U.P.B. Sci. Bull., Series C, Vol. 85, Iss. 1, 2023 ISSN 2286-3540

SDLC AND THE IMPORTANCE OF SOFTWARE SECURITY

Andreea-Iulia CONCEA-PRISĂCARU[1], Tudor-Alin NIȚESCU[2], Valentin SGÂRCIU[3]

Security vulnerability mitigation in the Software Development Life Cycle (SDLC) represents an important step when designing and delivering a software product, in the context of omnipresent high potential risk factors. The mitigation of security threats can be performed more easily in the present due to the variety of secure code scanning tools and the awareness spread on this area by organizations such as OWASP, which are frequently analyzing and classifying the latest software security vulnerabilities.

Keywords: software development life cycle, software security, security threats, OWASP, secure code scanning, vulnerability mitigation

1. Introduction

As we live in the technology era, there is no secret that software development has had and continues to have a constantly increasingly growth on the market. Furthermore, the diversity of software solutions is remarkable in a lot of areas, such as: IT, automotive, banking, medical, educational, marketing etc. Moreover, these solutions are easing our day-to-day activities by saving our most important resources, time, money and energy. It is true that the IT field and the software solutions may have a lot of advantages, as we highlighted earlier, but we shouldn't neglect the disadvantages either. Here I am referring to the software vulnerabilities, malwares and cyberattacks that we are exposed to via these software applications. In this regard, our paperwork will be focused on software development life cycle (SDLC) and the weight of software security in this process.

In the next section we will present the context in which we approached this topic, but also relevant findings from the same area of interest.

[1] PhD student, Dept. of Automation and Industrial Informatics, University POLITEHNICA of Bucharest, Romania, e-mail: andreea.concea@stud.acs.upb.ro
[2] PhD student, Dept. of Automation and Industrial Informatics, University POLITEHNICA of Bucharest, Romania, e-mail: tudor_alin.nitescu@stud.acs.upb.ro
[3] Prof., Dept. of Automation and Industrial Informatics, University POLITEHNICA of Bucharest, Romania, e-mail: valentin.sgarciu@upb.ro

2. State of the art

The need for secure programming and software security has been in the center of attention lately. This trend has been triggered by remote working. During the pandemic, remote working emerged as a lifeline for all of us, a lot of areas/businesses migrating their activity to the online environment.

This fact involved using a variety of software solutions available now on the market (open source / licensed), connecting to VPN (virtual private network), transferring documents/information and more. All the activities mentioned above are exposing the users of the applications to a multitude of risks, risks which we should prevent and mitigate as much as possible. That being said, the need for software security is clearly visible and the main stakeholders of this process are the software security researchers and the software development companies. This section presents the current state of the art in the field of software security, focusing on the latest findings.

"The Software Security Threat" topic is found in the spotlight of a group of Ukrainian researchers, their paperwork is concentrated on documenting the main security threats concepts and the determination of threat ratio. Their study presents the advantages and disadvantages between web applications and standalone applications, but also the different types of risk factors we are exposed to, for each category. The results of the study have shown that standalone applications are more vulnerable, having a greater risk score. Their work highlights the importance of minimizing the risk score and the factors that are influencing it (the possibility of performing the threat, investment attractiveness and the attacker qualification degree). With a high level of impact over the risk score of the application and the minimization of it, they mention the volume of scientific research in the field, the preparation level of the engineers and the software tool used for the determination and mitigation of the vulnerabilities [1].

Moreover, not only the researchers were interested in this area, but also the software developers and their employers. The topic was taken to the research level by the business enterprises, and it was materialized on an extensive study among different roles (developers, scrum masters, managers) which addressed this important element of the security development life cycle. The results were not very pleasing, they have shown the lack of awareness, training and competence on this subject and at the same time the need of increasing the knowledge and development level on this extent. Among the programmers it was found that a percentage of 34% software developers are not pleased with the performance of the software solution used within their company, showing there is place for improvement on the framework solution side. Besides that, a percentage of 56% developers don't consider they are prepared with suitable test collections in order to perform secure software development. It was concluded that the knowledge on the software security side should be increased, this thing being able to be achieved by improving or upgrading the software solutions, increasing the knowledge level on different type of roles inside the teams and last but not least by following secure coding standard and performing secure code scanning [2].

Another group of Finnish researchers approached the software security topic in the context of different management methodologies. They observed that security activities are usually present in the business requirements, and they are performed at the initial stages of the development process. Although security engineering practices were initially challenging, as time grew by, also the knowledge, documentation and improvements in this field did. The most popular management technologies present on the survey were Scrum and Kanban, while the most common positions were developer and software architect. This study highlights the fact that security requirements are important for performing secure development, on the second place coming the software architecture and design, among with the development guidelines and secure code standards. Furthermore, software safety and security seem to be beneficial for the Agile way of working, although they may require compromising the flexibility on some ends [3].

Another study based on static code analysis topic presented some popular coding standards and software solutions which can improve the software security level of a project. They have followed the standard code review process, but they have improved it by adding several iterations to it, those iterations being based on the feedback of the reviewer. They have used some performant solutions used for static code scanning: Cppcheck, FindBugs and SonarQube, their experiment showing that SonarQube was the best solution for their application, having the greatest level of coverage on scanning the code for different type of vulnerabilities among the 3 solutions [4].

Besides coding standards, software methodologies and software tools for code scanning and vulnerabilities detection, the researchers were interested also in analyzing the software vulnerabilities categories. A survey based on software attacks and vulnerabilities has shown that among the most encountered vulnerabilities within a system they found injection, outdated software and denial of service. These vulnerabilities were detected with the help of static of static code scanning, and they can be further analyzed and mitigated by following the secure code practices and standards [5].

The current state of the art chapter presented some of the most important concepts that should be taken into account in order to increase the software security level: risk score, vulnerabilities, static code scanning, software solutions, management methodologies, code standards and others. Having those said, in the following section we will describe in more details the entire life cycle of software development, we will identify and analyze the possible security threats and what can we do to prevent and mitigate the risk.

3. Methodology

3.1. SDLC

SDLC (Software Development Life Cycle) represents the process followed in all software development projects. This process is used by all software development enterprises and it can have different variations depending on the company, project and management methodology. The main stages of SDLC are planning, analysis, design, implementation, testing and integration and last, but not least, maintenance.

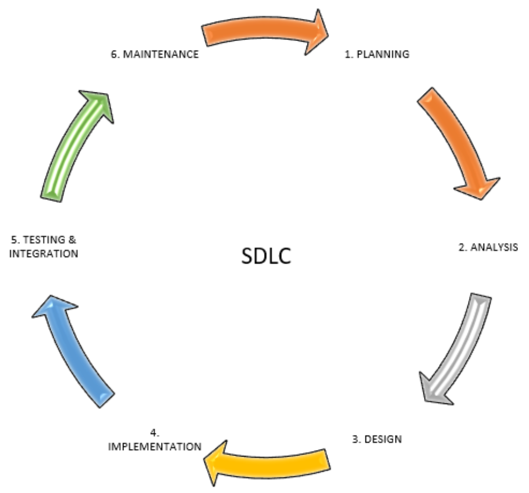


Fig. 1. SDLC stages

Planning is the most important phase of the process; this activity involves defining the initial requirements. The planning should involve both sides, customer (business side) and service provider (development team). Once the requirements are defined, they are further analyzed and refined in the next phase of the process.

Analysis stage involves defining and documenting the product requirements. Once the requirements are clearly defined, the final documentation (SRS – Software Requirement Specification) should be reviewed and approved by the customer.

The SRS document is later used by the architects in the design phase. The architects will propose several design options, building a documentation for it (DDS – Design Document Specification). The document must be reviewed by all the parties involved in the process and based on some KPI's (Key Performance Indicators) the best design will be chosen.

The development phase, as the name already mentions, is the phase were the software development starts, based on the DDS design. The developers will have to follow different category of standards and guidelines, such as coding, risk, security and others.

The testing phase supposes testing the product obtained after the development phase, finding possible issues and fixing them until the product will respect the quality standards that were agreed in the terms. Once the product is developed and tested, it is ready to "go live" and it will be available for the customers. The maintenance stage involves maintaining the product during its lifetime. It may involve bug fixing, adding new features or even improvements. The customer's feedback is very important at this phase [6].

3.2. Software security

As we have described the stages of SDLC, we can move on to software security in the context of SDLC. The software security concept involves protecting the software product during its lifetime against malicious attacks and other possible risks. Secure SDLC refers to the integration of several security checks and scans in the earlier stages of SDLC. With this purpose, the development team should be able to identify and fix the security vulnerabilities by following the security concepts, principles and standards. Some of those concepts are risk assessment, security testing, secure code review and threat modeling [7].

When speaking about security vulnerability mitigation, most of the software development companies follow global standards, trying to avoid the most critical security concerns. One of the most widely recognized list of software vulnerabilities, which gets updated every three years is the OWASP Top Ten Web Application Security Risks. Since web applications are present in a high percentage of software companies, this document provides a strong guideline when it comes to identifying high risk security issues [8].

OWASP (Open Web Application Security Project) is an NGO that targets the improvement of the software security area. The latest updated list from them comes from 2021, this report being based on the work of security experts from all around the world, using a methodology to calculate the Top10 Risk Rating (Fig.2). The risks are ranked depending on the severity of the vulnerabilities, their frequency and their potential impact on an application. The latest version of OWASP Top Ten highlights these vulnerabilities [9]:

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures

9. Security Logging and Monitoring Failures
10. Server-Side Request Forgery (SSRF)

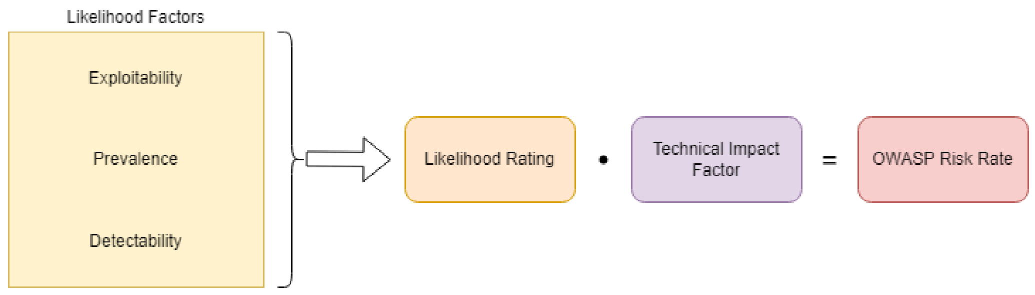


Fig. 2. Calculating OWASP Top10 Risk Rating

The purpose of this list is to offer the software companies that watch closely the OWASP updates an insight into the most common and dangerous security risks, so that the respective companies evaluate these risks in their applications in order to minimize their presence as much as possible. The impact of these vulnerabilities on business and users depends on the type of the attack and data sensitivity, leaving the businesses in financial losses [10].

3.3. Secure code scanning

Code scanning can be performed by a wide variety of tools, most of them trying to identify security vulnerabilities by static code scanning. These tools test the codebase provided against different types of tests, and then highlight the vulnerabilities found, some of them giving tips in solving and mitigating them. From the list of the most popular tools used for detecting and scoring software vulnerabilities we can count: SonarQube, Checkmarx, Synopsis, Veracode, Raxis and others.

One of the most popular software solutions for secure code scanning is Checkmarx, which is a static application security testing (SAST) tool used to identify flaws and vulnerabilities. This tool is used to analyze the uncompiled source code of an application, finding vulnerable patterns and badly written code within the source. The advantages offered by this solution are [11]:

- ✓ Integration in automatic delivery processes – it can be scheduled to run automatically after each source code change, therefore enforcing the secure coding standards offered by this solution.
- ✓ It is a fast solution, when speaking about finding the critical vulnerabilities of an application, locating the exact code lines with flaws.

- ✓ Cost efficient – by offering the option to detect and remediate issues as they are encountered makes it a strong candidate to be integrated in the SDLC, saving important time and money.

However, there is also a list of disadvantages for this solution, such as [12]:

- ✓ High number of false positives/negatives – this solution sometimes struggles to identify whether the identified security issue is an actual vulnerability or not. This requires human intervention from the developers, as they should try to identify by themselves if that was indeed a real threat or just a false alert.
- ✓ Incapability of reviewing the compiled code, identifying vulnerabilities based on business logic flows – this disadvantage is somehow obvious, as Checkmarx only uses static code scanning to identify security threats.
- ✓ Limited code coverage – since the imported libraries and the configurations used are not supposed to be represented in the application codebase, Checkmarx struggles to locate potential issues that can occur between them.

Having these in mind, in the next section we will present a use case of the Checkmarx solution in the context of scanning the entire codebase of a web application written in Java, integrated in an automated pipeline to trigger the scan, highlighting the vulnerabilities found and the steps taken to fix them in the code of the application.

4. Proof of concept

In order to demonstrate the applicability of the security concepts presented above, and also the automatic finding of software vulnerabilities, we have integrated the Checkmarx tool in a build pipeline of an application for scanning the entire codebase of a Web application that consists in a wide set of files and a huge number of line of codes, mostly written in Java (around 95%), and JavaScript (around 5%).

A build pipeline represents a set of automated jobs to be run on certain conditions (for example, triggered when the source code of the application is changed). In our case, the pipeline created consists in 2 steps (Fig. 3), **Build**, where the application is packaged and **CheckmarxCodeScan**, where a new Checkmarx scan is triggered (this scan can take even hours to be finished, depending on the number of lines of code scanned, so the pipeline build will not wait for it to finish, it will display ‘success’ if the build has been submitted successfully). The Checkmarx step is not available by default in Azure DevOps (like the Build stage), so we designed it to copy the packaged application from the first step and upload it to the Checkmarx platform, triggering an automatic scan of the code. The report will be later on available on an email or directly on the Checkmarx platform.

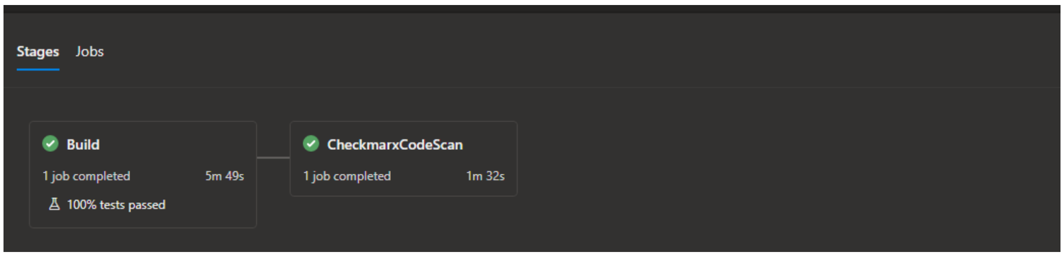
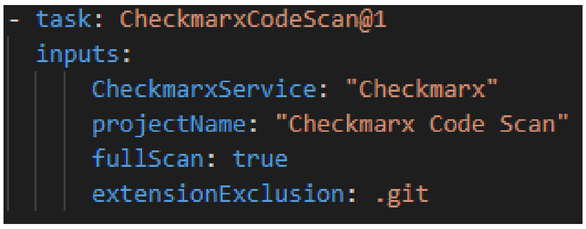


Fig. 3. Pipeline run for a Checkmarx scan

The pipeline has been created using Azure DevOps, which is a CI/CD (Continuous Integration and Continuous Delivery) tool, and it will run on each change of the application's source code. The task created for the Checkmarx pipeline integration can be found below:



```
- task: CheckmarxCodeScan@1
  inputs:
    CheckmarxService: "Checkmarx"
    projectName: "Checkmarx Code Scan"
    fullScan: true
    extensionExclusion: .git
```

Fig. 4. Checkmarx task

The vulnerabilities found in the Java code were on High, Medium, Low and Info categories, and on the JavaScript code, no High vulnerabilities were detected (Fig. 5). In the Fig. 6, we can see that almost half of the vulnerabilities were classified as Info, in addition to almost 39% classified as Low. Therefore, only around 11.5% of the vulnerabilities found were targeted by a more in depth analysis of the code.

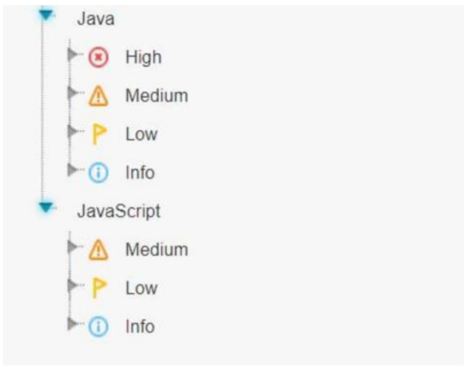


Fig. 5. Vulnerability categories detected

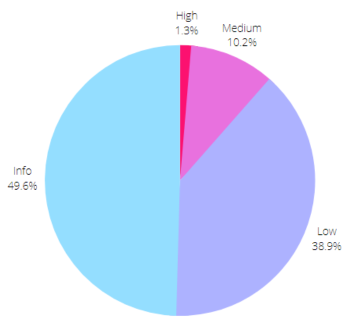


Fig. 6. Checkmarx scan results

Based on these results, we made a top 3 for High and Medium vulnerabilities detected, as these were considered by us to considerably increase the risk score of the software application. These being said, in Fig. 7 and Fig. 8 we can see the illustration of that, and we can also notice that one of them, which was the most encountered High vulnerability (Injection), with roughly around 90 occurrences, is one of the Top 10 OWASP, being number 3 on that list.

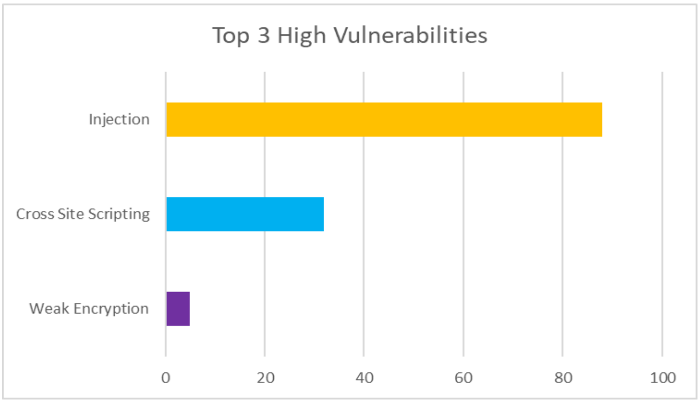


Fig. 7. Top 3 High Vulnerabilities

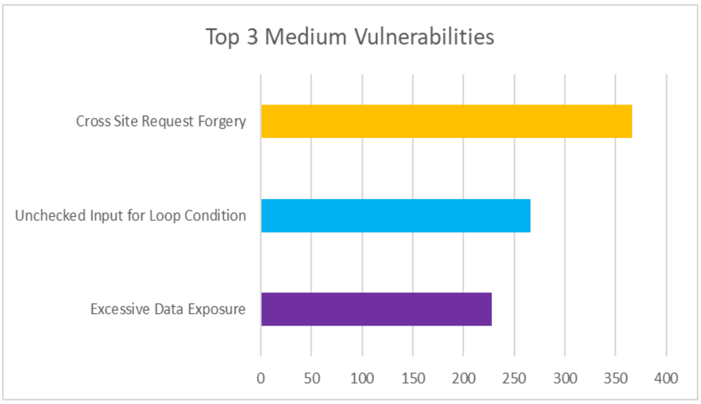
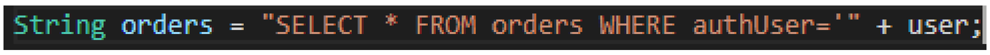


Fig. 8. Top 3 Medium Vulnerabilities

As a proof of concept, we made an analysis for the Top 3 High and Medium Vulnerabilities found, in order to conclude if they were false positives or real threats (with fixes provided). Our analysis started with the High vulnerabilities (Injection, Cross Site Scripting and Weak Encryption). The presence of High vulnerabilities in an application should always be worrying, as these could impose critical security threats to it. For the Injection category, we didn’t find any false positives. Checkmarx detected wrongly constructed Java SQL statements, by using String concatenation, which is a very bad practice, since the String provided can be also an SQL statement that could be interpreted and executed. An example of before and after fix can be found below (Fig.9):

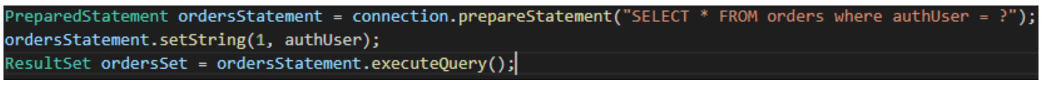


```
String orders = "SELECT * FROM orders WHERE authUser='" + user;
```

Fig. 9. SQL Injection code example

In the code snip from above, in order to retrieve the orders of an user authenticated in the application, an SQL statement is executed, retrieving all the order records based on the authenticated user id. But, for example, if an attacker provides **123’ OR 1=1**, this will result in the following SQL query: **SELECT * FROM orders WHERE authUser=’123’ OR 1=1**. The modified query will return all the records from the orders table, where the authenticated user id is 123 or when 1 is equal to 1. Since 1 is always equal to 1, this query will return all order records from the table.

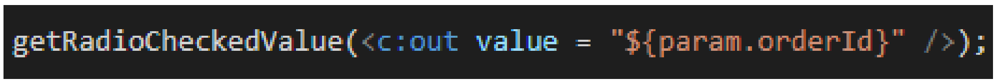
The fix for this vulnerability would be the usage of prepared statements instead of string concatenation, as this will force the user input to be interpreted directly as literal, not allowing SQL interpretations anymore (Fig. 10)



```
PreparedStatement ordersStatement = connection.prepareStatement("SELECT * FROM orders where authUser = ?");
ordersStatement.setString(1, authUser);
ResultSet ordersSet = ordersStatement.executeQuery();
```

Fig. 10. SQL Injection fix

For Cross Site Scripting category, we did not find any false positives, and most of them were related to user input embedded straight in the output, without any validation, like in the example from below (Fig. 11):

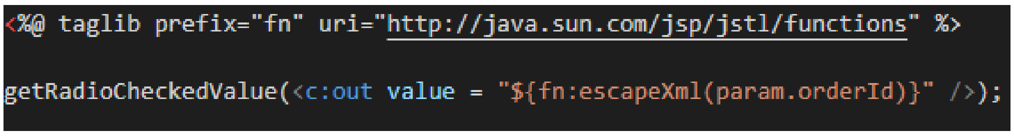


```
getRadioCheckedValue(<c:out value = "${param.orderId}" />);
```

Fig. 11. Cross Site Scripting example

In this case, an attacker would be able to execute scripts in the user’s browser, by inserting malicious code between <script> /* malicious script code </script> tags. This will directly reflect in the user’s browser, and in most of the cases the user will pretend that this is the intended functionality of the application.

Since these vulnerabilities were found in .jsp files (Java Server Pages), these being the frontend part of the application, taking user input from the Java code, one of the best approaches for this situations would be input sanitization using JSTL escapeXml function (Fig. 12), this preventing the user input to be interpreted, treating it strictly as a literal.

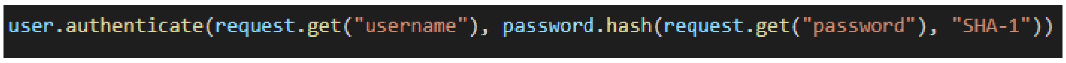


```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

getRadioCheckedValue(<c:out value = "${fn:escapeXml(param.orderId)}" />);
```

Fig. 12. Cross Site Scripting fix

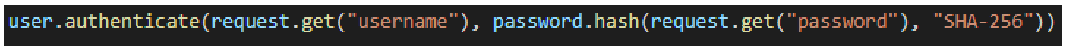
For the Weak Encryption vulnerability, Checkmarx detected the usages of any SHA algorithms weaker than SHA-256 (Fig. 13).



```
user.authenticate(request.get("username"), password.hash(request.get("password"), "SHA-1"))
```

Fig. 13. Weak Encryption example

In order to fix this problem, we replaced in the code the usage of SHA-1 algorithm with SHA-256, and we also updated the SHA-1 encrypted passwords stored in the database with SHA-256 encrypted ones, to validate the new hashing algorithm.

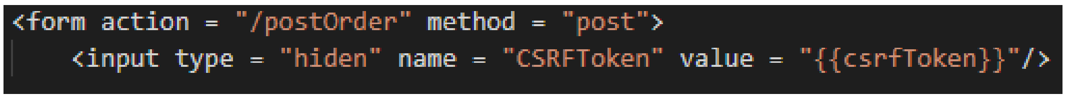


```
user.authenticate(request.get("username"), password.hash(request.get("password"), "SHA-256"))
```

Fig. 14. Weak Encryption fix

Compared to High vulnerabilities, where we didn't detect any false positives, in the Medium findings categories we encountered quite a high number of false positives in our analysis.

For the Cross Site Request Forgery category, most of the problems that we found in the code were related to the fact that an attacker could create a copy of a page and use it in a phishing attack to steal user's cookies. This mostly encounters when the application executes 'POST' requests, and an attacker intercepts the cookies of a user and executes requests on its behalf. The fix that we considered was the addition of a CSRF token as hidden input in the POST requests (Fig. 15), generated using a call to the Java code. That would not allow an attacker to copy the HTML page, as it will not have the csrf token on the copied page.



```
<form action = "/postOrder" method = "post">
    <input type = "hiden" name = "CSRFToken" value = "{{csrfToken}}"/>
```

Fig. 15. CSRF example

In the case of Unchecked Input For Loop Condition, Checkmarx highlighted all the repetitive instructions in the code that could go on an infinite loop execution. However, many of them were false positives, as the code relied on database side parameters when defining the loops. The possible issues that we found were in the case when the loop was constructed based on user input provided numbers. In this case, an attacker could try to input huge numbers in order to slow down the application or even break it down. In these cases, we just added a limit to the user provided numbers, to prevent the occurrence of this vulnerability.

In the case of Excessive Data Exposure, Checkmarx highlighted all the attributes that were included in the session by the application. However, after our analysis, we concluded that this is the intended functionality of the application, as these session attributes are later used in the application, and we found only false positives on this category.

Other than that, we also computed the execution time of the scanning done by Checkmarx, it took roughly about 7 hours to scan the entire codebase of the application and create the vulnerability report. The uncompiled code was scanned against all the vulnerabilities defined on the Checkmarx application, so this is one of the key performance factors when it comes to execution time. If it gets updated in the future with more vulnerability categories, the performance times will surely increase. Furthermore, if the codebase of the application increases, this will also bring up the scanning time of the code, even though the new code is written in a secure way or not.

5. Conclusions

This paperwork highlights the importance of software security integration in the SDLC. Software security guidelines, such as OWASP Top 10, are raising awareness of the main software security vulnerabilities and their impact. Software security vulnerabilities can cause serious damage in terms of time, money and other resources, their prevention using automated code scanning tools being a very useful solution. Using such tools, not only we can detect the actual vulnerabilities that an application has, but we will also be able to determine potential vulnerabilities that might appear from the future updates on the software code.

The software security processes should be implemented in the early stages of the SDLC, in order to reduce the eventual time and/or financial impact that the security threats from the code level could have on the software product.

We have proved the applicability of Checkmarx, by integrating it in an continuous delivery tool, in order to scan the code before it gets released, and we also provided an analysis over the detected vulnerabilities. The results of our experiment have shown which were the most frequent encountered vulnerabilities in our application: Injection, Cross Site-Scripting, Weak Encryption (High) and Cross Site Request Forgery, Unchecked Input for Loop Condition, Excessive Data Exposure (Medium). From the High vulnerabilities category, almost all of them (more than 95%) required a fix in the code, while for the Medium vulnerabilities category, less than 60% required a fix in the code.

In our case, if the application would have been released, for example, with the Injection vulnerability, without it being detected by Checkmarx and later on fixed in the code, the potential risk could have been huge, if an attacker would have detected and exploited it. By scanning the application for vulnerabilities, identifying, analyzing and later on fixing the potential issues, we can reduce the risk score of the application, preventing and protecting it from malicious attacks.

Last but not least, it is a good practice to spread awareness and learn about secure coding when working in software development activities.

A future research for this subject would be the development of a tool that parses the code highlighted as vulnerable by Checkmarx, and based on the category of that vulnerability, it tries to automatically fix the vulnerability, as we found out that some of the code fixes were quite repetitive, and those could eventually be fixed on an automated approach, based on predefined code templates for each category (for example, replacing String concatenated SQL statements with Prepared Statements in the Java code, by detecting the table name and the parameters, and constructing the SQL query in the Java code in the proper and secure manner). This would significantly reduce the time allocated for fixing some of the vulnerabilities, although it would still require at least a quick check from a software engineer in order to assure that the fix has been applied correctly, given the context of the application.

REFERENCES

[1] *S. Semenov, V. Davydov, N. Kuchuk, I. Petrovskaya*, "Software security threat research", 2021 XXXI International Scientific Symposium Metrology and Metrology Assurance (MMA), September 2021.

[2] *S. Dziwok, T Koch, S. Merschjohann, B. Budweg, S. Leurer*, "AppSecure.nrw Software Security Study.", arXiv preprint arXiv:2108.11752, August 2021.

[3] *K. Rindell, J. Ruohonen, J. Holvitie, S. Hyrynsalmi, V. Leppanen*, "Security in agile software development: A practitioner survey", Information and Software Technology 131, 106488, 2021.

[4] *D. Nikolic, D. Stefanovic, D. Dakic, S. Sladojevic, S. Ristic,* "Analysis of the tools for static code analysis", 20th International Symposium INFOTEH-JAHORINA (INFOTEH) Symposium INFOTEH-JAHORINA (INFOTEH), pp. 1–6, 2021.

[5] *H. Chen, M. Pendleton, L. Njilla, S. Xu,* "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses.", ACM Computing Surveys (CSUR) 53.3 (2020): 1-43, 2020.

[6] *S.Z. Iqbal, M. Idres,* "Z-SDLC Model: A New Model For Software Development Life Cycle (SDLC)", International Journal of Engineering and Advanced Research Technology (IJEART) ISSN: 2454-9290, Volume-3, Issue-2, February 2017.

[7] *A.H.A. Kamal, C.C.Y. Yen, G.J. Hui*, *P.S. Ling, F. Zahra* "Risk Assessment, Threat Modeling and Security Testing in SDLC", arrXiv preprint arXiv:2012.07226, December 2020.

[8] *S. Rafique, M. Humayun, Z. Gul, A. Abbas, H. Javed*, "Systematic Review of Web Application Security Vulnerabilities Detection Methods", Journal of Computer and Communications, 2015.

[9] *OWASP,* "OWASP Top Ten Vulnerabilities", owasp.org (accessed March 2022).

[10] *B.N. Matthew,* "Understanding the top 10 OWASP vulnerabilities.", arXiv preprint arXiv:2012.09960, 2020.

[11] *L. Jinfeng*. “Vulnerabilities mapping based on OWASP-SANS: a survey for static application security testing (SAST).” Annals of Emerging Technologies in Computing (AETiC), Print ISSN, 2020.

[12] *M.F. Ramadlan*, "Introduction and implementation OWASP Risk Rating Management", Open Web Application Security Project, 2019.