# Griffin: specification-based RASP approach against SQL injections in MySQL

**Nuno Miguel Gião de Alcácer Bombico**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisors: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Prof. Rui Filipe Lima Maranhão de Abreu

### Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Prof. João Fernando Peixoto Ferreira

**January 2021**

Para a minha família...

# Acknowledgments

I would like to thank Professor Rui Maranhão and Professor Pedro Adão for the help provided throughout this work and Instituto Superior Técnico for the opportunity to be able to work in this thesis. I would also like to thank my family for the support and motivation provided throughout this five years studying on Instituto Superior Técnico. Some finals words for my friends, for the good times and help provided during those years.

# Resumo

O Griffin foi criado com objetivo de proteger aplicações web de Java contra ataques em bases de dados MySQL, usando as especificações da aplicação. O Griffin foi criado numa ferramenta Runtime Application Self-Protection, openRASP, em que se alterou a forma de detectar injeções SQL. Como as injeções de SQL funcionam alterando a estrutura pretendida pelos programadores das queries enviadas, o Griffin aprende as estruturas das queries legítimas que são enviadas para a base de dados para detectar quando estas são alteradas. O Griffin funciona em duas fases: aprendizagem, onde coleciona as especificações pretendidas pelos programadores em relação às queries enviadas pela aplicação e deteção, onde usa a informação recolhida para detectar alterações na estrutura, verifica se o input dos utilizadores vai de encontro às especificações da base de dados e verifica se os valores estão de acordo com as especificações dos programadores. O Griffin foi testado contra a deteção do openRASP de injeções de SQL, para verificar o tempo que adiciona aos pedidos e a precisão na deteção de ataques. Também tem uns testes para averiguar o tempo adicionado na fase de aprendizagem. Os testes demonstram que o tempo adicionado pela aprendizagem é razoável, mas que a fase de deteção adiciona mais tempo aos pedidos do que o openRASP. O Griffin detetou todas as injeções que têm sucesso, ao contrário do openRASP, que nos testes efetuados sobre aplicações não bloqueou 83 injeções. Este número não conta com a deteção de especificações mal implementadas, que o openRASP não faz deteção.

**Palavras-chave:** Injeção SQL, base de dados MySQL, Runtime Application Self-Protection, aplicações web de Java

## Abstract

Griffin is a solution that protects Java Web Applications against SQL injections in MySQL databases by using the applications' specifications. It is built on top of openRASP, which is a Runtime Application Self-Protection tool, replacing its SQL injection detection. As SQL injections work by altering the regular structure of the queries issued to the database, Griffin learns the structures intended by the developers of the SQL queries that are issued to the database and uses that information to identify when malicious users change the structure of the SQL query issued. It works in two phases: learning, where it gathers the specifications intended by the developers, and detection, where it checks the structure of the query being issued, checks if the user input matches the database schema, and also checks if the user input is according to the developers' specifications. Griffin is tested against openRASP's SQL injection method regarding added overhead to regular requests and accuracy in detecting SQL injections. There are also tests used to measure the overhead introduced in the learning phase. It shows that the added overhead in learning is not significant, but adds a higher overhead in the request process time than openRASP. However, when measuring the accuracy, Griffin beats openRASP by detecting all tested injections that would be successful, unlike openRASP, which in total did not block 83 injections. This number does not count with the detection of badly coded specifications, which openRASP does not detect.

**Keywords:** SQL Injection, MySQL Database, Runtime Application Self-Protection, Java Web Application

# Contents


Acknowledgments . . . v

Resumo . . . vii

Abstract . . . ix

List of Tables . . . xiii

List of Figures . . . xv

**1 Introduction** **1**

1.1 Objectives . . . 3

1.2 Contributions . . . 3

1.3 Thesis Outline . . . 3

**2 Background** **5**

2.1 Types of vulnerabilities . . . 5

2.1.1 Injection Vulnerabilities . . . 5

2.1.2 Business Logic Vulnerabilities . . . 8

2.1.3 Session management vulnerabilities . . . 8

2.2 Protection Mechanisms . . . 9

2.2.1 Software development life cycle approaches . . . 9

2.2.2 Protection techniques taxonomy . . . 10

2.3 Runtime Application Self-Protection . . . 11

**3 Related Work** **13**

3.1 Business logic flaws approaches . . . 13

3.2 Protection against SQL injections . . . 14

3.2.1 Alakazab and Khresiat . . . 14

3.2.2 R-WASP . . . 15

3.2.3 CANDID . . . 15

3.2.4 AMNESIA . . . 16

3.3 Runtime Application Self-Protection solutions . . . 16

3.4 openRASP . . . 17

3.4.1 OpenRASP's SQL Injection detection . . . 18

4 **Implementation** **21**
4.1 Griffin's general description . . . . . 21
4.2 Learning phase . . . . . 21
4.3 Detection phase . . . . . 24
4.4 Summary of differences between Griffin and openRASP . . . . . 26
4.5 Implementation details of Griffin . . . . . 27
4.5.1 Specifications intended for the queries . . . . . 27
4.5.2 Learning phase details . . . . . 29
4.5.3 Detection Phase details . . . . . 29

5 **Results** **35**
5.1 Test setup . . . . . 35
5.1.1 Overhead in requests . . . . . 36
5.1.2 Accuracy . . . . . 37
5.1.3 Learning overhead . . . . . 38
5.2 Results and discussion . . . . . 39
5.2.1 Results for overhead in requests . . . . . 39
5.2.2 Results for accuracy . . . . . 41
5.2.3 Results for learning overhead . . . . . 47
5.3 Summary . . . . . 48

6 **Conclusions** **49**
6.1 Future Work . . . . . 50

**Bibliography** **51**

# List of Tables


4.1 Differences between openRASP's SQL Injection detection and Griffin's . . . . . . . . . . . 26

5.1 Web applications being tested . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.2 Java Vulnerable Lab detailed injections results . . . . . . . . . . . . . . . . . . . . . . . 42
5.3 Verademo detailed injections results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.4 Secure Milk Carton detailed injections results . . . . . . . . . . . . . . . . . . . . . . . 43
5.5 Insecure Bank detailed injections results . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.6 Security Shepherd detailed injections results . . . . . . . . . . . . . . . . . . . . . . . . 44
5.7 Authentication bypass test results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.8 Extra tests results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

# List of Figures


2.1 Vulnerabilities categories and their attacks, as seen in Deepa and Thilagam (cf. [4]) . . . 6
2.2 Types of SQL injection attacks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 Security in SDLC according to the phase, as seen in Deepa and Thilagam (cf. [4]) . . . . 9
2.4 Categories to identify protection mechanisms . . . . . . . . . . . . . . . . . . . . . . . . . 11

4.1 Griffin's learning phase . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 Griffin's detailed detection phase for non prepared statements. Blue means it was in openRASP and was changed, green means it is new . . . . . . . . . . . . . . . . . . . . . . 33
4.3 Griffin's database schema check for prepared statements. Blue means it was in openRASP and was changed, green means it is new . . . . . . . . . . . . . . . . . . . . . . . . 33
4.4 Griffin's specification check for prepared statements. Blue means it was in openRASP and was changed, green means it is new . . . . . . . . . . . . . . . . . . . . . . . . . . 34

5.1 Results for Java Vulnerable Lab . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.2 Results for Verademo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.3 Results for Secure Milk Carton . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.4 Results for Insecure Bank . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.5 Results for Security Shepherd . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.6 Results observed for the learning phase in Java Vulnerable Lab, Secure Milk Carton, and Insecure Bank . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.7 Results observed for the learning phase in Security Shepherd . . . . . . . . . . . . . . . . 48

# Chapter 1

# Introduction

Security is a very important topic as millions of people use web applications that store private information every day, making them a desirable target for attackers.

When developers are creating web applications, it is important that they take security into account by patching common vulnerabilities. However, protecting a web application is a tough task because not only it is impossible to develop applications that are one hundred percent secure, but also there is the trade-off between functionality and security. Because of this, attackers could explore vulnerabilities that are in the code, and that could impact the service provided by the web application, for example, via a denial of service or data breaches, or even impact the companies monetarily or causing them to lose reputation. Deepa et al. [1] mentioned a couple of numbers that are important to realize the importance of cyber security: 75% of legitimate applications have unpatched vulnerabilities and during 2015 there were reported one million attacks against web applications.

A big target of attackers is databases, as they store information that is private and should not be exposed to regular users. With the big number of vulnerabilities found in web applications, the databases could be left exposed and subject to attacks to retrieve or tamper with the private information stored in them. This is especially dangerous in legacy applications, as they can have vulnerable code that cannot be patched, leaving vulnerabilities to be explored. In Fry [2], it is mentioned that legacy applications have technical debt, as they are applications either launched years ago, lack a build process or pipeline, have limited documentation, or the applications are no longer developed or maintained, but these applications are still critical to businesses, so they are a big risk in case of failure. Another category mentioned that benefits from this approach are applications that are acquired by companies and have limited support.

One way of detecting and preventing attackers from exploiting vulnerabilities is by monitoring the web application during runtime, to detect when they are attacking it and stop them from harming the application. One technology that can do that is called Runtime Application Self-Protection - RASP -, which is used to protect the application during runtime. This technology has access to the context of the requests performed by users, knowing where the input will be used at. As it is a technology that is attached to the application, it knows exactly what is sent to the database, allowing it to scan the queries and protect the data, making it useful at blocking the potential attacks directed at the database. RASP

is ideal for the protection of the applications that have source code that cannot be changed, because it does not need changes in the source code, which means the vulnerabilities that are left in the code can be protected.

The proposed approach, Griffin, which will be available in open-source, is a solution that is designed to protect and guard databases during runtime. The Griffin tool is built on top of an existing open-source RASP solution, openRASP [3], and is designed to protect MySQL databases from SQL injection attacks in Java Web Applications by extracting the intent of the developers in the form of the specifications of queries that are sent to the database. With the assist of openRASP, it catches the queries that are issued to the database and performs a series of checks to validate that the queries are safe and will not modify or expose the database's data. Griffin works in two phases: learning and detection. In the learning phase, Griffin will model every safe SQL structure that is issued by the application to the database, which corresponds to the developers' intent and will be used as specifications. In the detection phase, it will: (1) compare the structures observed to the ones that were learned, (2) check if the parameters that are going to be used are legal and will not make the database launch exceptions, and (3) be able to check every user input and see if they are legal according to specifications that are provided by the developers. These specifications are another feature of Griffin and they allow developers to add information related to the business logic of the application and also check for sub types of the main SQL types, for example, to allow the database to only store integers higher than a certain number. This last feature is useful for MySQL databases in versions below 8.0.16, where the check operation that performs this type of validations is not present. It can also be used to give context to the queries.

The solution will be described and compared to the existing openRASP's SQL injection detection mechanism, and will also be evaluated against it, regarding overhead introduced in the application's regular requests and accuracy in the detection against attacks. Note that Griffin has an additional test to see the overhead introduced while performing the learning phase.

In the obtained results, Griffin added a higher overhead to the regular requests with SQL queries that are issued by the application. In the worst-case scenario, without explicit specifications input by the developers', the highest overhead introduced was 50.88%, while the highest overhead added by the explicit specifications to Griffin's worst case without the explicit specifications was 14.99%. As mentioned, openRASP added a lower overhead, reaching a maximum of 14.51% overhead added to the requests. In terms of detection, Griffin managed to prevent every successful injection from harming the database, while openRASP registered some successful injections. Note that Griffin also blocked a higher number of possible injections. Griffin blocks every query that changes the structure, meaning that it will block payloads that do not provide valid injections for the application.

Moving to the overhead added in executing a set of requests in the learning phase, it was obtained a maximum of 13.68%. In summary, it is possible to observe that Griffin has a higher added overhead to regular requests, compared to openRASP, but managed to detect and prevent more attacks from being successful.

## 1.1 Objectives

Griffin is focused on creating a RASP solution to detect SQL injection attacks on MySQL databases. The thesis has the goal of protecting java web applications that have code that cannot be changed, such as legacy applications, against SQL injections using the specifications of the queries intended by the developers, which are implicit in the structure of the queries issued, using the specifications of the database, and also using explicit specifications provided by the developers. These explicit specifications will have the goal of creating subtypes for the SQL types that exist.

## 1.2 Contributions

With the implementation of Griffin, this thesis will be focused on the following contributions:

- Implementation of a SQL injection detection mechanism in RASP using the intent of the developers via the extraction of implicit specifications that are obtained from the regular structures of the queries issued and the database schema
- Implementation of a mechanism in the RASP solution that allows the creation of subtypes in MySQL databases, by using explicit specifications provided by the developers' of the application. This mechanism is useful for MySQL databases in versions below 8.0.16, where the check operation that performs this type of validations is not present.

## 1.3 Thesis Outline

The thesis will be separated in the following chapters. Background, where it will be provided context on software security regarding vulnerabilities, attacks, and protections against attacks, and describe the concept of Runtime Application Self-Protection. Related word, where it will be showed work in the area of detection using specifications, detection of SQL injections, and Runtime Application Self-protection, mentioning in the end how the base openRASP solution works. Implementation, where it will be described how Griffin works and how it compares to the base openRASP SQL injection detection. Results, where the two solutions will be compared according to the tests performed. Conclusion, where the thesis is concluded and present possible future work.

# Chapter 2

# Background

In this chapter, it will be explained some concepts regarding software security in the following sections. First, it will be described the types of vulnerabilities that exist, then the types of techniques that identify and mitigate the damage that is done by exploring vulnerabilities. Following that, describe the different software development life cycles that exist to see where this solution acts, and finally, introduce Runtime Application Self-Protection.

## 2.1 Types of vulnerabilities

According to Deepa and Thilagam [4], vulnerabilities are flaws in the application that arise from problems while coding, that can lead to serious damage to applications when attacks exploit those problems. The attacks happen due to flaws in input validation, authentication and authorization mechanisms, management of session information, and other bugs that compromise the intended functionality of applications. Input validation flaws allow attackers to inject malicious commands in the application, leading to *injection vulnerabilities*. The flaws in authentication and authorization mechanisms come from erroneous authentication and access-control policies, allowing attackers to access private web pages and perform unauthorized operations, and making the application deviate from its normal behavior when business logic is not enforced, leading to *business logic vulnerabilities*. Lastly, the flaws in session management appear in improper generation and handling of session tokens, allowing the attacker to hijack the session of a valid user by exploring *session management vulnerabilities*. Figure 2.1 shows the vulnerabilities and its attacks. Next, we will look at each category and show the type of vulnerabilities that exist within them.

### 2.1.1 Injection Vulnerabilities

First it will be approached the different types of injection vulnerabilities, and Deepa and Thilagam [4] divide it in *SQL Injection*, *Cross-site scripting (XSS)* and *Other injection vulnerabilities*. As the SQL Injection is more relevant to this solution, it will be explored in more detail.

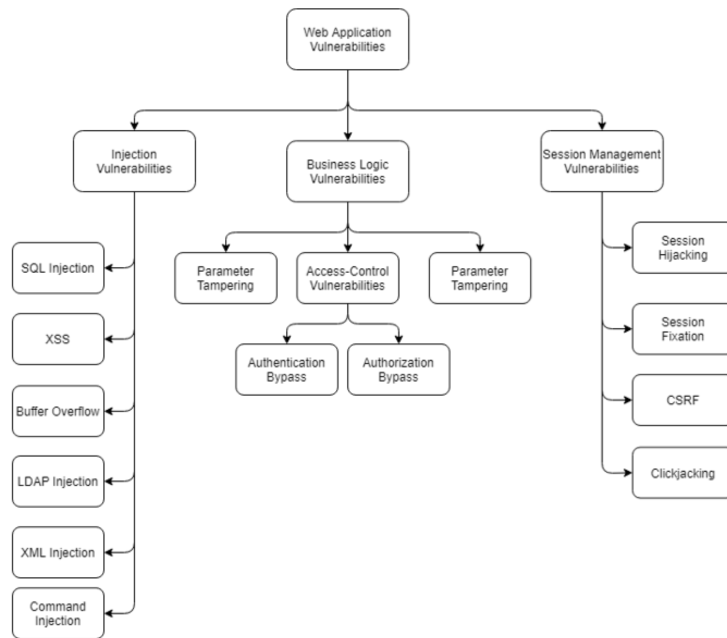


Figure 2.1: Vulnerabilities categories and their attacks, as seen in Deepa and Thilagam (cf. [4])

### SQL Injection

SQL injection - SQLi - is used by attackers to compromise the data that is stored inside a database, either by disclosing private data stored or by changing it. According to Halfond et. al [5] the vulnerabilities can be explored by injections directly through user input, cookies, server variables, or indirectly via second-order injections where inputs will trigger an SQLi attack when used later. Using an example provided by Halfond et. al [5], for the *queryString=”UPDATE users SET password=’” + newPassword + ”’ WHERE userName=’” + userName + ”’ AND password=’” + oldPassword + ”’”* , this query is vulnerable to SQLi because if an attacker sets userName to *“admin’--”*, the tokens “--” correspond to the commentary keyword, meaning that the rest of the query will be ignored. This makes it possible to set any password to admin, as the old password will not be checked.

Halfond et. al [5] also mentions that SQLi attacks can be divided into the following group: *tautology*, *illegal/logically incorrect queries*, *union query*, *piggy-backed queries*, *stored procedures*, *inference*, and *alternate encodings*. The difference is in the payload used, where it changes according to the intent of the attack.

*Tautology* is an attack with the purpose of making the queries issued to the database always evaluate to true and is successful when the code shows at least one result of the query to the users. These attacks are performed with the intent of bypassing authentication mechanisms, extract data from the database, and can identify injectable parameters [5].

*Illegal/Logically incorrect queries* attempts to make the queries raise syntax, type conversion, or logical errors when evaluated by the database. It allows the gathering of important information of the database, typically as a learning step to perform other types of attacks. The attack takes advantage of error pages that are returned by web applications, as the error messages generated may have too much

information that helps attackers identify which injections can be used and be successful [5].

*Union Query* is used to exploit parameters that allow an attacker to change the data set that is returned by queries. By injecting a SELECT statement with UNION, attackers have the freedom to pick what the payload in this statement will execute and decide what information they would like to see. The main goal of this attack is to extract valuable data from the database but can also be used to bypass authentication [5].

*Piggy-backed queries* has the goal of modifying the queries that are issued to the database, making it perform actions that were not the ones expected by the developers. This type of injection is different from others, because in this case the attackers are not trying to modify the original query and instead attempt to execute new queries that will be "piggy-backed" by the original query. This means that the database will not only execute the original query but also could execute any SQL command, making this type of attack have a very harmful effect. This attack is used to extract data, modifying data, perform denial of service, or execute remote commands desired by the attacker [5].

*Stored procedures* is a type of SQLi that attempts to execute the stored procedures that are stored in the database. After determining the database that is being used, the attacker can attempt to execute the stored procedures that are linked to the database, including the ones that interact with the operating system. The authors also mention that developers fall into a false sense of security with the stored procedures, as they think that using them makes the database protected against SQLi. The attacks of this type have the intent to perform privilege escalation, denial of service, or even execute remote commands [5].

*Inference* is an attack where queries are altered to generate true/false answers from the database regarding values that are stored in the database or data about the database itself. In general, attackers use this type of attacks when the application is protected against revealing information via error messages. As there is no useful information from those messages, the commands injected in the database have the purpose of observing changes on the website. There are two well-known techniques to perform inference: *blind injection* and *timing attacks*. In the blind injection technique, the behavior of the page is observed by making the queries be evaluated to either true or false. The pages behave normally when the issued queries are evaluated as true and the pages issuing queries evaluated as false will make the pages behave differently, allowing the attacker to understand the structure without seeing error messages. The second technique, timing attacks, is focused on gaining information based on delays on the response from the database. In this technique, the attacker uses payloads with conditional statements together with delay commands that will have a known amount of time, which will inform the attacker if the condition is true or false. In summary, attackers use inference attacks to identify injectable parameters, extract data, and determine the schema of the database being used [5].

Finally, in *Alternate Encodings*, these attacks focus on modifying the text being injected to avoid the defensive mechanisms and practices that the applications may use. This attack is going to be used together with others, as this will enable the use of the other types of injections. The purpose of this attack is simply to evade detection, as a common defensive mechanism, according to the authors, simply scan for certain "bad characters", like quotes or comment operators [5].

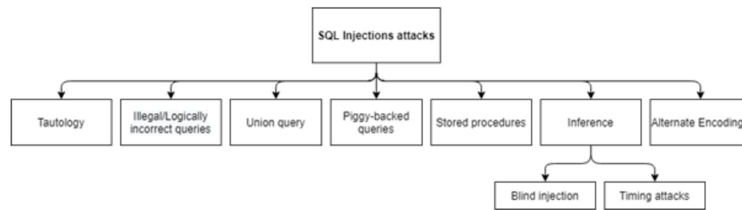


Figure 2.2: Types of SQL injection attacks

The figure 2.2, it can be seen a summary of the different types of SQLi that exist.

### 2.1.2 Business Logic Vulnerabilities

For this type of vulnerabilities, Deepa and Thilagam [4] state that the most common types are *parameter manipulation*, *access control vulnerabilities*, and *application flow bypass*. *Parameter manipulation* allows the attacker to change the behavior of applications, and the attacks are performed by violating the semantic restrictions of the inputs that are provided by a user interface or manipulated in the HTTP request and cookies. This is due to faulty implementations of business logic, enabling the possibility of bypassing client-side validations. Those flaws lead to attacks that are successful because the validate performed server-side of the input allows the attackers to bypass the verifications on client-side. These attacks are known as parameter manipulation/tampering attack [4].

*Access control vulnerabilities* are observed when there is an improper control of access-control policies that allows attackers to be able to access restrict resources. These access-control policies are used to guarantee that users can only access the private information that they should have access to, so failing to enforce them will lead to attackers having access to private information, which should only be accessed by a highly privileged user of the application. The access control vulnerabilities lead to two types of attacks that are possible: *authentication bypass* and *authorization bypass*. In the first type of attack, an attacker gains access to resources that are only available for users that are logged-in, while in the second type of attacks the attacker can obtain resources, for example, directly via URL, bypassing the authorization checks. This last type of attack can also be called *forceful browsing attack* [4].

Finally, in *application flow bypass*, an attacker can bypass the workflow expected from applications, for example, bypassing the payment phase from an order [4].

### 2.1.3 Session management vulnerabilities

According to Deepa and Thilagam [4], these vulnerabilities happen due to improper handling of session variables. Exploring these vulnerabilities lead to the following attacks: *session hijacking*, *session fixation*, *cross-site request forgery*, and *clickjacking*. *Session hijacking* is an attack where attackers steal the session tokens that belong to users, allowing the attackers to perform operations that they should not have access to. In *session fixation*, attackers attempt to elevate their session token to an authorized token, to steal a user's session. *Cross-site request forgery* - CSRF – allows attackers to perform malicious operations on the user's behalf. In *clickjacking*, attackers attempt to trick users into clicking on objects

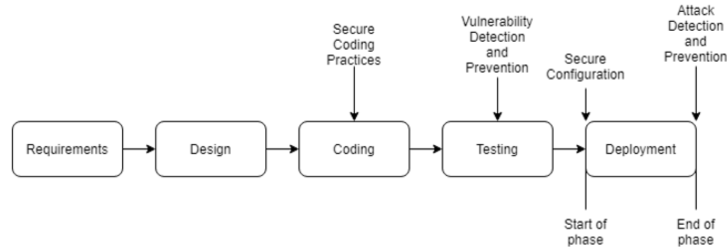


Figure 2.3: Security in SDLC according to the phase, as seen in Deepa and Thilagam (cf. [4])

located in web pages, controlled by the attackers, to perform operations without the user's consent. To finalize, *session fixation* and *session fixation* target the users' session ID, while CSRF and clickjacking target the browser, as both attacks attempt to perform requests and actions that are not desired by the legitimate user [4].

## 2.2 Protection Mechanisms

In this section, a background of concepts and mechanisms will be provided below. First, the approaches according to the software development life cycle and then a taxonomy of the protection mechanisms.

### 2.2.1 Software development life cycle approaches

Another way to characterize protection mechanisms is according to the software development life cycle - SDLC - phase they cover. Figure 2.3 shows a summary of the SDLC phases and the approaches used in them. Deepa and Thilagam [4] mention, from previous research, that security mechanisms should be incorporated during the phases *Construction phase*, *Testing phase* and *Post-Deployment phase*. In *construction phase*, developers should adopt defensive coding practices and guidelines, as these will prevent the application from having critical vulnerabilities. The *testing phase* is used to cover flawed implementations in the previous phase, usually due to lack of knowledge about security. In this phase, to add another security layer, is introduced vulnerable detection mechanisms, that is responsible to identify the vulnerable parts of the code, which should be eliminated to protect the application. The vulnerable detection mechanisms are divided in two techniques: *static analysis* and *dynamic analysis*. In the first one, the application's source code is thoroughly scanned to find vulnerabilities that exist, but it detects many false positives because the code is not analyzed during runtime and fails to detect flaws that are only discovered using dynamic analysis. In dynamic analysis, web applications are tested by using inputs with attacks and analyze how it responds. In this case, it does not generate a high number of false positives, but is not as thorough, as not every case will be tested. Finally, in *Post-Deployment phase*, it can be added protection detection or prevention mechanisms to web applications, where they intercept the requests between client and server to block attacks but note that they add an additional overhead to the web applications [4].

### 2.2.2 Protection techniques taxonomy

To protect web applications against attackers, there are different approaches that can be used to stop and prevent attacks. Prokhorenko et al. [6] provides a general classification for the techniques used to protect web applications. The classification is split into two categories: *subject of observation* and *decision base*. *Subject of Observation* corresponds to what the protection mechanisms are going to analyze and is separated into *inputs*, *application*, and *outputs*, while *Decision base* describes how the protection mechanisms will base their decisions when they need to act and it is divided into *statistics*, *policy*, and *intent*. These subcategories will be analyzed below and it will start with the ones from *decision base*. For *statistics* decisions, it focuses on making decisions based on previous statistics, to solve the problem of zero-day attacks, as the decisions are based on statistics and not on a set of rules, which means that there would be no rules to bypass to perform attacks. These techniques model the application's normal behavior and assume that attacks are abnormal events. To achieve this, two steps are required to implement the solution. The first step is a training phase that is responsible to determine the application's normal behavior. The second step is where the detection phase happens, where the behavior observed is compared to the one observed in the learning phase to detect when there is an abnormal activity. This approach has some problems that can arise while using it. First, if during the learning phase an attack is used, it will be a normal occurrence and it will not be blocked from harming the application. The second problem raised is that the learning phase must be thorough, because it is not desirable that functionalities that are rarely used are seen as attacks. If the web application is changed, it is also needed to update the model to reflect the new changes. The third issue raised is in what is being considered while modeling, because what is not being considered in the model could be exploited to bypass the technique. The final problem mentioned is regarding self-training systems, i.e., systems that adjust their model during runtime, and while it is said is a theoretical problem, it states that there could happen a malicious re-training of the system. Moving to the detection phase, it has the problem when picking the thresholds to compare to the learned model [6].

In *policy* decisions, the detection mechanisms act based on a set of rules created according to previous observations or wishes of protection desired by developers. The rules that are used correspond to attack traits and signatures and are used during runtime while monitoring web applications. While it is a practical approach, the rules may need to be updated often, as new attacks appear. This also means that using this type of protection will always be vulnerable to zero-day attacks, as this type of detection is not able to detect unknown attacks (unlike the previous, that could detect new types of attacks by observing the behavior of the web application changing). Another issue raised is that once the set of rules is known, attackers have the possibility to bypass the detection mechanisms by obfuscating the signature [6].

*Intent* decisions are based on the original intentions of the developers of the application. This is performed based on the specifications of the application itself, but the authors mention that ideally there would be a set of specifications covering the behavior of the application, but as that is rare, the best source is the application's source code. This approach also has two phases, where in the first one the intentions of the developers are obtained from the application's source code (the learning phase) and

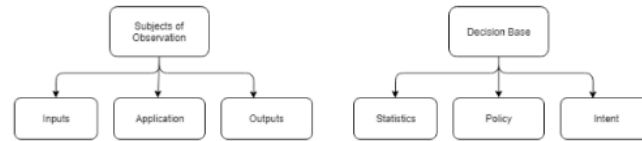


Figure 2.4: Categories to identify protection mechanisms

then during the second step the application is monitored. One relevant example is the SQL structures issued by the application [6].

Now it will be described the *subject of observation* categories, which describes how the protection mechanisms will base their decisions when they need to act. In *input* analysis, the techniques perform continuous input monitoring, as a significant class of attacks is made due to improper input validation. When the inputs arrive at the application, they are analyzed and then handled in case they are attacks. It is mentioned two ways to deal with malicious inputs: drop the transaction or attempt to sanitize it to fix the input. This type of techniques may act even when inputs are not an actual attack, as it may be impossible to predict if the input will correspond to a successful attack [6]. *Application* analysis focuses on the application itself and not on some behavioral aspects. As the thorough analysis of source has a high complexity, the attention is shifted to specific parts of the application. One application of this analysis would be to focus on the code fragments that have user input, applying appropriate sanitization on those code fragments would protect the application from attacks on those inputs. Another focus could be the detection of logic application flaws, by viewing the application as data flow paths. This is relevant because when an application has a big number of possible paths, there is a higher probability of it being missing necessary sanity checks in some paths. With this information, it would be possible to detect attacks based on unchecked user input. The application analysis also carries problems like high memory usage and the assistance of developers being required [6] . Finally, in *output* analysis, the focus of the analysis is the external behavior of web applications. Choosing the external behavior is limited to the interactions that these applications have with external environments. The two most common outputs to look are HTTP request and response pairs, and database access, while there are some fewer common interactions like file system access or generic network access. While observing the outputs may be useful, it is common to pair the output information with the input information. This is due to the problem that output alone is limited to a subset of attacks, as for example, SQL injections that tamper with database data are not observable. Focusing the output on an interaction alone is also not effective, as attacks can be performed in multiple interactions, rendering one interaction useless. This means that output-oriented analysis needs to analyze and monitor multiple interactions at once [6]. Figure 2.4 shows a summary of the categories of the vulnerabilities and possible exploits.

## 2.3 Runtime Application Self-Protection

Now it will be present the technique that will be used in this solution, Runtime Application Self Protection. Runtime Application Self Protection [7] - RASP - is focused on web applications and is attached to

them at runtime, so that it can monitor the web application and detect when the input provided makes web applications behave differently and perform different operations than intended. This approach is used instead of monitoring the inputs and perform decisions solely based on them, and it is possible because RASP works at the application level, so it knows the context where the inputs provided are going to be used. This is the big advantage against other mechanisms such as firewalls, because they only have access to inputs, so their detection capabilities lose accuracy when stopping malicious activities. The requirements mentioned for RASP solutions by Čisar and Čisar [7] are that RASP has to "have the capacity to protect applications by detecting and blocking attacks", "have overview and direct impact on application logic and data flow, configuration, executed operations and data processing to precisely detect attacks", and "be incorporated in the application execution environment. This incorporation ought to be noninvasive or require no/insignificant intrusiveness into application code". Regarding monitoring, the different monitor approaches mentioned for web applications are: *server filters and plugins*, *library/Java Virtual Machine replacement*, and *virtualization/replication*. In *server filters and plugins*, there are plugins that filter requests and check for attack signatures to block possible attacks before executing application code. *Library/Java Virtual Machine replacement* is based on replacing libraries, JAR files, or Java Virtual Machine, and it can learn the application's regular behavior and follow application calls, so it is able to apply rules as the requests are caught. Finally, *virtualization/replication* is focused on analyzing and learning the behaviors of web applications via a virtualized replica of them, and when the learning phase finishes, the rules are synthesized and are connected to the application requests [7]. Fry [2] also mentions another monitoring method, *binary instrumentation*, where the monitoring and control elements are in the application itself via instrumentation, so when it is executed, the monitoring elements identify security events. The advantages and disadvantages mentioned by Čisar and Čisar [7] are the following. The advantages it has more knowledge of the system's operations (regarding application logic, events, and data flow), meaning that it increases the level of security, the protection mechanisms are active with data from creation to destruction, and that it can also help organizations reaching administrative requirements. Some other advantages are that it has negligible false positives, due to the knowledge level mentioned, little maintenance requirements, superior coverage and compatibility, and that it can be integrated with static security testing solutions. On the other hand, the main disadvantages mentioned are that the applications must be individually protected, the performance impact that it has, as the solutions are implemented to monitor the application during runtime meaning that it adds overhead to the normal requests performed, and that it can give a false sense of security, meaning that organizations could create applications that are not secure and use RASP as a defense, but RASP cannot protect against every attack. Other disadvantages include that servlet filters only have access to HTTP, replacement only accesses platform class, and virtualization only accesses lower layer calls.

# Chapter 3

# Related Work

In this chapter, it will be shown the related work in this area. First, it will be introduced approaches that use specifications to detect vulnerabilities in business logic flaws, then present approaches that detect SQL injections, move to existing RASP solutions, and finally, present the concrete tool that this solution is built on top of.

## 3.1 Business logic flaws approaches

Black-box business logic flaws detection means that the detection of the flaws is made without access to the source code of the applications. The following approaches use this technique to protect applications: Swaddler, BLOCK, and SENTINEL.

#### Swaddler

Swaddler [8] is an approach that models the relationships between the internal state and critical execution points to be able to detect attacks in PHP applications. First, it passes by a learning phase that models the internal state of the application by creating profiles for program blocks and then it moves to a detection phase that looks for anomalous states against the learned model, using the program blocks that were learned.

#### BLOCK

The BLOCK tool [9] is designed to target state violation, with PHP application to be the focus of the paper. To do so, it infers the intended behavior between client and application by extracting a set of invariants regarding requests/responses and the associated session variable values during an attack-free execution. After collecting the set of invariants, it uses it to evaluate the requests and responses at runtime. Any request that deviates from the set of invariants that was learned is seen as a potential state violation attack. The implementation is done as a proxy, using WebScarab, so it is placed between the user and the application, having access to all the messages that pass between them and access to the user's session.

#### SENTINEL

The SENTINEL tool [10] is a tool that also checks for business flaws on PHP applications, but this approach is only focused on database logic flaws. The tool models the application as an extended finite state machine and models database accesses by gathering invariants from SQL queries together with session variables, during attack-free executions, to use as the application specification, so that when it sees a query that breaks the invariants learned it is treated as a potential attack. The specification is composed of SQL signatures, which is the junction of the script where the query is declared and a skeleton structure of the SQL query. This structure is constructed to avoid infinite possibilities of different SQL queries. White-box business logic flaws detection means that the detection of such flaws is performed knowing the source code of the application. The Waler tool uses this approach to detect attacks.

#### Waler

The Waler tool [11] tackles logic flaws with an approach that is split into two analysis techniques: dynamic execution to infer likely program invariants and model checking. First, it performs a dynamic analysis to observe and model the application's normal behavior and its specifications, and then with what was learned it filters the specifications to reduce false positives. After getting the specifications, it performs model checking over symbolic input and identifies program paths that have a high probability of breaking the specifications, meaning that there is a type of web application logic flaw.

## 3.2 Protection against SQL injections

In this section, it will be presented approaches that are used to protect web applications from SQL injections. First, it will be described two approaches that use a different technique than the one used in the solution, as these approaches use tainting techniques and not the specifications of the queries. Then, CANDID, an approach where the intents of the developers in SQL queries are considered, by calculating the structure of the query expected and the one obtained during runtime. Finally, it will be presented AMNESIA, an approach that performs the detection of SQL injection attacks by deriving web applications' specifications regarding the queries that are issued with static analysis and then use them during runtime to perform detection. These last two approaches will be explored in more detail, as they have common aspects to what this solution uses.

### 3.2.1 Alakazab and Khresiat

The strategy proposed by Alakazab and Khresiat [12] is an approach that uses negative tainting together with SQL keyword analysis. The negative tainting is performed by storing untrusted markings, that are based on evasion methods. In SQL keyword analysis, it is performed a syntax-aware evaluation of query strings by checking if the input of the query matches with untrusted markings that have characters that

are not trusted. This check is performed by analyzing the SQL keywords and operators. In this approach, the goal and evaluation process were just based on illegitimate access to the database.

### 3.2.2 R-WASP

One other strategy is R-WASP [13], which is a runtime approach for WASP. This strategy is based on the concept of positive tainting, where data is identified and marked as trusted. The propagation of the tainted data is performed by tracking the trust markings by characters and is performed during runtime. One other capability of this approach is to perform an evaluation of the syntax of query strings before being sent to the database, where the strings that contain SQL keywords that are not marked as trusted. In this process, it is used a SQL parser to split the query into tokens and then check each token to identify if they are identified as trusted or not. The proposed strategy mentions that external data sources that should be trusted would be provided by developers, and this approach would treat and mark the data based on those sources.

### 3.2.3 CANDID

CANDID [14], CANdidate evaluation for Discovering Intent Dynamically, is an approach that bases its detection technique around prepared statements and was designed for java web applications. The approach uses a technique that the authors call dynamic candidate evaluation, where the SQL query structures that are intended by the developers are automatically derived, from every location. The authors in Bisht et. al [14] mention that “the approach has two simple but powerful ideas: (i) the notion that the string operations computed on any particular program path capture the symbolic structure of the corresponding programmer-intended query, and (ii) a simple dynamic technique to mine these programmer-intended query structures using candidate evaluations.”. At runtime, the structures of the intended queries are constructed dynamically when a program reaches a location that will issue a query SQL, and they are created from attack-free candidate inputs. These candidate inputs must also reach the same path as the regular input would. The structures derived will be in the form of a prepared statement, where every part of the query string that is not a SQL token is replaced by the placeholder question mark. With the structures, it is possible to verify if the inputs used by users are attacks or not. Going back to the candidate inputs, which are used to create the structure, the authors reach a consensus to use the number one as the input for inputs of type integer and use a string of “a”s with the same size of the input for the type string, as these two candidates are always non attacking inputs and produce the same flow, which is important, as this solution is flow sensitive. To implement this, it is proposed to transform the original application so that it computes, when queries are going to be issued, the value for the real inputs and the candidate inputs, being able to calculate both structures and compare them. For every string in the application, it is created another copy of the original one that will correspond to the candidate variables, where the goal is to copy query strings to have both strings performing the same operations, in order to create a structure from the same string. The arguments that are passed from functions are also duplicated in the code, to not lose the input information needed to

create the candidate inputs. The implementation of this solution can be implemented using source-code transformation, byte-code transformation, or JVM-level implementation.

### 3.2.4 AMNESIA

AMNESIA [15], which stands for Analysis and Monitoring for NEutralizing SQL-Injection attacks, is a tool that is specifically designed for the detection and prevention of SQL injection attacks and uses static analysis and runtime monitoring. The static analysis is performed to extract a model containing the normal queries that are issued by the application. At runtime, it monitors the application and checks the queries issued against the model previously generated. The technique consists of four main steps, where in the first one the application code is scanned to find the parts that issue SQL queries to the database, which are designated hotspots. The next one, for each hotspot, creates a non-deterministic finite-state automaton where the labels correspond to the tokens present in a query SQL. The model is composed by the query string and replaces the part that corresponds to user input with a special token. After collecting the model, it adds calls to the runtime monitor in every hotspot. Finally, it performs runtime monitoring against the model created. In this phase, when an application reaches a hotspot, the string is parsed into a sequence of tokens and checks if the query is part of the model or not. In this process, the matching mechanism performed by the runtime monitor attempts to match tokens with the identical literal value or a label, rejecting when it is not possible to match the query received to the model created. The implementation has some limitations, that were showed via evaluation that were not relevant (generated no false positives and negatives), like if the developer creates the query strings by combining hard-coded strings and variables, or that if the analysis does not see keywords as keywords, some false-positives in the detection as they are considered user input. This approach relies on the static analysis accuracy to detect the models, so if the strings are built in a way that it is ignored by the static analysis performed, they are lost when creating the model.

## 3.3 Runtime Application Self-Protection solutions

In the RASP solutions that exist, most of them are commercial and take different approaches at finding vulnerabilities at runtime. As seen in the background section, there are multiple ways to implement RASP. Utilizing binary instrumentation, Fry [2] mentions that *Contrast* uses the standard instrumentation API in Java so that it does not change the source code or the Java Virtual Machine. The instrumentation technique allows for the RASP agent to detect and respond to attacks with greater knowledge. Moving to virtualization, *Waratek* [16] uses a compiler-based approach, using just-in-time (JIT) compiler, that does not need to know the source code or force a restart to the application. With this approach, it is possible to correct vulnerabilities and insert security rules, protecting against known and zero-day attacks. Waratek mentions that they have ”state-of-the-art protection against known and unknown attacks without slowing down the application and without false positive”. In this compiler-based approach, it is possible to disable, using JIT compiler, attack vectors commonly exploited as, for example, it is possible

to deny the resources entirely if the application has no need to use them, or allow only specific resources according to the application. With this approach, it does not eliminate the vulnerabilities in the code, but they cannot be exploited. Finally, Waratek also provides support for legacy runtime upgrade. Another approach is by using plugins and Imperva [17], previously Prevoty, uses plugins to protect applications by using a method called Language Theoretic Security (LANGSEC), that will check the payload and as it knows if the execution of a given payload in a specific environment will perform an attack, it is able to block known and zero-day attacks. As this type of solutions are commercial, meaning that not everyone can get them, making this a problem because it makes people look away from RASP solutions when they cannot afford the deals. For this reason, Griffin was built on top of an open-source tool that will be described next.

## 3.4 openRASP

This solution will be built on top of openRASP [3], an open-source tool from Baidu security that is designed to work for PHP and some Java server applications, but the focus is the Java applications. The tool is designed to defend applications against attacks by placing hooks in dangerous calls, for example, database accesses, file writes/reads or file traversal. Before being ready to start the detection, first there is a startup phase that is responsible for the insertion of hooks and configurations. Before loading the bytecode in the Java Virtual Machine, the hooks that are inserted by the openRASP tool with the Javassist framework, so that it will be able to change the java code at bytecode level, meaning that it can add the hook points to every application without needing to dive into the application source code and making changes on it directly. During this bytecode manipulation phase, the bytecode is analyzed by a custom transformer that will decide if the class needs a hook or not, meaning that only classes that need hooks are fed into Javassist. If it needs a hook, then the Javassist will go through each method of the class bytecode and when it decided a method needs a hook, it will inject bytecode at the beginning or at the end of the method. After the injection, the new code is returned to the transformer, so that it gets loaded into the Java Virtual Machine. After the bytecode manipulation part, it initializes configuration information, generates the logs, and initializes the JavaScript plug-in. After this process, it is ready to perform detection. From this point on, when a request reaches a hook, the information about the request is collected and a JavaScript plugin is called to check if it corresponds to an attack with the Rhine engine, that compiles the JavaScript code into java bytecode. When the JavaScript plugin is executed, the response of the plugin dictates what actions should be taken [18]. This approach was picked, as it is the most popular open-source RASP tool. The tool itself has low overhead, which means it is possible to add a new detection mechanism without worrying about working on an already dangerous overhead. Next, it will be presented openRASP's SQL Injection detection in detail, which is what Griffin is replacing.

### 3.4.1 OpenRASP's SQL Injection detection

OpenRASP [3] operates just in runtime detection, having no learning phase. For the detection of attacks, it uses zero-rule vulnerability detection algorithms and the way it works is that when it obtains a query, it sends it to a script that will analyze the query. This script performs three different steps: (1) check if the user input changes the logic of the query, (2) check the query against some specifications that can be customized according to the needs, for example, it can be used to check if there are dangerous method calls, use of hexadecimal, stacked queries, and others, and (3) check the query string issued against the regular expression 'union.*select.*from.*information_schema'. The first two will be described below, as the last one is self-explanatory.

The explanation will be illustrated around the example:

*SELECT * FROM User WHERE username = "a" UNION SELECT * FROM User*

where the attacker sends a payload containing *a" UNION SELECT * FROM User*

The algorithm used for the detection of changes in tokens caused by the user input starts by collecting the values that are input by the user, which can be found in the web request parameters, and for each value it will first perform some basic analysis, like checking if the input has the length higher than a certain value, the default is 8. In this example, one of the values found in the web request is *a" UNION SELECT * FROM User*.

After those checks, it moves on for the next step which is checking if the value is present in the query and if it is, it obtains the index where the value is in the query, gets the size of the value, and splits the query into tokens. Each token has info about where it starts and ends, together with the string that corresponds to it. For this example, the tokens will be "Select", "*", "FROM", "User", "WHERE", "username", "=", "a", "UNION", "SELECT", "*", "FROM", "User". Each token will also know the position where it starts and ends so, for example, the token "SELECT" will have the information that it starts on index 0 and ends on index 5.

With this info, it will now see in which token the value from the input starts and then with the size of the value it will check in which token will that value end, with the information it has about where every token starts and ends. After knowing the token, the value starts and ends on, it will see if those tokens have a distance higher than 1. If they do, means that the logic of the query was changed and if it does not, it goes to the next step. Following the example, the index where the payload starts in the original query, character 'a', is in index 37, which means it starts on the 8th token. Looking at the payload provided, with size 27, the index where it starts plus its size will land on the 13th token. This means that the distance between the tokens is higher than one, which means that it will be blocked.

For the second part of the detection algorithm, where the check against the security specifications happens, the tokens obtained in the previous check are iterated and compared to tokens which would appear in case of said specification. For example, if it is specified to block hexadecimal, it will check each token to see if it starts with "0x".

However, two issues were raised where openRASP's approach was not able to detect some attacks. First, when the user input that was in the http request is changed by the application and is not present in the query issued exactly as in the request, it leads to openRASP not performing the token analysis. The

second is that there is also a problem with the tracking when using prepared statements, because these are ignored, and the application reusing the input in a place where that input is not given by the user will lead to that input never being analysed for token analysis, which means some attacks manage to be successful. In these cases, the second and third step in the detection process covers some attacks, but those rules can be bypassed.

# Chapter 4

# Implementation

In this chapter it will be described how Griffin works and the differences versus openRASP, with a summary of the main differences in the end. After that, Griffin will be described with more detail.

## 4.1 Griffin's general description

Griffin is a RASP security solution for Java Web Applications that is used to protect MySQL databases during runtime. It works on top of another RASP solution, openRASP, and it uses openRASP's hooks to catch queries that are going to be issued to the database, performs security checks to it, and decides whether the query is secure for the database or not.

As SQL injections aim at changing the structure of the regular queries sent to the database, the main idea behind Griffin is learning the structure of the safe queries intended by the developers, so that during runtime it is possible to detect when the structure changes with user input, to detect every type of SQL injection. This means that code that has vulnerabilities and cannot be patched, such as legacy applications, will have a defensive mechanism that is responsible to defend the application without the need to change the code, which is possible using bytecode manipulation.

It works in two different phases: learning and detection. In the first phase, it will create a model containing the specifications from extracting structures of the safe queries intended by the developers for the database to process. In the detection phase, it will analyze, during runtime, each query when it is issued and compare it to the model that was obtained during learning, together with some checks regarding the values that are used in the queries. While Griffin uses a two phase approach, openRASP only performs its detection operations during runtime. A comparison between the features of openRASP and Griffin is presented in table 4.1 of Section 4.4.

## 4.2 Learning phase

In this section it will be described how the learning phase works. Note that openRASP does not use a previous learning phase, so there will be nothing to compare to it in this phase.

Starting with a general view of the algorithm in this phase, after the query issued is caught in the hook:

- Collect the method and class name that issued the query
- Create the structure of the query
- Create the specification with the structure of the query, the method name, and the class name
- Check if the specification is repeated or not
- According to the response:
  - If it is new, write it in a file
  - If it is repeated, ignore

The learning phase is responsible for the creation of a model that represents the normal queries that are sent to the database during the execution of a web application. This phase aims at creating specifications that will cover the variations of normal and secure queries. Each specification consists of the structure of the queries together with the class name and method name that issues queries of that type of structure. This means that the specifications of the queries in the model are going to be the ones that Griffin will accept, so if any query is left out of the model, that query and its variations will be blocked.

The idea is to turn every query into (an abstract) safe prepared statement, so the structures will consist of the regular query, but the user input will be replaced by the placeholder "?". For example, if the query issued is *SELECT * FROM User WHERE id = 123 AND name = "John"*, the query will be transformed into *SELECT * FROM User WHERE id = ? AND name = ?* after the construction of the query. With this approach, it will cover every different correct variations of the structure intended by the developers.

In order to decide when to capture the specifications, it is used the approach of Query Synthesis [19], which is an active learning technique. It is a machine learning technique where the learner requests labels for an unlabeled input space, where the request is typically in the form of queries *de novo* that are generated by the learner. With this labeling mechanism, it is able to make decisions about the labeled data later.

Now it will be described how this phase works, and where the query synthesis is used. In the figure 4.1, this phase is illustrated.

During attack-free executions of a web application, Griffin will intercept each secure query that is sent to the database and, for each query, it will create its structure. After the structure is created, it will extract, from the stack trace, the method and class where this query was issued, and will store it together with the query in a file, if that specification with the structure, class, and method, was not seen yet. For example, if the query issued during an attack-free execution is *SELECT * FROM User WHERE username = "foo"*, the structure obtained for this query will be *SELECT * FROM User WHERE username = ?* where the value "foo" is replaced by ?, meaning that every query with this structure will be

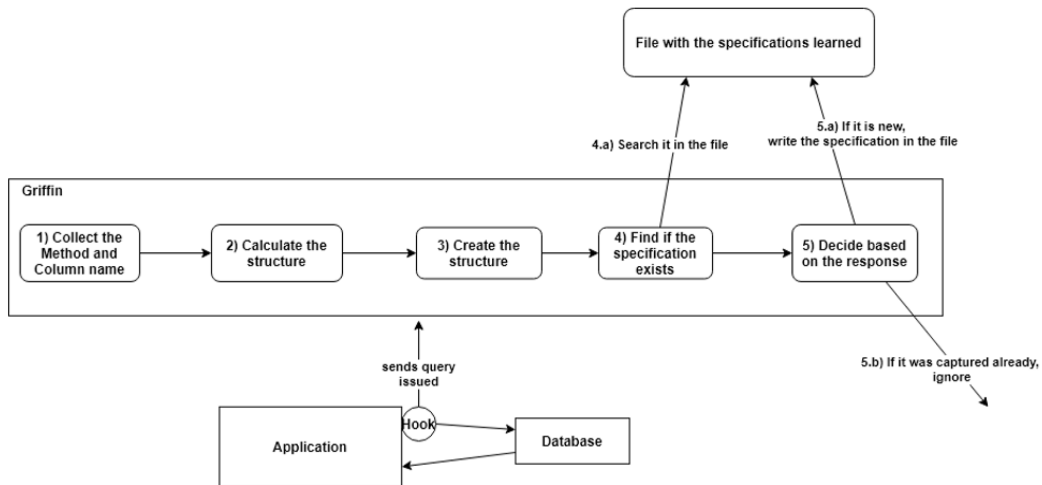


Figure 4.1: Griffin's learning phase

considered secure, and it will be put together with the method and class, for example, method getUser and class User. At this point, the learner has this data and it needs to be labeled, which is where the Query Synthesis comes into play, because the learner needs to know if this specification is new or was observed before, as the final model will not have duplicated specifications. So, when the learner puts the specification together, it sends to an oracle that will label the information as new or duplicated based on the current model learned. If it is new, the new specification is appended to the model, else it is ignored.

In case the application uses prepared statements to issue queries to the database, the query structure is already created, as it will send, for example, *SELECT * FROM User WHERE username = ?*. This means that the learner in Query Synthesis will just create the specification by putting the structure together with the method and class name, consult the oracle to label the specification and append it to the model or not, according to the label.

The final model will be stored in a YAML file, queries.yml, where each YAML document in that file will consist of Class, Method, and Structure of the query. For example, the specifications in the file will be of the following type:

| CLASS: Hobby > 0 | CLASS: Hobby > 0 | CLASS: Hobby |
|---|---|---|
| METHOD: getHobby | METHOD: getHobby | METHOD: getGuestMovies |
| STRUCTURE: SELECT games FROM users WHERE id = ? | STRUCTURE: SELECT movies FROM users WHERE name = ? AND date = ? | STRUCTURE: SELECT movies FROM users WHERE name = ? |

In figure 4.1 it can be seen a summary of this phase. To obtain this model, the application should be exercised using attack-free test suites to collect every possible SQL structure issued by web applications, as long as they cover every structure.

## 4.3 Detection phase

In this section, it will be described how the detection mechanism works and the differences in how Griffin and openRASP's SQL injection detection operate. The detection phase is active during the runtime of the web application and it intercepts the queries that are issued from the application to the database, making sure that the queries the database receives are safe and do not compromise its state. The general idea can be seen in figure 4.2.

In summary, the steps performed by Griffin are the following, after a query is caught in the hook:

- Collect method and class name that issued the query
- Create the structure of the query
- Create the specification with the structure of the query, the method name, and the class name
- Check if the specification is valid, i.e. in the model learned
  - If it is not, block the request
- Check if the values used in the query have the correct type and size
  - If they are not correct, block the request
- Check if the values used are according to the specifications provided by the developers

In comparison, OpenRASP operates by sending the query into a script that performs three steps: (1) check if the user input changes the logic of the query, (2) compare the query against security specifications according to the needs, for example, it can be used to check if there are dangerous method calls, use of hexadecimal, stacked queries, and others, and (3) check against the regular expression 'union.*select.*from.*information_schema'.

The steps that a query passes when a query is received during the detection phase will now be described. After being caught in the hook, the method and class name where the query was issued are obtained and then the query will be transformed into its structure, like in the learning phase. When that information is gathered, it will be searched in the resulting learning YAML file that contains the model of the application, and look if it is present, in order to check that the query is legal. This means that inputs from users that attempt to change the structure of the query will be blocked, as that structure was not expected because it was not used in the learning phase. Revisiting the example used, if it receives the query *SELECT * FROM User WHERE username = "a" UNION SELECT * FROM User*, with the user input *a" UNION SELECT * FROM User*, the structure obtained will be *SELECT * FROM User WHERE username = ? UNION select * FROM User* and this structure was never seen during regular execution, so the request is blocked. For this same query, openRASP would detect that the number of tokens that exist in the query was changed by the user, as the user input introduce the extra tokens "UNION", "SELECT", ..., "User", which have a difference of more than one.

After that, Griffin will check if the values that are used in the query are legal. It takes the information of the parameter (that corresponds to the user input values being analyzed) directly from the schema of

the database, meaning its type and its legal size, for example, VARCHAR(20) says that the type String with maximum size 20. With this information, it is going to be attempted to transform the value obtained to the type expected and then check if the size is legal. If there is a problem with the type or the size, the query is blocked. For prepared statements, as the values are sent separately from the query issued, there is an additional hook that catches the values used, so that Griffin can check them. In this check, the size of the value is compared to the size in the database schema like the regular values, but the type check performed is different. The type of the value issued is obtained by transforming the method that issued the prepared statement value, setXXX, to a set of compatible types that the database accept. For example, if the values comes from a SetString and the database schema for that column expects a VARCHAR, they are compatible. One example of incompatibility is a value being issued via SetString to a column that expects an Int, which is blocked. Note that Griffin does not have every possible compatibility, which is a limitation.

OpenRASP does not evaluate the parameters sent by the users, it just uses them to see if the structure was altered by the users. But openRASP can be used to block requests, when the database launches exceptions.

Finally, there is a check against specifications that are input by the developers. This is a feature that openRASP does not have, as it only uses the user input to search them in the query. Not only that, in the prepared statement usage it does not know the values that are used, while Griffin observes the values being issued in them. These specifications will contain constraints in data inserted in the database and are written in a YAML file, specSchema.yml. For example, the specifications in the file will be of the following type:

| | | |
|---|---|---|
| expression: var1 $>$ 0<br>var1: cost | expression: var1 $>$ 0<br>var1: foo | expression: var1 + var2 $>$ 3<br>var1: Id1<br>var2: Id2 |

To provide a concrete example, for the constraint that the values in the column 'id' must be higher than 10, it can be translated to id $>$ 10. These constraints will be evaluated using a JavaScript engine and its operation eval, and the result will determine if the specifications were broken or not. This feature is not supported by openRASP and is useful for MySQL databases in versions below 8.0.16, where the check operation that performs this type of validations is not present. Running an eval type of operation is usually insecure, but in this case it is safe because the inputs that reach this check were previously checked for its type and size, for example, if someone tries to run a string in an Integer type of column, it will not reach this stage, because the type is invalid for that column. Note that in the case of strings, they will be evaluated as a string and not its content, so they will not execute random commands.

To demonstrate with an example of the detection, if the query caught in the hook is *UPDATE User SET id = 9*, it will be obtained that the parameter "id" has the value 9. For the specification above, it will be evaluated 9 $>$ 10, which is false, meaning that the specification was broken, and the query is blocked. The figure **??** represents a summary of this phase.

## 4.4 Summary of differences between Griffin and openRASP

| Differences | openRASP | Griffin |
|---|---|---|
| **Learning** | • None | • Capture the specification of every unique safe SQL structure in a model (queries.yml) |
| **Detection** | • Perform token analysis to see if the logic was changed by the users, according to the number of tokens created by their input<br>• Scan tokens according to security specifications<br>• Compare the query string issued against a regular expression of the information schema | • Compare the specification of the query issued against the model (queries.yml) that was obtained during learning<br>• Compare the values used in the query to the database schema (schema.sql)<br>• Compare the values used in the query to the specifications of the developers (specSchema.yml) |
| **SQL exceptions** | • Extract SQL error code from the exception message, when the MySQL server is compatible with the JDBC driver. If they are not, it cannot get them<br>• It can block the requests that raise SQL exceptions during this monitoring | • Evaluate values in user input and compare to the database schema in order to predict exceptions that would be raised<br>• It blocks if the value is illegal according to the schema |
| **Prepared Statements checks** | • None | • Checks if the values sent have the size and type according to the database schema<br>• The type check is performed by checking if the Set method used to send the value is compatible with the type of the column in the database schema |
| **Extra specifications checks** | • None | • Check if the values used in the queries are according to the specifications input by the developers |

Table 4.1: Differences between openRASP's SQL Injection detection and Griffin's

On table 4.1, the main differences of both approaches are summarized. It shows that that Griffin performs a learning and detection phase, while openRASP does not, as it performs the checks directly on the query that is issued, without previous knowledge. Regarding the detection steps performed during

runtime, they are different, as Griffin builds a structure from the query observed, compares to the specifications learned and then evaluates the user input. OpenRASP analyzes the query, together with the user input, without using previous knowledge or extra information of the application.

Another difference is in the handling of SQL exceptions, where Griffin calculates if an exception would be raised, while openRASP can monitor the connection between the JDBC and the MySQL server for launched exceptions, when these are compatible. Moving to prepared statements, Griffin performs checks to the values and methods used, while openRASP does not perform any check.

The final difference is the Griffin's ability to perform a specification check to the application, according to specifications written by the developers, keeping the database with valid data by allowing the creation of a subtype, as this type of problems is not a direct SQL injection problem regarding the structure of the query but rather a business logic problem. OpenRASP does not perform this type of check.

Before moving into Griffin's implementation details, another example of the both tools detecting a SQL injection will be provided for the query *SELECT * FROM users WHERE id = 'abc' UNION SELECT * FROM users*, where the malicious payload is *abc' UNION SELECT * FROM users*.

In Griffin, during the learning phase the specification of query was built with the structure *SELECT * FROM users WHERE id = ?*, together with the method and class name that issued it. During the detection phase, the structure that would be generated from the malicious query would be *SELECT * FROM users WHERE id = ? UNION SELECT * FROM users*, which was never seen during the learning phase and it would be blocked.

OpenRASP would collect the payload sent by the users, check if and where the input is in the query, and then split into tokens ['SELECT', '*', 'FROM', 'users', 'WHERE', 'id', '=', 'abc', 'SELECT', '*' , 'FROM', 'users' ]. Note that each token also has information about where it starts and ends in the query. Now it checks in which token the user input starts, 8th token, and then sees that the token it ends is the 12th. As both tokens have a distance higher than 1, it gets blocked

## 4.5 Implementation details of Griffin

In this section it will be described the details behind the implementation of Griffin. Note that there are four types of statements that are considered: UPDATE, INSERT, SELECT, and DELETE.

### 4.5.1 Specifications intended for the queries

In this part it will be described the process of creating the structures of the queries and then the collection of the class name and method name to collect in order to create the specification intended for each query.

First, it will be described the process of creating the structure of the queries. Note that the creation of the structures is only for the cases where the query is not in the prepared statement format, but the rest of the process is the same. When a SQL query is issued, it is caught by a openRASP hook that was modified to send the queries to be analyzed by the Griffin solution.

First, the process of obtaining the method name and class name will be described. Both method

and class are extracted from the stack trace, by ignoring the methods that were called in this tool and methods and classes that correspond to the query being issued. This approach was picked instead of using java reflection to mark every method, because some methods were hidden, for example, in the Java Servlet Pages of the applications. As they were present in the stack trace, this methodology was picked. However, in this approach, when there is Java reflection involved, the method and class will only know that the query was issued via reflection and not the exact method name created using reflection. Although it does not know the method and class name that are correct, Griffin will still block SQL injections.

Moving to the creation of the structures, the algorithm will be presented below, going into further detail after it.

- Split the queries, in case there is a UNION clause
  - The statements in the query will be constructed separately
- Collect the values and check if it a column name
  - If it is a column, ignore
- Get the index of the value
- Construct a new query, substituting the values in the indexes for a placeholder
  - The statements in the UNION clause are put together, reconstructing the query

In both learning and detection, the structure of the query that was issued is calculated. The process starts by collecting the values and the column names where those values are used. The mechanism behind obtaining the values is implemented by iterating through the query and identifying key symbols. For the select, update, and delete that symbol is the equals sign, as it is the token that is used to link the column name to the value, for example *SELECT * FROM Users WHERE id1 = "a"*, and for each equals sign, the column name, value, and the index of the value is extracted. For insert, the values are located inside parenthesis, for example *INSERT INTO Users (id1, id2) VALUES ("a", "b")*, so both column names and values are extracted by guiding through the commas and the parenthesis, where the index where each value is also stored. For each value, it will be checked if it is a comparison between columns, for example, *WHERE id = costumerId*, so that these that are obtained in the learning phase will maintain as part of the structure because it will not be considered as an element that can change. After that, a new query will be built, where the indexes will be used to know where to insert the placeholder. This placeholder will cover every variation of the query, which is important, as it allows Griffin to collect every possible structure of the application instead of needing to capture every single query, which is impossible and not practical.

There is a special clause used in the statements, UNION, that is used to chain multiple queries. To overcome this, the queries are split in the start and the previous operation is performed to each query. After the structure for every separated query is created, they are put together in the regular full UNION

query. When the structure is created, it will be put together with the method name and class name where the query was issued.

### 4.5.2 Learning phase details

In the learning phase, after the learner calculates the specification with the methods described above, it needs to consult the oracle to label the data as new or not. In order to decide, the YAML file is opened and it searches if the specification was calculated before. With this information, it can label the specification properly according to that information. If the specification is labelled as new, it is added to the file, else it is ignored. This phase will end when every possible query structure is obtained. Again, this does not mean every possible query, it just means the structure, without the values. These will be obtained using attack-free executions to collect the structures of the regular safe interactions. In the end there will be a YAML file, queries.yml, which will have the different structures captured by Griffin. The file will be separated by documents and each document will have the "Class", "Method", and "Structure" of a query.

### 4.5.3 Detection Phase details

This part will be separated in the three checks performed by Griffin in this phase, and in the end it will be showed a summary on how the sequence is performed.

#### Check against the structures intended by the developers

In the detection phase, the specification calculated will be searched in the specifications that were collected in the previous phase. The YAML file is previously loaded when the service launching the web applications, in this case Tomcat, is initiated, which means that it does not interfere with the request time. If the specification is present, Griffin will continue with the checks and if it is not, the request is blocked, as every user input that makes a simple change to the structures intended by the developers will be blocked. After that, the user input is analyzed to make sure the values are in the correct type and sized, which will be described.

#### Check against the database schema

This process will be separated between the approaches not using prepared statements and the ones using them.

#### Regular queries - non prepared statements

First, it will be described the process for the approaches not using prepared statements. After the specification is checked, the values are sent to be analyzed in a module. The values are sent together with the column name and the real query that was issued. If the real query issued is of UNION type, the query sent to the analysis is the portion that corresponds to the value being analyzed. To make

the check, the table name used in the query is extracted to search for the specification of the table in the database. For update/insert/delete, the table is extracted directly by iterating the query until the table part, while for the select statements, as it can be an operation between tables, all the tables in the operation are extracted. The specification file of the database, schema.sql, is also loaded when the YAML file is loaded. This file is generated from the database itself via command-line, for example, using mysqldump.

With the schema and the table in hand, the specification for the table can be obtained by iterating the schema until it finds the table. With the schema of the table, it will be searched for the column that corresponds to the value being analyzed. The type of the column data is obtained directly, like other information such as if the data can be NULL. The bounds of the arguments are calculated according to the following. The size for some types is found in the schema of the database, in the column next to the type, like VARCHAR(20) and others have the size implicit in the type, like INT. Griffin will search for the size when they are explicit and, in the others, it uses the default size. There are some cases where the types can have the size explicitly or when it is not present it will use the default size, which Griffin correctly identifies. For the date and time, the size check, in reality, is a check to the format and correctness of a date or time. This is assuming the developer correctly extracts the schema file, so that file will not have wrong formats, for example, VARCHAR without the explicit value.

The algorithm for the check of the correct size and type detection is:

- Check if the query is attempting to put a NULL in a column that cannot be NULL
- Check if the value being evaluated is a possible column, as those do not need to be checked
- Check if the specification of the column is of type Enum
  - If it is of type Enum, the value is evaluated to see if it is a valid value in the Enum
- Check if the size of the value corresponds to the specification of the column
  - This check performs implicitly the type check, as first it will attempt to convert the value in the type the schema expects, deeming it illegal if the conversion is not possible.

Note that the checks for the bound depend on the type, and the types supported for this are subtypes of String, Integers, Decimals, and Dates and Time. If one of the checks finds something illegal, the query is blocked.

#### Prepared Statements

For prepared statements, there was a need to introduce a new hook that catches the prepared statement values that are sent, together with the index which corresponds to which token the value corresponds to. With this new hook when a value arrives it is sent to analysis. As the query is not present, whenever a query arrives it is saved as it indicates the last query that arrived so that when the values are caught, there is the information of the query saved and, in this case, it is stored in a file, paramsCheck.txt.

The check starts by retrieving the query from the file, and if it a UNION query, it splits into single queries. After that, it uses the index to find the column name, and if it was a UNION query, also finds the statement where that value is being used. After that, the schema is extracted and with this information, the algorithm is equal to the described above, with one difference. The type check is performed by analyzing the method that was used to issue the prepared statement.

As in prepared statements the values are issued via a method call according to the type they have, the hook that sends the catches the values will also send the method that was used to send the value, which will then see if it compatible.

#### Check against developers' specifications

If the previous checks are passed, it will be performed the final check, where the input will be analyzed against the developers' specifications. For the specification check, there are also differences between the approach of non-prepared statements and prepared statements.

Starting with the non-prepared statements, obtaining the values and column names is simple as they were all retrieved for the previous checks, so they are sent to perform the analysis described in the above explanation of it.

For the prepared statements, as the values are obtained at a different time there is a need to save the pairs until they are all collected. For that, at the end of the second check described above, the values that come from prepared statements are stored in a file, prepStmtParamValue.txt. Whenever a new value from prepared statements arrives, it is checked how many indexes the query has (i.e. number of question marks, which are the tokens that symbolize where the values will be inserted) and if the current index corresponds to the last value. If it does, then it means that the check is ready, so the file is opened, and the column names and values are extracted and sent to perform the specification check.

The rest of the explanation is common to both prepared and non-prepared statements.

The specification check file is in a YAML format, which is also loaded before like the schema.sql file. In this file, each specification will be in a different YAML document inside the same file, specSchema.yml. This file will consist of the expression, with variables where the column name would be, and the information of what column that variable corresponds to. For example, the specification id $>$ 10 would be split into "expression: var1 $>$ 10;" and "var1: id". This means that the var in the expression will correspond to the column id. Note that this file is created by the developers with the specifications they intend to be checked.

The algorithm for this check will be detailed below:

- Check if one specification needs to be evaluated with the values in the query issued
  - If it does, insert the initialization of the variable with the value
- Repeat the steps above until every specification was checked
  - Create a set of specifications needed to be evaluated
- Evaluate the specifications that were initialized

In more detail, Griffin will check first for the variables and see if the column they correspond to was changed and if it was, it will insert the initialization of the variable. For the example above, with the query *UPDATE user SET id = 9*, Griffin will check if the column id was changed, and as it was, it will introduce the initialization “var var1 = 9;”. After all the substitutions, the resulting expression to be evaluated by a JavaScript engine, also loaded previously with specSchema.yml, will be “var var1 = 9; var1 $>$ 10;” which is false, meaning that the query would be blocked.

During the operation of building the expressions to be evaluated, a list of expressions ready to be evaluated is created, so that only expressions that received at least one change are evaluated. This means that some expressions may be evaluated and be incomplete, as the query does not use every variable in a certain expression, but those expressions will raise exceptions in the JavaScript engine and the result is ignored. The list of specifications will be evaluated, stopping if there is a broken specification.

Note that for each document inside the specSchema.yml file, an expression may have multiple variables. Also, with this configuration, having multiple specifications where the only thing that changes is the column is easier to produce, as it is only needed to copy the document and replace the variable for the desired column, avoiding having to replace in the full expression every occurrence of the variable.

#### Final details

In this part it will be given final details with a summary of the detection phase. In order to reach Griffin's detection, the SQL hook in openRASP was changed to send the information to a different method. The hook sends the information to a handler, that was also altered to guide the detection phase. The handler calculates the structures and collects the values and their respective column name. Then, it sends to a new module that performs the structure check and sends the response back if the request should or not be blocked. After that, the values, the column names, and the query issued is sent to another new module that performs the schema check. This module sends the response and if the request is not blocked, the values and column names are sent to the specification module, also new, which will perform its checks and decide. If a check responds with block, hook handler will stop the checks and block the request without proceeding. Figure 4.2 shows the normal cycle with requests that do not have prepared statements.

For prepared statements, both schema and specification check are different. In the schema check, the values are issued separately from the query, so there was a need to add a new hook to catch those values. When a value is obtained, it is sent to the hook handler that will send the values to the schema check module. The module will check a file to get the last query issued, which where that value will be used and perform the check. Figure 4.3 shows a summary of the explanation. For the specification check, when the hook handler receives values being caught in the hook, it will compare if every value was issued or not, by comparing the index to the total number of positions that will be filled with values in the prepared statement. If it corresponds to the maximum index, it will look in a file for the pairs of values and column names. This file is responsible to keep the values issued by the prepared statement and the respective column name, which is discovered in the database schema check. After that, it will perform the check normally. This can be seen in figure 4.4.

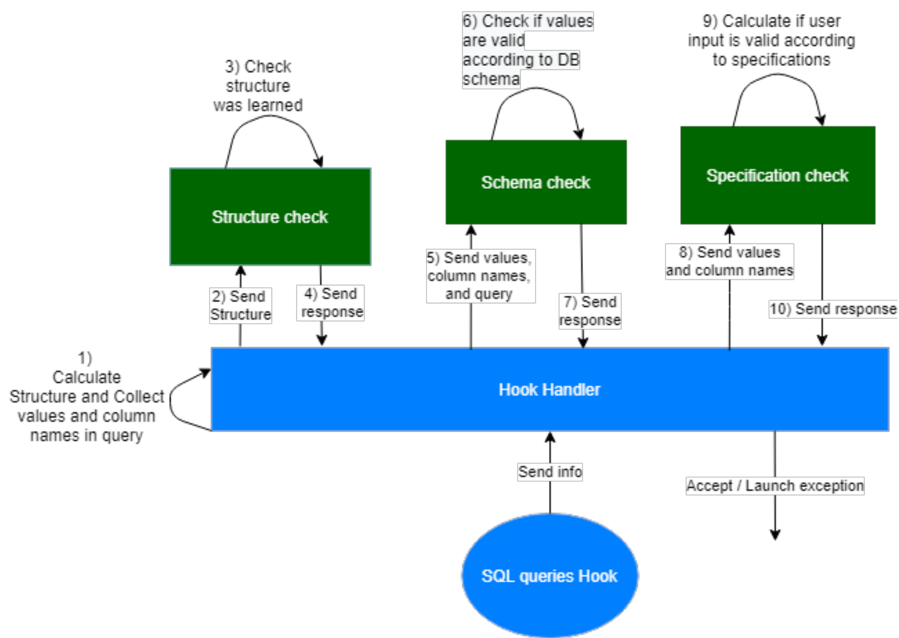


Figure 4.2: Griffin's detailed detection phase for non prepared statements. Blue means it was in openRASP and was changed, green means it is new

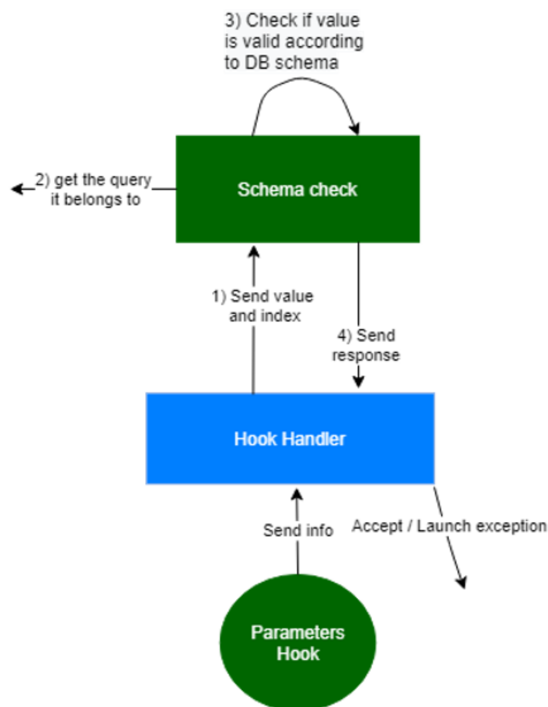


Figure 4.3: Griffin's database schema check for prepared statements. Blue means it was in openRASP and was changed, green means it is new

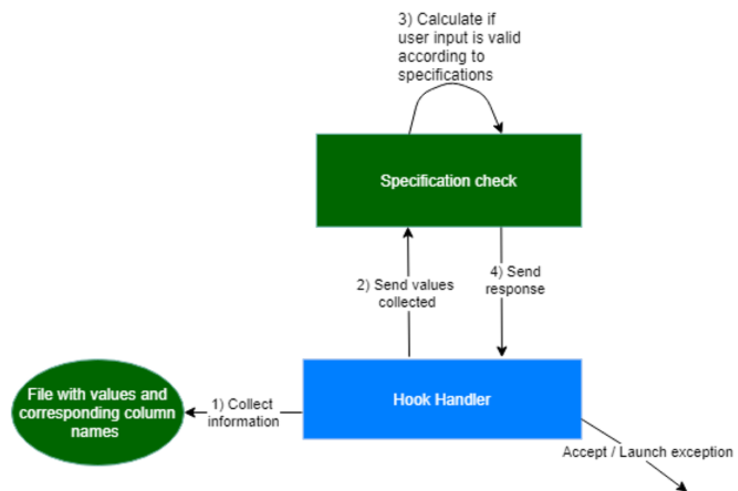


Figure 4.4: Griffin's specification check for prepared statements. Blue means it was in openRASP and was changed, green means it is new

# Chapter 5

# Results

In this chapter it will be shown the test results. First it will be described the setup used to test Griffin, which will be tested against openRASP's SQL injection detection.

## 5.1 Test setup

For the evaluation of Griffin, its performance is compared to openRASP's SQL injection detection and to the applications without any protection mechanism, with the goal of seeing how Griffin compares to the tool that it is built on top of. To perform the tests five open-source vulnerable java web applications that make requests to a MySQL database were picked and tested in three types of tests: added overhead in a normal request, accuracy of the solution, and an extra test that determines the overhead on executing Griffin's learning phase. The applications tested were Java Vulnerable Lab [20], Verademo [21], Secure Milk Carton [22], Insecure Bank [23], and Security Shepherd [24]. The application's characteristics regarding number of lines (generated using David A. Wheeler's 'SLOCCount') and the number of unique queries that each application issues can be seen in table 5.1. Note that the number of unique queries was calculated by running the application with Griffin's learning phase and navigate the application, in order to collect the total number of structures.

The applications tested were selected accoring to the following criteria: (1) is a vulnerable Java Web Application, (2) connects to a MySQL database, and (3) be able to deploy the application in Tomcat.

| Application | #lines | #unique queries |
|---|---|---|
| Java Vulnerable Lab | 2524 | 13 |
| Verademo | 5120 | 21 |
| Secure Milk Carton | 835 | 5 |
| Insecure Bank | 65685 | 11 |
| Security Shepherd | 91306 | 21 |
| Total | 165460 | 71 |

Table 5.1: Web applications being tested

These applications were deployed into Tomcat, version 9.0.37.0 and are connected to MySQL version 8.0.22 via MySQL connector version 8.0.16. The tests were performed in a Ubuntu 20.04.1 LTS Virtual

Machine. Between test runs, Tomcat is restarted to apply the changes that are needed to perform the tests and MySQL is running with every protection mechanism disabled. For Griffin test runs, the files that it needs to work in each different test are generated before the tests and then put in the correct place in order to work properly. Also, openRASP is using all of its types of detection for SQL injections.

### 5.1.1 Overhead in requests

Starting with the added overhead in a normal request, it was created a script, for each application, that makes a simple interaction with the application (and has to send at least one query to MySQL) and it times the amount of time that a request took to arrive. For example, starting a timer, followed by performing a login, and then stopping the timer and collecting the time it took. These interactions are attack-free, simulating the normal usage of the applications, and are repeated for one hundred times and then the mean is calculated. The tests were performed, as stated previously, with the application without any protection, with openRASP, and with Griffin, and the latter has two different configurations tested, which are split in two big groups: with a specification schema and without it.

Starting the configuration without the specification schema, the goal is to measure the overhead that it adds to a normal request from an application, without the specification schema, so that the results only reflect the overhead from the detection of the structure of the query and the schema check. This configuration is then split in three different configurations, as the performance of Griffin varies with the file that is generated from the learning phase, as requests that use queries from the top have a lower overhead compared to using queries from the end of the file generated, as they spend more time searching for the structure that was used. The configuration used are: the request issues a query with a structure of the fifth structure in the file learned, the request issues a query with the structure of the twenty fifth query in the structure, and the request issues a query with the structure of the fiftieth query in the structure. Note that there are cases where a request issues more than one SQL query, so in these cases they are put together in the file. For example, if it issues two queries, they would be fifth and sixth, and so on. Due to limitations of the applications, as they do not have that number of total queries, the queries that exist are repeated until the query that is used in the request is in the place for the test.

The specification schema configuration uses the previous last fiftieth query setup, as using a fixed query number allows the comparison of what is the impact of using the specification schema in these different scenarios alone, which is the goal of this configuration. It is created a specification file for the application with twenty specifications and then is performed three tests: best case, where the query(ies) in the request only affect one specification (the last specification), middle case, affecting half of the specifications (every other one, decided not to do them together to simulate them being a part in the file), and worst case, affecting every single specification.

Lastly, each graph will include two extra pieces of information. First, the time difference between the application without any protection and openRASP, and then to Griffin. The second is the time difference between openRASP and Griffin's different setups.

### 5.1.2 Accuracy

In order to test the accuracy of the solution, i.e. if it detects SQL injection attacks, it was performed tests separated in three parts. The first part is a more specific type of tests, where it analyzes the behavior of the detection tools against a small sample of SQL injections. In the second part, the authentication mechanisms of the applications are injected with a set of injections, where it is obtained the number of injections that had success and how many of the injections inserted were blocked. In part three, exploitable points of the application are injected with two sets of SQL injections. This third part expands the results observed in the first part with a wider range of injections. Note that all of the SQL injection payloads were obtained and modified from PayloadOfAllThings [25] and SQL-Injection-Payloads [26]. The modifications performed are to allow the injections to be relevant to the web applications' injection point, so changes like table names, database, and column names are changed to relevant ones according to each application.

#### Detailed analysis

In the first tests, a set of different cases with SQL injections that were successful against the original application without protection. Part of the injections were performed directly in the application, but the majority was performed via script. The script ones were performed by observing the different http responses. Responses that correspond to code 200 (ok) or 500 (server error) are the normal behavior of the injections being successful, and when the protections are inserted, if a code 302 (redirect) appears, means that the injection was successfully blocked. The set of injections has different types of injections, in order to attempt to test the broadness of the detection mechanisms regarding SQL injections.

The final test cases showcase the specification check performed by Griffin, and to do so, two specifications for the applications were created that were used to showcase differences to openRASP's detection method. These specifications were simulated to be badly coded in the application, so that the original application accepted the requests "wrongly". There was also an attempt to use different types of specifications to show some of the specifications accepted by Griffin. One final note is that the payloads used in this test come exclusively from PayloadOfAllThings.

#### Authentication bypass

In the second set of tests, the authentication mechanism of the applications was injected with SQL injections and the behavior of the base application was compared to the behavior of the application with the protection tools. It was collected the number of successful authentication bypasses and the number of injections that were blocked by each tool. To do so, it was created a script that sends a login request with the injection payloads and the responses were observed. In this case, it is easy to see if the payload was successful, as the responses had characteristic words, like "welcome". So when the characteristic words were observed in the response, it was counted as successful. If the response had a different behavior than the normal responses, it was counted as a block. The abnormal behavior was responses with either the block information or a connection error, both which are not observed during the normal

interactions with the application. Both these behaviors are not observed when the login fails, which corresponds to a failed attempt to attack the authentication mechanism. This means that it is possible to isolate the behavior between successful injection, failed injection, and a blocked injection. The injection payloads used in the tests are from both PayloadOfAllThings and SQL-Injection-Payloads.

#### Extra tests

For this final part, each application was tested with two different sets of SQL injections being used in vulnerable points. The results were obtained for each tool protecting the application. In the first set, the injection payloads were tuned for each application to contain columns, tables and databases relevant to them. The string token that each point uses is also modified to maintain the queries relevant. For example, if a query is placing the input between quotes, the injections are modified to use quotes. In these cases, it is only tracked the number of injections that were blocked, where the ones not blocked are classified as successful. For the second set, the payloads used were the same used in the authentication bypass, to test these points even further. One final information is that there are two extra cases in the tests. The first one, regarding Security Shepherd, exercised two vulnerable points as it was possible to do so. The second case is in Verademo, where there is one vulnerable point that was not possible to reach with a script, so it was decided to inject five different queries in the openRASP solution and see if Griffin could block them. This was decided as the goal of these extra tests are to see how the tools behave against a wider set of injections, from the results observed in then detailed analysis. The new injection payloads in the first set of tests were modified payloads from both PayloadOfAllThings and SQL-Injection-Payloads.

### 5.1.3 Learning overhead

For the final tests only the original application and Griffin are used, as openRASP does not have a learning phase. The goal of these tests is to see what is the overhead that is added during the learning phase of Griffin.

To calculate the overhead for this phase, for each application is created a script that sends multiple requests to the application and times how long those take, for a certain amount of times and then calculates the mean. These set of requests are simulating small samples of test suites. Four of the applications are timed seventy times and one, Java Vulnerable Lab, is just twenty five, due to limitations regarding SQL connections. The requests will issue SQL queries and Griffin will calculate the structures and create the file with the model, so between runs the file with the model is removed to add the time of creating the file.

## 5.2 Results and discussion

In this section the results obtained will be showed and commented.

### 5.2.1 Results for overhead in requests

The tests performed in this part have the goal to perform a detailed analysis of how both tools react injections, where each application was injected with a small set of injections. Note that it was also added test cases that showcase Griffin's specification schema detection capabilities. Before talking about the results, note that in the figures that will be showed there will be six symbols. Starting with (i), (ii), and (iii), these correspond respectively to the request issuing a query that is fifth on the learned queries list (best case), a query that is twenty-fifth, and fiftieth (worst case). The symbols (a), (b), and (c) correspond respectively to the request issued matching one specification (best case), the last one, matches ten specifications, every other one, and matches twenty specifications (worst case), all. Note that each graph will include two extra pieces of information. First, the time difference between the application without any protection and openRASP, and then to Griffin. The second is the time difference between openRASP and Griffin's different setups.

The results can be found in figures 5.1, 5.2, 5.3, 5.4, and 5.5, where each figure corresponds to each application. Note that Security Shepherd is in another range of values, so its overhead results is in a different scale.

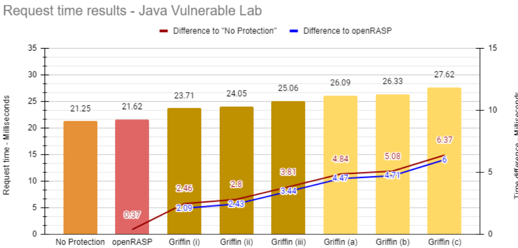


Figure 5.1: Results for Java Vulnerable Lab

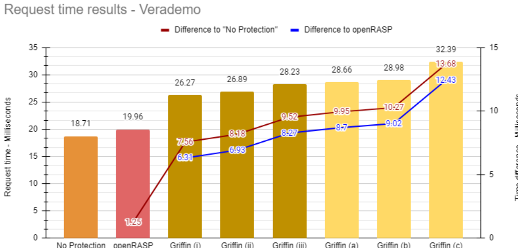


Figure 5.2: Results for Verademo

Comparing openRASP to the base web applications without any protection, openRASP achieves an added overhead that goes from 1.74% (Java Vulnerable Lab) to 15.41% (Secure Milk Carton). In Security Shepherd, it adds 0.03%. In the other two apps, the overhead added is 6.68% (Verademo) and 9.51% (Insecure Bank).

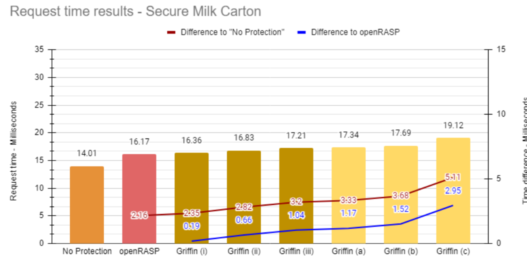


Figure 5.3: Results for Secure Milk Carton

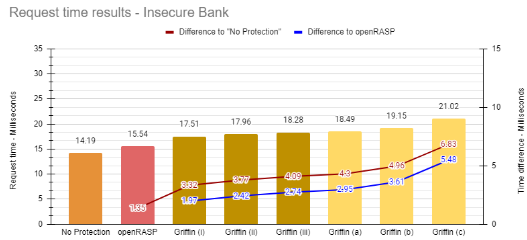


Figure 5.4: Results for Insecure Bank

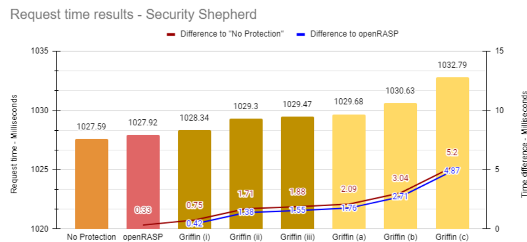


Figure 5.5: Results for Security Shepherd

Between Griffin and the applications without protection, the worst case (setup (iii)) that was captured achieves overheads from 17.93% (Java Vulnerable Lab) to 50.88% (Verademo). In Security Shepherd, the overhead observed was 0.18%. The other apps add an overhead of 22.84% (Secure Milk Carton) and 28.82% (Insecure Bank). The overhead observed in Griffin compared to openRASP is expected, as it performs more operations to check the values of the queries used and not only check the structures of the queries.

Now it will be compared the overhead that was added from introducing the specification schema. In this case, it is compared Griffin's worst case without specification (setup (iii)) to the worst case using specifications, where all specifications are checked (setup (c)). The lowest difference observed was 10.22% (Java Vulnerable Lab), while the highest was 14.99% (Insecure Bank). Security Shepherd has a difference of 0.32%. The other two observed are 11.10% (Secure Milk Carton) and 14.73% (Verademo). This result shows that adding the specification schema checks will not add a noticeable penalty in performance compared to the performance observed without performing this type of check.

Comparing Griffin's best (setup (i)) to worst case (setup (iii)) without specifications, the lowest difference was 4.40% (Insecure Bank), while the highest one is 7.46% (Verademo). Security Shepherd has a 0.11% difference. The other two applications have a difference of 5.20% (Secure Milk Carton) and 5.70% (Java Vulnerable Lab). In this case, it is possible to conclude that adding more queries in a file

will not add an unreasonable overhead, as the difference between best and worst case is the request issuing a query that is deeper in the model, which needs to go through the model that was learned.

Turning to the difference between best (setup (a)) and worst case (setup (c)) of specification schema, the difference goes from 5.86% (Java Vulnerable Lab) to 13.68% (Insecure Bank). Security Shepherd has a difference of 0.30%. The other two differences were 10.27% (Secure Milk Carton) and 13.01% (Verademo). In this one, the conclusion is similar to the above, as the added overhead is also reasonable. This means that adding more specifications has a reasonable penalty, as the difference between best and worst case is the number of specifications activated by the query issued to the database.

Regarding the differences between the tools, openRASP, and Griffin's setups, it show that highest difference between Griffin and the base solution was 13.68 milliseconds, where it is also observed the highest difference between Griffin and openRASP, 12.43 milliseconds. It is seen in Verademo, where, as described above, issues three queries and uses prepared statements, which increases the overhead of the request. The lowest difference observed between the base solution and Griffin was 5.11 milliseconds, in Secure Milk Carton, where it was also observed the lowest difference in overhead between openRASP and Griffin's setups, 4.87 milliseconds.

### 5.2.2 Results for accuracy

As mentioned above, the tests performed are separated in three parts. First, it will be analyzed in detail how both tools respond to a set of injections performed to the applications. After that, it is performed a set of injections for authentication bypass, and finally, it will be attempted to inject another set of different types of attacks to each application.

#### Detailed analysis: Results

The tests performed in this part have the goal to perform a detailed analysis of how both tools react injections, where each application was injected with a small set of injections. Note that it was also added test cases that showcase Griffin's specification schema detection capabilities. In the test results, green corresponds to correct action, while red means wrong. For example if an accept is marked as red, means that the correct behavior is to block. The test results can be seen in figures 5.2, 5.3, 5.4, 5.5, and 5.6.

In Java Vulnerable Lab, Secure Milk Carton, and Insecure Bank the results obtained are equal. Griffin and openRASP detect nearly the same types of injections, but openRASP fails to detect the one that has a string plus comment token ("--") and does not have the capability to figure out faulty implementations of the application's specifications. The simple string plus comment payload bypasses openRASP's detection because during the check it crashes, as the tokens it generates stop after the comment and because it bypasses a check, it will then try to search in a token position that does not exist. It detects if there are more tokens than the string and comment, as it generates the extra tokens until the comment stops the token generation, but it works with the string and just the comment next to it. Griffin blocks because the structure of the query will be different by just adding the comment marks,

| Java Vulnerable Lab | | | |
|---|---|---|---|
| **Injection** | **No Protection** | **open-RASP** | **Griffin** |
| Safe search | accept | accept | accept |
| 3 or 1=1 | accept | block | block |
| 1 UNION SELECT @ --+ | accept | block | block |
| 1 AND extractvalue (rand(),concat(CHAR(126), version(),CHAR(126))) -- | accept | block | block |
| 5 OR IF(MID(@@version,1,1) = '8', sleep(3),1)='2' | accept | block | block |
| 3 and 1=2 UNION ALL SELECT load_file('/etc/passwd'), ..., load_file('/etc/passwd') (load file is repeated 9 times) | accept | block | block |
| 3 and 1=2 union SELECT 1,2,3,3,4,5,6,7,8,9 FROM users WHERE username LIKE 'admin' -- | accept | block | block |
| 3 and 1=2 union SELECT 1,db,3,4,5,6,7,8,9 FROM INFORMATION_SCHEMA.PROCESSLIST | accept | block | block |
| ' UNION ALL SELECT LOAD_FILE('/etc/passwd'),2,3 -- x ' | accept | block | block |
| Dump In One Shot | accept | block | block |
| a' -- (update description, will change description of everyone to the string) | accept | accept | block |
| *The next injections are supposedly taking advantages of badly coded specifications* | | | |
| Unsafe - **input 5 chars** (description must have more than 5 chars) | accept | accept | block |
| Safe - **input 6 chars** (specification above) | accept | accept | accept |
| Unsafe - **search for id number 1** (not possible to search for id 1) | accept | accept | block |
| Safe - **search for id number 3** (specification above) | accept | accept | accept |

Table 5.2: Java Vulnerable Labdetailed injections results

| Verademo | | | |
|---|---|---|---|
| **Injection** | **No Protection** | **open-RASP** | **Griffin** |
| Safe operation | accept | accept | accept |
| ben' or '1'='1 | accept | accept | block |
| b' UNION SELECT @ --+ ' | accept | block | block |
| ' AND extractvalue (rand(), concat( CHAR(126), version(),CHAR(126))) -- ' | accept | block | block |
| 5' OR IF( MID(@@version,1,1) = '8', sleep(3),1)='2 | accept | block | block |
| ' union SELECT 1 FROM users WHERE username LIKE 'a% | accept | block | block |
| a' UNION ALL SELECT LOAD_FILE('/etc/passwd'),2,3 -- x ' | accept | block | block |
| abc' union SELECT state,2,info FROM INFORMATION_SCHEMA.PROCESSLIST | accept | block | block |
| a' -- ' (update real name, will turn every name in database in 'a') | accept | accept | block |
| Username: test"), ("admin","admin was hacked Password: test Confirm Password: test Real Name: Ms SQL Hacker Blab Name: SQL Hacker - register for a second order injection | accept | block | block |
| 1' and (select sleep(10) from dual where database() like '% | accept | block | block |
| *The next injections are supposedly taking advantages of badly coded specifications* | | | |
| Unsafe - **input 0 char** (blabs cannot be empty) | accept | accept | block |
| Safe - **input 1 char** (specification above) | accept | accept | accept |
| Unsafe - **real name with letters and numbers** (name must only contain letters and not be empty) | accept | accept | block |
| Unsafe - **empty name** (name must only contain letters and not be empty) | accept | accept | block |
| Safe - **real name with only letters** (specification above) | accept | accept | accept |

Table 5.3: Verademo detailed injections results

| Secure Milk Carton | | | |
|---|---|---|---|
| **Injection** | **No Protection** | **open-RASP** | **Griffin** |
| Safe search | accept | accept | accept |
| a' or '1'='1 | accept | block | block |
| 1' UNION SELECT @ --+ ' | accept | block | block |
| ' AND extractvalue(rand(),concat(CHAR(126),version(),CHAR(126 -- ' | accept | block | block |
| 5' OR IF(MID(@@version,1,1) = '8', sleep(3),1)='2 | accept | block | block |
| 1' and (select sleep(10) from dual where database() like '%') -- ' | accept | block | block |
| ' union SELECT 1,2,3 FROM users WHERE username LIKE 'u% | accept | block | block |
| abc' union SELECT 1,state,info FROM INFORMATION_SCHEMA.PROCESSLIST | accept | block | block |
| the boss' -- (login and bypass password hash) | accept | accept | block |
| ' UNION ALL SELECT LOAD_FILE('/etc/passwd'),2,3 -- x ' | accept | block | block |
| Dump In One Shot | accept | block | block |
| *The next injections are supposedly taking advantages of badly coded specifications* | | | |
| Unsafe - **input 8 chars** (comments must have more than 8 chars) | accept | accept | block |
| Safe - **input 9 chars** (specification above) | accept | accept | accept |
| Unsafe - **not starting with "hello"** (comments must start with "hello") | accept | accept | block |
| Safe - **comment starts with "hello"** (specification above) | accept | accept | accept |

Table 5.4: Secure Milk Carton detailed injections results

| Insecure Bank | | | |
|---|---|---|---|
| **Injection** | **No Protection** | **open-RASP** | **Griffin** |
| Safe search | accept | accept | accept |
| a' or '1'='1 | accept | block | block |
| 1' UNION SELECT @ --+ ' | accept | block | block |
| ' AND extractvalue (rand(), concat( CHAR(126), version(),CHAR(126))) -- ' | accept | block | block |
| 5' OR IF( MID(@@version,1,1) = '8', sleep(3),1)='2 | accept | block | block |
| ' union SELECT 1,state,info,4 FROM INFORMATION_SCHEMA.PROCESSLIST -- x ' | accept | block | block |
| n' and (select sleep(3) from dual where database() like 'i__________ | accept | block | block |
| ' union SELECT 1,2,3,4 FROM account WHERE username LIKE 'd___ | accept | block | block |
| john' -- (login, bypass password check) | accept | accept | block |
| ' UNION ALL SELECT LOAD_FILE('/etc/ passwd'),2,3 -- x ' | accept | block | block |
| Dump In One Shot | accept | block | block |
| *The next injections are supposedly taking advantages of badly coded specifications* | | | |
| Unsafe - **Transfer -1.0** (transfer amount must be higher than zero when in the same query the description starts with "TRANSFER:") | accept | accept | block |
| Safe - **Transfer 1.0** (specification above) | accept | accept | accept |
| Unsafe - **send money to account "a"** (toAccount must end with "!") | accept | accept | block |
| Safe - **send money to account "a!"** (specification above) | accept | accept | accept |

Table 5.5: Insecure Bank detailed injections results

| Security Shepherd | | | |
|---|---|---|---|
| **Injection** | **No Protection** | **open-RASP** | **Griffin** |
| Safe search | accept | accept | accept |
| a" or "1"="1 | accept | block | block |
| 1' UNION SELECT @,@ -- ' | accept | accept | block |
| a" AND extractvalue (rand(), concat( CHAR(126), version(),CHAR(126))) -- " | accept | block | block |
| 5" OR IF( MID(@@version,1,1) = "8", sleep(3),1)="2 | accept | block | block |
| " union SELECT 1,state,info,4 FROM INFORMATION_SCHEMA.PROCESSLIST -- x " | accept | block | block |
| " union SELECT 1,2,3,4 FROM customers WHERE customerName LIKE 'J_______' – " | accept | block | block |
| n" and (select sleep(3) from dual where database() like 'S%') – " | accept | block | block |
| asx' union select version() -- ' | accept | accept | block |
| ' UNION ALL SELECT LOAD_FILE('/etc/ passwd') -- ' | accept | block | block |
| Dump In One Shot | accept | block | block |
| *The next injections are supposedly taking advantages of badly coded specifications* | | | |
| Unsafe - **pin using letters** (pin must be only numbers) | accept | accept | block |
| Safe - **pin using numbers** (specification above) | accept | accept | accept |
| Unsafe - **coupon code with numbers** (codes must be only letters) | accept | accept | block |
| Safe - **coupon code with letters** (specification above) | accept | accept | accept |

Table 5.6: Security Shepherd detailed injections results

so it does not find it in the learned queries and blocks it.

In Verademo, the analysis for the string plus comment payload also can be applied, together with the incapability to detect application's specifications. In this web application there is an extra case where an injection is successful. The first injection works because the payload used will be used with a prepared statement and it is not caught, being stored in the application without the string tokens being applied correctly (quotes or single quotes). When the user leaves and checks its profile again, this username is stored in a query that does not use prepared statements, so the injection is correctly placed when creating the query string. When the request is performed, there are no user values in this request, so openRASP does not catch or perform any detection to this query. This input also passes the next checks that are performed after the token analysis. Griffin, on the other hand, checks the structure of every query, whether there is user input or not, so when the structure is created and compared, it will not find this structure coming from the method, so it is detected. This is seen with extra five tests performed in the next part.

Finally, in Security Shepherd the previous analysis of the three applications for the incapability to detect application's specifications is also true. In this case, there is no input where the string plus comment would be effective, so it was not possible to see its result. But this time, there were two successful injections. The application changes the input in the method being tested, where it attempts to have some security. In this case, the method removes the comment tokens from the query that is sent to the database, so when openRASP searches for the value via user input to locate where the input starts, it does not find it in the query caught and fails to block the injection, as the comment tokens are not in the final query. After this check, it also passes in the other injection detection steps.

**Authentication bypass: Results**

To check how the solutions behave against authentication bypasses, it was injected payloads in the regular application and then compare the responses to the application using openRASP and Griffin's solution against SQL injections. The results obtained are split in two lines: number of successful attacks that were not detected and number of injection blocks. The table 5.7 contains the results of the tests.

| | | No Protection | openRASP | Griffin |
|---|---|---|---|---|
| Java Vulnerable Lab | **# successful injections** | 6/100 | 6/100 | 0/100 |
| | **# blocks of potential injections** | 0/100 | 44/100 | 75/100 |
| Verademo | **# successful injections** | 10/100 | 3/100 | 0/100 |
| | **# blocks of potential injections** | 0/100 | 46/100 | 75/100 |
| Secure Milk Carton | **# successful injections** | 19/100 | 11/100 | 0/100 |
| | **# blocks of potential injections** | 0/100 | 46/100 | 75/100 |
| Insecure Bank | **# successful injections** | 23/100 | 12/100 | 0/100 |
| | **# blocks of potential injections** | 0/100 | 43/100 | 75/100 |
| Security Shepherd | **# successful injections** | 0/100 | 0/100 | 0/100 |
| | **# blocks of potential injections** | 0/100 | 0/100 | 2/100 |

Table 5.7: Authentication bypass test results

In Java Vulnerable Lab, it is possible to see that openRASP did not block the 6 attacks that are successfully performed to the application, where Griffin blocked them. It is also possible to see the number of attempts blocked by Griffin is higher (44 vs 75).

In Verademo, the base app had 10 successful attacks. OpenRASP successfully blocked 7, with 3 injections finding success. In Griffin, no injection had success. In terms of blocked attempts, Griffin scored a higher number of detections (46 vs 75).

In Secure Milk Carton, the base app observed 19 successful injections, where openRASP observed 11 and Griffin blocked every injection. Again, Griffin blocked a higher number of attempts (46 vs 75). In Insecure Bank, the application was successfully attacked 23 times, openRASP was vulnerable to 12 attempts, and Griffin again blocked every attempt, while performing a higher number of blocked attempts (43 vs 75).

In Security Shepherd, as this application uses a prepared statement for the login, there are no successful injections in the three cases. When blocking, however, Griffin blocked two cases, as the values in the prepared statements had a length higher than the one seen in the schema.

In conclusion, Griffin blocked every attack that was successful and performed a higher number of blocks, as by looking purely at the structures, every bypass attempt that would successfully change the structure is blocked, even if it does not correlate to a successful attack. Note that achieving a higher number of blocks in unsuccessful attacks cannot be correlated to the regular interactions of the users to applications, as the regular interactions will not attempt to change the structure of the queries performed, meaning that in regular interactions these blocks would not occur. OpenRASP blocked some successful attacks, but still allowed 32 successful attacks out of 58 to the base applications.

#### Extra tests: Results

In the final part of the accuracy tests, each application was injected with two sets of SQL injections: a new set of injections with multiple types of injections and the same set of injections used in the previous test. The results reported are split in these two sets, and are a direct comparison between openRASP and Griffin. First, it is shown the number of requests blocked and from the number of requests that passed, it is then showed the number of them that were a successful attack.

Note that Verademo has an extra set of five injections, because it was not possible to perform such request via a script, so this set of injections was performed manually. The effort for these extra five tests is to investigate deeper the conclusion reached in the first part of the accuracy tests. The results can be consulted in table 5.8.

| | | | **openRASP** | **Griffin** |
|---|---|---|---|---|
| Java Vulnerable Lab | Set 1 | **# successful injections** | 6*/158 | 0/158 |
| | | **# blocks of potential injections** | 148/158 | 158/158 |
| | Set 2 | **# successful injections** | 1*/100 | 0/100 |
| | | **# blocks of potential injections** | 74/100 | 100/100 |
| Verademo | Set 1 | **# successful injections** | 2/158 | 0/158 |
| | | **# blocks of potential injections** | 135/158 | 146/158 |
| | Set 2 | **# successful injections** | 8/100 | 0/100 |
| | | **# blocks of potential injections** | 46/100 | 56/100 |
| | Extra tests | **# successful injections** | 5/5 | 0/5 |
| | | **# blocks of potential injections** | 0/5 | 5/5 |
| Secure Milk Carton | Set 1 | **# successful injections** | 0/158 | 0/158 |
| | | **# blocks of potential injections** | 135/158 | 158/158 |
| | Set 2 | **# successful injections** | 6/100 | 0/100 |
| | | **# blocks of potential injections** | 46/100 | 59/100 |
| Insecure Bank | Set 1 | **# successful injections** | 0/158 | 0/158 |
| | | **# blocks of potential injections** | 135/158 | 146/158 |
| | Set 2 | **# successful injections** | 6/100 | 0/100 |
| | | **# blocks of potential injections** | 42/100 | 57/100 |
| Security Shepherd *Location 1* | Set 1 | **# successful injections** | 0/158 | 0/158 |
| | | **# blocks of potential injections** | 135/158 | 146/158 |
| | Set 2 | **# successful injections** | 0/100 | 0/100 |
| | | **# blocks of potential injections** | 37/100 | 46/100 |
| Security Shepherd *Location 2* | Set 1 | **# successful injections** | 2/158 | 0/158 |
| | | **# blocks of potential injections** | 112/158 | 131/158 |
| | Set 2 | **# successful injections** | 8/100 | 0/100 |
| | | **# blocks of potential injections** | 34/100 | 57/100 |

Table 5.8: Extra tests results

In Java Vulnerable Lab, the first set of possible injections were all blocked by Griffin, while the application with openRASP's SQL injection detection returned an error page in 6 payloads, where the the query being used is leaked, which was considered in success, within the 10 attempts not blocked. In the second set, Griffin blocked every injection while openRASP did not block 26 attempts, where 1 was successful, where the behavior observed was the same described in the first set.

In the first set of Verademo, while Griffin did not block 12 attempts, it saw no successful injections. With openRASP it was observed 2 successful injections within the 23 not blocked. In the second set, Griffin's protection did not allow any injection in the 44 potential injections not blocked. In openRASP it

was observed 8 successful injections in 54 attempts not blocked. In the extra tests, Griffin blocked all five injections that were successful on openRASP.

In Secure Milk Carton, Griffin blocked every possible injection, while openRASP did not block 23 in the first set of injections, which did not result in a successful attack. For the second set of injections, openRASP did not block 54 attempts, where 6 were successful. Griffin did not block 41, but no attempt was successful.

In Insecure Bank, Griffin did not block 12 possible injections in the first set, where openRASP did not block 23. Both solutions did not allow any successful injections. For the second set, Griffin did not block 43 possible injections, but none resulted in a successful injection. With openRASP is was observed 6 successful injections within the 58 possible injections not blocked.

Finally, in Security Shepherd, the results are separated in two different locations. In the first location, both Griffin and openRASP did not allow any successful injection in the first set, where Griffin did not block 12 potential injections and openRASP did not block 23 possible injections. In the second set, both did not allow successful injections, Griffin did not block 54 possible injections and openRASP did not block 63. In the first set of the second location, Griffin did not allow any successful injection in the 27 possible injections not blocked. In the 46 possible injections not blocked, openRASP let in 2 successful injections. In the second set, Griffin did not block 43 possible injections, which did not result in a successful injection. OpenRASP did not block 66 possible injections, which resulted in 8 successful injections.

In summary, Griffin did not allow any successful injections, in both sets, while also blocking a higher number of possible injections. In openRASP it was observed 11 successful injections in the first set and 29 in the second set. Note that five of which came from manual injections, but it could be increased as the mechanism fails to track those payloads, as explained in the detailed analysis.

### 5.2.3 Results for learning overhead

It was also performed extra tests to determine the overhead that the learning process could introduce. For these tests, a set of requests was performed in order to measure how Griffin's learning phase behaves when dealing with multiple requests while in the learning phase. These set of requests are simulating small samples of test suites. The results can be observed in figures 5.6 and 5.7. As Security Shepherd's results are in a different scale, they are separated.

The overheads observed go from 3.64% (Java Vulnerable Lab) to 30.62% (Insecure Bank). Security Shepherd has an added 0.70%, while the other two add 8.43% (Secure Milk Carton) and 25.70% (Verademo). The overhead observed in reasonable, which means that for longer sets of requests it should not introduce a noticeable overhead. however, note that this is supposed to run only once.

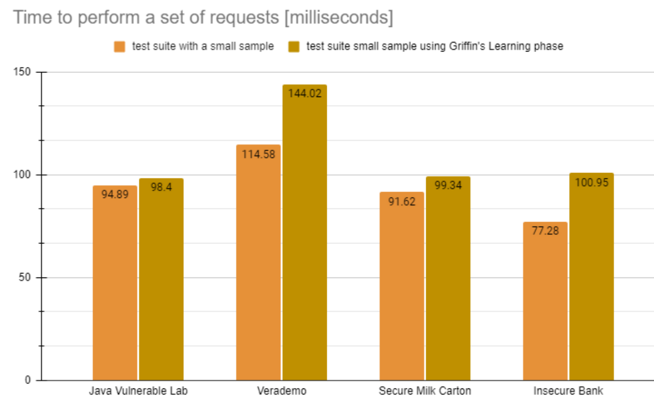


Figure 5.6: Results observed for the learning phase in Java Vulnerable Lab, Secure Milk Carton, and Insecure Bank

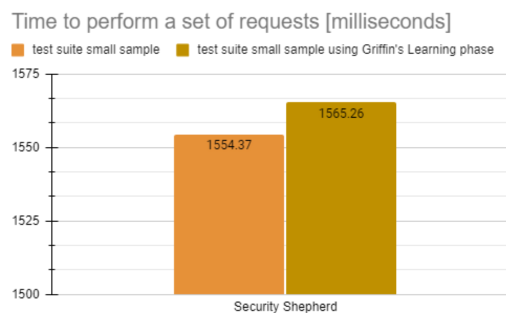


Figure 5.7: Results observed for the learning phase in Security Shepherd

## 5.3 Summary

In the results obtained it is possible to see that Griffin has an added overhead of compared to open-RASP's SQL injection detection, but was able to detect more injections that were used. In terms of overhead, the highest overhead observed from openRASP, without the specification schema, was 15.41%, while Griffin's worst case highest was 50.88%. Using the specification added, in the worst case, 14.99% overhead to Griffin's worst case was added to the version without the specification check, which means adding this check will not add an unreasonable penalty by adding this feature. It is also seen that between Griffin's best (query in top of the file) and worst case (query bottom of the file) the highest difference is 7.46%, which can be seen as the overhead when using applications with more queries will not suffer a noticeable penalty when the query being used is in the bottom. Finally, it is also compared if adding more specifications that are checked by one query increases a high overhead, but the highest observed between the best (query in the request is only checked by one specification) and worst (query in the request is checked by every specifications) was 13.68%, which is not high. When looking at the detection accuracy results, Griffin successfully did not let any successful injection in every test, blocking every relevant injection, and added an extended feature to the detection of incorrect data being inserted in the database. In the learning phase, the highest overhead observed was 13.68%, which is not high. This means that extracting every query will not add an unreasonable overhead. Note that this phase is only performed once.

# Chapter 6

# Conclusions

In order to protect web applications against SQL injections during runtime, one can resort to Runtime Application Self-Protection, RASP, to observe the full context and know the full query that is sent to the database.

As SQL injections happen when the structure of the queries are successfully altered, Griffin has the goal of knowing the specifications of the SQL queries issued by web applications that were intended by the developers, via SQL structures expressed as prepared statements. With that knowledge, during runtime it attempts to detect when users change the structure to attack the database, going against the normal behavior that was intended by the developers. Not only that, Griffin also evaluates the user input and makes sure that it follows the database schema. There are also specifications regarding the data stored in the database that can be tampered due to failed checks in the application, which Griffin can detect and prevent from happening with the help of specifications provided by developers. These capabilities are especially useful in applications where the source code cannot be changed, such as legacy applications.

Griffin is works in two phases: learning, where the specifications of the application create a model with attack-free requests, and detection, where the queries that are issued are compared to the model that was learned, their user input values are compared to the database schema, and the values can also be checked against specifications expressed by developers.

Regarding the evaluation of the tool, Griffin was tested against openRASP's implementation of SQL injection detection in terms of overhead added to regular requests, accuracy in the detection, and it was also tested the overhead introduced by the learning phase. In these tests it was showed that while openRASP's SQL injection detection is the solution that introduces a lower overhead, Griffin detected and blocked every attack, while openRASP failed to block some attacks that were successful.

In conclusion, Griffin brings the following benefits:

- Detecting every SQL Injection attack that alters the structure of the SQL query while making decisions on the full query and not just on the user input, which means that every query issued is analyzed and the mechanism will fail to track ”bad” queries
- Performing analysis on the values being input by the users in order to prevent exceptions from

being launched, due to wrong size or type. This type of checking is also perform on prepared statements, where it can also detect and help developers when there is an incompatibility between the Set method used and the column schema

- Allows developers to introduce specifications to create subtypes of the existing SQL types, meaning that if the code has bugs, Griffin will prevent them from being explored

## 6.1 Future Work

For the future, there are different paths to take. One of the paths for this solution could be extend this approach to other SQL languages by expanding Griffin with the syntax of them to create the structures. One other path is that it could be attempted to extend this approach to web applications written in other languages. One final suggestion is the adaptation of this type of detection to other injection attacks, like command injection. Approach is to work on this solution in improving the overhead it introduces versus openRASP's SQL injection detection. Other than that, there are still some operations that Griffin does not support, such as calculating the results from functions, in order to perform the checks for specifications.

# Bibliography

[1] G. Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais. Detlogic: A black-box approach for detecting logic vulnerabilities in web applications. *Journal of Network and Computer Applications*, 109:89–109, 2018.

[2] A. J. Fry. Runtime application self-protection (rasp), investigation of the effectiveness of a rasp solution in protecting known vulnerable target applications. Technical report, 2019.

[3] openrasp repository. `https://github.com/baidu/openrasp`, .

[4] G. Deepa and P. S. Thilagam. Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, 74:160–180, 2016.

[5] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, March 2006.

[6] V. Prokhorenko, K.-K. R. Choo, and H. Ashman. Web application protection techniques: A taxonomy. *Journal of Network and Computer Applications*, 60:95–112, 2016.

[7] P. Čisar and S. M. Čisar. The framework of runtime application self-protection technology. In *2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000081–000086. IEEE, 2016.

[8] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *International Workshop on Recent Advances in Intrusion Detection*, pages 63–86. Springer, 2007.

[9] X. Li and Y. Xue. Block: a black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 247–256. ACM, 2011.

[10] X. Li, W. Yan, and Y. Xue. Sentinel: securing database from logic flaws in web applications. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 25–36. ACM, 2012.

[11] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[12] A. Alazab and A. Khresiat. New strategy for mitigating of sql injection attack. *International Journal of Computer Applications*, 154:1–10, 11 2016. doi: 10.5120/ijca2016911974.

[13] M. Alattar and S. Medhane. R-wasp: Real time-web application sql injection detector and preventer. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 2:2278–3075, 05 2013.

[14] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2), Mar. 2010. ISSN 1094-9224. doi: 10.1145/1698750.1698754. URL `https://doi.org/10.1145/1698750.1698754`.

[15] W. G. J. Halfond and A. Orso. Preventing sql injection attacks using amnesia. In *ICSE '06*, 2006.

[16] Waratek. Runtime application self protection (rasp). Technical report, 2019.

[17] Imperva. Runtime application self-protection (rasp)). Technical report, 2019.

[18] openrasp architecture. `https://rasp.baidu.com/doc/hacking/architect/java.html`, .

[19] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[20] Java vulnerable lab repository. `https://github.com/CSPF-Founder/JavaVulnerableLab`.

[21] Verademo repository. `https://github.com/veracode/verademo`.

[22] Secure milk carton repository. `https://github.com/thomaslaurenson/SecureMilkCarton`.

[23] Insecure bank repository. `https://github.com/hdiv/insecure-bank`.

[24] Security shepherd repository. `https://github.com/OWASP/SecurityShepherd`.

[25] Repository with attack payloads. `https://github.com/swisskyrepo/PayloadsAllTheThings`, .

[26] Repository with sql injection attack payloads. `https://github.com/trietptm/SQL-Injection-Payloads`, .