**Session Puzzling Prevention**

After seeing different ways to identify and exploit session puzzling vulnerabilities, let's discuss how we can protect ourselves from these type of attacks. Improper handling of session variables is what typically introduces session puzzling vulnerabilities. In particular, the re-use of session variables, premature session population, or insecure default values for session variables can be the source of session puzzling vulnerabilities.

**Insecure Configurations**

Session puzzling vulnerabilities can occur in any web application that stores data in session variables. Let's look at a few of the misconfigurations that caused the session puzzling vulnerabilities in the previous sections and how to prevent them.

**Insecure Defaults**

Session puzzling vulnerabilities caused by insecure defaults typically occur when the session is initialized or reset to an inappropriate default value. The following is a simplified example code snippet:

Code: php

// login

if(check\_password($\_POST['username'], $\_POST['password'])) {

$\_SESSION['user\_id'] = get\_user\_id($username);

header("Location: profile.php");

die();

} else {

echo "Unauthorized";

}

// logout

if(isset($\_POST['logout'])) {

$\_SESSION['user\_id'] = 0;

}

We can see that upon login, the web application stores the user ID in the session variable to identify that is currently logged in. However, when the user logs out, the user ID is set to 0. This can be an insecure default if the user ID 0 is valid. This is often the first user in the database which is commonly the admin user.

To prevent this vulnerability, we should not set the variable to any default at all, but rather just unset the session such that no user\_id property is present. This corresponds to emptying the session variable completely. In PHP, this can be achieved with the following code:

Code: php

<SNIP>

// logout

if(isset($\_POST['logout'])) {

session\_start();

session\_unset()

session\_destroy();

}

**Common Session Variables**

Another issue we have seen in previous sections is the reuse of session variables. Look at the following code for a login function:

Code: php

if(isset($\_POST['Submit'])){

if(login($\_POST['Username'], $\_POST['Password'])) {

$\_SESSION['Username'] = $\_POST['Username'];

header("Location: profile.php");

exit;

} else {

<SNIP>

}

}

This web application also implements a password reset functionality with a code similar to the following:

Code: php

if(isset($\_POST['Submit'])){

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

if ($user\_data) {

$\_SESSION['Username'] = $user\_data['username'];

header("Location: security\_question.php");

die();

}

}

If a valid username is entered, the web application stores the username in the session variable to fetch the current user's security question in the next step. However, the session variable Username is re-used in the login process. This means that entering the username during the password reset process populates the same session variable that is used to check whether a user is authenticated, leading to the authentication bypass vulnerabilities we have seen previously.

To prevent this, it is best to never re-use session variables for different processes on the web application since it can be hard to keep track of how the different processes intertwine and may be combined to bypass certain checks. Additionally, a separate session variable should be used to keep track of whether a user is currently logged in. Following is a simple improved example:

Code: php

if(isset($\_POST['Submit'])){

if(login($\_POST['Username'], $\_POST['Password'])) {

$\_SESSION['auth\_username'] = $\_POST['Username'];

$\_SESSION['is\_logged\_in'] = true;

header("Location: profile.php");

exit;

} else {

<SNIP>

}

}

**Premature Population**

Another common source of session puzzling vulnerabilities is the premature population of session variables. This can happen by placing the session variable assignment outside of an intended if-statement or simply confusing the steps of a process on the web server. Let's look at an example of a login process:

Code: php

if(isset($\_POST['Submit'])){

$\_SESSION['auth\_username'] = $\_POST['Username'];

$\_SESSION['is\_logged\_in'] = true;

if(login($\_POST['Username'], $\_POST['Password'])) {

header("Location: profile.php");

exit;

} else {

header("Location: login.php?failed=1");

exit;

}

}

if (isset($\_GET['failed'])) {

echo "Login failed for user " . $\_SESSION['auth\_username'];

session\_start();

session\_unset()

session\_destroy();

}

We can see that the session variables are populated immediately after the form has been submitted. The user is then redirected if the login was successful, and an error message is displayed if the login failed. Afterward, the session is destroyed. Due to the premature population of the session variables, the user is thus considered logged in by the web server before the password is checked. This can easily be prevented by ensuring that the session variables are not populated prematurely, but only after the login process has been completed:

Code: php

if(isset($\_POST['Submit'])){

$\_SESSION['login\_fail\_user'] = $\_POST['Username'];

if(login($\_POST['Username'], $\_POST['Password'])) {

$\_SESSION['auth\_username'] = $\_POST['Username'];

$\_SESSION['is\_logged\_in'] = true;

header("Location: profile.php");

exit;

} else {

header("Location: login.php?failed=1");

exit;

}

}

if (isset($\_GET['failed'])) {

echo "Login failed for user " . $\_SESSION['login\_fail\_user'];

session\_start();

session\_unset()

session\_destroy();

}

**General Remarks**

The above examples highlight different issues that can cause session puzzling vulnerabilities. However, generally preventing session puzzling can be challenging because it might be difficult to spot, especially without access to the source code. In complex web applications that support multiple different processes that use session variables, session puzzling can be hard to identify even with access to the source code. Therefore, generally preventing session puzzling is not easy. However, there are some general best practices that we should follow when handling session variables:

* Completely unset session variables instead of setting a default value at re-initialization
* Use a single session variable only for a single, dedicated purpose
* Only populate a session variable if all prerequisites are fulfilled and the corresponding process is complete