AUTOMATIC TESTING OF SECURITY FLAWS IN WEB-APPLICATIONS

Sherwin Antonio Pantophlet
antoniopantophlet@gmail.com

August 18, 2017, 46 pages

Host supervisor: Arjan Janssen
Host organisation: DKTP Information Technology, https://www.dktp.nl/
Academic supervisor: Drs. P.R. GRIFFIOEN
ABSTRACT

The purpose of this thesis is to research the automatic testing of flaws in the security of software. The goals are specifically to understand how to categorise the various aspects of software-security with their respected software components and creating a step-by-step plan that can be applied to convert viable security aspects into automated tests.

To achieve this goal a literature study was conducted to analyse what other researchers discovered about this topic. Using the information found in the literature study and a research method inspired by the steps of an Action research, a roadmap was created and executed. This roadmap provides a systematic way to convert security aspects of software into automated software tests that focus solely on the possible exploits related to the aspect. The thesis also describes the setup and environment that was used for the execution of the roadmap, how it compared to more established security approaches, what the results and shortcomings were from this approach and how future research could improve and extend upon this topic.
Table of Contents

Abstract 2
Table of Contents 3
1. Introduction 5
2. Background 7
2.1. Related work 7
3. Problem statement 10
3.1. Context 10
4. Problem analysis 11
4.1. Research Questions 11
5. Research design 13
6. Requirements 15
6.1. Nonrepudiation & Accountability 18
6.2. Authenticity 20
6.3. Relevancy of software security aspects 22
6.4. Viability of security aspects 24
7. The roadmap 25
7.1. Misuse-case 25
7.2. Analysis 25
7.3. Create exploit scenarios 25
7.4. Construct test-cases for the scenarios 26
7.5. Convert the test-cases into automated test 26
7.6. Automated security test 26
8. Comparison with other software security approaches 27
9. Execution of the roadmap 29
9.1. Setup of the environment 29
9.2. Misuse-case 31
9.3. Analysis 31
9.4. Scenarios 31
9.5. Test-case 32
9.5.1. Verification 32
9.6. Automated test case 33
9.7. Results of the automated tests 34
9.8. Authenticity 35
9.9. Confidentiality & Integrity 35
9.10. Non-repudiation Accountability 35
10. Limitations 36
11. Discussion 38
12. Conclusions 39
12.1. Future research 40
Bibliography 41
Appendix A - Test Cases 43
Appendix B - Verification 45
1. Introduction

Test automation is software that automates any aspect of testing an application system with a capability to generate test inputs and expected results. It reduces the repetitive work that we do manually and also provides the result as 'pass' or 'fail' [5].

IEEE has many definitions of software testing, and one that was selected [3] that describes the complete process.

“This standard supports testing across the entire software development lifecycle, from static testing of requirements, specifications and other documentation, unit or component testing that is typically carried out by developers, integration testing of program modules, system testing of integrated systems, and user acceptance testing that is usually carried out by end-users. It also supports testing during maintenance cycles that typically occur after release.”

Software testing can be a very labour-intensive process, especially with larger systems that have ongoing development and release cycles. The manual testing of systems takes up too many resources, mainly, time and money. A literature study was carried-out about automated testing and testing for exploits in software and the results of this can be found in the second chapter of this thesis, from these sources the topic was formed for this thesis “Automatic Testing of Security Flaw in Web-Applications.”

The goal of this thesis is to provide solutions for the following questions:

1. What aspects of software security can be tested by using test-automation?
2. How to design a roadmap that can translate viable security characteristics into automated test?
3. What are the limitations of the roadmap?

The literature was one source of motivation to research this topic and the second source stems from the companies Portavita and DKTP Information Technology, they both invest a lot of time and resources towards the functional testing of software and will also provide context for this thesis. Both companies have a test coverage of automated test that cover about 70% and 30% of their systems respectively, but few of these automated test focus on testing the security of the systems. The rest of the introduction will focus on the scope of the topic and the structure of the document.

1.1. Scope

The term, security, covers a very broad topic. ISO/IEC 25010[11] describes security as a characteristic of a system with sub-characteristics, confidentiality, integrity, non-repudiation, accountability, and authenticity.

Haiyun Xu et al. created a model that’s an extension to the ISO/IEC 25010 model, and the extended model serves as a practical mapping system for properties that cover the characteristics [12].

This research will strictly focus on software; no hardware related methods or tests will be researched. Due to the limited time and resources available, not all of the security characteristics will be converted into automated test.
1.2. Structure
The thesis begins with a summary of the literature study. This chapter will discuss all the relevant research to this topic that was reviewed prior to beginning this thesis.
The problem statement chapter will provide the context of the research that will provide some background about the software that will be the subject of the research, a concise description of the issues that need to be addressed by this research and finally the concrete research questions that will form the focus of this thesis.
The chapter Research design explains in phases how the research was executed, what deliverables were produced in each phase and how they relate to the research questions.
The Requirements chapter gives an overview of prerequisites that will form the basis for the chapters that follow. Misuse-cases will be used in the chapter to visualise the requirements in an attempt to give a clear description of how potential exploits can be executed and eventually thwarted by counter-measures. Design and implementation will explain the plans and execution of the proposed solutions and the results of the experiment will be presented and analysed to attempt to verify if the proposed solution produces viable results. The chapter about limitations is constructed to discuss any shortcomings discovered during the execution phase and finally the discussion and conclusion will summarise all results of the thesis and how future research could extend upon this topic.
2. Background
A literature study was conducted to have a better understanding of the topics that relate to automated security testing. The following chapter is a summary of the findings.

2.1. Related work
There has been research in the recent past about the topic of automated testing for exploits in software. Vidar Kongsli wrote a paper to show how Selenium, a test automation framework with the main purpose of functionally testing web sites, could be used to create automatic tests that focus on security [1]. The research focused on the following exploits: cross site scripting (XSS), Cross Site Request Forgery (CSRF), broken authentication and access management, information leakage, and improper error handling. The results of the paper states that vulnerabilities related to insecure input handling, XSS, CSRF and Broken Authentication and Access Management were possible to be exposed and tested via the user interface but vulnerabilities related to “insecure cryptographic storage” will be difficult to test with this approach.

Job Jonkergouw also performed research that focused on how exploits could be discovered, but by using automated static and dynamic analysis [10]. The research designs used in both papers were a source of inspiration for how I conducted my experiments.

Sources were also reviewed that provided information on methods that could elicit security characteristics and visualise potential exploits of software. Initially, I looked at the ISO/IEC 25010 [11], because the ISO gives a clear overview of the different attributes that cover the security of a system and provides a clear domain for which characteristics of security are essential and could be explored.

ISO/IEC 25010 was a good starting point, but a more tangible concept was needed. The paper, written by Haiyun Xu et al [12], presents a practical model for rating the security of software products and computer systems. Figure 1 is an illustration of that practical model. The model serves as an extension of the ISO/IEC 25010 with a practical mapping system for properties that cover these characteristics. The model helps clarify what functionality is connected to what characteristic and gives a practical way to pinpoint what part of the system may have potential security risks and should be taken into account when prioritizing security tests.
The paper clarifies what functionality is connected to what security characteristic, and this provides a practical way to pinpoint what part of the system may have potential security vulnerabilities and should be taken into account when prioritizing tests that focus on security.

One of the main obstacles was how to document or visualise vulnerabilities, and research was done for an efficient and clear way to envision them. Several studies (Sindre, & Opdahl. 2004; Soomro & Ahmed. 2013; Herrmann et al. 2011) give detailed explanations of Risk-oriented Misuse cases and how they help organizations assess and prioritize potential risks. Misuse cases provide a very easy way to explain how malicious user(s) can use these exploits to compromise the system. It also provides a way to depict elements that can mitigate these exploits in one diagram.

The paper written by Jason Bau et al [6] carried out a study that analysed leading tools that were black-box web application vulnerability scanners. The study focused on three main categories the class of vulnerabilities tested, the effectiveness of the tools against target vulnerabilities and finally the relevance of the targeted vulnerabilities to vulnerabilities found in production websites. The papers also made reference to the Open Web Application Security Project (OWASP) the community that publishes the OWASP Top-10, which is a compressive list of the top 10 most critical software security exploits for web-applications. The main goal of this community and subsequently the Top-10 list is to promote awareness of the most prominent security flaws and exploits to which web-applications are exposed, but it also provides concrete scenarios on how these exploits could be executed. The OWASP Top-10 gives realistic reference points to what types of exploits and scenarios could be relevant to security characteristics and system properties displayed in the ISO/IEC 25010.

Mohamed Ghazouani et al [24] wrote a paper inspired by the subjectiveness of risk management methodologies and the focus of the study is to propose a mathematical formulation of risk by using a lower level of granularity of its elements: threat, probability, criteria used to determine an asset’s value, exposure, frequency and existing protection

![Fig 1. Mapping eleven system properties to ISO/IEC 25010 security sub-characteristics](image)
measure. The choice was made not to use the formula proposed by this paper but to use a simpler approach that was also mentioned in the paper namely the risk management matrix that is derived from the NIST SP 800-30 documentation[25]. The risk-matrix defined by NIST SP 800-30 is more subjective when determining a risk than the mathematical formulation proposed by Mohamed Ghazouani et al., but the simplicity of the risk-matrix makes it more approachable for different stakeholders that may be involved when actually having to discuss and/or analyse risks. Also the documentation was published by the National Institute of Standards and Technology (NIST) and has been cited over 1290 times. The level of risk that a security characteristic has will be taken into account when determining the relevancy of that characteristic.
3. Problem statement

The goal of this chapter is to provide a concise description of the issues that were addressed by this research, first by describing the context of the research, then a summary of the problem analysis, and finally the refined goal of the research with corresponding research questions.

3.1. Context

The software that was used for the experiments was a forked version of the Enterprise Resource Planning (ERP) system that DKTP uses. The system was developed in-house and is composed of four main modules: Customer relationship management (CRM), Sale module, Project and resource planning module, and Accounting. The version that will be used for the experiment will be 0.8-beta.

<table>
<thead>
<tr>
<th>Lines Of Code</th>
<th>Files</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>15,474 PHP</td>
<td>318</td>
<td>826</td>
</tr>
<tr>
<td>47 Directories</td>
<td>32,377</td>
<td>91 Classes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6,391 Statements</td>
</tr>
</tbody>
</table>

image 1: Statistics of DKTP ERP

Applied technologies

Each of the modules of the ERP makes use of a range of different technologies. The entire system was developed, using the RAD framework YII, and is required to run on Apache web server with PHP 5.4. The system uses a rational database model, managed by MySQL, and the interface is comprised of dynamically generated web pages and return basic HTML. These web pages include CSS for the makeup, JavaScript for several types of user interaction, and finally, JSON and XML for displaying large amounts of data.

Diagram 1. Static structure of a YII application
4. Problem analysis
Automated tests provide assurance to developers and testers that software is functioning according the specifications. The perspective of security is relevant with any major design or functional change to the architecture, but the testing of the security has not been the main focal point of the automated tests.

The literature that was reviewed for this thesis showed that the documentation or visualisation of vulnerabilities of the system in a standardised way is one of the main obstacles for testing software security. Also mapping these concepts to concrete system attributes and functionality shows to be a difficult task in some respect.

Interviews were held with developers, tester, privacy-, and security-officers to determine if this was, in fact, a main obstacle and to see what other obstacles played a role.

These findings from the interviews were presented to the same group of developers, tester, user, privacy-, and security-officers, and a discussion was constructed to see what issues needed to be solved before automated tests that focus on security can be created.

The following list of obstacles was compiled from the discussion.

• Software testing is a very labour intensive task, and the lack of resources comes into play.
• Risk-analysis has been made over various security issues/threats, but a clear, automated solution has not been defined.
• How to document and report these threats and the solution to mitigate them
• How to integrate the automated tests into the current test suites
• How to validate an automated test

These are the main obstacles that need to be addressed to ensure that a proper a solution is produced.

4.1. Research Questions
The goal of the research is to develop a framework or roadmap that could provide a way to automatically test software for security vulnerabilities. Motivation for the research has two main sources; the companies request for a suitable and reproducible method to automate the testing of security in software, and the literature that was discussed in chapter two. The literature provided a detailed picture of what security characteristics exists and their mapping to software components. This information contributes focus and scope for the research.

Question 1: What aspects of software security can be tested by using test-automation?

The topic of security is a very broad subject. To have a comprehensive understanding on how this can be properly tested in an automated way there should be research into what aspects of software security exist and which of these could possibly be viable entry points for automated tests. The approach to answer this question will be done by analysing characteristics of security aspects on their relevancy and viability. This analysis will help filter which security aspects should be testable by using test-automation.
Sub-question 1.1: What security aspects are relevant?
The set of security aspects discovered should be analysed further to conclude which aspects are important enough to be tested using test automation. The relevancy of the aspect will be based on the risk if there is a breach in the security and the probability of the breach actually happening.

Sub-question 1.2: Which security aspects are viable candidates for test automation?
The goal of this question is to determine if the security aspects that were regarded as relevant are also viable candidates for test automation. To achieve this goal a static analysis will be executed to determine if it’s theoretically possible to test the security aspects by using test automation. The static analysis will look at the system attributes that are connected to these aspects to decide they are viable candidates for automated test.

Question 2: How to design a roadmap that can translate viable security characteristics into automated test?
Once a viable list of testable security characteristics is constructed a method should be in place to convert them into automated security tests. The method is a step-by-step plan that provides a guide on how to convert the concept of the aspects into an concrete automated test that would be able to detect vulnerabilities in the software.
The roadmap should adhere to the following criterium:

- **Reproducible:** The list of the steps for the roadmap has to be reproducible to ensure that the code and data are assembled in a way so that another group can re-create all of the results.
- **Explicit:** Each step of the roadmap should be clear on what actions and artefacts are expected on how it contributes to the end result.
- **Traceable:** Documenting an artefact produced by a step and providing bi-directional traceability between the associated steps.

The second part of this question is to investigate if it’s possible to implement a working prototype by using information provided by the roadmap.

Sub-question 2.1 How does it compare to other software security approaches that are used in the software development cycle?
A paper written by Mohammed, N. M et al [25] identified 52 software security approaches used in the software development cycle and categorised them into five categories. By applying the same systematic mapping to the roadmap it would give a better indication of how it compares to the other more established software security approaches.

Question 3: What are the limitations of the roadmap?
The main goal of the question is to make sure that anyone who may consider to use the roadmap on system knows clearly what are the limitations are of the roadmap and what kind of impediments can be encountered when trying to implement it.
5. Research design

The purpose of the previous phases was to analyze the current situation and clarify the main research goals. Once these concepts were defined, a general description was gathered of the architecture, what security measures were in place, and how are they tested. The research method for this investigation was inspired by the Action Research methods, but does not contain the steps that are part of Action research because they do not completely fit with this research topic. In Action Research, the researchers attempt to solve a real-world problem, while simultaneously studying the experience of solving the problem. The adopted solution is desirable knowledge gained from the research, empowering particular individuals or groups, and facilitating a wider change [13]. The research will be split into four phases with their respected deliverables and which research question will be answered or partially contribute to an answered.

Phase 1: Requirements

This phase defined and modelled the requirements for the automated tests that were eventually created. To do this, misuse-cases were used. This phase explained the general purpose of these diagrams, how they are structured, and how they will be translated into an automated test. The results of this phase was an array of misuse-cases that were based on the characteristics defined by the model created by Haiyun Xu et al [12]. The model and characteristics were chosen because they are mapped to practical system attribute. This provides systematic way to connect the relationship between the misuse-cases and what parts of the system they may affect. The secondary goal of this phase was to create a priority list that will influence which misuse cases will be translated from diagram to a test-case and, eventually, constructed into an automatic test. The secondary goal main purpose was to provided additional scope to the research because of limited time and resources.

This phase helped to answer the first question “What aspects of software security can be tested by using test-automation?”

Phase 2: Design the roadmap and implement a prototype

In this phase a systematic and traceable way to translate the misuse-cases designed in phase one into automated security test was created. The second research question described a criteria that the created roadmap adhered to. The second part of this phase implemented a prototype based on the roadmap. The results of the prototype served as verification that roadmap can produce viable solution for converting misuse-cases in to automated test that detect vulnerabilities while also meeting the requirements of the second research question.

This phase helped to answer the second question “How to design a roadmap that can translate viable security characteristics into automated test.”

Phase 3: Results

The prototype created in the “Design and implementation” phase was used to generate a data set of results of the test that will attempted to exploit the system. These results were gathered and aggregated into graphs to provide a detailed picture of how the tests performed.
This phase helped to answer the second question “How to design a roadmap that can translate viable security characteristic into automated test.”

**Phase 4: Analysis of the results**

The data generated by the prototype in the previous phase was analysed to assess and help systematically verify wether the roadmap could produce a system to see if the solution could potentially solve the current problems faced with automating security focused selenium test.

This phase helped to answer the second question “How to design a roadmap that can translate viable security characteristics into automated test.”
6. Requirements
The following sub-chapters will explain each security characteristic and the corresponding system property. As a part of the explanation, I will utilise misuse-cases to show how these system properties could be potentially exploited.

6.1. Misuse-cases
This research will use misuse-cases as reference to describe a potential security vulnerability and input for creating the automated security test. Misuse cases were introduced by Sindre and Opdhal [19] and is an extended model of standard UML use cases with the main purpose of modelling potential security concerns. Sindre and Opdahl defined *misuse case* as a list or sequence of steps that, if performed by an agent successfully, cause harm to the stakeholder and/or to the system. Misuse-cases will be used to create an automatic test that attempts to carry-out the exploit(s) that they portray. The end of this chapter will explain the transformation process in detail. The following part of this chapter explains the components that are added to the standard UML use cases that turn them into misuse cases.

They define *misuser* as an actor who is willing to use the system with unfavourable intents. Initially, only threats were modelled as misuse cases.

Misusers execute actions that may compromise the security of the system. These actions also expose a potential weakness in the system.

A weakness of the system is defined as a vulnerability, and this threatens security of the system.

Mitigation defines system functionality or attributes that protect it from identified vulnerabilities.
6.2. Confidentiality & Integrity
The characteristic confidentiality ensures that data is accessible only by those who are authorized. The characteristic Integrity promotes the prevention of unauthorized access and/or modifications. These characteristics are usually represented by the following five system properties.

Secure data transport
Data should be transported over a secured interface that ensures protection against it being accessed by anyone who isn’t authorized.

Diagram 1. Misuse-case for secure data transport

The attacker attempts to listens to the stream of data between the user and the system to obtain credential information but the system uses a secure transport medium and ensures that any data that is communicated between the user and the system is encrypted.

Secure data storage
If data is stored, the storage medium (databases and file-servers) should also be secured against unauthorized access. Database encryption and access control fall under this category.

Diagram 2. Misuse-case for secure data storage
The attacker knows the location of where data is stored and attempts to gain unauthorised access to this data. The attempt is mitigated because the system encrypted the data that is stored and uses access management to ensure only authorized users and/or clients have access to the data.

**Authorized data access**

The system has a way to authorize what data can be accessed by what entity.

![Diagram 3. Misuse-case for Authorized data access](image)

The attacker attempts to gain access to the system but is blocked because only user with valid credentials are allowed to access the system.

**In/output verification**

When data is being stored or retrieved, the content of the data should be verified according to specifications. In the case of data being not corrected, it should either be rejected by the system or stripped of any harmful content. Exploits, such as SQL-injection, are a perfect example of these kind of exploits.

![Diagram 4. Misuse-case for in/output verification](image)
The attacker, who can also be an authorised user of the system, sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database. This attack is blocked by the system by either rejecting the file or purifying the information in the file so that it does not affect the system.

6.3. Nonrepudiation & Accountability

Nonrepudiation represents actions or events that can be proven to have taken place, and Accountability stands for actions of an entity that can be traced only to that specific entity. Combining these two characteristics gives us the possibility to trace when, what, or who did an action or triggered an event to access or modify a specific entity.

Strength of proof

The accountability and integrity of the information connected to that proof is accounted for and wasn't tampered with or contaminated by outside sources.

Diagram 5. Misuse-case for strength of proof

The attacker, who can also be an authorised user of the system, attempts to corrupt information in the system. The system uses encryption and access logs to ensure the integrity of the information and in the event of corruption it’s possible to prove that the corruption actually happened.
Logging completeness

Logging completeness assures that the logging mechanism provides a complete picture of what happens when actions are executed or events are triggered.

![Diagram 6. Misuse-case for logging completeness](image)

The attacker, who can also be an authorised user of the system, modifies data that he is not authorised to access. The actions that allowed this corruption of the data were logged and are retraceable for a system administrator to determine how, what and when the attack actually happened.

Unique identification

Unique identification ensures that entities, actions, and events have unique and traceable identities. This attribute is closely tied to logging completeness, because it enhances the integrity of the logging mechanism.

![Diagram 7. Misuse-case for unique identification](image)
The attacker, who is an authorized system user, executes malicious actions on unauthorized data. The actions where executed but the id of the attacker is known because The actions where logged and connected to his unique id.

6.4. Authenticity

This characteristic revolves around the act of confirming the truth of an attribute of a single piece of data (datum) or entity. In the case of users, this is confirming the actual identity.

Access management strength

Access management can be broken down in to three categories. This is achieved by picking one or more authentication method(s) and each method has an inherent strength “authentication method strength”. The inherent strength of each method also depends on the implementation “authentication method implementation” and prevention of circumvention “authentication enforcement”.

![Diagram 8. Misuse-case for access management strength](image)

The attacker uses leaks or flaws in the authentication or access management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users. In this case the attacker attempts a brute-force attack on the module that is responsible for authentication. The attack is blocked by the module because it was built to respond to these type of attacks.
**Session management strength**

The session manager governs a way to preserve certain data across subsequent accesses. An attack against a poor implementation can break the entire site, because a session attack’s main goal is to obtain access to an application under someone else’s credentials. XSS-attacks and SQL-injections could be used to exploit sessions.

![Diagram 9. Misuse-case for session management strength](image)

The attacker, who can also be an authorised user of the system, sends text-based attack scripts that exploit the interpreter in the browser to hijack the session of another user. This attack is block by the system by either rejecting the file or purifying the information in the file so that it does not affect the system.

**Secure user management**

User manager controls user-credentials and roles; if this module of the application has flaws, attackers may have the ability to access and modify credentials that provide access to sensitive information.

![Diagram 10. Misuse-case for Secure user](image)
The attacker, who is an authorized system user, changes parameter values that directly refers to system resources and/or information for which the user isn’t authorized to access. This is blocked by combined execution of the access-manager and user-manager.

### 6.5. Relevancy of software security aspects

To assess the relevancy of a software security aspect the choice was made to examine two points. The first point is to check if a security aspect could be mapped to one or more software exploits found in the OWASP TOP-10 list. The second point is to do an risk analysis using the risk-matrix defined in the NIST SP 800-30 documentation[25].

If a software security aspect that could not be mapped to one of the OWASP TOP-10 list and falls in the category low-risk will not be seen as relevant enough to be used for further investigation.

OWASP TOP-10 is a list of the top 10 most critical software security exploits for web-applications. Mapping the exploits to the software security aspects gives an overview of how they relate to the real-world and what category of exploits happen frequently enough to be considered a critical software security concern.

<table>
<thead>
<tr>
<th>Web Application Security Risks</th>
<th>Security aspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 Injection</td>
<td>Confidentiality &amp; Integrity</td>
</tr>
<tr>
<td>A2 Broken Authentication and Session Management</td>
<td>Authenticity</td>
</tr>
<tr>
<td>A3 Cross-Site Scripting (XSS)</td>
<td>Confidentiality &amp; Integrity</td>
</tr>
<tr>
<td>A4 Broken Access Control</td>
<td>Authenticity</td>
</tr>
<tr>
<td>A5 Security Misconfiguration</td>
<td>Authenticity</td>
</tr>
<tr>
<td>A6 Sensitive Data Exposure</td>
<td>Confidentiality &amp; Integrity</td>
</tr>
<tr>
<td>A7 Insufficient Attack Protection</td>
<td>Nonrepudiation &amp; Accountability / Authenticity</td>
</tr>
<tr>
<td>A8 Cross-Site Request Forgery (CSRF)</td>
<td>Confidentiality &amp; Integrity</td>
</tr>
<tr>
<td>A9 Using Components with Known Vulnerabilities</td>
<td>Confidentiality &amp; Integrity / Nonrepudiation &amp; Accountability / Authenticity /</td>
</tr>
<tr>
<td>A10 Underprotected APIs</td>
<td>Confidentiality &amp; Integrity / Nonrepudiation &amp; Accountability / Authenticity /</td>
</tr>
</tbody>
</table>

*Table 1. Mapping 10 most critical software security exploits to security aspect*

“Confidentiality & Integrity” and Authenticity are both mapped to six web application security risks. Nonrepudiation & Accountability was mapped to three web application security risks.
The risk matrix defined by the NIST SP 800-30 documentation[25] combines two factors, namely “Threat likelihood” and the “Impact”, when determining assessing a risk. The likelihood rating indicates the probability that a potential vulnerability may be exercised within the construct of the associated threat environment. The probability assigned for each threat likelihood level is 1.0 for High, 0.5 for Medium, 0.1 for Low. The adverse impact resulting from a successful threat exercise of a vulnerability. The value assigned for each impact level is 100 for High, 50 for Medium, and 10 for Low.

The combining the values of the two factors results into risk index. the NIST SP 800-30 documentation regards the following Risk scale: High ( >50 to 100); Medium ( >10 to 50); Low (1 to 10). Any security aspect that falls in the low scale will be considered to be not relevant enough to be automated.

<table>
<thead>
<tr>
<th>Impact</th>
<th>Low (10)</th>
<th>Medium (50)</th>
<th>High (100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threat likelihood</td>
<td>High(1.0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Medium(0.5)</td>
<td>-</td>
<td>Confidentiality &amp; Integrity</td>
</tr>
<tr>
<td></td>
<td>Low(0.1)</td>
<td>-</td>
<td>Nonrepudiation &amp; Accountability</td>
</tr>
</tbody>
</table>

*Table 2. risk matrix for software security aspects*

An exploit in the authenticity of a system can result in allowing someone with invalid or forged credentials access to a system. This category has six spots in the OWASP TOP-10 with 3 of those spots in the top 5 suggesting that the frequency and probability to attempt to take advantages of security flaws in the authenticity of a system is high.

A breach in the aspect of confidentiality & Integrity can result in allowing an unauthorized person to access, modify or delete data and can lead to the leakage or invalidation of confidential or important data, this scenario alone can be catastrophic for any company. This category also earned six spots in the OWASP TOP-10 suggesting that the frequency and probability to attempt to take advantages of security flaws is aspect of is moderately frequent.

Nonrepudiation & Accountability; Nonrepudiation represents actions or events that can be proven to have taken place, and Accountability stands for actions of an entity that can be traced only to that specific entity. This category earned three spots in the OWASp TOP-10 suggesting but on the lower end of scale. Having a low frequency and medium impact results in a low risk.

All three software security aspects could have been mapped to one or more an exploit found in the OWASP-TOP and they also had a risk level of medium or High according to the risk-matrix. The results of this chapter show that all of the security aspects were seen relevant according to the criteria and they will be investigated further to determine if they are viable candidates for automated test.
6.6. Viability of security aspects

To determine if the relevant security aspects are viable candidates to be converted into automated tests, an analysis of each system attribute connected to the security aspect was done.

The analysis looked at the following properties:

- Is the system attribute connected to an interface and be accessed by the user?
- Does the interface have user feedback?

The following list shows the mapping of the system attributes to the properties. Each mapping has a simple “Yes” or “No” to assert if the property is applicable to the system attribute.

<table>
<thead>
<tr>
<th>Security Aspect</th>
<th>System Attribute</th>
<th>User interface</th>
<th>User feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality &amp; Integrity</td>
<td>Secure data transport</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Secure data storage</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Authorized data access</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Nonrepudiation &amp; Accountability</td>
<td>In/output verification</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Strength of proof</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Logging completeness</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Authenticity</td>
<td>Unique identification</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Access management strength</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Session management strength</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Secure user management</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 4. List of viable candidates for automated security tests

Authorized data access, In/output verification, Logging completeness, Unique identification, Access management strength, Session management strength and Secure user management meet the requirements to be deemed as viable candidates to be converted into automated tests. Due to a limited amount of time and resources, not all the possible misuse-cases will be converted into a selenium test, thus a selection of at least one sub-characteristic from each category will be made for further investigation. The following sub-characteristics have been selected: Secure data transport, Unique identification In/output verification, Authorized data access, Logging completeness, and Authorized data access.
7. The roadmap
To have a better understanding of the process on how a misuse-case is translated from a diagram to into a functional test a roadmap was be created. The misuse-case that will be used for the explanation is “In/output verification”. The following diagram shows the main steps of this process. Inspiration for the roadmap came from a case study written by Evans, G. [23] that analyses the requirements captured in use cases and transforms them into implementable representations that can be directly coded.
Every phase of the roadmap will explain what the goal of that phase is, how it should be executed and what artefacts are expected to be produced by each phase.

Diagram 11. Transformation diagram

7.1. Misuse-case
Misuse cases were introduced by Sindre and Opdhal [19] and is an extended model of standard UML use cases with the main purpose of modelling potential security concerns. They also provide a way to depict elements that can mitigate these exploits in one diagram. The goal of this step is to create a array of misuse-cases that would represent potential threats that could compromise a system.

7.2. Analysis
Using the information from the table created by Haiyun Xu et al and requirements of the misuse cases the second step of roadmap is to analyze the component(s) of the system. The analysis produces probable components or interfaces of a system that can be utilised by a malicious user or exploit.

7.3. Create exploit scenarios
The third step is to create a concrete scenario of an exploit. A scenario gives detailed descriptions or list of instructions of how the exploit is executed. Examples of scenarios can be on the OWASP top ten list.
7.4. Construct test-cases for the scenarios
The fourth step is to construct test-cases from the scenarios. All test-cases should give a
detailed description of how the exploit works and what steps are needed to assert if the
exploit has compromised the system.
This step also include a verification process that will ensure the constructed test-cases will
adhere the expected specifications and they will fulfil their intended purpose. The
template used for documenting the test-cases can be found in the appendix of this
document.

7.5. Convert the test-cases into automated test
The fifth and final step would be to convert the steps found in the test-case into an
automated test. The method and technologies used in the step is up to the person or team
who has the task to execute the conversion. Using the example of input/output verification
the review material showed that there are various combinations of characters that are
used for injection exploits. The automated security test for this test-case should be carried
out with varying inputs to ensure that the test is a representative simulation of an actual
attempt to exploit the system.

7.6. Automated security test
All of the previous steps of the roadmap serve as specifications for how the automated test
should be designed and this step serves as the actual implementation of those requirements
into a automated test. The output of the final step of the roadmap produces an automated
test for that should conform to the specifications created by the misuse-case and should
follow all the procedures defined in the test-case. The choice of technologies, Architecture
and/or programming language used to create these automated test depends on the
environment and resources of the developers involved.
8. Comparison with other software security approaches

Mohammed, N. M et al [25] did a study that identified 52 software security approaches used in the software development cycle and categorised them into five categories. By applying the same systematic mapping to the roadmap it would give a better indication of how it compares to the other more established software security approaches.

The list of categories are:

- **Secure requirements modelling:** Security requirements are non-functional requirements that relate to system confidentiality, integrity and availability. The focus of the models describe functional behaviour that enforces security.

- **Vulnerability identification, adaption and mitigation:** The category investigates the source code searching to identify possible vulnerabilities and use methods to remove or mitigate these vulnerabilities.

- **Software security focused process:** this approach focuses on activities that helps developers build more secure software and address security compliance requirements.

- **Extended UML-based secure modelling profiles:** The extension of Unified Modelling Language for integrating security related information in UML specifications.

- **Non UML-based secure modelling notations:** Notations that create models for integrating security related information that are not based on or extensions of the UML specification.

<table>
<thead>
<tr>
<th>Secure requirements modelling</th>
<th>Vulnerability identification, adaption and mitigation</th>
<th>Software security focused process</th>
<th>Extended UML-based secure modelling profiles</th>
<th>Non UML-based secure modelling notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misuse-cases</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Analyze system components</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Create exploit scenarios</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Construct test-cases</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Convert test-case in automated test</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

*Table 5. Mapping of roadmap steps to software security approach categories*

The roadmap is composed out of different steps and because of this it could be mapped to three categories namely “Secure requirements modelling”, “Vulnerability identification, adaption and mitigation” and “Extended UML-based secure modeling profiles”. The step that focuses on misuse-cases can be mapped to two categories “Secure requirements modelling” and “Extended UML-based secure modeling profiles” and this step can be executed in the requirements, design or coding phase. The analysis step is can be a static, dynamic or hybrid analysis of the source code and this is mapped to “Vulnerability identification, adaption and mitigation”. The creation of exploit scenarios, the construction
of test-cases, and conversion of test-cases into automated test are also mapped to “Vulnerability identification, adaption and mitigation” these steps can be executed during or after the coding period.

The results of the paper showed that the most referenced category was “Vulnerability identification, adaption and mitigation” accounting for 49% of all the sources used in the study and 41% of the sources considered executing security checks during the coding phase of the software development lifecycle while applying security checks during the entire software development lifecycle had received less consideration. The second most frequent category is “Extended UML-based secure modeling profiles” accounting for 22%. Comparing the roadmap to the results of the paper shows that the roadmap falls in line to what research has shown to be the more popular methods when it comes to software security approaches.
9. Execution of the roadmap

This chapter will give an overview of the execution of the roadmap, that will translate misuse cases into automated tests. The focus will be on one security aspect namely Confidentiality & Integrity and the corresponding system attribute Input/Output verification.

A decomposition of the environment that will be used to execute the test is explained in the following sub-chapter followed by each step of the roadmap with their respected artefacts.

9.1. Setup of the environment

The following diagram depicts all the important components. Each part of the of the diagram will get a brief description about what they are and their purpose. The inspiration for this design came from Andreassen, O. O., & Tarasenko, A. (2014) [15]. Continuous integration is not a requirement for any steps of the roadmap but having an environment that supports this process makes it possible to repeatedly execute security tests once they have been created.

![Diagram 11. Design of the total environment](image)

**Jenkins**

Jenkins (version 1.607), by default, provides various services that support continuous integration. Jenkins will check for changes in the various repositories; these include the application-, selenium- and phpunit codebase. Once a change is pushed to specific branches, Jenkins will run the test-suite (phpunit and/or selenium). If all the tests pass,
then Jenkins will execute a script that runs the sonar-runner, which will produce reports on the quality of the code.

**Sonar**

Sonar (version 5.0) is an open-source platform that analyses and inspects the quality of the codebase. The purpose of sonar in this setup is to provide continuous reports about the quality of the code. Sonar can also be used to search for specific constructs that may provide vulnerabilities for security.

**Repository**

The git repository is where all the source code of ERP is stored. This is the subject of the automated security test.

**Selenium**

Selenium is a test automation framework with the main purpose of functionally testing web-based application and websites, and this repository will be the source code of the automated security tests. All the translated misuse cases will be converted into selenium test cases. When new code is pushed to specific branches (currently integration and/or staging), the Selenium suite will be executed. It was explicitly chosen not to implement a graphical user interface compared to other tools, such as the one developed by OWASP, because it gave me the option to design the prototype with less restrictions and can be executed from almost any environment. The approach also forces to have flexible methods to configure various attributes, and the main architecture can be modified without having too many dependencies between components.

**PHPUnit**

These unit tests verify that the core functionality of the system hasn’t been broken due to changes. Every time new code is pushed to any branch, unit tests will be executed. These do not focus on security, but they were implemented to ensure that no functionality is broken if necessary modifications occurred during the test period. Unit tests are triggers by git-hooks.

**Browser scope**

Selenium and the forked version of the software can run in multiple browsers, but because of the limited time and resources, all tests will be conducted in Firefox (version 36.01) to ensure consistent results.
9.2. Misuse-case
The use-case for in/output verification was created in chapter six. For the use-case a hacker/malicious user attempts to use any form system in/output exploits to execute code or commands that may compromising the system. The system attributes connected to this misuse-case use Html-purification and file/input-verification to mitigate injection exploits.

9.3. Analysis
In the case of “In/output verification”, any input field or container that can accept or display data from the database can be used for this exploit. All form elements for the system are generated by classes. From this information it was deduced that there are seven different input types used in all of the forms.

9.4. Scenarios
The scenario will be based on the description of an exploit found in the OWASP list of top 10 security exploits and flaws. In the case of “input/output verification”, OWASP top-ten has “A3-Cross-Site Scripting (XSS)” that defines the following scenario:

Scenario for input/output verification:
“Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources such as data from the database.”
9.5. Test-case

<table>
<thead>
<tr>
<th>Test-case input/output verification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Characteristic</strong></td>
</tr>
<tr>
<td><strong>Test Case Name</strong></td>
</tr>
<tr>
<td><strong>System</strong></td>
</tr>
<tr>
<td><strong>Design date</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre-conditions</strong></td>
</tr>
</tbody>
</table>
| **Instructions** | 1. Log in as a user  
2. Go to create new account  
3. Enter required fields  
4. Enter “XSS-script” into the open text-field  
5. Verify that the xss-script has not been executed |
| **Expected results** | Malicious scripts and text are not accepted or stripped |
| **Post-condition** | — |

*Table 6. Test-case for input/output verification*

9.5.1. Verification

The test-cases that derive from the misuse-cases need to be verified to ensure they are testing what they need to test and to guarantee that enough was tested.

In/output verification will simulate a user attempting to save invalid content into the system. The system should reject the input of the user and return a notification or warning.

All html form elements are generated by a class, and there are 7 different input types used in all of the forms. Each form has input validation and most forms use html-purification before saving information to the database. For every input type, an extensive list of common exploits will be inserted to see if the system accepts the input and observed to see if the exploit is executed.
9.6. Automated test case

The following diagram is the class-diagram from the source code of the input/output verification test.

![Class Diagram](image)

Diagram 13. class diagram of the InputOutputVerificationTest

All tests extend from the class “DefaultTestCase” this class initialises all components needed to perform a test. Each test uses an array of “Page” classes. Page classes are mappings to actual webpages and make it possible to navigate to different areas of the application. A list of instructions are carried out with an explicit verification(s) at the end of the test to check if the exploit was successful or not.

The following snippet of the code shows how parameterisation is used to execute a test multiple times with varying input variables. All the output of each test will be collected and reports will be automatically generated with a overview of all the results by the test-suite.

```java
@Parameterized.Parameters
public static List<Object[]> data() throws IOException, ParseException {
    int amount = DefaultJsonReader.readJsonFile(FILE_LOCATION_FOR_XSS_SCRIPTS, LIST_TYPE).size();
    return Arrays.asList(new Object[amount][0]);
}

@Test
public void runXssAttackMultipleTimes() {
    simulateXssAttack(xssScripts.get(currentParameter));
    currentParameter++;
}

private void simulateXssAttack(String xssScript) {
    createLogHeader("User attempts insert and execute Xss-Attack: " + xssScript);
    createCompanyWithXssScript(xssScript);
    activateXssAttack();
    AssertThatTheAttackFailed();
}
```

image 2: Source snippet of input/output verification test
9.7. Results of the automated tests
This sub-chapter will discuss the outcome of the tests that were created by following the roadmap in the selenium prototype. There were security attributes that made the selection, but their misuse-cases could not have been converted into selenium tests, and the reason for this will be discussed in the “discussion” chapter. To ensure that all results can be evaluated in the same way, all tests were built with the following notion. A security test is constructed to ensure that no exploits were executed and could be detected by the test; if a test passes, then no exploits were found, and if it fails, an exploit was detected.

![Fig 3. Output of Allure reports](image-url)
9.8. Authenticity
Authenticity had one test that focus on “Session Management strength”, and there were no failures. Access management strength and Secure user management Strength weren’t tested, but test-cases were created and could have been covered with Selenium.

9.9. Confidentiality & Integrity
Confidentiality & Integrity had in total 118 tests. Two tests were related to “Authorized data access” and had no failures. “In/output verification” had 116 tests of which 50 had failures. The 50 failures were tests that allowed xss-scripts to be executed after being entered and submitted into input fields.

9.10. Non-repudiation Accountability
Security attributes of Non-repudiation Accountability, such as "Logging Completeness" and "Unique identification". Each security aspect had one test to verify that no exploits were executed and both were successful.
10. Limitations

The main goal of this question is to make sure that anyone who may consider to use the roadmap knows clearly the most important limitations and what kind of impediments can be encountered when trying to execute it. The conclusion stem from the discoveries and experiences while executing the roadmap and analysis of the results that it produces.

- Unable to automate certain misuse-cases: There were attributes and their corresponding misuse-cases that proved too difficult to test without access or modification to the internal source code of the system, or there wasn’t enough time to produce a test that could be properly verified. The attributes that fell into this category were Secure data storage, Secure data transport, and Strength of proof. Secure data storage focuses on where the data is stored, and selenium focuses on the functional aspect of the system. Secure data transport focuses on the transport medium of the data between the various components, and not the functional aspects of the system. Strength of proof requires an array of information to calculate the actual strength of the proof, and a selenium test wouldn't provide the amount of analytical data that would be needed to give a definite conclusion.

- No support for parallel execution of activities: The way the roadmap is structured doesn’t promote the parallel execution of steps and this may not be the most optimal approach when multiple stakeholders such as developers, testers or other teams have the capacity to execute a step of the roadmap but can not because the previous step(s) have yet to be completed.

- Not having a representative test environment: The specifications for test environment should be created once the misuse-cases are defined, this can save time and effort because this process can be done while the later steps of the roadmap are being executed. If this is not done or poorly executed it may result in having a test environment that is not representative of the production environment and this may lead to tests being invalidated.

- Heavy dependancy on the implementation step: The environment or the choice of implementation can influence the amount of dependencies that are needed to create an automated security test. The majority of the roadmap is meant to be executed during the coding phase. In this phase it's possible due to restrictions or impediments there may be a change in the implementation of a feature and the automated test that was created for this specific feature or set of features may not yield valid results. The restriction for when implementing a feature may cause a change in the design of the software and could result in a misuse-case becoming invalid or more misuse-cases may be needed to properly verify that the feature is secure. Also features that rely on third-party software or plugins may also be more difficult to test because of the possibility of unknown external behaviour. The framework chosen to execute the automated test can add extra dependencies and these dependencies can contribute to how or what can be tested. The framework used to create the automated test for this thesis was Selenium and one of the major problems encountered was browser version support. When a new version update was released it broke all automated test. More limitations became more evident when an attempt was made to create tests that focused on security attributes that weren’t directly tied to an external interface. Tests that weren't suitable
for selenium could still be automated by using other frameworks that don’t focus purely on the functional aspect of the software, like JUnit and/or PHPUnit.

- Lack of security expert knowledge: Not having personnel with enough experience with the software and the knowledge of how exploits can be executed may lead to the misuse-cases that don’t cover enough to verify that a given feature is properly protected against potential exploits.
11. Discussion
The results from chapter nine showed that the automated tests that were created by following the roadmap defined in chapter seven were able to execute exploits and proved that there were vulnerabilities in the security of the software. The results show that the version of the software used for the experiment (0.8-beta) was very vulnerable to cross-site scripting attacks thus falling into the security aspect of “Confidentiality & Integrity”. Approximately 43% of the automated tests failed, these tests showed that a malicious script was injected into the system, executed, and consequently detected by the test. These results raised a lot of concern, because the vulnerability was a system-wide problem and potentially has a very big impact on the integrity of the system. Once the results were presented, each failure was manually tested to verify that the findings were correct and all 50 of the failures were verified.

<table>
<thead>
<tr>
<th>Build version 0.8-Beta</th>
<th>Failed</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authenticity</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Confidentiality Integrity</td>
<td>50</td>
<td>116</td>
</tr>
<tr>
<td>Nonrepudiation Accountability</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Totals</td>
<td>50</td>
<td>119</td>
</tr>
</tbody>
</table>

Table 7. Summary of results

While not a part of the results, these same tests were executed on a recent version of the software (version 1.1). This version of the software only had one failure, but this failure was deemed negligible, because it was an isolated issue and was subsequently removed in a patch update.

<table>
<thead>
<tr>
<th>System Attribute</th>
<th>Tested with Selenium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure data transport</td>
<td>No</td>
</tr>
<tr>
<td>Secure data storage</td>
<td>No</td>
</tr>
<tr>
<td>Authorized data access</td>
<td>Yes</td>
</tr>
<tr>
<td>In/output verification</td>
<td>Yes</td>
</tr>
<tr>
<td>Confidentiality &amp; Integrity</td>
<td></td>
</tr>
<tr>
<td>Strength of proof</td>
<td>Possible (but requires more data)</td>
</tr>
<tr>
<td>Logging completeness</td>
<td>Yes</td>
</tr>
<tr>
<td>Unique identification</td>
<td>Yes</td>
</tr>
<tr>
<td>Nonrepudiation &amp; Accountability</td>
<td></td>
</tr>
<tr>
<td>Access management strength</td>
<td>Yes</td>
</tr>
<tr>
<td>Session management strength</td>
<td>Yes</td>
</tr>
<tr>
<td>Secure user management</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 8. Summary of the results from build version 0.8
12. Conclusions

“Question 1: What aspects of software security can be tested by using test-automation?”

The literature study of chapter two touched on various aspects of software security. One of the topics found in the paper written by Haiyun Xu et al [12], presented a practical model for rating the security of software products and computer systems. This gave insight to what characteristics of software security are possible and could be directly mapped to system attributes. Using that perspective for guidance, the misuse-cases defined in the chapter about requirements visualise the potential threats for the possible characteristics, thus creating a list of possible security characteristics that could be automated. By combining the mapping of the possible security aspects to the OWASP top-10 list and the risk analysis defined by the risk-matrix specified in the NIST SP 800-30 documentation[25] a list of relevant aspects was created for the final analysis. The final analysis was aimed at determining if the relevant security aspects are viable candidates to be converted into automated test. This was done by analysing each of system attributes connected to the security aspect and using a simple systematic mapping to see if they met the minimum criteria to become automated test. The results of the filtering showed that Confidentiality & Integrity, Nonrepudiation & Accountability and Authenticity are viable candidates for test automation.

“Question 2: How to design a roadmap that can translate viable security characteristics into automated test?”

The roadmap that was created in chapter seven analyses the requirements portrayed in the misuse-cases and with a series of transformations turns the misuse-cases into implementable representations that can be directly programmed into automated security tests. To give better perspective on how the roadmap compared to other software security approaches the systematic mapping of categories defined by the study done by Mohammed, N. M et al [25] was used to map the type of approach this roadmap can be compared to. The comparison showed that it could be mapped to three categories namely “Secure requirements modelling”, “Vulnerability identification, adaption and mitigation” and “Extended UML-based secure modeling profiles”. The comparison showed that the roadmap falls in line to what research has shown to be the more popular methods when it comes to software security approaches. Based on the results generated by these automated tests, it was concluded that the automated tests created by following the roadmap could detect possible security exploits. The results showed that approximately 43% of the automated tests were able to execute an exploit. They where test that did cross-site scripting attacks and are mapped to the software security aspect Confidentiality & Integrity. This security aspect had a medium probability of happening with a high impact on the risk matrix, these vulnerabilities could have been extremely damaging if they were still present on the production version of the software.

“Question 3: What are the limitations of the roadmap?”

The execution of the roadmap revealed some limitation or short-comings that were encountered, they were the inability to automate misuse-cases without access or modification to the internal source code of the system, No support for parallel execution of activities, heavy dependancy on the implementation chosen and finally the possible Lack
of security expert knowledge may lead to misuse-cases that don't cover enough to verify that a given feature is properly protected against potential exploits.

### 12.1. Future research

It is possible to perform more research in addition to what was investigated for this thesis. The system attribute, “Strength of proof”, was proven difficult without a proper amount of data. Research on how to detect or measure the accountability and integrity of the information to ensure that it wasn’t tampered with or contaminated by outside sources could be a very interesting avenue to investigate. Also, Modern Applications have become more interactive than ever, and JavaScript plays a major role in this. JavaScript is, by nature, an event-driven and very dynamic language. It also interacts with the Document Object Model in the runtime. These attributes make it a very attractive choice for developers, but they are also what make it so challenging to test. Automated security focused tests for JavaScript could be an interesting direction for further research.
Bibliography


Appendix A - Test Cases

<table>
<thead>
<tr>
<th>Test-case Authorized data access and Secure user management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Characteristic</strong></td>
</tr>
<tr>
<td><strong>Test Case Name</strong></td>
</tr>
<tr>
<td><strong>System</strong></td>
</tr>
<tr>
<td><strong>Version</strong></td>
</tr>
<tr>
<td><strong>Design date</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre-conditions</strong></td>
</tr>
<tr>
<td><strong>Instructions</strong></td>
</tr>
<tr>
<td>1.</td>
</tr>
<tr>
<td>2.</td>
</tr>
<tr>
<td>3.</td>
</tr>
<tr>
<td>4.</td>
</tr>
<tr>
<td>5.</td>
</tr>
<tr>
<td>6.</td>
</tr>
<tr>
<td>7.</td>
</tr>
<tr>
<td>8.</td>
</tr>
<tr>
<td><strong>Expected results</strong></td>
</tr>
<tr>
<td><strong>Post-condition</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test-case input/output verification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Characteristic</strong></td>
</tr>
<tr>
<td><strong>Test Case Name</strong></td>
</tr>
<tr>
<td><strong>System</strong></td>
</tr>
<tr>
<td><strong>Version</strong></td>
</tr>
<tr>
<td><strong>Design date</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Pre-conditions</strong></td>
</tr>
</tbody>
</table>
## Test-case input/output verification

<table>
<thead>
<tr>
<th>Instructions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Log in as a user</td>
</tr>
<tr>
<td>2.</td>
<td>Go to create new account</td>
</tr>
<tr>
<td>3.</td>
<td>Enter required fields</td>
</tr>
<tr>
<td>4.</td>
<td>Enter “XSS-script” into the open text-field</td>
</tr>
<tr>
<td>5.</td>
<td>Verify that the xss-script has not been executed</td>
</tr>
</tbody>
</table>

**Expected results** Malicious scripts and text are not accepted or stripped

**Post-condition** —

## Test-case input/output verification

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Confidentiality &amp; Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case Name</td>
<td>input/output verification</td>
</tr>
<tr>
<td>System</td>
<td>DKTP-CRM</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Design date</td>
<td>01-04-2015</td>
</tr>
<tr>
<td>Description</td>
<td>Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources</td>
</tr>
</tbody>
</table>

**Pre-conditions** —

<table>
<thead>
<tr>
<th>Instructions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Log in as a user</td>
</tr>
<tr>
<td>2.</td>
<td>Go to create new account</td>
</tr>
<tr>
<td>3.</td>
<td>Enter required fields</td>
</tr>
<tr>
<td>4.</td>
<td>Enter “XSS-script” into the open text-field</td>
</tr>
<tr>
<td>5.</td>
<td>Verify that the xss-script has not been executed</td>
</tr>
</tbody>
</table>

**Expected results** Malicious scripts and text are not accepted or stripped

**Post-condition** —
Appendix B - Verification

The test-cases that derive from the misuse-cases need to be verified to ensure they are testing what they need to test and to guarantee that enough was tested.

Verification of Authorized data access test

The first test-case attempts to find vulnerabilities in two system attributes, namely: Authorized data access and Secure user management. These tests will be a simulation of a malicious user, using a brute-force attack to possibly gain access to the system. A set combination of usernames and passwords will be used to verify that only proper credentials will allow access. For a control, a correct username and password will be entered to see that access is possible.

The following combinations will be used in the brute-force attack:

- The combination of invalid usernames with invalid passwords
- The combination of invalid usernames with valid passwords
- The combination of valid usernames with invalid passwords
- Control: Combination of a valid username with its corresponding password

All combinations that have invalid credentials should be denied, and only the control test should be allowed access to the system.

The second test-case (Authorized data access) will be using an user that has limited access to the system with a valid username and password, and this user will attempt to access parts of the system that aren’t permitted.

There are 7 different account types in the system, and there are 41 unique actions that can be executed, but because of the Access Control Lists (ACL), there is a total of 105 combinations. For each account, a set of selected actions that are not permitted for that specific will be executed. The test expects that the action does not execute, and the user is redirected to a warning page.

The third test-case (In/output verification) will simulate a user attempting to save invalid content into the system. The system should reject the input of the user and return a warning.

All html form elements are generated by a class, and there are 7 different input types used in all of the forms. Each form has input validation and most forms use html-purification before saving information to the database. For every input type, a list of common exploits will be inserted to see if the system accepts the input and observed to see if the exploit is executed.

Verification of Authorized Logging completeness test

The test-case (Logging completeness and Unique identification) will be utilising an array of different users that execute actions. The test should be able to access the log page and locate the set of actions with their unique attributes. The actions should have a unique identifier that is only connected to that action.
Verification of Authorized data access test
The test-case (Session management strength) will simulate a user who saves a malicious text that contains a script that will attempt to execute an XSS-attack on the system. The system should strip any potentially malicious content or reject it entirely. Only one HTML form element accepts any format of text and doesn’t use validation once the content is saved.