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

Nuno Bombico
nuno.bombico@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

January 2021

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

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 it is impossible to develop applications that are one hundred percent secure. This means 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. [13] 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 [14], 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.

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 Runtime Application Self-Protection, RASP. It knows he context of the requests performed by users, knowing where the input will be used. This technology is attached to the applications, knowing

exactly the queries issued to the database, allowing it to protect the data. 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.

Griffin, which is an approach to protect MySQL databases from SQL injection attacks in Java Web Application, is built on top of an existing RASP solution, openRASP [4]. It is designed to secure databases by extracting the intent of the developers, in the form of the specifications, of what the are the correct queries that are sent to the database. With the assist of openRASP, it gets 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 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. In the detection phase, it will compare the structures observed to the ones that were learned, check if the parameters that are going to be used are legal and will not make the database launch exceptions, and finally it is 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 check for sub types of the main SQL types, for example, to allow the database to only store integers higher than a certain number. Not only that, it can also be used to give business logic context to the queries. Note that Griffin will be available in open-souce.

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.

In terms of evaluation, this solution will be compared to the existing openRASP's SQL injection detection mechanism. Griffin also has a test to see the overhead introduced while performing the learning phase. In the results obtained, 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. 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.

2. Background

In this section, it will be briefly described some concepts regarding SQL injections, which are the attacks that this solution aims at detecting. After that, it will be introduced the concept of Runtime Application Self-Protection.

2.1. SQL Injection

According to Deepa and Thilagam[12], 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. In one of those problems, input validation flaws, allow attackers to perform injections to the application, leading to *injection vulnerabilities.* Griffin is a solution that aims at protecting web applications against SQL injections, which is a subtype of injection vulnerabilities.

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. [15] 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. For example, 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 comment 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. Regarding the SQLi attacks, Halfond et al. [15] mentions they can be split into: *tautology*, *illegal/logically incorrect queries*, *union query*, *piggybacked queries*, *stored procedures*, *inference*, and *alternate encodings.* The difference is in the payload used, where it changes according to the intent of the

attack. Depending on the type of attack used, the intents can go from being as simple as identifying injectable parameters to more serious attacks such as bypassing authentication mechanisms, extract data, modifying data, or execute random commands.

2.2. Runtime Application Self-Protection

Runtime Application Self Protection [11], 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 the web application behave differently. This is possible because RASP works at the application level, so it knows the context where the inputs provided are going to be used. Regarding monitoring, the different monitor approaches mentioned for web applications are: *server filters and plugins*, *library/Java Virtual Machine replacement*, and *virtualization/replication*. Fry [14] also mentions another monitoring method, *binary instrumentation*.

3. Related Work

To protect against SQL injections, there are different approaches. Using specifications, there is CANDID [10], which bases its detection technique around prepared statements as the intent of the developers. In runtime, it calculates two structures, one from attack-free inputs to simulate the intent of the developers and another from the specific inputs used by the user. Another approach is AMNESIA [16], that uses static analysis to create models and then performs runtime monitoring using those models.

Regarding RASP solutions, most of them 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 or simply to try them. Because of that, the tool picked is a OpenRASP [4], a popular open-source RASP tool from Baidu security. OpenRASP is designed to defend applications against attacks by placing hooks in dangerous calls via Javassist, for example, in database accesses. After the bytecode manipulation is performed, 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 [3]. Specifically for the SQL injections detection, this script performs three different steps: (1) it checks if the user input changes the logic of the query, by collecting the user input from the http request and checking if it changes the number of tokens of the original query, (2) checks the query against security 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) checks the query string against the regular expression 'union.*select.*from.*information_schema'. This approach has two issues. When the user input that was in the http request is changed by the application, it cannot find it in the query issued. The other issue is a problem with the tracking when using prepared statements, as these are ignored and the application reusing the input in a place where that input is not given by the user. In these cases, the second and third step covers some attacks, but those rules can be bypassed. The tool itself has low overhead, which means that it is possible to add a new detection mechanism without working on an already dangerous overhead.

4. Implementation

Griffin is a RASP security solution for Java Web Applications that is used to protect MySQL databases during runtime. It works on top of openRASP and uses its 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 the structure changing with user input, to detect every type of SQL injection. It works in two different phases: learning and detection. In the learning phase, it will create a model containing the specifications of the safe queries intended by the developers. 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, not using a previous learned model. Next, it will be detailed how both phases work, where Griffin and openRASP's approach will be compared in the detection phase, as openRASP does not have a learning phase.

4.1. Learning Phase

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 structures that will cover the variations of normal and secure queries.

The idea is turning every query into (abstract) safe prepared statements, so the structures will consist of the regular query, but the user input will

be replaced by the placeholder "?". With this approach, it will cover every different correct variations of the query. In order to decide when to capture the specifications, it is used the approach of Query Synthesis [17], 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.

During attack-free executions of a web application, Griffin will check 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 the method and class where this query was issued, from the stack trace, and together with the query it will be stored 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 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. Note that when the queries use prepared statements, they are already in the correct structure, so the learner will just obtain the method and class, and then consult the oracle to label the specification calculated. 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. In figure 1 it can be seen a summary of this phase. To obtain this model, the application could be exercised using attack-free test suites to collect every possible SQL structure issued by web applications, as long as they cover every structure.

4.2. Detection Phase

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. Griffin will check (1) if the query issued has the correct structure, (2) check if the values used in the query have the correct type and size, and (3) check the values against specifications input by the developers. OpenRASP also has three steps, where it generates the tokens of the original query and then (1) calculates the number of tokens that the user input created to check if the logic of the query was changed, (2) checks every token against security specifications. After that, (3) checks the query string against the regular expression 'union.*select.*from.*information_schema'. After being caught in the hook, the method and class name are obtained, then the query will be transformed into a structure and 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 check 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. For example, if it receives the query *SELECT * FROM User WHERE username = "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 ob-

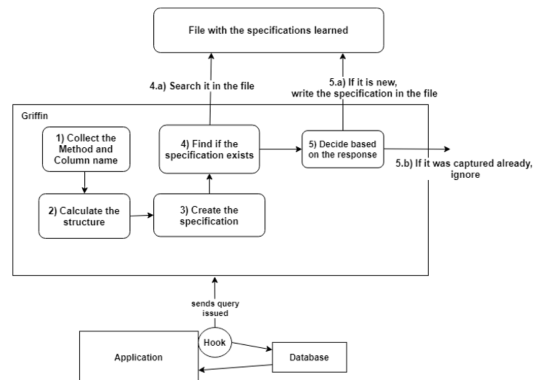

Figure 1: Griffin's learning phase

tained 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 obtained at a different time, there is an additional hook that catches the values and then checks them. The type check has an additional element, as the type comes from the prepared statement method, which is a quick check for developers that may use an incompatible method to send the prepared statement value. The correct type is obtained from the schema and then it is checked which methods are compatible and if the value came from one of those methods. After that, the values are checked for the size. For example, if the schema has a column with type INT and receives a string, the request is blocked. If it receives an integer with an illegal size, for example, when an unsigned integer receives a negative number or an integer with a size that is too big, it blocks. For the prepared statement, if the application issues a value with setString into a column with INT specification, the request is also blocked. 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. OpenRASP does not have support for this feature, as it does not evaluate the values of the inputs used. These specifications will contain constraints in data inserted in the database and are written in a YAML file, specSchema.yml. For example, a constraint could be that the values in the column 'id' must be higher than 10, which 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 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 2 shows the

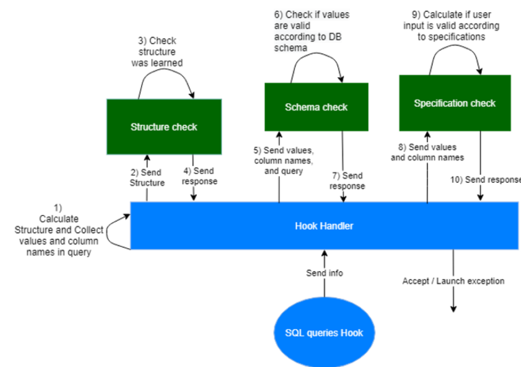

Figure 2: Griffin's detection phase flow

flow of Griffin working when a query arrives via non prepared statement. The prepared statements are similar, but the second and third check are performed when the values arrive from a new hook that was created, and also obtain extra information from hooks, such as last query for the second check and every value issued via prepared statements for the third phase.

5. Results

For the evaluation of Griffin, its performance will compared to openRASP's SQL injection detection. First it will be compared the overhead added by both solutions, then the effectiveness in blocking SQL injections, and the overhead added while performing Griffin's learning phase. The tests were performed in five open-source vulnerable java web applications that issue MySQL queries: Java Vulnerable Lab [2], Verademo [9], Secure Milk Carton [7], Insecure Bank [1], and Security Shepherd [8]. The applications' characteristics regarding number of lines (generated using David A. Wheeler's 'SLOCCount') and the number of unique queries that each application issues can be consulted in table 1. 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 in a configuration 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. Finally, the injection payloads were obtained and modified from PayloadOfAllThings [5] and SQL-Injection-Payloads [6].

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

Table 1: Web applications being tested

5.1. Performance overhead

To measure the overhead that Griffin adds to the application's requests, it was measured the time that it takes the normal request to be processed and compared against openRASP and multiple Griffin setups. The Griffin setups are split in two big groups: using the specification schema and not using it. Within the first group, three setups were tested: the request issues a query that is fifth on the learned queries list (best case), a query in twenty that is twenty-fifth, and fiftieth (worst case). In the figures that will be showed, the configurations will be called (i), (ii), and (iii), respectively. In these tests, the query is the same, but is positioned according to the tests and the other queries are repeated to allow the query to be positioned in the correct place. Also, there are tests where the request issues multiple queries and in those cases the queries are put next to each other in the file. In the second group, where it is included the specification schema, the queries issued are all place in the fiftieth place in the learning file, so that the only variance would only be the number of specifications being evaluated. Of a total of twenty specifications created, the tests are split in: request matches one specification (best case), the last one, matches ten specifications, every other one, and matches twenty specifications (worst case), all. In the figures that will be showed, these configurations will be called (a), (b), and (c), respectively. 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. The results can be found in figures 3, 4, 5, 6, and 7, where each figure corresponds to each application. Note that Security Shepherd is in another scale, so its overhead results is in a different scale.

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). Between Griffin and the applications without protection, the worst case

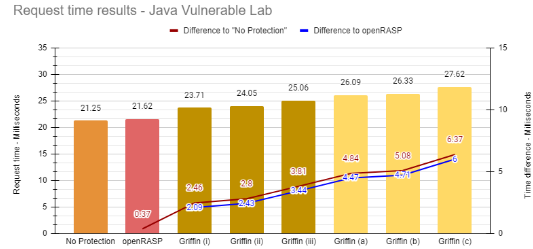

Figure 3: Results for Java Vulnerable Lab

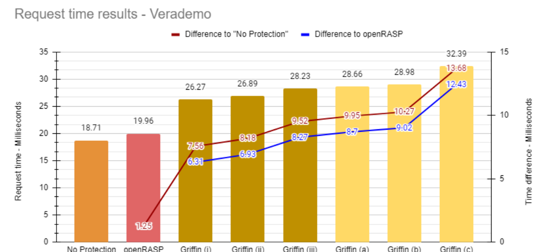

Figure 4: Results for Verademo

(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 add an reasonable 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

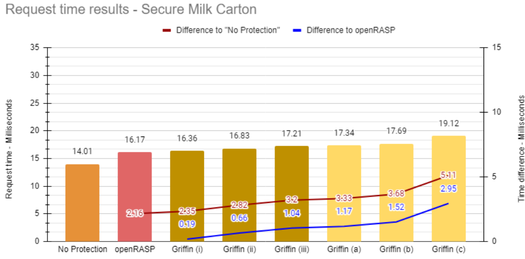

Figure 5: Results for Secure Milk Carton

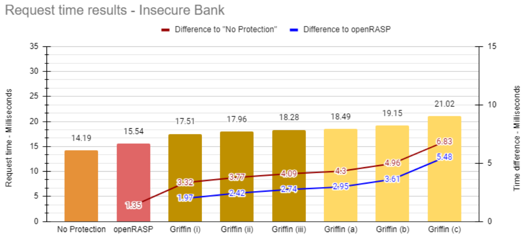

Figure 6: Results for Insecure Bank

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. Accuracy: Authentication bypass

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 2 contain the test results

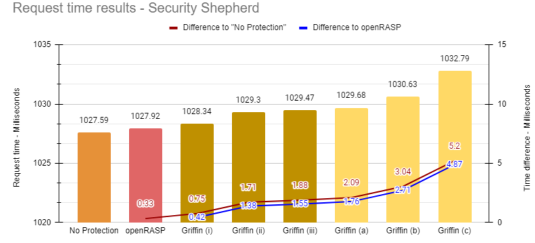

Figure 7: Results for Security Shepherd

for each application.

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). Finally, 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.

5.3. Accuracy: Multiple Injections

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. In these tests, the payloads in the

		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 2: Authentication bypass test results

first set were changed according to each application, to maintain the payloads relevant. Note that Verademo has an extra set of five injections, because it was not possible to perform such request via a script, they were performed by hand. The effort for these extra five tests was to investigate deeper one of the flaws presented about it. The results can be seen in table 3.

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, while also blocking a higher number of possible injections. OpenRASP allowed 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.

5.3.1 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 8 and 9. 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 not big, which means that for longer sets of requests it should not

			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 3: Extra tests results

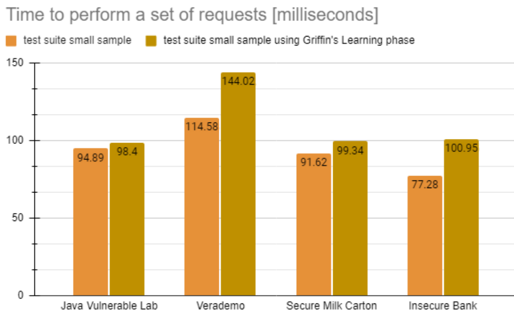

Figure 8: Results observed for the learning phase in web applications

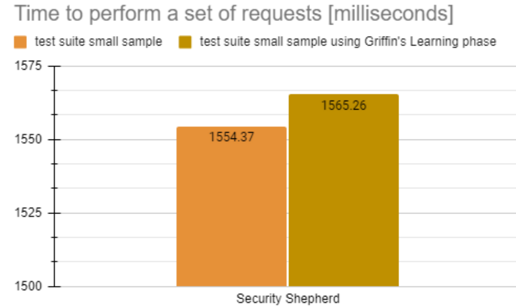

Figure 9: Results observed for the learning phase in Security Shepherd

introduce a big overhead. However, note that this is supposed to run only once.

6. Conclusions

RASP can be used to protect web applications against SQL injections during runtime, as knows the exact query sent to the database with the user input, as it has access to the full context. 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. 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.

Acknowledgements

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 while I was working on Griffin.

References

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

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

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

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

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

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

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] W. Halfond, J. Viegas, and A. Orso. A classification of sql injection attacks and countermeasures. 01 2006.

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

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