File Inclusion Prevention

This module has discussed various ways to detect and exploit file inclusion vulnerabilities, along with different security bypasses and remote code execution techniques we can utilize. With that understanding of how to identify file inclusion vulnerabilities through penetration testing, we should now learn how to patch these vulnerabilities and harden our systems to reduce the chances of their occurrence and reduce the impact if they do.

File Inclusion Prevention

The most effective thing we can do to reduce file inclusion vulnerabilities is to avoid passing any user-controlled inputs into any file inclusion functions or APIs. The page should be able to dynamically load assets on the back-end, with no user interaction whatsoever. Furthermore, in the first section of this module, we discussed different functions that may be utilized to include other files within a page and mentioned the privileges each function has. Whenever any of these functions is used, we should ensure that no user input is directly going into them. Of course, this list of functions is not comprehensive, so we should generally consider any function that can read files.

In some cases, this may not be feasible, as it may require changing the whole architecture of an existing web application. In such cases, we should utilize a limited whitelist of allowed user inputs, and match each input to the file to be loaded, while having a default value for all other inputs. If we are dealing with an existing web application, we can create a whitelist that contains all existing paths used in the front-end, and then utilize this list to match the user input. Such a whitelist can have many shapes, like a database table that matches IDs to files, a `case-match` script that matches names to files, or even a static json map with names and files that can be matched.

Once this is implemented, the user input is not going into the function, but the matched files are used in the function, which avoids file inclusion vulnerabilities.

Preventing Directory Traversal

If attackers can control the directory, they can escape the web application and attack something they are more familiar with or use a `universal attack chain`. As we have discussed throughout the module, directory traversal could potentially allow attackers to do any of the following:

- Read `/etc/passwd` and potentially find SSH Keys or know valid user names for a password spray attack
- Find other services on the box such as Tomcat and read the `tomcat-users.xml` file
- Discover valid PHP Session Cookies and perform session hijacking
- Read current web application configuration and source code

The best way to prevent directory traversal is to use your programming language's (or framework's) built-in tool to pull only the filename. For example, PHP has `basename()`, which will read the path and only return the filename portion. If only a filename is given, then it will return just the filename. If just the path is given, it will treat whatever is after the final / as the filename. The downside to this method is that if the application needs to enter any directories, it will not be able to do it.

If you create your own function to do this method, it is possible you are not accounting for a weird edge case. For example, in your bash terminal, go into your home directly (cd ~) and run the command `cat .?/.*/.?/etc/passwd`. You'll see Bash allows for for the `?` and `*` wildcards to be used as a `.`. Now type `php -a` to enter the PHP Command Line interpreter and run `echo file_get_contents('.?/.*/.?/etc/passwd');`. You'll see PHP does not have the same behaviour with the wildcards, if you replace `?` and `*` with `.`, the command will work as expected. This demonstrates there is an edge cases with our above function, if we have PHP execute bash with the `system()` function, the attacker would be able to bypass our directory traversal prevention. If we use native functions to the framework we are in, there is a chance other users would catch edge cases like this and fix it before it gets exploited in our web application.

Furthermore, we can sanitize the user input to recursively remove any attempts of traversing directories, as follows:

Code: php

```php
while(substr_count($input, '../', 0)) {
    $input = str_replace('../', '', $input);
};
```

As we can see, this code recursively removes `../` sub-strings, so even if the resulting string contains `../` it would still remove it, which would prevent some of the bypasses we attempted in this module.

Web Server Configuration

Several configurations may also be utilized to reduce the impact of file inclusion vulnerabilities in case they occur. For example, we should globally disable the inclusion of remote files. In PHP this can be done by setting `allow_url_fopen` and `allow_url_include` to Off.

It's also often possible to lock web applications to their web root directory, preventing them from accessing non-web related files. The most common way to do this in today's age is by running the application within `Docker`. However, if that is not an option, many languages often have a way to prevent accessing files outside of the web directory. In PHP that can be done by adding `open_basedir = /var/www` in the php.ini file.

If these configurations are applied, to should prevent accessing files outside the web application folder, so even if an LFI vulnerability is identified, its impact would be reduced.

Web Application Firewall (WAF)

The universal way to harden applications is to utilize a Web Application Firewall (WAF), such as `ModSecurity`. When dealing with WAFs, the most important thing to avoid is false positives and blocking non-malicious requests. ModSecurity minimizes false positives by offering a `permissive` mode, which will only report things it would have blocked. This lets defenders tune the rules to make sure no legitimate request is blocked. Even if the organization never wants to turn the WAF to "blocking mode", just having it in permissive mode can be an early warning sign that your application is being attacked.

Finally, it is important to remember that the purpose of hardening is to give the application a stronger exterior shell, so when an attack does happen, the defenders have time to defend. According to the FireEye M-Trends Report of 2020, the average time it took a company to detect hackers was 30 days. With proper hardening, attackers will leave many more signs, and the organization will hopefully detect these events even quicker.

It is important to understand the goal of hardening is not to make your system un-hackable, meaning you cannot neglect watching logs over a hardened system because it is "secure". Hardened systems should be continually tested, especially after a zero-day is released for a related application to your system (ex: Apache Struts, RAILS, Django, etc.). In most cases, the zero-day would work, but thanks to hardening, it may generate unique logs, which made it possible to confirm whether the exploit was used against the system or not.

Start Instance

1 / 1 spawns left

Waiting to start...

Questions

Answer the question(s) below to complete this Section and earn cubes!

Cheat Sheet

Get VPN Key

Target: Click here to spawn the target system!

SSH to with user "htb-student" and password "HTB_@cademy_stdnt!"

+ 0 What is the full path to the php.ini file for Apache?

Submit your answer here...

Submit

+ 0 Edit the php.ini file to block system(), then try to execute PHP Code that uses system. Read the /var/log/apache2/error.log file and fill in the blank: system() has been disabled for _____ reasons.

Submit your answer here...

Submit

Hint

Previous Next

Cheat Sheet

Go to Questions

Table of Contents

Remote Code Execution

- PHP Wrappers ✓
- Remote File Inclusion (RFI) ✓
- LFI and File Uploads ✓
- Log Poisoning ✓

Automation and Prevention

- Automated Scanning ✓
- File Inclusion Prevention

Skills Assessment

- Skills Assessment - File Inclusion

My Workstation

OFFLINE

Start Instance

1 / 1 spawns left