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

Luis Campo-Giralte, Ricardo Jimenez-Peris, Marta Patiño-Martinez
Facultad de Informatica
Universidad Politecnica de Madrid (SPAIN)
luiscampogiralte@gmail.com, {rjimenez,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 organizations 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 (new), polymorphic, and metamorphic malware embedded in data streams (requests) that target web servers. We rely on a novel technique to detect harmful binary code injection (i.e., exploits) in HTTP requests that is more efficient than current techniques based on binary code emulation or instrumentation of virtual engines. The detection of exploits is done through sandbox processes. 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.**

I. 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 [1]. Snort [2] a widely used intrusion detection system (IDS), has more than one third of the identified signatures devoted to detect web server attacks [1]. 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 rendered useless traditional IDS techniques. These 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 [3]. 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 [4] are some known polymorphic engines used by virus programmers. Worm writers have also started using metamorphic and polymorphic engines like ADMmutate [5], PHATBOT [6], CLET [7], and JempiScodes [8]. Garbage and NOP insertions, register shuffling, equivalent code substitution, and encryption/decryption are some of the techniques used to generate polymorphic and metamorphic malware.

*This research has been partially funded by the Spanish National Science Foundation (MICINN) under grant TIN2007-67353-C02, the Madrid Regional Research Council (CAM) under the AUTONOMIC project (S-0505/TIC/000285), and the European Commission under the NEXOF-RA project (FP7-216446).

Some recent techniques for network payload anomaly detection have focused on discriminating regular network traffic from malicious one by looking at the byte distribution [1], [9]–[12]. 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 [14]–[18]. Static analysis techniques can be evaded by resorting to code obfuscation techniques (e.g. self-modifying, overlapping code, obfuscated initializations, etc.) Dynamic analysis based on binary code emulation [19]–[24] 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 automatically against both zero-day, polymorphic and metamorphic exploits embedded in the input data stream (requests to the web server) that inject binary code to hijack the web server control flow. 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 II presents related work. Section III presents our proposed techniques and PolyVaccine, the system implementing them. The accuracy and performance of the proposed techniques are evaluated in Section IV. We discuss the limitations of our approach in Section V and present our conclusions in Section VI.

II. Related Work

There are 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 [2] is a popular IDS based on signatures. Unfortunately, it is hard to keep updated the signature base. Signature-based systems cannot deal with polymorphic and metamorphic exploits since they cannot be characterized by signatures. Fnord [25] is able to detect mutated sleds by searching for long sequences of one-byte NOP-equivalent instructions. However, it is unable to detect advanced sleds such as trampoline sleds [15]. Shield [26] uses transport layer filters to block the traffic that exploits a known vulnerability. Therefore, it is vulnerable to zero-day, polymorphic and metamorphic exploits. Polygraph [3] generates signatures for polymorphic and metamorphic exploits however, it is only able to deal with early poor obfuscation techniques.

Anomaly-detection systems characterize legitimate traffic in order to detect anomalies in the traffic that might lead to potential attacks. Some systems monitor the payload of network packets [9]. Others record the average frequency of occurrences of each byte in the payload of a normal packet [10]. In [1] the normal behavior of server-side components is characterized and deviations from the established profile are detected. Different models for detection of HTTP attacks are proposed in [16]. Anomaly detection systems can be evaded by transforming an attack instance into another one so that the IDS is not able to recognize, for instance using mutation techniques at the network [11] or application level [12]. Several mutation toolkits are available (e.g., Whisker [27], AGENT [28], and Fragroute [29]). Mimicry attacks also evade anomaly IDSs [16], [30].

Static analysis of binary code in the network flow is also used for the detection of unknown and polymorphic code. In [14] a control flow graph is built and analyzed for potential malicious activity (e.g., identifying loops, calls and jumps to absolute addresses, interrupts and RETs combined with other stack modifying instructions). STRIDE [15] detects polymorphic sleds used by buffer overflow attacks. Structural analysis of binary code to find similarities between different worm instances is performed in [16]. In [17] it is proposed 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 the use of register values, self-modifying and overlapping code, use of context outside the network message (e.g. bytes from the server code). STILL [18] uses static analysis to detect polymorphic and metamorphic exploits. It disassembles the input data stream and generates a control flow graph. The main assumption is that self-modifying (obfuscation) and indirect jump exploit

codes first acquire the absolute address of the payload. It also assumes that registers are initialized by means of absolute values (therefore, being vulnerable to register initialization obfuscation). Our system does not disassemble neither is based on these assumptions.

Dynamic analysis (aka emulation) either emulates binary code to detect potential exploits encoded in network data or emulates server execution to detect actual exploits. In [20] static analysis is used to locate the decryption routine and a decryption loop. Then, emulation is used to avoid false positives. Binary code emulation through all the potential offsets of the received packet is done in [21].

Vigilante [19] uses binary rewriting to emulate the server process execution. It performs dynamic data flow analysis. If any data from the network either reaches the instruction pointer, is executed or passed as parameter to a critical function, then Vigilante detects an exploit. Vigilante is highly accurate, however emulating the full server is costly. DACODA [22] emulates the full system. In [23] 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. The detection of polymorphic code in [24] is based on the fact that 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. A polymorphic exploit is detected if there are writes to different memory addresses and a transfer of control to one of such modified addresses.

PolyVaccine is able to detect efficiently zero-day, polymorphic and metamorphic exploits. The main differences among the aforementioned techniques and PolyVaccine are the following. 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: exploits perform a system call. With respect full server emulation [19], [22], our approach mainly differs in that is inexpensive and provides a similar accuracy whilst full server emulation is too costly for an online system that might have high loads.

III. 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 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 inject and activate bynary code that performs 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 is the one that modifies the behavior of the web server either stopping it or modifying its operation. Stopping its operation can be dealt with as a crash (e.g. rebooting) since without invoking system operations the attack cannot persist. Modifying web server operation it is extremely complex and typically will not work on different web servers with different contents. In general modifying the system operation is done modifying files what requires invoking system calls.

A. Detecting Binary Code Injection

Our solution aims at having a high accuracy level with low computational cost. It is based on the use of sandbox processes where HTTP requests are executed to identify 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. The monitored execution via ptrace [31] has an overhead of 50% in processes that make syscalls. However, in our case syscalls are the exception, so the main overhead is due to process switching between the tracing and traced processes. In [32] it is proposed a more performant alternative to ptrace that reduces the overhead by tracing process on the kernel thus reducing the tracing overhead. We are currently evaluating it.

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.

In PolyVaccine regular exploits (non-polymorphic ones) are detected like in Snort and other IDS. It looks for 80cd bytes that match the interruption 80 used by system calls. In our evaluation we have only found 38 segments with that byte. These occurrences correspond to POST operations.

B. Reducing Detection Cost: Caching and Pooling

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. In this way 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, creating a sandbox process for executing each iteration of the detection can be 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. Since in many executions illegal instructions are executed or memory access violations are produced, in order to avoid the death of the sandbox process, a signal handler is associated to it that handles all signals. This enables to catch the signal and resume the execution from a different offset. In order to deal with infinite or very-long loops the execution is limited by a timeout (currently 2 seconds). The sandbox process can be killed if the timeout expires, in which case, a new one is created to maintain the pool with the target number of processes and avoid potential delays waiting for the sandbox process creation.

C. 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 these fields. 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 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, that we term *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.

D. Dealing with Non-Compliant Requests

There are a number of requests that do not fully fulfill the HTTP standard requirements [33]. In some cases, the reason is that firewalls hide information sent by clients from some organizations, such as the address and name of proxies.

[1] The pool enables parallel execution of the detection iterations what enables to exploit the power of current multi-cores.

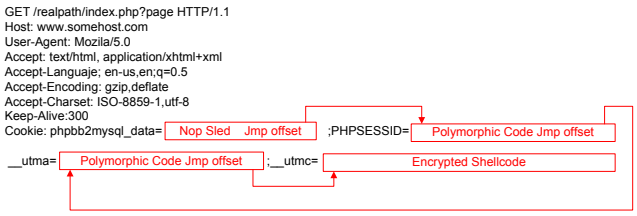

Figure 1. Cookie exploit

In some other cases, there are customizations from some browsers that do not fully comply with the RFC for HTTP requests. PolyVaccine has been enhanced to deal with these kinds of requests not conformant to the standard but widely accepted by web servers.

E. 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 first send an HTTP request with a field containing the decryptor and the sled vector of the attack, while the rest of the HTTP request is fine. Then, in a second HTTP request, the whole attack would be sent containing the decryptor, the sled vector, and the encrypted shellcode with the exploit (e.g. 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 happens 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. web spiders that periodically traverse the web contents or well-known clients). In order to keep caching effective, all requests are considered for request frequency, but only the fields corresponding to web requests that have also been sent from trusted clients are cached.

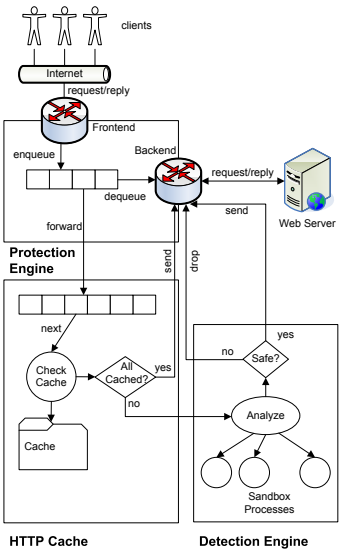

Figure 2. PolyVaccine distributed architecture

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 impact in the performance of caching.

F. Distributed Architecture

The distributed 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. Each component can be deployed on a different node. In particular, multiple nodes with instances of the detection engine can be used to parallelize the detection activity.

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 [34]. 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 triggers the delivery of 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 was 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.

G. Pseudocode of the Detection Engine

As aforementioned the detection engine is in charge of taking each request and executing it from those offsets considered unsafe (those that have not been cached). There might be multiple instances of the detection engine in different nodes to enable parallel detection processing. Each instance of the detection engine is materialized as a process with a pool of children sandbox processes. The detection process orchestrates the execution. Each sandbox process has a memory image of the web server code. When a request is received to be analyzed it has associated a set of offsets from which it should be executed. The detection engine proceeds as follows for each offset:

1) A segment is created via mmap with the following information: The codes for register initialization, the code of a jump to the corresponding offset, the memory image of the request, the code to raise a Unix signal to avoid termination of the sandbox process and enable its reuse.
2) Sets a timer for limiting the maximum execution time of the sandbox process to avoid infinite or long loops.
3) A signal handler for all Unix signals is associated to the sandbox process. Then, control is transferred to the sandbox process by means of ptrace.
4) The execution of the sandbox process is stopped either by a system call (intercepted by ptrace), by a Unix signal (due to illegal instructions, illegal memory access, etc.; to guarantee this even for successful executions, a Unix signal is raised after the request is executed), or it is killed if the timer goes off.
5) If the sandbox process execution is stopped by ptrace due to a system call, the request is considered harmful and reported to the protection engine and no more executions are made.
6) If the sandbox process execution is stopped by a signal, the signal handler catches it avoiding the death of the sandbox process, and resumes its execution from the next offset to be tried. If all offsets have been tried without a system call then the request is reported as safe to the protection engine. The web server memory image is restored in the sandbox process.
7) If the sandbox process is terminated by the watchdog, a new sandbox process is created with the corresponding web server memory image.

IV. Evaluation

A. Evaluation Setup

The goal of the evaluation is to measure the overhead and the accuracy of PolyVaccine. For this purpose a series of experiments were run including the evaluation of PolyVaccine accuracy with real traffic and PolyVaccine overhead with web server industrial benchmark (an Internet book seller). The web server was run on one node and the three

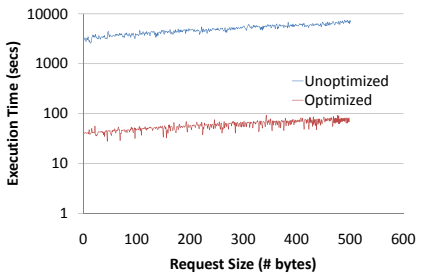

Figure 3. Execution time for unoptimized and optimized sandboxes

components of PolyVaccine were run on another node. Both nodes were an Intel core duo 3.1 GHz, 2 GB RAM running Ubuntu 8.

B. Detection Cost

The first experiment aims at quantifying the computational cost of PolyVaccine 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.

As it can be seen in Fig. 3 (top line), 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. 3 (bottom line) 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.

C. 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 I 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 [5], Clet [7] and Metasploit [35] 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 III-C). **All exploits without exception were successfully detected (100% of true positives)** by our detection engine. This means that our detection engine exhibits a high accuracy as dynamic analysis approaches do.

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

Table I
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	7:00
1,2 Gb	49.807	16.575	7:40
1,4 Gb	61.013	19.848	7:00
5,3 Gb	271.080	91.255	48:00

Table II
ANALYZED TRAFFIC

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 II summarizes the size and time span of the traces. As it can be seen a total of 25.1 GB of traffic were analyzed corresponding to 84 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.

D. 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 largest trace of real traffic. Fig. 4 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

[2]For anonymity reasons we cannot provide the name of the web servers.

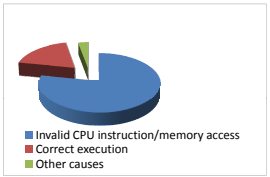

Figure 4. Distribution of the results of the execution

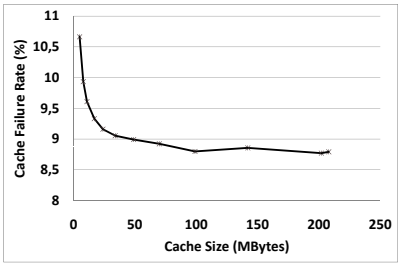

Figure 5. Cache failure ratio for increasing cache sizes

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 and then creating a new one).

E. 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. 5 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 10.75%. With a small cache of 50 MB, the hit ratio of the cache is already 91% (i.e. 100-9). 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 resulting in a cache hit rate over 98%.

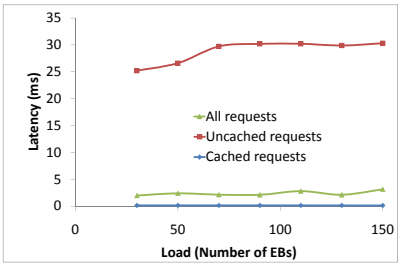

Figure 6. TPC-W Benchmark. Latency introduced by PolyVaccine.

F. 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 [36]. TPC-W is a transactional web benchmark based on an Internet bookstore. We use the shopping mix workload (default workload) that consists of 20% of buy transactions 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 on one site, configured as in the previous experiments, and we have used two additional sites: one for running the clients that inject the load and another one for running the web server. The latter is an Intel core duo 3GHz, 1 GB RAM, running Ubuntu 8, Apache 2.0, MySQL 5.0, and a PHP implementation of TPC-W (http://pgfoundry.org/projects/tpc-w-php). Clients run on a Pentium 1.2 GHz, 256 Mb RAM.

Figure 6 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.

V. 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 [1]) 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 III, 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 after a reboot (otherwise, it would have to modify files and therefore invoking system calls).

VI. 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] W. Robertson *et al.*, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *NDSS*, 2006.

[2] Snort, "http://www.snort.org/."

[3] J. Newsome *et al.*, "Polygraph: Automatically generating signatures for polymorphic worms," in *IEEE S&P*, 2005.

[4] P. Szor, *The Art of Computer Virus Research and Defense*. Symantec press, 2005.

[5] K2, "Admmutate." http://www.ktwo.ca/ADMmutate.tar.gz

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

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

[8] M. Sedalo, "$www.shellcode.com.ar/en/proyectos.html$."

[9] C. Krügel *et al.*, "Service specific anomaly detection for network intrusion detection," in *SAC*, 2002.

[10] K. Wang and S. Stolfo, "Anomalous payload-based network intrusion detection," in *RAID*, 2004, pp. 203–222.

[11] M. Handley *et al.*, "Network intrusion detection: evasion, traffic normalization, and e2e protocol semantics," in *USENIX Security*, 2001.

[12] G. Vigna *et al.*, "Testing network-based intrusion detection signatures using mutant exploits," in *ACM CCS*, 2004.

[13] O. Kolesnikov *et al.*, "Advanced polymorphic worms: Evading IDS by blending in with normal traffic," 2004.

[14] R. Chinchani *et al.*, "A fast static analysis approach to detect exploit code inside network flows," in *RAID*, 2005.

[15] P. Akritidis *et al.*, "Stride: Polymorphic sled detection through instruction sequence analysis," in *SEC*, 2005.

[16] C. Krügel *et al.*, "Polymorphic worm detection using structural information of executables," in *RAID*, 2005.

[17] M. Christodorescu *et al.*, "Semantics-aware malware detection," in *Symp. on Sec. and Priv.*, 2005.

[18] X. Wang *et al.*, "Exploit code detection via static taint and initialization analyses," in *ACSAC*, 2008.

[19] M. Costa *et al.*, "Vigilante: end-to-end containment of internet worms," in *SOSP*, 2005.

[20] Q. Zhang *et al.*, "Analyzing network traffic to detect self-decrypting exploit code," in *ASIACCS*, 2007.

[21] T. Toth and C. Krügel, "Accurate buffer overflow detection via abstract payload execution," in *RAID*, 2002, pp. 274–291.

[22] J. Crandall *et al.*, "On deriving unknown vulnerabilities from zero-day poly & metamorphic worm exploits," in *CCS*, 2005.

[23] M. Polychronakis *et al.*, "Network-level polymorphic shellcode detection using emulation," *J Comp Virology*, 2007.

[24] ——, "Emulation-based detection of non-self-contained polymorphic shellcode," in *RAID*, 2007.

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

[26] H. Wang *et al.*, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," in *SIGCOMM*, 2004.

[27] RFP, "A look at whisker's anti-ids tactics." $www.wiretrip.net/rfp/txt/whiskerids.html$

[28] S. Rubin *et al.*, "Automatic generation and analysis of nids attacks," in *ACSAC*, 2004.

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

[30] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *CCS*, 2002.

[31] ptrace, "man ptrace." http://linux.die.net/man/2/ptrace

[32] W. Yongzheng *et al.*, "A user-level framework for auditing and monitoring," in *ACSAC*, 2005.

[33] http /1.1, "$http://www.w3.org/protocols/rfc2616/rfc2616.html$."

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

[35] "Metasploit project." http://www.metasploit.org

[36] T. P. P. Council, "TPC W Benchmark," 2000. $www.tpc.org/tpcw/spec/tpcw_V17.pdf$