Week 7 Homework 5

Matt Rieser

SDEV 325 – Section 6381

October 14, 2018

Professor Zachary Fair

Improper Restriction of Excessive Authentication Attempts

This porous defense weakness will be demonstrated by utilizing a simple web application. To allow for greater code portability, the MySQL part of the LAMP stack has been omitted, and PHP $_SESSION functionality will allow for critical data to persist. In a production environment, an SQL database would need to be utilized as a matter of best practice.

Vulnerable Application

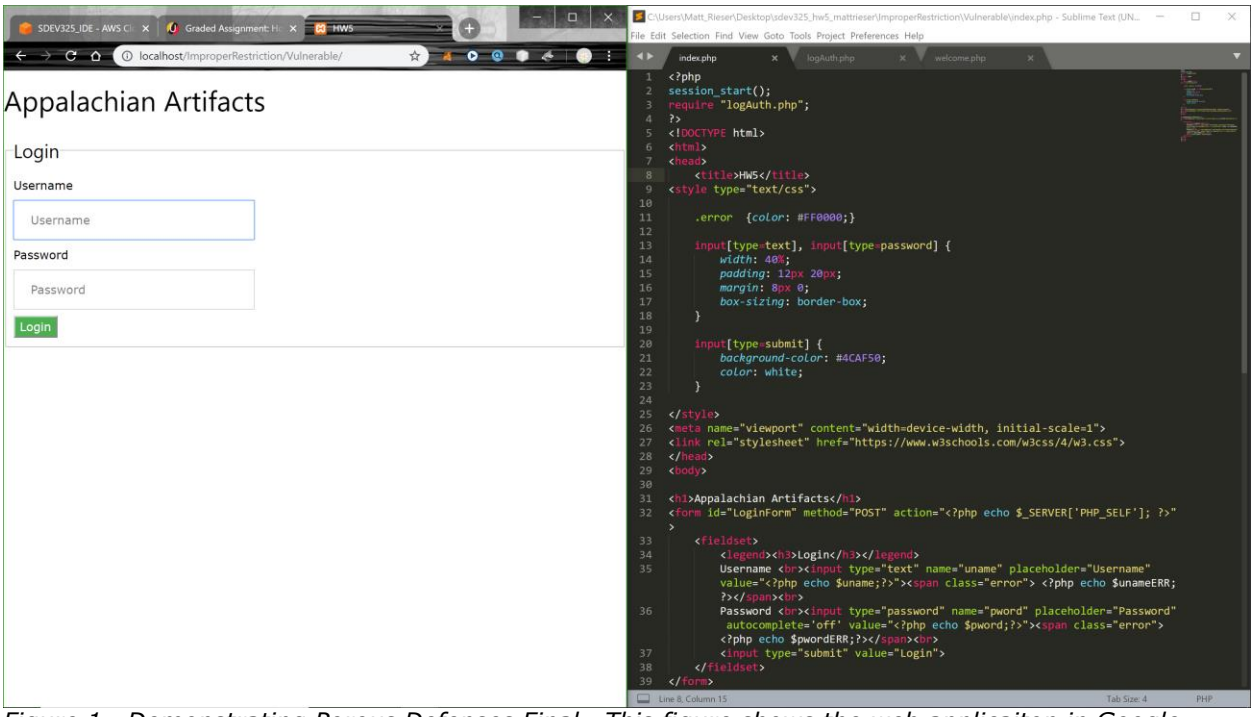

Figure 1. Demonstrating Porous Defenses Final. This figure shows the web applicaiton in Google Chrome browser (left) and it's source code in the Sublime Test (right).

In *Figure 1*, **index.php** is a simple HTML login form that collects username and password credentials, passes the information to the logic hidden behind the statement on line 3.

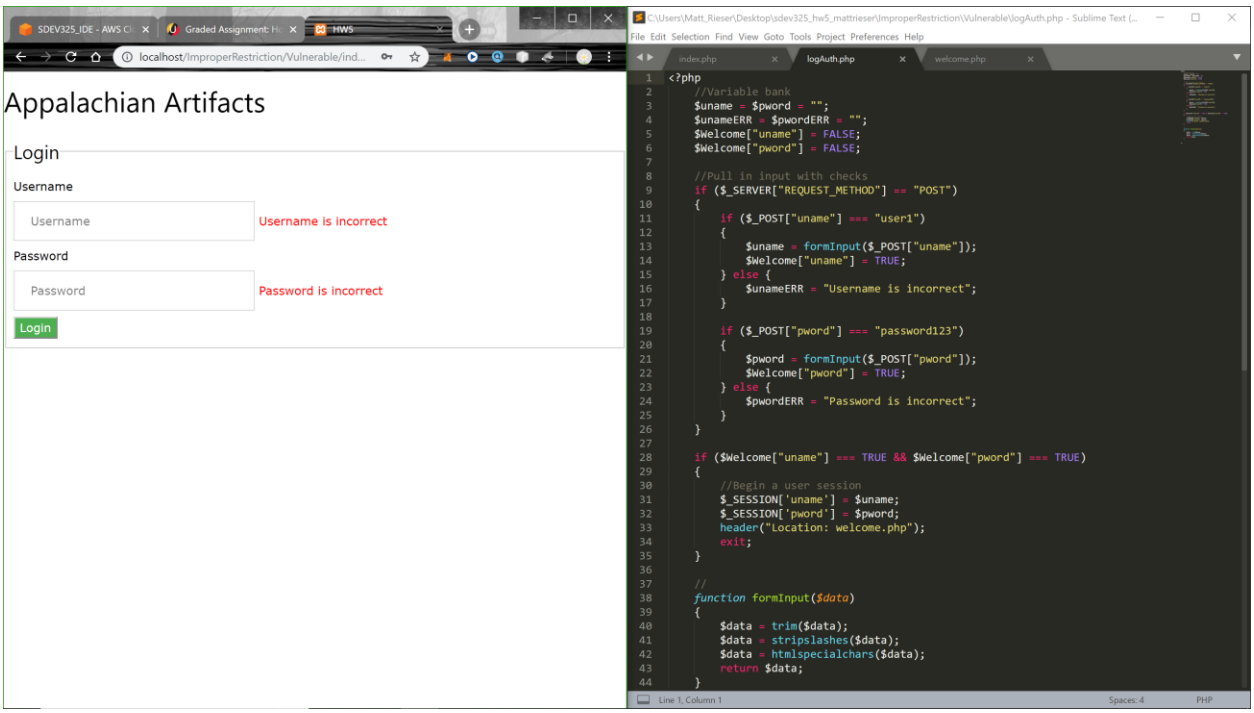

Figure 2. Demonstrating Porous Defenses Final. This figure shows the web applicaiton in Google Chrome browser (left) and it's source code in the Sublime Text (right).

Notice the code on the right (see *Figure 2*), which contains the application logic to handle user authentication. Lines 9-26 accomplish two main tasks: 1. Sanitize user input via the formInput function (defined on line 38). 2. Validates user provided credentials (lines 11, 19). Beginning on line 28, the program begins a user session, stores relevant information, and redirects the user to the **welcome.php** page. Additionally, notice when a user enters incorrect credentials, the application highlights the respective input field with a red lettered error message.

A closer examination of the **logAuth.php** code reveals there is no restriction of authentication attempts. The issue thereof is that an attacker can utilize password cracking tools such as John the Ripper or hashcat with Selenium to automate the process of recovering the password for a given username. If the password reset policy is public as well this makes it more convenient for the attacker.
The way CWE-307 will be mitigated in the demo application is by implementing a mechanism to lockout the target account after some number of failed authentication attempts. The site administrator will then have to unlock that account before it can be used again. Other remedies include: disconnecting the user after a small number of failed attempts, implementing a timeout, and requiring a computational task on the user's part (as recommended by OWASP).

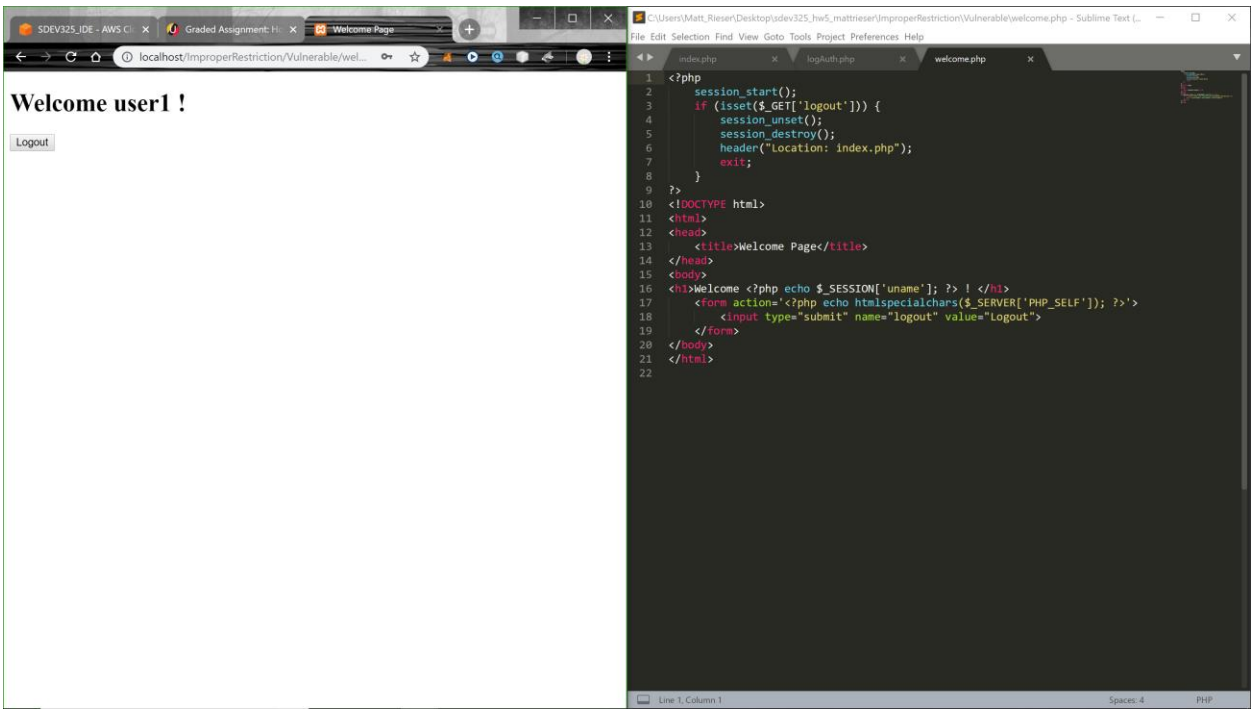

Figure 3. Demonstrating Porous Defenses Final. This figure shows the web applicaiton in Google Chrome browser (left) and it's source code in the Sublime Text (right).

In *Figure 3*, notice **welcome.php** code welcomes the user and allows them to logout. In a production environment, this page would represent a user profile or admin panel. Logging out redirects the user to the **index.php** and nullifies the session data.

Patched Application

This version of the application will apply a lockout mechanism to the single registered account, user1, after 3 failed login attempts. The patched version retains the same project structure from the vulnerable version.

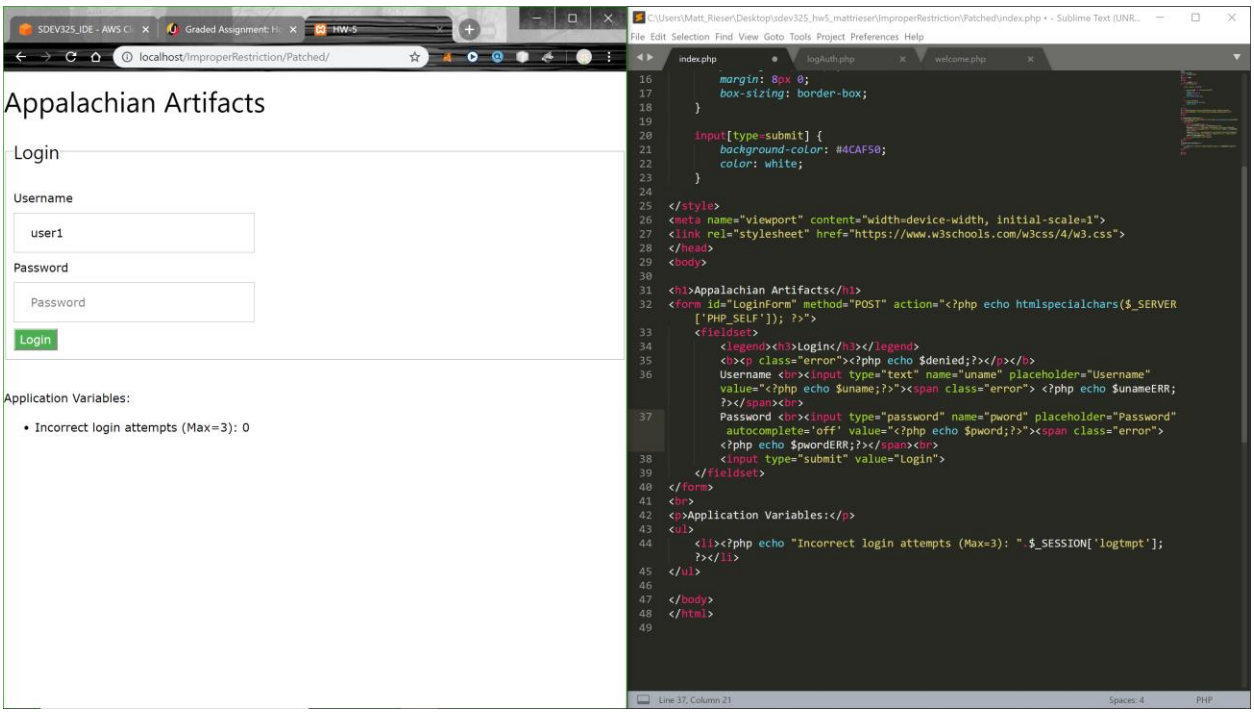

Figure 4. Demonstrating Porous Defenses Final. This figure shows the web applicaiton in Google Chrome browser (left) and it's source code in the Sublime Text (right).

In *Figure 4*, notice the application on the left has added session variables at the bottom of the webpage to better illustrate when the account is in danger of being locked out. The source code on the right reflects this change and shows us that the failed login attempts are tracked by the Session Superglobal array at the index “logtmpt” (line 44).

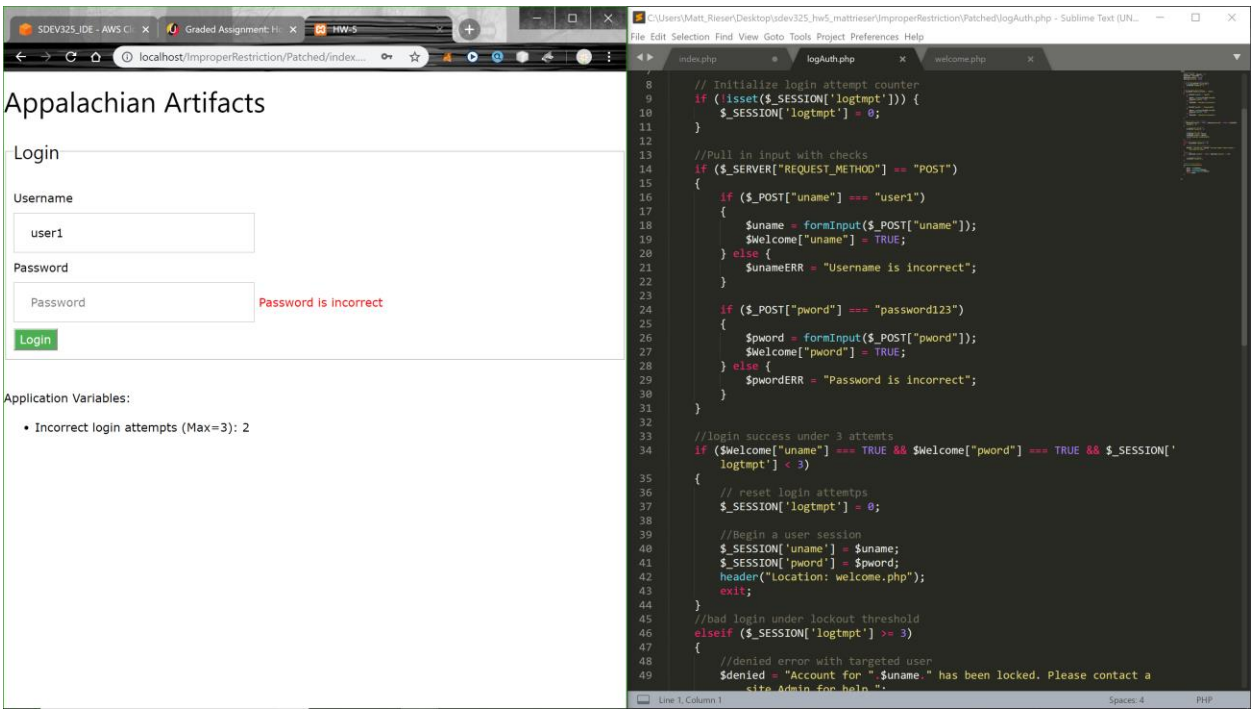

Figure 5. Demonstrating Porous Defenses Final. This figure shows the web applicaiton in Google Chrome browser (left) and it's source code in the Sublime Text (right).

In *Figure 5*, user1's account is being targeted for authentication and has one more attempt before it's locked. In the source code, we can see the initialization logic behind the login attempt tracking (lines 9-11). This sets the attempt counter to 0 if the session has not already been set, thus allowing the variable to persist through multiple attempts and be incremented.

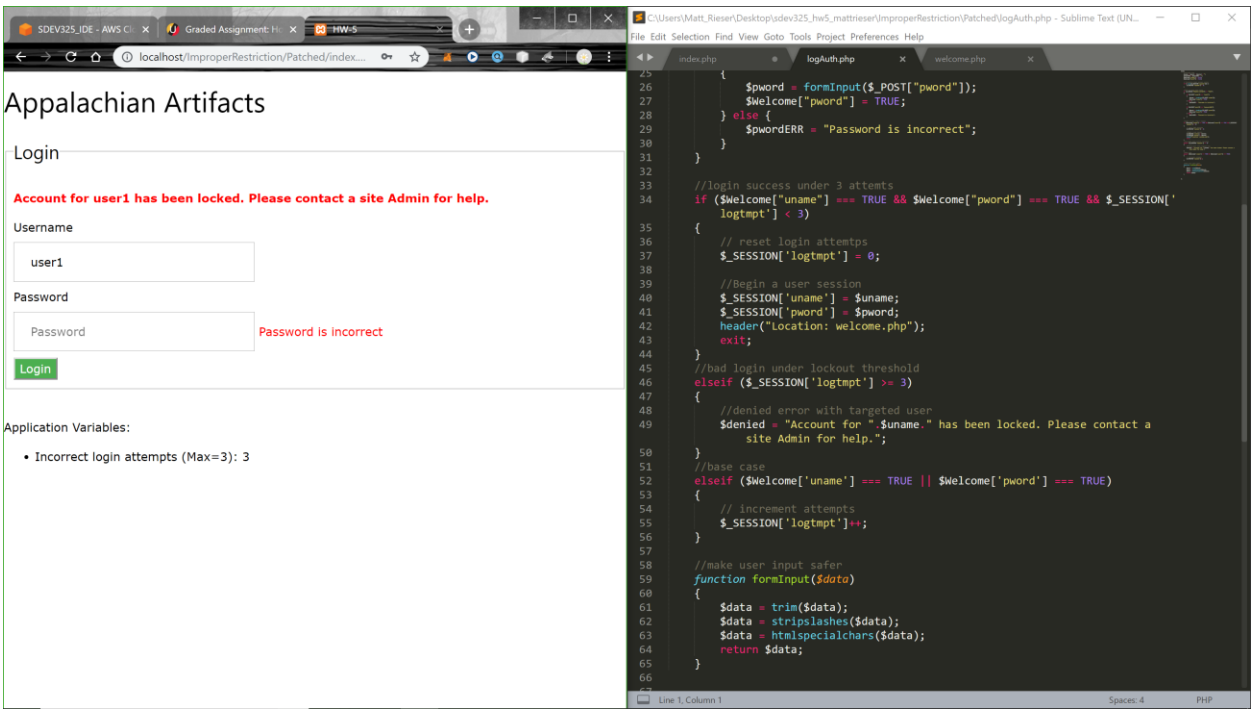

Figure 6. Demonstrating Porous Defenses Final. This figure shows the web applicaiton in Google Chrome browser (left) and it's source code in the Sublime Text (right).

When the account crosses the lockout threshold the application provides a bit of feedback in bold red message prompting the user to contact the site admin (see *Figure 6*). Technically, this will not stop an attacker that notices you can clear the session cookies (**NOTE:** this is how you reset the application) to obtain a fresh set of attempts. Any of the mechanisms described for mitigation purposes will require a proper database implementation to aid the login monitoring and tracking. However, this solution illustrates the issue and is much easier to setup and test.

Lines 34-56 is the heart of the authentication mechanism (see *Figure 6*), which is comprised of three cases:

1. Case 1 (line 34) – checks the correct username, password, and less than 3 login attempts, which allows for the correct number of attempts since we start the count at 0. If this route is triggered by the program these things will happen:
    a. Session login attempt count reset (line 37).
    b. Relevant variables stored for later use in the session (lines 40, 41).
    c. The user is redirected to **welcome.php** page and the script exits (lines 42, 43).
2. Case 2 (line 46) – checks if the login attempts have hit the threshold. Should thereof be true the program will:

   a. Set the denied error message about contacting the admin for help.
3. Case 3 (line 52) – check if the username or password was set and increment the login attempts by +1. (In hindsight, having the login attempts incremented if the username if "user1" would be better functionality. However, this will work because the application assumes every attempt is for user1, even if the username is not set accordingly.)

Figure 7 shows that a successful authentication attempt would result in the **welcome.php** page being displayed and the user1 greeting as the header. The code also shows the login attempts being reset which is redundant since they are previous wiped out in Case 1.

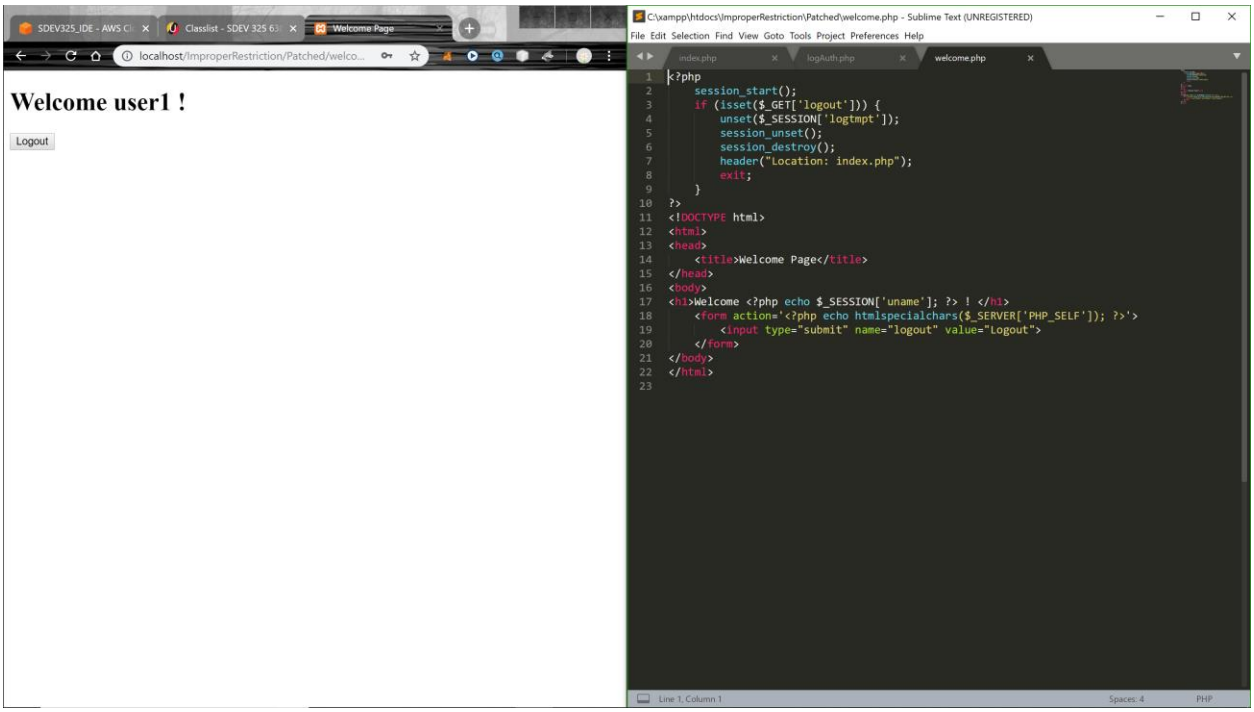

Figure 7. Demonstrating Porous Defenses Final. This figure shows the web applicaiton in Google Chrome browser (left) and it's source code in the Subliime Text (right).

Use of a One-Way Hash without a Salt

This application is a simple Java program that covers Salting hashes and broken cryptographic function.

Very simple application with a main, some login logic, and a function for the heavy crypto lifting.

Vulnerable Application

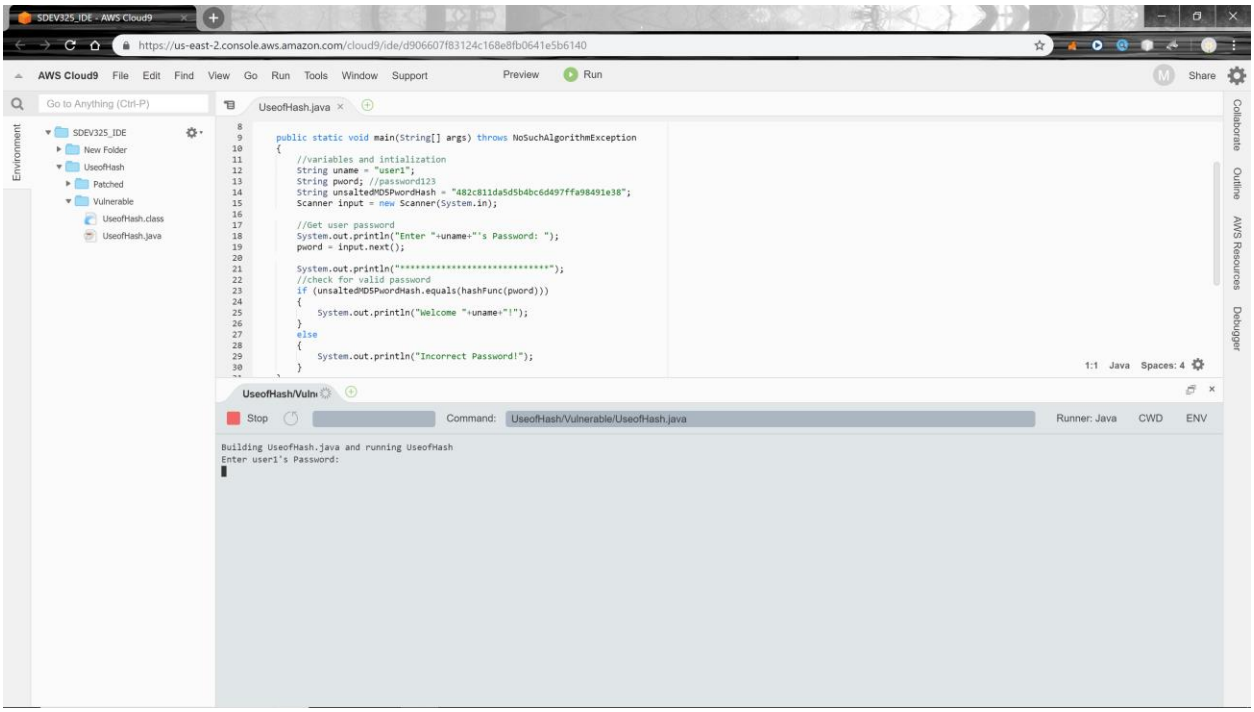

Figure 8. Demonstrating Porous Defenses Final. This figure shows AWS Cloud 9 environment, Java source code (top), and the output console (bottom).

In *Figure 8*, the running application is waiting for the user to enter their password in the output console. Lines 14-16, can be thought of as the database of user. Again, this simplifies the set up and testing but still illustrates the weaknesses. The program must be run for every authentication attempt that is made, which will suffice since we are here for the behind the scene magic.

Notice that the hashFunc() function (lines 35-47) takes a string and returns, in this case, an MD5 (line 42) hash of the string that was passed to it as its only parameter. Lines 38-40, return null if the string passed to the function is null.

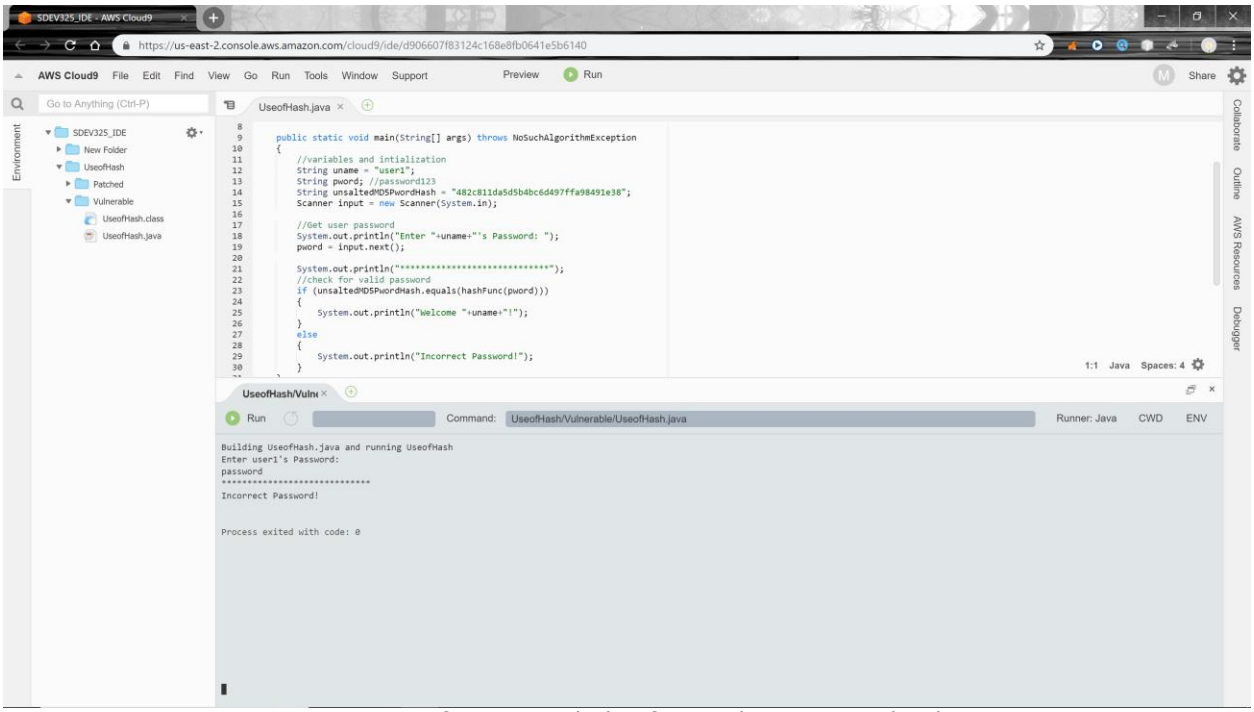

Figure 9. Demonstrating Porous Defenses Final This figure shows AWS Cloud 9 environment, Java source code (top), and the output console (bottom).

In *Figure 9*, the user has submitted the incorrect password and is informed before the program terminates. The application logic behind this logic mechanism is the block of source code from line 25-32, this check the password against the MD5 representation of the user's password from the "database". The import thing to understand here is that an attacker has a significantly less difficult time computing the password for two reasons: 1. MD5 hashes should never be used to obfuscate user passwords because it has become obsolete due to the rapid increase in computing power. 2. The hashFunc function never salts the password before the password is hashed and verified against database.

The reason we salt hashes is to increase the computation power and time required to effectively guess the password by brute force. By following the OWASP recommended best practice of salting hashed passwords, the level of complexity renders brute force attacks prohibitively inefficient for attackers.

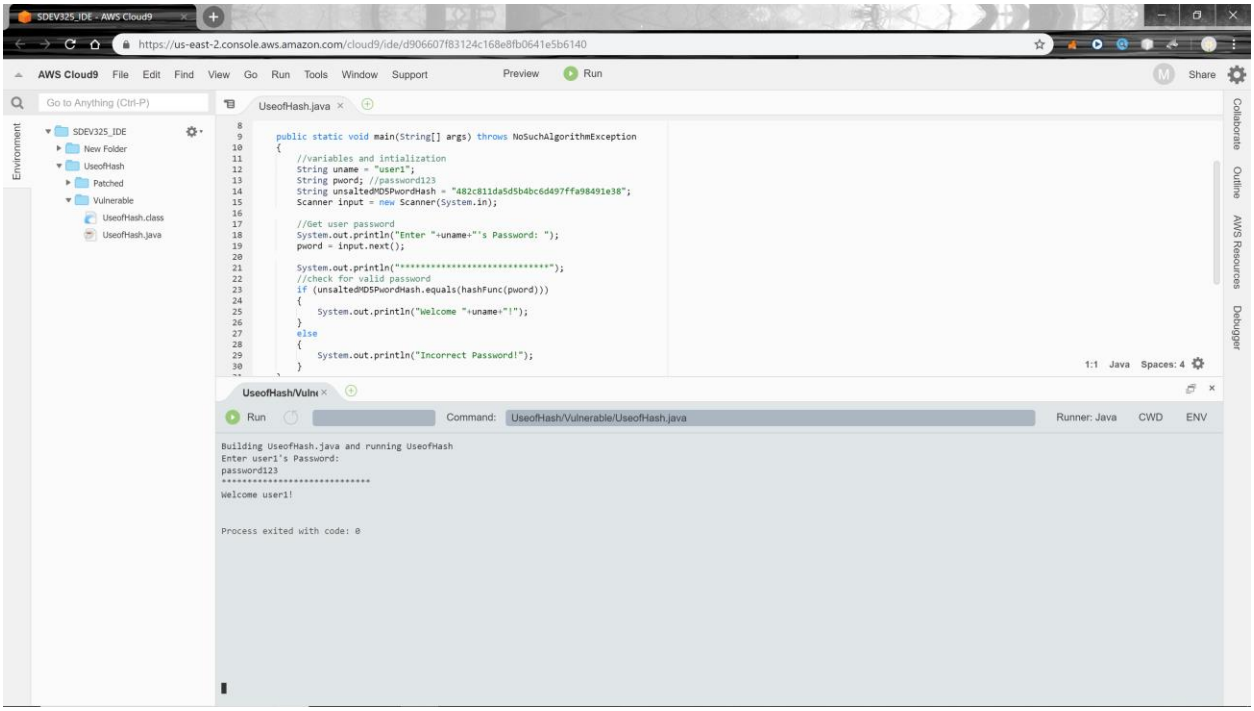

Figure 10. Demonstrating Porous Defenses Final. This figure shows AWS Cloud 9 environment, Java source code (top), and the output console (bottom).

The application successfully authenticated the user before terminating (see *Figure 10*).

Patched Application

In this version of the Java user authentication application we will add salting to hashFunc and switch the cryptographic method from MD5 to SHA-256. The application structure is retained from the vulnerable version.

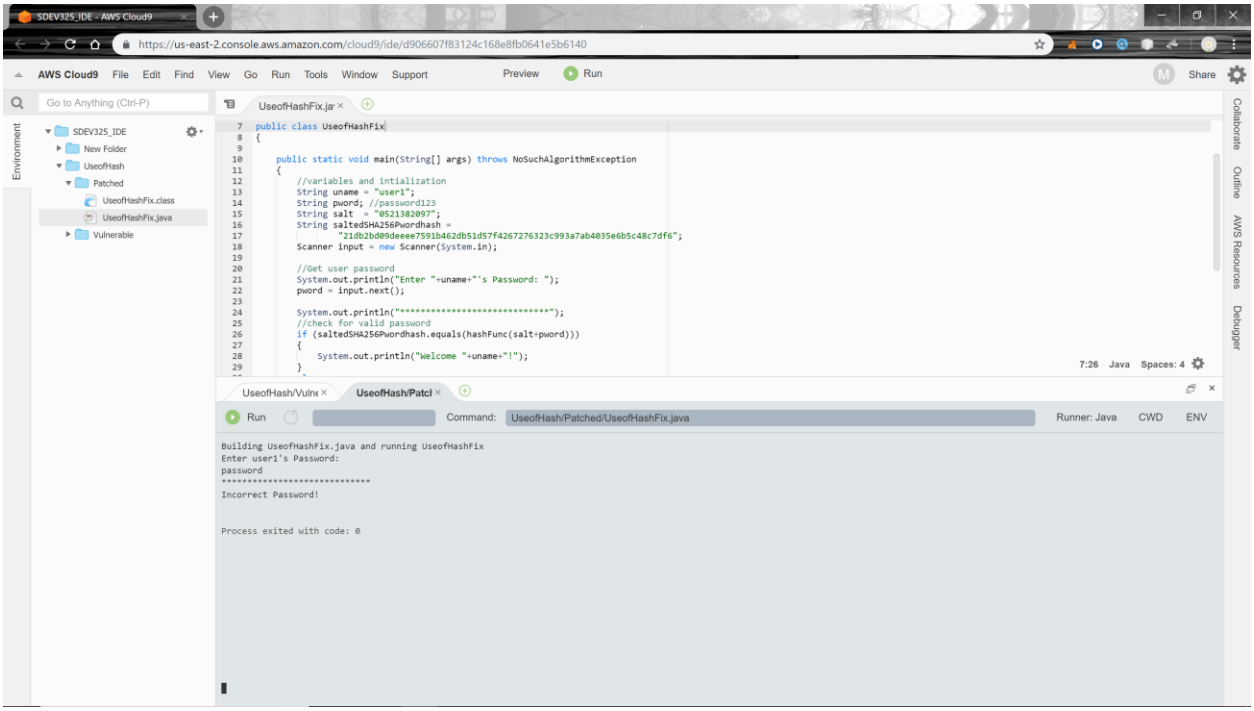

Figure 11. Demonstrating Porous Defenses Final. This figure shows AWS Cloud 9 environment, Java source code (top), and the output console (bottom).

In *Figure 11*, we have added a few things but the basic structure of the program remains largely unchanged. The database now contains a variable salt which hold a randomly generate ten-character string array. The hash representation of the user's password has been updated to SHA-256 (line 44) to mitigate the CWE-327, since MD5 is not secure. Notice, line 27, concatenates salt with pword then hashes both, thus mitigating CWE-759. In a proper implementation of this with a SQL database would randomly generate a salt for each user and store for that user.

Since the incorrect password was enter in *Figure 11*, we see the error message before application termination.

NOTE: the actual password is commented next to the variable pword in the user "database" block of source code.

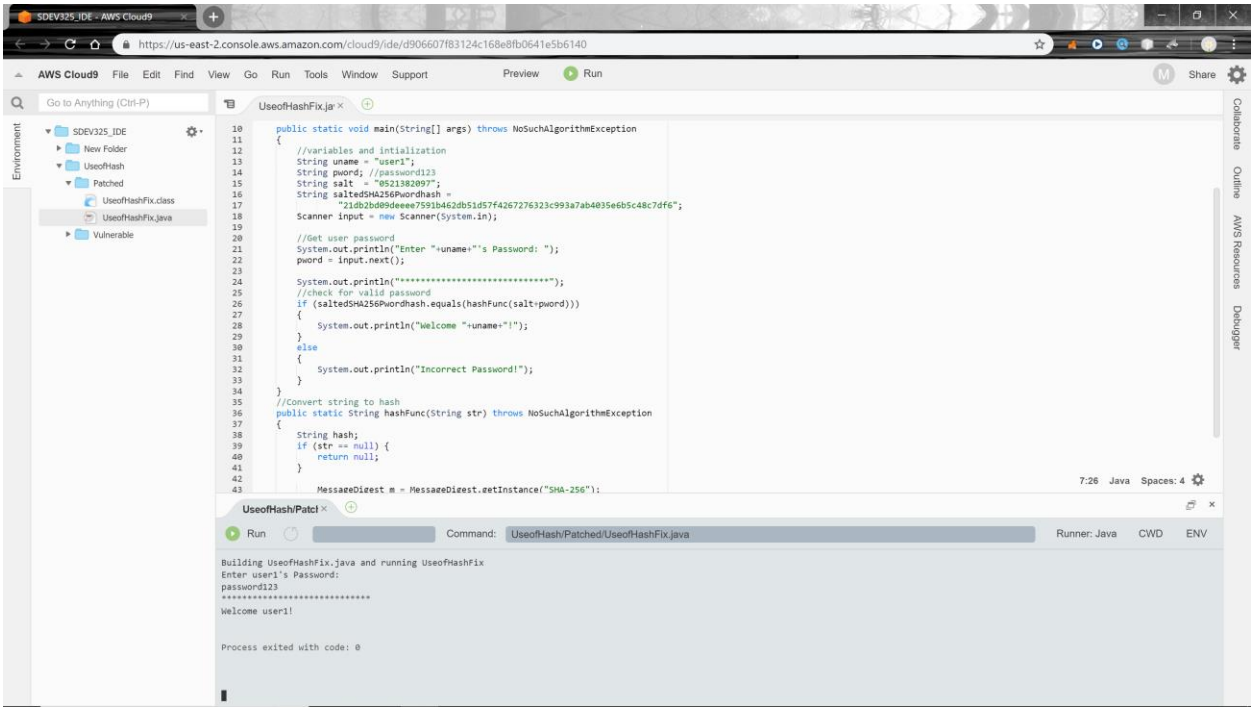

Figure 12. Demonstrating Porous Defenses Final This figure shows AWS Cloud 9 environment, Java source code (top), and the output console (bottom).

In *Figure 12*, the application successfully authenticates the user before terminating.