Server-side code injection attacks: a historical perspective

Jakob Fritz[1], Corrado Leita[1], and Michalis Polychronakis[2]

[1] Symantec Research Labs, Sophia Antipolis, France,
{jakob_fritz,corrado_leita}@symantec.com
[2] Columbia University, New York, USA, mikepo@cs.columbia.edu

Abstract. Server-side code injection attacks used to be one of the main culprits for the spread of malware. A vast amount of research has been devoted to the problem of effectively detecting and analyzing these attacks. Common belief seems to be that these attacks are now a marginal threat compared to other attack vectors such as drive-by download and targeted emails. However, information on the complexity and the evolution of the threat landscape in recent years is mostly conjectural. This paper builds upon five years of data collected by a honeypot deployment that provides a unique, long-term perspective obtained by traffic monitoring at the premises of different organizations and networks. Our contributions are twofold: first, we look at the characteristics of the threat landscape and at the major changes that have happened in the last five years; second, we observe the impact of these characteristics on the insights provided by various approaches proposed in previous research. The analysis underlines important findings that are instrumental at driving best practices and future research directions.

1 Introduction

Remote code injection attacks used to be one of the main vectors used by malware to propagate. By leveraging unpatched vulnerabilities in the increasingly large and complex software base in modern computing devices, attackers manage to divert the control flow towards code of their choice injected into the victim memory. The injected code, usually called shellcode, is normally constrained in terms of size and complexity, and is thus typically used to upload to the victim a second, larger executable file, the malware. This very simple mechanism, through different variations, has been responsible for the propagation of most modern threats and the infection with malware of home computers as well as banks, corporate networks, and even industrial control systems.

Historically, most of the remote code injection attacks used to be carried out against vulnerable network services easily reachable from the Internet without any need of user involvement. Many vulnerabilities in Windows SMB protocols, for instance, have been used for this purpose. However, server-side code injection attacks are now perceived by the community as an outdated problem. An increasing use of personal firewalls on end user machines (facilitated by the

choice of major vendors to ship their OSs with firewall services enabled by default) has decreased the effectiveness of server-side exploits at breaching security perimeters. At the same time, modern operating systems have adopted security mechanisms such as Data Execution Prevention (DEP) that render the task of successfully hijacking control flow increasingly difficult. In recent years, the propagation methods of choice have therefore shifted towards client-side vectors such as drive-by downloads, e-mail, and social engineering attacks.

This work aims at exploring this perception through a quantitative analysis, by looking at the evolution of the threat landscape in recent years and by evaluating the effectiveness of state-of-the-art detection and analysis techniques at coping with these threats. Is the detection of server-side code injection attacks a fully understood and solved problem deemed to become irrelevant in the long term, or are there still significant research or operational problems in the way we are tackling these threats? The answer to this question is particularly important when considering recent advanced threats such as Stuxnet [1] and Duqu [2]. While originally introduced in the target environment through USB sticks or email attachments, after the initial intrusion these threats needed to expand their installed base to reach the systems of interest (e.g., a SCADA engineering station to infect PLC code). This phase could not rely on user involvement and was carried out through server-side exploits, which were successful while keeping the infection mostly undetected by operators. The problem of detecting and understanding server-side exploits is therefore still a prominent one, despite the change in their role.

An analysis of the threat landscape on server-side code injection attacks is particularly challenging for a variety of reasons.

1. **Time evolution.** Most security datasets span several months. However, an understanding of global trends requires access to a stable data collection source, active and consistent in its observations across longer periods of time.
2. **IP space characterization.** Different groups have shown already in 2004 that scanning activity is not uniformly distributed across the IP space [3,4]. Former analyses focused mostly on high level attack profiles and packet volumes and have not gone as far as trying to characterize more in depth the differences in the observations. However, it is commonly believed that full visibility over server-side threats is possible only by spreading observation points across as many networks as possible, a requirement associated with high maintenance costs.
3. **Stability.** In order to compare observations and draw conclusions, the collected data needs to be stable, i.e., the data collection infrastructure needs to behave consistently throughout the observation period. Only in this case it will be possible to reliably attribute differences in the observations to changes in the threat landscape.

In this work, we build upon the outcome of the operation of an open distributed honeynet called SGNET [5]. SGNET was built with the above challenges in mind and attempts to provide an unbiased and comparable overview over the

activities in the IP space. The free partnership schema on top of which the system is built (sensors are contributed by volunteering partners on a best-effort basis) renders the dataset particularly challenging to analyze (the sensor population varies widely), but it still represents a unique and previously unexplored perspective over the IP space. We have been able in fact to reassemble a total of 5 years of network traces, accounting for a total of 31.7 million TCP flows.

Through the raw data at our disposal, we aim at tackling two core questions: i) understand the long-term trends and characteristics of the server-side exploits observable in the wild, and ii) assess the impact of these characteristics on commonly used practices for the detection and analysis of server-side exploits. Of particular interest is the analysis of the impact of long-term trends on knowledge-based approaches: we want to explore the practical feasibility of tackling real-world threats by fully relying on a priori knowledge on their characteristics. To the best of our knowledge, thanks to the unique characteristics of our dataset, this constitutes the first large scale analysis of the server-side threat landscape across the two previously mentioned dimensions: visibility over a long time span, but also visibility across different networks of the IP space. Against our expectations, we discover a diverse, challenging scenario that is tackled by different state of the art techniques with a highly diverse level of success.

2 Detecting server-side exploits

An exploit against a server-side vulnerability typically comprises one or more messages crafted to move the victim into a vulnerable state, followed by the injection and execution of shellcode. Various approaches have been used to hinder shellcode detection through obfuscation, encryption, and polymorphism [6]. Nowadays, return-oriented programming (ROP) [7] payloads represent the highest level of sophistication, as the shellcode execution (if any [8]) depends on the previous execution of code sequences that already exist in the exploited process.

When trying to collect information on server-side exploits, two main directions have been followed in the security literature. Standard intrusion detection approaches have attempted to leverage knowledge on known threats to recognize further instances of these threats in network environments [9,10]. On the other hand, researchers have tried to develop more generic approaches aiming to detect previously unknown attacks, without requiring detailed knowledge on their specificities. Honeypots and shellcode detection techniques are two prominent examples of such approaches, which respectively try to leverage two different inherent characteristics of code injection exploits: for honeypots, the lack of knowledge on the network topology and thus on the real nature of the honeypot host; for shellcode detection techniques, the need to transfer executable code to the victim to be run as a consequence of an exploit.

2.1 Honeypots

Honeypots detect attacks by following a simple paradigm: any interaction carried out with a honeypot host is suspicious, and very likely to be malicious. Two

broad honeypot categories can be identified: high interaction honeypots, where attackers interact with a full implementation of a vulnerable system, and low interaction honeypots, where attackers interact with a program that emulates a vulnerable system by means of scripts or heuristics.

Observing that the state of a honeypot has changed is far from determining how the honeypot was attacked, or from capturing the precise details of the attack. To aid analysis, systems such as Sebek [11] allow for detailed monitoring of system events and attacker actions. Still, such an approach requires an operator to manually analyze the results and manage the honeypot, which is time consuming and not without risk. Consequently, several approaches aim to automate attack detection and analysis through the identification of changes in network behavior [12] or the file system [13], and facilitate (large scale) deployment and management of honeypots [14,15,16]. Argos [17] can accurately pinpoint an exploit and its shellcode by leveraging a CPU emulator modified to include taint tracking capabilities. Instrumenting a virtual machine in such a way incurs a performance overhead prohibitive for use in production systems. Shadow honeypots [18] allow the integration of real servers and honeypots through more heavily instrumented replicas of production systems.

Despite their progress in automated shellcode detection and analysis, high interaction honeypots are often too expensive for large scale deployments. For this reason, researchers have worked on tools that simulate vulnerable services using scripts of a lower level of complexity. Honeyd [19] was the first highly customizable framework for the emulation of hosts or even entire networks. Subsequent systems incorporated (partial) protocol implementations, detailed knowledge of well-known exploits, shellcode analysis modules, and downloaders for collecting malware samples. These concepts are implemented in Nepenthes [20], its python counterpart Amun [21], and more recently Dionaea [22]. Differently from its predecessors, Dionaea implements a richer protocol stack and relies on a CPU emulator called libemu [23] for identifying any shellcode contained in an attack.

All these systems rely however on detailed knowledge about the exploitation phase. Additionally, Amun and Nepenthes rely on a set of knowledge-based heuristics for the emulation of shellcode: they are able to correctly handle only those decryptors and payloads that are implemented in their shellcode emulation engine. The coverage of these heuristics with respect to the threat landscape is so far unexplored. To benefit from the simplicity of low interaction techniques and the richness of high interaction honeypots, a number of hybrid approaches have been proposed. Among them is GQ [24], an Internet telescope that combines high-interaction systems with protocol learning techniques, and SGNET [5,25] which also leverages protocol learning techniques to monitor server-side exploits by means of a network of low-complexity sensors (used in this work).

2.2 Shellcode detection

Shellcode detection approaches focus on detecting the presence of malicious machine code in arbitrary streams. Initial approaches focused on creating signatures that match specific shellcode features such as NOP sleds or suspicious system

call arguments. However, machine instructions can be obfuscated quite easily, rendering signature-based approaches ineffective [26,27], while the code can be adjusted to thwart statistical approaches [28,29,30]. Despite this fact, a set of static signatures for the identification of common shellcode parts is still currently maintained as part of multiple Snort rulesets.

As it is not feasible to create signatures for the myriad of different shellcode instances by hand, several approaches have been proposed for automated signature generation based on invariants extracted from groups of related network flows [31,32,33]. However, automatic signature generation requires a minimum number of attacks to work and has difficulties in dealing with polymorphic shellcode [34]. To counter polymorphic worms, Polygraph [35], PAYL [36], PADS [12], and Hamsa [37] attempt to capture (sequences of) invariants or statistically model byte distributions of exploits and polymorphic payloads. However these are themselves vulnerable to attacks that mimic normal traffic [38,39]. An alternative approach to signature matching is vulnerability-based signatures, which focus on matching invariants that are necessary for successful exploitation, instead of implementation-specific exploit patterns [40,41].

Given the limitations of signature-based approaches in the face of zero-day attacks and evasion techniques, several research efforts turned to the detection of shellcode through static analysis. Initial approaches focused on detecting the NOP sled component [42,43], while later work attempted to detect sequences ending with system calls [44], or focused on the analysis of control flow graphs generated through static analysis [45,46,47,48].

Unfortunately, code obfuscation even in its simplest form can prevent code disassembly from being effective, and obtaining the unobfuscated shellcode entails some form of dynamic analysis. Both nemu [49,50] and libemu [23] implement a x86 cpu emulator for performing dynamic analysis of shellcode. Both approaches utilize getPC heuristics to identify potential offsets in strings to start execution from. However, where nemu attempts to identify polymorphic shellcode by combining the getPC heuristics with detection of self-references during the encryption phase, libemu focuses on the execution of the entire shellcode in a minimalistic environment which allows (emulated) execution of system calls. Both approaches allow the generation of understanding on the payload behavior: nemu is able to identify the plaintext payload generated by the decryption loop [51]; libemu instead fully executes the shellcode, including the payload, and allows the identification of the executed system calls. An alternative high-performance implementation is adopted by ShellOS [52], which uses a separate virtual machine to monitor and analyze the memory buffers of a virtual machine.

3 Dataset

Our analysis is based on an extensive data set of server-side attacks collected by the SGNET distributed honeypot deployment [5,25] over a period ranging from the 12th of September 2007 until the 12th of September 2012, i.e., exactly 5 years.

3.1 Raw data

SGNET is an initiative open to any institution willing to access the collected data, where partners interested in participating are required to contribute by hosting a honeypot at the premises of their network.

SGNET is a hybrid system that combines high interaction techniques (Argos [17]) with protocol learning techniques [53,54], allowing SGNET sensors to gradually learn to autonomously handle well-known activities. Thanks to this learning process, SGNET honeypots are capable of carrying on rich interactions with attackers without requiring a-priori knowledge of the type of exploits they will be subjected to. The implementation of the sensors has changed over the years, and their logging capabilities have changed as well. This leads to limitations in our ability to compare insights provided by the SGNET internal components, whose implementation and characteristics have changed. For instance, SGNET leveraged different versions of argos [17], a costly but very reliable technique for the identification of code injection attack by means of memory tainting. Only certain releases of SGNET stored the Argos output, and the information is thus available only on a small portion of the dataset. Despite the inability to leverage this type of information, the SGNET maintainers have decided to collect since the beginning of the project full packet traces of all the interactions observed by the active honeypots, which now amount to more than 100GB of raw data that are made available to all partners. Despite the different capabilities of the sensors in handling code injection attacks, this raw data can be used as a benchmarking platform for the analysis of the performance of different analysis and detection tools.

The SGNET project has enforced on all participants a number of rules to ensure the stability and the comparability of the observations. All sensors run a well-defined and controlled software configuration, and each sensor is always associated to 3 public IP addresses and to a well defined emulation profile. The profile of the honeypots has changed only once throughout the observation period, in February 2011, when the original emulation profile (a Microsoft Windows 2000 SP0 OS running IIS services) was upgraded to Windows XP SP0. It is clear that, as a side-effect of the partnership schema enforced by the project, the dataset at our disposal is *sparse*: the honeypot addresses do not belong to a single network block but to a variety of organizations (ISPs, academic institutions, but also industry) spread all over the world. This is a very important and rather unique property that allows us to have visibility on a variety of different segments of the IP space, and also considerably reduces the concerns associated to the detectability of the honeypots, and the representativeness of the data it collected. Each sensor is associated to only three, often non-contiguous, IP addresses in a monitored network. Differently from larger honeynets, creating a list of the addresses monitored by SGNET is an extremely costly action that to the best of our knowledge was never carried out so far. The sparsity of the observations also introduces important challenges in the analysis. SGNET honeypots are in fact deployed on a voluntary basis, and this causes significant fluctuations in the number of active honeypots throughout the deployment lifetime. Over

Detector name	Description
snort	Flags flows as attacks whenever any exploit-specific alert is raised by Snort.
snort-shellcode	Flags flows as attacks whenever any generic shellcode-detection alert is raised by Snort.
snort-et	Flags flows as attacks whenever any exploit-specific alert is raised by Snort using the Emerging Threats (ETPro) ruleset.
snort-et-shellcode	Flags flows as attacks whenever any generic shellcode detection alert is rasied by Snort using the Emerging Threats (ETPro) ruleset.
amun	Static heuristics for the detection of common packers and payloads used in the Amun honeypot.
libemu	Used in this paper to flag flows as attacks by means of a set of getPC heuristics.
nemu	Flags flows as attacks when a polymorphic shellcode is detected, or a plaintext payload matching certain heuristics.

Table 1. Summary of the detection methods considered in the paper.

these five years, the deployment varies from a total of 10 active sensors to a maximum of 71, achieved in 2010. In general, as we will see in Figure 3, the achieved coverage of the IP space varies significantly. This variability needs to be taken carefully into account throughout the analysis.

3.2 Identifying exploits

Among the different exploit detection techniques proposed in the literature, we have chosen to focus on three classes of approaches that are used in operational environments and that are suitable to offline analysis of captured traces. The three classes are associated with a different level of sophistication and reliance on a-priori knowledge, as summarized in Table 1.

Signature-based approaches. We include in our study the two most commonly used rule sets for the Snort IDS [9]:

- The official Snort ruleset, generated by the SourceFire Vulnerability Research Team (VRT). We have used the rules version 2931 (9 October 2012).
- The ruleset provided by Emerging Threats, that is now maintained in the context of a commercial offering. While an open version of the ruleset is still available, we have been granted access to the more complete ETPro ruleset (May 2013) that was used for the experiments.

For both rulesets we have identified two classes of signatures. Some attempt to detect specific network threats, and thus incorporate detailed information on the activity being detected (e.g., a particular vulnerability being exploited through a specific service). Other signatures are instead more generic, and attempt to identify byte sequences that are inherent in the transmission of a shellcode independently from the involved protocol or vulnerability. For each ruleset, we have defined two separate detectors: a detector flagging any flow triggering one

of the generic shellcode detection signatures (with suffix *-shellcode*) and another flagging any flow triggering any of the attack-specific signatures.

Shellcode emulation heuristics. Widely used honeypot techniques such as Nepenthes [20] and its python counterpart Amun [21] use a set of heuristics to identify unencrypted payloads, as well as common decryptors. While not designed specifically for the purpose of attack identification, the shellcode identification component of these honeypots is particularly critical to their ability to collect malware: while simple exploit emulation techniques are often sufficient to collect payloads, the inability of the honeypot to correctly emulate a shellcode will render it completely blind to the associated malware variant. This is particularly relevant considering the prominent role these technologies still have nowadays in contributing fresh samples to common malware repositories.

CPU emulators. Finally, we have included in the study two widely known CPU emulation approaches for the detection of shellcode, namely libemu [23] (used in the Dionaea [22] honeypot) and nemu [49]. We have used libemu in its most common configuration, which uses heuristics for the identification of getPC code to detect the presence of a valid shellcode. The approach followed by nemu is instead more sophisticated and applies runtime execution heuristics that match certain instructions, memory accesses, and other machine-level operations. Nemu has been extended to also detect plain, non-self-decrypting shellcode using a set of heuristics that match inherent operations of different plain shellcode types, such as the process of resolving the base address of a DLL through the Process Environment Block (PEB) or the Thread Information Block (TIB) [51].

For each detected shellcode, Nemu generates a detailed trace of all executed instructions and accessed memory locations. For self-decrypting shellcodes, we extracted the decryption routine from the execution trace by identifying the seeding instruction of the GetPC code (usually a `call` or `fstenv` instruction), which stores the program counter in a memory location. Nemu also identifies the execution of loops, so we consider the branch instruction of the loop that iterates through the encrypted payload as the final instruction of the decryptor. To account for variations in the operand values of the decryptor's instructions, e.g., due to different encryption keys, shellcode lengths, and memory locations, we categorize each decryptor implementation by considering its sequence of instruction opcodes, without their operands [55].

We have chosen to exclude from the analysis the identification of ROP payloads [56] and ShellOS [52]. ROP attack detection requires detailed assumptions on the configuration and runtime memory of the targeted application. Similarly, ShellOS is not particularly suitable for offline analysis as it requires replaying the collected traffic against an instrumented virtualization environment.

4 A historical perspective

The five years of data at our disposal allow us to step back, and critically look at the evolution of the threat landscape and the impact of its changes on the tools at our disposal. How is the threat landscape structured across the IP space, and

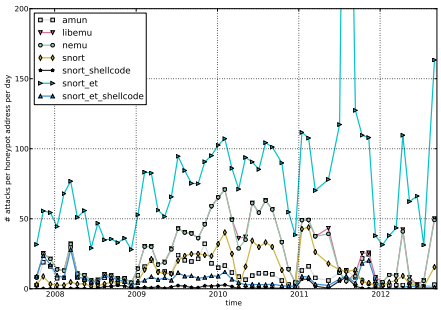

Fig. 1. Attacks per day, per honeypot, detected by different tools.

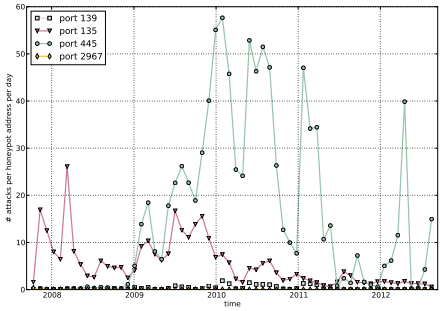

Fig. 2. Attacks per day, per honeypot, for different targeted ports.

how has it evolved over the years? What is the impact of this evolution on the different intrusion detection practices?

Figure 1 graphically represents the information at our disposal. Each of the tools introduced in the previous section has flagged a certain amount of flows as "attacks." In order to take into account the varying number of sensors, we have normalized the number of observed events with the total number of honeypot sensors known to be active in a specific day. For better readability of the graph, we have sampled the daily observations into monthly averages. The *snort-et* detector is particularly noisy due to its inherent characteristics: intrusion detection systems go beyond the detection of code injection attacks and focus also on other threats. For instance, the spike observable in July 2011 is associated to a large amount of SSH scan activities generated by a misconfigured sensor. But even factoring these differences, we can see a significant variance in the number of flows identified by the various detectors, and only *libemu* and *nemu* almost perfectly overlap in the number of detected attacks.

Figure 2 shows the distribution across time of the ports receiving the highest attack volume. Not surprisingly, the three ports with the highest volume are the typical Windows ports (445, 139, 135). However, their distribution over time has changed significantly. Back in 2008, most of the observed attacks were against the DCE/RPC locator service. While this type of exploits has only slightly diminished over the years, it has been overtaken in 2009 by a much higher attack load on the Microsoft-DS port (445). Exploits against port 2967 (only 53 sessions) have been observed only for a few weeks in 2008, but have never been observed since then. We have no reason to believe that these trends can be associated to any kind of change in the level of attack sophistication; rather, these trends directly reflect the evolution of the vulnerability surface for the different services over the years. Attacks leveraging vulnerabilities that were left unpatched by the largest group of users are those who became more successful.

This first high level picture underlines important trends in terms of attack volume. The attack volume per installed honeypot increases steadily with a major peak at the end of 2009 (which as we will see coincides with the initial

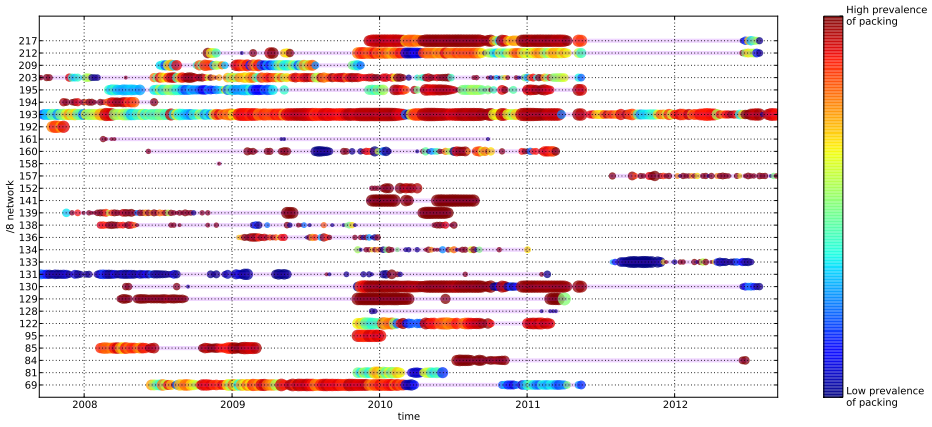

Fig. 3. Evolution of attacks observed in different \8 networks.

spread of the Conficker worm). The second half of 2011 coincides instead with an overall decrease of attack activity. A full understanding of this trend is possible only by going more in depth in the dataset and understanding the distribution of the attacks across the IP space.

4.1 Characterizing the IP space

The fact that the scanning activity across the IP space is not uniformly spread is well known, and was documented by different research groups already in 2004 [3,4]. However, due to the intrinsic difficulty associated to dispersing monitoring sensors across the Internet, previous work had leveraged low-interaction honeypots and had limited the analysis to the identification of different packet rates [4] across different networks or the identification of different high level attack profiles [3]. The information at our disposal in this paper is different: we have visibility on the complete exploitation phase on a variety of identical honeypots dispersed across the Internet.

Attack volumes. This unique perspective is shown in Figure 3, in which we have looked at the way the observed events are distributed over the different networks the SGNET deployment was monitoring. Every y-coordinate is associated to a specific \8 network monitored by one or more SGNET sensors. The size of the circles is proportional to the logarithm of the number of attacks observed in a given network on a given day. By just looking at the volume of attacks in the different networks, we can see that the their distribution is not constant over the IP space: certain sensors receive considerably more attacks on a daily basis than others. We believe this diversity in attack volume to be the culprit for the apparent decrease in attacks observed in Figures 1 and 2. In May 2011, the SGNET deployment was upgraded to a new version, but the rollout of new sensors was slowed down to tackle potential problems or bugs. As a consequence to this, the

deployment has lost visibility on several "high-volume networks" consequently lowering the average number of attacks per honeypot sensor.

Attack complexity. Figure 3 also represents using color codes the level of complexity of the observed attacks. Specifically, we have leveraged the output of *nemu* to identify the presence of a packing routine in the shellcode pushed by the attackers to the victim. Warmer colors are associated to networks in which most of the attacks observed on a daily basis leverage shellcode packing, while colder colors are associated to networks hit by simpler attacks leveraging plain shellcode. While certain networks expose a clear evolution from a lower sophistication period to a prevalence of packing, other networks are consistently characterized by solely low or high sophistication attacks. For instance, network 133.0.0.0/8 has been monitored solely in the last part of 2011 and beginning of 2012 but was consistently affected by only low-sophistication attacks in a period in which most attacks observed in other networks showed a clear predominance of shellcode packing practices.

4.2 Packers and payloads

It is clear from the high-level analysis performed so far that the practice of packing has been widely used for the distribution of shellcode, especially after 2009. In a previous work, a smaller dataset was used to analyze the prevalence of different packers [55]. The dataset at our disposal provides a wider perspective that can allow us to identify common practices and long-term trends.

As explained in Section 3.2, nemu analyzes the decryption routine of a packed shellcode, identifying loops and allowing us to categorize the decryption routines as a sequence of opcodes [55]. At the same time, the execution of the decryption routine in nemu's CPU emulator reveals the unencrypted payload. By applying heuristics inspired by those used in knowledge-based approaches such as *amun* or *nepenthes*, we can easily classify the different plaintext payloads into different types. Over the five years, we have identified a total of 37 distinct decryption loops, which is a result comparable to findings described in previous studies [55], and 15 plaintext payload implementations. Figure 4 offers a comprehensive view over all the different ways in which packers and payloads have been combined together. With packers identified by a numeric ID and payloads by an alphanumeric string, we have connected each packer and payload with an edge whenever the two were associated on a given destination port. The size of the circles is proportional to the logarithm of the number of occurrences of that packer or payload, while the width of edges is proportional to the logarithm of the number of times a packer and payload combination was observed on a given port.

Figure 4 provides a quantification to a well known scenario in the context of server-side exploits, where both payloads and packers are being freely combined together. Popular payloads such as the HTTP one have been encrypted with different packers, possibly as part of different malware implementations. Conversely, specific decryptor routines are used across multiple payloads. For instance, packer 6 has been used in conjunction with four different payloads (*Mainz bindshell 1*, *Mainz connectback 1*, *Mainz connectback 2*, *HTTP*) and was

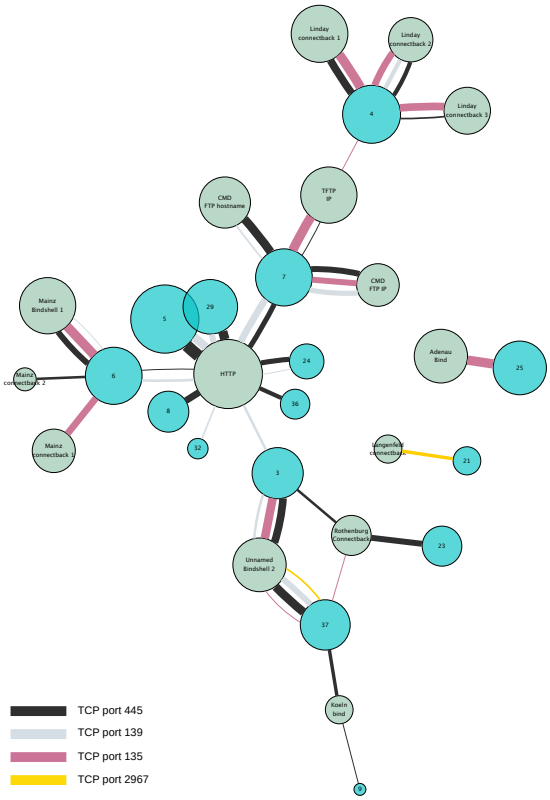

Fig. 4. Relationship between shellcode packers and the associated decrypted payloads.

possibly applied by means of a separate packing tool applied to different plain shellcode payloads. At the same time, most combinations are used across different ports, and thus completely different execution environments.

Most importantly, the association between packer, payload and vulnerable service port can be used to create an approximate definition of "activity type" that we can use to study their evolution over time. The result is shown in Figure 5, where each association of port number, payload type and packer identifier is shown evolving across the five years of data at our disposal. The size of each circle is proportional to the logarithm of the number of hits per day per honeypot address associated to that combination. The coloring is associated instead to the breadth of the activities, i.e., the percentage of currently active sensors where the specific combination was observed on that day. Cold and dark colors are associated to activities that were observed on a small number of sensors, and are therefore "more targeted." Figure 5 underlines very important facts.

Long-lived activities. Some packer-payload combinations are extremely long-lived, and span the entire five years of the dataset. This includes several old exploits against the RPC DCOM service, one of which (port 135, payload "Ade-

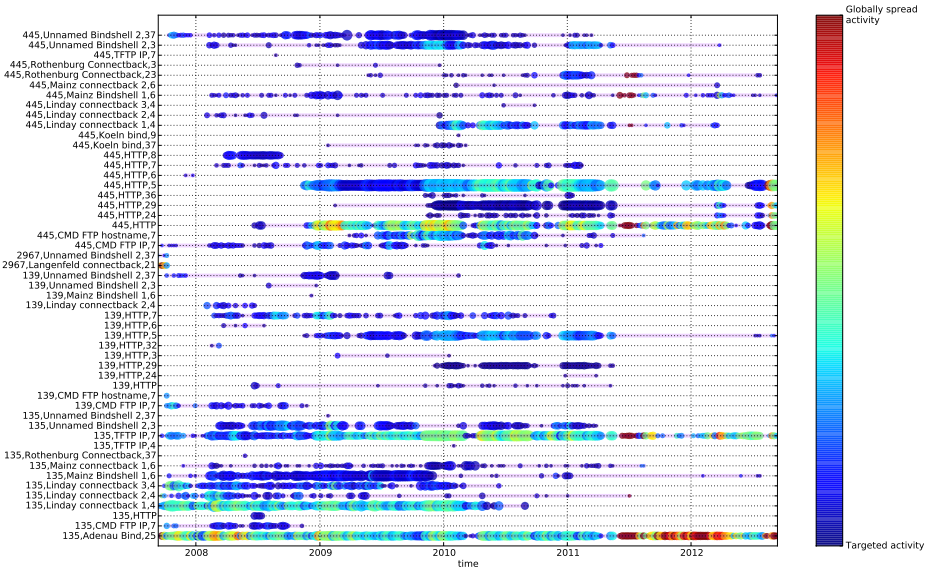

Fig. 5. Evolution of different activity types (identified by specific combinations of port,payload and packer) over time.

nau bind", packer 25) we believe to be associated to the almost 10-year-old Blaster worm. Similar considerations hold also for more recent threats: for instance, one of the most visible activities (port 445, payload "HTTP", packer 5) appears for the first time in November 2008 and persists since then, and is associated to the spread of the Conficker worm. Assuming a constant propagation strategy, the population of hosts infected by these specific malware families has not changed significantly over a very long period of time. This fact is, per se, rather alarming: little or nothing seems to have been done to reach out to infected victims, and well-known threats can survive undisturbed across years by breeding within populations of users with low security hygiene.

Targeted activities. We can identify a different type of activities in our dataset: certain cases have been observed by a limited number of sensors and for rather limited timeframes. Some packer-payload combinations have appeared for a single day, and have been observed by a single honeypot sensor. The dataset has been generated only by monitoring a few dozens of networks, and shows that the task of having a comprehensive view and understanding of these extremely short lived, sparse activities is extremely challenging. This opens important questions with respect to knowledge-based approaches to intrusion detection, and on their ability to successfully detect activities that are clearly costly to observe.

4.3 Defenses

We have pictured in the previous section a scenario that involves a combination of long-lived activities associated to old, but still active, self-propagating malware.

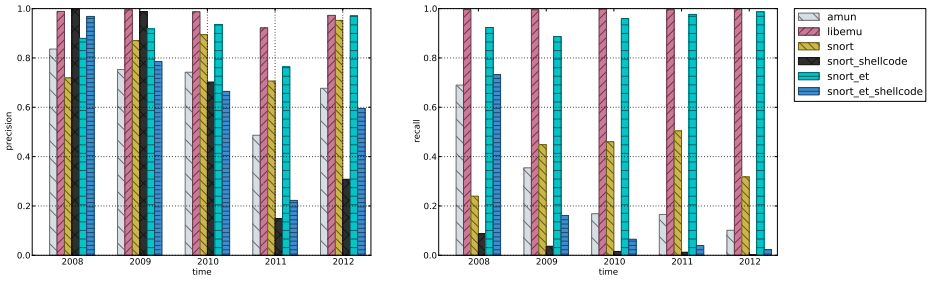

Fig. 6. Precision and recall of the detection tools using Nemu as ground truth.

Shorter, bursty activities are also present, which probably are associated with botnets, instructed by the bot herder to scan only specific ranges of the IP space for their self-propagation. This scenario is a challenging one: only by being in the right "place" at the right moment will it be possible to identify the activity. Detectors relying on a priori knowledge of all possible attack vectors are likely to face considerable challenges at dealing with these cases.

We have defined in Section 3.2 a number of different detectors characterized by varying level of complexity and reliance on knowledge of the attack vector. We range from detectors such as *snort* and *snort-et* that fully rely on such knowledge, detectors such as *snort-shellcode*, *snort-et-shellcode* that attempt static heuristics for the detection of shellcode, to *amun* that includes dynamic unpackers for common shellcodes, to *nemu* and *libemu* that leverage CPU emulation for the detection of inherent characteristics of a shellcode and avoid any assumption on the characteristics of the exploit that is injecting the shellcode itself. In order to evaluate their performance, we elect *nemu* as most generic approach for the identification of a shellcode. By not relying on sole getPC heuristics and by trying to identify self-reference, implicit in any unpacking routine, nemu is likely to be the most reliable source of information at our disposal.

We have thus evaluated all the tools performance against the nemu ground truth and computed precision and recall. Commonly used in information retrieval and classification, the precision of a tool expresses the fraction of retrieved instances that are relevant, i.e., the fraction of events flagged by a tool as malicious that are considered malicious by Argos. The recall expresses instead the fraction of relevant instances that are retrieved, i.e., the fraction of malicious instances identified by nemu that have also been identified by the tool. For a given period of time, defining t_p as the number of true positives, f_p as the number of false positives, and f_n as the number of false negatives, the precision and recall are computed as:

$$precision = \frac{t_p}{t_p + f_p} \qquad recall = \frac{t_p}{t_p + f_n} \tag{1}$$

Figure 6 shows the evolution of each tool's performance in terms of precision and recall over time. We can observe the following:

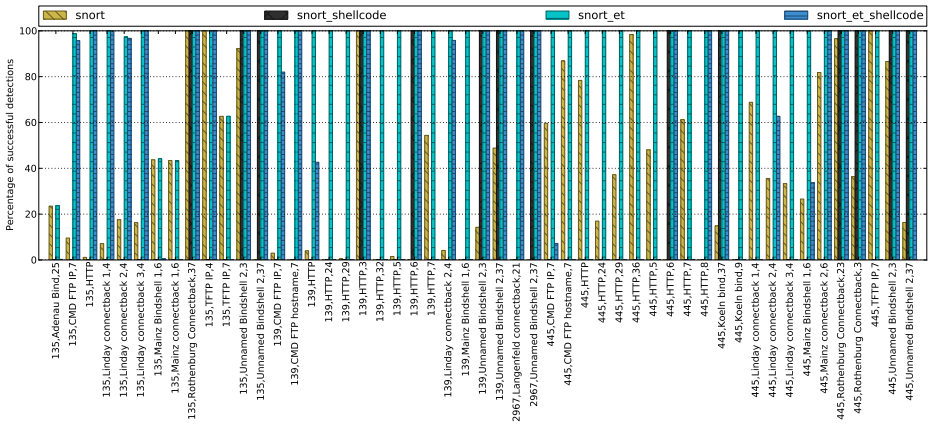

Fig. 7. Detection performance of the various tools when dealing with different combinations of packers and payloads, using nemu as ground truth.

Simple shellcode identification heuristics are unreliable. Detectors attempting to identify the presence of a shellcode in a completely static fashion (*snort-shellcode* and *snort-et-shellcode*) or through unpacking heuristics (*amun*) consistently decrease in performance across the years. From a precision standpoint, the degradation seems to be associated to an increasing false positive rate. From a recall standpoint, the heuristics leveraged by *amun* and the *snort-et-shellcode* achieved acceptable performance in 2008, detecting around 70% of the attacks, but have quickly dropped until 10% or below in recent years.

Nemu vs libemu: the importance of comparative studies. We have identified some discrepancies in the performance of the two most generic detection methodologies. Upon manual inspection, we have seen that *libemu* (which relies on the identification of getPC code and on the presence of valid x86 instructions) flagged the transfer of some executables (malware being downloaded by the honeypots) as exploits, leading to a drop in recall. However, we have also identified some cases that were correctly marked as exploits by *libemu* but were missed by *nemu*. Nemu could not correctly execute the decryption loop due to due to lack of support of a CPU instruction in the emulator code.[3]

The cost of knowledge. We observe a surprising difference between the two knowledge-based approaches, namely *snort* and *snort-et*. In both cases, it is difficult to reason about precision: given the nature of the dataset, we expect a considerable amount of network traffic to trigger IDS alerts without constituting an exploitation attempt (as we have seen already in Figure 1). When looking at the recall, instead, we see that the *snort* detector consistently detects only around 50% of the observed exploits, confirming the community perception regarding the challenges associated with the use of knowledge-based approaches at dealing with the complexity of the threat landscape. However, the *snort-et* dataset reveals a

[3] The issue has been reported to the developers and has now been fixed.

completely different picture. The ruleset has consistently achieved a coverage of more than 90% and its performance has increased since 2010. Interestingly, 2010 also coincides with the time the commercial version of the ruleset was launched, probably with an increase in resources allocated to the collection of information on threats and to the generation of signatures. The lower recall in the years before 2010 could be conjectured as being due to a lower amount of resources devoted to the collection of intelligence in those years. These facts show that full coverage over the threat landscape is a costly, but not impossible operation: community-driven approaches can only go up to a certain point at addressing a problem whose solution requires an amount of resources achievable only by commercial entities.

Signature robustness. Figure 7 explores more in depth the recall performance of the signature-based detectors on a per-activity basis. Static shellcode detection heuristics detect a limited range of activities, but in many cases are rather consistent: for instance, both *snort-shellcode* and *snort-et-shellcode* detect all occurrences of packer 37 and packer 3 regardless of the payload or the service being exploited. This is however not true in other cases: *snort-et-shellcode* has inconsistent performance at dealing for instance with packer 4, that evades detection when combined with specific services or specific payloads. When looking at exploit detection signatures we also detect a varying degree of inconsistent behavior: the *snort* detector, and to a much lesser degree also the *snort-et* one, often flag only a percentage of an activity as malicious. This is an indication that, despite the extensive research work on the topic [35,36,12,37], the correct identification of invariants is often a manual process.

4.4 The limitations of knowledge

Figure 7 underlines an important limitation of knowledge-based approaches. The two activities associated to port 2967 have been observed at the very beginning of the dataset, and for a very limited amount of time. In that case, only shellcode detection heuristics and the *snort-et* detector have been capable of identifying a threat. Knowledge-based approaches seem to struggle at coping with stealthy or highly targeted activities.

Figure 8 delves into the correlation between the difficulty of detecting an event and its global scale. We analyze the different activity types according to the spread of the attacking population over the IP space (X axis), the spread of the victim population (i.e., the honeypots being hit, Y axis) and the average number of detectors capable of identifying the activity. Colder colors represent activities that are difficult to detect, while warm colors represent well-detectable activities. Most well-detectable activities are associated to a widely spread attacker and victim base (e.g., worm-like behavior), although we do identify a few cases where well-detectable activities involve a small number of attackers and victims. When moving away from the graph diagonal, we see how more localized, "botnet-like" activities target a very small number of sensors (small Y coordinates), while being spread across the IP space (large X coordinates).

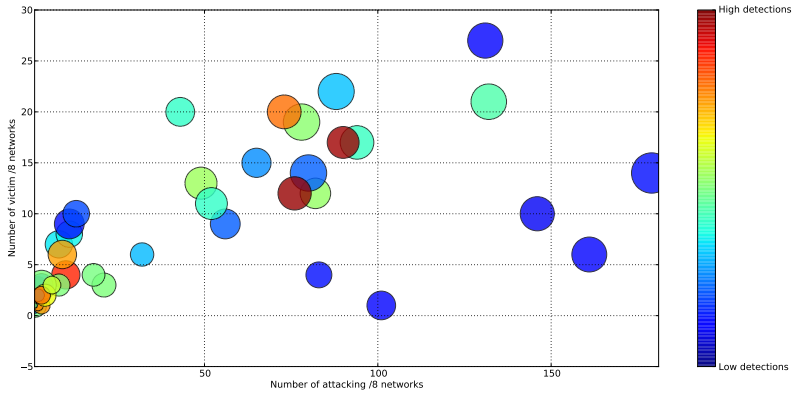

Fig. 8. Influence of the activity size on its detectability.

In general, searching for intrinsic properties of a threat instead of attempting to fully model its characteristics is a much more promising direction. Indeed, the previously mentioned activities targeting port 2967 have been detected by the shellcode heuristics for both of the rulesets under examination. However, these detectors have underlined the limitation of signatures: only a few activity types have been successfully detected, and due to the very small amount of invariants present in packed shellcode (most of the associated signatures search for very short byte patterns in the entire payload) they are prone to false positives or even squealing attacks [57]. Only sophisticated—and costly—dynamic approaches such as *nemu* and *libemu* have proven to be robust against the challenges posed by the threat landscape.

5 Conclusion

This paper has provided a comprehensive overview of the threat landscape on server-side code injection attacks. We have leveraged a privileged observation point, that of a distributed honeypot deployment that for five years has monitored a variety of networks across the IP space. The collected data, available to any institution interested in participating, has allowed us to provide a historical perspective on the characteristics of the attacks over the years, and on the performance of common state-of-the-art tools at detecting them. We have been able to substantiate with experimental data a number of key observations that should drive future work on threat monitoring and research in intrusion detection, the most important of which are the following:

Full visibility on Internet threats is difficult to achieve. Malicious activities are diverse over time and across the IP space. Different networks observe attacks of different complexity, and several threats appear as highly targeted, short-lived activities that are particularly challenging and costly to identify.

Threat persistence. In parallel to targeted, short-lived activities we can clearly identify in the dataset long-lived activities associated to well known worms and

botnets. Despite these threats being very old and well understood, we do not identify any significant decrease in their attack volume over a period of five years. This underlines an important divergence between state of the art practices and the scarce security hygiene that seems to be associated to certain user populations. Simple, known threats persist undisturbed across the years.

Limitations of knowledge. Knowledge-based intrusion detection approaches have shown clear limitations. The achievement of an acceptable visibility on the threat landscape is possible but is likely to require the investment of a non-negligible amount of resources for the creation of a comprehensive perspective on current threats. And even in such case, the generation of robust signatures for the detection of threats is hard. Server-side exploits are likely to be used more and more in the context of targeted, long-term intrusions to propagate within the target environments. The challenges observed in this work are likely to be amply amplified in these contexts. More generic—but costly—approaches seem to be the only promising research direction for the detection of these threats.

Acknowledgements

This work has been partially supported by the European Commission through project FP7-SEC-285477-CRISALIS and FP7-PEOPLE-254116-MALCODE funded by the 7th Framework Program. Michalis Polychronakis is also with FORTH-ICS. We also thank EmergingThreats for having granted us free access to the ETPro ruleset.

References

1. Symantec: W32.Stuxnet Dossier version 1.4. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf` (February 2011) [last downloaded October 2012].
2. Symantec: W32.Duqu The precursor to the next Stuxnet. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next_stuxnet_research.pdf` (November 2011) [last downloaded October 2012].
3. Dacier, M., Pouget, F., Debar, H.: Honeypots: Practical means to validate malicious fault assumptions. In: Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on, IEEE (2004) 383–388
4. Cooke, E., Bailey, M., Mao, Z., Watson, D., Jahanian, F., McPherson, D.: Toward understanding distributed blackhole placement. In: Proceedings of the 2004 ACM workshop on Rapid malcode, ACM (2004) 54–64
5. Leita, C., Dacier, M.: SGNET: a worldwide deployable framework to support the analysis of malware threat models. In: 7th European Dependable Computing Conference (EDCC 2008). (May 2008)
6. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the infeasibility of modeling polymorphic shellcode. In: Proceedings of the 14th ACM conference on Computer and communications security (CCS). (2007) 541–551
7. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and Communications Security (CCS). (2007)
8. Bennett, J., Lin, Y., Haq, T.: The Number of the Beast (2013) `http://blog.fireeye.com/research/2013/02/the-number-of-the-beast.html`.
9. Roesch, M.: Snort: Lightweight intrusion detection for networks. In: Proceedings of USENIX LISA '99. (November 1999) (software available from *http://www.snort.org/*).

10. Paxson, V.: Bro: A system for detecting network intruders in real-time. In: Proceedings of the 7th USENIX Security Symposium. (January 1998)
11. honeynet.org: Sebek (2012) `https://projects.honeynet.org/sebek/`.
12. Tang, Y., Chen, S.: Defending against internet worms: A signature-based approach. In: INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE. Volume 2., IEEE (2005) 1384–1394
13. Zhuge, J., Holz, T., Han, X., Song, C., Zou, W.: Collecting autonomous spreading malware using high-interaction honeypots. Information and Communications Security (2007) 438–451
14. Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A.C., Voelker, G.M., Savage, S.: Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In: Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP). (2005) 148–162
15. Jiang, X., Xu, D.: Collapsar: A vm-based architecture for network attack detention center. In: Proceedings of the 13^{th} USENIX Security Symposium. (2004)
16. Dagon, D., Qin, X., Gu, G., Lee, W., Grizzard, J., Levine, J., Owen, H.: Honeystat: Local worm detection using honeypots. In: Recent Advances in Intrusion Detection, Springer (2004) 39–58
17. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. SIGOPS Oper. Syst. Rev. **40**(4) (2006) 15–27
18. Anagnostakis, K.G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E.P., Keromytis, A.D.: Detecting Targeted Attacks Using Shadow Honeypots. In: Proceedings of the 14^{th} USENIX Security Symposium. (August 2005) 129–144
19. Provos, N.: Honeyd: a virtual honeypot daemon. In: 10th DFN-CERT Workshop, Hamburg, Germany. Volume 2. (2003)
20. Baecher, P., Koetter, M., Holz, T., Dornseif, M., Freiling, F.C.: The nepenthes platform: An efficient approach to collect malware. In: Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID). (2006)
21. : Amun: Python honeypot (2009) `http://amunhoney.sourceforge.net/`.
22. : Dionaea: catches bugs (2012) `http://dionaea.carnivore.it/`.
23. Baecher, P., Koetter, M.: libemu (2009) `http://libemu.carnivore.it/`.
24. Kreibich, C., Weaver, N., Kanich, C., Cui, W., Paxson, V.: [gq]: Practical containment for measuring modern malware systems. In: Proceedings of the ACM Internet Measurement Conference (IMC), Berlin, Germany (November 2011)
25. Leita, C.: SGNET: automated protocol learning for the observation of malicious threats. PhD thesis, University of Nice-Sophia Antipolis (December 2008)
26. K2: ADMmutate (2001) `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`.
27. Detristan, T., Ulenspiegel, T., Malcom, Y., Underduk, M.: Polymorphic shellcode engine using spectrum analysis. Phrack **11**(61) (August 2003)
28. Obscou: Building ia32 'unicode-proof' shellcodes. Phrack **11**(61) (August 2003)
29. Rix: Writing IA32 alphanumeric shellcodes. Phrack **11**(57) (August 2001)
30. Mason, J., Small, S., Monrose, F., MacManus, G.: English shellcode. In: Proceedings of the 16th ACM conference on Computer and communications security (CCS). (2009)
31. Kreibich, C., Crowcroft, J.: Honeycomb – creating intrusion detection signatures using honeypots. In: Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II). (November 2003)
32. Kim, H.A., Karp, B.: Autograph: Toward automated, distributed worm signature detection. In: Proceedings of the 13th USENIX Security Symposium. (2004) 271–286
33. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI). (December 2004)
34. Kolesnikov, O., Dagon, D., Lee, W.: Advanced polymorphic worms: Evading IDS by blending in with normal traffic (2004) `http://www.cc.gatech.edu/~ok/w/ok_pw.pdf`.
35. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically Generating Signatures for Polymorphic Worms. In: Proceedings of the IEEE Symposium on Security & Privacy. (May 2005) 226–241
36. Wang, K., Stolfo, S.J.: Anomalous payload-based network intrusion detection. In: Proceedings of the 7th International Symposium on Recent Advanced in Intrusion Detection

(RAID). (September 2004) 201–222
37. Li, Z., Sanghi, M., Chen, Y., Kao, M.Y., Chavez, B.: Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In: Proceedings of the IEEE Symposium on Security & Privacy. (2006) 32–47
38. Newsome, J., Karp, B., Song, D.: Paragraph: Thwarting signature learning by training maliciously. In: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2006)
39. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks. In: Proceedings of the 15^{th} USENIX Security Symposium. (2006)
40. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In: Proceedings of the ACM SIGCOMM Conference. (August 2004) 193–204
41. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: Proceedings of the IEEE Symposium on Security and Privacy. (2006)
42. Toth, T., Kruegel, C.: Accurate Buffer Overflow Detection via Abstract Payload Execution. In: Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID). (October 2002)
43. Akritidis, P., Markatos, E.P., Polychronakis, M., Anagnostakis, K.: STRIDE: Polymorphic sled detection through instruction sequence analysis. In: Proceedings of the 20^{th} IFIP International Information Security Conference (IFIP/SEC). (June 2005)
44. Andersson, S., Clark, A., Mohay, G.: Network-based buffer overflow detection by exploit code analysis. In: Proceedings of the Asia Pacific Information Technology Security Conference (AusCERT). (2004)
45. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005)
46. Payer, U., Teufl, P., Lamberger, M.: Hybrid engine for polymorphic shellcode detection. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). (July 2005) 19–31
47. Chinchani, R., Berg, E.V.D.: A fast static analysis approach to detect exploit code inside network flows. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2005)
48. Wang, X., Pan, C.C., Liu, P., Zhu, S.: Sigfree: A signature-free buffer overflow attack blocker. In: Proceedings of the USENIX Security Symposium. (August 2006)
49. Polychronakis, M., Markatos, E.P., Anagnostakis, K.G.: Network-level polymorphic shellcode detection using emulation. In: Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). (July 2006) 54–73
50. Polychronakis, M., Markatos, E.P., Anagnostakis, K.G.: Emulation-based detection of non-self-contained polymorphic shellcode. In: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2007)
51. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Comprehensive shellcode detection using runtime heuristics. In: Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC). (December 2010)
52. Snow, K.Z., Krishnan, S., Monrose, F., Provos, N.: ShellOS: Enabling fast detection and forensic analysis of code injection attacks. In: Proceedings of the 20th USENIX Security Symposium. (2011)
53. Leita, C., Mermoud, K., Dacier, M.: Scriptgen: an automated script generation tool for honeyd. In: 21st Annual Computer Security Applications Conference. (December 2005)
54. Leita, C., Dacier, M., Massicotte, F.: Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In: 9th International Symposium on Recent Advances in Intrusion Detection (RAID). (September 2006)
55. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: An empirical study of real-world polymorphic code injection attacks. In: Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET). (April 2009)
56. Polychronakis, M., Keromytis, A.D.: ROP payload detection using speculative code execution. In: Proceedings of the 6th International Conference on Malicious and Unwanted Software (MALWARE). (October 2011) 58–65
57. Patton, S., Yurcik, W., Doss, D.: An achilles heel in signature-based ids: Squealing false positives in snort. Proceedings of RAID 2001 (2001)