Detecting SQL Injection Attacks using Text Mining and Bro IDS

Yue Guan
Graduate Student
Information Security Institute
yguan11@jhu.edu

Vedasagar Karthykeyan
Graduate Student
Information Security Institute
vkarthy1@jhu.edu

Qiqing Huang
Graduate Student
Information Security Institute
qhuang24@jhu.edu

***ABSTRACT*--THE PURPOSE OF THE PAPER IS TO PRESENT A NOVEL SQL INTRUSION DETECTION MECHANISM USING TEXT MINING AND BRO IDS. IN ORDER TO DETECT KNOWN WEB-BASED ATTACKS, INTRUSION DETECTION SYSTEMS ARE USUALLY EQUIPPED WITH A LARGE NUMBER OF SIGNATURES. BUT BY OBFUSCATING THE INPUT AN ATTACKER CAN SUCCESSFULLY FOOL THE IDS AND GAIN ACCESS TO THE BACKEND DATABASE. THE PAPER WILL EXPLORE THE USE OF TERM FREQUENCY-INVERSE DOCUMENT FREQUENCY TO EXTRACT THE MOST DISCRIMINATIVE SQL INJECTION FEATURES AND USE THESE FEATURES TO DETECT SQL INJECTION ATTACKS USING BRO IDS SCRIPTING LANGUAGE.**

Index Terms—SQL Injection, BRO IDS, Text Mining, TF-IDF

I. INTRODUCTION

Injection is at the top of the TOP 10 vulnerabilities published by OWASP. It can be of various types like SQL Injection, LDAP Injection etc. But we will concentrate on SQL Injection as this is the most common kind of injection, which can give sensitive information about the database. When our web application passes any SQL query to the back-end database to do some operation, if the user input is not properly sanitized and checked, attackers can construct some malicious SQL queries, which can give them access to the database. They can extract sensitive information about the data stored and also what type of table and attributes are present. Getting hold of such information can be very harmful for any organization. So it is highly important that we detect such attacks and prevent them from obtaining any data.

A. What is SQL Injection

SQL injection is an attack where an attacker can give malicious SQL queries in the web input form or into the page URL with the final aim being to cheat the server to execute the malicious SQL command.

If we enter the URL www.jhu.edu into the browser, this URL address is only a simple request to the page. There is no dynamic request to the database, so this URL is not susceptible to SQL Injection. But while we can enter the URL www.jhu.edu/academic?id=1, when we transport the parameter "id" using the URL and at the same time we provide the value of "id", which is 1 in this case. Because this kind of URL is asking for a dynamic request to the database so if the security measure of enforced while developing the website is not so good, we can insert the malicious SQL command in this URL.

According to the principle, SQL injection can be divided into two kinds, one is injecting at the platform level, and another is injecting at the code level. The former one is possible because of insecure database configuration or the vulnerability of the database platform. The latter is mainly because the programmer didn't do the required filtering and sanitization of the input, then let the malicious SQL command injected.

II. RELATED WORK

A lot of researchers have already proposed numerous ideas or trial to detect SQL injection. The paper [8] gives a good overview of SQL Injection and walks through the anatomy of an attack and talks about some advanced SQL Injection techniques using sprocs, that is stored and extended procedures that are pre installed in MS-SQL 2000.

AMNESIA [5] uses a model-based approach by defining a set of hotspots, that are points in the application code that send SQL queries to the database. They then come up with models of SQL queries for each hotspot to detect illegal queries before their execution into the database. They first perform a static analysis where the web application's code is analyzed to automatically build a model for the legitimate queries that are to be expected. One of the main drawbacks of the proposed solution is that it requires changes to be done to the web application's source code in order to successfully send data to the runtime monitoring model. Our implementation is better as we do not need to make changes to website and we just add a layer of IDS for all the traffic passing to the web app.

And there is another way to detect the SQL injection by using the query tokenization. As we all know, there are lots of chances that a user may inject in the query then result in a

different database request when using dynamic queries. In 2010, Ntagwabira Lambert and Kang Song Lin [10] develop a method that detects SQL injection attacks by checking whether user inputs cause changes in the query's intended result. They separate tokenizing original query and a query with injection, the tokenization is performed by detecting a space, single quote or double dashes and all strings before each symbol constitute a token.

SQL-IDS [7] uses a lexical analysis of the SQL queries to categorize them into normal and malicious queries. Each SQL statement is divided into syntactic units based on each character. This arbitrary input passes through a lexical analysis process in which the characters are grouped into tokens. The next step is to check the syntactical correctness of the tokens produced from each SQL statement produced in the previous phase. An SQL statement is considered to be valid if it does not violate the syntactical rules that exist in the corresponding specification. The main drawback of this implementation is that the model sticks to the defined syntactical rules and compares the new queries with this. This can lead to a large number of false positives as the model only fits a particular specification. Any deviation from such kind of specification will lead to performance degradation. Our implementation also improves on this idea of parsing the SQL Injection commands and extracting the most important characters and words from them. And we do not have defined syntactical rules, but we define the application level semantics that can be expected from the SQL commands.

In SQLiDDS [6] the authors propose a SQL detection mechanism by first converting the SQL statements into sentence like forms by classifying every character into a word. Then a model is trained by applying Term Frequency-Inverse Document Frequency (TF-IDF) on the converted sentences to extract the most discriminative words and characters that occur in a malicious SQL query. Through static and dynamic analysis using document similarity measures they assert that it is sufficient to examine the part of an SQL query after the WHERE keyword in order to detect the presence of any injection attack, which significantly narrows down the scope of processing. This kind of technique involves some machine learning and text mining on the data for extracting the most important characters. We make use of this approach and apply the same document similarity measure, but to extract the characters with the highest document frequency and use them to write the Bro scripts.

In [10] the authors talk about using BRO IDS to record the network traffic and reconstruct the SQL injection attacks using network forensics. This talks about how an IDS can be put in place rather than making changes to the website's source code as proposed in AMNESIA and SQLGuard. This approach is very useful as the Network IDS can be installed at the ingress point of traffic and instead of making changes to the application itself, we can just define the application layer semantics into the IDS.

In [12] the authors give examples of BRO scripts that can be used to define application layer semantics and detect XSS and SQL injection attacks. These are basic example scripts that be used as a starting point for our implementation.

III. WHAT IS BRO IDS

Bro [1] is an open-source, Unix-based Network Intrusion Detection System (NIDS), as we can see from the figure 1, the idea of it is using the libpcap to get the data packets from the network, and then let these data packets go through the event generation engine and rule engine to be abstracted and reorganized to a series of events. These events will be further analyzed by the strategy scripts, the basic event itself can raise an alarm, and after the analysis of strategy analysis scripts, it also can raise an alarm.

Bro IDS takes in a pcap file and also the Bro script and adds any alerts to a notice.log file and HTTP requests it sees to http.log file. We detect 2 kinds of attacks using Bro IDS, one when the SQL Injection command is present in the Uniform Resource Identifier of the HTTP GET request and second, an SQL Injection command sent as an HTTP POST request in a HTML form on the web page.

Bro IDS detects both these kinds of attacks and adds the attack query to the notice.log file. it also mentions the attacker and victim IP addresses and ports. The notice.log file can be parsed to calculate the number of alerts generated. This can be compared to the http.log file, which consists of all the requests made.

The hit rate and the false alarm rate is calculated as:

$$\frac{Number\ of\ alerts\ generated}{Total\ number\ of\ requests} = \frac{\#\ of\ alerts\ in\ notice.log}{\#\ of\ alerts\ in\ http.log}$$

This calculation is done for both the malicious traffic and the normal traffic. The hit rate is calculated in case of the malicious pcap file and false alarm rate with the normal pcap file.

IV. OUR PROPOSED APPROACH

Two key ideas, using document similarity measure like term frequency-inverse document frequency to extract the features and defining application level semantics using Bro IDS constitute the core of our approach.

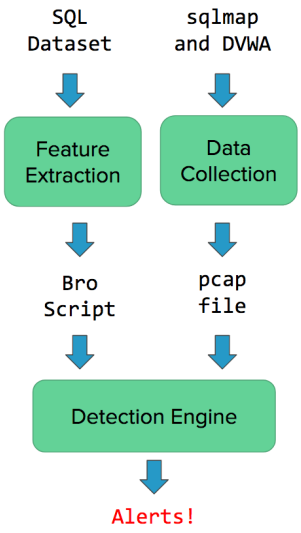

Figure 1. Our Approach

The first step in our approach was to collect data by using sqlmap which is an automated SQL Injection tool and running this on Damn Vulnerable Web Application and collecting the traffic as pcap files. The next step was to extract the most useful features by using the SQL Injection command dataset we found online. We ran TF-IDF on this dataset to extract the most important characters that occur in a malicious SQL Injection command.

The final step was to write Bro scripts using the features extracted using TF-IDF and using this script along with the pcap file generated and pass these to the Bro detection engine. The detection engine generates alerts whenever it sees the malicious SQL features in the pcap file.

A. Data Collection

HTTP defines different types of methods to interact with the server, the main two types are GET and POST. And we collect these two types of data.

1. **SQL Injection in the HTTP URI - GET requests**

GET request will include the parameter in the URI. Like the link http://www.my.jhu.edu?id=1, "id" is the parameter, which is according to the database behind the website, and "1" means the value of this parameter is 1. So when the browser get this link, it will send the request to the web server, and web server will search the parameter which is "id" and the value of this parameter is "1" in its database and return the data.

If the web server doesn't filter well to the input URI, then the attacker can include the malicious sql language after the parameter. Like we can try to append "-1 union select 1,2,database(),4,5" after the parameter. In this way the URI will be "http://www.my.jhu.edu?id=-1 union select 1,2,database(),4,5", it will get the current database name and user who connected to the database if there is no good filter.

The database behind the website can be various, it can be access, mysql or mssql, so the sql language should be modified according to the different grammar and different database structure of the databases.

Figure 2. SQL Injection in HTTP URI

As you can see from the screenshot, we are trying to do SQL injection in the HTTP GET request with the Submit Query.

2. SQL Injection in the HTTP Body - POST requests

POST request will include the parameter in the request body. The attacker may include the malicious SQL language in the search box or the submit form. If the input to this is not escaped and properly sanitized, then the attacker can gain access to the database. For example, in Figure 3 below, the attacker passes the query "admin%' AND 9383=9383 AND '%'=' ". This will return TRUE for 9383=9383 and will give the attacker access even though the password is wrong. So, the developers need to make sure that the input is checked before passing to the backend. Such malicious requests must be ignored altogether.

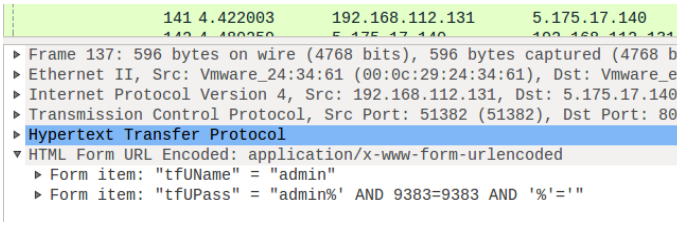

Figure 3. SQL Injection in HTTP Post

Like in the above picture, the tfUPass is a input box for client to enter the password, the attacker has included the malicious sql language in the form. If the input is not properly sanitized and validated, then the attacker can get admin access to the database.

2) Generate SQL injection

We use SQLMAP [11] to generate the sql injection. SQLMAP is a powerful open source automatic sql injection tool. It can be used to detect and exploit the sql injection vulnerability. It already includes numerous sql injection payload in it.

We ran SQLMAP on Damn Vulnerable Web Application (DVWA) [3] for doing SQL Injection as both a GET request

and also POST request. We can also specify the level and risk of SQL injection attacks to be performed. For our current project, we set the level of SQL injection to 5 and the risk level to 1.

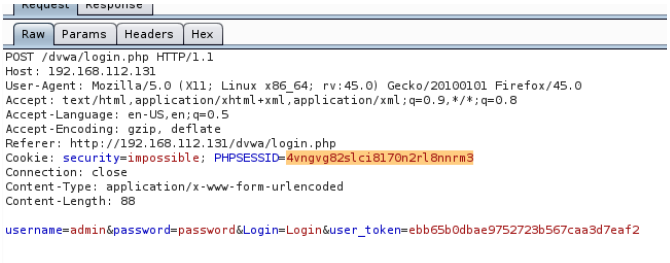

Figure 4. Extracting the PHPSESSID

We had to first extract the cookie from the Damn Vulnerable Web Application to use with SQLMAP. We used Burpsuite to intercept the communication and extracted the PHPSESSID from that. We need this cookie to pass as a parameter to the HTTP GET Request to successfully launch SQL Injection attacks.

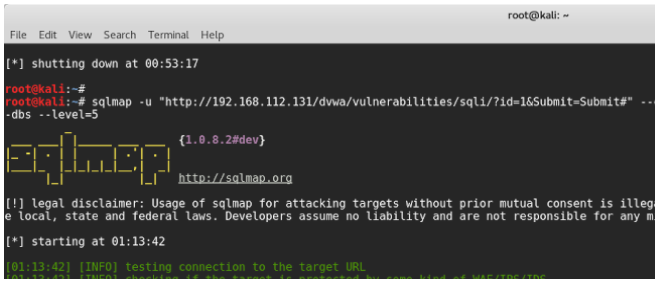

Figure 5. SQLMAP on DVWA

3) Data Collection

There are many ways to collect the data packets of the network traffic. Like using Wireshark or TCPDUMP. Wireshark is a software that able to capture the network packets of the certain network card of the computer. The data captured will be saved as .pcap format. Wireshark also can be used to look over the details of the file it captured. TCPDUMP is a widely used command-line to sniff packets. It is available in most of the Linux based operating systems.

We can use this command "tcpdump -i <interface> -s 65535 -w <some-file>" to capture the certain data packets going through the certain interface. So while running SQLMAP on DVWA, we also ran TCPDUMP and collected all the network traffic going through the interface. This resulted in a pcap file with the SQL injection in the GET and POST parameters.

4) Normal Traffic

We also generated Normal HTTP traffic by making many HTTP GET and POST requests and running TCPDUMP. This resulted in a pcap file with valid HTTP traffic. We saw that many of the requests consisted of characters found in SQL Injection attacks. This means that our model with have false positives if we use fewer characters for detecting SQL injection attacks. The more characters we use, the lesser false alarm rate we should get. So, to extract what characters should be used to detect the attacks, we used a document similarity measure known as Term Frequency-Inverse Document Frequency (TF-IDF).

B. Term Frequency-Inverse Document Frequency

Term Frequency-Inverse Document Frequency is a numerical statistic which shows how important a word is to a document in a corpus. It gives us a numeric output for every word and the importance of that word in that corpus.

The formula for calculating it is given by:

$$tfidf(t,d,D) = tf(t,d) \cdot idf(t,D)$$

where, t is the term, d is one document and D is the corpus of all documents.

- tf(t, d) is the term frequency of term t in document d.
- idf(t, D) is the inverse document frequency of term t in the corpus.
- idf(t, D) is given by:

$$idf(t,D) = log \frac{N}{|\{d \in D : t \in d\}|}$$

where N is the total number of documents in the corpus, and the denominator denotes if a particular term from the corpus is present in the document.

For example, take the words "Computer Intrusion Detection" and suppose we have a corpus that consists of 100 documents which are all related to computers. We need to find out the importance of the three words in that corpus. Since the word "Computer" will be very common in all the documents, the term frequency will be high. But we are more interested in the documents that relate to "Intrusion Detection". The Inverse Document Frequency for "computer" will be low since the denominator in idf will be high. So the total tf-idf for "Computer" will be low and on the other hand, would be high for the terms "Intrusion" and "Detection".

We used this approach to extract the most discriminative and important characters from SQL Injection command dataset [13]. We used the python package scikit-learn and used the TfidfVectorizer which is a feature extraction module for text mining. We used the raw SQL injection commands as the training dataset. We pass different parameters to the Vectorizer based on our needs. For example, ngram_range is set to (1,1) which means that we are only interested in extracting single characters from the dataset instead of n-grams. The analyzer is set to word to analyze hex characters. Token_pattern is set to '.' which means that single characters will be treated as words and the frequency will ne calculated. min_df denotes the minimum document frequency the character must have. It is set to '0.05' based on empirical results.

As you can see we get the most important characters from the dataset, these are: '"', '*', '"', '#', '(', ')', ',', '-', ':', ';', '>', '<', '=', '_'. We also extracted the most frequent words occurring in a malicious SQL Injection Command like 'SELECT', 'DROP' 'DELETE' and also the numbers from [0-

9]. These characters denote the most important and frequent characters that appear in an SQL Injection command. More number of these character would mean that the query is more malicious and should be detected.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.cross_validation import train_test_split

X = dataframe.raw_sql
y = dataframe["type"]
vec_opts = {
    "ngram_range": (1, 1),  # n-grams
    "analyzer": "word",     # analyze hex words
    "token_pattern": ".",   # number of characters to treat as a word
    "min_df" : 0.05         # ignore terms that have a document frequency lowe
}
v = CountVectorizer(**vec_opts)
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np

X = v.fit_transform(X)
features = v.get_feature_names()
features
```

```
[u' ',
 u'"',
 u'#',
 u"'",
 u'(',
 u')',
 u',',
 u'-',
 u'.',
 u'0',
 u'1',
 u'2',
 u'3',
 u'4',
 u'5',
 u'6',
 u'7',
 u'8',
 u'9',
 u':',
```

Figure 6. Tf-Idf on the SQL dataset

C. Bro Scripting

After extracting the most important characters, the next step was to define application layer semantics to detect any attacks with the help of Bro IDS. The features extracted were used to write the Bro Scripts. The Bro script looks for the characters in the HHTP URI and HTTP Post request. Every time it sees the malicious

We used a scoring mechanism depending on the number of different malicious characters or features present in the SQL command. The more malicious SQL characters a command consists of, the higher its score would be, and it can be deemed more malicious.

V. ANALYSIS

In the analysis part, we apply the concepts of hit rate and false alarm rate to create an ROC curve.

The hit rate, is the probability of detection(PD) or, also called as detection (hit) rate indicates the probability of correctly identifying an attack. Here is the formula: ***Hit Rate = True Positive Rate = True Positive / (True Positive + False Negative).*** In our project, the true positive is the situation that detecting the real attack correctly. False negative is the situation that not detecting the attack and letting it pass. For example, there is SQL injection attack, however, you suggest it is not an SQL injection attack by IDS, it is a False Negative.

When it comes to false alarm rate, it is the probability of false alarm (PFA) or false alarm rate the probability of alerting on a non-attack. Here is the formula for PFA: ***False Positive Rate = False Positive / (False Positive +True Negative).*** In our project, the situation of false positive is, although there is no SQL injection attack, the IDS suggests that an attack is happening. True Negative, apparently speaking, means indicates a non-attack or normal traffic correctly.

Bro IDS detects both these kinds of attacks and adds the attack query to the notice.log file. it also mentions the attacker and victim IP addresses and ports. The notice.log file can be parsed to calculate the number of alerts generated. This can be compared to the http.log file, which consists of all the requests made.

The hit rate and the false alarm rate is calculated as:

$$\frac{Number\ of\ alerts\ generated}{Total\ number\ of\ requests} = \frac{\#\ of\ alerts\ in\ notice.log}{\#\ of\ alerts\ in\ http.log}$$

This calculation is done for both the malicious traffic and the normal traffic. The hit rate is calculated in case of the malicious pcap file and false alarm rate with the normal pcap file.

In order to demonstrating the concepts of hit rate and false alarm rate in a more concrete way, we create a confusion matrix. It is directly showing the portion of phenomenon combining notion of hit rate and false alarm rate.

Predict / Real	Related (1)	Not related (0)
Been detected (1)	**True Positive**	**False Positive**
Not been detected (0)	**False Negative**	**True Negative**

In this project, we changed the detection threshold from 2-9 and also reached a highest threshold of 15 to calculate the hit rate and the false alarm rate respectively. An attack is detected only when the score in the Bro script crosses this threshold. Here is the result from our dataset:

Thresholds	Hit Rate	False Alarm Rate
2	0.992982	0.857143
3	0.961404	0.369048
4	0.901754	0.250000
5	0.873684	0.226190
6	0.771930	0.023810
7	0.673684	0.023810
8	0.673684	0.023810
9	0.673684	0.00
15	0.00	0.00

The Bro engine only raises an alert when the score crosses this threshold. Because of the characters of our dataset, we can easily compute the rates from the result of our IDS. From the result, it is obviously that having a low threshold resulted in a high hit rate and a high false positive rate. As we increased the threshold, the hit rate decreases gradually, but the false positive rate decreases abruptly. So, this means that for higher values of threshold we get very little false alarms but the hit rate remains relatively high. For a high threshold of 15, we get both hit rate and false alarm rate as 0.

Based on the result, we would like to introduce a concept, ROC curve. In signal detection theory, a receiver operating characteristic, or simply ROC curve, is a graphical plot of the sensitivity for a binary classifier system as its discrimination threshold is varied. It plots each value of the parameter, for example, in this project, we applied the PD vs. PFA under different thresholds, to analyze the performance of the IDS system . We draw the ROC curve to evaluate the efficiency of our SQL injection IDS system. The ROC curve is the following:

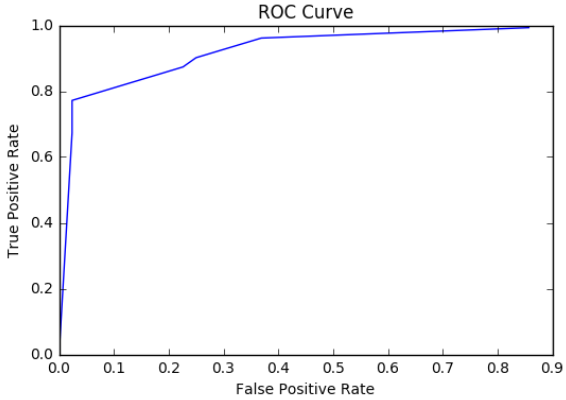

Figure 7. ROC Curve

With the analysis of ROC curve, we evaluate the true positive rate of the detection system. The top-left in the picture represents the true positive is almost 1, which means the detecting correctness. The ROC curve characteristic is that if the line is closer to the top-left, it means the system detecting actions are more accurate. From the picture, it shows that our curve is relatively close to the top-left. We conclude that the IDS system we established is efficient.

VI. Conclusion

SQL Injection prevention had been one of the most important measures that any developer needs to take care of. These attacks are highly popular due to their ease of implementation and high impact. A successful SQL injection can give an attacker access to highly sensitive data. So, we need to come up with smart techniques that can quickly identify if an attack is taking place.

Our proposed implementation makes use of the fact that certain characters and words are more frequent in a malicious SQL query and by extracting these we can define a statistical model to train on. TF-IDF is a very useful metric for extracting the most important characters from SQL Injection dataset. The detection will be performed online using Bro IDS which will raise an alert if it encounters any of the malicious tokens. By analysis the hit rate and false alarm rate of the detection system, combining the concept of ROC curve, we can conclude that our model's efficiency and correctness.

A. Future Work

For the future work, we are looking forward to constructing a standard threat model and making a statistical result for optimizing the intrusion detection technique primarily focusing on the SQL Injection attack. For example, there are some other advanced machine learning techniques, cross-validation, n-grams that could fit our model well. In an advanced perspective, we can select sequences or n-grams of words found in SQL injection attack instead of keywords in our detection mode and testing the accuracy using cross-validation method. It may achieve results better than previously implemented mechanisms.

We are only detecting plain SQL injection attacks. We can extend this idea to detect more advanced types of SQL injection attacks like Blind SQL Injection attacks or timing based SQL Injection attacks. Blind SQL Injection attacks ask the database True or False questions and gathers information based on the reply it gets from the database. Timing based SQL Injection attacks measure the time it takes for the database to reply to query. Longer times indicate that the query was successfully executed which means that it was correct and incorrect queries will correspond to shorter reply times. We can look for new and novel ways to detect such kinds of attacks based on the queries executed.

VII. References

[1] "The Bro Network Security Monitor." *The Bro Network Security Monitor*. Web. 02 Apr. 2017.

[2] Buehrer, Gregory T., Bruce W. Weide, and Paolo A. G. Sivilotti. "Using Parse Tree Validation to Prevent SQL Injection Attacks." *Proceedings of the 5th International Workshop on Software Engineering and Middleware - SEM '05* (2005). Print.

[3] *DVWA - Damn Vulnerable Web Application*. Web. 14 May 2017.

[4] Gupta, Mukesh Kumar, M.c. Govil, and Girdhari Singh. "Static Analysis Approaches to Detect SQL Injection and Cross Site Scripting Vulnerabilities in Web Applications: A Survey." *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)* (2014). Print.

[5] Halfond, William G. J., and Alessandro Orso. "Amnesia." *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering - ASE '05* (2005). Print.

[6] Kar, Debabrata, Suvasini Panigrahi, and Srikanth Sundararajan. "SQLiDDS: SQL Injection Detection Using Document Similarity Measure." *Journal of Computer Security* 24.4 (2016): 507-39. Print.

[7] Kemalis, Konstantinos, and Theodores Tzouramanis. "Sql-Ids." *Proceedings of the 2008 ACM Symposium on Applied Computing - SAC '08* (2008). Print.

[8] McDonald, Stuart. "SQL Injection: Modes of attack, defense, and why it matters." White paper, Government Security. org (2002).

[9] Pomeroy, Allen, and Qing Tan. "Effective SQL Injection Attack Reconstruction Using Network Recording." *2011 IEEE 11th International Conference on Computer and Information Technology* (2011). Print.

[10] Ntagwabira, Lambert, and Song Lin Kang. "Use of Query Tokenization to Detect and Prevent SQL Injection Attacks." *2010 3rd International Conference on Computer Science and Information Technology* (2010). Print.

[11] "Sqlmap®." *Sqlmap: Automatic SQL Injection and Database Takeover Tool*. Web. 02 Apr. 2017

[12] Varadarajan, G. K., and M. Santander Peláez. *Web application attack analysis using Bro IDS*. Technical report, SANS Institute. InfoSec Reading Room. http://www. sans. org/reading-room/whitepapers/detection/web-application-attack-analysis-bro-ids-34042 (last accessed August 2015). 2.2. 1.3, 2012.

[13] ClickSecurity. "ClickSecurity/data_hacking." *GitHub*. 19 Jan. 2014. Web. 01 May 2017. https://github.com/ClickSecurity/data_hacking/tree/master/sql_injection/data