CMP-6045B Developing Secure Software – Report 1: System Development and Testing

Group UG-4:
100263580 – Steven Diep
100251448 – Sam Humphreys
100225776 – Martin Siddons
6214363 – Chris Sutton

1 Introduction

Statistics report found that roughly 50% of production web applications contained 'high-risk' vulnerabilities (Positive Technologies, 2020). They go on to say that 82% of these vulnerabilities were located in the application code, suggesting that developers favour functionality over security and that production code is not being tested sufficiently to prevent these flaws. With an average cost of a data breach to an organisation of $3.86 million. (IBM, 2020) in 2020, the likelihood and impact of vulnerabilities in web applications are both high. This report will show an analysis of measures that were identified in our previous individual research that have been selected to be implemented in our system for mitigating some of the main vulnerabilities of a web application, as well as evidence of extensive testing measures used to attempt to ensure intended behaviour.

2 Individual Report Reuse

2.1 Report Commonalities

After an arranged meeting regarding the planning of the blog website, we gathered similarities between individual reports for each vulnerability. We found that returning a generic response with a consistent processing time and administer a lockout if the user attempted over five times would be suitable to manage account enumeration. Sessions should be temporary and should be used with 'HTTPOnly' cookies to prevent hijacking, access should also be denied from outdated browsers and ensuring authentication should always use SSL or TLS. Using prepared/parametrized queries and adopting a privilege principle among users by limiting access can mitigate cases of SQL injection attacks. Cross-site scripting can be controlled by validation and sanitization of data, certain characters should also be escaped before entering the server, encoding for pages should be in UTF-8 not ASCII, and whitelisting of HTML markup. Generation of cross-site request forgery tokens should be random and unique to prevent attackers from guessing the correct token, using a complex algorithm such as SHA-256 to generate a token should be appropriate to achieve randomness and uniqueness, furthermore, a user is forced to re-authenticate when sensitive actions are taken such as deleting a profile.

2.2 Additions from Steven (100263580)

Use secure protocols like HTTPS not HTTP because attackers can sniff the network to view cookies which the attacker can manipulate. Selecting a strongly typed language does not require memory management which mitigates the risk of a stack buffer overflow attack. Research was conducted on the usability of two factor authentication, and they found that users spend less time logging in compared to other methods of authentication (Reese et al., 2019).

2.3 Additions from Sam (100251448)

Introducing Captcha to verify login or IP blocking can tackle with unsophisticated enumeration techniques. Be wary of using libraries that have known vulnerabilities as this could be a major security hole. Two factor authentication is 99.9% less likely to be compromised compared to other authentication techniques (Weinert, 2019).

2.4 Additions from Martin (100225776)

No hints in URL, page title or packets for attackers to take advantage of in the case of account enumeration. Type must be assigned correctly for the database to correctly identify input (True in the Surname field should be recognized as a name not a Boolean value), the use of drop-down lists or spinners to select some inputs to restrict what can be entered to prevent SQL injection attacks and be mindful of the amount of space to give the user for input. Generating CSRF tokens should be per-session, since per-request generates issues using the back button and strict 'SameSite' should not be used on a blog as it forces re-authentication, which is overkill.

2.5 Additions from Chris (6214363)

Enforcing a minimum password length and encourage users to make a unique password when registering. Periodic checks should be made if the user's IP has changed for hijacking. Hashing sensitive information in case of a successful database attack, threat actors will not be able to unhash the stored data without knowing what hashing algorithm used.

3 User Testing

The purpose of a think-aloud experiment is to provide a user with some short tasks to perform within the application and have them narrate their experience. It is useful for testing critical parts of the interface, as well as to get a user's subjective opinion of the look and feel of the design (Jaspers et al., 2004). In this example, three users were observed while performing three tasks, and their narration was transcribed. The tasks were chosen for their frequency of use, and to highlight difficult or confusing parts of the site:

- Create a new account
- Log into your account
- Make a post
- Change your username
- Reset your password

Overall, the site was found to be functional by participants, although the styling was limited, and the interface was confusing in parts. Several common themes of complaint were apparent:

- The site was considered to be sparse and visually unappealing by all participants
- The users found there was no easy way to navigate to the settings page
- Some of the error messages were not informative
- The password reset email was usually blocked by spam filters

Given that the objective of development was security rather than visual appeal, the changes that were made focussed on those elements that were critical to usability. A link was added to allow a user to click on their username and view their settings, and the error messages were reworded to be more precise. The email function was rewritten to print to console for the sake of testing and linked to a Gmail account for the live version.

Thankfully, most of the security improvements did not impact usability in any noticeable way. Certain compromises were made, such as delaying server response by a fraction of a second to avoid account enumeration but this was not noticed by any of the users. In fact, some of the security features, such as two factor authentication, are now so commonplace that users did not comment on their inclusion. They were 'to be expected'.

4 System Testing

Throughout development of the system, we performed module testing to ensure the function we were adding to the project was working as expected. This involved black box testing where we wrote down what edge cases could come up as inputs as well as core cases through equivalence partitioning, then built the tests into specified files named after each Python file being tested, within the tests folder with data chosen to fit the equivalence partitions chosen. Implementing tests in specified files and folders allowed us to use the flexible unittest library within Python to easily implement and run tests in a controlled environment.

Integration testing ran alongside module testing, where tested code we wrote in blog.py to utilise the functions previously written and tested through module testing. Testing here involved treating each page as a black box and simulating the client logging in and submitting requests and forms to the flask server, checking that the data coming back was as expected. We feel that having the module testing previously complete made integration testing much easier as we could tell when a change broke a function without worrying if it was an issue with integration.

System testing was performed as integration was complete. For this we used path testing to understand how much code coverage we had in the tests already performed. For this we used the tools available within the PyCharm development environment to run our unit tests folder with Code Coverage enabled. This produced a report showing which statements and branches within each script were covered by code, and highlighted branches within PyCharm that were not covered. We then manually added additional tests to ensure that the missing branches were covered, where it was practical to do so. From research, we found that 100% code coverage was impractical, but that 70-80% was a reasonable goal (Cornett, 2006). However, as we uncovered issues through these additional tests, such as error messages not being shown to the user, we slowly increased the number of branches covered. This resulted in 96% of blog.py being covered by tests, as can be seen in the results in appendix A, figure 1.

Acceptance testing was accomplished by each member of the group before submission by extensively using the attack techniques previously taught to us, against our own site to discover weaknesses. Due to comprehensive testing during the project, fortunately no issues were found at this stage.

5 Conclusion

Due to the analysis of our individual research findings, we were able to quickly identify suitable techniques that fit with our use-case and limitations. Thorough testing of the implemented mitigation measures, and their gradual improvement based on feedback, has resulted in what we believe to be an overall effective system for dealing with the main web vulnerabilities.

References

Cornett, S. (2006). Minimum acceptable code coverage. **URL:** `https://www.bullseye.com/minimum.html`.

IBM (2020). How much would a data breach cost your business? **URL:** `https://www.ibm.com/security/data-breach`.

Jaspers, M. W., Steen, T., Van Den Bos, C., and Geenen, M. (2004). The think aloud method: a guide to user interface design. *International journal of medical informatics*, 73(11-12):781–795.

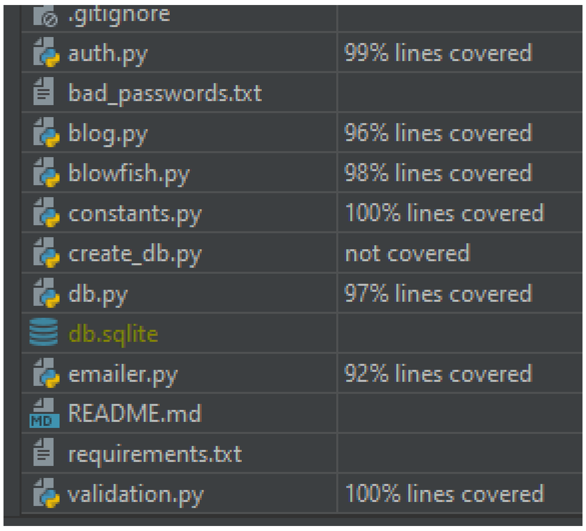
Positive Technologies (2020). Positive technologies: 82 percent of web application vulnerabilities are in the source code. **URL:** `https://www.ptsecurity.com/ww-en/about/news/82-percent-of-web-application-vulnerabilities-are-in-the-source-code/`.

Reese, K., Smith, T., Dutson, J., Armknecht, J., Cameron, J., and Seamons, K. (2019). A usability study of five two-factor authentication methods. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, Santa Clara, CA. USENIX Association.

Weinert, A. (2019). Your pa$$word doesn't matter. **URL:** `https://techcommunity.microsoft.com/t5/azure-active-directory-identity/your-pa-word-doesn-t-matter/ba-p/731984`.

A Appendix A

A.1 Figures



.gitignore	
auth.py	99% lines covered
bad_passwords.txt	
blog.py	96% lines covered
blowfish.py	98% lines covered
constants.py	100% lines covered
create_db.py	not covered
db.py	97% lines covered
db.sqlite	
emailer.py	92% lines covered
README.md	
requirements.txt	
validation.py	100% lines covered

Figure 1: A list detailing the percentage of code in each script covered by unit tests. Generated by PyCharm using Python's built in 'coverage' library.