Lab 4

Dan Beck

December 1, 2020

SDEV-350 7380

Prof. Carl Eyler

Lab 4

1. **Concerns with Overall Functionality**

Upon initial review of the given PHP code, there are a few concerns that stand out. One the best practices for securing a database is to have "secure authentication to the database is used" and that "only authorized users have access to the database" (Database Hardening Best Practices). The database does not have authentication to connect, consequently, if anyone has access to getDB(), then they can connect to the database and access all information in the database. The database needs to have a profile that has roles defined that are granted only to users that are qualified to make changes. This can prevent anyone from accidentally altering, accessing, or deleting data in a database.

Regarding passwords within a database, "strong passwords in the database are enforced when technically possible, and database passwords are encrypted when stored in the database or transmitted over the network" (Database Hardening Best Practices). On the code that is updating a user's information, there is nothing that enforces the password change. The password is also not encrypted or decrypted when being accessed or updated in the database.

Finally, the database is not closed at the end of the code, leaving the database exposed to hackers. It is good practice to close the database after all transactions have been completed to keep the program and database running smoothly.

2. **Recommendations**

Starting with the first line of code, the connection should be established only after a password is entered for a user that has the authority to make changes to the database. Figure 1 is an example of what should be used to gain access to the database. This would ensure that not just anyone with access to the code can access the database.

```
$conn = getDB($databaseAccessUsername, $databaseAccessPassword);
```

Figure 1, adding of username and password to access database

Along with the password to enter the database, it would also be beneficial for the creation of a profile, role, and user that is granted access to those roles, so that only certain employees can make changes to the database. Figure 2 is an example of a profile being created, a user being created, and assigned a role that allows them to create, alter and select data from tables in the database.

```
$sql = "CREATE PROFILE NewProfile LIMIT
    PASSWORD_VERIFY_FUNCTION ORA12C_VERIFY_FUNCTION
    SESSIONS_PER_USER 3
    FAILED_LOGIN_ATTEMPTS 4
    PASSWORD_LIFE_TIME 120
    PASSWORD_LOCK_TIME 1/24;
CREATE USER LoggedInUser
    IDENTIFIED BY LoggedInUser
    DEFAULT TABLESPACE Users
    QUOTA 30M ON Users
    PROFILE NewProfile
CREATE ROLE DBRole;
GRANT alter table, select, insert TO DBRole;
ALTER ROLE DBRole ADD MEMBER LoggedInUser"
```

Figure 2, creation of profile, users and roles that are assigned to user

Password enforcement, encryption and decryption are key to keeping private data secure. Figure 3 is an example of the password being passed to a password complexity method, passed through a password encryption method before being inserted into the database. The passwordComplexity() method could verify that the password meets ORA12C PL/SQL password complexity verification. The passwordEncryption() method could encrypt the password before sending the parameter to the database.

```
if (passwordComplexity($pwd)== true)
{
    echo 'Password is valid!';
}
else
{
    echo 'Invalid password.';
}

if (passwordEncryption($pwd)== true)
{
    echo 'Password is encrypted!';
}
else
{
    echo 'Password cannot be encypted.';
}
```

Figure 3, adding of password encryption and password complexity methods

The end of the code should close the database after the retrieving and updating is complete. Figure 4 shows what should be placed at the end of the code to make sure that the database connection is closed after completion.

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

Figure 4, addition of closing the database upon completion of transactions

Implementing each of these changes will make the code larger and slightly slower to run. The increase in size and execution time will create a more impenetrable barrier against attacks. This grows increasingly important as the databases continues to grow.

3. SQL Injection

With the given PHP code, SQL Injections would raise cause for concern. Figure 5 shows the simple query, ‘0 OR 1=1’, that can be sent into the database. “When this query is passed to the database, it will return all the user’s information it has stored, and rows are added to the page

to show them" (SQL Injection: Vulnerabilities & How to Prevent SQL Injection Attacks). This is a major security concern that many hackers can expose and would need to be mitigated immediately.

```
$conn = getDB();
$sql = "SELECT id, firstname, lastname, salary, birth, ssn,
phonenumber, address, email, nickname, Password
FROM data
WHERE id= '0 OR 1=1'";
$result = $conn->query($sql))
```

Figure 5, example of SQL Injection in PHP code

4. Fixing the Code to Prevent SQL Injections

As previously discussed, adding password complexity and encryption to the database, as well as the addition of roles and privileges to users helps prevent attacks. Also previously shown how a SQL injection could be executed on the supplied code. Ways to prevent SQL Injections is by using "prepared statements and parameterized queries. These are SQL statements that are sent to and parsed by the database server separately from any parameters. This way it is impossible for an attacker to inject malicious SQL" (SQL Injection Prevention Cheat Sheet). The given code only passes the parameter straight to the database instead of using parameterized queries. This practice is one of the main weaknesses that allow for SQL injections. Figure 6 shows an example of the id parameter passing through a prepared statement before being passed into the database. Using this example, "if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1" (SQL Injection Prevention Cheat Sheet). This prevents almost any SQL injection attempt from entering the database and is simple to learn and understand for developers.

```
$stmt = $pdo->prepare("SELECT id, firstname, lastname, salary, birth, ssn,
phonenumber, address, email, nickname, Password
FROM data WHERE id = :input_id and password = :input_password");
$stmt->execute(array('input_id' => $id));
foreach ($stmt as $row)
{
    //verify the statement for each row
}
```

Figure 6, example of PHP code that prevent SQL Injections

References

Database Hardening Best Practices. (n.d.). Retrieved November 29, 2020, from https://security.berkeley.edu/education-awareness/best-practices-how-tos/system-application-security/database-hardening-best

SQL Injection: Vulnerabilities & How to Prevent SQL Injection Attacks. (n.d.). Retrieved December 01, 2020, from https://www.veracode.com/security/sql-injection

SQL Injection Prevention Cheat Sheet. (n.d.). Retrieved November 30, 2020, from https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html