File Upload

Category: Web
Difficulty: Easy
Author: Kolja

Description

(yes, this was the whole description...)

Summary

The challenge gives us a full docker setup the remote server is using. After a bit of browsing around and testing the application and the source code, it is fairly obvious that the goal of the challenge is uploading a PHP reverse shell and using it to read out the `/flag` file.

How not to implement authentication

The first hurdle of this challenge is the login system, allowing only admins to upload files. The application checks this by requesting the following SQL query from the user database:

```
SELECT username FROM fileupload_users WHERE username = /*username of current user*/ AND staff = 0x1;
```

and checking if the output contains a username or not. The only user having the `staff` bit set is the preregistered `administrator` account with a random password.

The only way to be able to upload files would be to log in with a new account with the same name as the admin account, but surely the register page has some way of stopping you from creating a new user with the same name as

an already existing one, right?

On the register page, it first checks whether the username is taken or not, by requesting the following SQL query:

```
SELECT id FROM fileupload_users WHERE BINARY username = ?
```

and checking if the output contains any results. Looking at this query, we can see the keyword `BINARY` being used, making it a case-sensitive comparison, which it normally is not. We can use this knowledge to create an account with the name `Administrator`, which lets us upload files, because the query to verify an administrator does not use a case-sensitive comparison.

Wroom Wroom

With the ability to upload files, we need to get the server to execute our own PHP code. Unfortunately, there are two countermeasures we need to circumvent. First, the server only accepts files that do not have one of the common PHP file extensions. Luckily for us, we can just upload a `.htaccess` file with a single line:

```
AddHandler application/x-httpd-PHP .txt
```

letting the server handle all `.txt` files as PHP files.

The second countermeasure is the server scanning the whole uploaded file for the `PHP` starting characters `<?`:

```
<?PHP
//...
if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file)) {
        $message = "The file " . htmlspecialchars(basename($_FILES["fileToUpload"]["name"])) . " has
been uploaded.";
        // Open file and check for <?
        $uploaded_file = fopen($target_file, "r") or die("Unable to open file!");
        $file_content = fread($uploaded_file, filesize($target_file));
        fclose($uploaded_file);
        if (str_contains($file_content, '<?')) {
                // If found write this to the log for later reporting
                error_log("Some hackers tried to upload code to our server. Pls check if we were
compromised");
```

```
                // Overwrite the offending file with a warning to the hackers.
                $uploaded_file = fopen($target_file, "w");
                fwrite($uploaded_file, "Nice try hackers!");
        }
}
//...
?>
```

At closer inspection we can find that the server first stores the uploaded file without checking for illegal characters and only afterwards checks for them, opening a small time window in which we are able to request the unchecked uploaded file and letting the server execute our own PHP code. Now we can just find out the filename of the flag file and read it out.

Mitigation

To secure the login, a unique identifier for every user is needed. This can either be an uuid or the username, but it is important that no user is able to disguise as another user(which was the problem in this case). The file upload can be secured by only storing the file after the content has been checked to not contain dangerous content.