UNIVERSITY OF NAIROBI

SCHOOL OF COMPUTING AND INFORMATICS

LONG SHORT TERM MEMORY BASED DETECTION OF WEB BASED SQL INJECTION ATTACKS

BY

MWARUWA CHAKA MWARUWA

P52/6208/2017

SUPERVISED BY

DR. EVANS MIRITI

AUGUST 2019

This project report is submitted in partial fulfillment of the requirements for the degree of Master of Science in Computational Intelligence, University of Nairobi.

Declaration

This project, as presented in this report, is my original work and has not been presented for award in any other university.

Name: Mwaruwa Chaka Mwaruwa **Reg No:** P52/6208/2017

Signature: _____ **Date**: _____

This project has been submitted with my approval as the University supervisor.

Name: Dr. Evans Miriti

Signature: _____ **Date**: _____

Abstract

The internet has experienced considerable growth in the past decade due to increased ease of access and growth of mobile technologies. The internet is increasingly being used for important transactions such as financial transactions. With this growth, security has become a major concern as sophisticated attacks continue to be observed on various systems. Injection attacks are one of these attacks, and it's prevalence has remained high in the past few years, having been at the top of the OWASP top ten list in 2013, 2015 and 2017. Existing signature based intrusion detection systems use known attack signatures, hence it's difficult for them to keep up with the ever changing attack landscape. Existing work using neural networks focuses on one kind of injection attacks, hence leaving out vulnerability to the other kinds of injection attacks.

This study presents a machine learning based approach to detect injection attacks. We develop a method of collecting a diverse dataset of injection attacks, by using *sqlmap* and a custom python script to send requests to a vulnerable application. We then develop and train a neural network model using long short term memory (LSTM) networks that detects injection attacks. We then test the model to determine its performance so as to evaluate its ability to detect these attacks.

The model shows a good detection performance, reaching an accuracy of 95.4%. The model is superior to other similar works due to its ability to detect the eight different kinds of sql injection attacks, compared to similar works that are not as diverse.

We found that LSTM recurrent neural networks are a sufficient tool for the detection of injection attacks due to their ability to correctly classify the attacks from genuine requests. We further keep a log of all detections from the model, which can be used to retrain it hence learn from new attacks, making it a better solution for the ever changing attack landscape compared to the existing signature based methods.

Dedication

I dedicate this work to my parents and siblings for always believing in me. Your motivation keeps me going.

Acknowledgement

I express my utmost gratitude to Allah (S.W.T) for the health, the blessings and enabling me do this study.

My sincere gratitude goes to my supervisor, Dr. Evans Miriti for all the help, suggestions and encouragement during this study. I am deeply indebted.

To my mother Ms. Chizi Mangale and my father, Mr. Chaka Dete, for all the encouragement, support and prayers that has brought me here.

To all my family, friends and colleagues, I thank you for all the support you accorded me during the time of this study.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AUC –	Area Under Receiver Operator Characteristics Curve
DVWA –	Damn Vulnerable Web Application
FPR –	False Positive Rate
HTTP –	Hypertext Transfer Protocol
LSTM –	Long Short Term Memory
OS –	Operating System
OWASP –	Open Web Application Security Project
RNN –	Recurrent Neural Networks
ROC –	Receiver Operator Characteristics
SQL –	Structured Query Language
SVM –	Support Vector Machine
TPR –	True Positive Rate

Chapter 1: Introduction

1.1 Background

The internet has experienced considerable growth in the last decade. This has been largely driven by the increased ease of access and growth of mobile technologies. The internet today represents a popular medium for communication and even transactions. According to Arbor Networks, web traffic accounts for over 50% of this internet traffic, showing that the internet is mostly used to access web applications (Anstee, 2017).

Security remains a key challenge for various organizations on the internet. It is reported that t 47% of all attacks detected are SQL injections, followed by local file inclusion at 38% and cross site scripting (XSS) at 9% (Akamai, 2017). Insecure software has presented a serious risk to various critical infrastructure, including financial, defense and healthcare. Software is becoming increasingly complex and connected, hence securing it is becoming increasingly difficult. The pace of software development has become rapid, bringing a need to quickly discover and resolve any security issues accurately and quickly. The software developers developing most software today are not as experienced as the developers of software infrastructure such as databases, hence may not have deep understanding of the security vulnerabilities that their software expose. This results to poorly developed code full of security vulnerabilities being deployed and exposed to the entire internet (Valeur, Mutz and Vigna, 2005).

Injection attacks represent a significant risk to web applications. They are ranked at the top of Owasp top ten list, which ranks common vulnerabilities by prevalence. Injection attacks refer to when untrusted data containing rogue SQL queries, OS or LDAP commands are passed to a computer system. This untrusted data may result unprecedented consequences when run leading to data loss or leakage. OWASP is a community organization that enables organizations to build trustworthy applications by increasing security awareness (Owasp, 2017).

Secure coding practices have been proposed as a way of preventing these and other attacks. To protect applications against SQL injection attacks, software developers are advised to use parameterized queries. These are queries where the parameters are included in a query, then the arguments to replace these parameters are given during execution. For command injection attacks, developers are advised to always escape shell command arguments, so as to avoid

injection of arbitrary commands. These methods work, however, inexperienced programmers may not always know the right approaches hence introducing vulnerabilities by using the vulnerable coding practices. This brings about the need for methods that are able to protect such vulnerable applications (Howard and Leblanc, 2003).

Static scanning tools are used to identify vulnerabilities in source code by identifying whenever insecure coding practices are used. Dynamic scanning tools attempt to identify vulnerabilities when code is executed. Static and dynamic application scanning tools can be used to identify vulnerabilities in applications, hence aiding developers to patch their applications against injection attacks. However, the accuracy of these tools is not yet good enough, hence may not be able to detect all vulnerabilities in the applications. Commercial tools have been developed with better accuracy than their open source counterparts, but are expensive and out of reach for most developers working on small projects (Shin, Williams and Xie, 2014).

Penetration testing refers to a method of trying out real attacks on a computer system in order to identify potential security vulnerabilities. This method is usually used to test web applications and identify vulnerabilities in the web applications, the potential risks when those vulnerabilities are exploited. The challenge with doing penetration tests on web applications is that some tests are destructive, hence if successful they could take down a web application. This requires that a dedicated test instance of the web application is set up. The tests are also complicated and may be out of reach for most inexperienced developers (Deuble, 2012).

Intrusion detection systems (IDS) are usually configured with signatures that represent existing vectors. This approach has been challenged by the large number of vulnerabilities discovered daily and vulnerabilities introduced by applications developed in house, making it difficult to keep the signatures updated. Developing the signatures itself is a challenge, since it requires significant security expertise (Kruegel and Vigna, 2010). Anomaly detection systems with the ability to learn have been proposed to improve on the shortcomings of signature based IDS. Various classification methods have been proposed using neural networks and support vector machines. The existing methods rely on analyzing the analysis of SQL statements to identify the anomalous ones.

1.2 Problem statement

Injection attacks have become more prevalent in the past few years, retaining the top position in the Owasp top ten list for both 2013 (Owasp, 2013) and 2017 (Owasp, 2017). According to IBM X-Force analysis of managed security services (MSS), injection attacks are the most common attacks against organizational networks. In fact, between January 2016 and June 2017, injection attacks accounted for 47% of all reported attacks (IBM, 2017). In Q3 2017, Akamai observe SQL injections to be the most prevalent attack vector, composing 47% of the attacks detected by their systems (Akamai, 2017).

To mitigate against these and other attacks, signature based intrusion detection systems (IDS) are usually configured with the known attacks. Unfortunately, it is difficult to keep up against the changing attack landscape due to the number of vulnerabilities discovered daily and vulnerabilities introduced by internally developed software. Systems equipped with anomaly detection can supplement existing IDS tools by providing the ability to learn new attacks (Kruegel and Vigna, 2010).

Recurrent neural networks have been proposed for anomaly detection of SQL Injection attacks (Valeur, Mutz and Vigna, 2005). However, a key limitation with existing research is that they are trained with only one kind of SQL injection (tautology based), hence they are untested with other kinds injection attacks.

1.3 Objectives

The general objective is to develop a neural network model that can detect web-based injection attacks using long short term memory recurrent neural networks (LSTM).

The specific objectives are:

I. To develop a method of recording and labelling request samples for analysis and detection of malicious input requests
II. To train an LSTM model capable of detecting SQL injection attacks in web requests
III. To test different LSTM network model parameters and determine the parameters that result in the best model performance
IV. To test and validate the model's ability to detect malicious requests

1.4 Justification

The growth of web traffic and increased threats on web applications has necessitated the development of security systems that are able to adapt to the changing attack vectors. Injection attacks have become even more popular, with the Owasp top ten report ranking them as the most popular attack (Owasp, 2017).

This project will provide additional security for web applications, enabling developers and businesses have stronger mitigation against these kinds of attacks. This project can also be used to complement existing security measures, therefore improving the security of web applications against these kinds of attacks.

This research will be helpful for developers of security software by providing them with a method they can implement in their systems to mitigate the risk of injection attacks. This will make their software more effective while guaranteeing stronger security for their clients.

With improved security, the growth of the World Wide Web and its use for sensitive transactions such as financial transactions will be sustained. Improved user confidence can also be achieved if businesses can assure users that they have deployed adequate security measures in their applications.

Chapter 2: Literature Review

2.1 Introduction

This chapter surveys the nature and different types of injection attacks and the popularity of each subset of these attacks. It then studies the various methods of detecting injection attacks, their strengths and drawbacks, then narrows down to machine-learning based methods of detecting injection attacks and proposes a solution to detecting injection attacks.

2.2 Injection attacks

Injection attacks refer to when untrusted data containing rogue SQL queries, OS or LDAP commands are passed to a computer system. This untrusted data may result unprecedented consequences when run leading to data loss or leakage (Owasp, 2017).

There are several kinds of injection attacks, the most common ones being SQL injection and OS command injection. About 83% of the reported injection attacks are OS command and SQL injection attacks. The figure below shows the prevalence of the different kinds of injection attacks as analyzed by IBM in 2007.

Operating system command injection involves an attacker injecting existing operating system commands into the application functions. This could allow an attacker to escalate privileges or run arbitrary commands with diverse consequences.

This attack vector is also common, and can cause adverse effects when successfully executed on a system. This attack vector is mostly used to escalate privileges and gain access to other parts of a system that would otherwise not be accessible to the attacker.

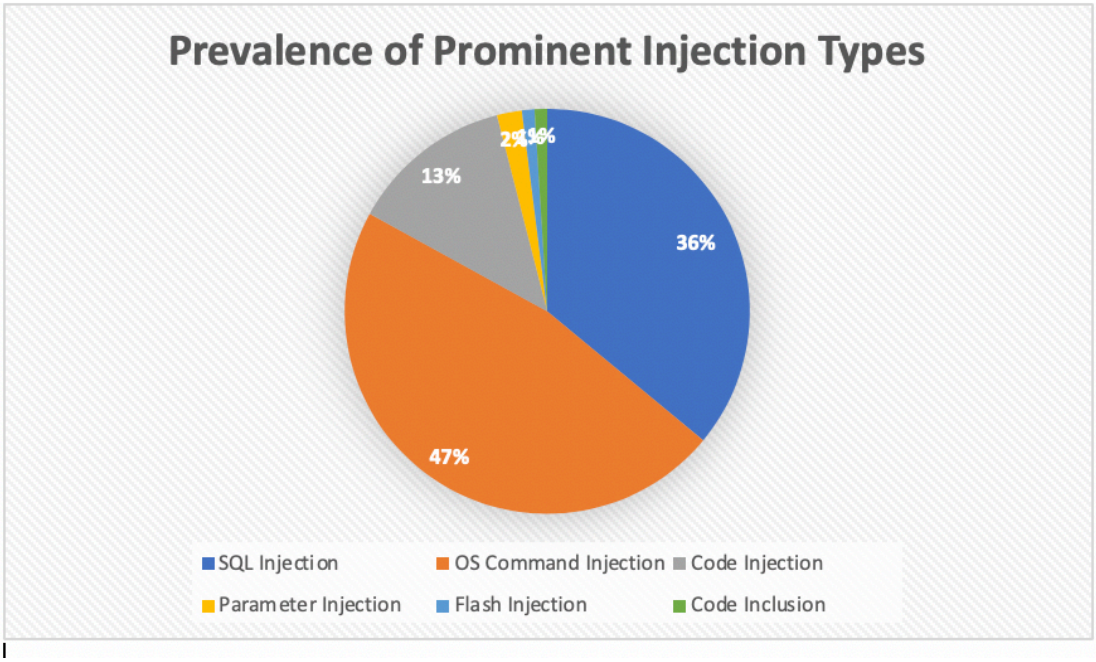

Figure 1: Prevalence of the types of Injection Attacks (IBM, 2017)

2.2.1 SQL Injection

This is when an attacker passes SQL commands that could modify application behavior and run arbitrary SQL queries on its database. These queries could expose confidential data, add or modify data without following proper authorization.

As a result of the popularity and level of risk of SQL injection attacks, a lot of research has been done and as result comprehensive formal definitions of the various types of SQL injection attacks are available. These works also prescribe tools to detect and prevent these attacks (Ray and Ligatti, 2014). However, these tools and methods require a skilled developer to employ, and may not be useful to secure applications that are already developed with these vulnerabilities.

There are seven popular types of SQL injections. These give an overview of the common tricks attackers use to modify queries to result into undesired results.

I. Tautologies

These are where the attacker attempts to bypass authentication or application controls by using an *'or'* in the injected query to introduce a tautology. These usually introduce an *'or'* condition that evaluates to true (Sheykhkanloo, 2015).

II. Illegal or logically incorrect queries

These are where the attacker tries to identify vulnerable parameters to extract data from the database by providing incorrect queries. These mostly result into errors that can reveal important details about the backend database or data in the database (Sheykhkanloo, 2015).

III. Piggy backed queries

This is where the attacker aims extract or modify data or execute arbitrary commands on the backend by executing multiple statements where only one statement would usually allowed. This is usually done by embedding a query inside another query and even commenting out a part of the intended query. The signature of such an attack is the presence of the delimiter ";" in the query (Sheykhkanloo, 2015).

IV. Union query

This is an attack where attackers bypass authentication, application controls or extract data from a backend database by combining two or more queries using a "*union select"* statement (Sheykhkanloo, 2015).

V. Stored procedures

These are attacks that attackers attempt to run stored procedures on a given database in order to result into unintended consequences like privilege escalation (Sheykhkanloo, 2015).

VI. Inference attacks

This is an attacks where attackers test for the possible vulnerabilities of a database when even where no data is returned to a user. An example of such an attack would be one used to test whether the current user a system user (Sheykhkanloo, 2015).

VII. Alternate encoding

Alternate encoding attacks are where an injected command is obscured by using encoding methods. The commonly used encodings is hexadecimal. These can be used to bypass validation, and can obscure other kinds of attacks (Sheykhkanloo, 2015).

2.3 Prevention and detection of injection attacks

To protect applications from SQL injection attacks, different methods have been prescribed to either detect the attacks as they happen, or prevent them from happening. Static analysis and secure coding practices have been used to prevent applications from being vulnerable from injection attacks, while signature based and machine learning based methods are used to detect injection attacks in applications.

2.3.1 Static analysis and secure coding practices

Injection attacks usually arise from unchecked user input which causes undesired consequences by running unintended commands on a server or a database. Consequently, one way of protecting a server or a database is by using secure coding practices which sanitize user input before it is merged with a command or an SQL statement to ensure the intended command or query is run, and not one modified by an attacker. Static analysis and secure coding practices represents methods to preven:t injection attacks, though does not include detection of the attacks.

Static analysis refers to techniques where code is automatically reviewed by a program to identify sections of the code with potential vulnerabilities, for example where user input is merged into an SQL query without first properly sanitizing it, or where old vulnerable libraries are used instead of their newer versions that have these vulnerabilities patched. Static analysis helps enforce secure coding practices, by identifying the sections in code that developers may have used insecure practices. This combined with effective manual code reviews can be used to eliminate most insecure code practices and help fortify web applications against injection attacks among other attack vectors (Livshits and Lam, 2005).

Novice programmers may not know of these secure coding practices, hence leaving applications developed by these inexperienced programmers vulnerable. Many of the static analysis tools out there are commercial applications that are expensive to obtain. This bars most of the programmers working in a non-institutional setting from obtaining access to these tools, hence leaving them to develop applications without this crucial step that could bring assurance to the quality of the application (Livshits and Lam, 2005).

Static analysis is limited to merely identifying potentially unvalidated and non-sanitized input, however do not go into the detail of checking the correctness and completeness of the validation measures, hence there is a risk of leaving improperly sanitized input that may still be vulnerable to attacks (Bandhakavi *et al.*, 2007).

2.3.2 Signature based injection detection

Signature based methods match requests to known attack patterns or signatures in order to detect known attacks. This is the method of detection that is most popular with web application firewalls and intrusion detection systems available today. These devices inspect web traffic and attempt to match attack signatures in the web traffic (Maor and Shulman, 2005).

Snort and Suricata are open source softwares that use this method to detect injection and other attacks. Snort uses rules that contain the signature and define what happens when that signature matches a given web request. Alnabulsi, Islam and Mamun propose five Snort rules that detect SQL injection attacks. The five rules achieve better precision and recall measures than existing studies done for the same at the time (Alnabulsi, Islam and Mamun, 2014). Signature based injection detection systems have shown a weakness that they are easy to evade, and rules are usually written for simple SQL injection methods such as tautology based SQL injection. Maor and Shulman have in their paper discussed several methods that can be used to trick signature based injection detection, and effectively run SQL injection on servers protected by these methods (Maor and Shulman, 2005).

Signature based methods require that rules or signatures are written for each known attack vector, hence suffer the weakness of being vulnerable from unknown attacks. Given the pace at which new attacks are discovered, signature based methods can easily be circumvented by using new attack methods that signatures may not already be written for (Kruegel and Vigna, 2010).

2.3.3 Machine Learning Based Injection Detection

Machine learning methods have been implemented using various algorithms for the detection of SQL injection attacks. The two common machine learning methods used to detect injection attacks are natural language processing methods and anomaly detection methods.

I. Natural Language Processing Methods

This includes methods that use language technology to detect SQL injection attacks. A method of using a graph of tokens and support vector machines (SVM) to detect SQL injection attacks has been proposed. In this approach, SQL injection is detected by modelling SQL queries as a graph of nodes then using the centrality measure of the nodes to train a support vector machine (SVM). The approach consists of four main steps, converting an SQL query into a sequence of tokens while preserving its structural composition, generating a graph of tokens as nodes, training an SVM and finally using the SVM to detect malicious queries. This method is tested with five applications gives results of over 99% for both accuracy and precision (Kar, Panigrahi and Sundararajan, 2016).

Parse tree validation has also been suggested for the detection of SQL injection attacks. A parse tree is a method of representing a statement to show the grammatical structure of that statement in its language. This technique compares two parse trees to determine if their structure is the same. When the original and the injected query are compared and difference in the structure is noted then we can conclude that an injection has been detected (Buehrer, Weide and Sivilotti, 2006). The drawback of this approach is that parse trees of each SQL query run in a system will need to be recorded so that they can be compared with the parse tree of an incoming query to determine whether they are alike or not. This reduces the practicality of this method especially in complex systems that run many different queries.

II. Anomaly Detection Methods

These methods model injection attacks as an anomaly detection problem, where normal SQL queries is classified differently from the injected queries.

Anomaly detection is the identification of instances that stand out to be dissimilar from the rest in a dataset. Anomalies can represent errors or indicate a new underlying process (Chalapathy, Menon and Chawla, 2018). This area has recently been the subject of many studies due to its applicability in engineering areas such as sensor failure, network monitoring, cyber security and surveillance (Ergen, Mirza and Kozat, 2017). Anomaly detection is an example of a one-class classification problem, where given data points originated from a single class but contaminated with outliers, you find the class boundary to separate the outliers from the class members (Pauwels and Ambekar, 2013).

Support Vector Machines (SVM)

One-Class Support Vector Machine (OC-SVM) is a popular supervised approach to detecting anomalies. This works by constructing a smooth boundary around the majority of the probability mass of data (Rawat and Shrivastav, 2012). Rawat and Shrivastav propose a method of using SVM to classify SQL statements into either the original query or the suspicious query. The original query is the intended query with the right parameters added into it. The suspicious query is one sent by an attacker to produce undesired results. The model achieves a detection time of 15ms and an accuracy of 94.67%. These are good results, however, the limitation of their study is that they experiment with only tautology based SQL injections, hence the model may not be very accurate with other types of SQL injection. This can be improved by having other kinds of SQL injection in the training and testing dataset, to ensure the model is balanced among the known kinds of SQL injection (Rawat and Shrivastav, 2012).

An adaptive method of detecting malicious queries in web attacks has been proposed by Zhang and Dong. The study uses an SVM hybrid to classify query strings as malicious or not. The study has the advantage of being adaptive, that is unknown queries are logged and then later labelled and used to train and improve the SVM model. This strategy ensures that the model is always updated with new attack vectors and can potentially result into better performance when deployed in real life scenarios (Dong and Zhang, 2017).

The accuracy of SVM is greatly impacted with choosing the correct kernel function. The correct kernel function will greatly impact the ability of the model to classify the input data. Also choosing the appropriate hyperparameters that will allow for sufficient generalization performance is usually a big challenge for SVM models. Support vector machines, like all other non-parametric algorithms suffer from the lack of transparency in results, since the output cannot be presented as a simple parametric function (Auria and Moro, 1998).

Hidden Markov Method (HMM)

Hidden Markov Model (HHM) is a powerful method for modelling statistical sequences. The system being modeled is assumed to be a Markov process with unobserved (hidden) states generating an observable state sequence (Kar *et al.*, 2016).

The approach consists of normalizing a query into a sequence of tokens, then using this as the observation state sequence for input into two sets of HMM ensembles. The HMMs in each ensemble are then trained with both safe and unsafe queries, and each of their output combined to give the overall output (Kar *et al.*, 2016).

This method achieves an accuracy of 99.57% (Kar *et al.*, 2016). The major drawbacks of this models are that they are expensive in terms of compute time, which could limit their large scale use. The implemented model adds a detection time for each query, hence for a page with 10 queries this could lead to an overhead of 100ms which is detectable to the user (Kar *et al.*, 2016) On a large scale system, this would be expensive to implement and could lead to significant delays.

The study (Kar *et al.*, 2016) implements an application that would be used in the database firewall layer (between the application and the database). The major drawbacks of this approach is that the database firewall will likely have to process a lot of queries that are not affected by user input, hence cannot be injected. This further leads to high overhead and latency in processing queries.

Recurrent Neural Networks

Neural networks have been applied in many classification problems. Neural networks have the advantage of superior performance over complex, highly dimensional data sets as compared to support vector machines (Chalapathy, Menon and Chawla, 2018).

Kruegel & Vigna propose a system based on anomaly detection of Web based attacks. Their approach focuses on analyzing HTTP requests, specifically the GET requests that use parameters to pass values to server side programs. Their anomaly detection process analyses a number of different models to identify anomalous requests associated with a certain program. They test for the effects of using attribute length, attribute character distribution, structural inference, token finder, attribute presence or absence and attribute order to the detection of anomalous requests. A different learning method is used for these attributes and their results compared at the end. The length, character distribution and structural inference models are seen as very effective in detecting anomalous requests (Kruegel and Vigna, 2010). A similar approach is also proposed for SQL Injection attacks by Valeur, Mutz and Vigna. Their approach relies on using multiple models to characterize normal SQL behavior. The model is

able to detect SQL injection attacks with high accuracy and low performance overhead (Valeur, Mutz and Vigna, 2005). These approaches can be improved by doing a combined analysis of web requests and SQL queries. This approach further reduces the false positive rate by adding more features used for the anomaly detection (Kruegel and Vigna, 2010).

Recurrent neural networks are able to learn from both current and previous inputs, which can potentially improve predictions that depend on previous dependencies. This is very useful for problems that can be modelled in time series, such that a current event is dependent on a previous event and a trend can be observed over a time period (Bontemps *et al.*, 2017).

An alternative method of using recurrent neural networks (RNN) to the detect SQL Injection attacks has been proposed. In this approach, the problem of detecting anomalous queries if transformed into a time series prediction problem. The queries are divided into tokens, which are sent into the detection system to predict the next token using the previously seen tokens. In learning, the RNN is trained by back-propagation through time (BPPT) algorithm (Skaruz and Seredynski, 2007).

Long short term memory networks are an improvement of vanilla recurrent neural networks that were built to solve the issues of exploding and vanishing gradients, enabling these networks to be able to learn complex long term dependencies over longer times. This enables them to better relate current events to previous events, increasing performance prediction for such problems (Bontemps *et al.*, 2017).

LSTM recurrent neural networks have been widely proposed for Anomaly Detection. Non-collective approaches usually view an anomaly as a single point. In these approaches, the detection models cannot use information from previous points in time to evaluate the current. Long short term memory networks (LSTMs) are have produced excellent performance in time series problems due to their ability to relate current events with previous events (Bontemps *et al.*, 2017).

LSTM recurrent neural networks have shown good results in similar problems. Skaruz and Seredynsky have used RNNs to detect anomalous SQL queries from normal queries by encoding the SQL keywords in SQL statements and then modelling the queries as a time series problem, where recurrent neural networks excel (Skaruz and Seredynski, 2007). The same

process can be useful with Command and SQL injection attacks, allowing us to take advantage of the excellent performance of these networks and get superior predictions.

LSTM recurrent neural networks have been proposed for sequence aggregation rules in network traffic. The study aims to detect various attacks by using sequence modelling and two bi-directional LSTMs. The study achieves an area under the curve (AUC) of 0.71 for all the attack types. The study demonstrates the strength of LSTM in anomaly detection problems, having achieved this performance for a classification of over 10 different types of attacks (Radford, Richardson and Davis, 2018). This work could be improved by focusing the model on specific attacks, hence achieving even better results.

2.4 Long Short Term Memory Networks

Recurrent neural networks are an improvement to the traditional feed-forward neural networks that have feedback loops within the nodes of its layers. This enables them to use current input as well as their previous output when evaluating the current output. This enables the recurrent neural networks to have short term memory, since it can use its previous output. As the weights are updated in the neural network nodes, old outputs are increasingly have less impact on the current input, hence recurrent neural networks can be said to have short term memory.
Since simple recurrent neural networks have short term memory, they are unable to relate long terms dependencies, where the cause and effect relationships are separated by long time differences. This makes them result into poor performance where long term dependencies are required to make a correct prediction.

The continuous weight updates result into a common challenge with simple recurrent neural networks, that of exploding and vanishing gradients. When the gradients are continuously multiplied by a factor greater than 1, they tend to quickly explode due to the compounding effect, and when this factor is less than 1 the gradients will vanish.

Long short term memory (LSTM) networks are a subset of recurrent neural networks that present a solution to the problems of recurrent neural networks. The LSTM networks solve the problems of exploding and vanishing gradients as well as that of long term dependencies by introducing long term memory in the cells of the recurrent neural networks.

An LSTM cell is made up of three gates. These gates control the information flow in the cell. These are the forget gate, the input gate and the output gate. The LSTM cell maintains a state or memory that can be used when evaluating the output. This state is what gives the cell the long term memory, enabling it to comprehend long term dependencies. The figure below shows the LSTM cell structure.

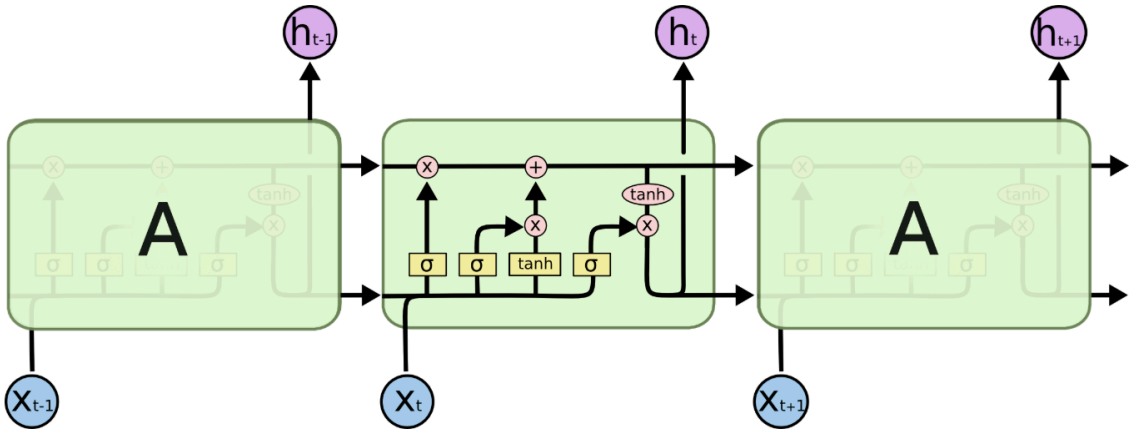

Figure 2: The structure of an LSTM cell when unfolded in the time domain (Olah, 2015)

The top vertical line in the cell state, that can be denoted by C_t. The current input (X_t) goes into the cell through the forget gate. The sigmoid in the forget gate determines whether to delete information from the cell state. The sigmoid output is a number between 0 and 1, which determines how much information is retained in the cell state.

The next layer (input gate layer) determines what new information will be stored in the cell state. The sigmoid layer of the input gate determines which values will be updated, while the tanh creates a vector of candidate values that will be updated. These are then combined at the pointwise multiplication stage and added to the state by the pointwise addition stage.

The final decision is what to output, which is done by the output gate layer. The output will be a filtered version of the cell state. The sigmoid determines what part of the cell state will be pushed to the output. The tanh is used to limit the values fetched from the state to between -1 and 1. The output of the tanh is multiplied with the output of the sigmoid to give the output, denoted by h_t .Since these are cells of a recurrent neural network, the current output will be used as input at the next time frame t+1, which is repeated until the network finishes evaluating (Olah, 2015).

LSTM networks benefit from being able to evaluate long term dependencies, hence have shown great potential in evaluating time series problems. This is because the cell state enables them to relate cause and effects that may be separated by long sequences, which simple recurrent neural networks are not able to evaluate (Hochreiter and Schmidhuber, 1997).

2.5 Gaps in Literature

There are diverse works done in SQL Injection detection using signature detection and machine learning methods such as SVM, Hidden Markov model and recurrent neural networks, including LSTM Networks. However, there are key improvements that can be done to these studies to improve their performance and build a better detection model that can be used in practical applications.

The models reviewed are usually skewed towards detection of tautology-based attacks, while there are other equally popular SQL injection attack types. This is evident in the works of Rawat and Shrivastav, who work on an SVM model for classifying SQL statements. These works can be improved by using a more balanced dataset to train the model, hence ensuring that the model works for all other SQL injection types (Rawat and Shrivastav, 2012).

The studies explored in this chapter focus on detecting SQL injection on the database firewall layer. This approach faces several challenges including the firewall having to classify all SQL queries, including the ones that do not include user data hence are not vulnerable to injection. Web application firewalls are already popular, and a solution that can work on this layer would be more practical for most web applications, in addition to being able to be trained to detect injection attacks among other attacks. Radford, Richardson and Davis have worked with this approach and they have been able to detect various kinds of attacks (Radford, Richardson and Davis, 2018).

Adaptive learning techniques benefit from being able to learn from new attacks when they occur. In adaptive techniques, the requests observed during detection stage are logged and later used to further train the model, hence the model improves with time and is able to learn any new attacks that may come (Dong and Zhang, 2017). This presents a good way to improve our model, by having the model learn from the samples observed during detection.

Most studies explored have achieved good accuracy and precision, with values up to 99% seen in the SVM model by Rawat and Shrivastav, even though the scope of detection is limited. The challenge with these models would be to retain the accuracy but improve the scope to include all kinds of injection attacks.

2.6 Proposed Solution

The proposed solution will consist of an SQL injection detection middleware that will protect a target application. The middleware will be situated before the application and all requests going to the application will first be passed through the injection detection middleware. The middleware is capable of classifying injection attacks from safe requests, will allow safe requests and block the unsafe request containing attacks.

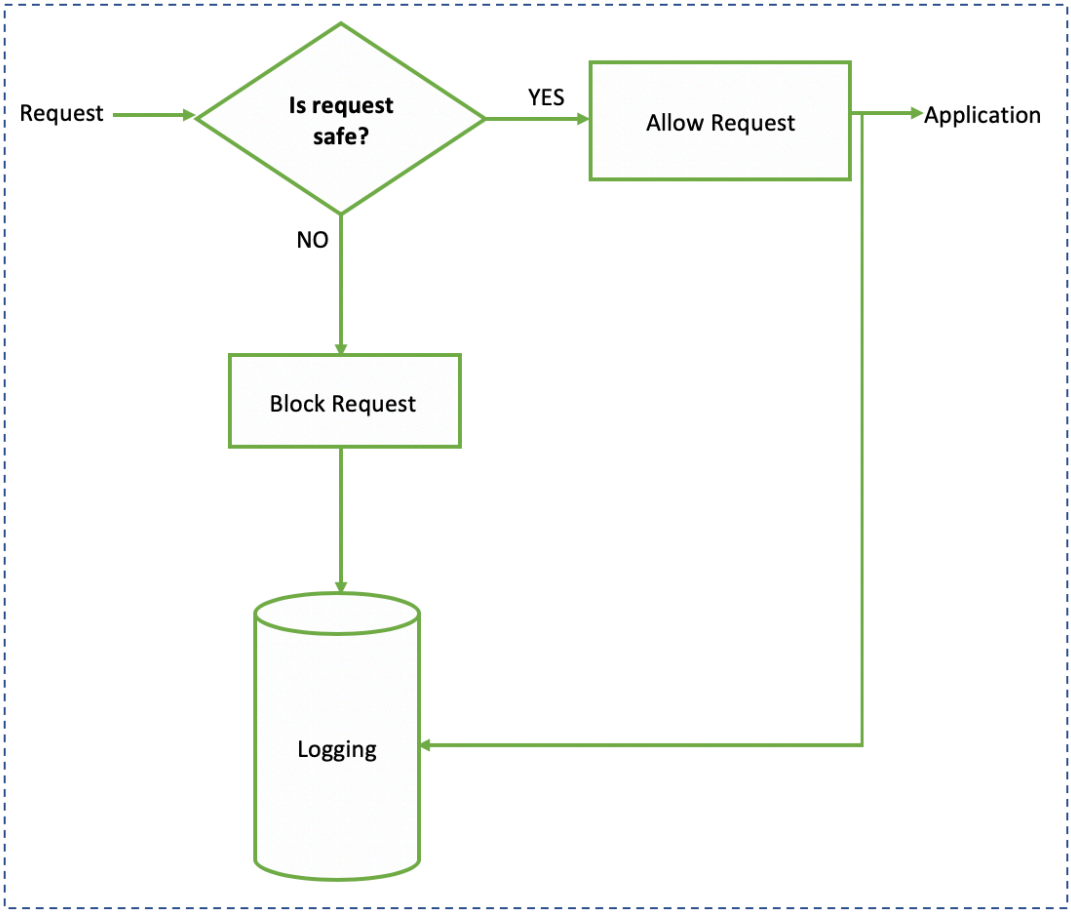

Figure 3: Web request middleware during detection

The middleware will also push all the requests, together with the decision made on the requests to a log file. This log file can then be analyzed to determine the model performance and then to retrain the model to improve its performance.

Chapter 3: Methodology

3.1 Introduction

This section covers the step by step guide taken to execute the research. This includes the collection of data, cleaning and feature extraction, train the model and testing and validation of the model.

This study takes an experimental research design. Experimental research refers to a strategy that investigates cause and effect relationships. It seeks to prove or disprove a causal link between a factor and an observed outcome (Oates, 2006). The main objective of this research is to build a neural network model using long short term memory (LSTM) recurrent neural networks, and prove or disprove their ability to be used to detect injection attacks.

In the course of achieving this, we built a method of collecting request samples for our training data then used various request sources to collect both normal and injected requests. We then cleaned the data collected and extracted the features required to train our LSTM model. We then used this data to train the model and ran experiments with various LSTM network parameters and recorded the results to determine the best model performance. The results were recorded and analyzed to determine whether the model was successfully able to detect injected requests.

3.3 Data Collection

To train the model, we needed to collect typical normal and injected requests that a web application can receive. This data was collected by logging requests going through a middleware embedded in a web application. This was achieved by modifying the web application to record all requests made to it to a log file, which was then used for further processing.

For this research, we needed a collection of normal and injected requests. The injected requests needed to have a mix of the various kinds of injection attacks so as to ensure the model is trained to detect all kinds of injection.

To obtain this data, we used the following tools:

I. An SQL injection testing tool (*Sqlmap*) – This is a commonly used tool to test web applications for injection attacks. It generates different kinds of attacks to test for various vulnerabilities. All tests available on *Sqlmap* were run with the DVWA application as the target. The procedure involved downloading and installing *sqlmap*, then running the command "`python sqlmap.py -u http://dvwa.local/vulnerabilities/sqli?id=1`". This command generates the attacks and sends them to the application. The sqlmap program replaces the id parameter in the url with the attacks.

II. A python script that generates both safe and unsafe requests. This script was develop to produce a mix of all the eight types of SQL injection mentioned in chapter two hence would be useful to ensure the collected dataset is balanced. The script was run and its output is requests that are sent into the target application.

For the target application, we adopted an instance of the damn vulnerable web application (DVWA). DVWA is an application built in PHP that exposes multiple vulnerabilities. This applications is aimed to provide an environment for testing of ethical hacking skills and also provide a platform for software developers to learn how to secure their web applications. For our case, we made modifications to the application to log the requests made to the application together with label information on whether the request is an attack or is legitimate.

Collection of the requests was done by first running *sqlmap* on the application then running the python script to send additional requests to the applications. A header was used to label the requests from the script as secure or not secure, while all requests from *sqlmap* were labelled as not secure.

3.4 Data Cleaning and Feature extraction

In this stage, we were interested in dimension reduction to find the set of minimal features that best classifies the data (Staudemeyer and Omlin, 2014).

The data collected contained the following attributes, whose relation to the injection attacks was not known:

I. The request method
II. The request headers
III. The target URL of the request
IV. The input data
V. The source IP address
VI. The source user agent

Furthermore, the data contained a complicated structure that could be reduced to make it easy to train the model with the dataset. We then singled out the important information from the dataset, which was the 'secure' header to serve as the label, and the input data, which is the text used to train the model.

The dataset was reduced to the two parameters, the content and the label, which was then saved to a csv file that would be used to train the model.

3.5 Training and experimentation

3.5.1 Data preparation

The cleaned data obtained after feature extraction was then used for training the model. This being a classification problem on supervised model, the logs were labelled that indicate whether their content was secure or not. A 'secure' header was added to each line of the CSV file, and was labelled with 1 for secure content, and 0 for content that was not secure.

The dataset was then divided into the two samples. We used the first 90% of the requests were used as the training sample and the last 10% was the test sample.

To train a neural network using text training data, the data was transformed into a numerical form which is fit for training in a neural network. In this case, the text was converted into a sequence of numbers. This was achieved by writing each unique word in the training sample into a dictionary, then using the indices of the words in the dictionary as their numeric equivalent. This produced a sequence of numbers used to train the model.

3.5.2 Model Development

The model was developed using TensorFlow's implementation of LSTM. TensorFlow provides an implementation of most machine learning algorithms that developers can use to implement their models.

Models in TensorFlow are usually implemented in layers containing various components. Our model consists of the following components:

I. Embedding layer for text processing and shaping into the input shape required by the LSTM
II. LSTM layer, containing the neural network
III. Dense layer for output shaping
IV. Activation and dropout layers that convert the continuous output into binary output that is expected of our binary classifier

The model was trained using the training data prepared as per the section above, and then evaluated using the test data samples we had set aside.

3.5.3 Running the experiment

We trained the model using various LSTM model parameters so as to determine the best parameters we can achieve with the model. Training was done using the training sample of the dataset, then the test sample was used to test the model by doing detections and comparing with the labels. The performance of the model was then recorded for comparison across different parameters. The parameters that were tested are:

I. Set up training with random hyper-parameters
II. Vary the parameters comparing performance
III. Perform k-fold cross validation and measure final model performance

The first element of this experiment was to train the model using assigned parameters, then measure the performance of this model. The LSTM model was set up and trained using the dataset, now divided into 90% training instances and 10% testing instances.

To determine the best model parameters, we varied the parameters while comparing the performance of the models with the different parameters. For each parameter, a confusion matrix was built and receiver operator characteristics chart was made. These were then

compared for the selection of the best model. The following variations of the parameters were considered:

I. Varying the LSTM blocks between 4, 16, 64 and 128 blocks
II. Using 1, 2 and 3 hidden layers

The two parameters were varied and the results of the confusion matrix and the receiver operator characteristics (ROC) chart recorded. The best model performance was then picked using the area under the ROC charts and the performance parameters from the confusion matrix.

The next step was to perform cross-validation, so as to determine the model performance using k-fold cross validation. This was done by dividing the data into 10 equally sized (or nearly equal) datasets, then using each of these sets as the validation set and the rest of the dataset as training dataset. The performance of the model was then taken as an average of the performance parameters from the confusion matrix.

3.6 Testing and validation

At this stage, we tested the model to determine its performance. The trained model was given real data to classify and the results then saved for analysis. The performance parameters we looked to collect in this experiment were:

I. Confusion matrix
II. Receiver operator characteristics
III. Cross validation

3.6.1 Confusion matrix

In classification problems, the performance of a model can be obtained from a confusion matrix. The table below shows the confusion matrix that was used in this research.

Table 1: Confusion matrix

	Predicted	
	Secure	**Not Secure**
Secure	True positive (TP)	False Negative (FN)
Not Secure	False Positive (FP)	True Negative (TN)

From the confusion matrix, the following performance measures can be obtained:

I. Accuracy
II. Recall (Sensitivity or True positive rate)
III. Precision (Positive predictive value)
IV. Specificity (True negative rate)
V. False positive rate

From the confusion matrix, the accuracy of the model can be calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

The accuracy is also referred to as the calculation rate, and is a measure of how the different instances are classified correctly (Staudemeyer, 2015).

Recall, also referred to as sensitivity or true positive rate refers to the portion of positive instances correctly classified as positive. It is calculated as:

$$Recall = \frac{TP}{TP + FN}$$

Precision refers to the probability that an instance is classified correctly. The precision score is calculated as

$$Precision = \frac{TP}{TP + FP}$$

The specificity or true negative rate refers to the portion of negative instances that are correctly predicted as negative. It is calculated as:

$$Specificity = \frac{TN}{TN + FP}$$

The False Positive Rate (FPR) is an important metric for this model that determines the quality of the classification. This is because the false positives represent legitimate users that are flagged as making malicious requests. For high throughput applications, this could mean a lot of legitimate users not having access to the system. The False Positive Rate (FPR) can be calculated as below.

$$FPR = \frac{FP}{TN + FP}$$

It is desirable that the False Positive Rate remains as low as possible, to minimize the number of mis-classified legitimate requests.

3.6.2 Receiver Operating Characteristic Analysis

The receiver operating characteristic (ROC) analysis evaluates the performance of an algorithm over a range of possible operating scenarios. The ROC graph will be a plot of the true positive rate (TPR) (y-axis) against the false positive rate (FPR) (x-axis) at various classification thresholds. To compute points in the ROC curve, the model will be evaluated at different threshold values and the TPR vs FPR plotted.

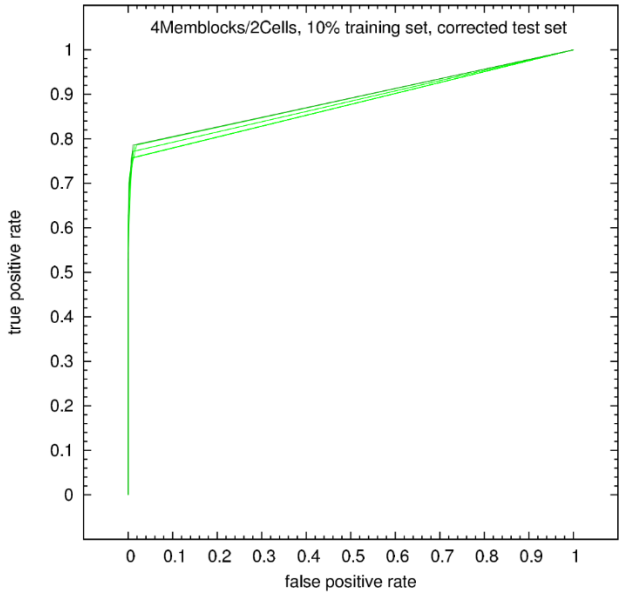

Figure 4: Sample ROC chart

The two-dimensional area under the curve provides an aggregate measure of performance across all thresholds. It is a threshold invariant measure, since it ranks the quality of the models prediction regardless of the classification threshold chosen (Staudemeyer, 2015).

3.6.3 Cross Validation

Cross validation is a method of evaluating machine learning models by dividing data into two parts, one to train and the other to test the model. In typical cross-validation, the test and training instances cross-over such that each instance will be used as test or training in subsequent rounds.

In k-fold cross validation, the data is divided into k equal or nearly equal segments or folds. For each of the k folds, one of the fold is used as the validation (test) set while the rest of the k-1 sets are then used as the training or learning set. The performance of the model is then taken as an aggregate of the performance over the k fold. Different aggregation measures such as average can be used for this (Payam, Lei and Huan, 2008).

In this study, we performed a tenfold cross validation. This was done by dividing the dataset into ten nearly equal parts, with the samples that go into each fold randomly selected from the dataset. Each of the folds was then designated as a test sample and the remaining dataset used to train the model. This was repeated for all folds, and the final performance calculated as the mean performance of each of the ten folds.

Chapter 4: Results and Discussion

4.1 Introduction

The main goal of this research is to collect data and train an LSTM model that is able to classify requests containing injection attacks. In chapter 3, we collected the training and test data, improved the data and used it to train the model. We then evaluated the model, noting the key performance characteristics of the model and also determined the best network parameters for the highest performance of the model.

This chapter outlines the results from conducting the experiment as described in chapter 3. It then goes ahead to discuss these results, what they mean and comparison to similar studies done.

4.2 Data Collection

The data for this study was generated as described in the methodology section. Using the two methods 11,093 requests were collected, 42% of which were generated from *sqlmap* while the rest came from the custom python script. In this dataset, 41% were labelled as secure and 59% were labelled as not secure.

Below is a sample of 10 requests in the dataset.

Table 2: Sample requests in the dataset

Content	Source	Label
-1221")) OR ELT(9911=1136,1136) AND (("YbYz"="YbYz	sqlmap	not secure
1]-(SELECT 0 WHERE 1518=1518 AND 4093=3287)|[1	sqlmap	not secure
1'||(SELECT RQUC WHERE 8367=8367 AND 1975=1975)||'	sqlmap	not secure
1))) AND 3144=(SELECT (CASE WHEN (3144=3144) THEN 3144 ELSE (SELECT 5593 UNION SELECT 2035) END))-- rtUY	sqlmap	not secure
'; sp_execwebtask() --	script	not secure
Suzanne Torres'; drop table Steven Ramos; --	script	not secure
Marks'; insert into Jenkins set Clark = (354)200-0039x6685 where Gonzalez = 978-0-435-11180-9; --	sqlmap	not secure
economic	script	secure
program	script	secure
'; exec(Michelle Russell) --	script	not secure

This dataset was distributed among the injection types as follows:

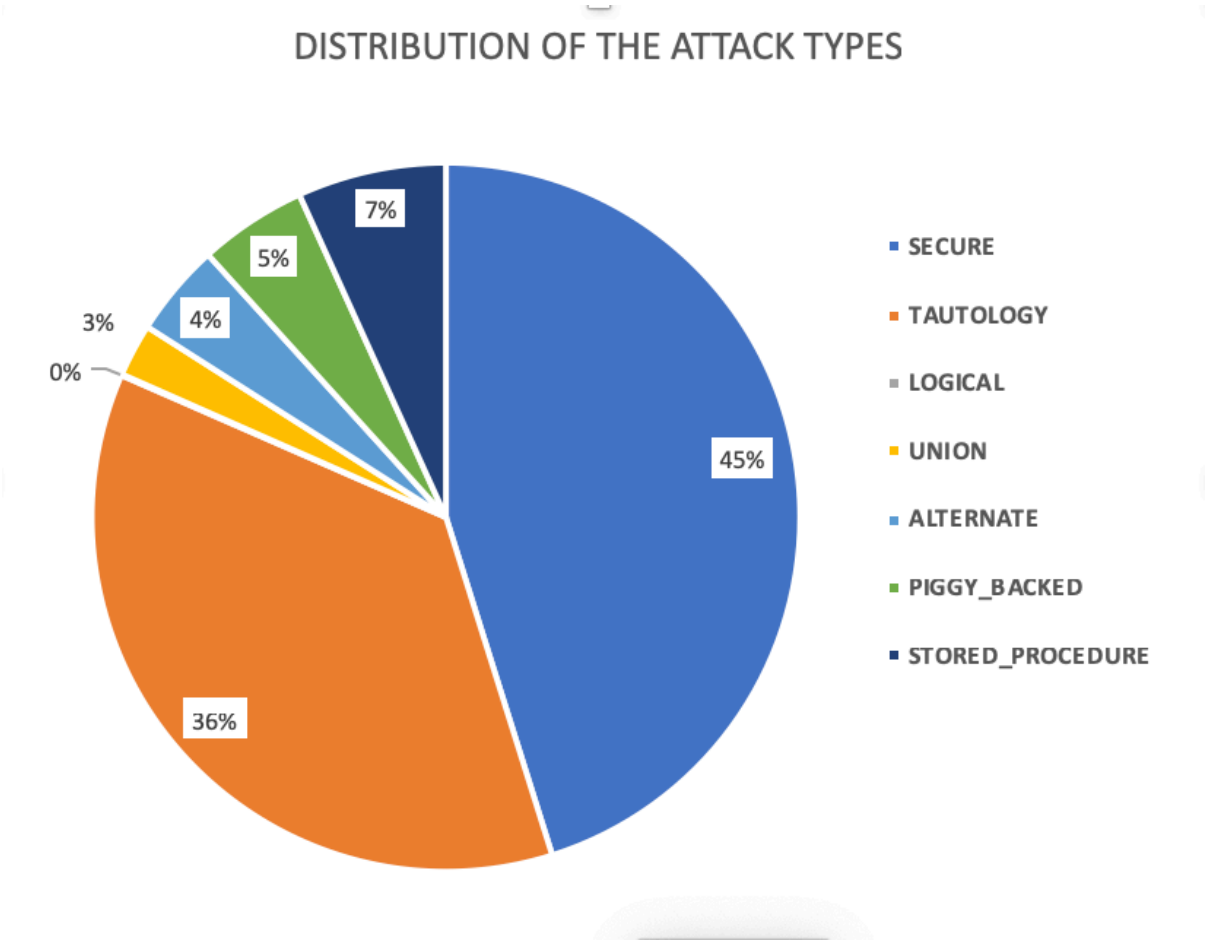

Figure 5: Distribution of the different attack types in the dataset

4.3 Model Evaluation Results

The model was built, then the parameters varied over the following possible values:

I. Varying the LSTM blocks between 4, 16, 64 and 128 blocks
II. Using 1, 2 and 3 hidden layers

The results for each of these was recorded in a confusion matrix and the receiver operator characteristics (ROC) chart. The best model from the confusion matrices and the ROC chart was then selected and k-fold cross validation was done to determine the final model performance.

4.3.1 Confusion Matrix

The table below shows the performance of the various models as obtained from the confusion matrices of each of them.

Table 3: Performance of the models with varied parameters

Description	TP	TN	FP	FN	Accuracy	Recall	Precision	TNR	FPR	AUC
1 hidden layer, 4 blocks	459	595	55	1	0.950	0.998	0.893	0.915	0.085	0.977
2 hidden layers, 4 blocks	460	589	61	0	0.945	1.000	0.883	0.906	0.094	0.939
3 hidden layers, 4 blocks	460	589	61	0	0.945	1.000	0.883	0.906	0.094	0.966
1 hidden layer, 16 blocks	457	595	55	3	0.948	0.993	0.893	0.915	0.085	0.956
2 hidden layers, 16 blocks	460	589	61	0	0.945	1.000	0.883	0.906	0.094	0.955
3 hidden layers, 16 blocks	454	595	55	6	0.945	0.987	0.892	0.915	0.085	0.976
1 hidden layer, 64 blocks	459	595	55	1	0.950	0.998	0.893	0.915	0.085	0.976
2 hidden layers, 64 blocks	460	589	61	0	0.945	1.000	0.883	0.906	0.094	0.941
3 hidden layers, 64 blocks	456	595	55	4	0.947	0.991	0.892	0.915	0.085	0.976
1 hidden layer, 128 blocks	0	650	0	460	0.586	0	0	1.000	0	0.327
2 hidden layers, 128 blocks	459	595	55	1	0.950	0.998	0.893	0.915	0.085	0.975
3 hidden layers, 128 blocks	460	589	61	0	0.945	1.000	0.883	0.906	0.094	0.953

Three model setups achieved the best performance, with equal measures across all of the 5 parameters. These are the ones highlighted in green in the table above, being the models with 1 hidden layer 4 blocks, 1 hidden layer 64 blocks and 2 hidden layers 64 blocks.

We note that the performance difference between the different model parameters is not very large, hence all the different variations of the model are able to sufficiently classify secure requests from non-secure requests. The exception to this is the model with 1 hidden layer and 128 blocks, which posted the poorest performance compared to the rest. The simpler models

are marginally better, pointing to the fact that the more complex ones are prone to over fitting hence increasing the number of errors.

These results show that simple LSTM networks are able to better classify the requests, compared to more complex ones. The simple networks are already offering good performance, hence as they get more complex the classification becomes less efficient due to the many weights a complex neural network needs to evaluate to make a decision. Furthermore, a simple model is more desirable compared to a complex one since a simpler model requires less compute resources than a more complex one. This makes the simpler model more suitable for large scale use.

We also observe that the model attains good recall values for all model parameters, with the selected models achieving a recall of 0.988. This is a result of the high number of true positives and low prevalence of false negatives, also supported with the low false positive rate. The high recall shows that 98.8% of the positive (labelled as secure) samples are detected correctly, hence giving a high relevance to the results.

4.3.2 Receiver Operating characteristics

The receiver operator characteristics (ROC) for each of the models were then plotted, and the area under the curve (AUC) used to compare the models. The best model was then selected as the one with the highest area. The figure below shows the ROC chart for the models:

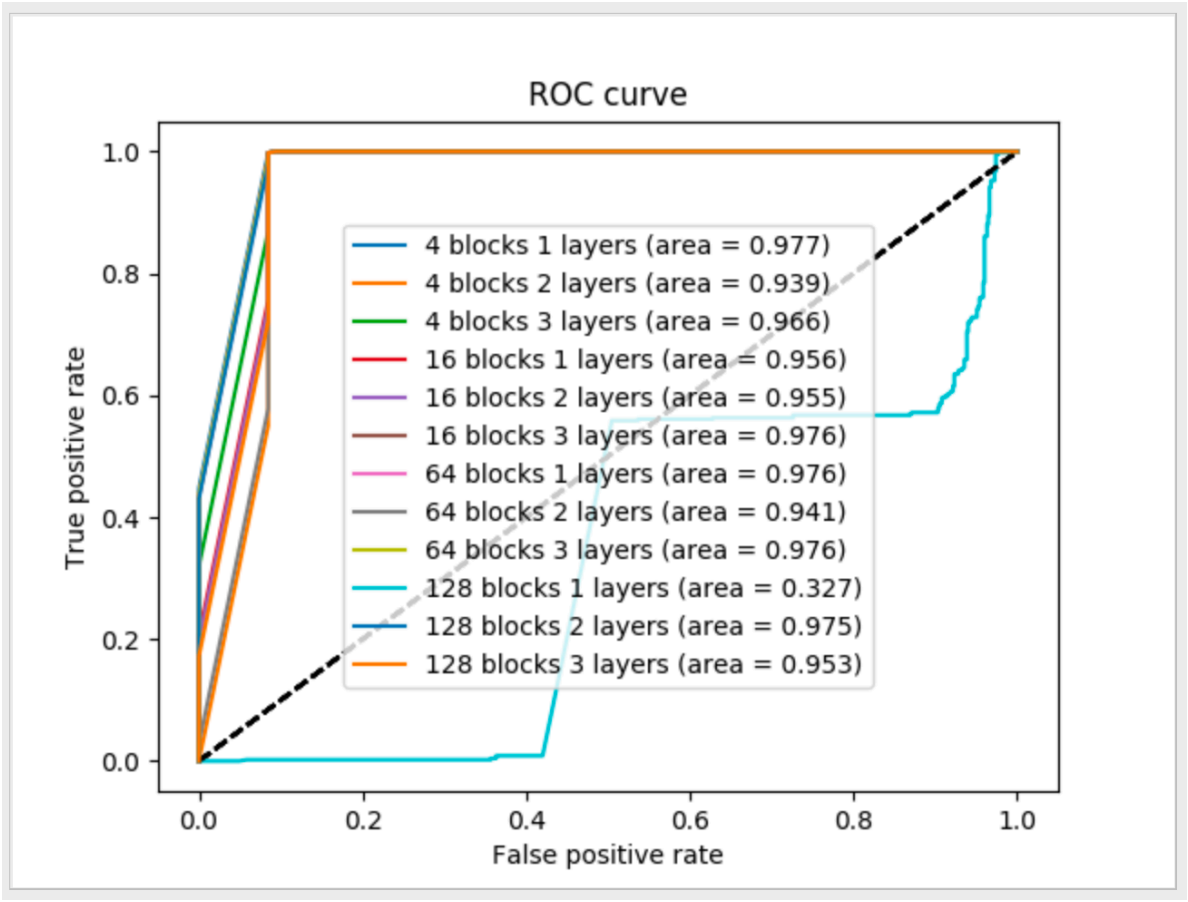

Figure 6: ROC chart comparing the various models

The best performance was obtained from the model with 4 blocks and 1 hidden layer which achieved an area under the curve of 0.977. Since this is among the best models selected from the confusion matrix performance comparison in *4.5.1* above, it was selected as the best performing model parameters. It is also the simplest model, hence it would require less compute resources to train and evaluate.

4.3.3 Cross Validation

The model selected from *4.5.1* and *4.5.2* above was then used for k-fold cross-validation to determine its performance. We divided the dataset into 10 folds then used each of them as a test set while using the rest of the training set. The results of each of the folds was then tabulated and the average taken as the final model performance.

Table 4: Cross validation results

Description	TP	TN	FP	FN	Accuracy	Recall	Precision	TNR	FPR	AUC
Fold 1	462	597	50	1	0.954	0.998	0.902	0.923	0.077	0.986
Fold 2	437	608	65	0	0.941	1.000	0.871	0.903	0.097	0.980
Fold 3	441	617	50	2	0.953	0.995	0.898	0.925	0.075	0.986
Fold 4	438	618	53	0	0.952	1.000	0.892	0.921	0.079	0.985
Fold 5	443	617	49	0	0.956	1.000	0.900	0.926	0.074	0.985
Fold 6	435	630	44	0	0.960	1.000	0.908	0.935	0.065	0.988
Fold 7	469	583	57	0	0.949	1.000	0.892	0.911	0.089	0.981
Fold 8	462	600	46	1	0.958	0.998	0.909	0.929	0.071	0.985
Fold 9	458	613	38	0	0.966	1.000	0.923	0.942	0.058	0.988
Fold 10	444	612	52	1	0.952	0.998	0.895	0.922	0.078	0.985
Average					0.954	0.999	0.899	0.924	0.076	0.985

The model performance from this method is found to be an accuracy of 95.4%, recall of 99.9%, precision of 89.9%, true negative rate of 92.4% and true positive rate of 98.5%.

4.3.4 Comparison of the detection accuracy of various injection types

The table below shows a comparison of the accuracy in detection of the different kinds of injection attacks

Table 5: Comparison of the detection accuracy of different injection attack types

Type	Accuracy
Tautology based	0.999
Illegal or logically incorrect	0.953
Piggy backed queries	0.972
Union queries	0.999
Stored procedures	0.999
Inference attacks	0.958
Alternate encoding	0.683

The model is seen to produce good accuracy with most injection types. There's however reduced accuracy for the alternate encoding types due to the difficulty differentiating this from other genuine inputs. This is a result of the nature of encoded strings, which look like a

combination of random strings. This makes it difficult for the model to differentiate these from random strings that can also be secure,

4.4 Discussion

The LSTM model was successfully built and it was able to classify requests as either injected or safe. Variation of the model parameters only resulted into small changes in the performance parameters, with an exception of one model whose performance reduced to an accuracy of 58.6%.

Over the 10-fold cross validation, the model consistently achieved an accuracy of over 94.1% showing the model was able to learn from any of the instances of the data, and the results were not as a result of features in any specific training or validation instances. The injection detection classifier reached an accuracy of 95.4%, recall of 99.99% and precision of 89.9%. The true negative rate or specificity was found to be 92.4%. This means the model classified 92.4% of the attacks correctly. The true positive rate was found to be 98.5%. The average area under the receiver operator characteristics (ROC) plot for the 10 folds was found to be 0.985. The area under the curve (AUC) determines how much a classifier is able to distinguish between classes. An AUC of 0.985 is therefore satisfactory and shows the model has a high ability to distinguish between the two classes of safe and unsafe requests.

Comparing the performance for the various injection types, it is seen that an accuracy of over 90% is obtained for all injection types with an exception of alternate encoding attacks where the accuracy is reduced. This is due to the nature of the attack strings of this type where they are seen as random strings, hence differentiating the pattern from other random strings that are not attacks is difficult.

4.4.1 Benchmarking with similar studies

The performance of the LSTM based injection attack classifier was benchmarked against similar techniques that have been used to detect injection attacks. The classifiers were benchmarked on the basis of the SQL injection types they are able to detect. Below is a table comparing the different studies and the various kinds of injection attacks they are able to detect?

Table 6: Comparison with similar studies by features

Tool	Reference	Taut.	Logically Incorrect	Union	Piggy backed	Stored procedure	Inference	Alt. Encoding
This study		Yes	Yes	Yes	Yes	Yes	Yes	Yes
SQLiGoT	(Kar, Panigrahi and Sundararajan, 2016)	Partial	Yes	Yes	Yes	Yes	Yes	Yes
SVM	(Rawat and Shrivastav, 2012)	Yes	No	No	No	No	No	No

Legend: Taut -Tautologies, Alt- Alternate

The SQLiGoT model has a weakness of being able to only partially detect tautology based injection attacks. This is a serious weakness since tautology based attacks are one of the common injection types experienced. This study not has this ability, but also has good performance over these attack types.

In terms of accuracy, the other comparable tools are only able to detect tautology based attacks, hence it would not be a fair comparison to compare the studies based on accuracy only. However, comparing the accuracy for tautology based attacks, our model still achieves better performance compared to the other two models.

Table 7: Comparison of accuracy for tautology attack detection

Description	Study	Technique	Accuracy (For Tautologies Only)
This study	-	LSTM	99.9%
SQLiGoT	(Kar, Panigrahi and Sundararajan, 2016)	Graph of tokens and SVM	99.63%
SVM	(Rawat and Shrivastav, 2012)	SVM	96.47%

From this comparison, we find that our tool has a more comprehensive ability to detect all kinds of injections. Other tools do not have the ability to detect all the other kinds of attacks. This combined with the decent accuracy, precision and recall obtained makes this tool better for detecting injection attacks.

Chapter 5: Conclusion

This chapter gives a brief conclusion to the study by outlining the achievements of this study, the contributions of this research, the challenges faced, recommendations and further work that can be done to build on this study.

5.1 Achievements

The main objective of this study was to develop a neural network model using LSTM to detect injection attacks in web requests. To achieve this, a method of collecting web requests for analysis was developed by modifying the damn vulnerable web application (DVWA) to log all requests received. The first objective of developing a method to collect web requests for analysis was thus achieved, and the requests were collected by sending requests from *sqlmap* and a custom python script to the DVWA installation. The LSTM network was then trained using Tensorflow and the data that had already been collected. At this stage, the second objective of training the neural network model was achieved. Different LSTM network parameters were tested in order to determine which one resulted into the best detection performance, and the best performing parameters were picked, hence achieving the third objective. Finally, the model's ability to detect injection attacks was tested by using a confusion matrix and cross validation. The model was found to have an accuracy of 95.4%. We can therefore conclude that the trained model was able to detect injection attacks with a satisfying performance, and thus all objectives were met.

The ability of an intrusion detection system to learn from new attacks is important in securing systems due to the changing nature of the attack landscape. The proposed system addresses this by continuously logging requests observed and their corresponding output. This can be used to further fine tune the model to achieve even better performance.

5.2 Contributions

This study contributes into the knowledge of how LSTM recurrent neural networks can be trained to detect injection attacks. It provides an improvement to the existing methods of detecting these attacks by giving a more comprehensive and accurate method that can adequately detect the various injection types. The knowledge provided in this study can be improved on and used for detection of injection attacks in real web applications, hence contributing to the safety of the internet in general.

The study also includes software that can be used to filter out attacks using any deep learning model implemented in Tensorflow. This software will be useful to researchers looking to test their models on a live environment, since they do not have to rebuild their own, instead they can focus on making their models better and reuse our program while testing.

5.3 Challenges

The major challenge faced when doing this study was the lack of a standard dataset that could be used in this study. Standard datasets are usually useful in machine learning problems since they are usually well vetted by experts to ensure that they are representative of occurrences in the real world. With standard datasets, it's also easy to benchmark a model's performance with other approaches since the data used is the same, and performance differences can only be attributed to the model itself, and not the training or test data used.

The absence of key performance parameters in most of the published similar works made it hard to compare the performance of this model with those developed in the other works, since these works did not include the performance of their models. This reduced the number of works that this study could be compared with. More comparisons would result into better benchmarking of this study to similar works.

5.4 Recommendations and future work

In this study, we collected data using two tools in order to obtain sufficient training data. Future studies can look into the development of standard datasets for injection attacks and other popular web attacks such as cross site scripting (XSS). This would be helpful to researchers in this area as they would be able to effectively compare the performance of different approaches while eliminating errors due to biases in the training data.

In future, studies can be done using other deep learning approaches in order to benchmark their performance with this study. These can also be tested in large scale deployments in order to determine their performance when used for heavy traffic web applications. Such tests will be useful in bringing these solutions closer to wide spread use in intrusion detection.

Studies can also be done on other attack types such as cross site scripting (XSS) to determine if LSTM networks are a suitable method to detect them. This will be useful in evaluating whether LSTMs can be used to build an all-round intrusion detection system.

References

1) Akamai (2017) ‘State of The Internet/Security - Q3 2017 Report’. Available at: http://www.dimtec.com/descargas/q3-2017-SOTI/security-report.pdf.
2) Alnabulsi, H., Islam, R. and Mamun, Q. (2014) ‘Detecting SQL Injection Attacks Using SNORT IDS’, (November). doi: 10.1109/APWCCSE.2014.7053873.
3) Anstee, D. (2017) *Trends in Internet Traffic Patterns*.
4) Auria, L. and Moro, R. A. (1998) ‘Support Vector Machines (SVM) as a Technique for Solvency Analysis’, *The American journal of physiology*, 275(3 Pt 1), pp. E432–E439.
5) Bandhakavi, S. *et al.* (2007) ‘CANDID: Preventing Sql Injection Attacks Using Dynamic Candidate Evaluations’, *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 12–24. doi: 10.1145/1315245.1315249.
6) Bontemps, L. *et al.* (2017) ‘Collective Anomaly Detection based on Long Short Term Memory Recurrent Neural Network’, *Agence nationale pour l’amélioration des conditions de travail*. doi: 10.1007/978-3-319-48057-2_9.
7) Buehrer, G. T., Weide, B. W. and Sivilotti, P. A. G. (2006) ‘Using parse tree validation to prevent SQL injection attacks’, (January 2005), p. 106. doi: 10.1145/1108473.1108496.
8) Chalapathy, R., Menon, A. K. and Chawla, S. (2018) ‘Anomaly Detection using One-Class Neural Networks’, (August), pp. 19–23. Available at: http://arxiv.org/abs/1802.06360.
9) Deuble, A. (2012) ‘Detecting and Preventing Web Application Attacks with Security Onion’.
10) Dong, Y. and Zhang, Y. (2017) ‘Adaptively Detecting Malicious Queries in Web Attacks’. doi: 10.1007/s11432-017-9288-4.
11) Ergen, T., Mirza, A. H. and Kozat, S. S. (2017) ‘Unsupervised and Semi-supervised Anomaly Detection with LSTM Neural Networks’, pp. 1–12. Available at: http://arxiv.org/abs/1710.09207.
12) Hochreiter, S. and Schmidhuber, J. (1997) ‘Long Short Term Memory’, 9(8), pp. 1–32. doi: 10.1162/neco.1997.9.8.1735.
13) Howard, M. and Leblanc, D. (2003) *Writing Secure Code - Practical Strategies and Techniques for Secure Application Coding in a Networked World*.
14) IBM (2017) *What you need to know about injection attacks*.
15) Kar, D. *et al.* (2016) ‘Detection of SQL injection attacks using hidden markov model’, *Proceedings of 2nd IEEE International Conference on Engineering and Technology, ICETECH 2016*, (March 2016), pp. 1–6. doi: 10.1109/ICETECH.2016.7569180.

16) Kar, D., Panigrahi, S. and Sundararajan, S. (2016) 'SQLiGoT: Detecting SQL injection attacks using graph of tokens and SVM', *Computers and Security*, 60(April 2016), pp. 206–225. doi: 10.1016/j.cose.2016.04.005.

17) Kruegel, C. and Vigna, G. (2010) 'Anomaly Detection of Web-Based Attacks', *Masterarbeit*, (November).

18) Livshits, V. B. and Lam, M. S. (2005) 'Finding Security Vulnerabilities in Java Applications with Static Analysis', *USENIX Security '05 (USENIX Security Symposium)*, p. 18. doi: 10.1.1.132.3096.

19) Maor, O. and Shulman, A. (2005) 'SQL Injection Signatures Evasion', *White Paper of Imperva Inc.*, (April). Available at: https://www.imperva.com/docs/IMPERVA_HII_SQL-Injection-Signatures-Evasion.pdf.

20) Oates, B. J. (2006) *Researching Information Systems and Computing*. Sage Publications Ltd.

21) Olah, C. (2015) *Understanding LSTM Networks*. Available at: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

22) Owasp (2013) *OWASP Top 10*

23) *- 2013*, *The Open Web Application Security Project*. Available at: http://owasptop10.googlecode.com/files/OWASP Top 10 - 2013.pdf.

24) Owasp (2017) *OWASP Top 10 -2017*. Available at: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

25) Pauwels, E. J. and Ambekar, O. (2013) 'One Class Classification for Anomaly Detection: Support Vector Data Description Revisited', 7987(August 2011). doi: 10.1007/978-3-642-39736-3.

26) Payam, R., Lei, T. and Huan, L. (2008) 'Cross-Validation'. Available at: http://leitang.net/papers/ency-cross-validation.pdf.

27) Radford, B. J., Richardson, B. D. and Davis, S. E. (2018) 'Sequence Aggregation Rules for Anomaly Detection in Computer Network Traffic'. doi: arXiv:1805.03735v1.

28) Rawat, R. and Shrivastav, K. S. (2012) 'SQL injection attack Detection using SVM', *International Journal of Computer Applications*, 42(13), pp. 1–4. doi: 10.5120/5749-7043.

29) Ray, D. and Ligatti, J. (2014) 'Defining Injection Attacks', (1), pp. 425–441. doi: 10.1007/978-3-319-13257-0_26.

30) Sheykhkanloo, N. M. (2015) 'A Pattern Recognition Neural Network Model for Detection and Classification of SQL Injection Attacks', *International Journal of Computer and Information Engineering*, 9(6), pp. 1380–1390. doi: 10.4018/IJCWT.2017040102.

31) Shin, Y., Williams, L. and Xie, T. (2014) ‘SQLUnitGen : SQL Injection Testing Using Static and Dynamic Analysis’, (December), pp. 0–1.

32) Skaruz, J. and Seredynski, F. (2007) ‘Recurrent neural networks towards detection of SQL attacks’, *Proceedings - 21st International Parallel and Distributed Processing Symposium, IPDPS 2007; Abstracts and CD-ROM*, (April 2007). doi: 10.1109/IPDPS.2007.370428.

33) Staudemeyer, R. C. (2015) ‘Applying long short-term memory recurrent neural networks to intrusion detection’, *South African Computer Journal*, 56(1), pp. 136–154. doi: 10.18489/sacj.v56i1.248.

34) Staudemeyer, R. C. and Omlin, C. W. (2014) ‘Extracting salient features for network intrusion detection using machine learning methods’, *South African Computer Journal*, 52(June), pp. 0–15. doi: 10.18489/sacj.v52i0.200.

35) Valeur, F., Mutz, D. and Vigna, G. (2005) ‘A learning-based approach to the detection of sql attacks’, *Intrusion and Malware Detection and Vulnerability Assessment*, pp. 123–140. doi: 10.1007/b137798.

Appendices

Appendix 1: Training code

```
from sklearn.model_selection import train_test_split, KFold
from sklearn.metrics import roc_curve, confusion_matrix, auc,
accuracy_score
from sklearn.preprocessing import LabelEncoder
from keras.models import Model
from keras.layers import LSTM, Activation, Dense, Dropout, Input, Embedding
from keras.optimizers import RMSprop
from keras.preprocessing.text import Tokenizer
from keras.preprocessing import sequence
from keras.callbacks import EarlyStopping
from sklearn.utils import shuffle
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt


class Trainer:

    max_words = 1000
    max_len = 150
    tokenizer = None
    X_train = None
    X_test = None
    Y_train = None
    Y_test = None

    def __init__(self, data):
        self.prepare_dataset(shuffle(data))

    def prepare_dataset(self, data):
        labels = list(data['Secure'])
        input_param = list(data['Content'])

        X = input_param
        Y = labels
        le = LabelEncoder()
        Y = le.fit_transform(Y)
        Y = Y.reshape(-1, 1)

        self.X_train, self.X_test, self.Y_train, self.Y_test =
train_test_split(X, Y, test_size=0.1)

        tok = Tokenizer(num_words=self.max_words)
        tok.fit_on_texts(self.X_train)

        self.tokenizer = tok

    def set_train_test(self, X_train, X_test, Y_train, Y_test):
        self.X_train = X_train
        self.X_test = X_test
        self.Y_test = Y_test
        self.Y_train = Y_train

    def run_training(self, no_blocks=None, hidden_layers=None):
        model = self.model(no_blocks, hidden_layers)
```

```
        if no_blocks is None:
            model.summary()

        model.compile(loss='binary_crossentropy', optimizer=RMSprop(),
metrics=['accuracy'])

        model.fit(self.tokenize(self.X_train), self.Y_train,
batch_size=128, epochs=10, verbose=1,
                  validation_split=0.1,
callbacks=[EarlyStopping(monitor='val_loss', min_delta=0.0001)])

        model.save('trained.h5')

        return model

    def train(self, no_blocks=None, hidden_layers=None):
        model = self.run_training(no_blocks, hidden_layers)
        if no_blocks is None:
            no_blocks = 64

        if hidden_layers is None:
            hidden_layers = 1

        desc = '{} blocks {} layers'.format(no_blocks, hidden_layers)

        self.results(model, desc)

    def train_multiple(self, blocks, hidden_layers):
        for block in blocks:
            for hidden_layer in hidden_layers:
                self.train(block, hidden_layer)

    def model(self, lstm_blocks=None, hidden_layers=None):
        if lstm_blocks is None:
            lstm_blocks = 64

        if hidden_layers is None:
            hidden_layers = 1

        print("Creating model with {} hidden layers, {} LSTM blocks
".format(hidden_layers, lstm_blocks))

        inputs = Input(name='inputs', shape=[self.max_len])
        layer = Embedding(self.max_words, 50,
input_length=self.max_len)(inputs)

        for i in range(0,hidden_layers - 1):
            layer = LSTM(lstm_blocks, return_sequences=True)(layer)

        layer = LSTM(lstm_blocks)(layer)
        layer = Dense(256, name='FC1')(layer)
        layer = Activation('sigmoid')(layer)
        layer = Dropout(0.5)(layer)
        layer = Dense(1, name='out_layer')(layer)
        layer = Activation('sigmoid')(layer)

        return Model(inputs=inputs, outputs=layer)

    def results(self, model, desc):
        prediction = self.predict(model, self.X_test)

        results = list(map(lambda i: round(i), prediction))
```

```
        ravelled = confusion_matrix(self.Y_test, results).ravel()

        # print("**** ravelled: "+ str(ravelled))
        tn = float(ravelled[0])
        fp = float(ravelled[1])
        fn = float(ravelled[2])
        tp = float(ravelled[3])


print('================================================================\n'
              'Confusion matrix \n'
              'TP: {} TN: {} FP: {} FN: {}'.format(int(tp), int(tn),
int(fp), int(fn)))

        accuracy = (tp + tn) / (tp + tn + fp + fn)
        recall = 0 if tp == 0 else tp / (tp + fn)
        precision =0 if tp == 0 else tp / (tp + fp)
        tnr = 0 if tn == 0 else tn / (tn + fp)
        fpr = 0 if fp == 0 else fp / (tn + fp)
        fpr_keras, tpr_keras, thresholds_keras = roc_curve(self.Y_test,
prediction.ravel())
        auc_keras = auc(fpr_keras, tpr_keras)


print('================================================================\n'
              'Test results \n'
              'Accuracy: {:0.3f} \n'
              'Recall: {:0.3f} \n'
              'Precision: {:0.3f} \n'
              'TNR: {:0.3f} \n'
              'FPR: {:0.3f} \n'
              'AUC: {:0.3f}'
              .format(accuracy, recall, precision, tnr, fpr, auc_keras))

        plt.figure(1)
        plt.plot([0, 1], [0, 1], 'k--')
        plt.plot(fpr_keras, tpr_keras, label=desc + ' (area =
{:.3f})'.format(auc_keras))
        # plt.plot(fpr_rf, tpr_rf, label='RF (area =
{:.3f})'.format(auc_rf))
        plt.xlabel('False positive rate')
        plt.ylabel('True positive rate')
        plt.title('ROC curve')
        plt.legend(loc='best')
        plt.savefig('roc.png')

    def accuracy(self, model, desc):
        prediction = self.predict(model, self.X_test)

        results = list(map(lambda i: round(i), prediction))

        accuracy = accuracy_score(self.Y_test, results)

        print("Model {} Accuracy {}".format(desc, accuracy))

    def predict(self, model, texts):
        sequences = self.tokenize(texts)
        sequences_matrix = sequence.pad_sequences(sequences,
maxlen=self.max_len)

        return model.predict(sequences_matrix)

    def tokenize(self, texts):
```

```
        sequences = self.tokenizer.texts_to_sequences(texts)

        return sequence.pad_sequences(sequences, maxlen=self.max_len)
```

Appendix 2: Detection Middleware Code

```
from flask import Flask, Response, Request, request
from requests import get, post
from keras.models import load_model
from model.Trainer import Trainer
import pandas

app = Flask('__main__')
SITE_NAME = 'http://dvwa.local'
DETECTION = True

TRAINER = Trainer(pandas.read_csv('dataset.csv'))
MODEL = load_model('saved_model.h5')
TRAINER.predict(MODEL, 'test')


@app.route('/', defaults={'path': ''})
@app.route('/<path:path>')
def proxy(path):
    queries = str(request.query_string).split('&')

    if predict(queries) is False and DETECTION is True:
        resp = Response("Attack detected!! \n Input:
"+str(request.query_string))
        return resp

    got = get(gen_url(path))
    headers = got.headers
    resp = Response(got.content)
    resp.headers['Content-type'] = headers['Content-type']
    return resp


@app.route('/', defaults={'path': ''}, methods=['POST'])
@app.route('/<path:path>', methods=['POST'])
def post_proxy(path):
    data = request.data
    posted = post(gen_url(path), data)

    resp = Response(posted.content)
    resp.headers['Content-type'] = posted.headers['Content-type']

    return resp


def gen_url(path):
    return SITE_NAME + '/'+path+'?'+request.query_string


def log(text, verdict):
    df = pandas.DataFrame({"Content" : [text], "Secure": [verdict],
"Source" : "Prototype"})

    with open('detections.csv', 'a') as f:
        df.to_csv(f, header=False)
```

```
def predict(queries):
    for query in queries:
        text = query.split('=')

        if len(text) < 2:
            continue

        text = text[1]

        if text == '':
            continue

        res = TRAINER.predict(MODEL, [text])

        truthy = bool(round(res[0]))

        print('Input: {} Detection: {}'.format(text, 'Attack' if truthy is
False else 'Safe'))

        log(text, truthy)

        return truthy

    return True


app.run(host='0.0.0.0', port=8090)
```