**Second-Order LFI**

Local File Inclusion (LFI) is a vulnerability that is typically easy to spot; furthermore, it is comparably easy to exploit unless we need to bypass a Web Application Firewall. Even then, most of the time, there is only a limited number of techniques to break out of the intended directory, for instance, using ../. However, if such characters are blocked, exploitation becomes impossible. Due to this nature of LFI vulnerabilities, attackers may overlook more complex forms of LFI vulnerabilities, which require a more in-depth understanding of the underlying web application to exploit. We will explore such an example as a second-order LFI in this section.

**Code Review - Identifying the Vulnerability**

Looking at the web application, we can see an adjusted version of the web applications from the previous sections. This time, we can update our username as well as the name of stored files:

A screenshot of a computer

Description automatically generated

Let us analyze the source code to identify if there is a way to include local files on the web server's file system. Analyzing how the web application interacts with the database in db.php, we can see that it no longer fetches file contents from the database but instead stores the files locally on the file system and displays them by fetching them. The web application stores the files in a folder named after the corresponding owner of the file, which is an obvious entry point for an LFI vulnerability:

Code: php

function fetch\_data($id){

global $conn;

$sql = "SELECT \* FROM data WHERE id=?;";

$stmt = mysqli\_stmt\_init($conn);

if(!mysqli\_stmt\_prepare($stmt, $sql)){

echo "SQL Error";

exit();

}

// execute query

$id = intval($id);

mysqli\_stmt\_bind\_param($stmt, "i", $id);

mysqli\_stmt\_execute($stmt);

$result = mysqli\_stmt\_get\_result($stmt);

$result = mysqli\_fetch\_assoc($result);

$owner = $result['owner'];

$name = $result['name'];

$path = '/var/www/' . $owner . '/' . $name . '.txt';

return array("name" => $name, "content" => file\_get\_contents($path));

}

To determine if we can exploit this LFI, let us explore what happens when we change our username or a file's name, starting with the latter. We might be able to leak any text file on the system by changing the filename to something like ../../../path/to/textfile; this would result in the path /var/www/htb-stdnt/../../../path/to/textfile.txt, thus leaking the file to us. We can analyze the logic implemented in edit\_filename.php:

Code: php

<SNIP>

$user\_data = fetch\_user\_data($\_SESSION['user']);

$data = fetch\_data($\_SESSION['file\_id']);

if(isset($\_POST['new\_filename'])){

$new\_filename = $\_POST['new\_filename'];

$user = $\_SESSION['user'];

$file\_id = $\_SESSION['file\_id'];

# reject hacking attempts

$invalid = strpos($new\_filename, '..') || strpos($new\_filename, '/') || strpos($new\_filename, '\\');

if($invalid) {

$\_SESSION['msg'] = "Invalid characters in filename! You have been logged out for security reasons.";

header("Location: index.php");

exit;

}

update\_filename($file\_id, $user, $new\_filename, $data['name']);

header("Location: display\_data.php");

exit;

}

<SNIP>

Unfortunately, the web application rejects filenames containing either .., /, or \, preventing us from escaping our user directory; this only allows us to leak files within our user directory, which is not a security issue.

Additionally, the file is moved to the new location in the function update\_filename in db.php:

Code: php

function update\_filename($id, $user, $new\_filename, $old\_filename){

<SNIP>

# move file to new location

$old\_path = '/var/www/' . $user . '/' . $old\_filename . '.txt';

$new\_path = '/var/www/' . $user . '/' . $new\_filename . '.txt';

rename($old\_path, $new\_path);

}

Since the web application prevents the apparent LFI vulnerability by implementing filters, let us move on to the functionality allowing us to change our username, with its corresponding logic implemented in edit\_username.php:

Code: php

$user\_data = fetch\_user\_data($\_SESSION['user']);

if(isset($\_POST['new\_username'])){

$new\_username = $\_POST['new\_username'];

if(update\_username($\_SESSION['user'], $new\_username)){

$\_SESSION['user'] = $new\_username;

header("Location: profile.php");

exit;

}

$msg = "Error! Username is already taken!";

}

This time, there is no filter. Thus we might be able to inject a sequence like ../ into our username, allowing us to change the intended directory files are read from. The function update\_username is implemented in db.php:

Code: php

function update\_username($user, $new\_username){

global $conn;

# check if user already exists

if (fetch\_user\_data($new\_username)){

return false;

}

# update username

$sql = "UPDATE users SET username=? WHERE username=?;";

<SNIP>

# update files

$sql = "UPDATE data SET owner=? WHERE owner=?;";

<SNIP>

return true;

}

We can see that the web application checks whether the username already exists, preventing us from changing it to an existing user's name to access their files. However, there is an apparent bug: the developers forgot to update the file paths when the username was changed. This leads to the following behavior:

Assume our user htb-stdnt owns a file named test.txt. The web application stores this file in the path /var/www/htb-stdnt/test.txt. If we rename the file to HelloWorld.txt, it will be moved to /var/www/htb-stdnt/HelloWorld.txt. If we now try to access the file via the new name, it will be loaded from that path and displayed in the web application. However, if we now change our name to test, the file path is not updated. Since the web application bases the directory files are read from on our username, the next time we try to access our file HelloWorld.txt, the web application attempts to read it from /var/www/test/HelloWorld.txt. However, since the file was not moved, this system path does not exist, so our file will not be displayed in the web application.

While this is a functional issue and not a security issue, we can explore this further to see if this can be escalated to a security issue. Since our username is not filtered for special characters, we can change a filename to match the name of a different text file on the filesystem we want to leak. We are limited to .txt files since the extension is hardcoded into the PHP code. If we change our username to change the directory the file is read from, the web application will leak that system file to us, leading to an LFI vulnerability. More specifically, this is our exploit plan. As a proof-of-concept, let us target a proof-of-concept file located at /tmp/poc.txt. To leak the file, we need to execute the following steps:

1. Rename any of our files to poc.txt. This moves our file to /var/www/htb-stdnt/poc.txt
2. Rename our user to ../../tmp. Due to the bug in the web application, no file is moved, and thus no file is overwritten
3. Fetch our file poc.txt. The web application will load the file from /var/www/../../tmp/poc.txt, thus leaking the targeted file to us

**Debugging the Application Locally**

To test our attack chain locally, we must first create our proof of concept file. We can accomplish this with the following command:

yovecio@htb[/htb]$ echo 'The Exploit Works!' > /tmp/poc.txt

Now, let us create our MySQL Docker container. To do so, let us use the following file to seed the database:

Code: sql

CREATE TABLE `data` (

`id` int(11) NOT NULL,

`owner` varchar(256) NOT NULL,

`name` varchar(256) NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `users` (

`id` int(11) NOT NULL,

`username` varchar(256) NOT NULL,

`description` varchar(256) NOT NULL,

`password` longtext NOT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

# htb-stdnt:Academy\_student!

INSERT INTO `users` (`id`, `username`, `description`, `password`) VALUES

(1, 'htb-stdnt', 'This is the user for HackTheBox Academy students.', '$2a$12$f4QYLeB2WH/H1GA/v3M0I.MkOqaDAkCj8vK4oHCvI3xxu7jNhjlJ.');

INSERT INTO `data` (`id`, `owner`, `name`) VALUES

(1, 'htb-stdnt', 'Lorem Ipsum');

Afterward, we can create a Docker container using the following command:

yovecio@htb[/htb]$ docker run -p 3306:3306 -e MYSQL\_USER='db' -e MYSQL\_PASSWORD='db-password' -e MYSQL\_DATABASE='db' -e MYSQL\_ROOT\_PASSWORD='db' --mount type=bind,source="$(pwd)/db.sql",target=/docker-entrypoint-initdb.d/db.sql mysql

Now, let us host the web application using PHP's built-in web server:

yovecio@htb[/htb]$ php -S 127.0.0.1:8000

[Thu Aug 17 10:47:20 2023] PHP 7.4.33 Development Server (http://127.0.0.1:8000) started

Lastly, we need to create the file on our filesystem that the web application expects. Since the path depends on our username and the filename, we need to create the file /var/www/htb-stdnt/Lorem Ipsum.txt, which we can do using the following command:

yovecio@htb[/htb]$ sudo mkdir /var/www/htb-stdnt/

yovecio@htb[/htb]$ echo 'This is the file Lorem Ipsum!' | sudo tee /var/www/htb-stdnt/Lorem\ Ipsum.txt

We can then access the web application at 127.0.0.1:8000 and should be able to access our file after logging in:

A screenshot of a computer

Description automatically generated

**Exploitation**

To exploit the second-order LFI vulnerability, we need to follow our exploit plan above. Firstly, let us change our filename to poc:

A screenshot of a computer

Description automatically generated

Now, let us update our username and set it to ../../tmp:

A screenshot of a computer

Description automatically generated

If we now select our renamed file poc, the web application breaks out of our intended user directory and leaks our proof-of-concept file:

A screenshot of a computer

Description automatically generated

While this LFI vulnerability is restricted as it only allows us to leak .txt files, it is still a security issue. Regardless of our simple sample web application, second-order vulnerabilities can be tricky to identify and exploit; this applies exponentially more in real-world complex web applications. Thus, it is crucial to analyze the source code closely in a whitebox penetration test to establish an overview of how different web application components interact to bypass security measures that protect only a limited number of components.