Software and Database Security

Software Engineering - EPITA

Team Members

Marwan Mustapha Mai

Omar Issa

Bishal Udash

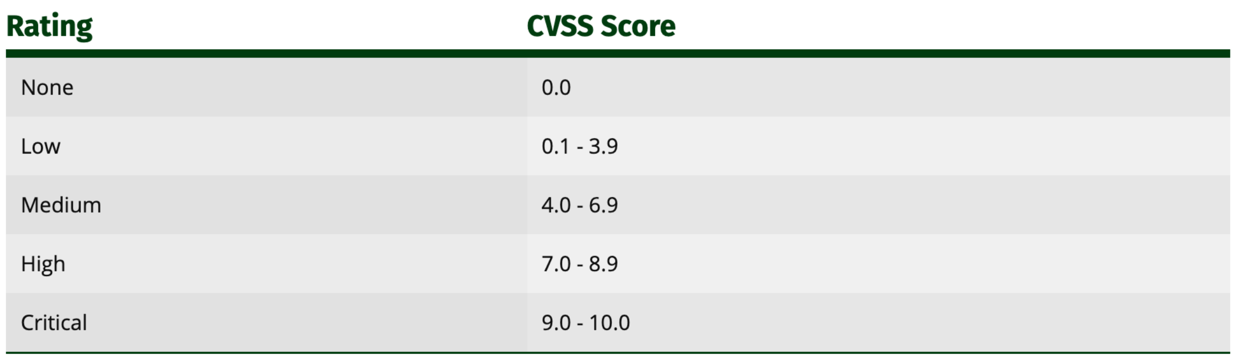
Date: 31/03/2024

Synthesis

The aim of this audit is to discover all possible vulnerabilities in a locally run desktop Bank application. The audit was done between 20/03/2024 to 31/03/2024.

The vulnerabilities are categorised based on the base score of Common Vulnerability Scoring System (CVSS)

This is a representation of the vulnerability categories and their scores



Rating	CVSS Score
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

Source link: https://www.first.org/cvss/v3.1/specification-document

The critical vulnerabilities are to be taken as the highest priority as their effects can bring all operations to a halt, this is followed by high-priority vulnerabilities up to lower vulnerabilities which do not have as serious effects or consequences and can come as recommendations to the organization.

A few remediations to such vulnerabilities include but are not limited to the following, going from critical to low threat vulnerabilities :

- Moving away from two-tire architecture to a more robust three-tier architecture which does not allow direct communication to the database
- Enforcing strong passwording procedures and credential management by the organization by using two factor authentication and alphanumeric long strings for passwords.
- Perform vulnerability tests on products or services before they are launched in-house, the attack.mitre.org resource can be used as a guide.

Vulnerabilities in Bank App

Use of Password Hash With Insufficient Computational Effort

Common Weakness Enumeration (CWE) ID : CWE-916: Use of Password Hash With Insufficient Computational Effort

CVSS 3.1 Base Score Metrics :

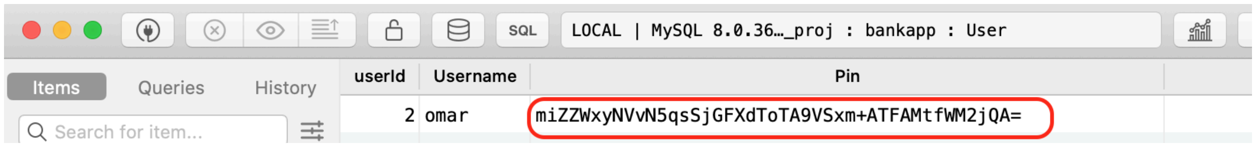
- AV:L/AC:L/PR:L/UI:R/S:U/C:H/I:H/A:L
- 6.8
- Risk level : Medium

Description:

- The application contains the feature to hash pins/passwords using SHA-256 algorithm
- This hash is then stored to the database directly as is

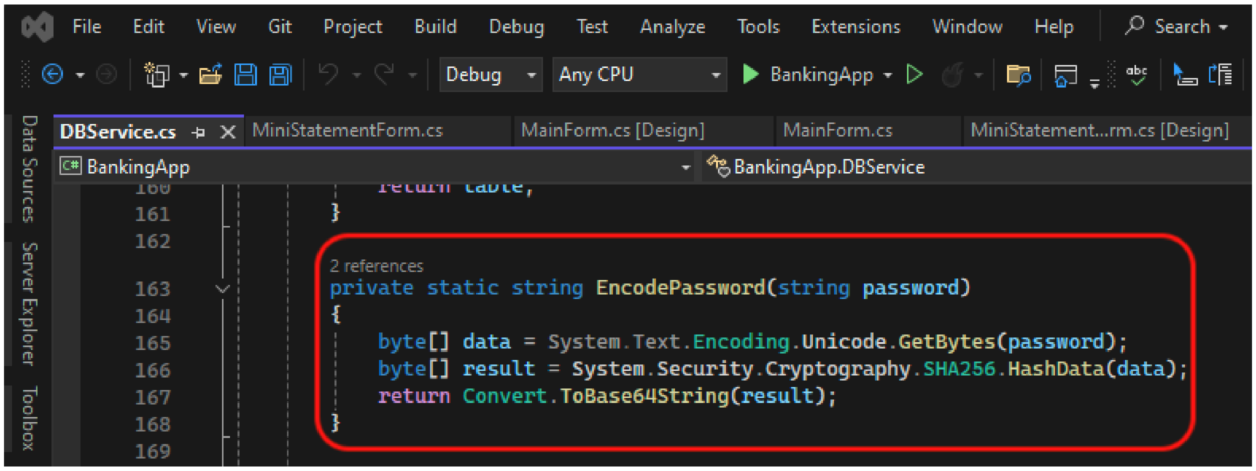
Exploitation:

- Since the application is using a well known hashing algorithm in the name of SHA-256, there a lots of tools online than be be leveraged to make similar hashes
- A tool such as Cyber Chef to generate a hash of similar passwords or pins
- Or an easier way would be for hackers to use a lookup table of compromised hashed passwords which are available online and on the darkweb called Rainbow Tables
- An attacker can run a script on the application to try all possible hashes and get access to the system by matching the hash
- This is the hashed password is show in the image below



userId	Username	Pin
2	omar	miZZWxyNVvN5qsSjGFXdToTA9VSxm+ATFAMtfWM2jQA=

- This shows the source code where the pin/password is being hashed with SHA256 and not having any salt to it


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search
Debug Any CPU BankingApp
DBService.cs MiniStatementForm.cs MainForm.cs [Design] MainForm.cs MiniStatement...rm.cs [Design]
BankingApp BankingApp.DBService
Data Sources Server Explorer Toolbox

```
        return table;
    }

    2 references
    private static string EncodePassword(string password)
    {
        byte[] data = System.Text.Encoding.Unicode.GetBytes(password);
        byte[] result = System.Security.Cryptography.SHA256.HashData(data);
        return Convert.ToBase64String(result);
    }
```

This is the source code and function responsible for hashing the pin/password

Remediation:

- The passwords must be "salted" to add more complexity to the hashing to achieve a more unique hash and more difficult to match
- Use unique salts for application having a large number of users to prevent the risks of attachers getting the slat
- Store passwords and other sensitive credentials in external configuration files or environment variables rather than embedding them directly in the source code. This allows for easier management and updating of passwords without modifying the application code.
- Employ secure password storage mechanisms such as cryptographic hashing algorithms such as Bcrypt to make the storage of passwords and data more secure

Use of Password Hash Instead of Password for Authentication

Common Weakness Enumeration (CWE) ID : CWE-836: Use of Password Hash Instead of Password for Authentication

CVSS 3.1 Base Score Metrics :

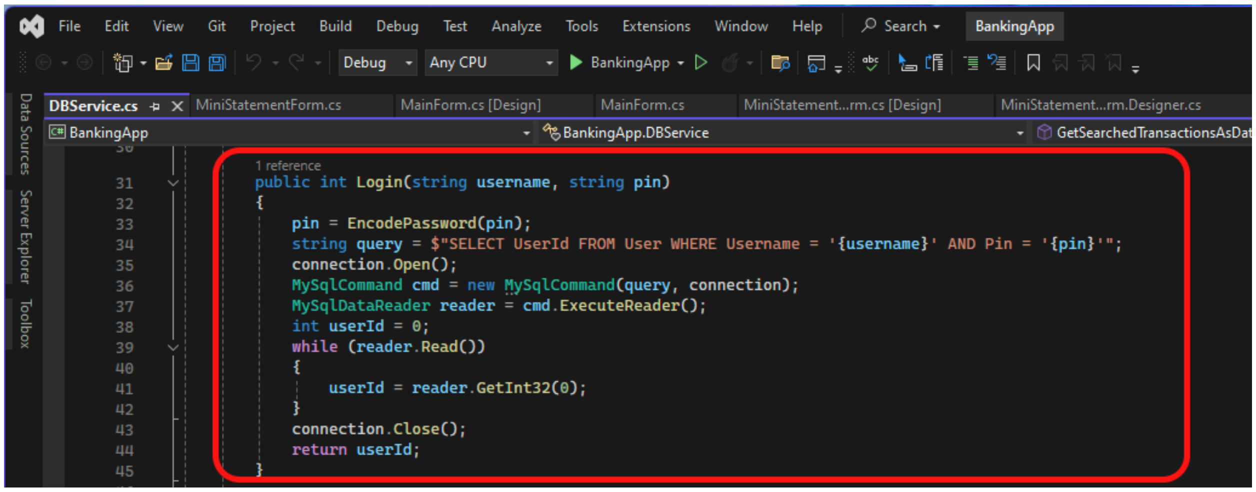
- AV:L/AC:H/PR:L/UI:N/S:U/C:H/I:H/A:H
- 7.0
- Risk level : High

Description:

- The application hashes the password at the client end before it is stored into the database as is

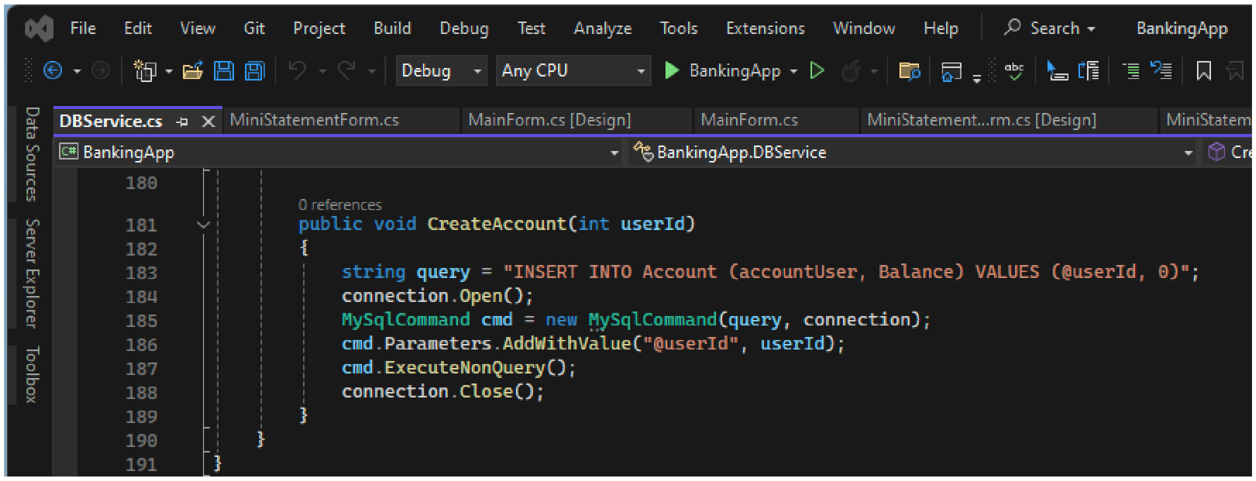
Exploitation:

- The password hash can be side stepped by an attacker by obtaining the hash when it's being generated at login or at registration
- A hacker can use SQL Attack techniques and can use another compromised account to use the hash obtained without knowing the original entered password text



```
public int Login(string username, string pin)
{
    pin = EncodePassword(pin);
    string query = $"SELECT UserId FROM User WHERE Username = '{username}' AND Pin = '{pin}'";
    connection.Open();
    MySqlCommand cmd = new MySqlCommand(query, connection);
    MySqlDataReader reader = cmd.ExecuteReader();
    int userId = 0;
    while (reader.Read())
    {
        userId = reader.GetInt32(0);
    }
    connection.Close();
    return userId;
}
```

- The code is being generated when the user is trying to create an account


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search BankingApp
Debug Any CPU BankingApp
DBService.cs MiniStatementForm.cs MainForm.cs [Design] MainForm.cs MiniStatement...rm.cs [Design] MiniStatem
BankingApp BankingApp.DBService
Data Sources Server Explorer Toolbox

```
        0 references
        public void CreateAccount(int userId)
        {
            string query = "INSERT INTO Account (accountUser, Balance) VALUES (@userId, 0)";
            connection.Open();
            MySqlCommand cmd = new MySqlCommand(query, connection);
            cmd.Parameters.AddWithValue("@userId", userId);
            cmd.ExecuteNonQuery();
            connection.Close();
        }
    }
}
```

This is the source code from the Bank App

Remediation :

- Secure Transmission: Ensure that the password is transmitted securely, using encryption (such as HTTPS) to prevent interception by attackers.
- Use Multi-Factor Authentication (MFA): Implement multi-factor authentication to provide an additional layer of security beyond just the password. This can involve something the user knows (a password), something the user has (a mobile device, biometric verification).

Tools Used

- Wireshark : Wireshark is a free and open-source packet analyzer.
- Mac Os native screenshot application
- Greenshot : Open Source screenshot app with photo editor
- 7zip : An opensource tool that is used for archiving/zipping and unarchiving which has a useful string too that can read strings in the executables of applications
- Echo Mirage : A network proxy that uses data definition language injection for intecerception to a database and using hooking techniques
- Cyber Chef : Online open source tool for hashing and using encryption and encoding tools
- HXD : HxD is a carefully designed and fast hex editor which, additionally to raw disk editing and modifying of main memory (RAM), handles files of any size.
- HeidiSQL : HeidiSQL is a free and powerful client for MariaDB, MySQL, Microsoft SQL Server, PostgreSQL and SQLite.

Reference

- https://attack.mitre.org/
- https://www.first.org/cvss/v3.1/specification-document
- https://nvd.nist.gov/vuln-metrics/cvss
- https://en.wikipedia.org/wiki/Bcrypt
- https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator?vector=AV:L/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N&version=3.1on Vulnerability Scoring System Calculator
- https://en.wikipedia.org/wiki/Rainbow_table
- https://en.wikipedia.org/wiki/Salt_(cryptography)