9.2 Solutions and Guided Walkthrough

9.2.1 Solutions

The code for the hardened web application (after mitigating against the vulnerabilities) is available on the CYOTEE GitHub page at: *https://github.com/CenterForSecureAndDependableSystems/CYOTEE/*. You can copy the files from the hardened application to your local web server's /var/www/html directory. If you copy the files, discuss with your peers and/or colleagues how you would have approached the tasks if starting from scratch.

1. Implement and run the vulnerable web application

   Follow the walkthrough to set up the vulnerable application

2. Identify where in the vulnerabilities exist

   - SQL Injection

     This vulnerability exists in the PHP file in which a static MySQL command has user inputs placed in via the HTTP GET method

   - Lack of Least Privilege Implementation

     This vulnerability exists because the same database user accesses all tables in each PHP file

   - Lack of Input Validation

     This vulnerability exists because the input the user places into the start date field is not validated

3. Mitigate against vulnerabilities

   - Mitigate against SQL Injection

     The use of prepared statements in PHP can prevent against some forms of SQL injection.

   - Implement a form of least privilege so that only necessary MySQL users have access to their respective pages

Create a new MySQL user for each table in the database which must be accessed. Grant each user access to their respective table. Set that user as the user connecting to the database in the PHP files.

- Perform input validation on the date field to ensure that a valid date is entered
  Rather than just accepting any input from the user, ensure that the value is in date format ***MM/DD/YYYY*** *and that the date is question actually occurred.*

9.2.2 Guided Walkthrough

In order to complete the challenges in this laboratory exercise, see the steps in this walkthrough.

Least Privilege

Least privilege is a concept used daily, but seems to seldom be implemented in computing. At the simplest level, the principle of least privilege applies to an asset and a set of individuals who may access the asset. The principle of least privilege suggests that only individuals who absolutely require access to the asset should be authorized for that access. Consider the following: You and your family have moved into a new home. You make four copies of the house key, one for each member of the family. Upon more thinking, you come to the realization that your children are dropped off and picked up from school so they never enter or exit the house on their own and therefore, giving them a key is unnecessary. If the children were to lose the key, the security of home has been compromised. By recognizing that your children never needed their own keys, you have implemented a least privilege scheme in which only individuals who require access to an asset (in this case a key) are granted that access. This is an example of implementing least privilege in the real world.

Web applications require multiple layers of security. One of the simplest layers of security to implement in web applications is least privilege; despite being simple, it frequently goes unimplemented. A typical web application will interact with a database via PHP. A simple way of implementing least privilege is to create multiple database users and grant each user different privileges based on what actions they must perform. An in depth guide of how to perform these

tasks will be outlined in the **Implementation** section [36].

SQL Injection

Many web applications include some sort text entry field to allow the user to enter data such as a username and/or password among other possibilities. The value entered into the text field is then typically used in some sort of a structured query language (SQL) query [37]. SQL is used to manage and query relational databases in computing. Queries allow a user to read data from, insert data into, and delete data from a database, among other actions. Database management applications such as MySQL [38] have specific syntax for queries. A simple web application with little security implemented will typically insert the user entered value directly into a partially constructed MySQL query. For example, a sample MySQL query in which the user entered value is represented by the variable `$userval` may look something like:

```
SELECT * FROM table WHERE user='$userval';
```

In this case, the query, when executed, will allow the user to view all data fields for a row in which the **user** field has the value indicated by the user input. When using partially constructed queries into which the user's value is dropped in, the SQL Injection [39] vulnerability is introduced.

A malicious user can take advantage of the fact that the web application takes the user input as is and enters it into the partially constructed query. If the malicious would like to be able to view all rows of data from the table, he/she can enter a specifically formatted value into the text input field to achieve the task. An example of this input is:

```
'  OR '1=1
```

Although this input looks like something out of a poorly written math textbook, let us identify what it does to the query. By inserting that value into the partially constructed query, the full query is:

```
SELECT * FROM table WHERE user='' OR '1=1';
```

Analyzing this query, the malicious intent is visible. The malicious user has formed a query such that MySQL will return a row if either the name field is blank, ***or*** if 1=1 (this always evaluates to true). Because 1=1 always evaluates to true, MySQL will return every row in the

table. Other values can be entered into the input field to perform other SQL queries.

SQL Injection Mitigation

Given that allowing a user to input a value through a text field introduces a vulnerability, the obvious question arises: "What do I do if I need to let the user enter a value?" One suggestion may be to check the value that the user entered and analyze it for potential SQL Injection; upon trying to perform this task, one finds it to be quite futile as there are an infinite number of ways a malicious user could format their input to perform a malicious task. Rather, it is best to handle the actual issue: the user input is inserted into the query exactly as it is read in. Rather than allowing the user input to be placed into the partially constructed query, it would be useful to have everything the user enters interpreted as a string literal. This can be accomplished through the use of what are known as *prepared statements* [40].

Prepared statements can be implemented through PHP to allow your web application to interact with the MySQL database. A prepared statement is implemented in three stages:

1. Prepare
2. Bind
3. Execute

In the preparation stage, the user writes a SQL statement template with certain values left unfilled. These values are specified by question marks (?) and are called parameters. Using the same query as discussed in the SQL Injection section, a sample PHP line to prepare a statement would be:

```
$query = $conn->prepare("SELECT * FROM table WHERE name=?");
```

In the bind stage, the user specifies what values should be used to fill the previously ambiguous fields identified by question marks. In addition to specifying the value, the user must also specify the type of the value. A sample PHP line to bind parameters would be:

```
$query->bind_param("s",$userval);
```

In the execution phase, the user simply specifies that he/she would like to execute the query which has been prepared and bound. A sample PHP line to execute the SQL query would be:

```
$query->execute();
```

Implementing prepared statements over traditional dynamic MySQL queries in PHP is not a challenging but improves the security of the web application significantly. Because the user's input value is being interpreted literally and there are no quotes to escape properly in the prepared statement as there were in the traditional statement, a malicious user cannot carry out the same SQL Injection described previously.

User Input Validation

Web applications often allow a user to enter data into a field which must be of a certain format, for example a birthday. However, web applications often do not validate that the value entered was actually in the format required. Consider the following application: employees at a company must enter the date they started working for the company. In the event that the desired input format is **MM/DD/YYYY**, what are some possible ways in which invalid data could be entered? A few possibilities which commonly strike a developer are that maybe the user enters the date in the wrong format such as **DD/MM/YYYY** or **MM/DD/YY**. Another common mistake could be that the user simply does not enter a date and instead provides some sort of bogus input.

Provided that the user did actually enter a date in the correct format, another check which should be performed is whether the date entered is a valid date. Examples of invalid dates include **11/31/2019** and **02/29/2019**. The first of the two invalid dates is invalid because November never has 31 days in the Gregorian calendar. The latter of the two is invalid because 2019 was not a leap year, therefore meaning February 29, 2019 never occurred.

Implementation

In order to demonstrate the steps for taking a vulnerable, basic web application and implementing the aforementioned security measures, a series of challenges has been created. To complete these challenges, an individual will have to successfully implement each of the security measures and therefore create a more secure web application. A detailed walkthrough of how this exercise was developed, as well as how you can recreate it is outlined in the following sections.

Implementing Least Privilege

Note that in the vulnerable web application, the PHP pages which establish communications with the MySQL database have the root user being used to logon to MySQL. The root user is granted what is known as **ALL PRIVILEGES** in MySQL; this means the root user has the ability to perform any action, on all tables in any database, in MySQL. Inspecting the PHP pages will reveal that one of the pages performs a query using the **SELECT** command and the other performs a query using the **INSERT** command.

This allows a secure programmer to implement least privilege. There is no reason to have the root user logging into the database from the PHP page, rather create a new user. Moreover, that new user only needs to have SELECT privileges on the table referenced in the code (in this case, the **employees** table in the **test** database. Additionally, another user can be created and granted only INSERT privileges on the **startdates** table in the **test** database.

The first step towards creating the new users is to log into MySQL using the root user. As a reminder, this can be done by entering the following command in the terminal and entering the root user password:

```
$ sudo mysql -u root -p
```

The next step is to create the two new users; in order to do so, enter the following command in the MySQL console:

```
mysql> CREATE USER '<webapp1>'@'localhost' IDENTIFIED BY "<webapp1>";
mysql> CREATE USER '<webapp2>'@'localhost' IDENTIFIED BY "<webapp2>";
```

You can replace the usernames and passwords (indicated by blue text) with whatever you would like but those are the values which are used in the final code.

Now that two new users have been created, it is necessary to grant the appropriate privileges to each user. Currently the users only have **USE** privileges. In order to grant privileges, enter the following into the MySQL console:

```
mysql> GRANT SELECT ON test.employees TO 'webapp1'@'localhost';
mysql> GRANT INSERT ON test.startdates TO 'webapp2'@'localhost';
```

This will allow the **webapp1** user to read data from only the **employees** table and the

Listing 9.2: *Vulnerable Authorization Query*

```
$sql = "SELECT * FROM employees WHERE name='" . $_GET["user"] . "' AND password='
    " . $_GET["pass"] . "';";

$result = $conn->query($sql);
```

Listing 9.3: *Hardened Authorization Query*

```
$sql = $conn->prepare("SELECT * FROM employees WHERE name=? AND password=?");

$sql->bind_param("ss",$_GET['user'],$_GET['pass']);

$sql->execute();
```

webapp2 user to insert data into only the **startdates** table, both in the **test** database.

Utilizing Prepared Statements

The use of prepared statements is one which allows a web application developer to mitigate against SQL Injection. As discussed previously in this document, prepared statements take the user input and insert them as string literals into the prepared statement rather than inserting the variable value directly into the dynamic SQL query template. In the web application, there are two files which require the use of user input in a SQL query; both of these files should implement prepared statements. In the file *vw-auth.php*, the original (vulnerable) lines associated with the query can be seen in listing **9.2**.

After implementing the prepared statements in the **hw-auth.php** file, the lines associated with the query can be seen in listing **9.3**.

In the file *vw-insert.php*, the original (vulnerable) lines associated with the query can be seen in listing **9.4**.

After implementing the prepared statements in the **hw-insert.php**, the lines associated with the query can be seen in listing **9.5**.

Listing 9.4: *Vulnerable Insertion Query*

```
$sql = "INSERT INTO startdates (name, date) VALUES (\"$user\",\"$date\");";

$conn->query($sql);
```

Listing 9.5: *Hardened Insertion Query*

```
$sql = $conn->prepare("INSERT INTO startdates (name,date) VALUES (?,?)");

$sql->bind_param("ss",$user,$date);

$sql->execute();
```

Listing 9.6: *Date Format Matching Regex*

```
$regex = "/(\d{2})\/(\d{2})\/(\d{4})/";

preg_match($regex,$date,$matches);
```

Viewing the differences between the vulnerable code and the hardened code will reveal that there is not much extra work required to implement the prepared statement. Due to the prepared statement formatting the string as a literal, the web application developer does not need to ensure that the double and single quotes are placed and escaped appropriately in the SQL query. In making the change from the vulnerable code to the hardened code using prepared statements, the process for returning the result of the query and printing is also changed slightly.

Performing Input Validation

Input validation may very well be a programmer's dream and nightmare simultaneously. A dream because it improves the security of the application but a nightmare because of the amount of effort needed to perform input validation. It can be a tedious task, particularly considering the potential edge cases for what types of input could be considered invalid. In this web application, input validation should be performed when the end-user is tasked with entering a date on the *hw-insert.php* page. At this point, prepared statements have been implemented in this file but nothing has been done to ensure that the user entered a valid date. As noted previously in this document, there are a few checks which should be performed in relation to input validation of the input provided on the *hw-insert.php* page.

The page requests that the user insert a date in **MM/DD/YYYY** format. The first check which should be performed is whether or not the input is even in this format. This task can be accomplished using regular expression (regex) matching. The lines of code seen in listing **9.6**, in the hardened *hw-insert.php*, file accomplish this task.

Listing 9.7: *Split Date into Substrings*

```
$da = explode("/",$date);
$m = intval($da[0]);
$d = intval($da[1]);
$y = intval($da[2]);
```

The code seen in listing **9.6** simply checks if the input provided by the user (held in the `$date` variable) matches the regex. The regex will match any input which is in the format of two digits, followed by a forward slash, followed by two digits, followed by a forward slash, followed by four digits (digits can be any number between 0 and 9).

After determining that the input entered is in the correct **MM/DD/YYYY** format, one should begin checking for other invalid inputs such as invalid dates. Examples of invalid dates include the following:

- If the month portion is less than 1 or more than 12.
- If the day portion is less than 1 or more than 31.
- If the month portion is April, June, September, or November and the day portion has more than 30.
- If the month portion is February and the day portion is more than 29.
- If the month portion is February and the day portion is 29 and the year portion is not a leap year.

These checks are easily performed in PHP but first some overhead is required. The date field is entered in as a whole in **MM/DD/YYYY** format but in order to perform the inequality checks listed above, the portions of the date must be split into individual parts. This is done by splitting the string on a delimiter character. The natural choice for the delimiter character in this case is the forward slash (**/**). This is done in PHP by using the **explode** function. The function returns an array of the substrings obtained by splitting on the delimiter character. The code seen in listing **9.7**, performs the splitting and stores the individual parts of the date in individual variables.

Listing 9.8: *Validate Month Value*

```
if($m < 1 || $m > 12)
{
        header("Location: hw-invaliddate.html");
        exit();
}
```

Listing 9.9: *Validate Day in All Months*

```
if($d < 1 || $d > 31)
{
        header("Location: hw-invaliddate.html");
        exit();
}
```

With the individual parts of the date stored in individual variables, inequality checks can be performed. Refer to the list below for the code snippets:

- If the month portion is less than 1 or more than 12, code seen in listing **9.8**.
- If the day portion is less than 1 or more than 31, code seen in listing **9.9**.
- If the month portion is April, June, September, or November and the day portion has more than 30, code seen in listing **9.10**.
- If the month portion is February and the day portion is more than 29, code seen in listing **9.11**.
- If the month portion is February and the day portion is 29 and the year portion is not a leap year, code seen in listing **9.12**.

Listing 9.10: *Validate Day in 30 Day Months*

```
if($m == 4 || $m == 6 || $m == 9 || $m == 11)
{
        if($d == 31)
        {
                header("Location: invaliddate.html");
                exit();
        }
}
```

Listing 9.11: *Validate Day in February for Any Year*

```
if($m == 2)
{
        if($d > 29)
        {
                header("Location: invaliddate.html");
                exit();
        }
        ...

}
```

Listing 9.12: *Validate Day in February for Leap Years*

```
if($m == 2)
{
        ...

        if($d == 29)
        {
                if( (($y % 4 == 0) && ($y % 100 != 0)) || ($y % 400 == 0) )
                {
                        //this is a leap year, which is valid
                }
                else
                {
                        header("Location: invaliddate.html");
                        exit();
                }
        }
}
```

Hardened Code

The hardened code, file by file, can be seen in the following listings, along with their respective file names.

Listing 9.13: *hw-index.html*

```
<html>
	<body style="background-color: black; color: white">
		<div style="text-align: center">
			<form action="auth.php" method="get">
				Username: <input type="text" name="user"><br>
				Password: <input type="text" name="pass"><br>
				<input type="submit">
			</form>
		</div>
	</body>
</html>
```

Listing 9.14: *hw-auth.php*

```php
<?php

$servername = "localhost";
$username = "webapp1";
$password = "webapp1";
$dbname = "test";

$conn = new mysqli($servername,$username,$password,$dbname);

$sql = $conn->prepare("SELECT * FROM employees WHERE name=? AND password=?");

$sql->bind_param("ss",$_GET['user'],$_GET['pass']);

$sql->execute();

$result = $sql->get_result();

if($result->num_rows > 0)
{
        print "Click you name to access your date entry form!<br>";
        while($row = $result->fetch_assoc())
        {
                $n = $row['name'];
                print "<a href='insertdate.php?name=$n'>" . $n . "</a><br><br>";

        }
}
else
{
        header("Location: invalid.html");
}

$conn->close();
?>
```

Listing 9.15: *hw-insertdate.php*

```php
<?php
        $n = $_GET['name'];
?>

<html>
        <body style="background-color: black; color: white">
                <div style="text-align: center">
                        <form action="insert.php" method="get">
                                Enter a date in MM/DD/YYYY format<br>
                                <input type="text" name="date"><br>
                                <input type="text" name="user" value="<?php echo $n;
                                    ?>" readonly><br>
                                <input type="submit">
                        </form>
                </div>
        </body>
</html>
```

Listing 9.16: *hw-insert.php*

```
<?php

$servername = "localhost";
$username = "webapp2";
$password = "webapp2";
$dbname = "test";

$user = $_GET['user'];
$date = $_GET['date'];

$regex = "/(\d{2})\/(\d{2})\/(\d{4})/";
preg_match($regex,$date,$matches);

if($matches == NULL)
{
        header("Location: invaliddate.html");
        exit();
}
else
{
        $da = explode("/",$date);
        $m = intval($da[0]);
        $d = intval($da[1]);
        $y = intval($da[2]);

        if($m < 1 || $m > 12)
        {
                header("Location: invaliddate.html");
                exit();
        }

        if($d < 1 || $d > 31)
        {
                header("Location: invaliddate.html");
                exit();
        }

        if($m == 4 || $m == 6 || $m == 9 || $m == 11)
        {
                if($d == 31)
                {
                        header("Location: invaliddate.html");
                        exit();
                }
        }

        if($m == 2)
        {
                if($d > 29)
                {
                        header("Location: invaliddate.html");
                        exit();
                }

                if($d == 29)
                {
                        if( (($y % 4 == 0) && ($y % 100 != 0)) || ($y % 400 == 0) )
                        {
                                //this is a leap year, which is valid
```

```
                        }
                        else
                        {
                                header("Location: invaliddate.html");
                                exit();
                        }
                }
        }
}


$conn = new mysqli($servername,$username,$password,$dbname);

if($conn->connect_error)
{
        die("Connection Failed: " . $conn->connect_error);
}

$sql = $conn->prepare("INSERT INTO startdates (name,date) VALUES (?,?)");
$sql->bind_param("ss",$user,$date);
$sql->execute();

?>

<html>
        <body style="background-color: black; color: white">
                <div style="text-align: center">
                        <h1>Insert Complete!</h1>
                        <button onclick="location.href='index.html'">Return to Login
                             Page</button>
                </div>
        </body>
</html>
```

Listing 9.17: *hw-invaliddate.html*

```
<html>
        <body style="background-color: black; color: white">
                <div style="text-align: center">
                        <h1>Invalid Date Entered!</h1>
                        <button onclick="location.href='index.html'">Return to Login
                             Page</button>
                </div>
        </body>
</html>
```

Listing 9.18: *hw-invalid.html*

```
<html>
        <body style="background-color: black; color: white">
                <div style="text-align: center">
                        <h1>Invalid Logon Attempt!</h1>
                        <button onclick="location.href='index.html'">Return to Login
                             Page</button>
                </div>
        </body>
</html>
```