Secure Programming module, Formative Assignment - Answers

Answer - 1:

The fundamental problem in the code is the mixing of Code and Data through trusted user input. The said mixing allows any attacker to exploit this code to make the interpreter see the data part as part of the code itself.

An example of such an exploit could be a specially crafted query in the form of **Listing 1** and the subsequent interpreted code as shown in **Listing 2**.

Listing 1: SQL Injection Example

```
' or author='admin'; --
```

Listing 2: SQL Injection Example

```
$author = $_GET['author'];
$query = "SELECT (*) FROM books WHERE author = '' or author='admin'; -- '";
$result = mysqli_query($query);
```

The two counter-measures to this kind of injection are:

1. **Prepared Statements**

   Prepared Statements provides a way to separate a SQL Query (Code) and the input (Data) by first preparing SQL queries and then binding it with the given input as shown in **Listing 3**.

   Listing 3: Using Prepared Statements to secure the code

   ```
   $author = $_GET['author'];
   // create a prepared statement
   $stmt = $conn->prepare("SELECT (*) FROM books WHERE author =:name");

   // bind parameters
   $stmt->bindParam(":name", $author)

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

2. **Stored Procedures**

   Stored Procedures work by storing the SQL Query (Code) in the form of specialized data structure called a **procedure**. (Note: This in itself, could be a cause of concern since it inherently trusts the procedure data structure to be well-formed and from a security standpoint, it might not be the best technique)

   Listing 4 demonstrates a stored Procedure for the given query and **Listing 5** shows the subsequent PHP code.

   Listing 4: Stored Procedure in SQL

   ```
   CREATE PROCEDURE searchByAuthor @author nvarchar(20)
   AS
   select * FROM books WHERE author = @author
   GO;
   ```

Listing 5: Using Stored Procedure to secure the code

```
$author = $_GET['author'];
// create a prepared statement
$stmt = $conn->prepare("CALL searchByAuthor(:author)");

// bind parameters
$stmt->bindParam(":name", $author, PDO::PARAM_STR);

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

Answer - 2:

An HTTPOnly Cookie as shown in **Listing 6** is a special form of cookie that can be included (if the browser supports it) in requests to disallow access to the cookie through any kind of client-side script (such as JavaScript).

This helps in preventing XSS attacks as they rely on client-side scripts to steal cookies which can be restricted adding the *HTTPOnly* flag to the cookie.

Note: The use of HTTP-only cookies is one of several techniques that, when used together, can mitigate the risk of cross-site scripting. Used alone, it cannot completely eliminate the danger of cross-site scripting.

Listing 6: HTTPOnly Cookie

```
Set-Cookie: USER=123; expires=Wednesday, 09-Nov-99 23:12:40 GMT; HttpOnly
```

Answer - 3:

1. The different kinds of UID comes from a fundamental need to control and monitor access control privileges in an Operating System.

   The UNIX Solution consists of these Unique IDs that can be assigned to a process namely effective UID or euid, real UID or uid and saved UID.

   - **Real UID:**
     This type of UID determines who owns the process. A process can set its real UID using the ***setuid()*** function call and it can get its real UID by using the ***getuid()*** function call.
   - **Effective UID:**
     This type of UID tells the kernel about the kind of permissions the process has while it accesses shared resources such as shared memory and semaphores. Effective UID can get procured by a process through the ***geteuid()*** function call and set by the ***seteuid()*** function call.
   - **Saved UID:**
     Saved UID acts as a saving space for the effective UID set when a program is executed and stores those effective UIDs for later use. It can be set by using ***setresuid()*** and its value can be acquired by calling the function ***getresuid()***.

2. When compiled by **root** and executed by **root** the given program outputs as shown in **Listing 7**.

Listing 7: output of C program

```
Opening  attempt 1 succeeded !
Opening  attempt 2 failed !
Opening  attempt 3 succeeded !
Opening  attempt 4 failed !
setuid  failed !
```

Going through the program we can see that the following happens:

(a) Since the ***testFile()*** function is called with the effective and real UID of ***root*** (which is 0) in the beginning and because it's the root which executed and owns the process, we get the first message indicating that we were able to open the file. (Since ***root*** is allowed to read from */etc/shadow*)

(b) Next, the ***seteuid(500)*** function is called with a UID of *500* and since all UID's above 0 are non-root UIDs, the program is running with non-root privileges now or has effectively dropped its privileges. The problem comes when we try to read the */etc/shadow* file since reading it requires the program process to have a privileged UID and thus we get the failing message. However it's worth noting that the real UID of the program is still 0 and thus the program has enough privileges to escalate its effective UID to a privileged one namely ***root***.

(c) Now, through the use of ***seteuid(0)***, the program escalates it's effective privileges by setting its effective UID to 0 which gives it the ***root*** privileges again. Hence, we get the third message of successfully opening the */etc/shadow* file.

(d) Here with the function call, ***setuid(500)*** the real UID of the program process is now changed to 500 which is a non-privileged ID. The difference here is since the real ID has be changed, the real owner of the program process has changed. Thus it has dropped it's real privileges and cannot escalate privileges to ***root*** since it doesn't have permissions to do that anymore. Therefore, we see the fourth attempt failed message.

(e) Since we had changed the real UID of the process in the earlier function call it is no longer possible to escalate privileges through ***setuid(0)*** since in order to escalate the real UID we already need to have the sufficient privileges. Therefore, we get the setuid failed message.

Answer - 4:

The fundamental problem in computer security is the nature of its complexity and its fragility in the face of relying on any single control measure that seemingly provides complete security. The *defence-in-depth* approach is a solution to this problem.

Defence-in-depth is a philosophy of looking at problems in Computer Security in a layered manner thereby assessing and securing every possible layer in a problem through various well-implemented and conservative procedures.

This type of approach allows for acceptable level of security even in the case of multiple failures across different layers.

An example of this approach can be found in counter-measures when dealing with injections in general, the three layers of defence are:

1. **Filtering**:

   The first line of defence includes filtering the Code part from the user input (Data) For e.g. removing the use of <script> tag in user input. The problem with filtering is that it requires a lot of coverage of all possible user input characters that could be deemed dangerous (Blacklisting). An example of filtering is shown in **Listing 8**.

   Listing 8: Example of filtering an email input in PHP

   ```
   <?php
   $email = "john.doe@example.com";

   // Remove all illegal characters from email
   $email = filter_var($email, FILTER_SANITIZE_EMAIL);

   // Validate e-mail
   if (!filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
       echo("$email is a valid email address");
   } else {
       echo("$email is not a valid email address");
   }
   ?>
   ```

2. **Encoding:**

   Encoding tries to replace HTML markups and other special characters with alternative safe representations such as URL-encoding the *<script>* tag to *&lt;script&gt;*. An example of Encoding is shown in **Listing 9** and the output is shown in **Listing 10**. Encoding also suffers from the problem of writing a safe and correct encoder to handle all possible characters.

   Listing 9: Example of encoding an URL in PHP

   ```
   <?php
   echo '<a href="http://example.com/department_list_script/',
       rawurlencode('sales and marketing/Miami'), '">';
   ?>
   ```

   Listing 10: Output of Encoding in Listing 9

   ```
   <a href="http://example.com/department_list_script/sales%20and%20marketing%2FMiami">
   ```

3. **Stored Procedures/Prepared Statements:**

   As discussed above, Stored Procedures and Prepared Statements help in keeping the Data and Code channels separate and hence they can be used as the last layer of defence in dealing with injections even if Filtering and Encoding doesn't work.

Answer - 5:

- The following is a non-exhaustive list of affected services (on the assumption that the victim could be a student, Teacher or TA):

  - Profile information about Students, Teachers and TAs can be harvested or tampered with.
  - Unauthorized access to the Files section of Students, Teachers and TAs containing sensitive and/or personal data.
  - Approved Integrations could be altered so as to create a persistent backdoor to the Canvas API through code.
  - Assignments could be tampered with.
  - Student Submissions could be seen, modified and tampered with by the attackers
  - Unauthorized feedback on the submissions could occur.
  - ePortfolio information can be changed. Also unauthorized collection of ePortfolios can occur.
  - Unauthorized access to the in-built mail system can wreak havoc on users especially when the victim is a Teacher or TA.
  - Quizzes can be tampered with especially in the case of Attendance Quizzes where incentives for an attack seems high.

- The most harm to a Student, Teacher or TA can be done in the form of the following:
  - The Single Sign-on system getting compromised and letting the attackers sign-in to all the other University of Birmingham web services.
  - Submissions section for the modules is tampered with.
  - The mail system is coerced by the attackers.
  - Files are tampered or deleted.
  - Attendance Quizzes are tampered with.

- The most harm from the perspective of the University would be:
  - The leaking of feedback assessment of the University collected from staff and students.
  - The leaking of sensitive mails shared between Teachers and TAs.
  - The leaking of Module details and schedule.
  - Professor and Students Personal details being leaked.
  - Leaking of announcements containing sensitive information with respective to staffs' personal schedule or internal college events.
  - The video recordings getting leaked.