PolyVaccine: Protecting Web Servers against Zero-Day, Polymorphic and Metamorphic Exploits

Luis Campo-Giralte
Universidad Politecnica
de Madrid
luiscampogiralte@gmail.com

Ricardo Jimenez-Peris
Universidad Politecnica
de Madrid
rjimenez@fi.upm.es

Marta Patiño-Martinez
Universidad Politecnica
de Madrid
mpatino@fi.upm.es

Abstract

Today web servers are ubiquitous having become critical infrastructures of many organizations. However, they are still one of the most vulnerable parts of the organization infrastructure. Exploits are many times used by worms to fast propagate across the full Internet being web servers one of their main targets. New exploit techniques have arouse in the last few years that have rendered useless traditional IDS techniques based on signature identification. Exploits use polymorphism (code encryption) and metamorphism (code obfuscation) to evade detection from signature-based IDSs. In this paper, we address precisely the topic of how to protect web servers against zero-day, polymorphic, and metamorphic malware. We rely on a novel technique to detect exploits through sandbox processes that detect harmful binary code injection in HTTP requests that is more efficient than current techniques based on binary code emulation or instrumentation of virtual engines. The technique is complemented by another set of techniques such as caching, and pooling, to reduce its cost to neglectable levels. Our technique has little assumptions regarding the exploit unlike previous approaches that assume the existence of sled or getPC code, loops, read of the payload, writes to different addresses, etc. The evaluation shows that caching is highly effective and that the average latency introduced by our system is neglectable.

1 Introduction

Today web servers are ubiquitous having become critical infrastructures of many organizations. However, they are still one of the most vulnerable parts of organization infrastructure mainly due to two reasons. First, they are open to any Internet user. Second, web servers must accept sophisticated requests with high degrees of variability due to implementations not fully compliant to RFCs regulating HTTP requests that results in complex and vulnerable code. Web-based vulnerabilities account for over 25% of all vulnerabilities exploited in the period 1999-2005 [23]. Snort [27] a widely used intrusion detection system (IDS), has more than one third of the identified signatures devoted to detect web server attacks [23]. These facts make the problem of securing web servers especially challenging even more taking into account that web server attacks result in substantial financial losses.

Traditional techniques for intrusion detection systems (IDS) are based on signature identification. Signature-based detection relies on some organization (typically, an anti-malware company) to detect the exploits (they are many times used by worms to fast propagate within organizations and across the full Internet) and characterizing them via signatures. These signatures are then propagated periodically to the IDSs that can then block the characterized attacks. Some approaches aim at classifying network packets and limiting their rate or discarding them when an anomalous situation is detected. Both techniques are useless against zero-day attacks that exploit unknown vulnerabilities. Due to this fact no signature has been generated to detect them, neither any security patch for the server is available. Therefore, web servers using signature-based IDSs are vulnerable against zero-day attacks.

In the scaling arms race of computer security new exploits have aroused in the last few years having rendered useless traditional IDS techniques. New exploits are polymorphic and metamorphic evading detection from IDSs. Polymorphic attacks encrypt their payload with different keys to propagate each time as a different instance complicating their signature based identification. However, polymorphic attacks can still be captured with signatures by focusing on the decoder and any other invariant parts like sled vectors as done by Polygraph [19]. Metamorphic attacks go beyond by using metamorphisms to obfuscate code resulting in variants with no common subsequence of bytes that can be characterized by a signature.

Polymorphism and metamorphism are techniques that are complex but thanks to the availability of toolkits even hacker apprentices are able to generate polymorphic attacks. Code polymorphism has been used extensively by virus programmers to write polymorphic viruses. tPE and Mistfall [29] are some known polymorphic engines used by virus programmers. Worm writers have also started using metamorphic and polymorphic engines like ADMmutate [12], PHATBOT [25], CLET [5], and JempiScodes [26]. Garbage and NOP insertions, register shuffling, equivalent code substitution, and encryption/decryption are some of the techniques used to generate polymorphic and metamorphic malware.

Some recent techniques for network payload anomaly detection have focused on discriminating regular network traffic from malicious one by looking at the byte distribution [15, 17, 16, 35, 23, 9, 32]. However, new techniques are able to encode polymorphic attacks to reflect the same byte distribution as regular network traffic [13] therefore, they become undetectable by such techniques that rely solely on payload statistics. New techniques have been proposed to deal with polymorphic and metamorphic attacks that rely on static analysis [3, 1, 14, 4]. Static analysis techniques can be evaded by resorting to code obfuscation techniques (e.g. self-modifying or overlapping code). Dynamic analysis based on binary code emulation [6, 36, 31, 8, 21, 20] avoids the evasion techniques for static analysis. Two classes of approaches have been proposed: Emulation of network packets as binary code and full server emulation. In the first class, there are strong assumptions on the shape of the exploit such as the existence of sled or getPC code, loops, read of the payload, writes to different addresses, etc. Additionally, binary code emulation is expensive and has to be performed from all possible offsets. In the second class of approaches, the emulation of the full server execution although very accurate is very costly to be used by online systems amenable of processing high loads.

In this paper, we address the issue of how to protect web servers in an automatic way against both zero-day, polymorphic and metamorphic exploits. The technique consists in using sandbox processes that execute HTTP requests as if they were binary code. Executing code by the CPU directly is orders or magnitude more efficient than binary code emulation whilst almost as effective. However, executing from every offset is still expensive in computational terms. For this reason, we complement the technique with another suite of techniques that enable us to minimize the fraction of requests that need to be analyzed, reducing the overall cost of the analysis to neglectable levels making the technique quite affordable. Another important contribution is the relaxation of typical assumptions on the shape of exploits. Our main assumption is the execution of a system call what is required for most purposes for which attacks are performed such as worm propagation, sending spam, destroying the system, disclosing information in files, participating in a distributed denial of service attack, executing a shell, etc. Our performance evaluation shows that the average latency of requests and the web server performance is barely affected.

The paper is structured as follows. Section 2 presents related work. Section 3 presents our proposed techniques and PolyVaccine, the system implementing them. The accuracy and performance of the proposed techniques are evaluated in Section 4. We discuss the limitations of our approach in Section 5. Finally, we present our conclusions in Section 6.

2 Related Work

We can distinguish four main kinds of approaches to detect polymorphic exploits: signature based systems, anomaly detection, static analysis and dynamic analysis. **Signature-based systems** generate ”signatures” that characterize common components in exploits. Snort [27] is a popular IDS based on signature detection. Unfortunately, it is hard to keep updated the signature base with respect to the new vulnerabilities that are continuously being discovered. Additionally, signature-based systems cannot deal with polymorphic and metamorphic exploits since they cannot be characterized by signatures. Fnord [2] is a Snort plugin [27] that is able to detect mutated sleds by searching for long sequences of one-byte NOP-equivalent instructions and can also be extended to deal with multi-byte sled vectors. However, it is unable to detect advanced sleds such as trampoline sleds [1]. Shield [34] uses transport layer filters to block the traffic that exploits a known vulnerability. Shield focuses on characterizing known exploits and therefore is vulnerable to zero-day, polymorphic and metamorphic exploits. Polygraph [19] is able to generate signatures for polymorphic and metamorphic exploits, however is only able to deal with early poor obfuscation techniques and not current sophisticated code obfuscation.

Anomaly-detection systems try to characterize legitimate traffic in order to detect anomalies in the traffic that might lead to potential attacks. They monitor the payload of network packets for anomalies [15, 16, 17]. Different models for detection of HTTP attacks are proposed in [14]. PAYL [35] records the average frequency of occurrences of each byte in the payload of a normal packet. More recently, [23] tries to characterize the normal behavior of server-side components and detecting deviations from the established profile, grouping similar anomalies. Attackers can evade anomaly detection system by transforming an attack instance into another one so that an IDS is not able to recognize the attack pattern such as mutation techniques at the network [9] or application level [32]. Several mutation toolkits are available such as Whisker [22], AGENT [24], and Fragroute [28]. Mimicry attacks are another kind of attacks that evade anomaly IDSs [14, 33, 30].

Static analysis of binary code in network flows is another approach for the detection of unknown and polymorphic code. In [3] a control flow graph is built and analyzed for potential malicious activity by identifying loops, calls and jumps to absolute addresses, interrupts and RETs combined with other stack modifying instructions. STRIDE [1] looks at detecting polymorphic sleds used by buffer overflow attacks. Structural analysis of binary code to find similarities between different worm instances is performed in [14]. [4] proposes a semantic technique that aims at recognizing variants of the same malware with the same signature. Static analysis techniques can be evaded by code obfuscation techniques to avoid accurate disassembly such as use of register values, self-modifying and overlapping code, use of context outside the network message (e.g. bytes from the server code).

Dynamic analysis (aka emulation) has been proposed avoid the shortcomings of static analysis. Dynamic analysis either emulates binary code to detect potential exploits encoded in network data or emulates server execution to detect actual exploits. [36] enriches static analysis with emulation to avoid false positives. The technique aims at detecting the decryption routine of polymorphic exploits by searching GetPC code to locate the decryption routine and a decryption loop. Then, emulation is used to avoid false positives. [31] proposes a technique based on binary code emulation through all the potential offsets of the received packet. From this analysis it extracts the maximum executable length that is used it to discriminate between legitimate and malicious requests.

Vigilante [6] uses binary rewriting to emulate the server process execution that is more efficient than traditional emulation. Vigilante performs dynamic data flow analysis tracking the data flow of network data within the server. If any data from the network reaches the instruction pointer, is executed or it is passed as parameter to a critical function, then Vigilante detects an exploit and creates a filter that characterizes it to patch non-instrumented servers. Vigilante is highly accurate thanks to dynamic analysis. The tradeoff is the high cost of emulating of the full server process execution. DACODA [8] follows a similar approach to Vigilante but it is based on the emulation of the full system unlike Vigilante that only emulates the server process. DACODA claims that in this way they can capture more complex exploits involving thread and process communication, and any other kernel-mediated communication. This increased accuracy involves a much higher price due to the full system emulation. In [21] the authors follow a technique similar to ours in which each network packet is emulated from all possible offsets. They detect polymorphic exploits by looking at the execution of some form of getPC code followed by a number of payload reads performed during emulation, since the decoding needs to perform reads of the payload in order to decrypt the shellcode. [20] proposes another technique for polymorphic code that does not use getPC code, loops, and payload reads. Instead, the decryptor uses individual arithmetic instructions with absolute values to compute the decrypted shell code and push instructions to write on the heap the decrypted shellcode. They characterize this form of polymorphic exploit by detecting writes to different memory addresses and a transfer of control to one of such modified addresses.

The main differences among the aforementioned techniques and PolyVaccine are two. With respect binary code emulation, firstly, our approach is more efficient thanks to binary code execution versus emulation and the use of caching that avoids most executions. Secondly, we have a single assumption on the shape of the exploit that lies in that it performs a system call. This contrasts with binary code emulation approaches [21, 20, 36] that make stronger assumptions on the behavior of the exploit (use of getPC code, loops, payload reads, writes to different addresses, etc.). With respect full server emulation [8, 6], our approach mainly differs in that is inexpensive and provides a similar accuracy whilst full server emulation is too costly for an online system with potentially high loads.

3 PolyVaccine

PolyVaccine is concerned with threats related to binary code injection through HTTP requests and addresses zero-day, polymorphic and metamorphic exploits. PolyVaccine makes little assumptions on the shape/behavior of the exploit unlike previous work [21, 20, 36] that assumes the existence of sled or getPC code, loops, read of the payload, writes to different addresses, self-contained exploits, etc. PolyVaccine only assumes that the exploit will perform a system call.

PolyVaccine does not address non-binary code injection exploits (using script languages or SQL statements) or exploits that do not perform system calls. This is a requirement for the purpose of most exploits such as worm propagation, disclosing private information, participating in denial of service attacks, and sending spam require invoking system calls to send messages, destroy the attacked system requires invoking system calls to delete files, executing a shell requires invoking the exec system call, and so on. The only kind of attack that is not considered would be to modify the behavior of the web server to stop or modify its operation. Stopping its operation can be dealt with as a crash since without invoking system operations the attack cannot persist. Modifying its operation is extremely complex and typically will not work on different web servers with different contents. In general modifying the system operation is performed by modifying files what requires invoking system calls.

3.1 Detecting Binary Code Injection

Current accurate techniques to detect binary code injection are those based on dynamic analysis [6, 21, 31, 8, 20]. Unfortunately, they are costly because they require binary code emulation that is computationally very expensive when performed over large network flows.

Our solution aims at having the same accuracy level with lower computational cost. It is based on the use of sandbox processes where HTTP requests are executed searching for harmful binary code. A pool of processes (sandbox processes) is created with an image of the web server. The technique for detecting potentially harmful binary code injection is achieved by using one of the sandbox processes in which the HTTP request is copied into its memory. Then, control is transferred to the first byte of the HTTP request. We transfer the control by means of *Process Trace* (ptrace) that intercepts system calls. If the execution has resulted in a system call, the HTTP request is potentially harmful, since it corresponds to executable binary code that ends up invoking an operating system call. In order to provide the HTTP request with the context it would find in a real setting, the image of the sandbox processes includes the web server code, register values after request reception, and the HTTP request is injected into the corresponding buffer of the web server.

Since we do not assume that the injected code uses some form of *GetPC* code, we treat the HTTP request as a piece of code that can start execution at any point of the request. That is, there is no knowledge at which point the potentially malicious exploit will hijack the control, and therefore, control to the sandbox process is transferred iteratively to the different offsets from 0 to the length of the HTTP request. In this way, we check all the possible starting points for the injected code. Iterating the execution along all offsets is expensive. In the following section we present a caching technique that enables us to drastically reduce the number of times a request is executed.

3.2 Reducing Detection Cost

In order to reduce the cost of the detection we take advantage of the observation that most HTTP requests (typically all, when no attack is being performed) are legitimate. Additionally, HTTP requests over a particular web server have many redundancies among them when observed as a whole. The basic idea is that an HTTP request that has already been analyzed does not need to be analyzed again. Therefore, a cache has been implemented in which previously analyzed HTTP requests are cached to avoid analyzing (executing) them again. By caching requests that are legitimate it becomes possible to drastically reduce the detection cost.

One issue with HTTP requests is that sometimes they are similar (they have many fields that are the same) but not exactly the same what could hinder the effectiveness of caching. For this reason, HTTP requests are parsed and split into fields, and instead of caching whole requests, individual fields are cached.

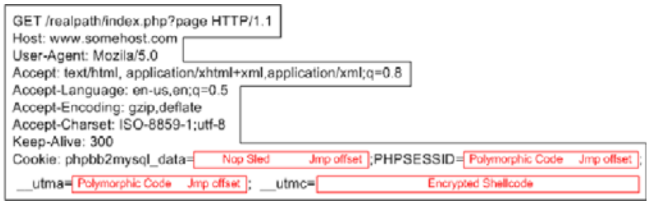
Although caching can be very effective, still creating a process for executing each iteration of the detection is too expensive. For this reason caching is complemented by means of sandbox process pooling and reuse. Basically, our detection system creates a pool of sandbox processes with the corresponding web server image. For each iteration, a sandbox process is used [1]. After injecting an HTTP request and setting the register values the control is transferred to a particular offset of the HTTP request. The sandbox process memory image is regenerated with the web server image to enable its reuse. In some exceptional cases the sandbox process can die, in which case a new one is created to maintain the pool with a minimum number of processes and avoid potential delays waiting for the sandbox process creation. In order to deal with infinite or very-long loops the execution is limited through a timeout of 2 seconds.

3.3 Dealing with Non-Cacheable Fields

Some HTTP request fields have singular values or values that are different for each client. This renders caching ineffective to deal with them. If the fields are rare, then there is no important overhead associated to them. Unfortunately, some of these fields are heavily used, such as *Cookie*. Therefore, it is necessary to deal efficiently with these fields. Cookies consist of a list of subfields. Therefore, they are like small HTTP requests that can be split into subfields. Many of these subfields are widely used and they have a particular format such as hash keys of fixed length. Other subfields have values that are repeated over requests and are therefore cacheable.

For those subfields with singular or per-client unique values with particular format (e.g. PHPIDSESS is a 32-character hexadecimal string) PolyVaccine checks their correctness in terms of length and format. If so, it is known that the subfield cannot exploit any vulnerability and therefore, there is no need to analyze it. PolyVaccine implements a simple and very robust parser (not vulnerable to overflows or other kind of binary code injection) for checking the correctness of fields and/or subfields. If a subfield passes the check, then it is harmless and simply disregarded for the detection analysis. Otherwise, it is considered potentially

[1] The pool enables parallel execution of the detection iterations what enables to exploit the power of current multi-cores. However, this optimization has not been implemented in the current system.



```
GET /realpath/index.php?page HTTP/1.1
Host: www.somehost.com
User-Agent: Mozila/5.0
Accept: text/html, application/xhtml+xml,application/xml;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1;utf-8
Keep-Alive: 300
Cookie: phpbb2mysql_data=                    ;PHPSESSID=                    ;
 __utma=                    ;  __utmc=
```

Figure 1. Cookie exploit

harmful and is analyzed to check whether it carries binary code. With this specific technique, PolyVaccine is able to deal with cookies and other uncacheable fields in an efficient way.

One could consider that cookie fields could be simply disregarded for the detection analysis since they cannot carry an exploit without the use of other fields. However, as we show in Fig. 1, a cookie can be designed to exploit a vulnerability without the need of any other field. This particular cookie, termed *poisoned cookie*, has been designed by us and has been morphed with ADMmutate toolkit. The cookie field `phpbb2mysql_data` contains the sled vector and a jump to the decryptor stored in the field PHPSESSID. The decryptor code is split between `PHPSESSID` and `_utma` fields. So, there is a jump from the last position in PHPSESSID to the beginning of the `_utma` field. The shellcode itself is encoded and stored at the `_utmc`. The decryptor, once it has decoded the shellcode, transfers the control to it.

Therefore, cookies and any other fields with singular values should be analyzed as any other field to detect potential attacks. PolyVaccine records the offsets that need to be tried for those subfields that are potentially harmful, that is, those that they are unknown or do not fulfill the length and/or format requirements. Then, the sandbox process is provided with the full contents of the HTTP request and the control is transferred iteratively to execute from all the offsets corresponding to the identified subfields.

3.4 Dealing with Non-Compliant Requests

There are a number of requests that do not fully fulfill the HTTP standard requirements [10]. In some cases, the reason is that firewalls hide information sent by clients from some organizations, such as the address and name of proxies. In some other cases, there are customizations from some browsers that do not fully comply with the RFC for HTTP requests (fields written in lower caps instead of caps). PolyVaccine has been enhanced to deal with these kinds of requests not conformant to the standard but widely accepted by web servers.

3.5 Cache Poisoning

Caching is very effective in reducing the cost of exploit detection. However, it also introduces problems that need to be addressed. Caching on a per field basis is susceptible to be poisoned by an attack customized to our system. A polymorphic attack might send first in an HTTP request a field containing the decryptor and the sled vector of the attack, but the rest of the HTTP request would be fine. Then, in a second HTTP request the whole attack would be sent containing the decryptor, the sled vector, but also the encrypted shellcode with the exploitation of the vulnerability (e.g. to provoke a buffer overflow). With the first request, the decryptor would simply not work since the shellcode is not present and no system call would be produced. Therefore, the field containing it would be considered harmless and cached. The same would happen with the sled vector. The HTTP request would be forwarded to the web server causing no harm. Upon receiving the second request, PolyVaccine would analyze the request by trying to execute it from the offsets of the fields containing new data, including the one containing the shellcode. The execution of the shellcode would not yield anything sensible, since it is encrypted, and no system call would be detected. When the request is forwarded to the web server, the exploit would take effect and now the execution of the sled vector and decryptor will result in decrypting the shellcode and executing it successfully.

This means that the cache should be protected from being poisoned. Otherwise the proposed method would expose the aforementioned vulnerability. There are different approaches in which cache poisoning can be avoided. The first one consists in having trusted clients (e.g. well-known clients used for testing the web server on a continuous basis or automatic clients that traverse the web contents periodically). Only the fields corresponding to web requests from trusted clients are cached.

The second approach to deal with cache poisoning builds upon the analysis of which cached fields with binary code can be later be composed with new fields in order to produce an exploit. The issue is that a field containing binary code that can be composed later with another binary code in other fields necessarily needs to use either indirections or obtain the program counter. This is because it will either need to jump to the binary code contained in other field or to self-modify the bytes following its field. In both cases, an indirection is needed (access to a memory position contained in a register or memory address). It turns out that the bytes corresponding to the x86 instructions with indirect addressing seldom appear in HTTP requests. A cached field can only be harmful if it contains bytes corresponding to instructions with indirections, so we can simply not cache them. Since their occurrence its low it does not have a big

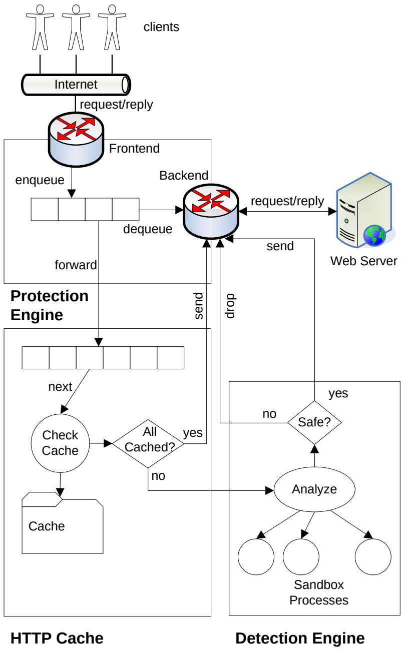


Figure 2. PolyVaccine architecture

impact in the performance of caching.

3.6 Architecture

The overall architecture of PolyVaccine is depicted in Fig. 2. The system has the following major components: the protection engine, the HTTP cache, and the detection engine.

The protection engine includes the network frontend and backend. The network frontend and backend of our system implement a push firewall authorization subsystem based on netfilter Linux architecture [11]. The network frontend stores all incoming TCP packets. All TCP packets related to one message but the last one are allowed to pass to the web server. Since it is the reception of the last packet what delivers the message from the network layer to the web server, the attack, if detected, can be avoided by simply dropping the last TCP packet of the message. The network frontend consists of a simple network queue in which every incoming TCP packet is queued awaiting to be defragmented an analyzed by the detection engine. Depending on the detection engine verdict, the packet is forwarded to the web server or dropped. The frontend is in charge of performing network packet assembly before it is processed by the cache and protection engine.

The HTTP cache is where fields from HTTP requests are stored. Every incoming request is parsed and split into fields. Each field is searched in the cache. If it is in the cache, the field is considered harmless by itself, so, the offsets corresponding to it will not be executed by the sandbox process (in the detection engine). If a field is not in the cache, then its offsets will be recorded to be executed by the detection engine. The whole HTTP request is then passed to the detection engine indicating the set of offsets that are safe (corresponding to cached fields) and the set of offsets that are not safe (corresponding to non-cached fields).

The detection engine maintains a pool of sandbox processes and receives HTTP requests together with a set of unsafe offsets that should be analyzed. For each offset to be tried, the detection engine copies the request to one of the sandbox processes (they already contain an image of the web server process) in the memory position corresponding to a web server buffer. The processes are run under the control of *Process Trace* (ptrace) utility from Linux. Ptrace enables a parent process to observe and control the execution of a child process. In particular, it enables to intercept all the operating system calls. We use this functionality to detect potentially harmful actions. In Linux, sending network messages, accessing the disk, etc. requires invoking operating system calls and this is how we detect potentially harmful activity. Then, control is transferred through ptrace to the process to start execution from the given offset. When the control is returned to the protection engine, it checks whether it finalized due to an intercepted system call or by a different reason (i.e., illegal instruction). If a system call was performed, an exploit has been detected and the HTTP request is dropped (the corresponding tcp packets). Otherwise, the HTTP request is marked as safe. The detection engine informs about the verdict to the cache and network backend. The network backend extracts the request from the queue and forwards it to the web server for being processed.

If the HTTP request is safe, to avoid poisoning the cache, those HTTP fields in the request that were not already in the cache are checked for bytes corresponding to instructions with indirections. If they contain any, then the field is not cached, otherwise is registered in the cache.

4 Evaluation

4.1 Evaluation Setup

The goal of the evaluation is to measure the overhead and the accuracy of PolyVaccine. For this purpose a series of ex-

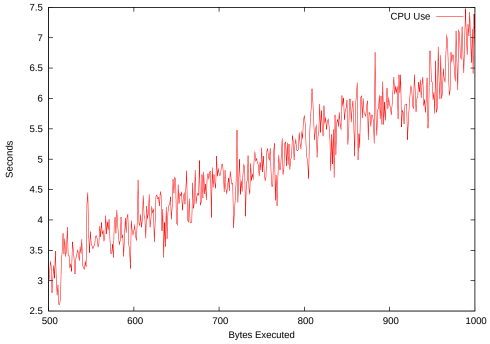


Figure 3. Execution time for unoptimized sandbox detection process from all offsets for increasing HTTP request sizes

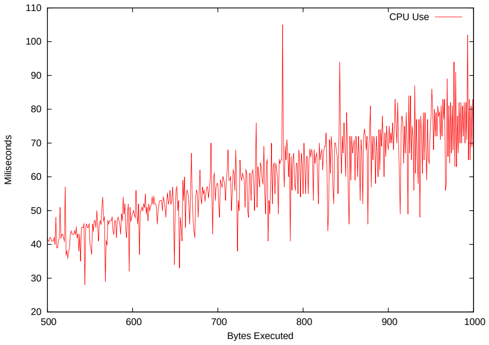


Figure 4. Execution time for optimized sandbox detection process from all offsets for increasing HTTP request sizes

periments were run including the evaluation of PolyVaccine with real traffic on one hand and in a real deployment (an Internet book seller web server). PolyVaccine was run on an Intel core duo 3.1 GHz with 2 GB of RAM and Ubuntu Hardy 8.

4.2 Detection Cost

The first experiment aims at quantifying the computational cost of our detection engine depending on the size of the HTTP request to be evaluated. Firstly, we show the detection analysis cost without optimizations. The cost incurred corresponds to the fork of the sandbox process, copy of the server code and HTTP request, and executing it with *ptrace* till it completes execution, a system call is intercepted, a unix signal is raised, or the timeout goes off. The experiment has used requests extracted from the traces of real traffic with the closest size (in excess) to the one to be evaluated and then truncated to the actual size of the experiment.

Fig. 3 depicts the results for the cost of the detection analysis. As it can be seen, the cost for executing a request from all offsets is reasonable ranging between 3 and 7 seconds for HTTP requests between 500 and 1000 bytes long. However this time is still high for an online detection system. For this reason the detection process was optimized via pooling in which the sandbox processes are reused and only the memory image is transferred (avoiding in this way process forking). Fig. 4 shows the detection times for the optimized sandbox processes. As it can be seen the optimization results in huge savings of computational cost. The range of 3-7 seconds is reduced to a range of 40-70 milliseconds, that is, a reduction of two orders of magnitude. This makes the detection process amenable of being used in an online detection system.

4.3 Detection Effectiveness

In order to evaluate the detection effectiveness we have used three kinds of exploits: non-polymorphic, polymorphic, and custom. The custom exploit is an exploit we have created that is encoded in single field, the cookie field. This exploit is specifically targeted for our system and therefore more harmful for it than any other polymorphic exploits. Table 1 summarizes the different kinds of exploits that we have used to evaluate the accuracy of PolyVaccine. In the case of polymorphic exploits we used some of the most well-known toolkits for mutating exploits such as ADMmutate [12], Clet [5] and Metasploit [18] for generating a high number of different instances of some known exploits. In the case of Metasploit we used the 12 different encoders provided with it and codify the ”peercast_url” exploit. We also used more than 25,000 worms generated by ADMmutate and 3,000 worms generated by Clet. The custom exploit is the ”poisoned cookie” we designed and explained earlier in the paper (Section 3.3). **All exploits without exception were successfully detected** by our detection engine. This means that our detection engine exhibits a high accuracy as dynamic analysis approaches do.

In order to evaluate the false positive rate we evaluated real traffic from a set of commercial web servers hosted at a large Internet provider[2]. Four traces were analyzed corresponding to different days and time intervals. Table 2 summarizes the size and time span of the traces. As it can be seen a total of 17.2 GBytes of traffic were analyzed corresponding to 21.7 hours of real traffic. **The number of false positives was null** in all the traces. This means that the technique is highly effective in avoiding false positives what is crucial for its applicability, since they lead to the rejection of legitimate HTTP requests.

[2] For anonymity reasons we cannot provide the name of the web servers nor the provider.

Toolkit	# instances
ADMmutate	26.733
CLET	3000
Metasploit	12 encoders*5 instances

Table 1. Toolkits and number of instances used

Trace size	# tcp segments	#tcp connections	observed period hours:min
948Mb	47.536	12.654	1:22
4,8 Gb	105.405	25.888	6:00
4,9 Gb	16.879	5.722	7:30
6,6 Gb	133.750	41.217	6:50

Table 2. Analyzed traffic

4.4 Profiling of Detection Finalization Causes

In this experiment we profile the different termination causes of the execution of a sandbox process. The profiling is done for the larger trace of real traffic. Fig. 5 depicts the results of the profiling. In most cases, 78%, the execution of the detection process terminates due to the execution of an invalid CPU instruction. Still, there is significant number of correct executions, 19%, despite they correspond to legitimate HTTP requests. This result is quite surprising and contradicts the existing assumption that legitimate HTTP requests would rarely yield to executable code. Finally, there is a small percentage of executions that end due to other causes such as floating exceptions, erroneous access to buses, the rest of unix signals captured by *ptrace*, and timeouts (e.g. due to infinite or very long loops). The percentage of timeouts is extremely low, below 0.003% out of all the executions. This is important since timeouts have a high cost in our system (they involve killing a sandbox process).

4.5 Cache Effectiveness

This experiment aims at measuring the effectiveness of the cache and quantifying the required size for real web traffic. We used the traces from the real web servers to perform this evaluation. Real traffic, unlike simulated traffic, can show what is the expected performance of our caching mechanism in a real deployment. In Fig. 6 we show the results for the largest traffic trace. The results for the other traces were similar so they are not shown. As it can be seen the cache failure ratio is quite low, lying between 8.5% and 9.5%. With a small cache of 50 MB, the hit ratio of the cache is already 91.5% (i.e. 100-8.5). This means that the caching is very effective and avoids the execution of a large percentage of the requests. We examined the cases where fields were not cached. It turned out that most of them could result in cache hits by doing a more intelligent string matching, what would mean that the cache hit rate can be in practice over 98%.

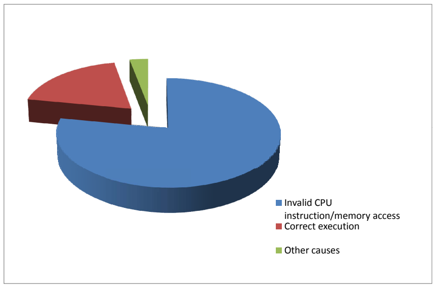


Figure 5. Distribution of the results of the execution

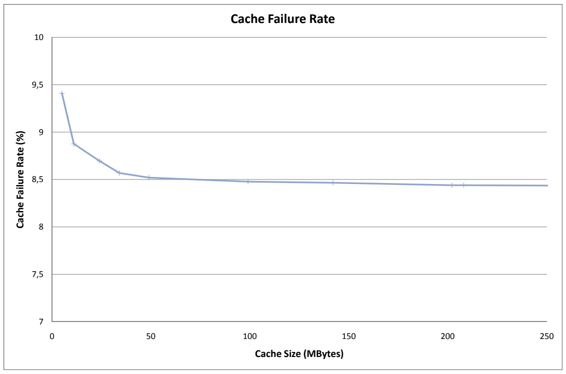


Figure 6. Cache failure ratio for increasing cache sizes

4.6 Performance Overhead in a Real Deployment

This experiment targets to measure the overhead that our detection system would introduce in a real setting. For this purpose, we have deployed a web server running TPC-W [7]. TPC-W is a transactional web benchmark performed in an Internet bookstore. We use the shopping mix workload (default workload) that consists of 20% of buy transactions

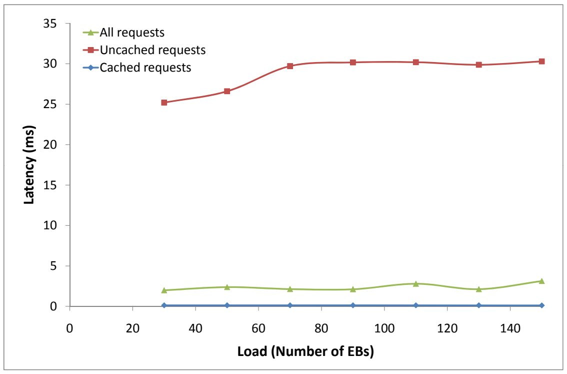


Figure 7. TPC-W Benchmark. Latency introduced by PolyVaccine.

and 80% of browse transactions. Emulated browsers (EBs) emulate users with separate connections and simulate the same HTTP network traffic as would be seen by a real customer using a browser.

In this experiment PolyVaccine runs in one site, configured as in the previous experiments, and we have used two additional sites: one for running the clients and another one for running the web server. The latter is an Intel core duo 3GHz, 1 GB of RAM, equipped with Ubuntu Hardy 8 running Apache 2.0, MySQL 5.0, and a PHP implementation of TPC-W (http://pgfoundry.org/projects/tpc-w-php). Clients, with very low computational needs, run on a Pentium 1.2 GHz, 256 Mb of RAM.

Figure 7 shows the results for increasing loads (i.e. as number of concurrent emulated browsers). The results measure the average introduced latency by our detection system. This average latency measures the time since a tcp message is defragmented (the last tcp segment has been received) till the verdict is given by the detection engine (at that point, if it is positive the tcp segment is forwarded to the web server). The curve ”all requests” shows the average latency introduced for all requests. The overhead introduced by our system is neglectable, around 22 ms, since the average response time for a TPC-W web request in our experimental setup is 6.8 seconds. We also show the results split among the requests for which all fields are cached, and the requests for which at least one field is not cached and the detection engine has to analyze the http request. The curve ”cached requests” shows the average time for the former case and the curve ”uncached requests” shows the time for the latter case. Cached requests are processed extremely fast, in 12 microseconds. Non-cached requests, since they involve running the detection analysis, are most costly, 30 ms (note that in most cases many fields of the request are cached, so not all the offsets need to be analyzed), but still neglectable when compared to the end-to-end latency of web requests.

5 Limitations

In the presented approach we have only considered the HTTP protocol in PolyVaccine. Since web-based vulnerabilities are a large fraction of all vulnerabilities exploited (25% during 1999-2005 and over 1/3 of snort filters [23]) this means that we have focused in a relevant problem. In any case the proposed system can be extended to other protocols amenable to caching, such as SOAP. PolyVaccine targets unix/linux systems based on x86 processors. Addressing the unix/linux platform for web servers is important since around 70% of the market share (according to the Nov. 2008 Netcraft web server survey, http://news.netcraft.com) corresponds to this platform. Polyvaccine can be easily extended to other CPU architectures. The only dependency on the CPU lies in the identification of CPU instructions corresponding to indirections that can be easily done for any CPU instruction set.

There is a particular kind of attacks that is not considered, namely those attacks that do not result in invoking system calls. This kind of attacks, as explained in detail in Section 3 is not as harmful as those performing system calls. Most harmful actions such as destroying part of the system, propagating to other servers, participating in a distributed denial of service attack, disclosing private information, executing a shell, propagating spam, etc. require performing system calls. An attack could just change the web pages replied to clients or stop the server operation. The former attack would be highly complex and possibly only work for a particular web server. The latter could be dealt with as a server crash since the effect of the attack will not persist a reboot (otherwise, it would have needed to modify files and therefore invoking system calls).

6 Conclusions

In this paper we have described PolyVaccine, a system for protecting web servers against day-zero, polymorphic and metamorphic exploits. PolyVaccine executes HTTP requests as if they were binary code and finds out whether the execution results in invoking system calls. PolyVaccine does not pose any other assumptions on the exploit, like the existence of GetPC code, heavy reads of the payload, etc. Its overhead, as demonstrated in the evaluation, is neglectable thanks to the effectiveness of caching and sandbox process pooling. The proposed approach is an alternative to code emulation that is computationally significantly more expensive whilst offering a similar degree of accuracy.

References

[1] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *Int. Conf. on Information Security (SEC)*, pages 375–392, 2005.

[2] F. M. architecture mutated NOP sled detector. http://www.cansecwest.com/sppfnord.c, 2002.

[3] R. Chinchani and E. van den Berg. A fast static analysis approach to detect exploit code inside network flows. In *Int. Symp. on Recent Advances in Intrusion Detection (RAID)*, pages 284–308, 2005.

[4] M. Christodorescu, S. Jha, S. Seshia, D. X. Song, and R. Bryant. Semantics-aware malware detection. In *IEEE Symp. on Security and Privacy*, pages 32–46, 2005.

[5] Clet polymorphic shellcode engine using spectrum analysis. http://www.phrack.com.

[6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP*, pages 133–147, 2005.

[7] T. P. P. Council. TPC W Benchmark, 2000.

[8] J. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM Conf. on Computer and Communications Security*, pages 235–248, 2005.

[9] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symp.*, pages 9–9, 2001.

[10] http /1.1. http://www.w3.org/protocols/rfc2616/rfc2616.html.

[11] N. http://www.netfilter.org/. http://www.netfilter.org/, 2008.

[12] K2. Admmutate. http://www.ktwo.ca/admmutate.tar.gz.

[13] O. Kolesnikov and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic. Technical report, Georgia Institute of Technology, 2004.

[14] C. Krügel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, pages 207–226, 2005.

[15] C. Krügel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *ACM Symp. on Applied Computing*, pages 201–208, 2002.

[16] M. Mahoney. Network traffic anomaly detection based on packet bytes. In *ACM Symp. on Applied Computing*, pages 346–350, 2003.

[17] M. Mahoney and P. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *ACM Int. Conf. on Knowledge Discovery and Data Mining*, pages 376–385, 2002.

[18] Metasploit project. http://www.metasploit.org.

[19] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symp. on Security and Privacy*, pages 226–241, 2005.

[20] M. Polychronakis, K. Anagnostakis, and E. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *RAID*, pages 87–106, 2007.

[21] M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, 2007.

[22] R. Puppy. A look at whisker's anti- ids tactics just how bad can we ruin a good thing? www.wiretrip.net/rfp/txt/whiskerids.html, 1999.

[23] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Symp. on Network and Distributed System Security (NDSS)*, 2006.

[24] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of nids attacks. In *Computer Security Applications Conf. (ACSAC)*, pages 28–38, 2004.

[25] SecureWorks. Phatbot trojan analysis. http://www.secureworks.com/research/threats/phatbot/.

[26] M. Sedalo. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *www.shellcode.com.ar/en/proyectos.html*.

[27] Snort. http://www.snort.org/.

[28] D. Song. Fragroute: a tcp/ip fragmenter. www.monkey.org/dugsong/fragroute, 2002.

[29] P. Szor. *The Art of Computer Virus Research and Defense. Advanced code evolution techniques and computer virus generator kits*. Symantec press, 2005.

[30] K. Tan, K. Killourhy, and R. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Recent Advances in Intrusion Detection*, pages 54–73, 2002.

[31] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002.

[32] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conf. on Computer and Communications Security*, pages 21–30, 2004.

[33] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conf. on Computer and Communications Security (CCS)*, pages 255–264, 2002.

[34] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, pages 193–204, 2004.

[35] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, pages 203–222, 2004.

[36] Q. Zhang, D. Reeves, P. Ning, and S. Purushothaman. Analyzing network traffic to detect self-decrypting exploit code. In *Symp. on Information, computer and communications security*, pages 4–12, 2007.