ENPM809W – Introduction Secure Software Engineering

Fix Lab 4: Secure Cryptography

Date Due: Tuesday, October 12, 2021 by 11:59 PM

Introduction

In this lab assignment, you will fix all the issues you uncovered in the last attack lab. This assumes you have completed both Lab 0: Lab Setup Guide and Attack Lab 3: Breaking Encryption. For this lab, you will need the ability to run the WebGoat.NET application, as well as debug it using Visual Studio 2019 Community Edition. You will also need to have completed the Git setup outlined in Lab 0 and the ability to push code changes to the remote repository on https://code.umd.edu.

For each of the following questions on the WebGoat.NET application, you will need to refer to your answer from Attack Lab 1, and fix the weaknesses found in the source code. Along with this, you will submit a writeup containing the following:

Phase Number - Question Number.

a) Provide the URL of the WebGoat.NET application page where you are exercising the question. **Do not** include any query parameters or any other special characters in the answer.
b) For the given CWE-ID, Filename, Line Number of the weakness identified in the previous Attack Lab, describe how you intend to fix the weakness. This can be as detailed as possible to explain how it will address the weakness.
c) Describe why your intended fix will address the weakness (either directly or indirectly) and whether to your knowledge it will prevent future attacks along the lines of the attack vector used in the previous lab. This can be as detailed as possible to explain why it will address the weakness.
d) List all the paths with filenames you are changing to implement the fix for the weakness.
e) **IMPORTANT:** Implement the fix for the question under a branch with the following naming convention:
    a. Lab<Lab #>_Phase<phase #>. For example, you can run:
        i. `git checkout main`
        ii. `git checkout -b Lab2_Phase1 # This is for Lab 2 Phase 1 fix`
    b. Now you implement your fix and commit the changes locally:
        i. `git add .`
        ii. `git commit -m “Implemented Lab 2 Phase 1.” # Commit with a message`
    c. Repeat earlier step as many times to get your fix working. Commit and push the newly created branch to the remote repository using:
        i. `git push origin Lab2_Phase1`
    d. **Identify** and **submit the commit id** (the long hexadecimal alphanumeric string from `git log --pretty=oneline`) as part of line item ‘e’ for the given question number in your writeup.

   e. Read https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History for more information.

Please NOTE: For the questions below, the word 'discuss' below refers to question items b) and c) from the list at the top of this lab handout and the word 'implement' below refers to question item e).

Phase 1: Fixing Password Storage (20 points)

As you found out in the previous lab, the passwords were being stored insecurely. You are going to fix this issue in the WebGoat.NET application.

1. The login page for the WebGoat Coins Customer Portal uses and stores the passwords without salts or using strong cryptographic hash functions. In this phase we will fix the first issue, namely storing passwords securely.
2. Now discuss and implement the fix ONLY for storing the passwords. For this do the following and answer using the format given at the top of this lab handout:
    a. First implement a class that securely generates a salt value and uses a common hashing algorithm on the password like SHA-256. See https://dotnetcodr.com/2017/10/26/how-to-hash-passwords-with-a-salt-in-net-2/ for more details.
    b. Change the MySqlDbProvider class's IsValidCustomerLogin() and UpdateCustomerPassword() methods to now use the new class.
    c. This may involve adding a DB migration script. Please include that script in your repository.
3. Remember to commit and push your code under the Lab4_Phase1 branch.

Phase 2: Correct and Strong Hashing (20 points)

As you found out in the previous lab, the identity mechanism in WebGoat.NET application is still vulnerable to dictionary-based attacks. You are going to attempt to fix this issue in the WebGoat.NET application.

4. The same login page for the WebGoat Coins Customer Portal that you fixed above could have used the ASP .NET Core identity mechanism version 3 that uses the PBKDF2 SHA-256 with a 128-bit salt, 256-bit subkey, and 10,000 iterations, which was easily cracked in the earlier lab. In this phase you are going to fix the next issue, namely using a stronger hashing algorithm.
5. You decide you want to change the cryptographic mechanism to something a bit stronger. Feel free to use 3rd party libraries to implement a stronger custom cryptographic hash algorithm or even just tweak the existing one by changing the number of iterations used. You may use a secure Hash function like Argon2. See https://www.twelve21.io/how-to-use-argon2-for-password-hashing-in-csharp/ for more details.
6. Now discuss and implement the fix for stronger cryptographic hash implementation and answer using the format given at the top of this lab handout.
7. Remember to commit and push your code under the Lab4_Phase2 branch.

Phase 3: Secure RNGs (10 points)

Looking at Sorin's custom .NET implementation of the SRP protocol, did you find a weakness in how he was implementing the protocol in terms of secure random number generation?

8. If you did, now discuss a potential fix for the issue you found and answer using the format given at the top of this lab handout. For item e) instead of submitting code, just writeup the changed line of code along with any fixed library "using" statements.
9. If you did not, please skip this question.

Phase 4: Using CPSA to identify cryptographic protocol vulnerabilities (20 points)

In the previous phase, if you read the NDSS paper proposing SRP, you will see the proof for security was provided by reducing the protocol to the Diffie-Hellman protocol. In this phase, we will show a man-in-the-middle attack on Diffie-Hellman Key exchange protocol since neither of the two parties are authenticated as part of the protocol before the key exchange takes place (it just assumes that is the case). The CPSA v3.6.7 is already installed on your VM and can be used in conjunction with the Diffie-Hellman protocol specification. The manual has Chapter 4.2 dedicated to modeling Diffie-Hellman (https://hackage.haskell.org/package/cpsa-3.6.7/src/doc/cpsamanual.pdf). Answer the questions given above with regards to the input and output from CPSA. **For item a), just put N/A.**

Submission Criteria

Please submit a writeup with:

1. Your name, UMD email ID and Lab Number.
2. The answers provided in the format given at the top of this lab handout.

Please only submit a **Word document** or a **PDF**. This lab contains code submission with branches for each phase. Please ensure the commits submitted are accessible by the auto grader.