CptS 428/528 Software Security and Reverse Engineering
Fall 2023
Team: Undergraduate ABFMJ-SecureDove
Project Deliverable 4

Team Members: Muhammad Ali Dzulfiqar, Flavio Alvarez Penate, Phearak Both Bunna, Jaysen Anderson, Mitchell Kolb

Objectives:

- Update your software design according to the enhanced/improved security requirements laid out during Milestone 3.
- Implement the new design that is expected to lead (the newer version of) your software product to enhance security against all the attacks/exploits you conducted during Milestone 3.
- Validate the more secure implementation against those attacks and exploits (i.e., the regression security testing the software product using the same attacks and exploits but now against the new implementation).
- Summarize the new design, securer implementation, and the validation results together as a report for this milestone.

Database API Key Being Visible

- **Problem Statement From Milestone 3:**
    - For SecureDove we use a third party postgresql hosting service called ElephantSQL. To use this service in our FastAPI backend we have to connect to the service using an API key. This key is very confidential and allows all users who have it, direct admin access to view, add, edit, and delete any and all data in our database using SQL code. Therefore it is important to make this API key string private when possible.
- **New Design:**
    - The solution we implemented to solve this problem contains four steps:
        1. Separate the api key into its own file called ".env" which contains a variable called "API_KEY" that has the value of the key to access the database.
        2. As per standard security protocol we also have a new file called "config.py". This file uses the load_dotenv python library to load the API_KEY variable from the .env file.
        3. We finally load the API_KEY value from config into main and use these lines of code to connect to the database.

           ```
           from config import API_KEY

           conn = psycopg2.connect(API_KEY)
           ```

        4. In a real life scenario for production we would then add the ".env" file to the .gitignore so the api key value stays secure only on the developers machines and we would send out the file to others as they need it if the team were to expand. This allows our project to stay on github while giving us the security that not anyone who views our codebase has the ability to access our database. NOTE: for this milestone we have included the .env file into the github to show the complete process of this solution.
- **Securer Implementation:**
    - This solution is more secure than the way we had it before because it increases reusability, modularity, and allows environment-specific configurations. For increasing security this solution of storing sensitive information like API keys in .env files allows us to minimize software vulnerabilities. If our code is exposed or leaked, before this fix was applied the API keys would've been exposed as well. By using environment variables and a configuration file, we can keep our API keys secure. For modularity and reusability, the config.py file serves as a central place to manage all of our application's configuration, including the API keys. This makes our code more modular and reusable. If we need to change the API key, we only need to do it in one place, the config.py file.

- **Validation Results:**
  - The solution to this problem is very easy to validate. We only have to test two things. Is the API_KEY actually abstracted so if a potential attacker has only our codebase but no .env file will they be able to access our database and does this solution still retain the ability for our existing code to function as expected. The answer to the first question is no, this potential attacker will not be able to access our database if they don't have the .env file even if they have our codebase. For the second validation the answer is yes, our app still functions as expected.

SQL Injections

- **Problem Statement From Milestone 3:**
  - These SQL injection methods don't completely crack the logging in. As you can see we still get an error that says that the credentials don't match. However, these methods do get us past the email checking stage which is still a vulnerability.
- **New Design:**
  - Luckily, SQL injections are a very common vulnerability found in programs that use SQL queries so our solution is straightforward. Our solution was to parameterize all of our SQL queries throughout the code base. This involves moving all of the input from the users into a tuple that is sent in as the second argument in our execute commands. Then, in the location where we need users info in our SQL queries, we put a '%s'. This works very similar to functions in the C language.
- **Securer Implementation:**
  - This solution is far more secure because we are now doing proper validation and sanitation. This will ensure that something like DROP TABLE Users; -- would be caught and handled and would not destroy our database. This also benefits in readability and maintainability. As this is the new standard for the codebase, any future implementation will look very similar.
- **Validation Results:**
  - Validation was broken down into two parts. Firstly, we needed to ensure that the changes didn't impact functionality. This involved using normal behavior via our endpoints to ensure that we hadn't broken anything with our changes. Secondly, we ran the same queries that we ran in the previous milestone that had gotten past certain safety measures that we had in place before. All of the endpoints worked as expected and none of them were vulnerable to SQL injections.

Brute Forceable Passwords

- **Problem Statement From Milestone 3:**
    - In the previous milestone, our team discovered how easy it is to brute force passwords that are short in length and don't cover a large array of characters. For example, it takes just mere seconds to go through every possible four character password containing just plain letters a-z. Therefore, our team needed to come up with a solution to prevent attackers from easily deciphering a user's password by adding password requirements during the registration step of our application.
- **New Design:**
    - The new password requirements design is as follows:
        1. When a user is registering and goes to enter a password, they will get an alert informing them of all of the password requirements they must satisfy.
        2. If a user enters a password that is less than 8 characters in length, they are warned and instructed to make their password more secure by making it longer.
        3. If a user satisfies the length requirement, their entered password is then checked to contain any character from the JS regular expression /[A-Z]/ using its test() method. This check is done to ensure that the user has at least one upper case letter in their password, thus doubling the number of possibilities an attacker will have to account for from just 26 (a-z) English letters to 52 (a-z and A-Z).
        4. If both of the previous checks succeed, then the password is checked to contain any number from the JS regular expression /[0-9]/. If this returns false, then the user is prompted to add a number to their password, otherwise, it proceeds to the next check.
        5. Finally, if all of the above succeed, one final check is done to ensure that the user entered password contains a special character from the JS regular expression /[!@#$%^&*]/. If this check fails, the user is prompted to add one of these characters to their password; otherwise, they can successfully sign up.
- **Securer Implementation:**
    - The solution for the new design demonstrated above is significantly more secure than our previous solution. Initially, we set many test users to have a simple 4 character password "test." We soon realized during the last milestone that while this was quite convenient for testing with various users, it was clearly a security flaw and needed to be addressed. By increasing the minimum character requirement and the possible character set for each password from 26 (a-z) to 70 (26 lower case letters, 26 upper case letters, 10 numbers, and 8 symbols), the number of possible password combinations has increased by a tremendous amount.
- **Validation Results:**

    - The validation for this change in design had three main objectives: improve security, maintain functionality, and limit user frustration. As previously discussed above, we have successfully achieved the first objective by improving the security of our application from brute force attacks during user registration. Then, we had to ensure that these new checks during the registration process do not prevent the user from signing up when all of the password requirements are met and provide the same functionality as the application did before. We needed to ensure that the user wouldn't be locked out from registering despite satisfying all of the password requirements and conditions. Finally, we kept user frustration in mind when creating the password requirements. While we could force users to create 32 character passwords that contain all sorts of symbols and special characters in order to increase security, this shouldn't be put to practice. It would be a very frustrating experience for a user to remember and type out such a long password every time they wish to log in. Therefore, the password requirements we came up with significantly increased security but also focused on user experience and ease of use.

Passwords are transmitted & stored in plain text

- **Problem Statement From Milestone 3:**
    - From milestone 3, we can see that the users' credentials, especially passwords are stored as plain text (no hashing or salting) in our ElephantSQL database. Not only that, we were able to use Wireshark to capture the network traffic and retrieve the emails along with the passwords that the users typed in. This is not a secure or practical way of storing user's credentials. Therefore we need to modify our application to make it more secure.
- **New Design:**
    - Our solution is to apply the SHA-256 hash algorithm to the passwords that the users typed in before storing in the database or before transmitting them over the network. We can utilize the ***crypto-js*** JavaScript library which provides useful and secure cryptographic functions for us to use when developing our program or application. Moreover, our team has decided to hash the passwords during both the *Registration* process and the *Login* process.
- **Securer Implementation:**
    - The solution we came up above is far more secure than the one we had before since it will ensure that all the passwords are stored as hashes and the credentials are also being transmitted over the network as hashes which cannot be reversed or converted back to the original plain text passwords. This is a very effective and secure way to store and transmit credentials over the network.
- **Validation Results:**

- ◦ The validation for this solution can be broken down into 3 parts. The first important part is to ensure that the application is still functioning as expected, especially making sure that the users can create accounts and can login to their account just fine. The second part is to check in our database whether the passwords for the new accounts are being stored as plain texts or as hashes. The last part is to capture the network traffic using WireShark to see if the passwords are being transmitted as plain texts or as hashes.
- ◦ As expected, the applications are still functioning like before and the passwords are being stored and transmitted as hashes.

Known Security Vulnerabilities in dependencies

- **Problem Statement From Milestone 3:**
  - From the previous milestone, our team found 9 security vulnerabilities associated with our npm (node package manager) packages including the needed dependencies and libraries for our application. Some of the versions of the packages are deprecated and not maintained anymore.
- **New Design:**
  - The solution that we found for this problem is simple and we can fix the security vulnerabilities in the npm packages by running this command in the terminal "***npm audit fix --force***".
- **Securer Implementation:**
  - The solution above is an automated approach to address security vulnerabilities in our project's dependencies and it will ensure that everyone is using the same versions of dependencies and ensure that all the packages are up-to-date. It is very crucial to keep dependencies up-to-date since it can help protect our application from known vulnerabilities that are being addressed in the newer versions.
- **Validation Results:**
  - The validation for this solution is simple. First we only have to make sure that the application is still working as expected. Then we can recheck for the number of known vulnerabilities in our dependencies by running this command in the terminal "***npm audit***". It should say 0 if there are no other known vulnerabilities in our dependencies.

URL Open Redirection Vulnerability

- **Problem Statement From Milestone 3:**
    - From the previous milestone, our team found that it is possible to bypass the login by modifying the URL from localhost:3000/login to localhost:3000/messages.
- **New Design:**
    - The solution that we found for this problem is to remove the URL parameter to prevent direct redirection post-login.
- **Securer Implementation:**
    - The solution above is to remove the URL for the landing page for logged and unlogged users. That way, when an unlogged user is trying to get into the app by the URL for the logged landing page, the page would just display the landing page for the user that hasn’t been logged in instead of the logged view through the URL.
- **Validation Results:**
    - The validation for this solution is simple. We only have to make sure that the logged view ‘http://localhost:3000/’ will display the view for logged user and will redirect to the unlogged view (redirects to login page) if the user is unlogged.