SGNET: a worldwide deployable framework to support the analysis of malware threat models

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

Abstract

The dependability community has expressed a growing interest in the recent years for the effects of malicious, external, operational faults in computing systems, ie. intrusions. The term intrusion tolerance has been introduced to emphasize the need to go beyond what classical fault tolerant systems were able to offer. Unfortunately, as opposed to well understood accidental faults, the domain is still lacking sound data sets and models to offer rationales in the design of intrusion tolerant solutions. In this paper, we describe a framework similar in its spirit to so called honeyfarms but built in a way that makes its large-scale deployment easily feasible. Furthermore, it offers a very rich level of interaction with the attackers without suffering from the drawbacks of expensive high interaction systems. The system is described, a prototype is presented as well as some preliminary results that highlight the feasibility as well as the usefulness of the approach.

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. Each of these vulnerabilities can be potentially exploited by an attacker to lead a computer resource to a state of failure. However, how many of them are used in practice remains unknown. In order to assess the dependability of a computer system or network resource, it is essential to have a good understanding of the malicious threats to which the system will be exposed.

Many different projects, such as DShield [16], the Internet Motion Sensor [5], the CAIDA project [7] or the Leurré.com project [15] take advantage of different techniques to collect data and thus offer different views on the attacks observed on the Internet. However, they lack the required level of interaction with the attackers to adequately understand and characterize the observed attacks. An extremely valid instrument to collect information is the honeypot technology. The deployment of honeypots in several locations of the IP space has underlined the fact that different blocks of addresses are attacked differently [14, 9]. It is thus extremely important to have in-depth information about these threats in order to study the feasibility of characterizing the different observed segments of the Internet. Also, recent work has shown the usefulness of gathering experimental data to model and better understand the threats due to attackers [24, 29]. Alata et al. have shown the merits of using honeypots for the statistical modeling of attack processes [20, 2] and, more specifically, the merits of high interaction ones [3]. The benefits due to high interaction honeypots, i.e. real machines offered as juicy targets for attackers, are counterbalanced by several practical issues. They have to be closely monitored to make sure that they are not used as stepping stones against third parties, leading to potential liability issues. Also, one must ensure that they are not used to store illegal material while, at the same time, preserving the privacy of innocent users redirected to them. Last but not least, they are expensive in terms of resources and, for all these reasons, are not amenable for large scale deployment purposes.

In this paper, we provide a solution that addresses these issues thanks to a framework that is scalable and offers almost the same amount of information than real high interaction systems for a specific class of attacks, namely server-based code injection attacks. We are aware of the fact that they correspond only to a subset of the possible attack scenarios. However, as of today, they are considered to be responsible for the creation of large botnets [27] and the preferred propagation mechanisms of a large number of different malware. Enriching the framework for other classes of attacks is left for future work.

We propose a novel distributed honeypot deployment called SGNET. SGNET obtains the aforementioned objective exploiting the strengths of the ScriptGen technology [22, 21] and dynamically combining them with other existing solutions, namely Argos [25] and Nepenthes [4]. However, SGNET is more than a simple deployment of existing tools. SGNET offers an overlay, based on an ad-hoc HTTP-like protocol called Peiros, to coordinate these entities and seamlessly integrate them into a distributed architecture. The result of this integration is a honeypot deployment able to *automatically learn* and *handle* server-based exploits, and *emulate* the code injection attacks up to the point of the malware download. All this is achieved while keeping the honeypot sensors free of any computational intensive task and enforcing a reliable containment policy. SGNET is thus suitable for the future deployment of a number of small honeypot platforms hosted by worldwide partners on voluntary basis. The information collected by these sensors is stored in an SQL database, enabling offline sophisticated treatment by interested partners.

SGNET is the first Internet deployment taking advantage of the ScriptGen technology. In [21], the technology was validated through a set of experiments conducted in a lab. In a way, the experiments artificially created the required conditions for the ScriptGen technology to work correctly. This left a number of open questions about the behavior of the technology with a more heterogeneous and *real* data source such as the Internet. This paper aims at answering these open questions by observing the behavior of ScriptGen when handling Internet attacks.

Also, this paper presents an important extension of the ScriptGen technology as presented in previous work [21]. We will show in this paper how we have been able to extend the technology, originally conceived to learn and emulate pure network interaction, to embed in the learning phase code injection information. This allows the SGNET to automatically learn and handle unknown exploits providing an observation potential much greater than any existing knowledge-based approach.

Last but not least, the results of this real life experimental validation show the usefulness of the approach and offer sound rationales in favor of a larger scale deployment with the collaboration of interested partners willing to join.

Summarizing, the contributions of this paper are threefold. Firstly, this paper consists in the first validation of the applicability of the ScriptGen technology to the diversity of attacks observed on the Internet. Secondly, this paper presents an important advance in the emulation capabilities of the ScriptGen framework through the integration of code injection information in the learning phase and its dynamic interaction with other entities. Thirdly, it offers the reader some insight on the kind of data that such platform could offer when widely deployed.

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

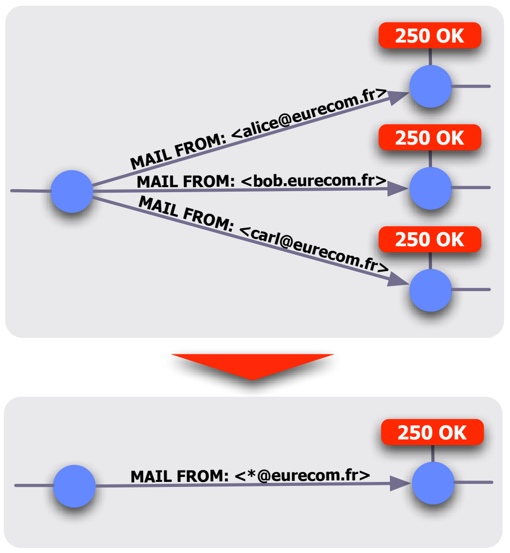
250 OK
MAIL FROM: <alice@eurecom.fr>
250 OK
MAIL FROM: <bob.eurecom.fr>
MAIL FROM: <carl@eurecom.fr>
250 OK
250 OK
MAIL FROM: <*@eurecom.fr>

Figure 1. ScriptGen FSM generalization

2 Introduction to ScriptGen

The concept of honeypot, existing in the literature since 1995 [18], was defined by Spitzner in [30] as *a network host whose value resides in being compromised by attackers*. Bailey et al. in [5] classify honeypots 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. Normally, a trade off exists between these two measures: obtaining high depth has an impact on the cost of the solution, and thus affects the achievable breadth.

The ScriptGen technology [22, 21] was created with the purpose of generating high-depth honeypots having a limited resource consumption. This is possible by *learning* the behavior of a given network protocol and represent this behavior under the form of a Finite State Machine. The generated FSM can then be used to respond to clients, emulating the behavior of the real service implementation at a very low cost.

The ScriptGen learning phase is completely protocol agnostic: no knowledge is assumed neither about the structure of the protocol, nor on its semantics. ScriptGen is thus able to learn any protocol as long as its payload is not encrypted. The ScriptGen learning takes as input a set of samples of network interaction between a client and the real implementation of a server. The core of the learning phase is the Region Analysis algorithm introduced in [22]: taking advantage of bioinformatics alignment algorithms [23], the algorithm exploits the statistical variability of the samples to identify portions of the protocol stream likely to carry a strong semantic meaning and discard the others. This enables us to rebuild a semantic abstraction as shown in Figure 1 for an excerpt of SMTP FSM.

The properties of the ScriptGen approach allow to perform a completely automated incremental learning of the activities as shown in [21]. ScriptGen-based honeypots are able to detect when a client request falls out of the current FSM knowledge (0-day attack) by simply detecting the absence of a matching transition. We showed in [21] how it is possible to react to this situation relying on a real host (an *oracle*) and proxying the conversation between the attacker and the host. The resulting conversation is stored and will lead to an FSM refinement as soon as enough samples of the same activity will be collected.

ScriptGen is able to correctly learn and emulate the exploit phase for protocols as complex as NetBIOS [21], but our early attempts to learn in a similar way the code injection behavior and represent it in the FSMs (inter-protocol dependencies [21]) proved to be insufficient. SGNET is born to address these limitations, offering a solution to the need for a more elaborate emulation of code injection attacks and for a stronger containment policy when *"executing"* injected shellcodes in order to retrieve the actual malware code itself responsible for the observed attacks.

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

The final objective of a code injection attack consists in forcing the execution of an executable code on a victim machine exploiting a vulnerable network service. Crandall et al. introduced in [10] 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 γ.

The payload that can be embedded directly in the network conversation with the vulnerable service (commonly called shellcode) is usually limited to some hundreds of bytes, or even less. It is often difficult to code in this limited amount of space complex behaviors. For this reason it is normally used to force the victim to download from a remote location a larger amount of data: the malware. In the context of this paper, we propose an extension of the original epsilon-gamma-pi model in order to differentiate the shellcode π from the downloaded malware μ.[1]

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 [32, 28], even unexperienced users can easily generate shellcodes with personalized behavior and reuse existing exploit code. This allows them to generate new combinations along all the four dimensions, weakening the correlation between them.

In order to retrieve precise information about a code-injection attack, all the four components of the epsilon-gamma-pi-mu model must then be observed. The ScriptGen approach is suitable for the learning of the exploit network interaction, offering the required level of interactivity with the client required to lead the attacker into sending code injection attacks. SGNET greatly expands this capability by coupling the ScriptGen approach with the program flow hijack detection capabilities of Argos [25] and with the shellcode emulation and malware download capabilities of Nepenthes [4].

When facing an attacker, the SGNET activity evolves through different stages, corresponding to the main phases of a network attack. SGNET distributes these phases to three different functional entities: *sensor*, *sample factory* and *shellcode handler*.

The SGNET sensor corresponds to the interface of the SGNET towards the network. The SGNET deployment aims at monitoring small sets of IPs deployed in multiple locations of the IP space, in order to characterize the heterogeneity of the activities along the Internet as observed in [14, 9]. SGNET sensors are thus low-end hosts meant to be deployed at low cost by different organizations and bound to a limited number of IPs. Taking advantage of the ScriptGen technology, the sensors are able to handle autonomously the exploit phase ϵ of attacks falling inside the FSM knowledge.

[1] This model does not cover the case of multiple download of executable files, i.e. the first downloaded executable performs the download of a second file. These more complex interactions are left for future work.

We saw in [21] the ability of the ScriptGen approach to rely on an *oracle* to handle unknown activities (i.e. 0-days) and automatically learn their behavior. The SGNET sample factory is an entity meant to provide samples of network interaction to refine the knowledge of the exploit phase provided to the sensors under the form of a FSM. Here we extend the concept of oracle by enriching the exploit information provided by the sample factory with information about the control flow hijack γ. To do so, we take advantage of a modified version of *Argos*, presented by Portokalidis et al. in [25]. Argos takes advantage of *qemu*, a fast x86 emulator [6] to implement memory tainting. Keeping track of the memory locations whose content derives from packets coming from the network, it is able to detect the moment in which this data is used in an *illegal* way. Argos was modified in order to allow the integration in the SGNET and load on demand a given honeypot profile with a suitable network configuration (same IP address, gateway, DNS servers, ... of the sensor sending the request). The profile loading and configuration is fast enough (less than 1 second) to be instantiated on the fly upon request of a sensor.

Together with new samples of protocol interaction to learn the exploit phase, the Argos-based sample factories provide information about the presence of code injections (γ) and are able to track down the position in the network stream of the first byte being executed by the guest host, corresponding to the first byte B_i of the payload π. We define here the payload π as the set of bytes following B_i in the network stream. The (in)validity of this assumption will be discussed in Section 5.2. The code injection information provided by the sample factory is integrated in the FSM learning, allowing the sensors to handle autonomously future instances of the same exploits up to the retrieval of the payload π. It is important to notice that the cost of relying on a sample factory to handle a network attack is much greater than the autonomous FSM-based operation performed by the sensors. One of the objectives of the SGNET will thus consist in trying to reduce the time spent in the learning phase, taking advantage of the collaboration of multiple distributed sensors and thus increasing the sample variability and the sample collection rate. Once a new attack has been learned and coded by a new path into the ScriptGen FSM, similar exploit instances do not need to be presented to the high interaction Argos machine but, instead, they can be autonomously handled by the ScriptGen sensor up to the point when malware has to be uploaded.

An important aspect of the SGNET sample factory is its ability to define a reliable containment policy. Memory tainting allows to detect precisely the moment in which the attacker succeeds in taking control of the virtual machine control flow. We are thus able to stop the virtualized host as soon as this event happens, blocking any opportunity to take advantage of the host as a stepping stone to attack others[2].

The final steps of the code injection attack trace are delegated to the SGNET shellcode handler. Every payload π identified by the SGNET interaction is submitted to a shellcode handler. This entity acts as an oracle to the sensors, providing information about the necessary network interaction to emulate the payload behavior and download the malware μ. The payload emulation is a too complex interaction to be represented in terms of a FSM; for instance, it would come down to represent all the possibly large file transfers with FSMs, which is clearly not an efficient way of proceeding. For this reason, differently from the sample factories, the network interaction generated by the shellcode handlers is never learnt in terms of FSM. In case of a code injection, the sensors will always rely on a shellcode handler that must thus be cheap in terms of resources. The SGNET shellcode handler is based on the Nepenthes tool [4].

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 although suffers from two restrictions: the limited vision on the exploits ϵ and the limited vision on the payloads π. Nepenthes takes advantage of a knowledge based approach to handle network attacks: the exploit and the code injection information is hardcoded in a set of *vulnerability modules*, one for each emulated exploit. Nepenthes applies then a set of heuristics to recognize the payload behavior and emulate it. The SGNET shellcode handler bypasses all vulnerability modules and directly feeds Nepenthes with the retrieved payloads π. In a way, the interaction between the SGNET sensors and SGNET sample factories generates a *generic* vulnerability module that automatically learns any exploit encountered by the sensors. This is a major contribution with respect to the previous work in Nepenthes [4] as we are getting rid of its main limitation, namely the need to develop a large number of highly specific vulnerability modules.

We still rely on the Nepenthes signatures to identify the payload behavior. However, the combination of Nepenthes with the behavior-based information provided by SGNET in presence of code injections allowed us to identify and address cases in which a successful code injection was falling outside the scope of the Nepenthes knowledge (see section 5.2).

4 The SGNET

We have implemented a SGNET prototype and deployed it in the Internet. In this Section we describe its design and implementation.

[2]Argos does not detect password brute-forcing, but this problem is addressable by a careful configuration of the guest VM

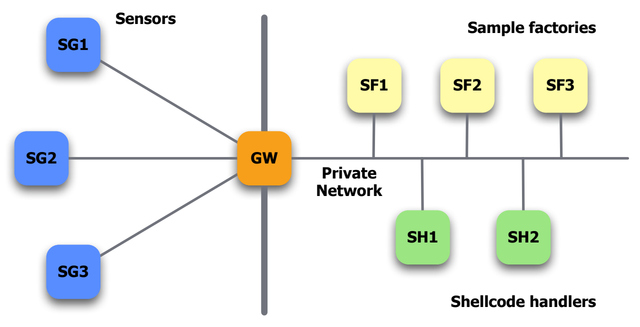

Figure 2. SGNET architecture

4.1 The architecture

The previous section introduced three main functional entities, namely sensors, sample factories and shellcode handlers. These entities are coordinated through an ad-hoc HTTP-like protocol named Peiros. The goals of this protocol are threefold.

Firstly, it must allow the sensors to send service requests to the other entities, asking for the instantiation of an oracle or submitting a payload to a shellcode handler.

Secondly, it must allow to exploit the distributed deployment of the sensors in order to maximize the statistical variability of the collected samples. That is, it must allow to share the collected samples in order to perform a centralized refinement. Also, it must allow to coordinate the distribution of the refined FSMs in order to make sure that all the deployed sensors share the same knowledge of the protocols.

Finally, it must allow to tunnel packets between the various entities, avoiding the need to modify packets endpoint addresses or modify the routers configuration in order to deliver tunneled packets to their destination. Delivering tunneled packets as application payloads over a normal TCP stream allows great flexibility and transparency in the placement of the various entities.

The SGNET architecture is shown in Figure 2. The sensors are deployed along the IP space, and due to the small resource requirements of the ScriptGen approach they can be easily deployed on low-end machines. The sample factories are deployed in a central farm on one or more high-end hosts, together with the shellcode handlers. It is very important to understand that this solution substantially differs from the classical honeyfarms where the farm is invoked for each and every attack not implemented by, for instance, a honeyd script. In SGNET the high interaction nodes (Argos machines) are only used when new attacks are observed. Upon collection of enough samples of attack activity (i.e. around 20) the ScriptGen FSMs will be refined and pushed to all the sensors, and the activity will be handled locally. This has a great impact on the dimensioning of the farm.

An additional component is introduced in the architecture, that is the *SGNET gateway*. The gateway is the core of the whole SGNET architecture. The gateway is the default home for every SGNET sensor. Each sensor on startup takes advantage of the Peiros protocol to connect to the gateway, and retrieve its own configuration. Centralizing the configuration details greatly simplifies the administrative tasks. Also, the gateway acts as an application level proxy for the Peiros protocol. SGNET sensors send service requests to the gateway, that transparently dispatches them to a free sample factory or shellcode handler. The gateway acts then as a load balancer for the various SGNET entities, using a simple round robin scheduling policy. Finally, the gateway centralizes the collection of samples generated by the various sensors and refines the ScriptGen FSMs. As soon as a new refinement is produced, the gateway is able to push the update to all the sensors, taking advantage of the Peiros protocol. All the sensors active at a given moment will thus have the same protocol knowledge, with some approximation due to network latency and retransmissions. It is important to understand that the architecture shown in Figure 2 is just a sample of composition of the different entities. The Peiros protocol can be easily extended to allow more complex configurations, for instance with multiple gateways to improve the system availability and scalability. This is left for future work.

4.2 SGNET interaction

The original 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 to be satisfactory since most of the exploits currently observable on the Internet target application-level vulnerabilities and not the TCP/IP stack. The practical experience in replaying Internet attacks such as Blaster [8] underlined the importance of preserving packet boundaries in order to correctly reproduce the attack trace. Also, preserving the TCP/IP headers allows to correctly reproduce attacks based on misuse of the TCP/IP header fields. The SGNET interaction is thus based on RAW proxying, preserving each TCP/IP packet during replay.

The SGNET interaction when handling an attacking source A interacting with a honeypot sensor H evolves through a set of phases represented in Figure 3. An attack can 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 differ-

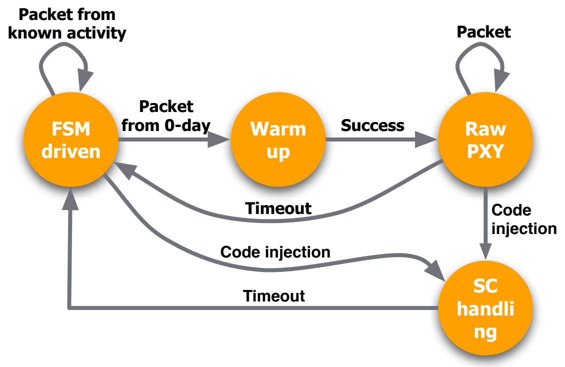

Figure 3. SGNET interaction

ent state for each couple (A, H) of attacking source A and target address H. Follows an example of interaction in the most elaborated case of a code injection falling outside the FSM knowledge:

1. The sensor handles the known interaction using the existing FSM (**FSM driven** operation). 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 H.

2. When facing an unknown request, the sensor sends a request to its Peiros endpoint (the gateway Gw) asking for an instance of sample factory and requiring the assignment of a specific network configuration to the guest OS.

3. The gateway forwards the request to a free sample factory, instantiating an oracle instance SF. When the instance is ready, the sensor is notified.

4. The sensor replays all the packets $P_1...P_i$ received from A towards the obtained sample factory SF to initialize the honeypot (**warm up** phase). The packets are tunneled over the Peiros connection and proxied by the gateway Gw, thus following the path $H \rightarrow Gw \rightarrow SF$.

5. The sensor then relies on the sample factory for continuing the conversation with A (**RAW proxying** operation). In this phase the sensor will prevent its TCP/IP stack from receiving any packet coming from the attacker A targeting IP H, dropping them using the ipqueue libraries. The attack packets $P_{i+1}, ...P_n$ will be instead pushed directly to the sample factory.

6. If a code injection is detected by Argos, the sample factory notifies the sensor through a Peiros message, providing information about the position of the payload.

7. The sensor closes the tunnel towards the sample factory, and sends a second request to its Peiros endpoint (Gw) to analyze the identified payload (**Shellcode handling** phase). The request is forwarded by the gateway to a shellcode handler SH.

8. The Nepenthes shellcode analyzer recognizes the shellcode, and notifies the sensor of the success through a Peiros message. In case of success, a tunnel ($H \rightarrow Gw \rightarrow SH$) is initialized and the Nepenthes download modules download the malware through the Peiros tunnel.

9. The gateway collects the interaction sample generated by the packets tunneled between the sensor and the sample factory and uses it to refine the FSM knowledge. If a new refined FSM is produced, it is pushed to all the active sensors.

Figure 3 shows that the SGNET lifecycle for a given attack trace may evolve through different sequences of phases. For instance, in the case in which the honeypot H is hit by a code injection already part in the FSM knowledge, the sensor will be able to retrieve the payload without requiring the instantiation of a sample factory, moving directly from the FSM driven operation to the shellcode handling phase.

5 SGNET experimental results

A prototype of the SGNET infrastructure was deployed on the Internet. The deployment has been running for a total of approximately 100 days, and has been collecting valuable information about the observed activities in an SQL database. This section addresses the validity and the potential of the proposed architecture by analyzing its behavior in the experimentation period.

Three main points are addressed here: firstly, the ability of the SGNET and the underlying ScriptGen approach to cope with the set of heterogeneous activities inherent to an Internet deployment. Secondly, the ability of SGNET to correctly handle and learn code injection attacks. Thirdly, we offer some insight into the kind of information retrieved by the deployment.

The prototype deployment is configured as follows. The gateway, the sample factories and shellcode handlers are in France. The two sensors are deployed in France and Australia. They thus represent respectively the best and worst case scenario in terms of network delays. All the sensors are associated to an unpatched Microsoft Windows 2000 machine, running the IIS services. Several open TCP and

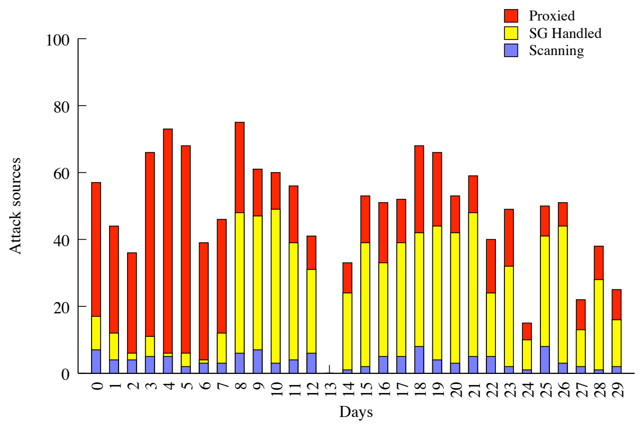

Figure 4. ScriptGen learning

UDP ports are associated to their corresponding FSM, such as TCP ports 135, 139 and 445, UDP port 137 and others.

5.1 ScriptGen and real attacks

In order to evaluate the performance of ScriptGen in learning real Internet activities, we start our experiment in a condition of 0-knowledge: the FSMs provided to ScriptGen at day 0 are emptied. ScriptGen is thus forced to handle any incoming network activity as a new activity. This allows to evaluate its capability to cope with heterogeneous and diverse samples. We focus this analysis on the first 30 days of operation[3]. This period corresponds in fact to the most interesting one from the point of view of the ScriptGen learning.

In a first approximation, we can distinguish three different kinds of activities: scanning activities, activities handled through the FSM knowledge, and activities handled through proxying. The scanning activities correspond to activities in which the source connects to one or more open ports (mainly HTTP, SMTP and HTTP, as well as some Windows-specific ports) without sending any payload, and thus without requiring any FSM knowledge to be handled. The evolution of the aforementioned classes of activities along the observation period is represented in Figure 4.

It is possible to observe from Figure 4 that the total number of attackers does not significantly change over time, showing that learning does not decrease the attractiveness of the honeypots. However there is a clear difference between the initial days and the last days of observation in terms of nature of the activity. While there is still a prevalence of proxied activities in the first week, at day 8 48 activities out of a total of 75 (64%) are handled by ScriptGen FSMs. Even at day 0, after only 8 hours of operation, ScriptGen has been able to collect enough samples to build a first path in the FSMs allowing it to handle 10 other activities of that kind appearing on that same day.

[3]Due to a technical problem, the system was not able to collect data on day 13

We monitored the evolution of the FSM size, and we saw that the acquired knowledge does not tend to explode as time goes by: after a constant growth in the first 10 days of experimentation, the FSM knowledge stabilizes to a number of 68 states, corresponding to less than 1 MB in size.

This is an important validation of the feasibility of taking advantage of the ScriptGen approach to handle Internet attacks. Figure 4 clearly shows the ability of the honeypots to handle autonomously the majority of the attacks after 10 days of unsupervised learning. More importantly, the generated knowledge is *stable*: no more paths are generated in the succeeding days. This means that the ScriptGen approach has been able to correctly generalize most of the exploit phases observed by the sensors.

On the other hand, Figure 4 underlines an important issue that could not be spotted in lab-based experimentation. During the last day of observation, day 29, we still had to proxy some of the traffic the treatment of which had not yet been successfully learned. This result can be explained considering the frequency of the various classes of network activities. The first activity being learnt by SGNET (in only 8 hours) is likely to correspond to the network activity of the Allaple worm [17]. The worm tries to bruteforce network share passwords, and thus each observed IP source generates hundreds of TCP sessions, and thus samples. Other common activities such as the Blaster propagation attempts [8] require approximately one week to be included in the FSM learning.

We can hypothesize that along with these frequently observed activities, a number of less frequent and diverse ones is present in the set. Being less frequent, the considered observation period, with only two sensors hosting a total of 4 IPs, is not sufficient to collect enough samples to perform the learning. In order to validate the hypothesis, we took advantage of the Leurré.com dataset and the clustering algorithm defined in [26]. We looked at one month of data collected by a honeypot platform deployed in the same subnet of one of the two sensors. Out of a total of 560 different clusters of similar activities, 540 of them correspond to infrequent activities, observed less than 50 times in the time span. At the same time, these 540 clusters amount to only 16% of the total observed activities. This is consistent with the SGNET observations plotted in Figure 4: in average, 21% of the activities between day 20 and day 29 resulted to be unknown to the FSM knowledge.

5.2 SGNET components interaction

In order to evaluate the SGNET ability to handle code injection attacks and integrate the additional information

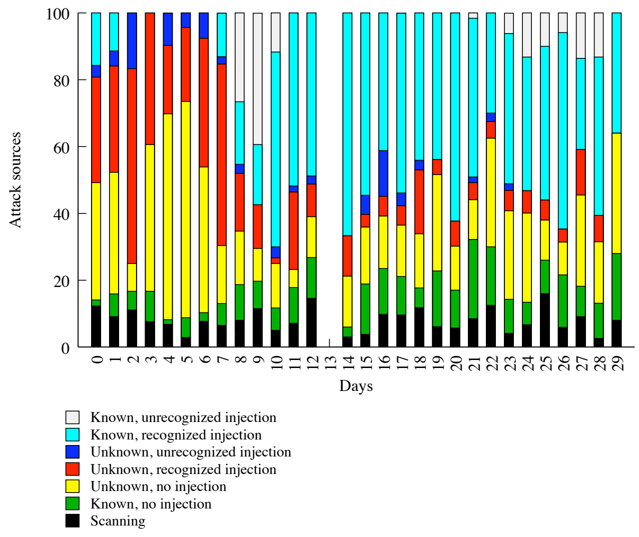

Figure 5. Handling activities

offered by the SGNET entities in the FSM knowledge, it is necessary to observe the learning from a different viewpoint, taking into consideration the interaction between the various entities. The analysis was run in the same observation period defined in the previous Section.

Figure 5 is a more detailed view on the activities shown in Figure 4. Ordered in the Figure from bottom to top, we can distinguish the following classes of activities: 1) scanning activities, connecting to open ports without sending any payload; 2) activities handled by the FSM knowledge; 3) activities handled through proxying; 4) activities handled through proxying that led to a code injection correctly recognized and emulated by Nepenthes; 5) activities handled through proxying that led to a code injection detected by Argos but not recognized by Nepenthes; 6) activities handled through the FSM knowledge able to extract a payload recognized by Nepenthes; 7) activities handled through the FSM knowledge that produced a payload not recognized by Nepenthes.

It is clear from Figure 5 that starting from day 10, an average of more than 50% of the observed activities are code injections completely handled by FSM-based operation without any need to rely on the resources of the sample factories. These activities mainly correspond to the spread of the Allaple worm (a polymorphic worm targeting ports 139/445) [17] and Blaster (port 135) [8]. After the first 14 days the amount of code injections not being handled by FSM-based operation is reduced to 5% of all activities.

It is extremely interesting to see how the set of activities identified in the previous Section for not having been learnt by ScriptGen actually corresponds to activities not leading to a code injection. Inspecting them, we found out application-level scanning activities, such as HTTP GET requests.

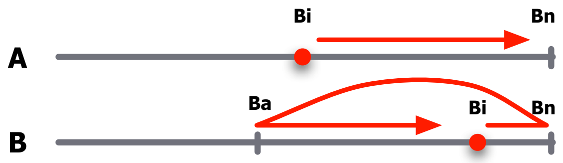

Figure 6. Finding the payload

Figure 5 validates the SGNET interaction, showing how the combination of the ScriptGen learning with the memory tainting provided by Argos is able to produce in a completely automated and unsupervised way a *generic* vulnerability module for Nepenthes.

The interaction between the behavior-based knowledge of ScriptGen FSMs with the knowledge-based Nepenthes approach becomes extremely interesting in the case of unrecognized code injections. Two main observations are worth being mentioned here.

In the beginning of the testing we ran into a considerable number of cases in which the shellcode was not recognized correctly by Nepenthes. The information provided by the Argos honeypots contains hints on the first byte B_i of payload π 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 6 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 6 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 π 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 by Nepenthes. This allows to backtrack from the initial hint given by Argos, that in these particular situations proved to be misleading. Since the payload recognition takes a very small time on the shellcode handler, the heuristic adds a minimum overhead.

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 shellcodes were not detected, catching thus our attention. After submission of the collected payload samples to the Nepenthes development team, they decided to modify their signature for one class of shellcodes (*bindfiletransfer:amberg*). 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. The modification of the signature allowed Nepenthes to handle and correctly download the recent Allaple [17] worm, that was previously hidden to Nepenthes honeypots. 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.

5.3 Degrees of freedom in the epsilon-gamma-pi-mu model

During its operation, the SGNET stores a variety of information about the observed attacks, covering all the four dimensions of the epsilon-gamma-pi-mu model. While the dimension of the deployment is at the moment insufficient to conduct a thorough analysis on this data, we can extract some preliminary information to offer to the reader a sample of its potential.

We consider here the whole 100 days of operation of the deployment. In this period, the SGNET observed 2151 code injections whose payloads π were correctly identified by the shellcode handler. Of these injections, only 532 of them successfully produced a malware sample due to failures in the malware download phase performed by Nepenthes. We are currently investigating this issue together with the Nepenthes team.

A precise and complete analysis of the various information components and their projection on the epsilon-gamma-pi-mu space is left for future work. We offer here some preliminary evaluation of the collected data.

Each traversal of a ScriptGen FSM that leads to a node marked as final point of a code injection corresponds to a combination of ϵ and γ values. We consider that different variants of γ will lead to different branches in the last step of the FSM traversal. According to this, two paths differing only in their last transition correspond to the usage of the same exploit ϵ but to different bogus control data γ.

In order to uniquely identify shellcodes containing variable parts (e.g. attacker IP address), we choose to use the combination of the protocol, port and output filename associated to the malware download phase as identifiers of π.

The malware, μ, is identified by the name returned by the F-Secure antivirus product when presented to it.

Using these definitions, we can identify a strong correlation in our data between the ϵ and γ dimension. The same does not hold for the other two dimensions, confirming the claims made in section 3. Two examples are worth being mentioned here.

Firstly, a specific (ϵ,γ) couple on port 135 TCP was hit 31 times and led to the generation of 23 different payloads π. This led to the download of different malware samples, such as “Hupigon.gen83”, “Backdoor.Win32.Vanbot.dt”. This shows that the same exploit is reused by different malware to inject different payloads in the system, forcing the victim to behave differently.

A specific (ϵ,γ,π) triplet on port 139 TCP was hit 401 times and led to the download of 6 different malware types μ, namely: 4 different variants of the previously mentioned Allaple worm (A,B,D and E), the “Backdoor.Win32.Rbot.bni” backdoor and one single sample named as “Virus.Win32.Virut.d”. As opposed to the previous case, we see here that also the payload π has been reused by the attackers to, ultimately, deploy different malware.

6 Related work

When considering the ScriptGen deployment in the SGNET architecture presented in this paper, it is possible to identify some similarities with the concept of *honeyfarm*[31]. Many different implementations exist, such as GenII Honeynets [1], Potemkin [33] and Collapsar [19]. All these approaches share the idea of running farms of virtualized hosts to handle traffic redirected from several remote locations to the centralized farm. SGNET architecture profoundly differs from these approaches with respect to two distinct characteristics.

Containment. When handling farms of virtualized hosts, a trade-off exists between the quality of the observations and the security of the system. A virtualized host can potentially be attacked and compromised, and eventually used as a stepping stone by attackers to compromise other systems. It is thus important to carefully define the policy according to which the network interaction of the farm can propagate to the Internet. For instance, it is advisable to allow the farm to perform DNS requests but at the same time it is essential to block any exploit attempt towards other hosts. The knowledge based approach used by GenII Honeynets or by Collapsar needs constant maintenance in order to avoid misjudgements in the containment decisions. Taking advantage of memory tainting techniques, SGNET employs a behavior-based containment in the detection of code injections that allows to stop the host execution as soon as an attacker successfully hijiacks the control flow.

Performance and delay. SGNET takes advantage of the ScriptGen approach to perform a *smart* selection of the traffic to be relayed to the virtualized hosts. While in normal honeyfarms all the traffic targeting the monitored addresses is constantly relayed to the virtualized hosts, in SGNET the allocation of a virtualized host to handle an activity is a *rare* event triggered by the observation of a new kind of activity. As we saw in section 5, most of the received attacks after the learning phase are handled locally by the sensors using ScriptGen FSMs. This property leads to less stringent performance and resource requirements for the farm of virtualized hosts, and also avoids impacting the delay observed by the attackers (i.e. the tunneling delay implicit in a worldwide deployment of sensors).

An important deployment worth being mentioned here is GQ [11, 13], a high interaction internet telescope. SGNET is a distributed deployment and GQ is an internet telescope. By design, they differ profoundly but they also share some similarities. GQ takes advantage of the protocol learning capabilities of RolePlayer [12] to filter out known activities and reduce the load on the telescope. Differently from SGNET, GQ relies on RolePlayer for the emulation of the whole attack trace. As explained in Section 3 and quantified in Section 5.3, we believe that the FSM model generated by ScriptGen or by RolePlayer fits only to the emulation of the exploit phase, but it is insufficient to model the complex interactions inherent in code injections. Last but not least, GQ suffers from the same containment issues seen for the honeyfarms. The interested reader will find in [21] a thorough comparison of RolePlayer and ScriptGen.

7 Conclusions

We presented in this paper a novel infrastructure to observe Internet attacks and to retrieve extremely rich information about their nature. 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 completely emulate the attack trace up to the successful retrieval of 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 overcome some of 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 experimental results showed in this paper are the result of the implementation of an initial prototype. The future deployment in different locations of the IP space of a wider SGNET deployment will allow us to gather a more detailed picture of the local threats observable in the Internet.

Acknowledgment

This work has been partially supported by the RNTR ACES project (contract number ANR05RNRT00103) and by the ReSIST Network of Excellence (contract number 026764). This work has been partially supported by the European Commissions through project FP7-ICT-216026-WOMBAT funded by the 7th framework program. The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission.

References

[1] Know your enemy: GenII honeynets. *Know Your Enemy Whitepapers*, May 2005.

[2] E. Alata, M. Dacier, Y. Deswarte, M. Kaâniche, K. Kortchinsky, V. Nicomette, V. H. Pham, and F. Pouget. CADHo: Collection and Analysis of Data from Honeypots. In *EDCC'05, 5th European Dependable Computing Conference*, Apr 2005.

[3] E. Alata, V. Nicomette, M. Kaâniche, M. Dacier, and M. Herrb. Lessons learned from the deployment of a high-interaction honeypot. In *EDCC'06, 6th European Dependable Computing Conference*, Oct 2006.

[4] 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.

[5] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The internet motion sensor: A distributed blackhole monitoring system. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, February 2005.

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

[7] Caida Project. The UCSD Network Telescope, www.caida.org, 2007.

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

[9] E. Cooke, M. Bailey, Z. M. Mao, D. Watson, F. Jahanian, and D. McPherson. Toward understanding distributed blackhole placement. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 54–64, New York, NY, USA, 2004. ACM Press.

[10] J. Crandall, S. Wu, and F. 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.

[11] W. Cui. *Automating Malware Detection by Inferring Intent*. PhD thesis, University of California, Berkeley, Fall 2006.

[12] W. Cui, R. H. Katz, and W.-t. Tan. Protocol-independent adaptive replay of application dialog. In *The 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[13] W. Cui, V. Paxson, and N. Weaver. Gq: Realizing a system to catch worms in a quarter million places. Technical report, ICSI Tech Report TR-06-004, September 2006.

[14] 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.

[15] 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.

[16] DShield. Distributed Intrusion Detection System, www.dshield.org, 2007.

[17] F-Secure. Malware information pages: Allaple.a, http://www.f-secure.com/v-descs/allaplea.shtml, December 2006.

[18] L. Halme and R. Bauer. AINT misbehaving-a taxonomy of anti-intrusion techniques. *Proceedings of the 18th National Information Systems Security Conference*, pages 163–172, 1995.

[19] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proceedings of the 13th USENIX Security Symposium*, pages 15–28, 2004.

[20] M. Kaâniche, E. Alata, V. Nicomette, Y. Deswarte, and M. Dacier. Empirical analysis and statistical modeling of attack processes based on honeypots. In *WEEDS 2006 - Workshop on empirical evaluation of dependability and security (in conjunction with the international conference on dependable systems and networks, DSN 2006)*, Jun 2006.

[21] C. Leita, M. Dacier, and F. 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] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.

[23] S. Needleman and C. 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] S. Panjwani, S. Tan, K. Jarrin, and M. Cukier. An experimental evaluation to determine if port scans are precursors to an attack. In *Conference on Dependable Systems and Networks (DSN 2005)*, pages 602–611, 2005.

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

[26] F. Pouget. *Distributed System of Honeypots Sensors: Discrimination and Correlative Analysis of Attack Processes*. PhD thesis, Institut Eurecom, 2006.

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

[28] E. Ramirez-Silva and M. Dacier. Empirical study of the impact of metasploit-related attacks in 4 years of attack traces. In *12th Annual Asian Computing Conference focusing on computer and network security (ASIAN07)*, December 2007.

[29] Ramsbrock, Berthier, and Cukier. Profiling attacker behavior following ssh compromises. In *Conference on Dependable Systems and Networks (DSN 2007)*, pages 119–124, 2007.

[30] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Welsey, Boston, 2002.

[31] L. Spitzner. Honeypot Farms, http://www.securityfocus.com/infocus/1720, August 2003.

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

[33] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

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