Leiden University

ICT in Business

Creating a decision model to help web developers prevent their web applications for vulnerabilities

Name:	Joost Miljoen
Student number:	s1390813
Date:	25/04/2020
1st supervisor:	Dr. W. Heijstek
2nd supervisor:	T.D. Offerman MSc.
SIG supervisor:	S. Rigal, MSc.

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Acknowledgements

Writing this thesis had not been possible without the help of several parties. First of all, I want to thank my supervisors. Dr. W. Heijstek and T.D. Offerman MSc. from Leiden University and S. Rigal from Software Improvement Group (SIG) in Amsterdam, The Netherlands. They helped me while structuring this thesis, by connecting me with other experts in the field and by giving valuable feedback.

Also, R. van der Veer (*cyber security expert*) and M. Bruntink (*head of research*) played a major role in conducting this research. Last but surely not least, I want to thank every other person that contributed.

Abstract

According to the Web Application Security Consortium (WASC), 57% of the existing web applications contains cross-site scripting vulnerabilities. The chance of finding types of information leakage is also high: 40 percent. So is the chance of discovering vulnerabilities to execute so-called (SQL) injections: 19 percent. There are many more types of vulnerabilities. With limited resources, it may be hard to find out which vulnerabilities should be fixed first to minimize the impact of the misuse of vulnerabilities in web application code.

Goal of this research has been ranking the most-occurring vulnerabilities based on their impact and the difficulty of misusing them. It also tried to give a solution, to prevent web applications for the misuse of vulnerabilities with a high impact. Those solutions were ranked based on the resources it cost to implement them.

This research has answered the main research question: *how can an easily usable decision model be designed for web developers to prevent security vulnerabilities?*

To find an answer on this main research question, three sub-questions were identified:

- Which types of cyber security vulnerabilities can be distinguished, specifically injection vulnerabilities for web applications?
- How can web applications be prevented for the vulnerabilities found and which tools or frameworks do already exist?
- How can the results from the first two sub-questions be combined in a coherent model?

Answers to those sub-questions have been given by identifying the most occurring cyber vulnerabilities based on existing literature and OWASP (Open Web Application Security Project). Also, experts from the field were involved by checking the completeness of the created overview. Based on this overview, several defense methods were identified. As well as for identifying cyber vulnerabilities, existing literature, OWASP and knowledge from experts in the field were used. Information that has been found was used to create a landscape with vulnerabilities, their impact and defense methods to prevent the misuse of them.

The project resulted in an accessible model with vulnerabilities and their defense methods, designed as a decision tree and based on the landscape that has been created during this research. It should help web developers in preventing their web applications against vulnerabilities with a high impact in an efficient way, while resources are scarce.

New studies should focus on expanding the model and measure its effectiveness in practice. This has been one of the limitations of this study, as time was too limited to measure if using the model that has been created results in a lower impact of the misuse of vulnerabilities in web application code. It also focuses on the most occurring vulnerabilities, while there are many more less occurring vulnerabilities that may have a high impact on organizations.

Table of Contents

List of figures

List of tables

1. Introduction

Last years, cyber security has becoming an increasingly popular theme. Large organizations became victim of information leakages, small and medium enterprises became victim of hacked web applications and for example governmental organizations became victim of ransomware. Statistics of the Web Application Security Consortium (WASC) show that the chance of finding types of information leakage in web applications is 40 percent. Also, 57 percent of existing web applications contains cross-site scripting vulnerabilities. A less percentage of the web applications contains vulnerabilities to execute so-called (SQL) injections: 19 percent. Information leakages, cross-site scripting and (SQL) injections are mentioned by OWASP (Open Web Application Security Project) as the most occurring vulnerabilities.

For organizations it has become important to react quickly on potential cyber vulnerabilities in their web application code. Vulnerabilities and exploits in well-known frameworks are shared in public repositories, there are web scanning tools that may be used by attackers to find vulnerabilities in web applications and attack automation have become more popular. Without much technical knowledge, well-skilled attackers and script kiddies may misuse cyber vulnerabilities. Depending on the type of cyber vulnerability that has been misused, the impact on organizations may be large.

With limited resources it may be hard to find out which vulnerabilities should be fixed first to minimize the impact of the misuse of vulnerabilities in web application code. Especially small and medium enterprises do not have the resources to invest in skilled cyber specialists. For those organizations an accessible and easy to read decision model could help to prevent web applications for the misuse of cyber vulnerabilities. Although preventing web applications against all types of vulnerabilities will be hard, such a model could lower the possible impact of those attacks.

There are online sources with information about cyber vulnerabilities and defense methods to mitigate them. Especially OWASP is a well-known and highly appreciated source of information. IBM shows that developers may be split in two groups; hybrid developers and cyber security developers. Hybrid developers often have a background in software development and may have followed individual courses to cyber security. For cyber security developers it works the other way around: they have a background in cyber security and may have followed individual courses to software development (Leaser, 2019).

A growing demand for cyber security professionals shows that hybrid developers do not always have the right skills to mitigate all types of cyber vulnerabilities based on free accessible (online) sources. Last six years, the number of job postings related to cyber security has been increased with 94%. By comparison the number of IT jobs in general has increased with 30%. (Leaser, 2019)

This study works towards an accessible and coherent decision model that helps (hybrid) developers in protecting their web applications against vulnerabilities. The model should not only help small and medium enterprises, but it should also give developers of larger organizations a basis to prevent their web applications against the most occurring and impactful vulnerabilities. An accessible model can aid developers towards addressing a wider variety of vulnerabilities. Also, this model can help to explain managers on which possible cyber vulnerabilities they should focus.

1.1 Research questions

As IBM showed, hybrid developers do not know everything about cyber security which makes it necessary to hire specialists from the field. However, hiring those specialists may be expensive. For example, a hybrid developer on average earns 10% less than a cyber security specialist (Leaser, 2019). The combination with the high percentage of vulnerable web applications, as has been mentioned in the introduction, makes that an accessible decision model can add value to the work hybrid developers execute. To come to this model, the following main research question has to be answered:

How can an easily usable decision model be designed for web developers to prevent security vulnerabilities?

Before answering this main research question, three sub-questions will be answered:

- Which types of cyber security vulnerabilities can be distinguished, specifically injection vulnerabilities for web applications?
- How can web applications be prevented for the vulnerabilities found and which tools or frameworks do already exist?
- How can the results from the first two sub-questions be combined in a coherent model?

1.2 Research objectives

The main objective of this study is offering a method and reference for developers that aids them to recognize possible security weaknesses in their code. It also offers possibilities to control those weaknesses. As vulnerabilities exist in different types, this research has been scoped to the most occurring vulnerabilities and vulnerabilities with the highest impact.

According to the Open Web Application Security Project (OWASP), cross-site scripting and SQL injections are part of the top 10 application security risks of 2017 (OWASP, 2017). OWASP is an open source project in the field of cyber security, to which many companies, schools and individuals contribute. The organization has about 200 departments spread over 100 countries. Once in a while they publish a top ten of application security risks. A second source that confirms that those types of cyber security risks occur relatively often are statistics from Edgescan (2018).

Because it will be almost impossible to cover all cyber vulnerabilities, this study focuses on the most occurring ones. The same holds for defense methods. Instead of collecting all possible defense methods, this study selects a combination of *low hanging fruit* and deeper mechanisms. The combination of both should help developers in protecting their web application.

The key objective of this research is the creation of two models; a clear overview or landscape with a combination of vulnerabilities and related defense methods and a decision model that helps developers with picking the right defense methods against vulnerabilities in their web application. For example, using different roles is not necessary for applications without user accounts.

1.2.1 Economic impact of cyber vulnerabilities

It is hard to define the impact of cyber vulnerabilities. Compare for example the social- and economic impact of cyber attacks. In this case, social impact can be defined as the amount or sensitivity of data lost during a cyber attack. With economic impact the costs to resolve (the damage of) a cyber attack is meant. For this research, the economic impact of different cyber attacks will be compared. On the one hand because of the business impact of cyber attacks, which correlates with the economic impact. On the other hand, because the amount of data about the economic impact is quite bigger. This will make it easier to find vulnerabilities that have to be considered for the decision model.

A recent study by Accenture (2017) found that malware and web-based attacks are the two most costly types of attacks. Companies are spending an average of 2.4 million US dollars on recovering from malware attacks and 2.4 million US dollars on web-based attacks. Denial of services and malicious insiders are with respectively 1.6 and 1.4 million US dollars less costly. Another result of this research is the fact that malicious code attacks are taking longer to resolve. As a result of this, those attacks are costlier than others. For malicious code attacks it takes an average of 55 days to resolve the problem, while resolving for example ransomware attacks of web-based attacks takes respectively 23 and 22 days.

1.2.2 Overview of defensive methods

Besides creating an overview of cyber vulnerabilities, an objective of this research is investigating different defensive methods. Which methods are used for defending web applications against cyber vulnerabilities? What is the impact of implementing those methods?

For example, some defending methods are part of an existing framework. This will make it quite easy to implement them in a web application. For others, solutions should be built from scratch, resulting in a larger number of required resources. For example, creating those solutions requires technical knowledge.

By listing the defensive possibilities and their difficulty, they can be ranked. The objective here is to find out, which defensive solutions should be implemented first.

1.2.3 Working towards a coherent model

Both parts will help to come to the main objective of this research: creating a coherent model, which helps developers and managers to find out which defensive solutions should be implemented first. Resources as time, technical knowledge and money are scarce in a lot of companies. This model helps them defending their web applications against risky cyber vulnerabilities. Not only by pointing out which cyber vulnerabilities should be mitigated first, but also by suggesting valuable solutions.

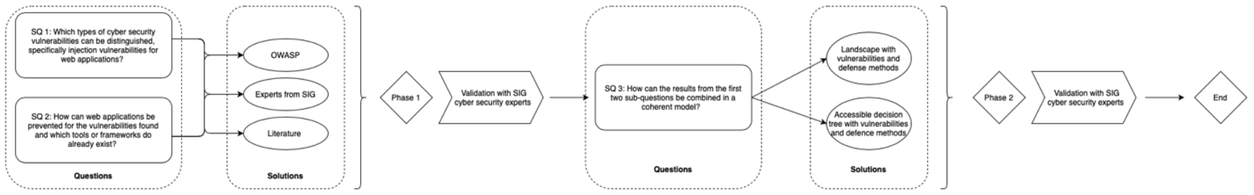

SQ 1: Which types of cyber security vulnerabilities can be distinguished, specifically injection vulnerabilities for web applications?
SQ 2: How can web applications be prevented for the vulnerabilities found and which tools or frameworks do already exist?
Questions
OWASP
Experts from SIG
Literature
Solutions
Phase 1
Validation with SIG cyber security experts
SQ 3: How can the results from the first two sub-questions be combined in a coherent model?
Questions
Landscape with vulnerabilities and defense methods
Accessible decision tree with vulnerabilities and defence methods
Solutions
Phase 2
Validation with SIG cyber security experts
End

Fig. 1: Overview of the used method during this study

1.2.4 Scoping of this project

This research focuses on security threats that evolve around "injections of instructions" in web applications. The number of existing vulnerabilities in web applications is far too high to mention them all. Thereby, the list with vulnerabilities is evolving during the period that has been worked on this research. The same holds for the number of defense methods. It is almost impossible to mention all possible defense methods against the found vulnerabilities.

This choice has been based on the number of occurrences of the vulnerabilities around injections. A common example is cross-site scripting (XSS). Also, the impact of those vulnerabilities. Misusing vulnerabilities itself will not directly have a high impact. For example, the server will not directly be damaged, and users may still have access to their accounts. However, as a result of misusing vulnerabilities around injections the server may spit out valuable data. An injection could for example result in an overview of usernames and passwords or an overview of financial details of customers.

The high impact of those vulnerabilities makes it relevant for developers to defend web applications against them. The lack of a coherent model with an overview of defense methods, makes focusing on those vulnerabilities in this research relevant. As the objective of this research is to help developers in preventing their applications against the most occurring vulnerabilities with a new decision model.

1.3 Research method

This research exists of different parts, with developing a decision model for defending web applications against cyber security risks as the main goal. To come to this model, the study will be done in different phases:

- *Phase 1:* literature research to the way injections, especially SQL injections and XSS injections, works. Those are the most occurring types of attacks on web applications these days, which will be further explained in the next chapters. The objective of this phase is to define different layers in web applications, vulnerable for these types of attacks. Also, this phase needs to give a better overview of the different types of vulnerabilities and their impact.
- *Phase 2:* in the second phase, different defense methods will be identified to prevent misuse of the found vulnerabilities. Also, ranking those defense methods based on their attractiveness is part of this phase. Attractiveness is determined by rating methods on their existence in frameworks, the possibility to implement them by yourself, the required technical knowledge and the impact of methods against vulnerabilities.

- *Phase 3:* the third phase of this research exists of designing the decision model, implementing relevant layers for defending web applications based on the literature research from the first phase and the identified defense methods of the second phase.

Validation of the models designed in this research will be done during talks with different experts in the field of cyber security. Those expert opinions should help in adjusting the models to an end product that helps developers in preventing their web application against the misuse of vulnerabilities. The experts who have been involved are mentioned in the acknowledgements of this thesis. In the discussion there has been zoomed in on ways to validate those models in a more extensive way.

1.4 Existing knowledge

There is a lot of existing research about the way SQL injections and cross-site scripting can be used to attack web applications, and for instance install malware on users' computers. Thereby, there is a growing amount of information about possibilities to defend web applications against cyber security risks and vulnerabilities.

For web developers, it will be hard to summarize this information and use it to prevent their own web applications against cyber security risks. As mentioned before, a coherent and accessible overview is missing. Here this study adds value, as different (injection) attacks and their possible defenses have been analyzed, to make defending web applications easier for developers.

Also, this decision model can form the basis of a new development or consultancy tool that checks code on cyber security risks mentioned in the model. To help users of such a tool, the tool can directly give a suggestion for minimizing those risks based on this new model.

1.5 Structure

The structure of this research corresponds to the different phases, as described in the research method. Chapter 2 and 3 focusses on web applications, injections and XSS attacks. In chapter 4 an overview including other vulnerabilities that occur often has been created. Also, in that chapter the impact and difficulty of misusing the identified vulnerabilities have been studied. Chapter 5 focuses on identifying different defense methods, as well as on classifying them based on their attractiveness. Based on the knowledge that has been gathered in those chapters, chapter 6 will focus on creating a coherent model. The last chapter, chapter 7, includes conclusions and a discussion on suggestions for further research.

2. History of web applications

The aim of this chapter is to give a brief introduction into the development of web applications, their growing popularity and the risks this brings with it. Also, HaaS-platforms are introduced and a short overview of the most occurring cyber vulnerabilities according to external organizations is given.

2.1 Popularity of web applications

Most new software development concerns web development. More and more consumers are using online banking apps, intranets within their organization and for example social networking sites. Even though frontend development is considered a specialization, within agile software development it is favorable when developers have “full stack” responsibilities to avoid artificial boundaries in responsibilities. Exactly these boundaries may cause delays and coding mistakes. This leads to a situation where a web developer is not only responsible for user-facing, but also for coding the backend. This made that in this study a developer is defined as a full stack web developer.

As more and more people are using those applications, for attackers it has become more interesting to misuse vulnerabilities. Databases of web applications are filled with valuable information. For example, it could contain combinations of usernames and passwords or financial details about customers. Thereby, information about known vulnerabilities in existing frameworks and extensions is shared online. It makes misusing vulnerabilities more interesting and easier for attackers. Thereby, attackers are using publicly available tools/scripts and machine learning to get access to complex web applications in quite an easy way.

Also, the market for attacks on those applications is exploding. Hacking as a Service (HaaS) is becoming a booming business. Attackers offering their services for a fixed price. For example, renting a botnet of 1.000 workstations to attack web applications with a DDoS-attack will cost $25 per hour (Makrushin Denis Makrushin, 2018). As criminals are asking 10 to 100 Bitcoins for stopping their attacks, it is a lucrative business. A DDoS-attack is not the only type of attack that can be ordered online.

2.2 Most occurring types of vulnerabilities

According to research of the Web Application Security Consortium (WASC), the most widespread vulnerabilities are cross-site scripting (XSS), SQL injections, HTTP response splitting and different types of information leakage (Web Application Security Consortium, 2008). This research shows that the chance to find a cross-site scripting vulnerability (57%) and different types of information leakages (40%) is high. The chance of finding a SQL injection vulnerability is somewhat lower: 19%, according to this research.

How does such a large number of web applications become victim of cyber security vulnerabilities? According to Edgescan (Edgescan, 2018), 29% of web applications has an insecure configuration. This means that e.g. developers are enabling debugging and for example, are using insecure HTTP methods and insecure protocols or unsupported frameworks. The fact that organizations often work with DevOps teams make it difficult to see under whose responsibility this configuration falls. A lack of clarity brings a high risk of errors with it. Also, 24% of those applications are lacking in client-side security. Working with a poorly secured server, there is a risk of cross-site scripting (XSS), clickjacking, form hijacking and for example cross-domain leakage. This is in line with statistics of WASC.

A third factor to consider is the combination of hacking on demand and online libraries with lots of information about vulnerabilities in existing frameworks and their extensions. This makes it quite easy for attackers to misuse those vulnerabilities and break into web applications or users' systems, without having much knowledge about those systems. Misusing vulnerabilities does not directly have a high impact on businesses. However, the possibilities an attacker has after breaking into a web application does have a high impact. An attacker could for example steal sensitive data of users, or could add malicious code on web pages and attract visitors to click on it.

2.3 Architecture of web applications

Web applications normally consists of two main components: *a database* and *webpages*. Most often this architecture is called a client/server-model, because of the relation between the server delivering information to a client. Those webpages contain server-side scripts, that can extract information from the connected database. Clients are consuming information the server serves them.

In most cases those web applications are built with three tiers:
- *Presentation tier* (web browser or rendering engine)
- *Logic tier* (a programming language)
- *Storage- or data tier* (database)

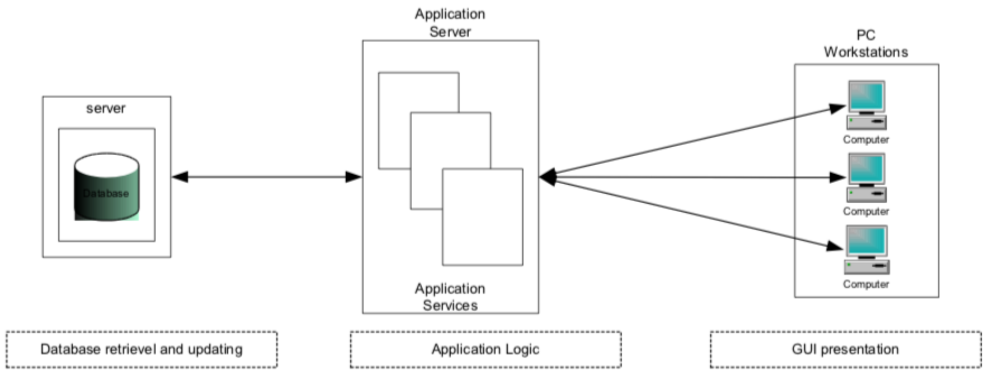


Fig. 2: Client/Server 3-tier architecture (Kambalyal, n.d.)

Last years, many new projects are built with more than three tiers. Those are often described as a multi-tier architecture or n-tier architecture, where 'n' refers to any number from one. Using more than three tiers is useful, while separating different functions of a web application. This makes it possible to prevent parts of the web application from using the same resources. As well as that editing part Y will not directly affect part X. An important different between the described 3-tier architecture and this n-tier architecture is the fact that a 3-tier architecture separates the three different layers only in concept. Within a n-tier architecture those layers can be separated physically. It helps developers in securing their application, because each layer can be secured and managed separately from the others.

2.4 Human factor in vulnerabilities

An important factor to look at by finding out why web applications may be vulnerable is the human factor. Based on Danhieux, a globally recognized security expert, application testing since 2015 has not shown much improvement in the number of vulnerabilities found. Thereby, the same old flaws keep coming up time and time again (Danhieux, 2018). For example, SQL injections exist since 1999. However, based on a report of Veracode one in three new scanned applications in 2017 still contains vulnerabilities for SQL injections. For this Veracode report (Veracode, 2017), 400.000 applications were scanned. In 70% of the time they contain vulnerabilities from the OWASP Top 10.

According to IBM, organizations should improve their web application security capabilities to prevent web applications from being successfully attacked. Implementing privacy and security as part of the Software Development Life Cycle (SDLC) should be an important part of it. Currently, there are many organizations that do not work this way (Poremba, 2019).

As part of implementing privacy and security in the SDLC, Danhieux suggests building in hands-on secure code training. Developers needs access to hands-on learning to improve their capabilities in writing secure code for web applications. For example, developers need to learn about the latest vulnerabilities in general and specifically for their own coding languages and frameworks (Danhieux, 2018).

For this research, a distinction between different types of cyber security attacks has to be made: the ones focusing on the server-side of a web application (i.e. SQL injections) and the ones focusing on users of it (i.e. XSS injections). Especially for those attacks that focuses on damaging users, developers do not have the possibility to prevent users for this as users have their own role in it.

2.5 Recap

In this chapter there has been zoomed in on the growing popularity of web development, in relation to the risks this brings with it. An increasing complexity of those web applications will make it even more difficult for web developers to mitigate every possible cyber vulnerability in it. Thereby, HaaS-platforms are easier to access and offering different types of attacks for relatively low prices to people without any technical knowledge.

3. Introduction into web application injections

The aim of this chapter is to explain how, in general, injections work. With those injections SQL queries can be manipulated to adjust user information or to request this information from the server. Later on, several other types of the misuse of cyber vulnerabilities in web applications will be discussed.

3.1 Introduction into web application injections

Untrusted data is one of the main drivers of injection attacks. In most cases untrusted data is sent via an HTTP request, for example in the form of URL parameters, form fields and headers. Those URL parameters and form fields can be manipulated easily, to execute a cyber attack. In some cases, those attacks are executed to directly damage the server-side of a web application, in other cases attacks are executed to directly damage the client-side.

An example of such an attack is a SQL injection, where untrusted data is included in a database query. Take for example a query as *"SELECT * FROM users WHERE name='" + request.getParameter("name") + "'";* where *"name"* can be replaced by a statement of a hostile user. Such a statement can include an attack. A SQL injection is just one example of an injection. There are many more: LDAP injection, XML injection, JSON injection, etcetera (OWASP, n.d.). In some cases, it is also possible to change the complete statement. Attackers can use concatenation to build up their own statements and force the server to spit out the requested data.

To prevent a web application from being damaged by an injection, all untrusted data should be analyzed and neutralized before it is sent somewhere else. Developers should treat untrusted data as if it contains a cyber attack.

3.1.1 Role of a parser in command injections

During a command injection attack, attackers try to target parsers. Parsers are a component of every interpreter and their main purpose is to determine if input data may be derived from the start symbol of the grammar. The way a parser works normally consists of three steps: a lexical analysis, syntactic analysis and semantic parsing (Techopedia.com, n.d.). Attackers try to manipulate data, to force such a parser to interpret data as commands. When a parser executes data as a command, an attacker can cause damage to web applications and related systems. In those parsers the key to a successful command injection can be found, or more important: the key to a successful defense method against those injections.

3.1.2 XSS attacks in relation to injection theory

Cross-site scripting (XSS) attacks are a form of an injection. For this special type of injections, the JavaScript engine in the browser is the interpreter and cyber attacks are executed in an HTML-file. For developers it is hard to defend applications for XSS injections, as HTML-files normally contains a lot of places to put hostile code. Thereby, there are many different valid ways for encoding. As mentioned before, every interpreter has his own parser. For defending a web application against injections, it is necessary to understand how this parser works. However, for

HTML-files there are several parsers: XML, JavaScript, CSS, etc. This makes it even harder to defend web applications against injections and in special, XSS attacks.

XSS can be seen as a misnomer. Although this type of attack is executed across sites in most cases, this is not always the case. It depends on the type of XSS attack the attacker uses. There is some difference between for example a DOM based XSS attack and a stored or reflected XSS attack. As the attack across sites could be seen as the most occurring one, XSS attacks have been assumed as this type.

3.2 SQL injections in web applications

Injections in web applications come in different types: direct and less direct. With a less direct SQL injection an attacker can insert or append SQL code into the application input parameters. Now this SQL code will be passed to a back-end SQL server for parsing and execution. A more direct form of SQL injection consists of direct insertion of the code into the database. To execute such a direct SQL injection, an attacker should find a way to break into the database (Microsoft, 2017)

There are several places where SQL statements can be executed. For example, a database server, web server or application server. Also, there are different types of commands that can be included in SQL queries executed by a server, for example: SELECT, UPDATE, INSERT and DELETE. In such a query there are different places to inject code: in the FROM section, the WHERE section or the ORDER BY section.

Different types of SQL vulnerabilities (Clarke, 2012):
- Incorrectly handled escape characters
- Incorrectly handled types
- Incorrectly handled query assembly
- Incorrectly handled errors
- Incorrectly handled multiple submissions
- Insecure database configuration

3.2.1 Testing for SQL injections

In most cases attackers don't have access to the application source code. Because of this, testing by inference is necessary. Testing by inference means that an attacker sends several requests to a server and tries to detect anomalies in the response from a server *(Clarke, 2012).*

There are several things to consider here:
- Identify all the data entries on a web application - try to find out how a web browser sends requests to the web server.
- While using web applications there will be a two-way communication between a server and a client. To make this communication possible, web applications use a protocol. In most cases this will be HTTP. This protocol defines a set of actions that a user can send to the server. For discovering SQL injections, two types are most used: GET and POST.

Another way to test web applications for injection vulnerabilities is using browser modification extensions. Those extensions can be installed in a web browser and make it possible to visualize hidden fields or remove size limitations. There are several browser modification extensions that can be installed in for example Mozilla Firefox or Google Chrome *(Clarke, 2012)*.

3.2.2 Cookies

Besides the vulnerabilities from GET- and POST-requests, cookies can be seen as a data entry. For each request to the server, cookies are sent. They are used for authentication and session control. Data sent in those cookies can be changed to manipulate the web server.

3.2.3 Headers

A third data entry that may be misused by attackers are headers. HTTP headers can be manipulated manually, which makes it possible to ask the server for valuable information. Those headers exist of different fields, that are part of the message that defines the operating parameters of the HTTP transaction.

Although there are several ways to test a web application for protentional injection vulnerabilities, it all depends on the fact if the server will accept a request of an attacker. After accepting and executing the request, valuable data may be shared by the web application.

3.3 Manipulating the database layer

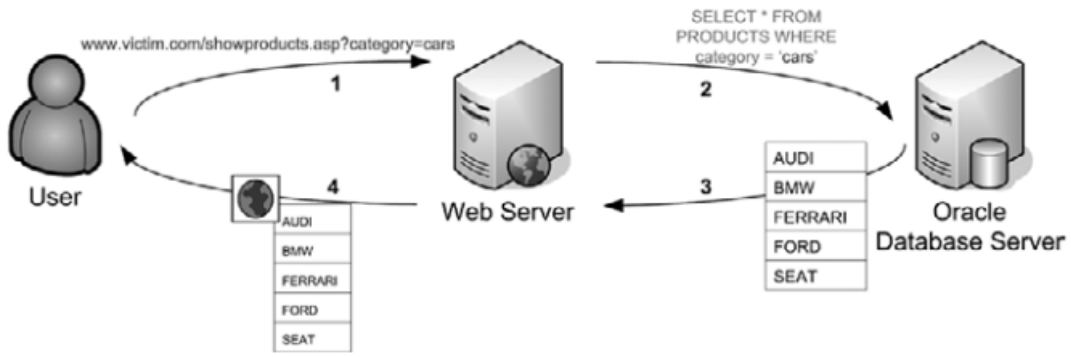

www.victim.com/showproducts.asp?category=cars
1
SELECT * FROM PRODUCTS WHERE category = 'cars'
2
User
Web Server
Oracle Database Server
3
4
AUDI
BMW
FERRARI
FORD
SEAT

Fig. 3: Information flow in a 3-tier architecture

While executing a SQL injection, the database layer will be manipulated. A user sends a request to the web server. From the web server a SQL query is sent to the database, which executes the query. In case the query can't be executed by the database, for example because requested data isn't reachable, the database sends an error back to the web server. The web server shows the error to the user.

An attacker wants to know how the database is responding on a request. It depends on the database type, how this database is reacting. There is a difference between a *Microsoft SQL Server,* a *MySQL server,* an *Oracle database* and a *PostgreSQL server (Clarke, 2012)*.

Based on the reaction of a server, an attacker can determine if and how a web server or web application is vulnerable to SQL injections. However, the web server doesn't always show a clear description of the error. The web.config-file determines how the web server will react after an error has occurred. It is important to mention that for this explanation the Microsoft .NET engine is used. Consider that not every web application reacts in this way, as not all of them are built with a Microsoft .NET engine. It does not mean that those applications are not vulnerable.

One way to execute a SQL injection or test a web application for potential vulnerabilities, is using an *always true*-statement. In practice this means that "OR'1' = '1" is added to the query. This addition can be seen as the most classic type of testing web applications for SQL injection vulnerabilities. It is not a search term, but a logic statement that will be blindly executed by the server of a vulnerable web application. There are many more ways of testing web applications, as it has been mentioned earlier. In case the web application is vulnerable for SQL injections, all rows in the database table will be returned. However, this *always true*-statement has a disadvantage. In case a table exists of millions of rows, it will take a while before those rows are returned.

In case of a "login" field, this *always true*-statement will help an attacker to find out if a field is vulnerable to a SQL injection. When the *username*-field is filled with "user' OR '1'='1", an error may occur. "Invalid password" is an example of such an error, but there are more possibilities. Now, the only thing that has to be checked is the response on an *always false*-statement. This means that the *username*-field will be filled with "user' AND '1'='2". If the response of the web server is different from the *always true*-input, the *username*-field is vulnerable to a SQL injection. The *always true*-statement may be used in a query concatenation to let the database spit out valuable data.

3.4 Recap

In this chapter there has been zoomed in on the way injections can be used to manipulate a web- or application server. Manipulating those servers gives an attacker the possibility to execute dangerous SQL queries. For example, an attacker can ask the server to export information about users or to adjust user passwords. This gave attackers access to accounts, which may result in even more damage. Injections are an important type of the misuse of vulnerabilities in web applications. However, there are many more potential vulnerabilities and attacks. In the next chapter different types of vulnerabilities will be discussed.

4. Creating an overview of cyber vulnerabilities

The second chapter gave some insights in the most occurring vulnerabilities. This chapter focuses on creating an overview of cyber vulnerabilities, by creating activity flows. Those activity flows give insight in possible weak points in web applications. Also, this chapter helps ranking vulnerabilities based on their economic impact and difficulty. Keep in mind, that there are several other vulnerabilities that are not considered in this overview.

4.1 Approach for this chapter

Before creating different activity flows, an overview of the most occurring cyber vulnerabilities according to existing research has been made. Activity flows were based on those vulnerabilities, to check if every potential vulnerability has been covered. For every vulnerability, its difficulty for attackers has been described. The OWASP Top 10 of 2017 (OWASP, 2017) is used as a basis for creating this overview, as well as several feedback moments with experts from the field and The Web Application Hacker's Handbook (Stuttard & Pinto, 2011).

4.1.1 Brute force attacks

A user has the possibility to enter a system, while combining a username with its password. The server has to check if this combination is correct. If this is the case, the user can enter the shielded part of an application. A session is created, so that a user does not have to repeat this action again and again (Stiawan et al., 2019). For attackers this function in web applications gives them the possibility to:

- An attacker tries to enter a large number of usernames and passwords, to guess the right combination. When the right combination is found, an attacker can enter the system.
- By entering an incorrect combination, an attacker may find out if a username exists in the system. This is the case when the system returns that only the entered password is incorrect. Now an attacker can try a large number of passwords for a specific username to enter a system. This type of attack can also be used, when an attacker wants to know if a user has an account on a "bad" website.
- An attacker attempts to enter a large number of usernames, to find out whether there is a logical follow-up of usernames (user234, user235, etc.). Now an attacker can combine this logical follow-up of users with standard passwords to gain access to the most recently created accounts.

Difficulty for an attacker: an attacker needs to have a server or strong pc to run a script, that tries a lot of username/password-combinations. It is a time-consuming process, without the certainty that the attacker will find the right combination.

4.1.2 Misuse of password change function

An application offers a user the possibility to change his password. While doing so, a user normally first has to enter his old password before choosing a new one. Here it is necessary to enter a new password twice to validate whether a user has spelled the password correctly. For attackers this function in web applications gives them possibilities:

- It may be possible that an application does not ask for the old password, assuming that a user can only use the form if the user is already logged in. An attacker can attempt to reach the form without first having to log in by entering the URL directly into the browser. An attacker now has the option to change the password and access the system.

Difficulty for an attacker: easy to try (for example with an owned account) and easy to get access, when it works. However, an attacker needs to find out if an account exists in the database of the application. Thereby, the web application has to use for example an unsafe framework with design errors in it. Those design errors make it possible to directly access the password change function, without having access to a user account. Also code written by developers of the web application may contain those type of design errors.

4.1.3 Misuse of account recovery function

A system offers users the option to reset their password. Users enter their username or email address, after which the system sends an email with a password reset link to the user. This may lead to the following situation:

- A system sends a link to a user to recover his password. It has been assumed that this link consists of different parameters. An attacker can manipulate this link by adjusting parameters (IDs, usernames, etc.). In this way an attacker can change the password of other users without having access to a user's email account.

In some cases, a user has to answer a recovery question to change a password. Here an attacker can try to find the answer by blindly guessing. A second option is requesting a hint to remember a password. For attackers this function in web applications gives them the possibility to:

- The password recovery form may allow an attacker to request a hint from the system. An attacker can then try to find the answer to this question via social hacking/engineering (i.e. finding answers via social media). (Bonneau, Bursztein, Caron, Jackson, & Williamson, 2015)
- A system can give a user the possibility to come up with his or her own account recovery hint. A user may be inclined to choose a simple question (What is your mother's name? What is your place of residence?). Attackers can find the answer to this question with information from social media. When the answer is found, an attacker can log in into the system.

Difficulty: easy to try (for example with an owned account) and easy to get
access, when it works. However, an attacker needs to find out if an account exists in the database of the application and social hacking can be a time-consuming process.

4.1.4 Cookies

The moment a user logs in into a system, a new session is created to prevent a user from having to log in again for each action. Information about this session is stored in a cookie, which is placed on a user's computer. For attackers this function in web applications can give them the possibility to:

- An attacker can trick a user into executing a piece of code (click on something, send malicious code in an email, etc.), which drives the user's computer to send a copy of a user's cookie. These cookies store information about a user's session within an application, that can be used by an attacker to get access to the user's account. (Stuttard & Pinto, 2011)

Difficulty: a more difficult attack, because an attacker has to find users of a certain web application and try to attract them to click on a malicious piece of code. Thereby, cookies are limited in their applicability. In most cases they are linked with the domain, where the cookies are issued. Thus, intercepting cookies will not directly mean that attackers can take over users' sessions within web applications.

4.1.5 Weaknesses in the generation of session tokens

A system offers a user the ability to log in, giving a user access to a trusted environment. The moment a user logs in, a session is created. As a consequence there is no need to log in when performing individual actions. A token is created to maintain the connection between the session and the server (Stuttard & Pinto, 2011). For attackers this function in web applications gives them the possibility to:

- An attacker could attempt to guess the value of other users' tokens based on the token created for himself in a session. By having a user looking up his own token, he gets an idea of the structure of tokens in general within an application. This can be seen as a type of information leakage, as it makes it possible to guess tokens issued to other users of the web application.

Difficulty: an attacker needs to know how a token is generated and which parameters it includes. Based on this information, an attacker can try to guess the token of other users. However, guessing the right token isn't easy because of the randomness in token values.

4.1.6 Weaknesses in the handling of session tokens

A system uses session tokens, that has to be sent via a secure connection. If not, attackers can try to steal data out of those tokens. For attackers this function in web applications gives them the possibility to:

- An attacker can access the information in a token when it is sent via an unsecured connection.
- An attacker can remove the token from the URL of the website, in order to decrypt it and steal information.

Difficulty: an attacker needs to know how to intercept a token from an unsecured connection. In some cases, the attacker needs access to the server. However, it makes more sense that the attacker will use the user side of a web application to intercept a token. In those cases, access to the server is not required.

4.1.7 Securing access controls / parameter tampering

Based on the URL of a website, users may have the ability to see where certain files are saved on a server. For example: http://url.com/files/documentX.doc. Here it has been assumed that a server doesn't check if a user has the necessary rights to visit certain files on a server. For attackers this function in web applications gives them the possibility to:

- To gain access to those files, an attacker can try to manipulate the URL of a website, by for example changing the document IDs or filename.

Difficulty: an attacker needs to find one file on the server, to analyze the URL
of it and start guessing the URL of other files. Although this will be time consuming, it is not that hard.

4.1.8 SQL injections

Many websites have a search field, for example, to search for products or pages. These search fields often work with parameters, which in some cases are processed in the URL of the website (after entering a search query). It is possible to adjust these parameters. Those injections come in different types, besides SQL injections also NoSQL injections and for example XPath injections work in this way (Clarke, 2012). For attackers this function in web applications gives them the possibility to:

- An attacker can manipulate the URL after entering a search query to make the database spit out something. This can be an overview of users and passwords, for example.

Difficulty: an attacker needs to check whether a web application is vulnerable for SQL injections. In case an application is vulnerable, an attacker needs to find the right query to get data out of the database.

4.1.9 XSS attacks

A website can be sensitive to an XSS attack in different ways. For example, an attacker can manipulate the URL of the website in such a way that when calling the URL, a piece of JavaScript code is executed. Pieces of code can also be called and executed via an input field (Fogie, Grossman, Hansen, Rager, & Petkov, 2007). A third way of executing this type of attack, is by hiding JavaScript code within the URL by coding them in a different way. For example, an attacker can use HTML coding instead of UTF8. Here the attacker hopes that the application accepts the code and executes it in its original lay-out. For attackers XSS vulnerabilities in web applications gives them the possibility to:

- An attacker can manually add a piece of code to the URL of a website. An attacker can then send this URL to users, hoping that they will use it. As a consequence the JavaScript-code is executed.
- An attacker can enter a piece of code in an input field on a website to have this JavaScript-code executed. For example, in a form or comment field.

Difficulty: an attacker needs to check whether a web application is vulnerable for XSS attacks. In case an application is vulnerable, an attacker needs to create a piece of code that executes a certain action on the users' system. In the last step the attacker has to attract people to click on a URL in an email or on a website.

4.1.10 Cross-site request forgery (CSRF)

After a user has created a session by logging in into a web application, a Cross-Site Request Forgery (CSRF) can be performed by an attacker. In this case it has been assumed that the session is still available (Jovanovic, Kirda, & Kruegel, 2006). For attackers this function in web applications gives them the possibility to:

- An attacker can send a victim a link by email to have the victim perform an action. The focus is not on stealing data, because the attacker cannot reach it. The purpose of a CSRF is to implement a change in the status of a user (i.e. transfer money, change an email address, etc.).

Difficulty: an attacker needs to check whether a user of the web application is vulnerable for CSRF injections. Here the attacker is less dependent on the framework of the web application. A second prerequisite is the fact that a user should be logged in into a web application. Based on this existing session, an attacker can try to attract a user to click on a URL in an email or on a website.

4.1.11 Default credentials

While creating new accounts, many frameworks and middleware use standard usernames and passwords (i.e. Admin - Admin). These passwords can be guessed relatively easily by attackers. For attackers this function in web applications gives them the possibility to:

- A brute-force attacker can try to combine a large number of standard usernames and passwords to gain access to the application server.

Difficulty: an attacker needs to guess default credentials for new or recently created user accounts on web applications.

4.2 Categorization of cyber vulnerabilities

The identified cyber vulnerabilities can be spread out over different categories. First of all, with the misuse of some of those cyber vulnerabilities it is possible to gain access to the application. This means that functions within the web application can be misused. Access can be gained via user accounts, system functions and external sources. With external sources parts of the web application are meant. For example cookies, that are stored on a user's computer. A second way vulnerabilities can be misused is by gaining access to data on the application server or database. A third way of misusing the vulnerabilities in web applications is gaining access to a user's system. Based on those categories, the different activity flows will be generated.

- **Gain access to the application**
  - *Via user accounts*
    - Brute Force Attacks
    - Misuse of account recovery function
    - Default credentials
    - Steal user credentials via a malicious WiFi-network
  - *Via system functions*
    - Misuse of password change function
    - Weaknesses in the generation of session tokens
    - Weaknesses in the handling of session tokens throughout their life cycle
  - *Via external storage of sensitive data*
    - Cookies

- **Gain access to data**
  - *Via URL-manipulation*
    - Securing Access Controls
    - SQL injections
    - Injecting into NoSQL
    - Injecting into XPath

- **Gain access to user's data/system**
  - *Via executing small parts of code*
    - XSS vulnerabilities
    - Cross-Site Request Forgery

Below you find a visualization of those categories and the different vulnerabilities, that are part of those categories. It is important to mention that the vulnerabilities are located on random spot below the categories. In other words, there is no ranking within the categories.

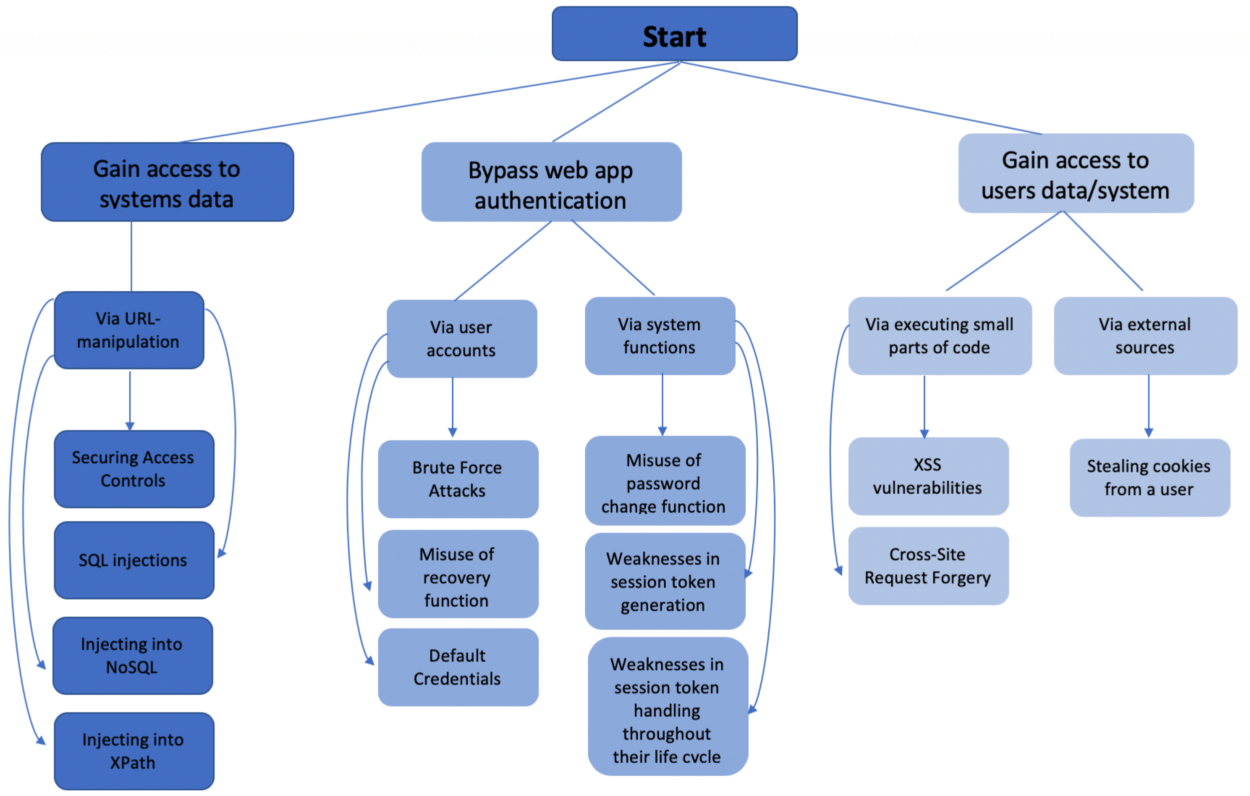
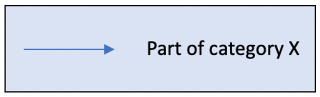

Start
Gain access to systems data
Bypass web app authentication
Gain access to users data/system
Via URL-manipulation
Securing Access Controls
SQL injections
Injecting into NoSQL
Injecting into XPath
Via user accounts
Brute Force Attacks
Misuse of recovery function
Default Credentials
Via system functions
Misuse of password change function
Weaknesses in session token generation
Weaknesses in session token handling throughout their life cycle
Via executing small parts of code
XSS vulnerabilities
Cross-Site Request Forgery
Via external sources
Stealing cookies from a user

Fig. 4: Overview of different attack types

Explanation:



4.3 Activity flows

The categorization of the identified vulnerabilities leads to four activity flows, which are further elaborated on the coming pages:

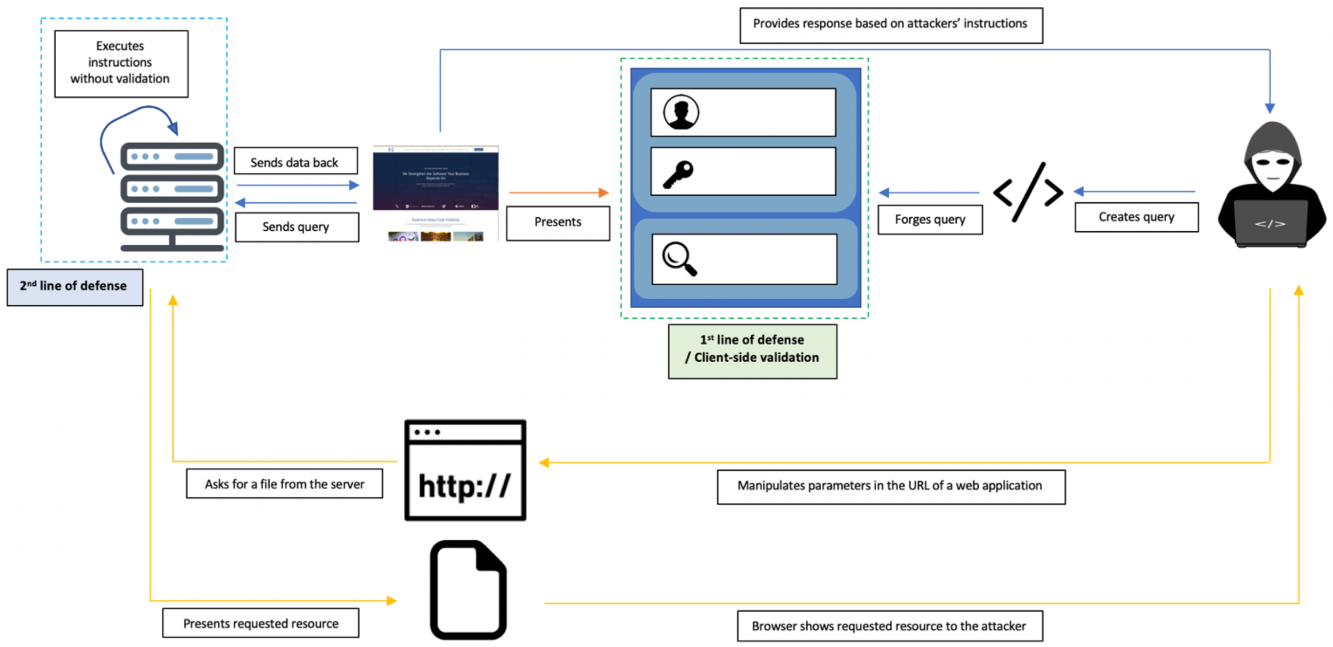
- Gain access to systems' data
- Gain access to users' data
- Bypass web app authentication (via user accounts)
- Bypass web app authentication (via system functions)

Those activity flows show in different steps, how a vulnerability can be misused by an attacker, as plain text can make it hard to understand the idea behind a cyber vulnerability. Thereby, those activity flows will help while checking if every relevant vulnerability is identified. In the legend below the activity flows a description of the different factors within the flow can be found. As earlier has been mentioned in this study, it is impossible to mention every vulnerability. Therefore, only the most occurring and impactful vulnerabilities based on the literature and OWASP Top 10 has been included.

In those activity flows there are different *lines of defense*. The first line of defense is shaped by the front end of the web application. For example, the way an input in a form field is handled by the application. The second line of defense focuses on the application server or application database. With the third line of defense, the role a user can play in preventing the misuse of cyber vulnerabilities by an attacker is meant.

Identifying different lines of defense will make it easier to find defense methods against the cyber vulnerabilities. In every line, there are ways to stop attackers from misusing one of the vulnerabilities in web applications.

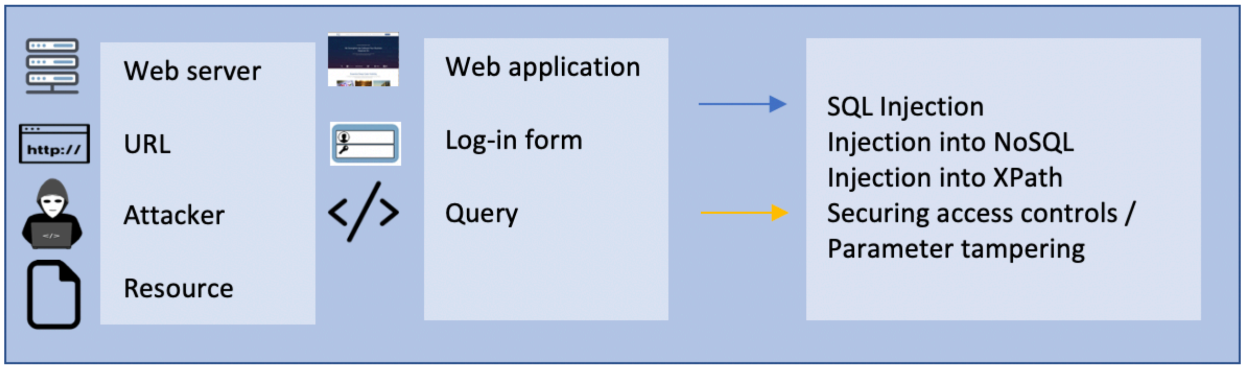
4.3.1 Gain access to systems' data



Provides response based on attackers' instructions
Executes instructions without validation
Sends data back
Sends query
Presents
Forges query
Creates query
2nd line of defense
1st line of defense / Client-side validation
Asks for a file from the server
http://
Manipulates parameters in the URL of a web application
Presents requested resource
Browser shows requested resource to the attacker

Fig. 5: Overview of attacks to gain access to systems' data

<u>Explanation:</u>



4.3.2 Gain access to users' data

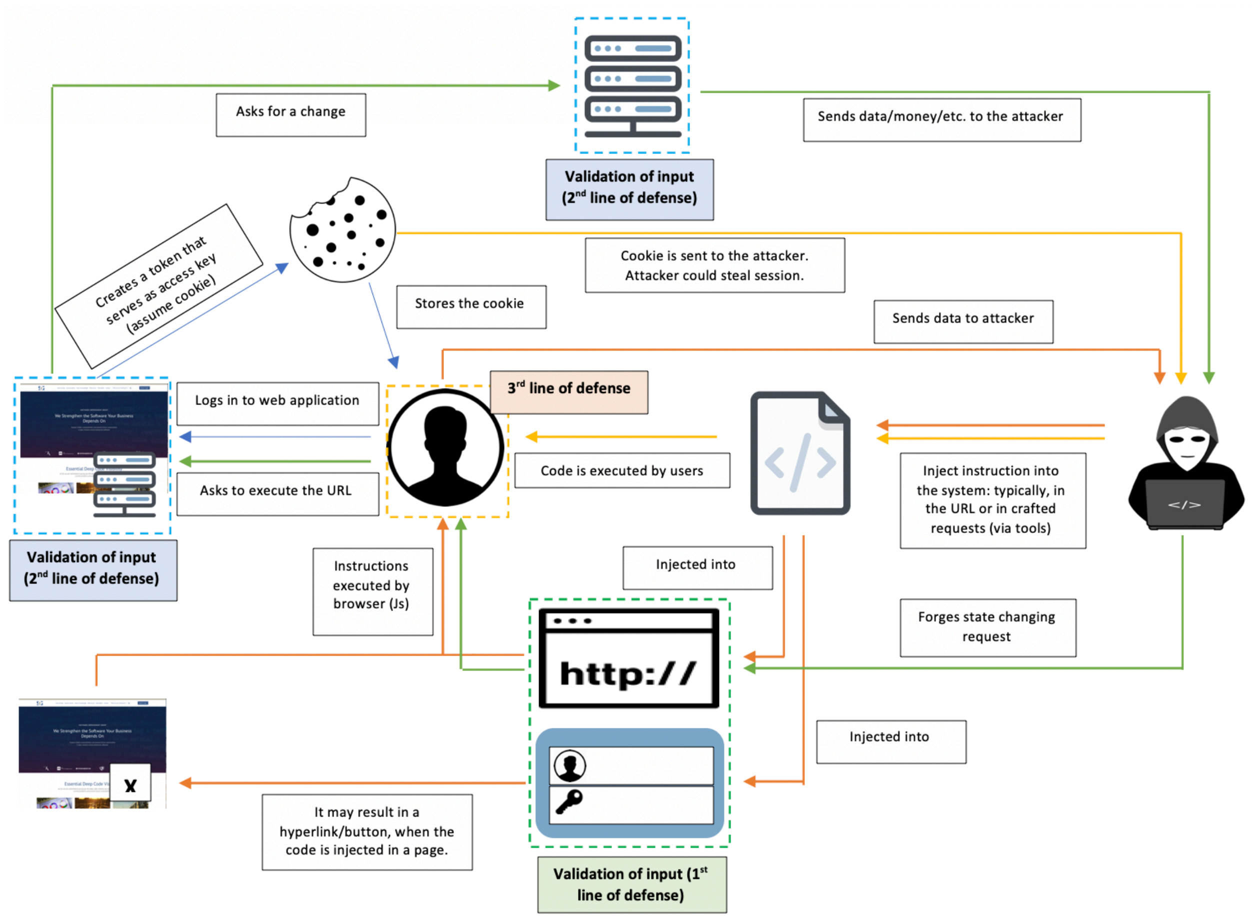
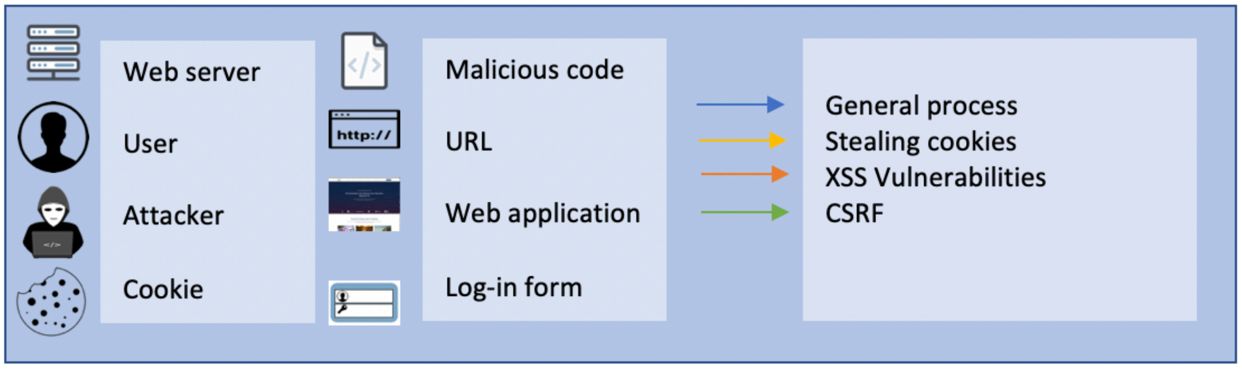

Asks for a change
Sends data/money/etc. to the attacker
Validation of input (2nd line of defense)
Creates a token that serves as access key (assume cookie)
Cookie is sent to the attacker. Attacker could steal session.
Stores the cookie
Sends data to attacker
Logs in to web application
3rd line of defense
Code is executed by users
Inject instruction into the system: typically, in the URL or in crafted requests (via tools)
Asks to execute the URL
Validation of input (2nd line of defense)
Instructions executed by browser (Js)
Injected into
Forges state changing request
http://
Injected into
x
It may result in a hyperlink/button, when the code is injected in a page.
Validation of input (1st line of defense)
</>

Fig. 6: Overview of attacks to gain access to user's data/system

<u>Explanation:</u>



4.3.3 Bypass web app authentication (via user accounts)

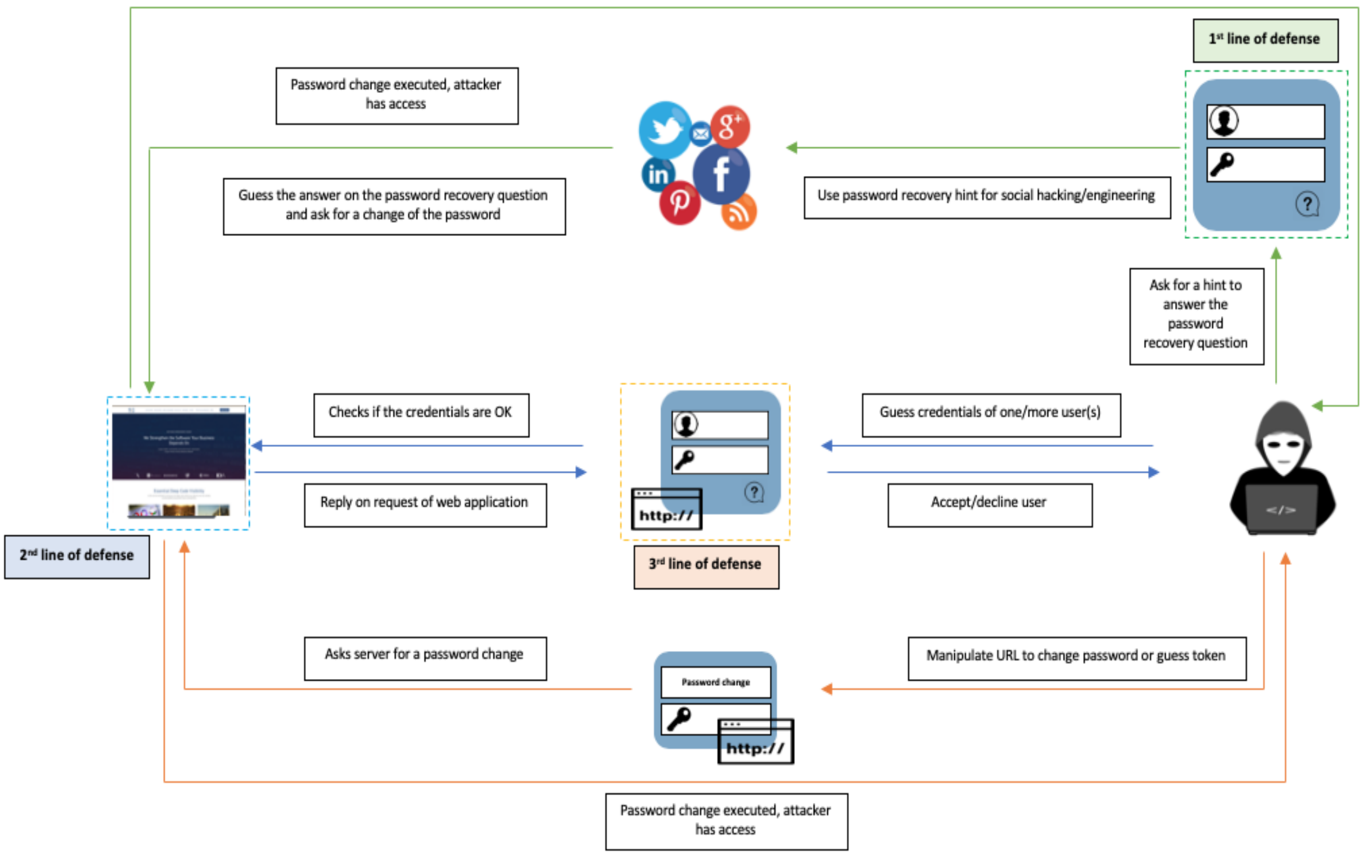
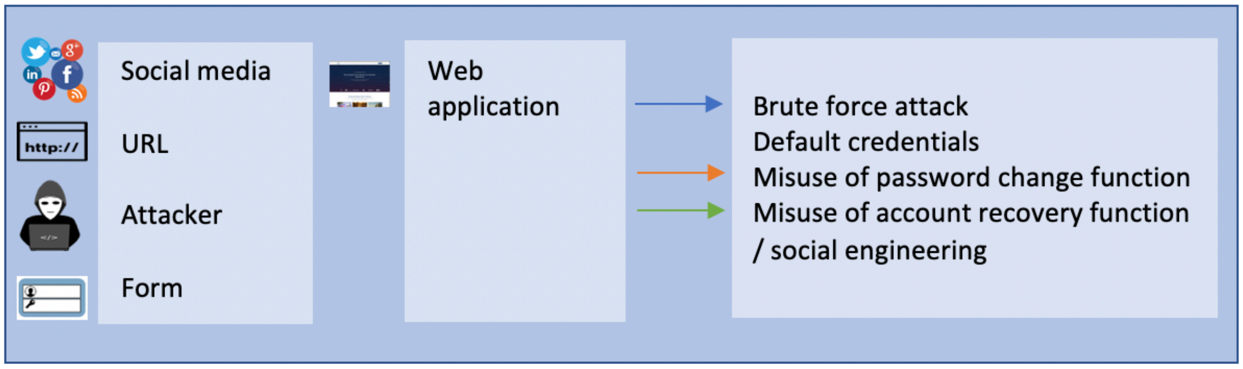

1st line of defense
Password change executed, attacker has access
Guess the answer on the password recovery question and ask for a change of the password
Use password recovery hint for social hacking/engineering
Ask for a hint to answer the password recovery question
Checks if the credentials are OK
Reply on request of web application
Guess credentials of one/more user(s)
Accept/decline user
http://
2nd line of defense
3rd line of defense
Asks server for a password change
Password change
Manipulate URL to change password or guess token
http://
Password change executed, attacker has access

Fig. 7: Overview of attacks to gain access to user accounts

<u>Explanation:</u>


Social media
http://
URL
Attacker
Form
Web application
Brute force attack
Default credentials
Misuse of password change function
Misuse of account recovery function / social engineering

4.3.4 Bypass web app authentication (via system functions)

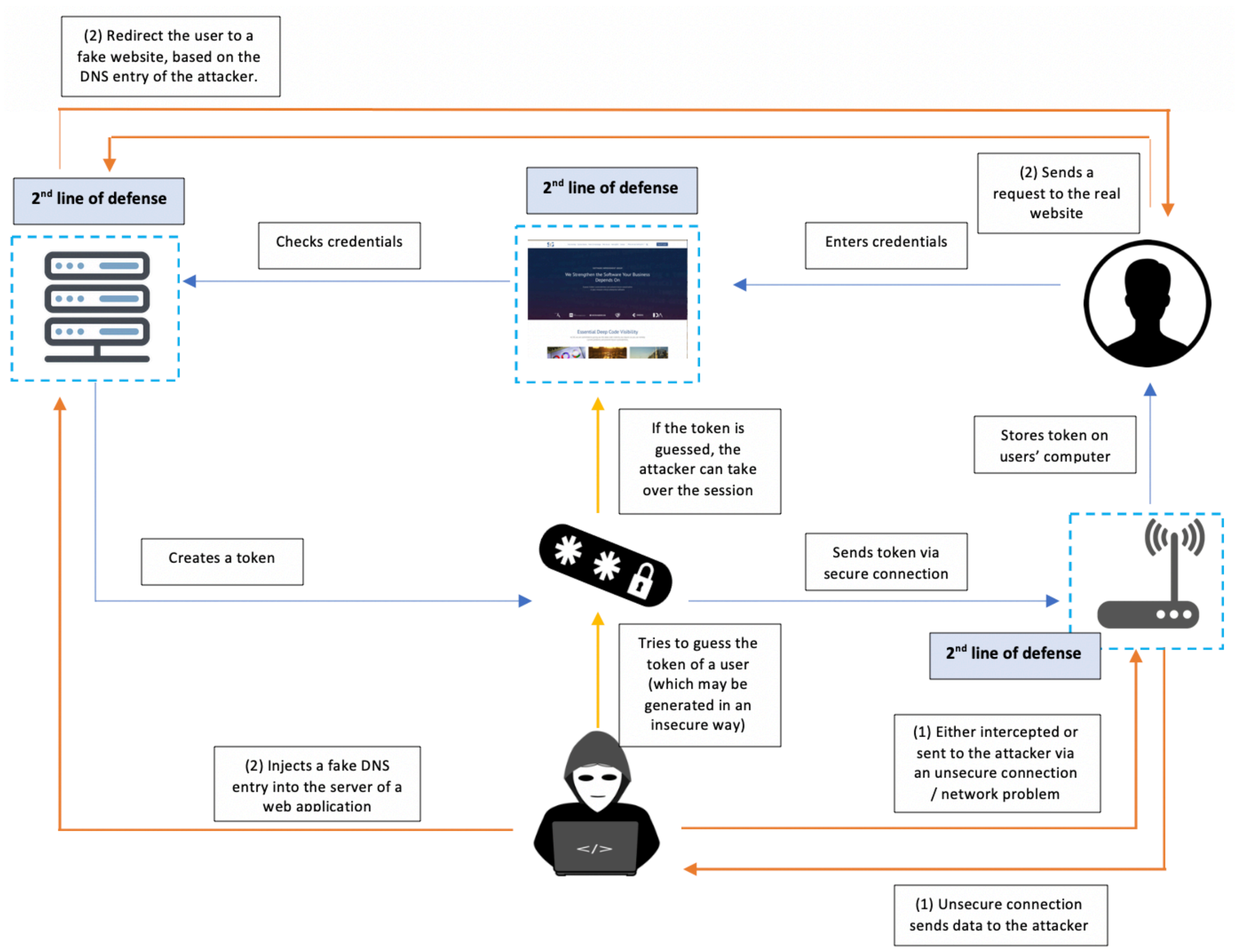
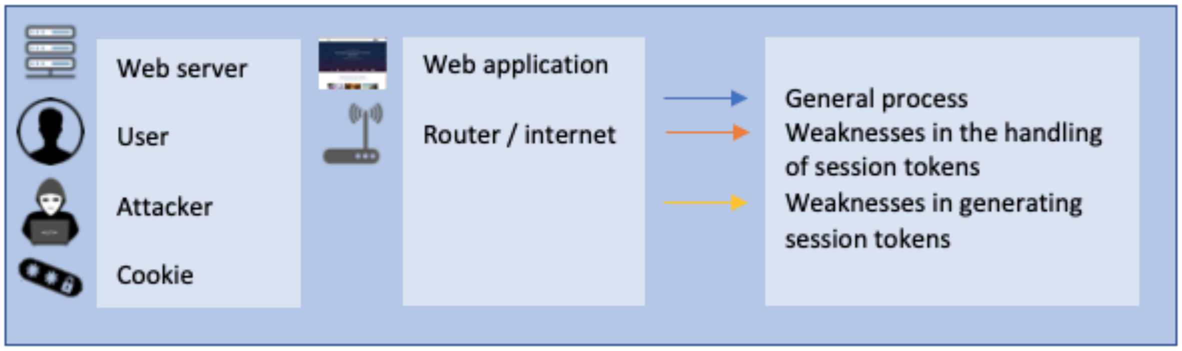

(2) Redirect the user to a fake website, based on the DNS entry of the attacker.
2nd line of defense
2nd line of defense
(2) Sends a request to the real website
Checks credentials
Enters credentials
If the token is guessed, the attacker can take over the session
Stores token on users' computer
Creates a token
Sends token via secure connection
Tries to guess the token of a user (which may be generated in an insecure way)
2nd line of defense
(1) Either intercepted or sent to the attacker via an unsecure connection / network problem
(2) Injects a fake DNS entry into the server of a web application
(1) Unsecure connection sends data to the attacker

Fig. 8: Overview of attacks to gain access to user accounts via tokens

Explanation:


Web server
User
Attacker
Cookie
Web application
Router / internet
General process
Weaknesses in the handling of session tokens
Weaknesses in generating session tokens

4.4 Level of difficulty

The level of difficulty describes the time and technical knowledge an attacker needs to misuse a vulnerability in web applications. Also, the *steps to be taken* are considered. For example, a brute force attack is quite straightforward while hijacking cookies requires different steps. For validation, several feedback moments with experts from the field were used.

It is important to understand that the level of difficulty is established from an attackers' view. In case the total score of an attack has a green color, this means that it is quite easy to execute this type of attack. The ones with a yellow or red color are more difficult. In the next part of this chapter, the level of impact based on the perspective of organizations will be defined. It is an important difference.

Name	**Time**	**Technical knowledge**	**Steps to be taken**	**Total**
Brute force attacks	3	1	1	5
Misuse of password change function	1	1	1	3
Misuse of account recovery function	2	1	2	5
Stealing cookies from a user	3	2	3	8
Weaknesses in the generation of tokens	3	3	2	8
Weaknesses in the handling of session tokens	2	3	2	7
Securing access control	2	1	1	4
SQL injections	2	2	2	6
Injecting into NoSQL	2	3	2	7
Injection into XPath	2	3	2	7
XSS Vulnerabilities	3	3	3	9
Cross-site Request Forgery (CSRF)	3	3	3	9
Default credentials	1	1	1	3

Table 1: Level of difficulty

<u>Difficulty (1 = low, 3 = high):</u>

- *Time:* the time it costs to execute an attack
- *Technical knowledge:* the necessary technical knowledge for executing an attack
- *Steps to be taken:* the number of steps that have to been taken for executing an attack

<u>Total number of points:</u>

3-5	Green
6-7	Yellow
8-9	Red

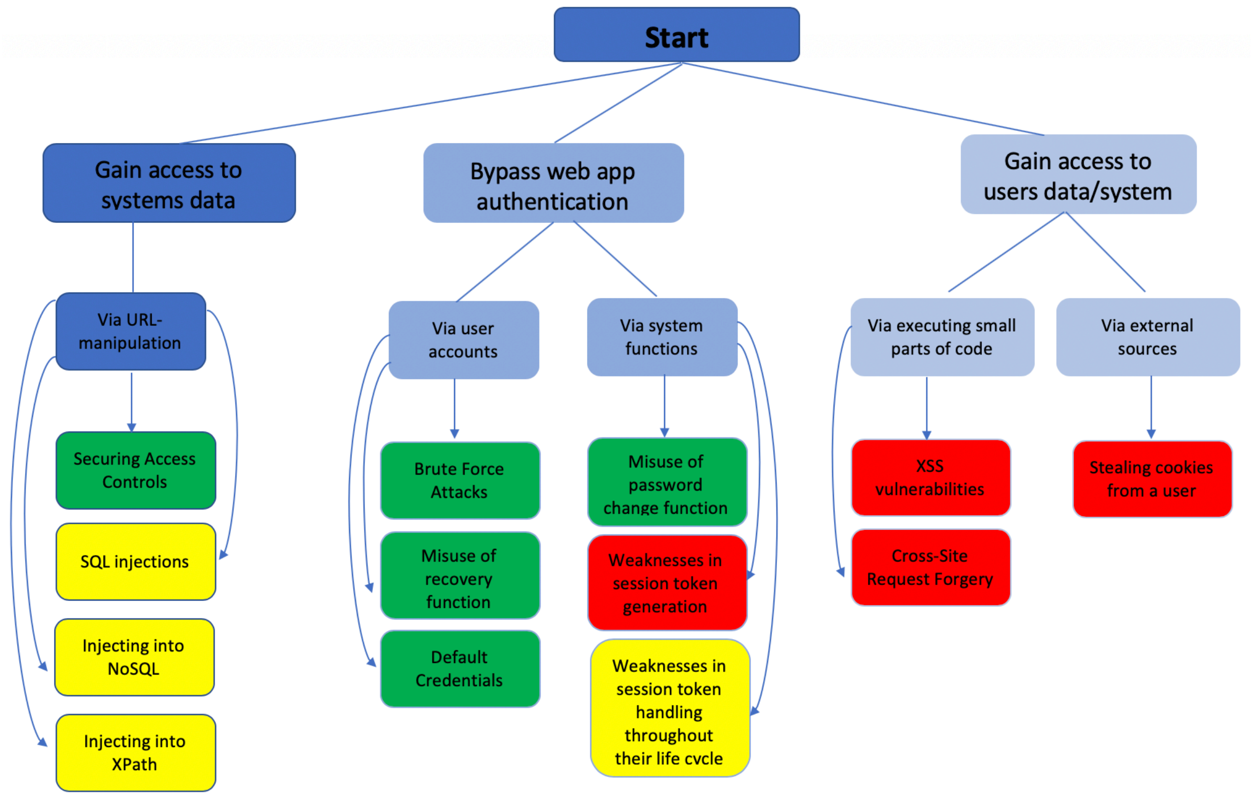

Start
Gain access to systems data
Bypass web app authentication
Gain access to users data/system
Via URL-manipulation
Via user accounts
Via system functions
Via executing small parts of code
Via external sources
Securing Access Controls
SQL injections
Injecting into NoSQL
Injecting into XPath
Brute Force Attacks
Misuse of recovery function
Default Credentials
Misuse of password change function
Weaknesses in session token generation
Weaknesses in session token handling throughout their life cycle
XSS vulnerabilities
Cross-Site Request Forgery
Stealing cookies from a user

Fig. 9: Overview of vulnerabilities and their difficulty

Explanation:

4.5 Level of impact

The level of impact describes the economic impact of misusing a vulnerability in web applications. How many users are affected, what is the value of data that is reached, what is the amount of damage on the system and what is the difficulty of reproducing an attack.

This model is based on Microsoft's DREAD-model (OpenStack, n.d.). Here DREAD stands for *damage, reproducibility, exploitability, affected users* and *discoverability*. The discoverability has been left out, because it is not as much a determining factor for impact weighting, as vulnerabilities must be assumed to be public these days. There are several libraries with lists of known vulnerabilities within frameworks, extensions and plug-ins. Thereby, discoverability has been covered by the ranking discussed in chapter 4.4 based on the attacker's perspective. As well as for the table with scores about the difficulty of attacks, feedback moments with experts have been used to discuss the way the attacks should be scored.

Here the ranking is built up from the perspective of an organization, while the table from the previous part of this chapter was built up from the attacker's perspective. In case the total number of points has a green color, it means that the impact of misusing such a vulnerability has a low impact on the organization.

Name	**Users**	**Value of data**	**Damage**	**Reproducibility**	**Total**
Brute force attacks	2	2	2	3	9
Misuse of password change function	2	2	2	3	9
Misuse of account recovery function	1	2	2	1	6
Stealing cookies from a user	1	2	2	2	7
Weaknesses in the generation of tokens	2	2	2	1	7
Weaknesses in the handling of session tokens	3	2	2	2	9
Securing access control	1	3	3	2	9
SQL injections	3	2	2	3	10
Injecting into NoSQL	3	2	2	3	10
Injection into XPath	3	2	2	3	10
XSS Vulnerabilities	2	2	3	3	10
Cross-site Request Forgery (CSRF)	1	2	3	1	7
Default credentials	2	2	2	3	9

Table 2: Level of impact

Impact (1 = low, 3 = high):

- *Affected users:* number of accounts involved/impact on users (indication*)
- *Damage (data):* value of data reached (considering an average web app)
- *Damage (systems):* potential damage to systems/accounts
- *Exploitability/reproducibility: the* easiness of reproducing attacks

Total # points:

4-6	Green
7-9	Yellow
10-12	Red

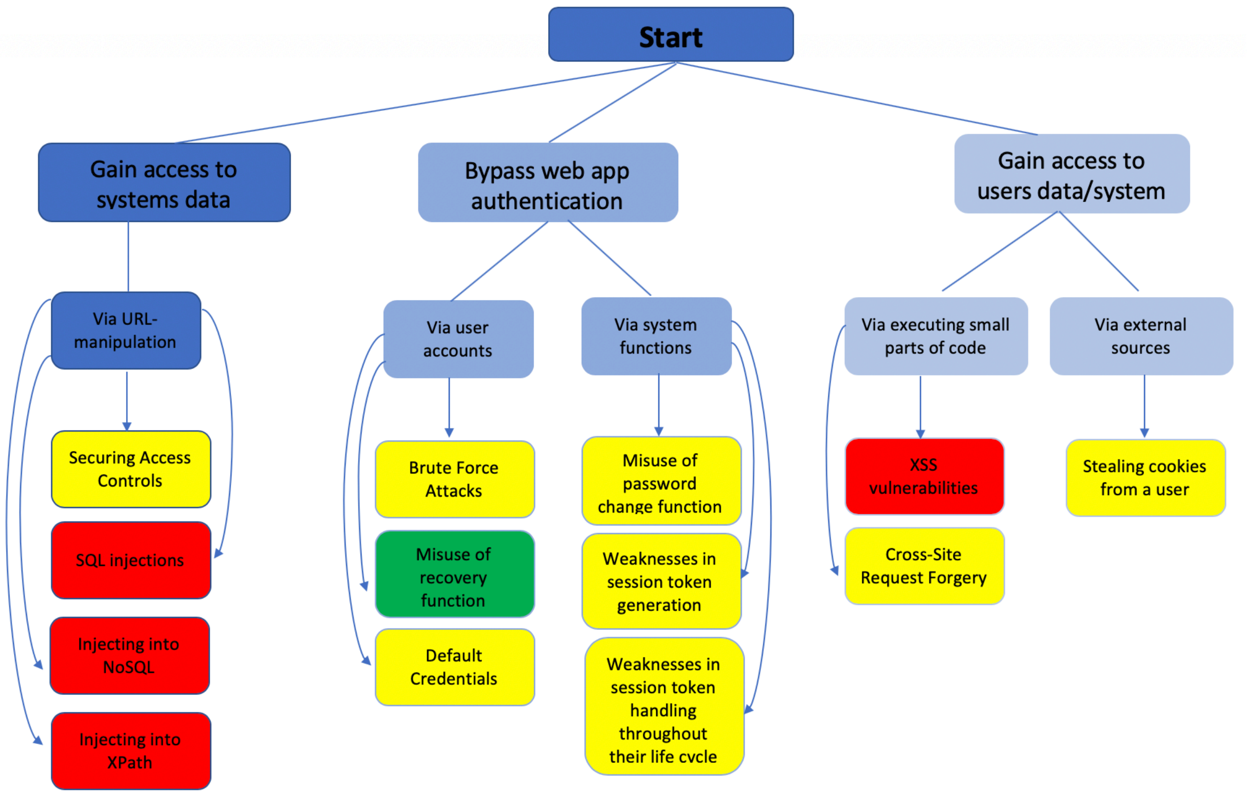
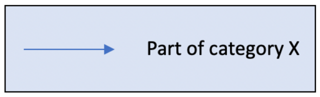

Start
Gain access to systems data
Bypass web app authentication
Gain access to users data/system
Via URL-manipulation
Via user accounts
Via system functions
Via executing small parts of code
Via external sources
Securing Access Controls
SQL injections
Injecting into NoSQL
Injecting into XPath
Brute Force Attacks
Misuse of recovery function
Default Credentials
Misuse of password change function
Weaknesses in session token generation
Weaknesses in session token handling throughout their life cycle
XSS vulnerabilities
Cross-Site Request Forgery
Stealing cookies from a user

Fig. 10: Overview of vulnerabilities and their impact

Explanation:



4.6 Recap

In this chapter an overview of the most occurring vulnerabilities in web applications was showed. For each of those vulnerabilities the difficulty for attackers to use them has been identified. Based on this several activity flows have been created, which helps in checking if none of the most occurring vulnerabilities is missing. In the last part of this chapter it has been tried to rank those vulnerabilities based on their economic impact and difficulty.

5. Overview of defense methods

The aim of this chapter is summing up different defense methods, that helps developers in protecting their web application against the identified vulnerabilities. Defense methods are sorted by vulnerabilities. As mentioned earlier, some defense methods can be used for protecting web applications against different vulnerabilities. This will become clear in the next chapters.

As well as by identifying possible cyber vulnerabilities in web applications, for creating this overview a combination of feedback moments with experts from the field, online sources and OWASP libraries has been used.

5.1 Brute Force Attacks

There are several defense methods that can help by preventing a web application for brute force attacks. Those defense methods can be categorized based on three measures:

- Adding an extra layer before entering an account;
- Using filtering based on usage behavior;
- Making guessing passwords more difficult.

5.1.1 Adding an extra layer before entering an account

CAPTCHA is one type of defense that will help against brute force attacks. While using this type of defense, an attacker has to take over a code before the system will check the combination of the username and password. Besides those CAPTCHA variants with a code, there are different other possibilities to prevent web applications for automatically executed brute force attacks by using CAPTCHA (Athanasopoulos & Antonatos, 2006). This defense method works well against automatically executed brute force attacks.

Also, multi-factor authentication will help web applications by preventing themselves for brute force attacks. In case the password is guessed right by an attacker, a second security layer will prevent that the attacker will get access to the application. There are several types of multi-factor authentication. For example: a text message on a mobile phone or a push notification via an app.

5.1.2 Using filtering based on usage behavior

Disabling accounts will help slowing down an attacker while (automatically) guessing an account's password. It means that an account will be deactivated for a certain period, when a wrong password is entered for more than three times, for example. Here, it can also be interesting to check the IP address from where a user tries to enter the web application. In case the IP address is used for the first time, an account can be disabled after two or three attempts. Does the web application recognize the IP address from earlier attempts in the past? In that case accounts should be disabled after four or five attempts, for example.

To prevent the web application for serious damage during brute force attacks, a developer should make a distinction between sensitive and non-sensitive accounts. In a way, that the usernames of for example FTP-users should never be used for front-end user accounts.

5.1.3 Making guessing passwords more difficult

A proper policy for the use of password topologies and password lengths/complexity is another defense method, to help developers preventing their web application against brute force attacks. A high password complexity means that a password should exist of at least ten characters, with at least one uppercase and lowercase character, a digit and a special character (Sönmez Turan, Barker, Burr, & Chen, 2010).

5.2 Misuse of password change function

Sometimes applications are using global cascading style sheets, which brings important risks with it. With a global style sheet, for attackers it could be possible to find out which roles are used in the backend of a web application. This information makes it easier to attack applications. Preventing a web application for such a information leakage is not the only reason developers should not work with a global cascading style sheet. Besides information about roles in a web application, an attacker could find out which features are included in a web application. For example, .profileSettings, .changePassword, .oldPassword, .newPassword, etc. (Wium Lie, 2005)

Furthermore, the password change function should not directly be accessible. The function should only be reachable for authenticated users in a web application. In some cases, this function is reachable by entering the direct URL to this function bypassing authentication.

5.3 Misuse of account recovery function

To prevent the account recovery function from being misused, password recovery questions should be safe. In such a way, that they should be memorable (easy to remember for users), consistent (answers should not change over time), nearly universal (usable for a wide audience) and safe (not easy to guess). As social hacking has become more popular, questions asking for family members or places of residence are not safe. (OWASP, n.d.-e)

A second defense method against the misuse of account recovery functions could be sending a token over side-channels. This can be compared with multi-factor authentication, as a token is sent over a side-channel as a mobile phone for example. With this token a user can unlock his account. Keep in mind, that this token should have a certain validity. In this way, you can prevent that an attacker intercepts this token by attacking someone’s email address (assuming the token is sent by email). (Bonneau, Bursztein, Caron, Jackson, & Williamson, 2015)

5.4 Cookies

Intercepting cookies can help attackers to get access to someone’s account. To prevent cookies from being stolen, using a transport layer security (TLS) could be the first step. Securing the transport layer means that a web application should always use an encrypted (HTTPS) connection. This secured connection should not only be used in the authentication process of users, but for the whole session.

There is a best practice for setting up and using HTTPS connections (OWASP, n.d.-h):

- The web application should never switch between HTTP and HTTPS connections, after the session ID (cookie) is generated. This works in two ways; from HTTP to HTTPS and vice versa.

Besides transport layer security also using a secure socket layer (SSL) could be a possibility. Both terms are often used interchangeably, although there is a difference between them. However, TLS version 1.0 can be compared with version 3.1 of SSL. In most cases, using commonly recommended TLS versions is regarded as safer than SSL (KPN Internedservices, 2017). Keep in mind that TLS version 1.0 is not the most recent version, at the time this research is done.

As you want to prevent a users' session from being stolen, session management should be an important part of the defense of web applications. Enforce a proper session management by, for example, introducing a policy for it. Such a policy could exist of requirements as (OWASP, n.d.-h):

- The session ID should not be descriptive, nor offer unnecessary details. It should be meaningless, to prevent the session IDs for sharing useful information with attackers.
- The session ID should be long enough to prevent brute force attacks.
- The session ID should not be predictable, based on other session IDs. An attacker should not be able to guess session ID of user X, after finding out the session ID of user Y.

Session management is not something new. Developers can use different frameworks, of which .NET, PHP, J2EE and ASP are examples. Building an own session management mechanism from scratch is discouraged, as it will bring risks with it. Also, it is important to always use the latest version of those frameworks.

5.5 Weaknesses in the generation of session tokens

As well as for preventing cookies from being stolen, using a transport layer security (SSL) or transport layer protection (TLS) will help developers by preventing session tokens from being stolen. The same holds for a correct session management, which can give guidelines to the way session tokens should be generated in web applications.

To add an extra layer of security on top of those three possibilities, developers could consider using encryption key storage guidelines (OWASP, n.d.-f). Where session management focuses on generation session tokens, those storage guidelines are focused on the way session tokens are stored. Developers can think of guidelines as:

- It is important that developers know where cryptographic keys are stored, and which memory devices are used for this. Preferably use a cryptographic vault to ensure that this data cannot be accessed by attackers.
- It is important that keys are never be stored as plain text. On the one hand to prevent them from being stolen, and on the other hand to prevent them from being used by attackers (in case they unfortunately are stolen)

- Ensure that keys have integrity protections applied while in storage (consider dual purpose algorithms that support encryption and Message Code Authentication (MAC)). (OWASP, n.d.-f)
- It is important that code of the web application on a standard level should never read or use the (cryptographic) keys. Also, operations on those keys should always happen within the used vault.

5.6 Weaknesses in the handling of session tokens throughout their life cycle

Weaknesses in the handling of session tokens throughout their life cycle could mean that those tokens can be stolen by attackers after generating them. There are several possibilities to prevent this, for example by encryption. Encryption is used in two ways within the back-end of web applications: symmetric and asymmetric. The difference between both types of encryption is in the number of keys used. In case of symmetric encryption only one key is used for encryption, as well as for decryption. With asymmetric encryption two keys are used; a public- and private key. Both keys are related to each other and used for encryption and decryption of those sessions' tokens.

A second way of preventing session tokens from being stolen after generating them is the authentication of end devices. There are different ways to authenticate those devices, for example by using pre-shared symmetric keys, trusted certificates and trust anchors. In case of web applications, the authentication of end devices is most often combined with user authentication (Rouse, 2015). Especially because of the increasing popularity of machine-to-machine (M2M) communications, end device authentication has become more important.

To come to this end device authentication, a message authentication code (MAC) can be used. A specific type of such a MAC is HMAC, which stands for keyed-hashing for message authentication. It is also known as hash-based authentication code. With this type of technology, first a secret key is distributed to the intended end devices. Now a situation arises where only the end device and the source know the HMAC key. It provides both parties the certainty, that a file or message is delivered to the correct end device. ("HMAC, Keyed-Hashing for Message Authentication", n.d.)

There are different cryptographic hash functions that can be used for HMAC, for example MD5 and SHA-1. It is important to understand that the strength of HMAC depends on the properties of the underlying hash function ("HMAC, Keyed-Hashing for Message Authentication", n.d.). Thereby, some hash functions (for example, MD5) create hashes which can be encoded by attackers. It makes that those hashes are not fully waterproof.

5.7 Securing Access Controls

While developing web applications, for developers it will be important to decide which users can use what types of functions. In most cases, web applications exist of secured and protected parts for certain user groups. There are different ways to secure access control, of which role based access control (RBAC) is an example.

Here individual roles and responsibilities for users are used, to prevent them from accessing restricted areas of web applications. Roles are defined based on the organizational structure in relation to the security policy. As there are many models for role based access control, it is quite easy to implement this. (Li, Liu, Wei, Liu, & Liu, 2010)

On top of this RBAC-layer in web applications, a second layer can be implemented. This second layer is called discretionary access control (DAC). Here users can get access to files in web applications based on their identity and/or their membership in certain groups. In most cases the owner of the resource is able to set the permissions for those groups. Every object has an owner, that defines who can get access to that object. (OWASP, n.d.-a)

5.7.1 Differences between RBAC and DAC

Although RBAC and DAC looks like they have the same functionality, there are some differences. This does not mean, that developers have to choose between both. RBAC is based on a list of roles, to which each user of the web application is assigned. Each user has at least one role but can get more. While using a web application, the system will check if a user is part of a certain role. If this is the case, access to a page or file will be given. In other cases, access is denied.

With DAC an access control list (ACL) is used. This list holds for just one resource within the web application. Resources may be for example files, database tables or registry keys. A user can be part of this ACL, which give the user access to the resource. For developers it will cost lots of time to generate those access control lists. A main advantage of RBAC over DAC is thus the ease of management (Stack Exchange, 2010). DAC can be seen as operating systems or UNIX's. It is to some extent a type of individual whitelisting per resource.

5.7.2 Role of Mandatory Access Control

Besides role based access control and discretionary access control, mandatory access control (MAC) can be a possibility. Do not confuse mandatory access control with message authentication code, what is also shortened to MAC.

A security policy based on mandatory access control is, in contrast to DAC, centrally controlled with roles. Here different levels of access are used, to which individual users are linked. Those users have access to all the resources that are higher in the security structure than his access level.

As well as with the difference between DAC and RBAC, DAC is more useful for granting access to specific resources than MAC. While MAC can be timesaving for securing access in web applications with large groups of users. An example of the use of MAC is Windows, where administrators have the possibility to see all resources. For guests and other users this is not possible. (Joan, 2018)

5.8 SQL injections

Earlier in this research a brief introduction to injections has been given, of which SQL injections are the most common ones. Although it can be hard to prevent web applications from being attacked by such an injection, there are ways to minimize this risk. Defense methods described in this part are not only useful for protecting web applications against SQL injections. They can also be used against NoSQL-, XPath- and LDAP injections.

5.8.1 White list input validation

White list input validation could be used as the first line in the defense of a web application. An alternative for this type of defense is black list input validation, of which the operation is exactly the opposite. Input validation is the practice of limiting the data that is processed by a web application to the subset that the system knows it can handle (Basirico, n.d.).

Take for example a phone number, which should probably be stored as a *varchar* in the database of web applications. Developers know that a phone number has a numeric format and exists of a certain number of characters between zero and nine. In case a phone number is submitted as *"xyz#123<>"* it should be rejected by the web application. Although this input can be stored in the database, it does not meet the requirements.

It is important to understand that using white- or black list input validation without prepared statements will not be enough. The chance of missing *bad data* which leads to damage on a server or web application is quite high. Especially, because attackers are finding new ways of bypassing this input validation. However, it can be an additional layer on top of the prepared statements and encoding function.

5.8.2 Encoding function for escaping variables

While executing an injection, attackers try to use disallowed variables in an input field within the web application. When the system does not react in a correct way on this input, the web application will possibly spit out secret information from database tables for example. In case an attacker succeeds in bypassing the input validation, using an encoding function helps developers by escaping those disallowed variables. Here it is important to make a distinction between escaping variables in a username input field and, for example, a search field. (OWASP, n.d.-i)

Characters that should be escaped by the encoding function are: * ( ) . & - _ [ ] ` ~ | @ $ % ^ ? : { } ! ‘. When executing variables with those characters, the system could react in the wrong way. For developers creating an encoding function is not necessary in most cases, as they are part of different frameworks. For example, .NET has an AntiXSS encoder class to convert unsafe characters. Alternatively, developers can use a framework for this. LINQPad is an example of such a framework (LINQPad, n.d.). These frameworks automatically escape variables while building a search query.

5.8.3 Prepared statements

Besides using input validation and an escaping function, prepared statements can be used to prevent web applications for injections. Those prepared statements are also known as parameterized statements or stored procedures (W3Schools, n.d.). Those can be seen as a type of template for queries, executed by the SQL-, NoSQL-, LDAP- or XPath-system. In those templates question marks are used for placeholders, where actual values can be inserted. The prepared statements are stored, until the parameters are filled in. In that case the query will be executed by the web application. For attackers it will not be possible to adjust or delete those statements.

Important here is the fact that those queries are generated beforehand, instead of during the use of an input field of a web application. In case there is a possibility to insert a complete query inside an input field, attackers could have access to information within database tables of those applications. Also, attackers may have the possibility to adjust or even delete complete tables with their queries.

The reason why those prepared statements are interesting for developers is in the order the query and data are sent to the server. Because those queries are prepared and stored beforehand, the data will always be sent afterwards and separately from those statements. This means that the input of an attacker can never be interpreted as a SQL-, NoSQL-, XPath- or LDAP query by the web application. Queries from an attacker will be handled as data. (ProgrammerInterview.com, 2015)

5.9 XSS Vulnerabilities

As well as injections in web applications, XSS attacks are popular. This type of attack exploits the browsers' trust in content it receives from the server. Although trust in the content does not often causes problems, an attacker may misuse this trust by manipulating content. Here the browser executes small parts of code, even when it is not coming from the server the website is connected to.

There are a lot of different ways to execute such an attack, which makes it hard to protect a web application for it. As well as for injections, there are several defense methods for possible XSS vulnerabilities.

5.9.1 Content Security Policy

Generating a content security policy may be a first step in the process. The primary objective of a content security policy (Mozilla, 2019), often abbreviated as CSP, is to mitigate and report XSS attacks. Keep in mind that a CSP could be seen as a first layer of defense. However, this will not be enough to mitigate every XSS vulnerability in the code of web applications. This can be explained by the fact that CSP will not be supported by every web browser, although the most popular ones do support it.

The idea behind a CSP is telling the browser of which domains content should be trusted. If a script is loaded from another domain, the idea behind an XSS attack, the script will not be executed. In case the script is part of a white listed domain, it will. In case plug-ins are using scripts that are stored on other domains, those domains should be added to the CSP of the web application. This can be time consuming for developers. (OWASP, n.d.-b)

For simple HTML websites it can be interesting to disallow scripts from every domain. This makes executing an XSS attack almost impossible. However, it limits developers in adding new functions to a web application.

5.9.2 White list input validation

White list input validation has been mentioned earlier as a defense method against injections. This type of defense also helps developers in preventing their web applications for XSS attacks. As well as with the prevention for injections, the objective of using white list input validation is limiting the data that is processed (Basirico, n.d.).

Developers give their web application instructions the way different types of data should be handled. In case the input of a user does not match the instructions, the input should be rejected. Earlier, the example of a phone number with deviant characters has been given. By giving the web application instructions about the types of data that should be handled, the chance of an XSS attack can be limited.

As mentioned before, the chance is quite high that some *bad data* may be missed while using white list input validation. Attackers will find new ways to bypass this line of defense. This is why white list input validation should be combined with other types of defense methods.

5.9.3 Escaping

Earlier, escaping certain input variables was described as a defense method for preventing injections. This type of defense is also interesting for preventing web applications for XSS attacks. An XSS attack can be executed by entering a small piece of HTML code into input fields. This code may instruct the server to execute externally located pieces of code, for example JavaScript. An example of such an input can be: *"Hello world, look at <script src="https://mywebsite.com/attack.js"></script>"*. (OWASP, n.d.-c)

For this example, the characters "*<*" and "*>*" should be escaped by the system, to prevent it for executing the JavaScript-code on *mywebsite.com*. With an escaping function a set of characters will be converted to for example "*&amp*", which makes that the attack will no longer be executed by the server as the input is not seen as a piece of code anymore. Other characters that should be escaped: *", ', &*.

Escaping can be compared with using an encoding function. In case of an encoding function the encoded instructions are translated back to its original layout (HTML, JavaScript and comparable languages), while an escaping function will not translate it back.

5.9.4 Sanitizer

To a certain extent using a sanitizer can be compared with the use of an escaping function. In both cases, the system is checking the input of a user for disallowed characters. However, a sanitizing function will not transform every HTML character in the input. Something the escaping function does. Using a sanitizer is thus interesting for websites that allow users to use HTML tags for editing their input on a forum or for example as a review.

A sanitizing function checks the input for HTML tags, that are not on the white list. This white list has been determined in advance and is part of the function code. There are several lists with allowed HTML tags for this input validation published on the internet (HTML5Lib, 2017), as well as code for using this type of function (Weber, n.d.).

Sanitizing functions look like white list input validation, which has been regarded as a defense method for injections. There is a difference: a sanitizing function is checking the input for certain HTML tags, while white list input validation is checking if the input meets different types of requirements. For example, the length of a phone number or the type of characters that is submitted by users. (OWASP, n.d.-c)

5.9.5 Output encoding

The use of an output encoding function does not only help developers preventing their web applications against XSS attacks. The use of such a function was already recommended for defending applications against injections. Attackers may use an input to hide server instructions for executing malicious code. Output encoding helps web applications preventing the server for executing those instructions by encoding all variable output before returning it to the end user. It helps cleaning up the input of an attacker.

Encoding means that for example the HTML markup is converted into so-called *entities*. So, "*<script>"* will be converted to "*&lt;script&gt;*". The idea behind this encoding is that the server will not run any script inserted by attackers. However, the HTML markup will be part of the web page. This type of encoding can also be used for JavaScript or other languages.

First, the server will convert the HTML markup to entities that cannot be run. Then the browser will download the encoded web page and convert it back to the original HTML markup. For end users this process is not visible. However, the fact that the browser did not run the script makes that XSS attacks on end users were not possible. (Ladkani, 2013)

5.9.6 Using JS-frameworks

One way to combine several defense methods against the misuse of XSS vulnerabilities, is using an existing JavaScript framework. The most popular ones are *Angular*, *ReactJS* and *VueJS* (Visser, 2017). Those frameworks are provided with different defense methods against the most occurring types of cyber attacks. In most cases, an escaping- and sanitizing function are part of them. In case developers choose for a less popular JS framework, they should check them on the integrated defense mechanisms. Although there are several defense methods implemented in those frameworks, developers need to take some steps before they are integrated well. Thereby, it can be smart to implement some extra defense methods.

One of the other reasons to choose for those frameworks, is the possibility to use libraries from other developers. In this way, web applications can be developed much faster and in most cases in a safer way. However, not all of those libraries are well developed. This makes that those libraries can bring cyber vulnerabilities with them (Sargent, 2017). For developers it is important to check those libraries for vulnerabilities, or only use popular and as safe marked libraries.

Also, it is important to use the newest version of those frameworks. As well as web applications, cyber attacks are being improved rapidly. Thereby, the different types of cyber attacks are growing. Without the newest version of those JS frameworks, web applications could be vulnerable for different types of attacks.

5.10 Cross-site request forgery

A cross-site scripting attack exploits the trust of a user in a particular web application. The way a cross-site request forgery, often abbreviated as CSRF, attack is executed can be compared with this. Instead of exploiting someone's trust in a web application, the attacker exploits the application's trust in a user's web browser. This is done in such a way, that the user is tricked into submitting a request in the web application they did not intend.

5.10.1 Token-based mitigation

Token-based mitigation is one of the most popular defense methods against cross-site request forgery attacks. This defense method can be split into state and stateless operations. In the first case, a synchronizer token pattern is used. In the latter one, developers choose for an encryption based token pattern. (OWASP, n.d.-d)

5.10.1.1 Synchronizer token pattern

To include the synchronizer token pattern defense method, every state changing operation within a web application requires a secure CSRF token. For example, CSRF tokens are used in forms or AJAX calls. Those tokens should be large and random values that are unique for every user session. This makes it almost impossible for attackers to guess the token value. One way of generating those tokens, is by using a CSPRNG: cryptographically secure pseudo-random number generator (OWASP, n.d.-d).

Thereafter a CSRF token can be added to the state changing operation through headers, which is the safest option. An alternative for using headers is adding tokens to hidden fields. Here it is important to prevent any leakage of those tokens. In case a token is part of the URL or server logs, attackers may hijack sessions and misuse sessions wherein the CSRF token is used.

In web applications where tokens are part of the header, it will be much more difficult to execute a CSRF attack. An XMLHttpRequest can be used to set custom headers, but it will not be possible to forge a POST request. Something that is necessary to execute the CSRF attack. In every session, the existence and validity of a user's token is verified.

The synchronizer token pattern defense is in most cases part of a framework. It is possible to add some extra CSRF defenses by adding external components to a web application.

5.10.1.2 Encryption based token pattern

Encryption based token pattern defense methods can be used as an alternative for synchronizer token pattern defenses. This type of defense is recommendable for stateless web applications. Here a CSRF token is generated which includes the session ID of a user and a timestamp. This timestamp is necessary to prevent the web application for replay attacks. (OWASP, n.d.-d)

The generated token will be embedded in a hidden field or header, which depends on the type of action that is done. Hidden fields are often used for forms, while headers are used in case of AJAX requests. When a user requests something from the server, the server tries to decrypt the included token. Thereafter the session ID and timestamp are checked, and the request is executed. In case a token cannot be decrypted, the request is rejected.

5.10.1.3 Double submit cookie

Using a double submit cookie can be seen as a third defense against CSRF attacks. As well as encryption based token pattern defenses, double submit cookies are most interesting for stateless web applications. With this defense method, a pseudo-random value is generated and returned to the end user as a cookie. This cookie is separated from the session ID of the user.

During the first request a user does, a CSRF cookie is added to the response. Then, for every new request the server checks if the requested resource must be CSRF protected. If this is the case, the server checks if there is any discrepancy between the CSRF cookie and the CSRF header. (OWASP, n.d.-d)

5.10.1.4 HMAC based token pattern

A fourth defense is the HMAC based token pattern. This defense method can be compared with the encryption based token pattern defenses. However, there are some minor differences between them. First of all, a HMAC function is used to generate new tokens. Such an algorithm, for example SHA256, can be much stronger than the way tokens are generated in the encryption based token pattern defenses. As well as for the encryption based tokens, the timestamp is added to the HMAC tokens. (OWASP, n.d.-d)

After generating the token, it is included in a hidden field of the header. When the end user is requesting something, the same key for generating the token is used to decrypt it. In case the decryption does not succeed, the request is rejected.

5.10.2 Other CSRF defenses

Using one of the described types of token-based mitigation can be seen as the basis of a good defense against CSRF attacks. In some cases, parts of web applications need an extra layer of defense. For example, while executing a money transfer or changing a password. In such cases, developers can add one of the other CSRF defense methods.

Re-authentication is an example of such an extra layer of defense. Here, the web application checks if the user who initially started a session also requested a money transfer or a password change. Besides asking a user to enter his password again, the user can be asked to fill in another type of token. For example, a code sent via SMS or email (IBM, n.d.).

Comparable with a re-authentication is using a CAPTCHA defense layer. The web application uses CAPTCHA to check whether the user is a real person or a bot. A user is marked as a real person when the CAPTCHA challenge is correctly solved. If not, the user may try a new challenge to get access to restricted areas within the web application.

Although CAPTCHA can help developers in preventing CSRF attacks, it is not a real solution. Every CAPTCHA challenge has his own ID, which is coupled with a solution. An attacker can find out which challenge ID is coupled with which solution and add them to the CSRF request.

Now, the CAPTCHA provider confirms the user is a real person and should get access to restricted areas while the end user should not (Detectify, 2018).

This problem can be solved by CAPTCHA providers, by deleting challenges that have been solved by users. In this case, a challenge ID can only be used once which makes it impossible for attackers to get access to restricted areas by adding them to the CSRF request.

5.11 Default credentials

To prevent a web application from using default credentials, there are several steps that can be taken. Earlier it has been described that there should be a distinction between sensitive and non-sensitive accounts. In case an attacker get access to a non-sensitive account, it should not have the same credentials as for example a sensitive FTP-account.

Also, it has been described that web applications should force users to use complex passwords with a length of at least ten characters that are changed every x months. While forcing users to change their password periodically, different password topologies can be used.

Another way to prevent attacks due to default credentials is using unique IDs for users. For example, in most cases *admin* has ID 1. Changing this ID to a random number, will make it more difficult for attackers to gain access to this or other accounts.

Fourth, it is important to decide which types of usernames can be used by users. For example, it is recommended to avoid the use of email addresses as usernames. First of all, because of user's privacy. Most email addresses reveal the name of users, as people combine their name and surname in their email address.

Another reason to avoid using email addresses as usernames is the fact that it helps attackers with a user harvesting attack. They can use the "*forgot my password"*-page to find out if a user exists in the user database, based on their email address (Stack Overflow, 2009).

5.12 General defense methods

So far, this chapter shows different defense methods for specific types of attacks. There also are several defense layers, that affect the exploitation of all types of vulnerabilities in web applications.

5.12.1 Application logging

Most organizations are using logging mechanisms, while implementing a new application in their infrastructure. However, this kind of logging is focused on network devices, operating systems, web servers, mail servers and database servers. Specifically logging the web application is not always used, while this can prevent an application for attacks in an early stage. It helps developers identifying security incidents, establishing baselines, providing information about problems (possible vulnerabilities) and unusual conditions.

5.12.2 Password storage

First of all, passwords should not be stored. It is justified to only store derivatives. In this way it is not possible for attackers to steal those unencrypted passwords from the web application's database. There are different hashing functions available, of which *Argon2, PBKDF2, Catena, Lyra2, Makwa, Scrypt* and *Bcrypt* are examples. Of those functions, Argon2 has been the winner of the password hashing competition in 2015 (Hatzivasilis, 2017). Argon2 optimize the resistance against GPU cracking attacks and side-channel attacks (Wikipedia, 2019).

5.12.3 File upload malicious code prevention

Most common file types that are used while uploading documents to a web application, are Microsoft Office-, Adobe PDF-documents and images. Those files could have been infected before the moment they are uploaded. The web application thus should scan the files, to detect possible malicious code in the file. (OWASP, n.d.-g)

5.12.4 Third-party JavaScript management

A good basis for safe web applications could be the use an existing JavaScript framework, of which examples were given in an earlier part of this chapter. While developing web applications with those frameworks, third-party code may be used. For example, adding an extra feature to the web application which is not part of the basic framework developers use. This brings several risks with it. Three major risks can be determined:

- The loss of control over changes to the web application by the third-party code. Data flows can be intercepted and interfaces could for example be changed.
- Arbitrary code could be executed on the system of a client. This means that a session may be hijacked or that a piece of code can users force to execute some actions.
- Sensitive information can be disclosed directly to the third party, as the user's browser will communicate with this third-party browser. For example, a cookie can be generated by the third party. When a user visits another web application which is working with this third party, the cookie will be recognized.

Based on those three main risks of using third-party code, it becomes clear that it is important to analyze this code before using it. (OWASP, n.d.-j)

5.13 Implementing defense methods

In this part, an indication of the attractiveness of implementing different defense methods will be given. Currently there is no model for categorizing defense methods based on their difficulty. First, the way the model for this research has been designed will be described, thereafter the different defense methods will be categorized.

5.13.1 Model to categorize defense methods

There are different ways to indicate how difficult it will be to implement defense methods within the code of web applications. The easier the implementation of those defense methods, the more attractive it can be to choose those methods. However, developers need to keep in mind that there are more factors. For example: does a defense method only helps in preventing a web application for a specific attack or does it prevent different types of attacks?

To get a reliable indication of the attractiveness of different defense methods, five different factors have been used:

- *Framework*: if and how often is a defense method part of an existing (JavaScript) framework? In case a defense method is part of a framework, developers do not always have to add extra defense methods for specific vulnerabilities.
- *Plug & Play*: how difficult is it to implement individual defense methods in existing code? Can existing solutions be used, or should developers adjust it specifically for their web application? Defense methods that can be used without any adjustment are the most interesting ones.
- *Ease of applications*: how much technical knowledge do developers need, to understand the defense method and implement it in their web application? An easy to implement defense method can be implemented at first.
- *Size*: how many functions within the web application are touched by a defense method? The more functions are touched, the more difficult it will be to implement defense methods in a correct way. In other words; how many functions should be changed before the defense method can be implemented?
- *Extensiveness*: to what extent do defense methods make the use of other defense methods superfluous? Some defense methods make it superfluous to implement other, smaller defense methods.

5.13.2 Classification of defense methods

The table below shows the attentiveness of implementing different types of defense methods. Each method can get a maximum of 15 points (3 for each factor). In case the score falls between 13 and 15 points, it gets a green color. Those methods are the most attractive to implement first. Methods with a score between 10 and 12 get a yellow color. The red color is given to any method with a score less than 10. Those are logically the least attractive to implement.

However, in some cases there is no green defense method against attacks via a certain vulnerability. Thus, a red color does not mean that this defense method will never be implemented in the code of web applications. In the next chapter of this research, both the impact of attacks via identified vulnerabilities and the attractiveness of defense methods are combined in the model. This model will tell developers, which defense methods should be used first.

It is important to mention, that scoring those defense methods is quite hard. This model is created from scratch, as there currently is no model for rating the attractiveness of those methods. While scoring, the most favorable cases have been assumed. For example, it has been recommended using frameworks with integrated defense methods where possible. However, every system and scenario will be different. It makes that in some cases extra steps are necessary to prevent web applications for the misuse of potential vulnerabilities by implementing extra defense methods within a framework. Scores may suggest that for those cases using a framework can be enough. In case of the most favorable scenarios, this should be.

Name	Framework	Plug & Play	Ease of application	Size	Extensiveness	Total
Disabling accounts	3	2	2	2	3	12
Multi-factor authentication	2	2	3	3	2	12
CAPTCHA	2	3	3	2	2	12
Non-sensitive accounts	3	3	3	3	1	13
Password length/complexity	2	3	3	3	1	12
Securing cascading stylesheets	2	2	3	1	2	10
Accessibility of password change function	3	3	3	3	1	13
Safe recovery questions	2	3	3	3	1	12
Token over side-channel	1	2	2	3	1	9
Transport layer security (TLS)	2	3	3	2	3	13
Session management	2	2	3	2	3	12
Encryption key storage guidelines	3	2	2	2	2	12
Encryption (symmetric & asymmetric)	3	2	2	2	2	11
Authentication of end devices	1	3	2	2	2	10
Data origin authentication (HMAC)	1	2	1	2	2	8
Role Based Access Control	2	2	2	2	2	10
Discretionary Access Control	1	1	2	2	2	8
Mandatory Access Control	2	2	2	2	2	10
White list input validation	2	3	3	2	1	11
Encoding function (escaping)	2	3	3	2	3	13

Prepared statements	2	3	3	2	2	12
Content Security Policy (CSP)	2	3	3	2	1	11
Sanitizer function	2	3	3	2	2	12
Output encoding function	2	2	2	2	2	10
Using a JavaScript framework	3	3	3	2	3	14
Synchronizer based token pattern	3	2	2	3	2	12
Encryption based token pattern	3	2	2	3	2	12
Double submit cookies	1	2	2	2	2	9
HMAC based token pattern	2	2	2	3	2	11
Re-authentication function	1	2	3	2	2	10
Unique IDs	2	3	3	3	2	13
Username policy (email/nickname)	2	2	3	3	2	12
Application logging	3	3	2	3	3	14
Safe password storage	2	2	2	3	3	12
Malicious code in uploaded files check	2	2	2	3	2	11
3rd party JavaScript management	2	2	2	3	3	12

Table 3: Attractiveness of implementing defense methods

Attractiveness (1 = low, 3 = high):

- Defense method is part of one or more frameworks
- Existence of plug & play options for a defense method (plug-in/extension)
- Necessary technical knowledge for implementing a defense method
- The amount of functions affected by implanting a defense method
- Defense method X makes it unnecessary to implement defense method Y or Z

Total # points:

5-10	Green
10-12	Yellow
13-15	Red

5.14 Recap

In this chapter different defense methods have been described and categorized by the identified vulnerabilities from earlier chapters. Based on the defense methods that have been found, they are ranked based on five factors. The ranking of both the vulnerabilities and defense methods will lead to the model, which will be designed in the next chapter.

6. Model for protecting web applications

In this last chapter the results from this research will be combined. First, the different vulnerabilities that may occur in the code of web applications have been identified. Second, different defense methods that work best to prevent web applications for attacks via those identified vulnerabilities have been described. The combination of both results in a new model, that helps developers and CEO's to understand on which possible vulnerabilities their focus should be.

6.1 Identification of vulnerabilities in web applications

For creating a new model, there has been started with implementing the different vulnerabilities found during this research. There is a difference in the style of the different lines used in the model. The continuous lines mark individual vulnerabilities in web applications, while dotted lines indicate a coherence between different vulnerabilities. In a sense that defense methods will help in preventing both types of vulnerabilities.

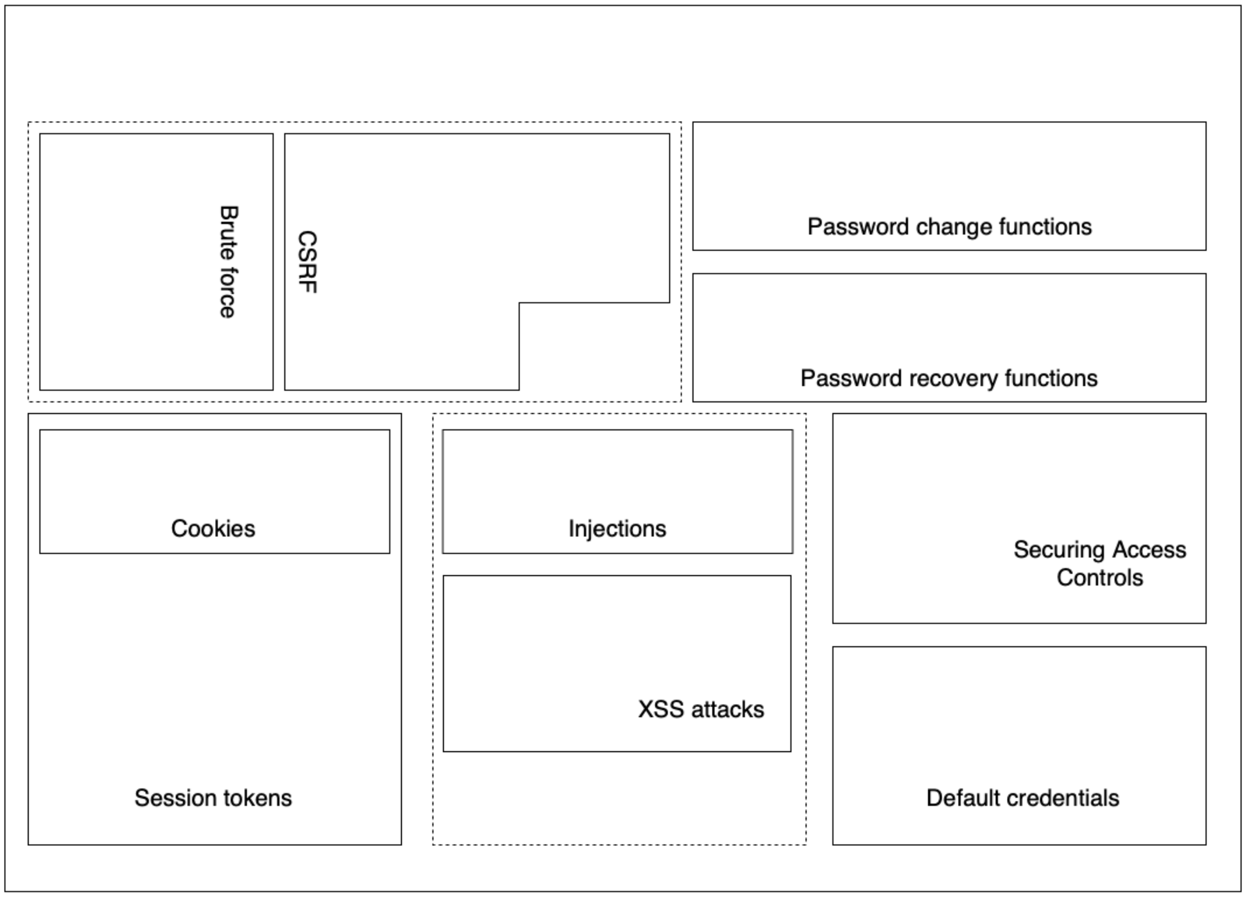


Fig. 11: Model with the identified vulnerabilities

6.2 Implementing different defense methods

The next step in creating a new model for developers and CEO's has been adding the defense methods that have been found. Some defense methods help developers by protecting their web application against several vulnerabilities, where others are focused on a single vulnerability. Defense methods outside one of the continuous lines are relevant for the defense against two or more vulnerabilities.

For example, CAPTCHA can be seen as a defense method that helps developers protecting their web application against brute force attacks and cross-site request forgery attacks. Using a JavaScript framework helps protecting web applications against all of those vulnerabilities. As in most of those frameworks some of the defense methods are already implemented.

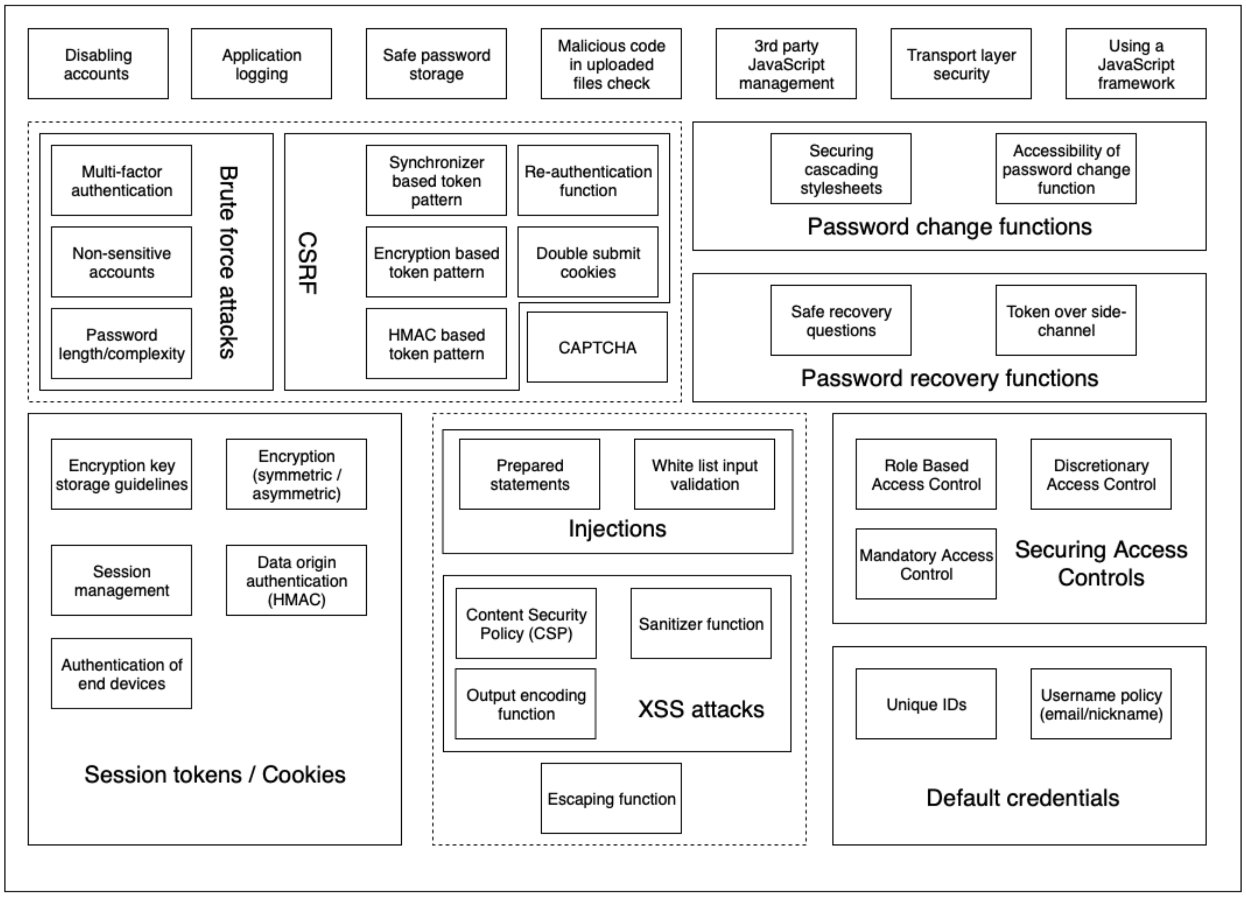

Disabling accounts
Application logging
Safe password storage
Malicious code in uploaded files check
3rd party JavaScript management
Transport layer security
Using a JavaScript framework
Multi-factor authentication
Non-sensitive accounts
Password length/complexity
Brute force attacks
CSRF
Synchronizer based token pattern
Re-authentication function
Encryption based token pattern
Double submit cookies
HMAC based token pattern
CAPTCHA
Securing cascading stylesheets
Accessibility of password change function
Password change functions
Safe recovery questions
Token over side-channel
Password recovery functions
Encryption key storage guidelines
Encryption (symmetric / asymmetric)
Session management
Data origin authentication (HMAC)
Authentication of end devices
Session tokens / Cookies
Prepared statements
White list input validation
Injections
Content Security Policy (CSP)
Sanitizer function
Output encoding function
XSS attacks
Escaping function
Role Based Access Control
Discretionary Access Control
Mandatory Access Control
Securing Access Controls
Unique IDs
Username policy (email/nickname)
Default credentials

Fig. 12: Model with vulnerabilities and defense methods

6.3 Combining risks and effort

In the last phase of combining the results of this research, the information about the impact of attacks via one of the identified vulnerabilities and the effort it costs to implement defense methods against them have been implemented. Effort can be seen as a relative term, as effort is coupled with the impact of defense methods on the defense as a whole. In some cases, it will cost developers some effort to implement a defense method, but this method could have a high impact on the defense as a whole. The defense method is green in such a case.

For vulnerabilities, the red colored blocks are the ones with the highest impact on organizations. So, based on the model, organizations should first focus on protecting themselves against injections and cross-site scripting attacks.

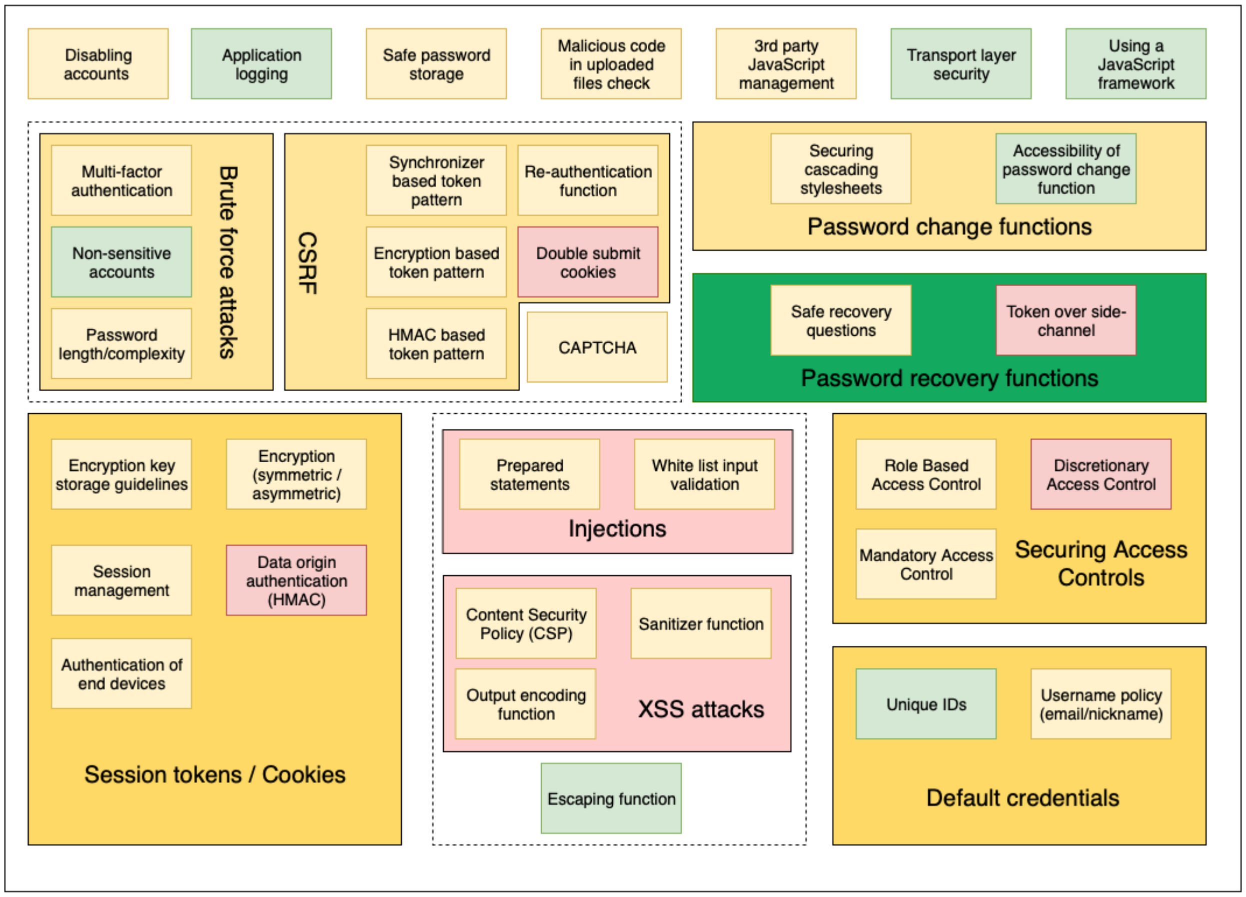

Disabling accounts
Application logging
Safe password storage
Malicious code in uploaded files check
3rd party JavaScript management
Transport layer security
Using a JavaScript framework
Multi-factor authentication
Non-sensitive accounts
Password length/complexity
Brute force attacks
CSRF
Synchronizer based token pattern
Re-authentication function
Encryption based token pattern
Double submit cookies
HMAC based token pattern
CAPTCHA
Securing cascading stylesheets
Accessibility of password change function
Password change functions
Safe recovery questions
Token over side-channel
Password recovery functions
Encryption key storage guidelines
Encryption (symmetric / asymmetric)
Session management
Data origin authentication (HMAC)
Authentication of end devices
Session tokens / Cookies
Prepared statements
White list input validation
Injections
Content Security Policy (CSP)
Sanitizer function
Output encoding function
XSS attacks
Escaping function
Role Based Access Control
Discretionary Access Control
Mandatory Access Control
Securing Access Controls
Unique IDs
Username policy (email/nickname)
Default credentials

Fig. 13: Final model with colored vulnerabilities and defense methods

In the different blocks the defense methods are colored with green, yellow and red as well. As it has been described earlier, green colored defense methods are quite easy to implement or have a high impact on the protection against vulnerabilities. So, for example, transport layer security and using unique IDs are two defense methods that should be implemented at first. Sending a token over a side-channel to protect misuse of the password recovery function, is not worth the investment in advance.

6.4 Moving towards a decision model

It can be hard to understand the current model without any knowledge about cyber security. To make it easier for developers to use insights from this research, the model has been translated into a decision model. For developers it is possible to walk through this decision model and decide which defense methods should be used to mitigate the most occurring vulnerabilities. The decision model can also be used to explain why certain measures have been taken to the C-line of organizations.

As well as in the other model in this research, colors are used to indicate the attractiveness of certain defense methods. The decision model shows that *application logging* and *transport layer security* are two defense methods that should always be used. After implementing those defense methods, developers can check which of the mentioned system functions are part of their own web application. In Appendix A, you will find a larger version of this decision model.

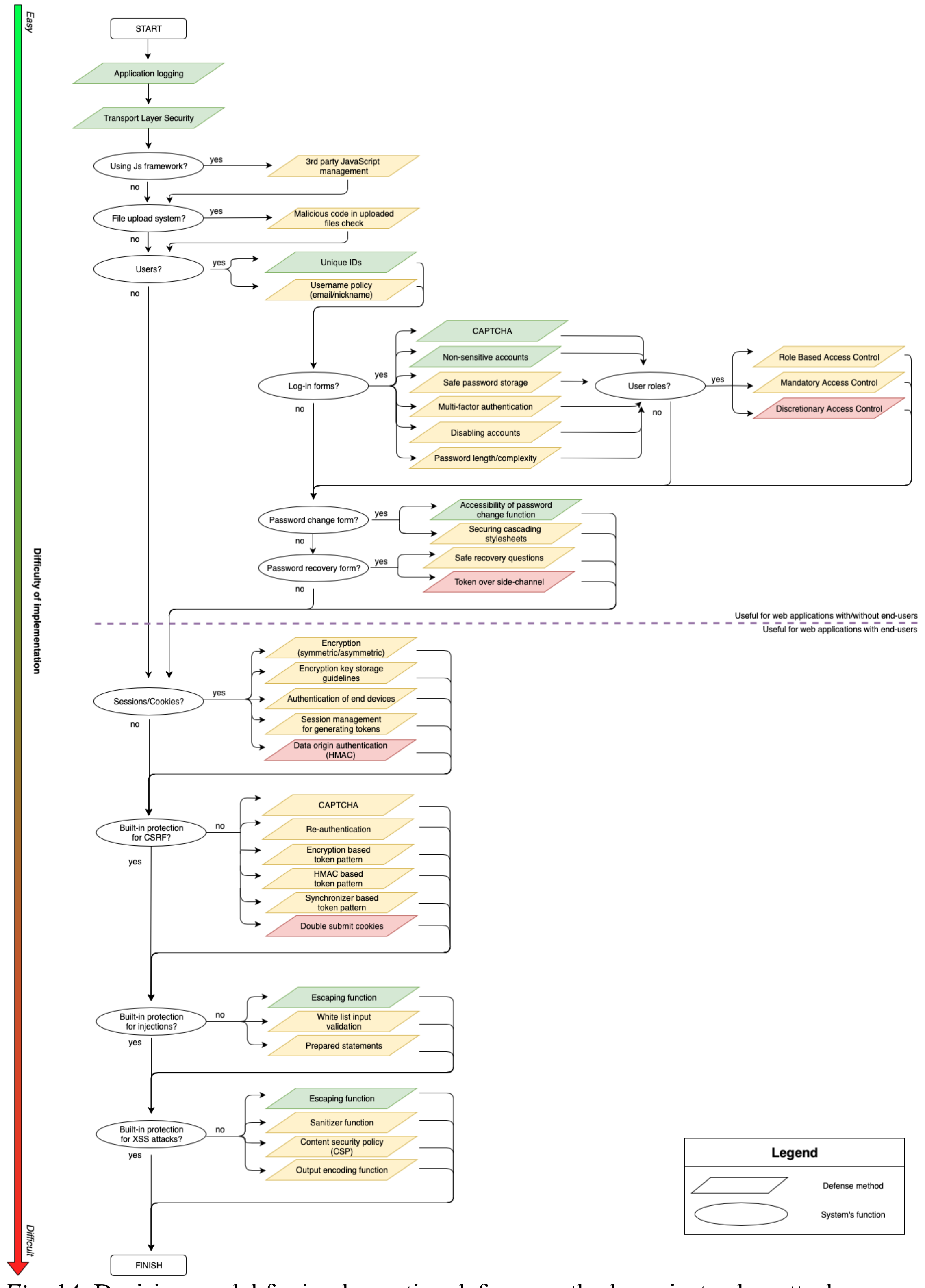

Easy
Difficulty of implementation
Difficult
START
Application logging
Transport Layer Security
Using Js framework?
yes
3rd party JavaScript management
no
File upload system?
yes
Malicious code in uploaded files check
no
Users?
yes
Unique IDs
Username policy (email/nickname)
no
Log-in forms?
yes
CAPTCHA
Non-sensitive accounts
Safe password storage
Multi-factor authentication
Disabling accounts
Password length/complexity
no
User roles?
yes
Role Based Access Control
Mandatory Access Control
Discretionary Access Control
no
Password change form?
yes
Accessibility of password change function
Securing cascading stylesheets
no
Password recovery form?
yes
Safe recovery questions
Token over side-channel
no
Useful for web applications with/without end-users
Useful for web applications with end-users
Sessions/Cookies?
yes
Encryption (symmetric/asymmetric)
Encryption key storage guidelines
Authentication of end devices
Session management for generating tokens
Data origin authentication (HMAC)
no
Built-in protection for CSRF?
no
CAPTCHA
Re-authentication
Encryption based token pattern
HMAC based token pattern
Synchronizer based token pattern
Double submit cookies
yes
Built-in protection for injections?
no
Escaping function
White list input validation
Prepared statements
yes
Built-in protection for XSS attacks?
no
Escaping function
Sanitizer function
Content security policy (CSP)
Output encoding function
yes
FINISH
Legend
Defense method
System's function

Fig. 14: Decision model for implementing defense methods against cyber attacks

6.5 Recap

This chapter combines the knowledge that has been collected during this research into a new model. First, a model implementing the different vulnerabilities and their defense methods has been developed. Based on this model a translation to a decision model has been made. This decision model can be seen as the end product of this research, as it tells developers which risks they face and which defense methods should be used to mitigate those risks.

7. Conclusion

During this research, the sub questions stated in the first chapter has been answered. Several choices were made to come to the models showed in chapter 6. This chapter focuses on the validity of those choices, the relation between the expectations beforehand and the results afterwards, limitations of this research and ideas for future work.

7.1 Validity of the research method

To come to the new model, two sub-questions in this research have been answered. First, this study has zoomed in on different vulnerabilities that can occur in web applications code. Here the expertise of cyber security specialists and statistics from different organizations that focus on cyber attacks have been used. The combination of both has been implemented in some activity flows, to check if all interesting vulnerabilities have been found. An indication of the business impact and difficulty of using vulnerabilities for attacks is based on the DREAD-model of Microsoft. A scale from one to three has been used to make ratings explainable. The difference between four and five on a scale from one to five is quite difficult to explain, while explaining the difference between two and three on the current scale is much easier.

In the second part of this study possible defense methods have been identified. Here, not every possible defense method has been mentioned but the study has been focused on the most common ones. Common ones are defined as the ones recommended by internationally accepted organizations. OWASP is an example of such an organization. As this is a non-profit organization, conflicts of interest were prevented. To check if those defense methods were useful for defending web applications against the identified vulnerabilities, they have been put on top of the activity flows and have been discussed with experts in the field.

Measuring the attractiveness of defense methods proved to be difficult, as there is no existing method for this. For this research a measurement method has been created, using five different factors. There are different ways to prove this measurement method helps in ranking defense methods. First, while defending physical buildings comparable factors are interesting to use. You want to optimize the defense in such a way, that it cost less time and budget to implement them. As well as for implementing code, technical knowledge can be necessary for using physical defense components. Second, those factors are factors that should be used while defending web applications.

In the last part of this research both have been combined; the identified vulnerabilities and defense methods to cover them. While identifying and describing different defense methods, it has been noticed that some of them are useful in covering several vulnerabilities. In the model this can be recognized to the different types of lines that are used. Also, some defense methods are interesting to use against the misuse of (almost) every vulnerability. Those are placed on top of the model.

7.2 Expectations and results

While writing a research proposal for this research, it has been found that some cyber attacks are used much more than others. For example, injections and cross-site scripting attacks are occurring a lot. The misuse of password functions is less attractive for attackers. This makes us think that either the difficulty of protecting web applications against those most occurring vulnerabilities is high, or the impact of those attacks is high.

This research led to two models that shows which vulnerabilities should be covered first, as well as that it tells developers which defense methods should be used. While analyzing the model, it has been seen that the impact of injections and cross-site scripting attacks is high. As well as that defending web applications against them is quite difficult (medium/high). On the other side, the impact of misusing a password recovery function is low. It shows that the expectations are in line with the results of this research.

Also, the most attractive defense methods from the model are currently implemented in the majority of web applications. For example, most web applications are based on an existing JavaScript framework. As well as the fact that many web applications are using CAPTCHA to prevent themselves against brute force and cross-site request forgery attacks.

Although the results of this model are in line with the expectations beforehand, it helped in further understanding which defense methods should be used first. As it has been mentioned in the first chapter of this research, there is a lot of information about covering possible vulnerabilities in web applications. However, in most cases this information is not structured well which makes it even harder to understand how risks should be mitigated.

This model shows what the impact of the misuse of different vulnerabilities could be in a clear way. As well as which defense methods should be used first to mitigate those risks. Here is where this model adds value to developers and companies as a whole. Not only developers are interested in a landscape of risks and solutions, also people in the C-line can profit from it. As it will be easier to explain them which risks there are and in which ones they should invest first.

7.3 Limitations of this research

This research helps to come to new insights. However, there are some points of discussion. Measuring the effects of this model on the number of incidents within web applications is quite difficult. The model should have been used for a period of at least a few months to come to a correct conclusion about the effectiveness of it. Thereby, in general, it will be difficult to measure of such a model will help developers in preventing cyber attacks. The model can be used as a directive, but it is not conclusive for the misuse of all possible vulnerabilities in web applications.

A second limitation of this research is the durability of the model. The world of IT is changing fast, as well as the types of attacks attackers use to misuse vulnerabilities in web applications. Although this model is useful for the coming years, it should be a dynamic one. Once a year the model should be updated, based on new vulnerabilities, outdated vulnerabilities and for example new defense methods.

Third, it was quite difficult to measure the attractiveness of different defense methods. Currently there is no model to rank those methods. This is why a new one has been developed, with the five factors that have been mentioned earlier. Although there are legitimate reasons to choose those factors, it is not conclusive. Other researchers may choose other factors, which could result in other classifications. Thereby, some assumptions about the way vulnerabilities are misused in web applications have been made. As mentioned earlier in this research, for every vulnerability or defense method the most logical or trivial one has been taken. However, the application of attacks and the difficulty of implementing defense methods may differ per system and scenario. It makes that scoring those attacks gives a good indication, but it may include some exceptions. In particular, the amount of effort required for consistent implementation of defense methods against the misuse of vulnerabilities can vary enormously.

7.4 Suggestions for further research

In this research the most occurring cyber vulnerabilities have been investigated. However, this does not cover every vulnerability that may occur after developing web applications. In future work the generated models can be further expanded. Not only by adding vulnerabilities and defense methods to the model, but also by diving deeper into the system behind web applications. The models that has been created focuses on the code of web applications, while there are possibilities to add defense layers on a server level. It may be interesting to add those defense methods to the models as well.

As it has been explained earlier, the model should be dynamic as cyber vulnerabilities and used technologies are changing. In the next years, the validity of this model can be researched. If necessary, vulnerabilities or defense methods can be left out or added.

A third idea for future work can be researching the effectiveness of this type of models. For example: how often are developers using models by developing their web applications? In case developers do not use those models that often, researchers may investigate how the use of models can be stimulated to decrease the number of vulnerabilities in the code of web applications.

7.5 Reflection

At the start of this research my knowledge about vulnerabilities, defense methods and cyber security in general was limited. I never expected that I would learn this much about the topic in a relatively short period. Along the way, I spoke with several experts in the field, read a lot about cyber attacks and defense methods and learned more about the aspects that cyber security includes. It became clear that cyber security is a broad topic, which makes it difficult to ensure that every relevant part is covered in a study.

It also showed me that this research is only a small step towards a model that covers everything. In my opinion, it will almost be impossible to create such a model. Attackers are getting smarter, while finding out new defense methods against attacks via vulnerabilities is not that simple. Altogether gave this research me a broad basis to continue working on expanding my knowledge about the topic and experts in the field a good starting point to expand the model.

At the end of my bachelor study I wrote a thesis about the use of data feeds to ensure safety during large events. This gave me a backpack with useful experience in setting up studies. The end result of this research, however, would never be as successful as now without the help of the people

around me at Software Improvement Group (SIG). They helped me with widening and then reducing the broadness of this research, with finding the right structure for this thesis and by giving feedback during the project.

Bibliography

OWASP. (n.d.). Injection Theory. Retrieved 20 April 2019, from https://www.owasp.org/index.php/Injection_Theory

Accenture. (2017). *Cost of cyber crime study*. Retrieved from https://www.accenture.com/t20171006T095146Z__w__/us-en/_acnmedia/PDF-62/Accenture-2017CostCybercrime-US-FINAL.pdf

Athanasopoulos, E., & Antonatos, S. (2006). Enhanced captchas: Using animation to tell humans and computers apart. *IFIP International Conference on Communications and Multimedia Security*, 97–108.

Basirico, J. (n.d.). Input Validation using Regular Expressions. Retrieved 5 November 2019, from https://blog.securityinnovation.com/blog/2011/03/input_validation_using_regular_expressions.html

Bonneau, J., Bursztein, E., Caron, I., Jackson, R., & Williamson, M. (2015). Secrets, Lies, and Account Recovery. *Proceedings of the 24th International Conference on World Wide Web - WWW '15*. https://doi.org/10.1145/2736277.2741691

Clarke, J. (2012). *SQL Injection Attacks and Defense* (2e ed.). London: Syngress.

Danhieux, P. (2018, October 25). The forgotten human factor driving web application security flaws. Retrieved 4 April 2019, from https://insights.securecodewarrior.com/the-forgotten-human-factor-driving-web-application-security-flaws/

Detectify. (2018, July 31). CAPTCHA does not prevent cross-site request forgery (CSRF). Retrieved 6 December 2019, from https://blog.detectify.com/2017/12/06/captcha-csrf/

Edgescan. (2018). *Vulnerability Statistics Report 2018*. Retrieved from https://www.edgescan.com/wp-content/uploads/2018/05/edgescan-stats-report-2018.pdf

Fogie, S., Grossman, J., Hansen, R., Rager, A., & Petkov, P. D. (2007). *XSS Attacks: Cross-site Scripting Exploits and Defense* (1st ed.). London: Syngress.

Hatzivasilis, G. (2017). Password-Hashing Status. *Cryptography*, *1*(2), 10. https://doi.org/10.3390/cryptography1020010

HMAC, Keyed-Hashing for Message Authentication. (n.d.). Retrieved 20 November 2019, from http://www.networksorcery.com/enp/data/hmac.htm

HTML5Lib. (2017, December 3). HTML5Lib Python [Software]. Retrieved 14 November 2019, from https://github.com/html5lib/html5lib-python/blob/master/html5lib/filters/sanitizer.py

IBM. (n.d.). Reauthentication. Retrieved 4 December 2019, from https://www.ibm.com/support/knowledgecenter/SSPREK_9.0.6/com.ibm.isam.doc/wrp_config/concept/con_reauthe.html

Joan, B. (2018, May 17). Difference Between MAC and DAC. Retrieved 5 November 2019, from http://www.differencebetween.net/technology/software-technology/difference-between-mac-and-dac/

Jovanovic, N., Kirda, E., & Kruegel, C. (2006). Preventing cross-site request forgery attacks. *Securecomm and Workshops, IEEE*, 1–10.

Kambalyal, C. (n.d.). *3-Tier Architecture*. Retrieved from http://channukambalyal.tripod.com/NTierArchitecture.pdf

KPN Internedservices. (2017, March 8). SSL, TLS, HTTPS. Wat is het en hoe werkt het? Retrieved 11 November 2019, from https://www.internedservices.nl/blog/ssl-tls-https-wat-het-en-hoe-werkt-het/

Ladkani, U. (2013, July 30). Prevent cross-site scripting attacks by encoding HTML responses. Retrieved 5 October 2019, from https://www.ibm.com/developerworks/library/se-prevent/index.html

Leaser, D. (2019, July 24). The demand for cybersecurity professionals is outstripping the supply of skilled workers. Retrieved 3 February 2020, from https://www.ibm.com/blogs/ibm-training/new-cybersecurity-threat-not-enough-talent-to-fill-open-security-jobs/

Li, D., Liu, C., Wei, Q., Liu, Z., & Liu, B. (2010). RBAC-Based Access Control for SaaS Systems. *2010 2nd International Conference on Information Engineering and Computer Science*. https://doi.org/10.1109/iciecs.2010.5678213

LINQPad. (n.d.). LINQPad - The .NET Programmer's Playground. Retrieved 6 November 2019, from https://www.linqpad.net

Makrushin, D. (2018, November 6). The cost of launching a DDoS attack. Retrieved 12 March 2019, from https://securelist.com/the-cost-of-launching-a-ddos-attack/77784/

Microsoft. (2017, March 16). SQL Injection. Retrieved 20 May 2019, from https://docs.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver15

OpenStack. (n.d.). Security/OSSA-Metrics. Retrieved 2 February 2020, from https://wiki.openstack.org/wiki/Security/OSSA-Metrics#DREAD

Mozilla. (2019, November 5). Content Security Policy (CSP). Retrieved 14 November 2019, from https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

OWASP. (n.d.-a). Access Control. Retrieved 7 October 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html

OWASP. (n.d.-b). Content Security Policy. Retrieved 12 November 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html

OWASP. (n.d.-c). Cross-site Scripting Prevention. Retrieved 14 November 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

OWASP. (n.d.-d). Cross-Site Request Forgery Prevention. Retrieved 29 October 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

OWASP. (n.d.-e). Forgot Password. Retrieved 10 November 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Forgot_Password_Cheat_Sheet.html

OWASP. (n.d.-f). Key Management. Retrieved 13 November 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html

OWASP. (n.d.-g). Protect File Upload Against Malicious File. Retrieved 12 December 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Protect_FileUpload_Against_Malicious_File.html

OWASP. (n.d.-h). Session Management. Retrieved 20 October 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

OWASP. (n.d.-i). SQL Injection Prevention. Retrieved 4 November 2019, from https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

OWASP. (n.d.-j). Third Party JavaScript Management. Retrieved 15 December 2019, from https://cheatsheetseries.owasp.org/cheatsheets/Third_Party_Javascript_Management_Cheat_Sheet.html

OWASP. (2017). Top 10 2017 - OWASP. Retrieved 7 March 2019, from https://www.owasp.org/index.php/Top_10-2017_Top_10

Poremba, S. (2019, July 9). Your Web Applications Are More Vulnerable Than You Think. Retrieved 10 April 2019, from https://securityintelligence.com/your-web-applications-are-more-vulnerable-than-you-think/

ProgrammerInterview.com. (2015, September 27). Prepared Statement Example. Retrieved 8 November 2019, from https://www.programmerinterview.com/database-sql/example-of-prepared-statements-and-sql-injection-prevention/

Rouse, M. (2015, December). What is endpoint authentication (device authentication)? Retrieved 15 November 2019, from https://whatis.techtarget.com/definition/endpoint-authentication

Sargent, J. (2017, October 23). Security vulnerabilities in JavaScript libraries are hard to avoid. Retrieved 20 October 2019, from https://sdtimes.com/angular/security-vulnerabilities-javascript-libraries-hard-avoid/

Sönmez Turan, M., Barker, E., Burr, W., & Chen, L. (2010). *Recommendation for Password-Based Key Derivation*. Retrieved from https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

Stack Exchange. (2010, November 14). What is the difference between RBAC and DAC/ACL? [Forum post]. Retrieved 15 October 2019, from https://security.stackexchange.com/questions/346/what-is-the-difference-between-rbac-and-dac-acl

Stack Overflow. (2009, August 20). What are the pros and cons of using an email as a username? [Forum post]. Retrieved 10 December 2019, from https://stackoverflow.com/questions/1303575/what-are-the-pros-and-cons-of-using-an-email-as-a-username/1303584

Stiawan, D., Idris, Mohd. Y., Malik, R. F., Nurmaini, S., Alsharif, N., & Budiarto, R. (2019). Investigating Brute Force Attack Patterns in IoT Network. *Journal of Electrical and Computer Engineering*, *2019*, 1–13. https://doi.org/10.1155/2019/4568368

Stuttard, D., & Pinto, M. (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Oxford: Wiley.

Techopedia.com. (n.d.). What is a parser? Retrieved 20 May 2019, from https://www.techopedia.com/definition/3854/parser

Veracode. (2017). *State of Software 2017*. Retrieved from https://info.veracode.com/report-state-of-software-security.html

Visser, J. (2017, June 14). Top 5: JavaScript frameworks. Retrieved 16 October 2019, from https://www.frontend-professionals.nl/top-5-javascript-frameworks/

W3Schools. (n.d.). SQL Stored Procedures. Retrieved 8 November 2019, from https://www.w3schools.com/sql/sql_stored_procedures.asp

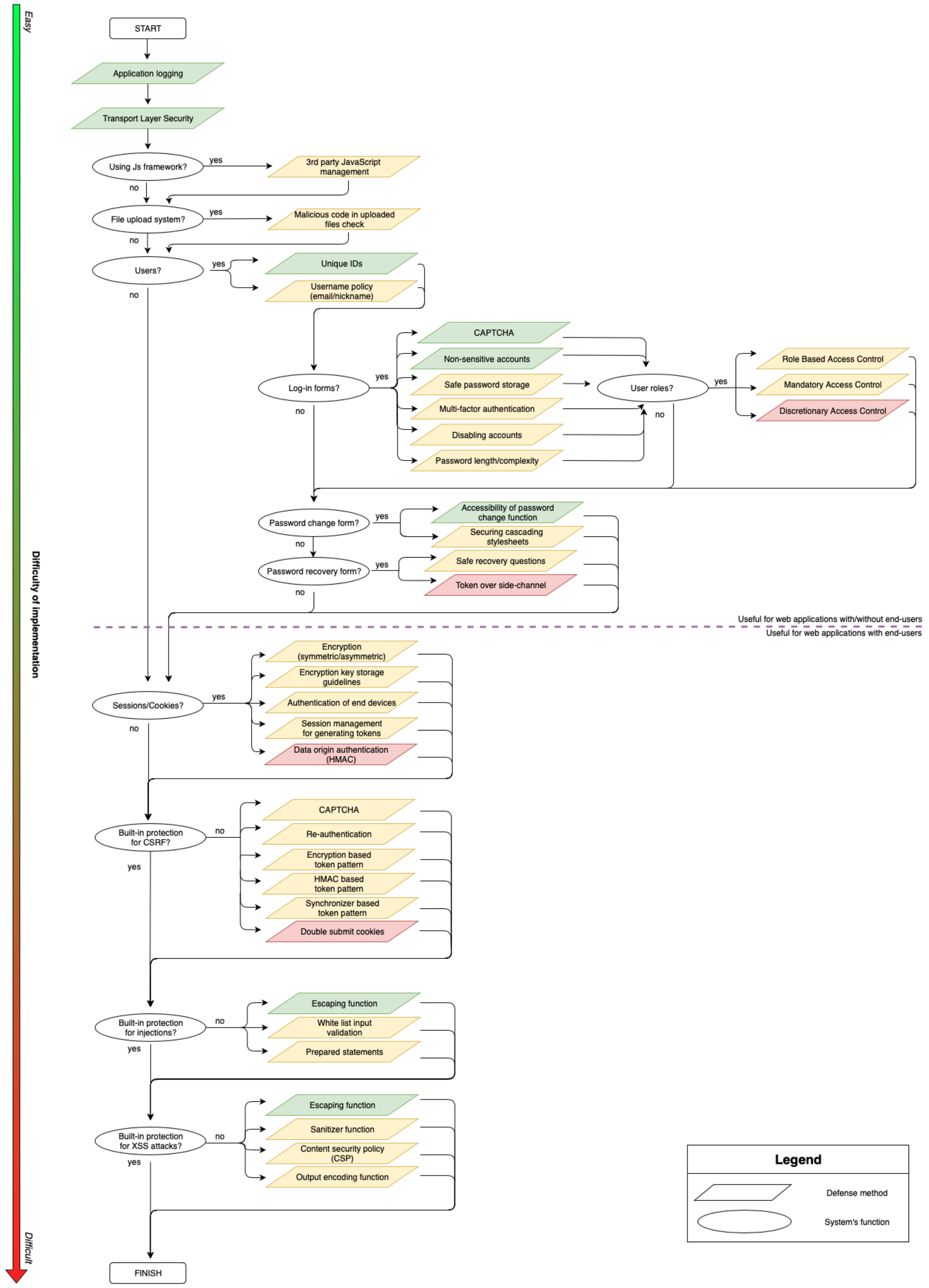
Web Application Security Consortium. (2008). *Web Application Security Statistics 2008*. Retrieved from http://projects.webappsec.org/f/WASS-SS-2008.pdf

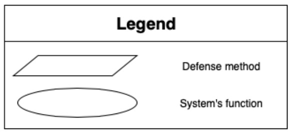
Weber, G. (n.d.). XSS Sanitize: sanitize untrusted HTML to prevent XSS attacks. Retrieved 16 November 2019, from http://hackage.haskell.org/package/xss-sanitize

Wikipedia. (2019, September 11). Argon2. Retrieved 11 December 2019, from https://en.wikipedia.org/wiki/Argon2

Wium Lie, H. (2005). *Cascading Style Sheets*. Retrieved from https://www.wiumlie.no/2006/phd/css.pdf

Appendix A: Decision model


Easy
Difficulty of implementation
Difficult
START
Application logging
Transport Layer Security
Using Js framework?
yes
3rd party JavaScript management
no
File upload system?
yes
Malicious code in uploaded files check
no
Users?
yes
Unique IDs
Username policy (email/nickname)
no
CAPTCHA
Non-sensitive accounts
Safe password storage
Multi-factor authentication
Disabling accounts
Password length/complexity
Log-in forms?
yes
no
User roles?
yes
no
Role Based Access Control
Mandatory Access Control
Discretionary Access Control
Password change form?
yes
Accessibility of password change function
Securing cascading stylesheets
no
Password recovery form?
yes
Safe recovery questions
Token over side-channel
no
Useful for web applications with/without end-users
Useful for web applications with end-users
Encryption (symmetric/asymmetric)
Encryption key storage guidelines
Authentication of end devices
Session management for generating tokens
Data origin authentication (HMAC)
Sessions/Cookies?
yes
no
Built-in protection for CSRF?
no
yes
CAPTCHA
Re-authentication
Encryption based token pattern
HMAC based token pattern
Synchronizer based token pattern
Double submit cookies
Built-in protection for injections?
no
yes
Escaping function
White list input validation
Prepared statements
Built-in protection for XSS attacks?
no
yes
Escaping function
Sanitizer function
Content security policy (CSP)
Output encoding function
FINISH


Legend
Defense method
System's function