Commix: Detecting and exploiting command injection flaws.

Anastasios Stasinopoulos, Christoforos Ntantogian, Christos Xenakis
Department of Digital Systems, University of Piraeus
{stasinopoulos, dadoyan, xenakis}@unipi.gr

Abstract

*Command injections are prevalent to any application independently of its operating system that hosts the application or the programming language that the application itself is developed.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. A prime example of a real, infamous command injection vulnerability that clearly depicts the threats of this type of code injection was the recently discovered Shellshock bug.Despite the prevalence and the high impact of the command injection attacks, little attention has been given by the research community to this type of code injection. In particular, we have observed that although there are many software tools to detect and exploit other types of code injections such as SQL injections or Cross Site Scripting, to the best of our knowledge there is no dedicated and specialized software application that detects and exploits automatically command injection attacks. This paper attempts to fill this gap by proposing an open source tool that automates the process of detecting and exploiting command injection flaws on web applications, named as commix, (**COMM**and **I**njection e**X**ploitation).This tool supports a plethora of functionalities, in order to cover several exploitation scenarios. Moreover, Commix is capable ofdetecting, with high success rate, whether a web application is vulnerable to command injection attacks. Finally, during the evaluation of the tool we have detected several 0-day vulnerabilities in applications.*

Overall, the contributions of this work are: a) We provide a comprehensive analysis and categorization of command injection attacks; b) We present and analyze our open source tool that automates the process of detecting and exploiting command injection vulnerabilities; c) We will reveal(during presentation) several 0-day command injection vulnerabilities that Commix detected on various web based applications from home services (embedded devices) to web servers.

1 Introduction

Code Injection, is a general term for attack types which consist of injecting code that is consequently executed by the 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 Injection [5]. In this paper, we will exclusively deal with command injection attacks. According to the OWASP [6], *"Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user supplied data (forms, cookies, HTTP headers etc.) to a system shell. In this attack, the attacker-supplied operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation".* The command injection attacks are also named in the literature as "shell command injections" or "OS (Operating System) command injections", because this type of attack, occurs when the application invokes the operating system shell (shell commands on Unix Based Systems, command prompt shell on Windows). In this paper, we will refer to these attacks simply as "Command Injections".

Command injections are prevalent to any application independently of its operating system that hosts the application or the programming language that the application itself is developed. Thus, they have been discovered in web applications hosted in web servers (Windows or *nix) 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 also found in 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 do not include a database, command injections are pervasive in IoT[7]. This is due to the fact that IoT devices run an embedded OS (i.e., typically Linux), thus executing system commands. It is even more alarming the fact that the security of IoT is of paramount importance, since a vulnerability in these devices may lead to privacy breach [8], data loss, or even put human life in risk. Moreover, it is noteworthy that many IoT devices lack a patching procedure to fix bugs and vulnerabilities[9]. That is, there is no automatic- and sometimes not even manual-procedure to update the vulnerable software, meaning that vulnerable IoT devices may indefinitely remain at risk once a command injection vulnerability has been discovered.

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. In particular, an attacker can gain access to resources that he/she 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 code injection 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 [14], Symantec Web Gateway [13], IBM Tealeaf CX [14] and Sophos Web Protection Appliance [15].

The above observations are clear indications that command injections attacks are one of the most dangerous class of code injections attacks that can be found nearly in all network devices that handle input data. Despite the prevalence and the high impact of the command injection attacks, little attention has been given by the research community to this type of code injection. In particular, we have observed that there are many software tools to detect and exploit other types of code injections such as SQL injections (i.e., sqlmap,SQLninja, BSQL Hacker etc.) or Cross Site Scripting (i.e., OWASP Xenotix XSS Exploit Framework, XSSer, etc). However, to the best of our knowledge there is no dedicated and specialized software application that detects and exploits automatically command injection attacks. We have only discovered some custom scripts [16] 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 are some third party plugins for security frameworks (e.g., Burp suite, IronWASP) that aim at detecting command injection attacks in automated manner. However, these plugins have very limited functionality (for instance they do not support post-exploitation functionality to upload a backdoor automatically or they support very trivial command injection scenarios). Finally, some of these plugins are paid as extra services.

This paper attempts to fill this gap by proposing an open source tool that automates the process of detecting and exploiting command injection flaws on web applications, named as commix, (**COMM**and **I**njection e**X**ploitation). More specifically, first we define, categorize and elaborate on command injectionsfor the better understanding of this attack vector. Next, we present and analyze the software architecture of Commix as well its broad functionality that greatly facilitate the detection and exploitation of command injection vulnerabilities.During the evaluation of Commix, we have observed that our tool able to detect, with high success rate, whether a web application is vulnerable to command injection attacks. Moreover, we have identified several 0-day command injection vulnerabilities in various applications.

The rest of the paper is organized as follows. Section 2 elaborates on command injections, while Section 3 analyzes the software architecture of Commix and highlights its advantageous characteristics. Finally, Section 4 includes countermeasures.

2 Analysis of Command Injections

Command injections vulnerabilities may be present in applications which accept and process system commands from the user input (see Figure 1). The purpose of a command injection attack is the insertion of an operating system command through the 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 that

can occur in Windows, Linux, or Unix operating systems as well as programming language-independent that may occur in applications written in various programming languages (such as C, C++, C#, JAVA, PHP, Perl, Python, Ruby etc.) or web-based applications written in Web Application Frameworks (such as ASP.NET, CGI, Python Django, Ruby on Rails etc.). The main reason that an application is vulnerable to command injection attacks is due to incorrect or complete lack of input data validation by the application itself.

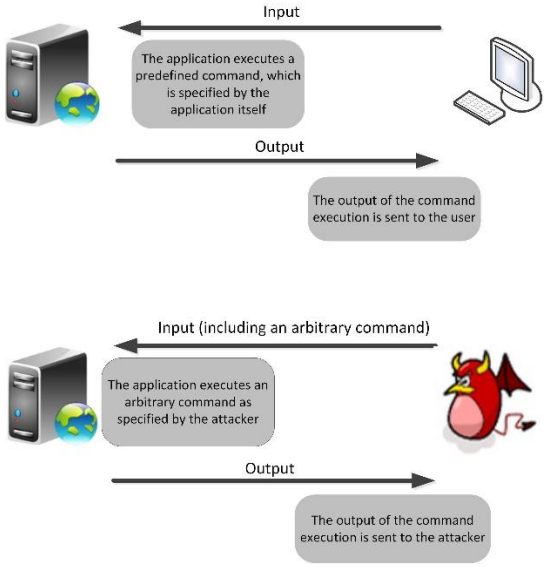

Figure 1: Command injection attack

In general, we identify two main categories of command injections. In the first category, which is named results-based command injections, the vulnerable application outputs the results of the injected command. Thus, the attacker can directly infer if his/her command injection succeeded or not. The second category is namedblind command injections. As it name implies, in this attack, the vulnerable application does not output the results of the injected command, and, therefore, these (i.e., the results) are not visible to the attacker. In the following, we elaborate on the two aforementioned command injection categories.

2.1 Results-based command injections.

As mentioned previously, in results-based command injection attacks the attacker can directly infer the result of the injected command through the response of the web application. We divide results-based command injection attacks into the following two techniques.

2.1.1 Classic results-based command injection

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

2.1.2 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 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.

2.2 Blind Command Injections

The main difference between results-based command injection attacks and blind command injection attacks lies to the way that the 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 two techniques.We divide blind command injection attacks into two different techniques: Time-based and File-based. In the following, we elaborate on these two techniques using code examples, in order to gain a better understanding of blind command injections, since these (i.e., blind command injections) have not been analyzed in the literature as extensively as results-based.

2.2.1 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. Note that the function in Bash shell that can introduce time delays is *"sleep"*. Thus, by observing time delays the attacker is capable of deducing the result of the injected command.

More specifically, assume the vulnerable web application, shown in Figure 2. This application 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. Since there is no echo function, 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., *"whoami"*), the results will not be shown in the screen.

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

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

Let assume that the attacker wants to inject, execute and read the output of the *"whoami"* command using time-based command injection. Note that the *"whoami"* command returns the effective username of the current user. In this case, the attacker can use a chain of commands that brute forces letter-by-letter the output. More specifically, the attacker sends the following HTTP request with the injected chain of commands (see figure 3):

```
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
```

Figure 3 : Attack vector that brute forces letter-by-letter the output.

The injected chain of commands can be more easily understood in the following format:

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

Figure 4 : A chain of commands that brute forces letter-by-letter the output.

The above chain of commands shown in figure 4 (i.e., the piping of the command *"whoami"* with *"cut"*, *"od"*, *"head"* and *"tr"*) obtains the first letter of the output of *"whoami"* command and converts it to the respective ASCII number. Next, the chain of commands check if this letter is the first character of the ASCII table by checking if it is equal to the ASCII 119 (i.e., the letter *"a"*), using the same time-delay technique as before. If it is, then the attacker continues with the second letter of the output. If it is not, then the attacker should check the next character of the ASCII table which is ASCII 120 (i.e., the letter *"b"*). This procedure continues until all the characters of the output are found. It is worth noting that through this technique, the process of finding a 8-characters long text (i.e., *"www-data"*) takes about 10 seconds maximum, since the discovery of each character can take up to 1.25 seconds.

2.2.2 File-based 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 he/she can write them to a file, which is accessible to the attacker. This command injection technique follows exactly the same methodology as the Classic results-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 "semiblind" command injection technique, 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 same vulnerable web application shown in figure 2.

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

Figure 5 : Execution of the "whoami" command, while the output is saved in a text file.

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

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

Figure 6 : The attacker can trivially read the text file that includes the output of the injected command.

An essential prerequisite to achieve this, 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 of the web server is not writable, an alternative solution for the attacker is to use temporary directories, such as *"/tmp"* or *"/var/tmp"* to store in a text file the output of the injected command. The limitation in this solution is that the attacker cannot read files located in these temporary directories through the web application, due to his/her limited privileges. To bypass this limitation, the attacker can apply the time-based blind command injection technique to read the contents of the text file. For example, the attacker can use the following HTTP GET request, in order to store the output of *"whoami"* command to a random file (i.e *"/tmp/UvlLSE5S.txt")* and subsequently how many characters there are in this file.

3 COMMIX

3.1 Software architecture

Commix [17] aims to greatly facilitate 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. Commix 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 takes a URL address (from any website) as an input with a GET/POST parameter. Then, the imported data is altogether investigated for command injection vulnerabilities.

As shown in Figure 7, the general structure of the tool is divided into three main modules: i) the Attack Vector Generator module, ii) the Vulnerability Detection module, and ii) the Exploitation module. The Attack Vector Generator module as it name implies, generates a set of command injection attack vectors. The latter are produced from a command injection separators list and the type of command injection that will be performed (i.e., Classic, Dynamic code evaluation, Blind-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 engine. The latter performs the command injection to the HTTP parameters of the vulnerable web application using one by one the received attack vectors. It is important to mention that although the two most common attack vectors are usually supplied to the web application through HTTP GET and POST method parameters, commix supports injecting commands in HTTP parameters, such as HTTP cookie, HTTP User-Agent and Referer header values**.**

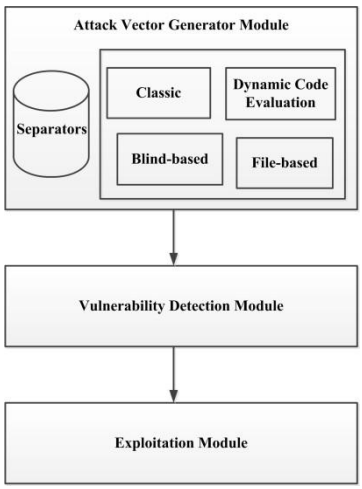

Figure 7: Commix Software Architecture

If the detection module determined that the application is vulnerable, then commix triggers the exploitation module to attempt automatic exploitation. In particular, the same attack vector that the detection module succeeded in performing the command injection is used by the exploitation module now to exploit the application with the difference that the injected command is selected by the user. If the exploitation of the vulnerability is successful, then the execution results will be displayed on the screen to the user. Note that the exploitation module provides also an integration interface with the exploitation framework Metasploit, in order to perform automatic exploitation for penetration testing scenarios.

3.2 Reducing false positives

An important feature of vulnerability detection module is the use of heuristics to reduce false positives. More specifically, during our initial experiments, we observed that commix produced several false positives. More specifically, we observed cases where the result of a command injection attempt was similar to what commix expected to receive (i.e., the output of the command), without however the injected command actually being executed. In particular, we have observed that, while there was use of random strings to determine whether or not there is vulnerability, there were several cases where the web application printed back that string, thus commix considers that the application is vulnerable. To address this issue and reduces false positives, the detection module of Commix makes use of heuristics, in order to "decide" whether an application is vulnerable to command injection attacks. More specifically, the procedure of the detection is as follows: initially, it tries to execute arbitrary commands, but only those for which the response that will be delivered after the execution, is known in advance. For instance, in order to "decide" whether an application is vulnerable to results-based command injection flaws, commix will try to echo three times a randomly generated 5-characters string combined with the result of a random mathematic calculation of two randomly selected numbers. Figure 8 shows an example of such a string where the random string is "NTAVG", while the calculation is the sum of the two randomly generated numbers 28 and 50.

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

Figure 8 : An example of a heuristic command injection to examine whether an application is vulnerable or not.

If all executed properly, we must take as response a string "NTAVG78NTAVGNTAVG", which is the union of the randomly generated strings combined with the result of the mathematic calculation. After receiving the response from the application, commix 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 by attempting another attack vector. The process is repeated up to the point where a vulnerability has been identified or until all possible combinations of command injection vectors are executed.

Regarding blind command injections, after several tests in many applications, we have found that there is high probability of false-positive results, due to response delays (i.e., random or accidental response delay of the target host). Therefore, we added a time-based false positive check to commix, which calculates the average response time of the target host. Then, the average response time, that was calculated previously, is

added to the default delay time, which is used by commix to execute command injection attacks. If, for any reason, we need to alter the default delay time, it is possible to specify a number of seconds for the applications to "sleep" between each HTTP(S) request.

3.3 Other Functionalities

Commix supports a plethora of functionalities, in order to cover several exploitation scenarios. 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/v0.2b-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., BurbSuite etc). In addition, Commix supports the execution of command injections through the Tor anonymity network for anonymity purposes [18].

In certain situations there is a need to test only one or just a few specific command injection techniques (i.e., only the "Dynamic code evaluation technique"). Commix supports an option, which can be used to specify one or just a few specific command injection techniques. Note that, by default, commix tests all injection techniques it supports against a target host.

Moreover, commix provides the capability to insert user's own suffixes and prefixes. In some circumstances, like on the code snippet in figure 9, the vulnerable parameter is exploitable, only if the user provides a specific prefix in the injection attack vector. To be more specific, in this example (see figure 9), the GET parameter *"ip"* is verified using the *preg_match()* function, whether it includes an IP address. For this reason, an IP address should be inserted as prefix at the beginning of the attack vector, followed by the injection command. Based on the same logic, an IP address can be inserted at the end of the attack vector, as a suffix.

```
<?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 9 : The "ip" parameter is verified using a regular expression.

During the development and testing procedure of commix, we encountered systems, which have limited available bash commands (i.e., *"cat", "echo"* etc.). Thus, many aforementioned injection techniques based on these bash commands failed. Therefore, we created an option for an "alternative" shell, which produces exactly the same attack vector for each of the aforementioned injection techniques. However, it makes exclusive use of specific commands which are available on a programming language and they are not bash commands. At the moment, this option supports only the Python programming language. The only prerequisite is that the specific programming language in which the conversion of the command will be based, should be pre-installed on the target system. We aim, in the next versions of commix, to add more alternative shells based on PHP and Ruby.

```
python -c "print 'LIZGD'+str(int(6+1))+'LIZGD'+'LIZGD'"
```

Figure 10 : A Python-based format of "decision" heuristic

There are several cases where we want to take actions such as system and user enumeration fast without dealing with complex bash system commands. For that reason, commix supports many "Enumeration" options. These options can be used to enumerate the target host in a fast and easy way. To be more specific, a

user can retrieve the current user name and/or check if that user has root privileges. It is also possible to retrieve the hostname, the operating system and the system architecture. Finally, it is possible to enumerate system usernames, users' privileges, access the *"/etc/passwd"* file and users' password hashes by accessing the *"/etc/shadow"* file, if it is readable by the current user. Commix also allows users to write or upload files automatically in the target system, by selecting the "File access" options. These options can be used to access files on the target host; for example, to read a file from the target host, to write to a file on the target host or to upload a file on the target host.

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 his own python modules, in order to perform whatever task he/she desires. At the time of writing the paper, commix comes with two modules. The first module, which is named as *"icmp_exfiltration.py"* supports the ICMP exfiltration technique, which exfiltrates data using the *"ping"* utility [19], [20]. The other module, which is named as *"shellshock.py"* can check the target host against the Shellshock vulnerability and then, if it seems to be vulnerable, it tries for an automated exploitation.

4 Countermeasures

The two most important programming techniques to prevent command injection vulnerabilities are: i) input validation and ii) escaping input data. The former (i.e., input validation) refers to the process of filtering (i.e., removing) dangerous characters from the input data. On the other hand, the latter (i.e., escaping input data), is used to render dangerous characters as simple text string, rather than interpreted by the OS as special characters 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 by using two different techniques: (i) Blacklisting and (ii) Whitelisting. Moreover, to escape input data, developers should use APIs as provided by the programming languages.

5 Acknowledgment

The publication of this paper has been partly supported by the University of Piraeus Research Center.

6 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] 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] ExploitDB, "Offensive Security Exploit Database Archive", https://www.exploit-db.com/
[17] Commix, https://github.com/stasinopoulos/commix
[18] Tor Project, https://www.torproject.org/
[19] Data exfiltration on Linux, http://blog.ring-zer0.com/2014/02/data-exfiltration-on-linux.html
[20] Exfiltrate Data using the old ping utility trick, http://blog.curesec.com/article/blog/23.html