# Black-box Testing  - Planning and Analysis

*Group 3*

## Introduction

In this exercise, our goal is to perform a blackbox test on a simple movie review web application. We will start with the OWASP Top 10 to get an overview of the most common vulnerabilities. Given the short timespan of this assignment, we will look through the OWASP testing guide and select some of the more relevant sections and do the suggested testing on the web app located at http://tdt4237.idi.ntnu.no:5003/.

## Focal points

In our test, we will focus on the application and the application's framework itself, not OS-level discrepancies, server misconfigurations or server vulnerabilities. We will base our testing on the OWASP Top 10 list, and extend it to include a few of our own categories, including but not limited to:

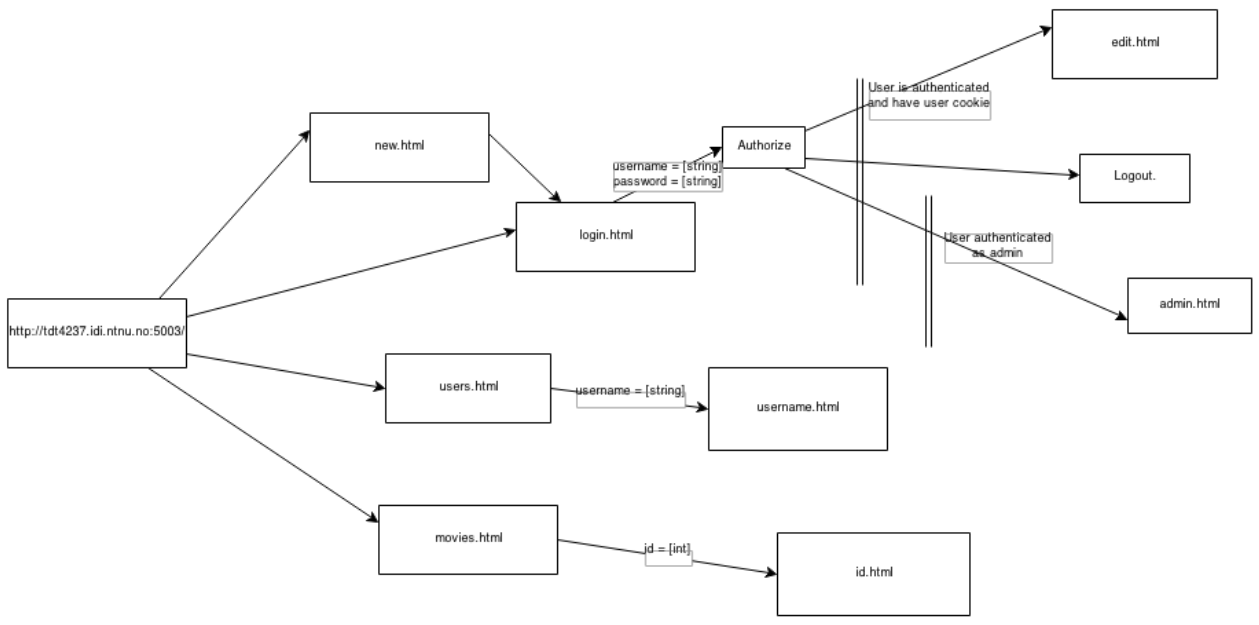
- Privilege escalation (OWASP-AZ-003)
- File Extension Handling (OWASP-CM-005)
- Infrastructure and Application Admin Interfaces (OWASP-CM-007)
- Brute force (OWASP-AT-004)
- Bypassing Authentication Schema (OWASP-AT-005)
- Logout and Browser Cache Management  (OWASP-AT-007)
- Password policy

Our priorities will be mainly directed towards authentication, access control, protection of sensitive data and privilege escalation. We chose to extend our search throughout the OWASP testing guide, version 3. The reason we chose to use version 3 is that version 4 is currently being reviewed and is not considered finished. Version 3 was therefore more suitable for our use.

## Analysis

The first part of our analysis will be to obtain an overview of the application's structure, what parts of the application are available in an unauthenticated state, and which parts require authentication.

In order to do this we will use a simple link-scanning tool and review the results to create a map of the site.



With our focal points in mind, and with the help of our application structure analysis, we will then move forward to begin the testing process. Our initial exploration will be conducted based on the OWASP Top Ten list, to cover the most prolific attack vectors. Secondly, we will look for vulnerabilities related to our own vulnerability categories, many of which are gathered from the OWASP Testing Guide v3, stated previously. Subsequently, we will analyze our results and look for patterns in the vulnerabilities (if any) that might lead to a combined vulnerability of some sort.

## Work breakdown

We first started to work on the OWASP top 10 vulnerabilities. We all sat down and worked together on each item in the list, testing them simultaneously on different pages in the pagemap. After we worked through the top 10 list, we continued working on other known vulnerabilities, gathered from the OWASP Testing Guide.

**Information gathering (OWASP-IG-003, OWASP-IG-006)**
First we tried to understand the applications logic. We used a proxy to look at the HTTP requests and responses. By doing this we discovered that the cookie contains an admin value in plain text, that the HTML source code contain the password hash for a given user and the site has a list of all users. By the end of the preliminary probing of the page we had a sitemap and an understanding of the workings of the application. The testing points we discovered was, the password variable, username, the possibility for injections and scripting.
We were also able to get responses including stack trace from the server. This in combination with injections made it possible to probe the internal workings of the database. Because of this we now know that the database is a SQL database.

**Authentication Testing**
We then tried to map out the different faults in the authentication system. We quickly discovered that the users has no password policy at all, which makes it possible for users to have short and easily guessable password. We tested both guessing (G03_0010) and bruteforcing (G03_0007) the site and both yielded a positive result. In the HTML we found password hashes for every user. These hashes were not salted and easy to check with rainbow tables. We also made a script that bruteforced the hashes, when we found out that they were SHA512.

**Configuration Management Testing**
During our testing we discovered that the path to the admin section was /admin. It was necessary to login to access this section. But after some further testing, we quickly discovered that if we added /delete/<USERNAME>/ (G03_0015) to the admin path, users were deleted without any question. This is really insecure and does not support the principle of least privilege.

**Authorization testing**
When we tried to found authorization issues we could not find any that directly would fall under any OWASP code in this area. But we noticed that a person or bot/crawler/spider could post movie reviews without a user account. This means that SQL injections can be done by anyone that can access the site (G03_0009).

**Session Management Testing**
In our testing for faults in the session management we found two different errors. The first one makes it possible to change a boolean attribute in the cookie to true which gave us admin (G03_0003). The other one we found was the possibility for cross site request forgery, this could be done by adding HTML to a site which loaded scripts that executed login request on another user's behalf (G03_0005). Also we found two errors that makes it possible manipulate a users session id. This two errors (G03_0011 and G03_0012) makes it possible to use the session id of a logged in user to log yourself in. Because there is no timeout on the session id you will be able to use a session id of any logged in user. Also the session ids are sent over an insecure (HTTP) connection.

**Data Validation Testing**
Our data validation testing gave us some worrying results. We found that every input field on the entire site (except the password fields) could be SQL injected (G03_0002). Also the input fields were open for HTML injection (G03-0004). This means that we could without problems load scripts from other resources. We also found that one could inject JavaScript by only a link to the user (G03-0001).

**Web Services and Ajax Testing**
Since the website has no REST functionality or API we could not find any issues that were related to Web Services and Ajax. We tried to use some different HTTP GET parameters against the server, but none of them yielded any useful results.

## Conclusion

After a thorough testing of the web application Movie Reviews, we found that this application is vulnerable for multiple kinds of attacks. It is vulnerable for injections, XSS, CSRF, Data exposure etc. During the execution of the tests almost every weakness we wanted to test for succeeded. From the OWASP Top-10 list there were only three weaknesses we weren't able to find.

Summary of found weaknesses in the OWASP TOP-10 list

| | |
|---|---|
| A1 - Injection | Check |
| A2 - Broken authentication and Session management | Check |
| A3 - Cross-Site Scripting | Check |
| A4 - Insecure Direct Object References | Not found |
| A5 - Security Misconfiguration | Check |
| A6 - Sensitive Data Exposure | Check |
| A7 - Missing Function Level Access Control | Check |
| A8 - Cross-Site Request Forgery | Check |
| A9 - Using Components with known Vul | Not found |
| A10 - Unvalidated Redirects and Forwards | Not found |

# White-box Testing  - Planning and Analysis

*Group 3*

In this exercise, our goal is to perform a whitebox test continuing the penetration test from last week. We will inspect the source code to see if we can find new vulnerabilities.
To help us analyse the source code we used the following tools:

- www.devbug.co.uk - only single file testing.
- RIPS - (No object oriented style support)
- CodeSniffer -
- RATS - Rough Auditing Tool for Security

## Focal points

In our test we will focus on the source code to check for new errors not discovered in the black-box testing. Some of our focal points will be input validation, use of non-secure built-in escape functions, use and configuration of frameworks, and method call stack.

## Analysis

Our plan was to first analyze the code, and create an overview of the file structure, the class hierarchy, and the method call stack. After we had created an overview, we would review the different use cases in the application, following the call trace and trying to identify additional vulnerabilities.

Next, we would perform a manual inspection of all the source files, but due to our lack of PHP understanding, both on how the language itself is structured and how the different builtins work, we had a clear disadvantage. We would divide and conquer the different source code files, while also keeping track of which classes and methods were connected to each other. This to ensure that we could follow the call stack in our analysis.

Mitigating our lack of experience with PHP, we would test the source code with the above stated static analysis tools.

## Work Breakdown

**Files, Classes and Call Stack**
We first got an overview of the file structure, class hierarchy and method call stack. When iterating through the different functions from the web applications, we did not find any additional vulnerabilities in the application, except from one, which we had originally found in the black box test, but failed to specify it as an insecure direct object reference OTG-AUTHZ-004[1].

**Manual code inspection (As a group)**

Due to our lack of experience with PHP, this was a somewhat tedious process, but after some reading up on how PHP works, we were able to analyze the code in a reasonable manner. When analyzing the different methods used in the application, which were accessible in some way through remote interaction with the web application, we failed to locate any further potential threats based on our understanding of the source code. Albeit with the exception of the missing direct object reference vulnerability that we did not include in the previous report's Top 10 checklist. We also found an error in the regex rule for usernames allowing usernames with underscore.

**Static Analysis Tools**

We first ran the codebase through the RISP tool, but quickly figured out that it did not support an object oriented code style. We were informed that this would cause failed negatives and potential non-reported vulnerabilities.

We also ran the source files through phpcs (Code Sniffer), but there were only warnings about missing explicit access modifiers and other general linting-errors, and no output that would indicate that there might be a potential security risk with the current configuration.

Finally, as a last attempt we ran the code through the RATS tool, but we didn't achieve anything new here either. The only output when we sat the security level to high where some return statements in different functions. We didn't manage to find any new weaknesses that would count as a potential security risk.

[1] OWASP Testing Guide v4. IDOR has no code in the v3 of the testing guide, but even though v4 is not in the curriculum, we choose to use the code from v4 for this risk, since v4 now is considered stable.
https://www.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_(OTG-AUTHZ-004)

**Improvements (Our suggestions to the all vulnerabilities we found)**

Most of the vulnerabilities we found during the white-box test where just confirmations on what we found during the black-box test. A suggestion on how to improve the vulnerabilities we've found would be to implement proper sanitizing of fields and data inputs. This applies for the faults G03_0001, G03_0002, G03_0004, G03_0005.

You can get admin by editing the cookie (G03_0003), and this should be handled on the server side, so that the users can't do it. To fix the issue with password hashes (G03_0006), you can simply remove the code-snippet that inserts the hash into the site (in the template showuser.twig). Since the hash is not salted (G03_0007), it can be checked up in rainbow tables. By adding salting to the hash this problem would be fixed.

The stack traces from the site is public (G03_0008) and this can be fixed by turning off stack traces and redirect them to mail etc (so that the problem fails silently).

The Movie Review site has no password policy (G03_0010), and a password policy should be implemented. The password form field is a normal textfield (G03_0013) and this makes it easy to spy on other users passwords. The field must be changed to a password field. In the User.php-file one can see that the regex-pattern for checking new passwords accepts underscore as a character, even though the validation error states that only alpanumeric characters are allowed.

There is no session timeout (G03_0011) so a user session works forever. Sessions should have a timeout. The web server has not been configured with HTTPS (G03_0012), and therefore the web server should be reconfigured with HTTPS. This will enable security when sending GET and POST requests.

Every user on the site is publicly available with username (G03_0014). The usernames should be at least only available to logged in users. There is no requirements to be admin when deleting users by a given link (G03_0015 and G03_0016), and this must be changed and it should be added proper access control to it.

## Conclusion

We started the planning of our white box analysis under the impression that having access to the source code would reveal several more vulnerabilities. It came as a surprise to us that we were unable to find any additional high risk vulnerabilities, even though we also tested different static analysis tools (none of which were actually any good though). This might be due to the fact that the vulnerabilities were so wide-open and countermeasures were not implemented at all. Thus, enabling us to locate these vulnerabilities without much effort and specific know-how of edge cases where proper escaping, or use of non-secure built in escape-functions took place. Then again it also might be our lack of thorough understanding of the PHP language that bore the brunt of us not achieving new high risk findings.

## Appendix

### *A - Vulnerability Report Form - Ex1_part2*

See attachment **vrf_ex1_part2.pdf**