Security Evaluation Using OWASP Testing Guide

Hampus Dunström
Linköping University
Linköping, Sweden
hamdu013@student.liu.se

Olof Holmberg
Linköping University
Linköping, Sweden
oloho254@student.liu.se

Supervisor: Ulf Kargén
Linköping University
Linköping, Sweden

***Abstract*—As more and more services are becoming web-based and publicly accessible, malicious actors are attempting to abuse or break these services for their own gain. Security is therefore becoming more and more relevant in the software industry. With creative attackers we need rigorous security testing. To learn more about how to perform this testing we followed one of the more prominent testing guides, the OWASP testing guide v4. We as novice security testers were able to find a few vulnerabilities but could also conclude that it takes experience and expertise to use the guide to its full extent.**

I. Introduction

A. Motivation

There is no longer any doubt about the Internet's impact on our society and how many services that have been moved online. We pay our bills, do our taxes and buy our food through the internet, often using different websites. For example, today we keep in touch with Facebook[1] and handle our savings with Avanza[2]. But internet does not only attract great services but also bad actors trying to abuse the websites and the services they provide. Therefore it is important to have security in mind when developing these services and even more important, validating the security through testing. Therefore we have chosen to test a method for security testing developed by the Open Web Application Security Project (OWASP) Foundation[3] suggested by R. De Jimenez in the conference paper Pentesting on web applications using ethical-hacking [1]. This method is described by OWASP in the OWASP testing guide v4 [2].

B. Research Questions

1) How much time does it take to perform the testing described in the OWASP testing guide v4?
2) What types of vulnerabilities are possible to discover when using the OWASP testing guide v4?
3) What prerequisite knowledge do you need as a tester to use the OWASP testing guide v4?

C. Planusup

We needed something to test the OWASP testing guide on. For this we choose the online scheduling service Planusup[4]. We chose it because we had full access to servers, source code and could create accounts as we pleased. Planusup is a scheduling service designed for the needs of student organisations and organisations with a lot of workers coming and going not working for extended periods of time. The service is provided through a website on which administrators that schedule workers have accounts were they create plans. A worker can then sign up to a plan through a link and the administrator can schedule the workers that have signed up.

[1]https://facebook.com
[2]https://avanza.se
[3]https://www.owasp.org/
[4]https://planusup.se

D. Limitations

There were two big limitations for this project. First we only had a certain amount of time, approximately 100 hours, divided over two persons during which we should perform the security testing, write a report, prepare presentations, present the work and perform other course related work. The second limitation is our knowledge and experience of security testing and the tools involved. Neither of us had any previous practical penetration testing experience.

II. Theory

The OWASP testing guide v4 contains 12 steps for testing the security of a web application. Here is a brief explanation of each step:

A. Introduction and Objectives

In the introduction and objectives section, the OWASP guide offers some explanation of terminology, e.g. what a threat and what a vulnerability is. The phases of testing is also described: The passive phase where the tester tries to understand the application and the active phase where the tester follows the 11 other steps. The section also contains a checklist of what to test when assessing the web application. [2, Chapter 4]

B. Information Gathering

In the OWASP information gathering section the purpose is to gather as much information about the application as possible. This is done by utilising several services (search engines, web services etc.), programs (ZAP attack proxy[5], netcat[6] etc.) and the application itself. The information that should be gathered is first and foremost information about the web application and the server or servers that the application runs on. Information should also be gathered about the network infrastructure to detect multiple servers, firewalls, proxies or other network entities. [2, Chapter 4.1]

[5]https://www.zaproxy.org/
[6]https://nmap.org/ncat/

C. Configuration and Deployment Management Testing

In this OWASP section the found server and network entities are checked to find out if they contain known vulnerabilities. If the tester has access to the application (grey box testing) the application source code and logs are checked for information leakage. The allowed HTTP methods for HTTP requests to the web application is also tested. [2, Chapter 4.2]

D. Identity Management Testing

In this OWASP section the different user roles and the registration process is tested. Retrieval of user account details by registration or failed logins is also tested as well as the registration username policy. [2, Chapter 4.3]

E. Authentication Testing

In this OWASP section the authentication of the application is tested. This includes testing for secure credential transport (HTTPS), testing the login function for lock out mechanisms, default credentials or the possibility of bypassing authentication. Tests should also be done for password policy and strength as well as password reset, security questions or alternative authentication. [2, Chapter 4.4]

F. Authorisation Testing

In the OWASP Authorisation Testing section the purpose is to test for unauthorised file access, bypass of the authorisation schema (unauthorised access to functions or resources). The possibility for a user to escalate the user's own privilege is also tested as well as the possibility to access objects by direct references (usernames, filenames etc.). [2, Chapter 4.5]

G. Session Management Testing

To test session management the OWASP guide recommends testing the security of any cookies (or other tokens) that are used. This includes if and when the token/cookie is renewed, that it is encrypted when sent over the network, that it is not vulnerable to cross site request forgery (CSRF). Tests for handling the token at logout and session timeout are conducted as well. [2, Chapter 4.6]

H. Input Validation Testing

To test the input validation of the application all different ways to provide input to the application must be identified. When they are identified all inputs are tested for common vulnerabilities such as cross-site scripting (XSS), HTTP parameter pollution, different injections (LDAP, ORM, XML, SSI, XPath, IMAP/SMTP), code injection, file inclusions, command injection and different overflows (buffer, heap, stack). HTTP requests are tested as well for splitting, smuggling and if suspicious requests are sent in the background. [2, Chapter 4.7]

I. Testing for Error Handling

In this OWASP section the process of gathering information from error messages is described. By causing different errors in the application information about the web server, application server or database might be shown to the user thus granting the attacker knowledge. [2, Chapter 4.8]

J. Testing for weak Cryptography

OWASP recommends testing the cryptography of the ciphers used in SSL/TLS and other cipher suites used for encrypting information in the transport layer. Care should be taken to make sure that no information is leaked when decrypting information to prevent information leakage of the encryption cipher and/or suite. Tests should also be done to make sure that no sensitive information is sent over unencrypted channels. [2, Chapter 4.9]

K. Business Logic Testing

The OWASP guide recommends several tests for business logic including possibility of forging requests, integrity tests for data, timing between processes and function use limit. They also recommend testing for malicious use or mis-use of the application including circumventing the work flow and uploading unexpected or malicious files. [2, Chapter 4.10]

L. Client Side Testing

To test the application on the client side OWASP recommends testing for HTML and CSS injections, document object model (DOM) based XSS, JavaScript execution and client side URL redirects and resource manipulations. Tests should also be done for Flash and WebSockets if the application uses these techniques. If local storage is used review needs to be done to check that sensitive data is not stored there. [2, Chapter 4.11]

III. Method

In this section we will explain how we performed each step in the guide.

We approached this guide evaluation by first looking through the guide and noting which different areas the guide suggested to test. These areas, or steps, were then prioritised with the application and its implementation as well as our own interests in mind. The prioritisation was needed due to time constraints and a very thorough guide so we were not sure that we had time to perform every step in the guide.

A. Testing

When testing a step/area in the guide we first began by reading the description of the step and trying to get a basic understanding of what the purpose was and that we understood what needed to be done in order to carry out the test.

We then looked at the examples presented, if there were any, and then at the suggested tools and decided if we would use any tools. The decision on which tools to use was based on if they made the testing easier (i.e. if the tool automated the testing or made it easier to access the information needed to evaluate the step) or if the tool was required to follow the examples in the guide.

When we had a basic understanding of the step and installed any tools that were needed we began the testing. The testing was done mainly by following the examples, if there were any, by following the testing description or by following the manual of the tools used.

The results and more in depth description of how each step was performed and the time each step took was then documented and presented as the results of the evaluation.

IV. Results

A. Information Gathering

The purpose with this step was to understand the system about to be tested. What type of applications are running, what libraries are used and what ports are open are some example questions that we sought answers to. The goal was to find possible attack vectors.

We used the search engines DuckDuckGo[7], Google[8] and the GoogleHacker[9] tool to look for information leakage somewhere on the internet. We found the GoogleHacker tool unintuitive and hard to use and got very few results using the search engines. There was no robot.txt file and we only found 4 results from site:planusup.se for example. At the time of testing the system had only been live for two months which might explain the results.

To fingerprint the web server we tested the tools httprint[10] on a Linux machine and the webtool Netcraft[11] as suggested by the guide. We manually reviewed the source code for meta tags and especially looked for robots.txt to look for information leakage there. Our ICMP requests were blocked preventing httprint from yielding any results. Netcraft yielded more results from scanning both the system and the server provider. Netcraft gave the system a risk rating of 1 (very secure) and found:

- Netblock owner DigitalOcean, LLC
- Nameserver ns1.digitalocean.com
- DNS admin hostmaster@planusup.se
- Nameserver organisation whois.networksolutions.com
- Hosting company DigitalOcean
- Hosting Country US
- IP: 157.230.19.190
- OS: Linux
- Web server: NGINX
- 3 Known trackers from Cloudflare, MaxCDN, jQuery all delivered through CDN
- Web browser targeting:
  - Browser MIME type sniffing is disabled
  - Strict Transport Security (HTTPS only)
  - Do not allow this site to be rendered within an iframe
  - Block pages on which cross-site scripting is detected
  - Detect and mitigate attacks in the browser

Enumerating the applications on the webserver was only done briefly using *nmap -PN -sT -p0-65535 planusup.se*[12] since we knew exactly what applications were running and had limited time. The command *host -l www.planusup.se ns1.digitalocean.com* on Linux was used to try and find a list of other domains pointing to the same IP address used by planusup.se. We found that the open ports 22/tcp, 80/tcp, 443/tcp on the server and no other domains active on the same IP even though other domains were pointing to that very same IP.

The source code was also reviewed for information leakage in comments and meta tags. To identify application entry points we used the ZAP attack proxy that identifies which requests are sent ant what data they contained to understand how the application formed typical requests and responses. We stored all the found endpoints in a spreadsheet. We found no information leakage in comments and meta tags, this is probably the result of the transpilation of the source code in which all comments are removed. Using ZAP attack proxy we identified 16 different endpoints that could be possible entry points for attacks. The application had a total of 24 endpoints.

Mapping all execution paths within the applications to understand the workflows was only done briefly since we already had good knowledge about the application and we were limited on time.

When fingerprinting the application and the frontend frameworks used within it we used the browser plugin Wappalyzer[13]. For further investigation WhatWeb[14] was used to learn more about the backend, what server it was running and how it operated. Wappalyzer identified the frontend libraries Bootstrap, JQuery and Stripe[15] including what version was used. It did not recognise that it was a React.js application or anything about the backend. On other websites Wappalyzer could identify the database, webserver and backend language used. WhatWeb on the other hand discovered that the backend was running a NGINX webserver and that it was redirecting to HTTPS. Both tools were easy to use with the Wappalyzer only taking a few minutes to install and WhatWeb was a script downloaded from GitHub that required Ruby[16] to run.

Mapping the architecture of the application consisted mainly of the network and server infrastructure and since the application was hosted through DigitialOcean. The complexity and scope of such a mapping was judged to be to time demanding for us to perform.

B. Configuration and Deployment Management Testing

The goal with this step was to understand the configuration of the server hosting the application. Looking at network infrastructure, backup management, admin interfaces, what type of files are on the server and who has access.

As previously mentioned an investigation into DigitalOceans network infrastructure was out of scope so we skipped the network configuration step in this part of the guide.

To save time we only used greybox testing to test the applications platform configuration. We looked at default applications installed with the webserver. Also looking at

[7]https://duckduckgo.com/
[8]https://www.google.se/
[9]https://www.owasp.org/index.php/Google_Hacker
[10]http://www.net-square.com/httprint.html
[11]https://www.netcraft.com/
[12]https://nmap.org/
[13]https://www.wappalyzer.com/
[14]https://tools.kali.org/web-applications/whatweb
[15]https://stripe.com/
[16]https://www.ruby-lang.org/en/

logging, investigating who had permissions to the logs, how long they are stored, how they are rotated and if any sensitive information was stored in the logs. We did not do a comment review since it had already been done in the previous step. From the file permission investigation we found that system logs had reasonable permissions and were rotated once a day and stored for a week. The webserver NGINX logs were owned by the user www-data and the group adm. www-data had full access to the logs while adm only had read permissions. NGINX logs were rotated after a certain file size was reached and fifteen archived logs were stored. The application logs don't seem to be rotated but just piles on. They are owned by a human user and the group www-data both have read and write permissions. On the application logs all users have read permissions and sensitive data such as stack traces was found in the application logs. There was not any unusual data in the system log or the NGINX logs but only the currently active logs were looked at. The default application files from NGINX was still on the server but not reachable from the outside and it was a HTML file with links to the NGINX documentation.

We did not test file extension handling because the application did not allow file uploads and only served very few files contained within the webroot directory.

Manual greybox testing was used to review backups and unreferenced files for sensitive information. We found that backups are automatically done as snapshots of the server each week by the hosting service DigitalOcean. To access the backups you need to log in to the DigitalOcean account which were secured with two-factor authentication.

Since we knew there were no application admin interfaces we did not attempt to enumerate them.

To test what HTTP methods that the server supports the following nmap command was used where *X* is the port number (only scanned the open ones: 22, 80 and 443): *nmap -p X –script http-methods staging.planusup.se*. We did not manage to make Netcat work on windows to perform a similar test. We found that only GET, HEAD, POST, OPTIONS were allowed on port 80 and 443 and only STATE, SERVICE allowed on 22. None of these methods are considered potentially risky methods by nmap.

The HTTP header Strict-Transport-Security[17] is important for security since it tells the browser to only use HTTPS and no HTTP requests. We tried to search for this header with the recommended (by the guide) command *curl -s D- https://domain.com | grep Strict* and the Firefox developer tool. We Looked at both planusup.se and google.se for reference. Using the recommended curl command we did not find any Strict-Transport-Security header one either domain. When we looked for the header using Firefox developer tool to look at network request and then we could see that the Strict-Transport-Security header was present on both domains we tested.

[17]https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

We tested the RIA cross domain policy[18] by searching for the file crossdomain.xml both using a browser and manually on the server. No crossdomain.xml files were found.

The guide suggests some permissions guidelines for different kinds of files. By manually running ls and namei -l commands we checked whether the permissions on the server followed OWASPs guidelines. Many files was readable and executable by anyone and did not follow the OWASP recommendations. For example anyone could read the application log files as mentioned but also the python scripts on the server was executable by anyone which was not recommended by OWASP.

C. *Identity Management Testing*

There are only two roles within the system, users and administrators. With so few roles we deemed the testing of role definitions unnecessary.

The user registration process was tested manually using knowledge about the inner workings of the system. The goal for this testing step in the guide was to answer the following questions:

1) Can anyone register for access?
2) Are registrations vetted by a human prior to provisioning, or are they automatically granted if the criteria are met?
3) Can the same person or identity register multiple times?
4) Can users register for different roles or permissions?
5) What proof of identity is required for a registration to be successful?
6) Are registered identities verified?
7) Can identity information be easily forged or faked?
8) Can the exchange of identity information be manipulated during registration?

We found out that anyone who is willing to pay the registration fee can register an administration account while users do not have accounts. When someone registers for an account there is no manual vetting process, but humans are notified every time an account is created. There is nothing stopping a person from registering multiple accounts as long as they use different email addresses and pay for each account. There are no other roles to register for other than administrator. To register for an account you need to provide the following information:

- valid email,
- credit card,
- organisation name,
- organisation number,
- the person registers name,
- cardholders name.

For validation, the email is verified with regex and domain lookup. The credit card information is verified by a third party, Stripe. The rest of the information is not verified. Everything but the credit card could be forged or faked easily.

[18]https://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html

We could not discover any way to exchange or manipulate the information during registration.

Since you can only register for one type of account we did not test the account provisioning process.

We used manual grey box testing to test if it was possible to guess account names. The goal was to see if the system would give away when a valid account name was used. We tested both the sign in process and the reset password process. This testing was also done with given information about who the user was to see if it made it easier to guess an account name. It was not possible to guess an account name since the application gave the same error message regardless if the account name was correct and the password wrong or if just the password was wrong. Since the account name always was an email address if you know the person owning the account it is much easier to guess the account name, because a persons email address is often public. But you will not get any confirmation that the account name is correct even if you know who owns the account so you can't be certain that you have the correct account name until you successfully sign in.

D. *Authentication Testing*

Using the Firefox developer tools we checked if the requests containing username and password was encrypted using SSL/TLS. The credentials were confirmed to be sent over an encrypted channel.

No known applications was found or is known to exist in the system that might have default credentials therefore we could not test if we could log in with any default credentials.

To test for weak lock out mechanisms we used black box testing. By manually simulating a brute force attack we checked if there seemed to be any lock out mechanism to prevent brute force attacks on accounts. When using black box testing there seemed to be no lock out mechanism in place at all. This was confirmed as a known vulnerability of the system.

We used black box testing to test to directly access pages that should need authorisation. Parameter modification was not an option since no GET parameters are used in the system. Session ID prediction and SQL injections were skipped for lack of time and low chance of success due to use of SQLAlchemy[19]. SQLAlchemy should prevent injections by properly encoding the queries before they are sent to the database unless circumvented in the implementation of the application. The "create plan page" and the "plan list page" were possible to access using direct page requests but no sensitive data was exposed and any action resulted in server errors with generic error messages.

We did not test for vulnerable "remember password" functionality since the system does not provide any.

To look for browser history and cache weaknesses we tested to sign out from a page with sensitive data on and then use the back button to see if the sensitive data was still accessible. The browser cached was checked manually by viewing the contents of the stored cache files in Mozilla Firefox on a Linux machine. Using this method it was not possible to access sensitive data using the back button or by looking in the browser cache.

We created accounts with weak passwords using black box testing and checked the registration code for password constraints using grey box testing. No password policy was found. Users can freely chose their passwords including passwords with low strength such as the password "1".

There was no security question functionality to test. The only way to reset a users password is through a link sent to the users email.

We manually tested the reset and change password functionality to gather the information needed to perform password reset and how the process works. We used Postman[20] to try and send requests for resetting and changing passwords simulating a CSRF attack. It is possible for anyone to request a reset password link for a specific user but to reset the password you either need the token in the email or the existing password. If you can get either of these you will be able to reset the password. It was possible to send requests for resetting and changing the password using Postman.

E. *Authorisation Testing*

To detect directory traversal attacks a grey box testing for possible attack vectors were performed by searching for the opening of files in the backend source code. The only place were files were opened were in maintenance scripts and testing scripts only accessible for those with access to the server. Therefore no further testing towards detecting directory traversal attacks were performed. The next step was to try and bypass authorisation schemes by testing to access functionality and information that should only be allowed if the user is signed in. This was done both as a user who have never signed in and as a recently signed out user. If you were recently signed out or had never signed in did not seem to matter, in both cases we were able to access pages made for signed in users but no data was displayed only error messages. To test for privilege escalation we needed to try and forge sessions tokens which are all created by a external JSON Web Token library[21] which we deemed too time consuming and not probable to give any results. To make sure no signed in user can access other users data Postman was used to test different POST request using a valid token but trying to access other users data, but the server only returned general error messages.

F. *Input Validation Testing*

When testing for reflected cross site scripting we searched for input fields that would allow a user to send a custom crafted response to other users. Nothing that could be used to test for reflected cross site scripting was found on the application. Either no such methods of providing input existed or they were using proper character encoding preventing this vulnerability.

[19]https://www.sqlalchemy.org/

[20]https://www.getpostman.com/

[21]https://github.com/jpadilla/pyjwt

To test XSS we tested the only way someone is able to present stored XSS to another user. This was in the application's "plan sign-up" and "share-plan" functions. These were tested by supplying the following strings as either worker names or plan name:

- <script>alert('xss');</script>
- " onfocus="alert('xss')
- "><script>alert('xss')</script>
- <script>alert('xss')</script>

These did not result in any successful XSS attack. Mainly due to React's built-in escaping of code. However these were still stored as malicious text strings in the database and might be a security risk if the frontend implementation is changed.

The available methods that can be used for HTTP verb tampering were tested and documented earlier in the configuration and deployment management testing. The ones allowed that the OWASP guide cites as unnecessary or bad is OPTIONS and HEAD. The HTTP OPTIONS method is used to describe communication options for a specified resource. The HTTP HEAD method is used when responding to a GET request without a body. OPTIONS is however needed by the application whereas HEAD is not and therefore the OWASP recommendation is to disable the HEAD method.

The application did not utilize HTTP parameters and therefore no testing in regards to HTTP parameter pollution was performed.

Since the application uses SQLAlchemy the risk of a successful SQL injection is basically eliminated and no testing of SQL injection was performed.

When testing for SSI injection[22] the following string was used in those inputs where the result is presented to other users: *<!–#include virtual="/etc/passwd" –>*. This did not result in any successful SSI injection attacks as react seems to prevent it by good character encoding. The malicious input is however still stored in the database

Since XML is not used in the application no testing of XPath injections[23] was performed.

Mail services for the application is handled by external services which were not considered to be in the scope of testing so no testing of IMAP/SMTP injections was performed.

The only input fields available in the application are for queries in SQLAlchemy or used in python functions. Unless SQLAlchemy or the python functions used are vulnerable to code injection the application should not be vulnerable to code injection. Therefore we did not test for arbitrary code injection since python and SQLAlchemy are already well tested against this and any vulnerable functions would probably be found already. Finding functions that are vulnerable and not already documented would be too time consuming for this project.

No file inclusions are done either local or remotely so no tests were carried out for file inclusions.

No commands are sent in the URL or in the HTTP requests so no tests were carried out for command injection vulnerabilities.

No tests were carried out for heap/buffer overflows since the back end is written in python and the front end in JavaScript. To perform a buffer overflow attack is very difficult in those languages and the testing would definitely take more time than what is available for this project.

The application provides no means of any upload so there was no point in testing for incubated vulnerabilities.

Format string vulnerabilities seems to be a more prevalent threat in applications that uses C or C++ so no tests were carried out in regards to string formatting.

Since the application do not use either the *Set-Cookie* or the *Location* header in the HTTP requests no tests for HTTP smuggling[24] were carried out. Tests for HTTP splitting[25] were not carried out due to the time it would take to craft custom requests and that nothing besides JSON formatted data is usually sent in them.

In order to test for incoming HTTP requests, a proxy is required on the server. To install and make sure the proxy works will take too much time and we therefore decided not to test this.

G. *Testing for Error Handling*

To test for error handling the OWASP testing guide suggested sending crafted requests using telnet. This did not work for us so we sent the requests using ZAP Attack Proxy. The responses from the custom requests did not contain any information leaks about the application and only contained generic error messages if any. The same was concluded for the login functionality. Failure to provide a correct combination of password and email is met with the same error message: *Wrong password and email combination*. So those error messages can not be used to gain information whether a email is used or not. Supplying a too long input as plan name will either be met with the message *Server error* if it is reasonably long or with no message at all, probably due to a crash, if it is extremely long. However no sensitive information is leaked.

H. *Summary*

We performed 8 out of 12 steps in the guide. On which we spent about 25 hours per person resulting in 50 hours of work for the testing. We found two vulnerabilities. The first one was incorrect file permissions, they were not as strict as they should be resulting in a possibility that an attacker could get easier access to logs and source code. Possibly even execute some source code if he or she could get a foothold in the server. The second one is that an attacker can brute force the log in if he or she knows the account name.

V. Discussion

Here we will discuss our findings and answer our research questions.

[22]https://www.owasp.org/index.php/Server-Side_Includes_(SSI)_Injection
[23]https://www.owasp.org/index.php/XPATH_Injection
[24]https://capec.mitre.org/data/definitions/33.html
[25]https://www.owasp.org/index.php/HTTP_Response_Splitting

A. Method

The methodology we followed is the one described in the OWASP testing guide, limited by the time constraints of the project and our own experience and knowledge. Firstly each step was followed as described in the guide, however there might be missed vulnerabilities that belong to a performed testing step but were not found due to us not knowing what to look for other than what was described in the guide. Secondly we did not have enough time to perform too much research before a testing step other than reading the guide which also may result in undiscovered security issues that require deeper knowledge, previous experience or extensive testing in order to be discovered. Finally we did not have enough time to go through every step in the guide which also may result in undiscovered security issues due to lack of testing in that specific area. This in turn may affect the results of our application testing since there still may be undiscovered issues present. If we would redo the testing we would definitely devote more time to research and understand each area better in order to better understand what should be tested, how it should be tested and when it has been tested properly. The downside is that the required time is drastically increased since there might be a lot of extra research required depending on what technologies, frameworks, etc. are used in the application. A good knowledge of the tools that the OWASP guide suggests is something that probably benefits a tester since they might be able to automate some tasks or do a more thorough testing than manual testing. This is not something that we have specifically looked into, but we noted late in the project that the tools have more functions than the examples from the guide suggests.

B. Results

When going in to the OWASP testing guide our hypothesis was that we would find more classical vulnerabilities such as SQL Injections and XSS attacks. Probably because those are the ones you hear a lot about in web-based systems. Instead we found out that modern frameworks such as SQLAlchemy and React.js handles these well known issues very well. Instead it is more design level things such as being able to guess account names, brute force passwords or abuse reset password systems that we found ourselves looking more into and having more "luck" in finding vulnerabilities. We also found out that it takes more time than we thought to perform all the testing in the guide, though not all testing is relevant for all kinds of systems. We also found errors that were not security related but still presented unintended behaviour by the application or lack of information to the user. So the guide may be useful for finding other application issues and not only those that are security related.

C. Research Questions

How much time does it take to perform the testing described in the OWASP testing guide v4?

We were not able to complete the entire testing guide, spending about 50 hours and completing slightly more than half of the guide while performing each step quite briefly. A definitive answer is impossible to give here since each system is unique but we believe that a more experienced tester could go through the entire guide in about 60 - 80 hours for a smaller system like Planusup. With that he or she would be able to do each step more thoroughly than we did.

What types of vulnerabilities are possible to discover when using the OWASP testing guide v4?

The guide covers a wide variety of vulnerabilities ranging from your standard SQL Injections, XSS attacks to design level issues with account and role management, data leaks in form of both user data and information leakage about how the system works, and issues with the network and server architecture. We are not able to list all the vulnerabilities that you could possibly find with the guide but we can conclude that it is a broad guide covering most vulnerabilities you can find in a web-based system.

What prerequisite knowledge do you need as a tester to use the OWASP testing guide v4?

To properly use the guide we would consider that a tester should at least have in depth knowledge in each of the testing areas described by the OWASP testing guide. If the tester is not knowledgeable in the areas the tester might not know if everything included in a testing step is properly tested. More difficult exploits might also be missed if the tester is not proficient enough in the technologies used, for example very good knowledge of SQL might be necessary in order to find subtle injections. So both knowledge of the information in the guide and also the technologies used by the application is required in order to properly use the OWASP testing guide. But with only a basic understanding of software security you will be able to follow the guide to a certain extent and it will probably improve your security even though you will not be able to use it to its full extent.

VI. Conclusions

After this work we can conclude that security testing is a time consuming task if done thoroughly and it also demands a certain degree of both expertise and experience to succeed. The OWASP testing guide is a good friend when performing security testing for beginners as it gives a good place to start and for more experienced testers it is a good checklist for remembering each step and not overlooking something.

References

[1] Rina Elizabeth Lopez De Jimenez. Pentesting on web applications using ethical-hacking. In *2016 IEEE 36th Central American and Panama Convention (CONCAPAN XXXVI)*, pages 1–6. IEEE, 2016.

[2] Matteo Meucci et al. *OWASP Testing Guide 4.0*. OWASP Foundation, 2014.