first to rely on an entity, such as a real host, able to act as an *oracle* and provide the correct answers to the attacker's requests. In order to build reliable paths for the ScriptGen

FSM, the samples must achieve enough statistical diversity to allow the Region Analysis to correctly infer the protocol semantics. For instance, if all the samples are generated by a single IP and the corresponding protocol encodes information about the target IP in the application payload, Region Analysis will wrongly treat that information as a fixed region. It is important thus to deploy multiple sensors and allow a distributed collaboration between them in order to achieve the necessary statistical variability.

After this step, if the path was correctly built the sensor will be able to take advantage of the FSM information to autonomously handle similar attacks. It is important to notice that while the learning phase is expensive in terms of resources, the handling of attacks based on the FSM knowledge is cheap. One of the objectives of the SGNET will thus consist in trying to reduce the learning phase, taking advantage of the collaboration of multiple distributed sensors and thus increasing the sample variability and the sample collection rate. We showed in [21] that a limited number of samples (around 50) is enough to generate a reliable protocol path for a given exploit. We will validate this result with real Internet attacks in Section 6.1.

4.2 Gamma: detecting the control flow hijack

The knowledge generated by the oracle and synthesized in the FSM allows the sensor to emulate the exploit autonomously. Normally, the output of the oracle is a network conversation, that thus provides information about the exploit emulation. If the oracle was able to provide information also about code injection parameters, it would then be possible to provide this information to the sensor and allow the sensor to extend its knowledge in terms of epsilon-gamma-pi-mu model.

We propose here a solution to implement an oracle for SGNET sensors focusing on two aspects: 1) the security measures to control the state of the oracle even after a successful code injection; 2) the ability of the oracle to provide, in addition to the network behavior information, also information about hijack points in the application control flow.

The SGNET entity having the purpose of acting as an oracle with respect to SGNET sensor is the SGNET sample factory. In order to address both security concerns and to extract information about the code-injection, we rely on *Argos*, presented by Portokalidis et al. in [25]. Argos takes advantage of *qemu*, a fast x86 emulator [5] to implement memory tainting. The sample factory takes advantage of the Argos honeypot system to achieve a different goal with respect to its original one. In SGNET, in fact, the Argos honeypots are not supposed to directly interact with attackers: they are always mediated by sensors. They are indeed factories of samples for the SGNET sensors, providing exploit (ϵ) and code-injection (γ,π) information.

When relaying on a sample factory, a sensor replays against the Argos emulated host the attack trace sent so far by the attacker. This may raise some security concerns: the attacker is in fact able to compromise the host, and thus may take

advantage of the compromised machine as a stepping stone towards other hosts. To prevent these undesired effects, we provide two levels of protection. First of all, all the packets generated by the guest host having as a destination address an IP different from the attacking source being handled by the sensor will be dropped. Secondly, the Argos technology takes advantage of the memory tainting technique illustrated in [25] to detect flow control hijacks. In the case of a detected code injection the sample factory will stop the execution of the guest operating system.

The generated network conversation does not include per se any information about possible successful code injections. We have been able to extract this information from Argos, extending the memory tainting technique to include information about the packets containing the code injection. When a sensor is forwarding the sample factory and the packets $P_1, ..., P_n$, if P_n triggers a code injection the sensor will be immediately notified. The sample factory is thus able to provide exact information to the sensor about a successful control flow hijack (gamma dimension). Also, it is possible to identify the network packets containing the shellcode.

The interaction with the sample factory allows the ScriptGen learning phase to incorporate code injection information inside the FSM. This allows the SGNET sensors to know when a FSM traversal corresponds to a code injection, and provides useful hints about the position of the payload. Nevertheless, for the security concerns mentioned before, the Argos host will be stopped *before* the payload execution. The handling of the payload information is delegated to a different entity.

4.3 Pi: Handling the payload

In order to handle the payload dimension, it is necessary to identify its position inside the protocol stream. We saw in the previous Section that the sample factory allows us to retrieve the position of the first network byte being executed by the guest host. We make here a very simple assumption: the injected payload will correspond to the network bytes *following* the first executed byte. Whenever a code injected is detected by a sensor, either through the interaction with a sample factory or through the information embedded in the FSMs, it will define as injected payload all the bytes of reassembled protocol stream following that byte. The (in)validity of this assumption will be discussed in Section 6.3.

The SGNET sensors are then entities able to correctly handle the exploit phase, and provide information about the presence of code injections and about the candidate payload π. This specification is compatible with that of the *Nepenthes* [3] vulnerability modules. We thus take advantage of the sensors to directly feed payloads into the Nepenthes shellcode manager bypassing all its vulnerability modules, i.e. circumventing its knowledge base. If a new attack tool is spreading whose characteristics fall out of the Nepenthes knowledge, SGNET will still be able to provide correct information about the exploit and the successfulness of the code injection. This is a major contribution with respect to the previous work in Nepenthes [3] as we are getting rid of its main limitation, namely the need to develop a large number of highly specific vulnerability modules.

4.4 Mu: downloading the malware

The final phase μ of the attack functionally corresponds to the Nepenthes download modules. The SGNET shellcode handler is not allowed to directly access the network. The only SGNET component having the right to directly access the network is the sensor: it is the sensor that handles the whole network conversation with the attacking client, emulating all the steps of the epsilon-gamma-pi-mu model.

We saw in Section 4.1 that the sensor relays on an oracle to emulate the exploit phase when the attack is unknown, and we saw that the ScriptGen approach allows to learn the exploit activity taking advantage of the generated samples. The μ phase corresponds to a network behavior that is always unknown to the sensor: we motivated in Section 3 the reasons for which the ScriptGen approach does not adapt to its learning. The SGNET shellcode handler is thus an oracle with respect to the malware download phase: the sensor will rely on it to generate the correct packets to download malware every time that a shellcode has been correctly recognized. Differently from the exploit emulation phase, the sensor will *not* try to learn the behavior of the shellcode and it will always rely on it. This solution does not impact scalability, since differently from the sample generators the shellcode handlers do not have significant resource requirements.

5 The SGNET

We have implemented a SGNET prototype and deployed it in the Internet. In this Section we show how we have implemented it.

5.1 The architecture

The SGNET aims at being a distributed system. The SGNET sensors must be deployable in different locations of the IP-space in order to increase the variability of the samples. The various SGNET sensors must exchange collected samples to learn the new exploits and offload the sample factories. A way to allow distributed communications between the various components is necessary. This goal is achieved through a simple TCP-based HTTP-like protocol designed specifically for this purpose: the *Peiros* protocol. Through this protocol, the sensors are able to send requests to the other entities, exchanging the various parameters needed for their initialization. With respect to Peiros, the SGNET entities are service providers, to which clients (the SGNET sensors) can subscribe.

In order to coordinate the sample distribution, we chose the centralized approach as shown in Figure 2. This greatly simplifies the complexity of the task and the synchronization between different sensors. A central location, called *SGNET gateway*, acts as a default home for all the SGNET sensors. The gateway acts as an application proxy for the sample factories and the shellcode handlers, deployed in a private network and not directly accessible from the sensors. It receives all the

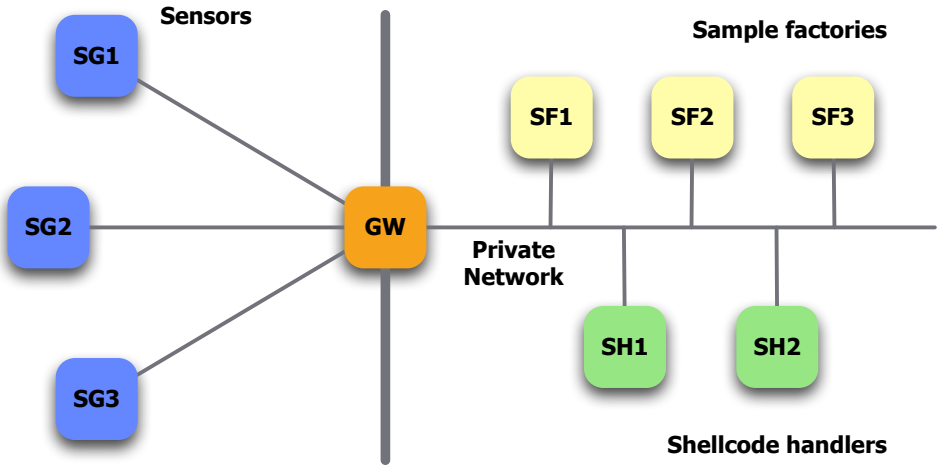

Figure 2: SGNET architecture

service requests from the clients, and dispatches them to the appropriate entity able to handle them. Multiple sample factories and shellcode handlers can be deployed on different hosts, and the gateway acts as a simple load balancer using round robin scheduling.

All the interactions between the sensors and the other SGNET entities are mediated by the gateway. The gateway is able to observe the network traffic between any sensor and the oracle generated by the sample factory, and it is thus able to collect samples of new attacks observed by *all* the sensors deployed in the SGNET. The gateway position as a centralized sample collector allows the centralized refinement of the FSMs taking advantage of the ScriptGen approach. The generated FSMs are then *pushed* to all the sensors at each update: all the sensors active at a given moment will thus have the same protocol knowledge, with some approximation due to network latency and retransmissions.

5.2 RAW Proxying

We propose in this paper an important contribution to the current state of the art of the ScriptGen technology. We showed in Section 4 that SGNET sensors need to rely on oracles to handle network conversations whose knowledge is not represented in the ScriptGen FSMs. This behavior was referred to as *proxying* in [21], since the sensor behaves as a proxy between the attacker and a real host (the oracle).

The initial proxying algorithm, as introduced in [21], was application level proxying. That is, the ScriptGen honeypot was handling reassembled TCP streams, or UDP data payloads. The information about packet boundaries was thus ignored. We considered this approach good since most of the exploits currently observable

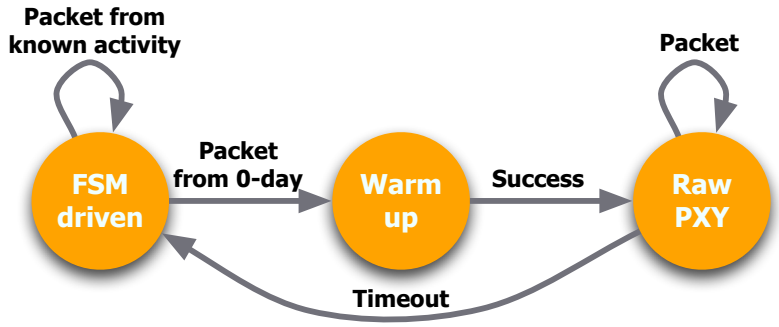

Figure 3: Proxying FSM

on the Internet target application-level vulnerabilities and not the TCP/IP stack. The practical experience in replaying Internet attacks such as Blaster [6] underlined the importance of preserving packet boundaries in order to correctly reproduce the attack trace. Also, preserving the TCP/IP headers would allow to correctly reproduce attacks based on misuse of the TCP/IP header fields. We thus introduce here a RAW proxying algorithm able to replay exactly the same attack trace observed by the sensor.

Given an IP T and an attacking source (IP address) S, a sensor defines an attack as the whole ordered sequence of packets $(P_1, P_2, ...P_n)$ sent by S towards T. An attack can then spread over several TCP sessions, UDP requests and ICMP packets. Using a finer proxying granularity such as the single TCP session or UDP packet sequence would be wrong: the attack trace may be the result of several state modifications obtained through multiple TCP sessions or UDP packet sequences. A sensor maintains thus a different proxying state for each couple (S, T) of attacking source and target address. Each couple (S, T) can be bound to three different states, which evolve according to the FSM represented in Figure 3:

- **FSM driven.** The sensor handles the exploit with the attacker taking advantage of its own FSMs. In this case, the sensor takes advantage of the normal kernel TCP/IP stack, that handles retransmissions and duplicate packets and provides to the sensor the application data stream. Taking advantage of the Netfilter ipqueue libraries [34], the sensor caches all the RAW IP packets $P_1...P_i$ sent from A to S.

- **Warm up.** When the sensor faces a request for which there is no knowledge in the FSMs, it needs to initialize an oracle H and act as a proxy to handle and learn the newly encountered attack activity. During this phase the sensor replays to the oracle the raw IP packets $P_1...P_i$ received in the previous state in order to reproduce the attack trace.

- **RAW proxy.** After the initialization, the sensor will prevent its TCP/IP stack to receive any packet coming from the attacker A targeting IP T, dropping

them using the ipqueue libraries. The attack packets $P_{i+1}, ...P_n$ will be instead pushed directly to H.

Proxying potentially arises an important issue. We can identify in a network protocol two different kinds of cookie fields: server-driven and client-driven. A server driven field is a protocol field whose value is decided by the server when answering to a request. It can be used by the client to determine other field values for the following requests. Client-driven cookie fields are instead set by the client in the client requests, and then used by the server to generate the following answers. In [21] we claimed the importance of handling the client-driven cookie fields to raise the emulation quality: if a client sets a cookie field, the ScriptGen-based emulators need to take into consideration its value when generating their answers. Server-driven cookie fields instead do not generate any concern with respect to the ScriptGen emulation: the logic to handle the transformation of the field in the following packets is embedded in the client. Nevertheless the initialization of the RAW proxying in the middle of a protocol interaction may lead to an inconsistency: the value chosen by the sensor and by the oracle host for the server-driven field could be different. The transition between FSM driven operation and RAW proxy will not be transparent to the attacker. In order to solve the problem, the sensor needs to compare the answers generated by the oracle in the warm up phase with the answer generated by the sensor when driven by the FSM knowledge, learn the modifications and reverse them in the RAW proxy phase.

Even if theoretically possible, we never observed this kind of inconsistency at application level; although, we encountered it at the transport layer form when dealing with TCP sequence numbers. When opening a TCP connection with the attacker, the TCP/IP stack of the sensor chooses an initial sequence number ISN_1. This initial sequence number is analogue to a server-driven cookie field: in the warm up phase, when establishing the connection the oracle will choose a different initial sequence number ISN_2. The replay engine needs thus to observe the answers generated by the oracle in the warm up phase in order to learn ISN_2. In the RAW proxy state, the sensor will need to reverse this modification changing all the sequence numbers related to the oracle by the quantity $ISN_2 - ISN_1$.

It is clear from this description that RAW proxying is comparable to a TCP session hijacking attack. In fact, the RAW proxy phase overrides the host TCP/IP stack and redirects the packets towards another TCP/IP stack which carries on the conversation. When, after a period of inactivity T_o, the sensor assumes the source S as "expired", it reverts its state to FSM driven allowing it to access again the hijacked TCP/IP stack. If T_o is not long enough, the TCP/IP stack will be desynchronized and will thus potentially generate TCP ACK storms. It is thus important to correctly tune T_o to a period of several minutes and to check the ACK ratio of each source to timely block these situations.

5.3 Attacks and IP addresses

When facing a newly encountered activity, a sensor needs to rely on an oracle provided by the sample factory. The sensor thus sends a service request through the Peiros protocol to a free sample factory. If enough resources are available, the sample factory initializes an Argos instance using the guest operating system associated to that sensor (e.g. a Windows 2000 configuration). It is possible to take advantage of the virtualization capabilities of Argos to load the memory snapshot of an already running system in less than one second, allowing thus an extremely fast initialization.

It would be possible in theory to always associate the same IP address to the guest host, and then perform NAT while replaying the packets from the sensor to the host. However, many exploits such as the LSASS exploit [1] put in the application payload the IP address of the target machine. The presence of NAT during the replay of the attack trace leads the attack to fail, since the IP address in the application payload will not match any more the IP address associated with the host network interface. Thus, the IP address of the host handled by the oracle *must* match the IP address of the sensor.

Matching the IP address of the host with the IP address of the sensor proved to be a difficult task. We implemented a "smart" DHCP server inside the sample factory. When initializing a host, the sample factory takes advantage of the DHCP protocol [19] to assign it the same IP address than the one of the requesting sensor. We found out that this solution is unfeasible when dealing with certain Microsoft operating systems. During our testing with a Windows 2000 unpatched system, we observed an extremely peculiar behavior. When assigned a new IP address, the host starts to broadcast its presence for a period of approximately 30 seconds. During this period, any communication attempt with the running services fails. This introduces an important delay in the sensor warm up phase, leading the attacking source to timeout. We consequently decided to store a different memory dump for each sensor being placed in the network. This allows to immediately initialize a guest host having the desired IP address in a negligible time at the expense, of course, of disk space.

5.4 Comparison with GQ

An interesting work sharing some similarities with SGNET is described in a technical report that can be found on the Internet, and called GQ [11]. GQ is a high interaction internet telescope, taking advantage of the application-level filtering capabilities of RolePlayer [10]. The main idea of GQ consists in increasing the breadth of a set of virtualized hosts performing a smart filtering on the observed activities. The well-known attack activities are handled by RolePlayer scripts, while new and interesting attack activities are left to the virtualized hosts.

GQ shares thus a similar idea to SGNET, but profoundly differs in the architecture.

First of all, SGNET is designed to be a distributed system. We are aware thanks to our efforts with the Leurré.com project that the attack activity is *not* uniformly spread along the whole IP space. SGNET aims at observing the attack activities in different locations of the IP space, deploying different sensors synchronized by a central entity. On the other hand GQ is a highly interactive Internet telescope, and thus aims at observing global trends and background radiation in large blocks of addresses rather than at observing threats in diverse environments.

Secondly, SGNET precisely separates the various phases of the epsilon-gamma-pi-mu model and handles each phase with a different entity. We believe that the FSM model generated by ScriptGen, or by RolePlayer, fits only to the emulation of the exploit phase but is insufficient to model the complex interactions inherent to the code-injections. A small modification in the injected payload can completely modify the network behavior of the attack, and representing all the possible behaviors in the FSMs would lead to an explosion of the number of paths. SGNET thus takes advantage of different entities that better fit to handle the different phases of the attack trace, dynamically switching between them. From our understanding of GQ, this does not seem to be the case.

Thirdly, SGNET is designed to operate in a completely unsupervised way. We will show in Section 6 how we are able to incrementally refine the protocol knowledge collecting new samples of protocol interaction. We showed in Section 3 that, differently from ScriptGen, RolePlayer does not suit well to automated learning. Indeed, no automated learning capabilities are mentioned in the current literature about GQ.

Finally, SGNET is able to model the whole attack trace without executing the code injection itself. Allowing the execution of the whole attack trace on real hosts taking advantage of virtualization, as proposed by GQ, raises security concerns that we prefer to avoid.

6 SGNET experimental results

A prototype of the SGNET infrastructure was deployed on the Internet and has been running for more than one month[2]. The various partners of the Leurré.com project are being invited to join the experimentation of the new infrastructure. At this time of writing, two SGNET sensors are operational: one, bound to a single IP, is running in France; another, bound to 3 sequential IPs, is running in Australia. All the sensors are associated to an unpatched Microsoft Windows 2000 machine, running the IIS services. Several open TCP and UDP ports are associated to its corresponding FSM, such as TCP ports 135, 139 and 445, UDP port 137 and others. More sensors will be deployed in the following months. Interesting results have already been obtained with this initial setup. They are used hereafter to validate the whole approach.

[2]Note to the reviewers: if accepted, we will be glad to enrich this Section with updated figures.

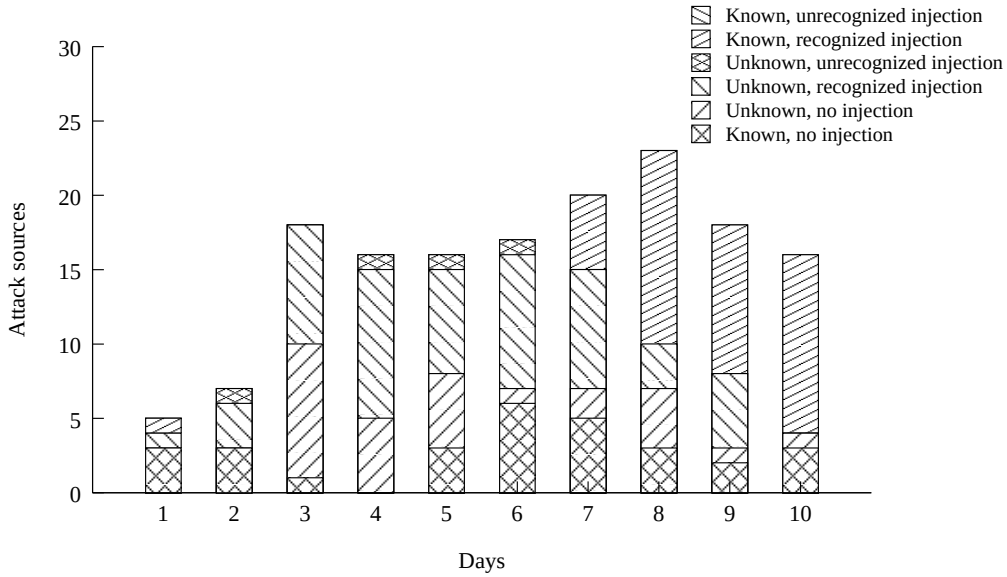

Figure 4: Learning phase

When evaluating the SGNET, several points must be considered: the reaction time, the stability of the FSMs, the correctness of the code injection information and the ability to download malware. The *reaction time* corresponds to the time that it takes to generate a new path in the ScriptGen FSMs for a given new attack activity. Once the path is generated, its *stability* is important: the semantic abstraction provided by the ScriptGen approach must be good enough to allow the sensors to handle autonomously future similar attacks without continuously creating new variations of that same path. The code injection information learnt from the sample factory must allow a sensor to provide the shellcode handler with correct payloads. Finally, the provided payloads must allow Nepenthes to effectively download malware. These various aspects will be addressed in the following sections.

6.1 Reaction time

In the following, we report on our observations of the behavior of the ScriptGen sensors on a specific port, the Microsoft DCOM Service Control Manager (TCP 135). This port has been chosen for two reasons. On the one hand, it corresponds to a relatively complex binary protocol, thus constituting a non-trivial example of automated learning. On the other hand, it is hit by a significant number of attackers, accounting for 32.5% of the total number of observed attackers hitting open ports.

On day 0, the experiment was started with a completely untrained configuration. Initially, all the FSMs provided to the SGNET sensors were empty. We then tracked the behavior of the SGNET in handling the attackers along the days. Figure 4 focuses only on the first 10 days of observed behavior for a stable SGNET

setup. Figure 4 groups the attackers according to how they have been handled by the SGNET. The attacks can be known, that is the sensor was able to handle the conversation taking advantage only of the knowledge represented in the FSMs; or they can be unknown, requiring thus the interaction with the sample factory. With respect to code injections, a code injection may or may not be detected, and may or may not be recognized by the shellcode handler.

In almost every day, we can notice a rather constant amount of attackers whose behavior is known to the sensor but who do not inject code. These sources are actually connecting to the port and then disconnecting immediately without sending any payload. This likely corresponds to scanning activities.

The first 6 days are dominated by unknown activities, that require the sensor to rely on the sample factory. On the 7th day, after having collected 73 attack samples for that single port, a new protocol path is generated by the SGNET and pushed to the sensors. Even if the breadth of the SGNET is at the moment limited, it has been able to incorporate in its FSMs a new activity in a relatively short amount of time.

These results clearly underline the ability of the ScriptGen approach to learn Internet attacks in a completely unsupervised way within a reasonable learning time. This result is an important point in validating the whole SGNET architecture. What has just been said for port 135 was also observed for another commonly exploited port, that is port 139 (NetBIOS Session Service). Other less commonly attacked ports have not developed yet stable protocol paths in their FSMs mainly because of the lack of samples. As soon as more sensors will be deployed on the Internet, the reaction time of the systems will decrease since more samples will be made available more rapidly. At this point, it is worth noting that, unfortunately, these two sensors are located within IP blocks of addresses that are among those that get the fewest number of attacks per hours, as indicated by our Leurré.com statistics.

6.2 Stability

When creating a protocol path, it is important to understand whether the semantic abstraction performed by ScriptGen is sufficient to correctly handle newer instances of the same attack. By design, a FSM path is never refined: once created it will never be modified. If the generalization is not sufficient, new attack samples will not traverse the path and will trigger again the proxying algorithm to rely on the sample factory. This means that if the generalization is not sufficient and the path is not stable the sensor will never be able to handle autonomously an attack. Also, this will lead to an explosion of the complexity of the FSMs, with many protocol paths never traversed by any sample.

According to what we have just said, the verification of the stability of the paths consists in observing a decrease of the unknown activities after a refinement of a FSM. Referring to Figure 4, the sudden increase on day 7 of known activities with recognized code injections is a clear validation of the stability of the generated

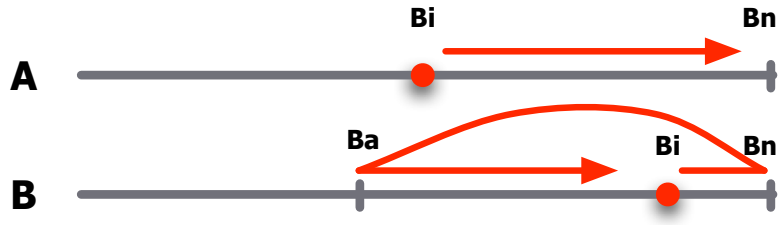

Figure 5: Finding the payload

path.

6.3 Recognizing the shellcode

Figure 4 shows that after day 7 SGNET sensors are able to successfully emulate exploits against port 135 and submit recognizable payloads to the Nepenthes framework. The same has happened for the protocol paths generated for port 139. It is also interesting to notice in Figure 4 a small fraction of activities that generated a code injection not recognized by Nepenthes. It is interesting to focus on these cases since they underline a possible failure of the knowledge based model used by Nepenthes.

In the beginning of the testing we ran into a considerable number of cases in which the shellcode was not recognized correctly. The information provided by the Argos honeypots contains hints on the first byte B_i of payload pi being executed by the host. When embedding this information in the new protocol paths of the ScriptGen FSM, we considered as payload all the following bytes $B_i, B_{i+1}, ...B_n$ up to the end of the reassembled application-level stream (Figure 5 A). This approach was often generating extremely short payloads, consisting only of a few bytes. The real behavior of these payloads is shown in Figure 5 B. The identified payload consists of a jump instruction to another memory location containing most of the payload, that was located *before* B_i in the reassembled application stream.

We revised our initial assumptions as follows. Given a reassembled application level stream $B_1...B_n$ identified by Argos as containing a payload pi at byte B_i, the sensor tries to submit a payload $\pi = (B_k, ...B_n)$ with $k \leq i$ to the shellcode handler. The index k is gradually decreased starting from i until the payload is recognized successfully. This allows to backtrack from the initial hint given by Argos, that in this particular situations proves to be misleading. Since the payload recognition takes a very small time on the shellcode handler, the heuristic adds a minimum overload.

This heuristic allowed to increase the recognition ratio of the shellcodes, unveiling a much more interesting phenomenon. In the last week of December 2006, SGNET logged a high number of shellcodes injected through port 139 and not being recognized by the shellcode handler. 147 out of a total of 200 submitted

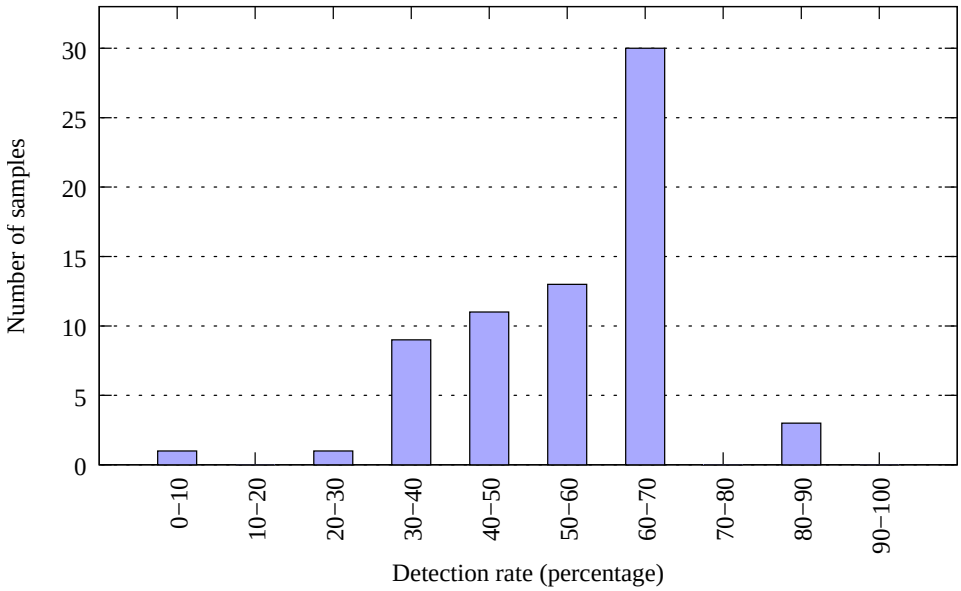

Figure 6: Antivirus detection rate for downloaded malware samples

shellcodes were not detected, catching thus our attention. After submission of the collected payload samples to the Nepenthes development team, the signature for one class of shellcodes (*bindfiletransfer:amberg*) was modified. ScriptGen was in fact collecting samples differing by 3 bytes from the original signature. This difference is probably due to the fact that the shellcode had been modified by using different opcodes for the same operations. This episode is extremely important since it underlines two facts: 1) the knowledge-based approach used by Nepenthes to detect and emulate shellcodes can be evaded; 2) the SGNET allows to observe these cases and take the appropriate measures.

6.4 Downloading the malware

Finally, some considerations must be made on the malware download phase. The amount of malware effectively collected by the SGNET with respect to the number of detected and successfully emulated code injections is rather low.

Looking at the SGNET logs for the period going from the 10th to the 26th of January 2007, we can deduce some interesting statistics. Out of 227 submitted shellcodes detected and recognized by the shellcode handler, only 60 led to successful download of malware samples. Of the remaining 167, 110 were actually corresponding to non-routable addresses mainly belonging to the 192.168.0.0/16 network. This result is extremely surprising, but can be justified by the fast utilization in the nowadays Internet of private addressing and NAT.

We can in fact recognize two different classes of downloads: those in which the

Name	Result (19/1)	Result (26/1)
AntiVir	found nothing	**found [Worm/Allaple.B.151]**
Authentium	found nothing	**found [W32/NetWorm.BL]**
Avast	found nothing	found nothing
AVG	found [Worm/Allaple.B]	found [Worm/Allaple.B]
BitDefender	found nothing	found nothing
CAT-QuickHeal	found nothing	found nothing
ClamAV	found nothing	found nothing
DrWeb	found nothing	found nothing
eSafe	found [Suspicious Trojan/Worm]	**found [Win32.Allaple.b]**
eTrust-InoculateIT	found nothing	found nothing
eTrust-Vet	found nothing	found nothing
Ewido	found nothing	**found [Worm.Allaple.b]**
F-Prot	found nothing	found nothing
F-Prot4	found nothing	found nothing
Fortinet	found [suspicious]	**found [W32/Allaple.B!worm.im]**
Ikarus	found nothing	**found [Net-Worm.Win32.Allaple.b]**
Kaspersky	found [Net-Worm.Win32.Allaple.b]	found [Net-Worm.Win32.Allaple.b]
McAfee	found nothing	found nothing
Microsoft	found nothing	found nothing
NOD32v2	found nothing	found nothing
Norman	found nothing	found nothing
Panda	found nothing	found nothing
Prevx1	found nothing	found nothing
Sophos	found [Mal/Packer]	found [Mal/Packer]
Sunbelt	found nothing	found nothing
TheHacker	found nothing	found nothing
UNA	found nothing	found nothing
VBA32	found nothing	found nothing
VirusBuster	found [Worm.Allaple.Gen]	found [Worm.Allaple.Gen]

Table 1: Example of submission

attacker forces the victim to download malware from a central location, such as an FTP or HTTP server, and those in which the attacker forces the victim to download the malware from its own host, usually taking advantage of a small TFTP daemon. The latter case can be greatly impacted by the presence of NAT: the attacker will use as address for the TFTP server the IP assigned to its own network card. In many cases, such as the WiFi/ADSL routers normally distributed by many ISPs, that address is not routable and belongs to a private network, masked to the outside world by NAT. In these cases the worm has no chance or propagating, even if it continuously scans the network for vulnerable machines.

Nevertheless, we have been able to successfully download malware samples. We submitted these samples to VirusTotal [31], a free service allowing to scan suspicious files using several well known antivirus engines, both commercial and open. At the moment of writing, VirusTotal is offering 28 different antivirus engines, most of them constantly updated with the latest signatures. Figure 6 shows the detection rate distribution for the submitted samples downloaded by the

SGNET. We define the detection rate of a malware sample as the ratio of antivirus softwares having identified the sample as containing something malicious. It is interesting to see how the detection rate is almost always below 70%, thus meaning that in average at least 8 antivirus softwares fail to recognize the submitted malware as being malicious. In parallel to well-known worms such as Blaster, we have been able to observe relatively recent malware samples such as Allaple.A, discovered according to the F-Secure database on the 7th of December 2006.

An important issue not addressed in the scope of this paper is the problem of broken malware downloads. Some of the upload methods used by attackers are based on unreliable protocols, and thus the emulated download phase may end up in a corrupted file. An example of this behavior is shown in Table 1. We submitted the sample to CWSandbox [12], a new sandbox implementation able to analyze the host-based behavior of a malware sample. According to the sandbox analysis, the file cannot be executed and is thus broken. All the successful detection cases can thus be considered as *false positives*. The presence of this kind of false positives mainly depends on the antivirus policy and in the aggressiveness of its detection engine. The even more surprising result is that when submitting the same malware a second time one week later, we observed that the detection rate had "improved". A more in depth analysis of the downloaded samples would be necessary to better understand this phenomena. It is indeed important to underline how, to collect meaningful detection statistics, it is important to recognize the broken samples taking advantage of technologies such as CWSandbox. This is out of the scope of this work and it is left for future investigation.

7 Conclusions

We presented in this paper a novel infrastructure to observe Internet attacks. We showed how, focusing on code injection attacks, we have been able to address the epsilon-gamma-pi-mu model and emulate the steps required to successfully download malware samples. We took advantage of three different approaches, namely ScriptGen, Argos and Nepenthes, and we have been able to exploit their strengths in addressing specific phases of the attack process. We showed how the ScriptGen approach can act as a generic vulnerability module for Nepenthes, providing behavior-based information and allowing to identify the limitations of the Nepenthes knowledge-based approach. Also, we have been able to concretely validate the ScriptGen approach by handling successfully real Internet attacks. The ongoing deployment of SGNET sensors in different locations of the IP space will allow us to gather a more detailed picture of the local threats observable in the Internet.

References

[1] Microsoft Windows LSASS Remote Overflow, http://www.osvdb.org/5248,

2007.

[2] P. Akritidis, E.P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. *20th IFIP International Information Security Conference*, 2005.

[3] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[4] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The internet motion sensor: A distributed blackhole monitoring system. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, February 2005.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[6] CERT. Advisory CA-2003-20 W32/Blaster worm, August 2003.

[7] Ramkumar Chinchani and Eric van der Berg. A fast static analysis approach to detect exploit code inside network flows. *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, 2005.

[9] J.R. Crandall, S.F. Wu, and F.T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[10] Weidong Cui, Randy H. Katz, and Wai-tian Tan. Protocol-independent adaptive replay of application dialog. In *The 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[11] Weidong Cui, Vern Paxson, and Nicholas Weaver. Gq: Realizing a system to catch worms in a quarter million places. Technical report, ICSI Tech Report TR-06-004, September 2006.

[12] CWSandbox - Automated Behavior Analysis of Malware. www.cwsandbox.org, 2007.

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

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

[15] Marc Dacier, Fabien 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.

[16] Marc Dacier, Fabien 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.

[17] Marc Dacier, Fabien 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.

[18] Marc Dacier, Fabien 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.

[19] R. Droms. Dynamic Host Configuration Protocol; RFC-2131. *Internet Request for Comments*, 2131, 1997.

[20] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. *Symposium on Recent Advances in Intrusion Detection*, September 2005.

[21] Corrado Leita, Marc Dacier, and Frédéric Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *RAID 2006, 9th International Symposium on Recent Advances in Intrusion Detection, September 20-22, 2006, Hamburg, Germany - Also published as Lecture Notes in Computer Science Volume 4219/2006*, Sep 2006.

[22] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.

[23] Saul Needleman and Christian Wunsch. *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.

[24] M. Polychronakis, K.G. Anagnostakis, and E.P. Markatos. Network-Level Polymorphic Shellcode Detection Using Emulation. *Proceedings of the*

GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), July, 2006.

[25] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. *Proc. ACM SIGOPS EUROSYS*, 2006.

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

[27] Moheeb Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *ACM SIGCOMM/USENIX Internet Measurement Conference*, October 2006.

[28] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Welsey, Boston, 2002.

[29] The Metasploit Project. www.metasploit.org, 2007.

[30] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *5th Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2002.

[31] VirusTotal. www.virustotal.com, 2007.

[32] VMware Inc. The VMWare software package, www.vmware.com, 2007.

[33] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. Sigfree: A signature-free buffer overflow attack blocker. *USENIX Security*, 2006.

[34] H. Welte. The Netfilter framework in Linux 2.4. *Proceedings of Linux Kongress*, 2000.