**Detecting SQL Injection Vulnerabilities in Web Services**

Chinni Krishna Kowsik. P

UG Student,

Department of Computer Science and

Engineering (Cyber Security),

School of Computing,

Amrita Vishwa Vidyapeetham,

Chennai

[kowsikperumalla@gmail.com](mailto:kowsikperumalla@gmail.com)

Uday Reddy. H

UG Student,

Department of Computer Science and

Engineering (Cyber Security),

School of Computing,

Amrita Vishwa Vidyapeetham,

Chennai

[hosurudayreddy@gmail.com](mailto:hosurudayreddy@gmail.com)

Saranya. G

Associate Professor,

Department of Computer Science

and Engineering (Cyber Security),

School of Computing

Amrita Vishwa Vidyapeetham, Chennai

[s\_saranya@ch.amrita.edu](mailto:s_saranya@ch.amrita.edu)

1. **ABSTRACT:**

In the world of web security, SQL injection is a notorious threat where hackers can sneak into databases by manipulating website input fields. This code acts as a vigilant guard, defending against such attacks in multiple ways. Firstly, it learns from a collection of safe and malicious SQL queries, becoming smart enough to recognize patterns in the queries users input. Think of it as learning from past mistakes to spot potential dangers. Secondly, the code plays a game of pretend. It imagines various tricky scenarios where hackers modify innocent-looking queries into something harmful. By doing this, it tests the system's defenses, ensuring it can handle unexpected twists and turns.

Lastly, it keeps an eye out for known tricks. Imagine having a list of cheat codes that hackers often use; the code checks user input against these codes, quickly identifying if someone's trying to play dirty. By combining these techniques, the code creates a powerful shield around web applications. It's like having a smart bouncer at the door, trained to spot troublemakers and keep them out. This way, web applications can focus on serving users securely, without worrying about sneaky hackers trying to break in.

This abstract breaks down the essence of the code's purpose in a more human-readable manner. The system evaluates these cases using the trained classifier, identifying potential vulnerabilities. Additionally, the code utilizes pattern matching techniques, employing predefined regular expressions to recognize known SQL injection patterns within user input. By combining machine learning, fuzz testing, and pattern matching, the system offers a robust defense mechanism against SQL injection attacks, enhancing the security posture of web applications.

**Keywords:**

Such, as “SQL Injection Prevention”, “Intrusion Detection System (IDS)”, “OWASP Top Ten” and “SQL Injection Scanning”, “SQL Injection”, “SQL Injection Detection”, “Intrusion Detection”, “Web Application Security”, “Vulnerability Assessment”, “Attack Detection”, “Signature-based Detection”, “Anomaly-based Detection”, “Pattern Matching”.

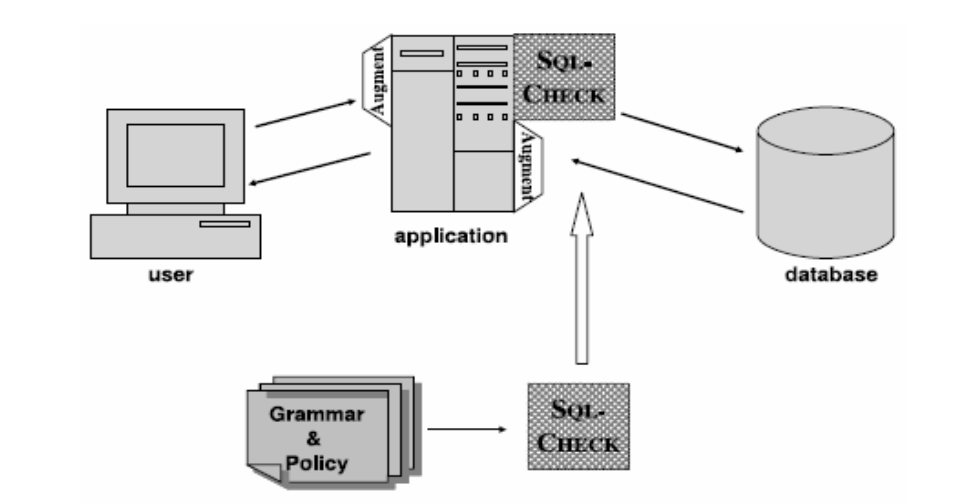
1. **INTRODUCTION:**

In the ever-evolving landscape of online security, safeguarding sensitive data from malicious attacks is paramount. One common and menacing threat faced by web applications is SQL injection, a technique employed by hackers to exploit vulnerabilities in database interactions. Imagine a scenario where a seemingly harmless user input field becomes a gateway for cybercriminals to manipulate databases and steal valuable information.

Web services are nowadays a strategic mean for data exchange and systems integration as they provide a simple interface between a provider and a consumer. Calls between consumers and providers consist of messages that follow the SOAP protocol, which, along with WSDL and UDDI, form the core of the web services technology.

SQL Injection is a type of injection or attack in a Web application, in which the attacker provides Structured Query Language (SQL) code to a user input box of a Web form to gain unauthorized and unlimited access. The attacker’s input is transmitted into an SQL query in such a way that it will form an SQL code. It is categorized as one of the top-10 2010 Web application vulnerabilities experienced by Web applications according to OWASP (Open Web Application Security Project) [1].

Additionally, the differences in the vulnerabilities detected and the high number of false-positives (35% and 40% in two cases) and low coverage (less than 20% for two of the scanners) observed highlight the limitations of web vulnerability scanners on detecting security vulnerabilities in web services. As mentioned before, SQL Injection is one of the most common and most critical types of vulnerabilities in database-centric web environments.



Web vulnerability scanners are well-known tools that allow testing applications against security issues. These tools provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. Most of these scanners are commercial tools (e.g., Acunetix Web Vulnerability Scanner, IBM Rational AppScan, and HP WebInspect), [7] but there are also some free application scanners (e.g., Foundstone WSDigger and wsfuzzer) with limited use, as they lack.

As mentioned before, SQL Injection is one of the most common and most critical types of vulnerabilities in database-centric web environments. In this paper we propose an approach to detect SQL Injection vulnerabilities in web services code. The proposed approach consists of a set of tests based on malicious parameters (i.e., attacks) that is applied to disclose code vulnerabilities. Web services responses are analyzed based on rules that help confirming the existence of vulnerabilities and eliminating false positives. [10]

Comparing to existing web vulnerability scanners, our approach has three key improvements:

1. We use a representative workload to exercise the services and understand the expected behaviour (i.e., the typical responses in the presence of valid inputs).
2. The set of attacks performed is a compilation of all the attacks performed by a large set of scanners plus many attack methods that can be found in the literature.
3. We apply well defined rules to analyse the web services responses in order to improve coverage and remove false positives. These rules include comparing the responses obtained when using malicious inputs with the normal responses (i.e., responses in the presence of a valid workload) and with the responses from robustness testing.
4. **LITERATURE REVIEW:**

Anamika Joshi and Geetha V et. al [6] proposed to SQL Injection Detection using Machine Learning (2014). The authors used a dataset consisting of 63 normal code queries and 51 malicious code queries for training, and 38 normal code queries and 26 malicious code queries for testing. The paper proposes a method for detecting SQL injection attacks using the Naïve Bayes machine learning algorithm combined with Role-Based Access Control (RBAC) mechanism. The paper addresses the SQL injection vulnerability, which is a common attack vector in web applications, where attackers manipulate user input to inject malicious SQL queries and potentially gain unauthorized access to a database. It combines the prior probability, likelihood, and RBAC information to make this classification. The authors evaluated their approach using a dataset and achieved an accuracy of 93.3%. This method can be used to enhance the security of web applications by detecting SQL injection attacks in real-time, preventing potential data breaches and unauthorized access to sensitive information. It offers a promising approach to bolster web application security and reduce false positive rates in SQL injection detection.

Kevin Ross et. al [7] proposed to SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources 2018. The dataset used in this paper is collected from two sources: incoming HTTP traffic between the traffic generation server and the web application server and the resulting MySQL traffic between the web application server and the remote database server. A correlated dataset is also created by combining features from both datasets. The paper employs machine learning techniques, including Decision Tree, Rule-based, Support Vector Machine (SVM), Neural Network, and Random Forest algorithms, to classify incoming traffic as normal or malicious. Feature selection techniques such as Correlated Feature Set (CFS) are used to reduce dimensionality. The primary vulnerability addressed in this paper is SQL injection, which is a serious security threat in web applications. SQL injection occurs when an attacker manipulates input data to execute unauthorized SQL queries on a database, potentially leading to data breaches and other security issues.

Maha Alghawazi, Daniyal Alghazzawi, Suaad Alarifi et. al [8] proposed to Detection of SQL Injection Attack Using Machine Learning Techniques: A Systematic Literature Review 2022. The paper discusses various datasets used in different studies for training and testing machine learning and deep learning models for SQL injection attack detection. These datasets include both real-world and synthetic datasets, such as those generated using mutation operators or collected from web applications. The paper reviews and summarizes the machine learning and deep learning techniques employed by researchers to detect SQL injection attacks. These techniques encompass a wide range of algorithms, including decision trees, support vector machines (SVM), neural networks (e.g., LSTM, CNN, MLP), ensemble methods (e.g., AdaBoost), and more. SQL injection attacks are a prevalent and severe vulnerability in web applications. They occur when attackers manipulate input data to execute malicious SQL statements, potentially gaining unauthorized access to databases and compromising data integrity and confidentiality.

Atefeh Tajpour, Mohammad JorJor zade Shooshtari et. al [9] proposed to Evaluation of SQL Injection Detection and Prevention Techniques in 2010. This paper from 2010 explores a range of techniques designed to detect and prevent SQL Injection Attacks (SQLIAs) in web applications. SQL Injection Attacks are a significant threat to the security of web applications, allowing attackers to exploit vulnerabilities in the application's input handling to gain unauthorized access to databases. The paper discusses various methods employed by attackers to find vulnerable parameters within applications, such as Union Query, Piggy-backed Queries, Stored Procedures, Inference, and Alternate Encodings. The paper then presents a comprehensive review of different approaches to detecting and preventing SQLIAs, highlighting both their strengths and limitations. These techniques encompass black-box testing, static analysis, runtime monitoring, and even modification of programming languages and APIs to promote secure coding practices. Some notable approaches mentioned include WAVES, JDBC-Checker, Taoutology Checker, SAFELI, CANDID, SQL Guard, AMNESIA, WebSSARI, Java Static Tainting, and more.

MeiJunjin et. al [10] proposed on “An approach for SQL injection vulnerability detection” in 2009. The paper proposes an approach for SQL injection vulnerability detection by combining static analysis, runtime detection, and automatic testing. They introduce a prototype tool called SQLInjectionGen and perform case studies on two small web applications to evaluate its effectiveness compared to static analysis using FindBugs. The approach involves generating hotspot-reaching test cases and attack test cases using a SQL query model and string argument instrumentation. The results show that SQLInjectionGen had no false positives and a small number of false negatives, making it effective in identifying SQL injection vulnerabilities, especially in legacy applications. The approach provides a way to detect vulnerabilities early and accurately, aiding in improving software security.

1. **PROBLEM STATEMENT**

The robustness and adaptability of web services face unprecedented challenges in the ever-changing landscape of cyber threats, particularly from the escalating sophistication of SQL injection attacks. These attacks, distinguished by their polymorphic nature and dynamic evasion strategies, call into question traditional paradigms of vulnerability detection and mitigation. In order to address this, there is an urgent need to conceptualize, develop, and rigorously test an advanced cognitive framework that not only proactively identifies SQL injection vulnerabilities in real-time within the intricate fabric of web services, but also autonomously adapts to emerging attack vectors, polymorphic techniques, and evasion strategies used by malicious actors.

1. **METHODOLOGY:**

In the realm of securing web applications against SQL injection threats, a meticulous and proactive methodology has been devised. This approach combines intelligence, creativity, and vigilance to fortify digital systems against potential attacks.

**1.** **Data-driven Learning:**

The methodology kicks off with a deep dive into data-driven learning. A diverse dataset of both harmless and malicious SQL queries acts as the foundation. By studying this wealth of information, the system learns to distinguish between benign user inputs and those that hide malicious intent. This initial phase educates the system, imparting it with the wisdom to recognize the subtleties of safe and harmful queries.

**2. Simulated Attack Scenarios:**

With knowledge in hand, the methodology ventures into the realm of creativity. Imagine a virtual playground where the system plays out countless attack scenarios. Through simulated attacks, the system explores the unexpected twists and turns that hackers might employ. This creative process, akin to brainstorming, enables the system to anticipate a wide array of potential threats.

*public boolean isRegistered (String id, String password) {*

*String driver = "com.mysql.jdbc.Driver";*

*String to = "jdbc:mysql://cc.com/credit";*

*Class.forName (driver).newInstance ();*

*Connection dbConn = DriverManager.getConnection (to);*

*String sqlQuery = "SELECT userinfo FROM users WHERE id =" + id + 11/ AND password = 1"+password + "";*

*Statement stmt = dbConn.createStatement ();*

*ResultSet rs = stmt.executeQuery (sqlQuery);*

*if (rs != null) return true;*

*else return false;*

*}*

**3. Pattern Recognition:**

Armed with acquired wisdom and creativity, the methodology incorporates a sharp eye for patterns. Known malicious patterns, reminiscent of familiar tricks used by mischievous hackers, are catalogued. The system vigilantly scans user inputs, comparing them against these predefined patterns. When a match is found, it raises a virtual eyebrow, signalling a potential intrusion attempt.

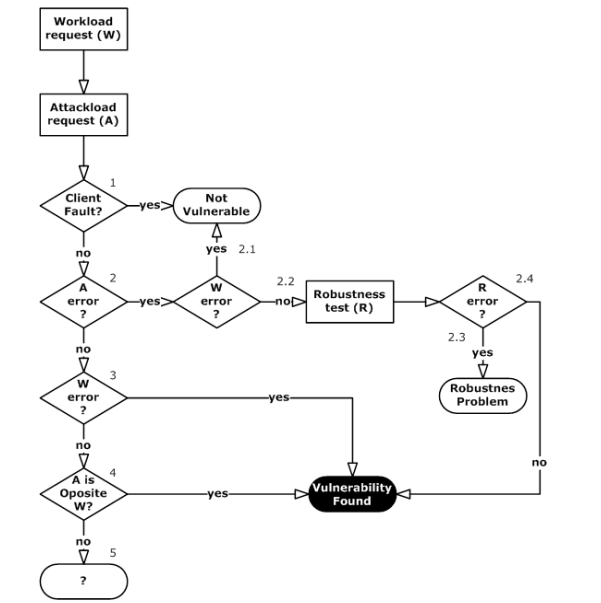
**4. Real-time Decision Making:**

The core of the methodology lies in real-time decision making. Each user input undergoes a rapid, yet thorough, analysis. The system evaluates it against its acquired knowledge, creativity, and pattern recognition prowess. Based on this evaluation, instantaneous decisions are made. If a query raises suspicion, the system acts swiftly, implementing security protocols to thwart the attempted attack.

**5. Continuous Learning and Adaptation:**

The methodology doesn’t stop at initial training. It embraces a philosophy of continuous learning and adaptation. New data, emerging attack strategies, and evolving patterns are constantly integrated into the system's knowledge. This adaptive nature ensures that the system remains one step ahead, ever-prepared to defend against the latest and most sophisticated SQL injection techniques.

In essence, this methodology mirrors the vigilance and adaptability of a seasoned security expert. By combining the lessons from historical data, the creativity to anticipate diverse threats, the sharpness to recognize patterns, and the agility to make real-time decisions, this approach forms a formidable defense against SQL injection attacks. It exemplifies the human-like intuition and analytical prowess embedded within a digital shield, safeguarding the intricate world of web applications from unseen dangers.



The first step is to determine whether the customer is willing to buy a product. This could be done by asking the customer a question, such as "Are you interested in buying this product?" or by observing the customer's behaviour, such as whether they are holding the product and looking at it closely.

If the customer is willing to buy the product, the program goes to the next step, which is to determine whether the customer can afford the product. This could be done by asking the customer about their budget or by checking the customer's credit score. If the customer can afford the product, the program goes to the next step, which is to determine whether the customer is ready to buy the product. This could be done by asking the customer if they want to complete the purchase or by observing the customer's behavior, such as whether they are putting the product in their cart.

If the customer is ready to buy the product, the program goes to the next step, which is to process the payment. This could be done by asking the customer for their credit card information or by processing a payment through a third-party payment processor. If the payment is successful, the program goes to the next step, which is to deliver the product to the customer. This could be done by shipping the product to the customer's address or by allowing the customer to pick up the product from a store. If the payment is not successful, the program goes to the next step, which is to ask the customer if they would like to try again. If the customer says yes, the program returns to the step where it determines whether the customer can afford the product. If the customer says no, the program ends.

Here is a more detailed explanation of each step in the flowchart: Determine whether the customer is willing to buy a product. This can be done by asking the customer a question, such as "Are you interested in buying this product?", or by observing the customer's behavior, such as whether they are holding the product and looking at it closely. If the customer is not willing to buy the product, the program ends. Determine whether the customer can afford the product. This can be done by asking the customer about their budget or by checking the customer's credit score.

If the customer cannot afford the product, the program ends. Determine whether the customer is ready to buy the product. This can be done by asking the customer if they want to complete the purchase or by observing the customer's behavior, such as whether they are putting the product in their cart. If the customer is not ready to buy the product, the program ends.

Process the payment.

This can be done by asking the customer for their credit card information or by processing a payment through a third-party payment processor. If the payment is not successful, the program asks the customer if they would like to try again. If the customer says no, the program ends.

Deliver the product to the customer.

This can be done by shipping the product to the customer's address or by allowing the customer to pick up the product from a store. Once the product has been delivered to the customer, the program ends.

1. **SQL INJECTION ATTACKS:**

Through a SQL query, a program can add, modify, or retrieve data in a database. SQL injection enables attackers to access, modify, or delete critical information in a database without proper authorization. Via SQL injection, attackers can also execute arbitrary commands with high system privilege in the worst case [2]. SQL injection has recently been one of the top issues in software security [1]. In many cases, SQL queries are dynamically constructed via user input. Despite there being several safer ways to make SQL queries in systems such as using Java’s Prepared Statement, queries are often dynamically generated in string concatenations, an unsafe and poor programming practice. For example, Figure 1 shows a sample program including a SQL query to authenticate a user via id and password. The query is dynamically created via the program statement in bold. In the query in Figure 1, id and password are obtained via user input.

A SQL injection attack occurs when an input from a user includes SQL keywords so that the dynamically-generated SQL query changes the intended function of the SQL query in the application. In the previous example, an attacker can enter the following input through the user inter-face for the values of id and password:

Username: ‘ OR ‘1’ = ‘1 Password: ‘ OR ‘1’ = ‘1

Which would generate the following query:

SELECT userinfo FROM users

WHERE id = ‘1’ OR ‘1’ = ‘1’

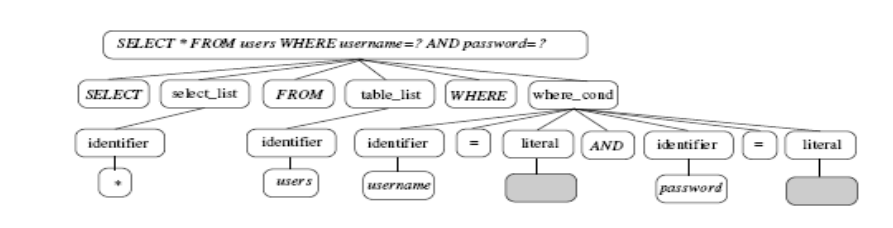
AND password = ‘1’ OR ‘1’ = ‘1’;

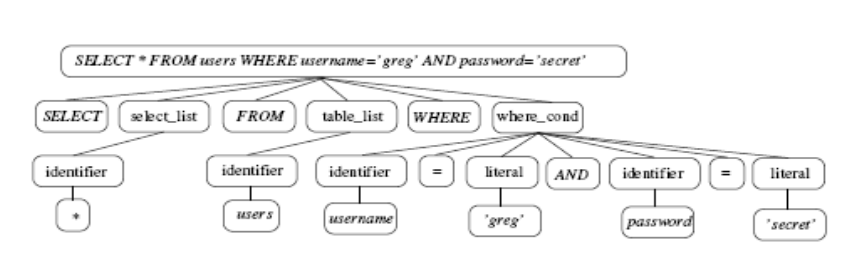
Because the given input makes the WHERE clause in the SQL statement always true (a tautology), the database returns all the user information in the table. Therefore, the malicious user has been authenticated without a valid login id and password. The use of tautology is a well-known SQL attack. However, there are other types of SQLI As using multiple SQL statements or stored procedures. SQL clauses such as “UNION SELECT”, “ORDER BY”, and “HAVING” are sometimes used to infer database structure. The attackers also can infer database structure by exploit error messages from SQL command failure or simply by trial and error.

1. **BACKGROUND AND RELATED WORKS:**

SQL Injection is the extension or modification of a web application's SQL statement by an attacker in order to extract or update information in the database that they are not authorized to access. Suppose a webapp generates the following SQL statement: SELECT author,title,year FROM books WHERE publisher = 'Wiley' and published=1 If an attacker were to enter a string such as: Wiley' OR 1=1— into the search form, this would result in the following :

**Query**: SELECT author,title,year FROM books WHERE publisher = 'Wiley' OR 1=1—'



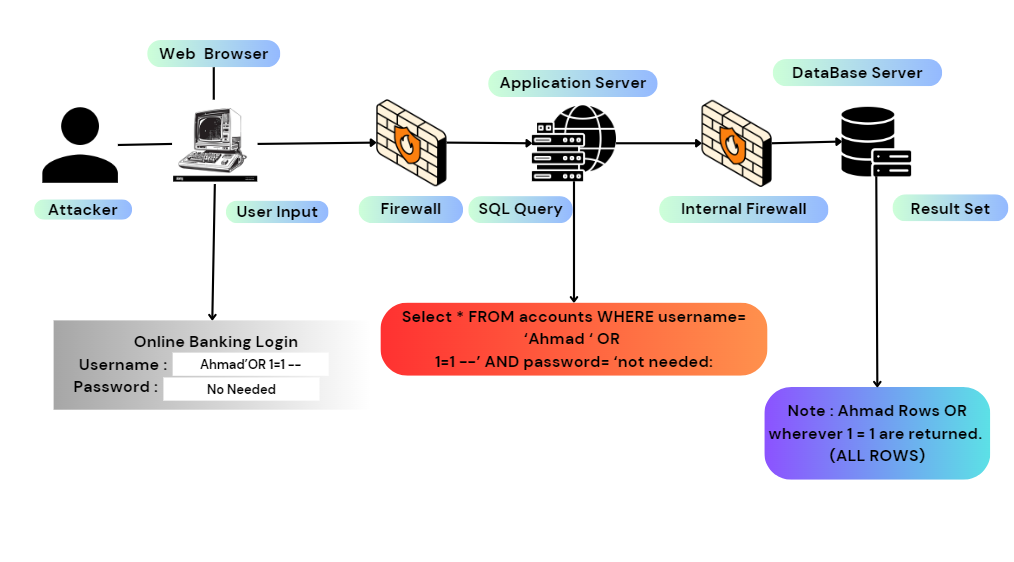


and published=1 This query would return every row in the database where the publisher is Wiley, or 1=1, which is always true, thus returning all rows in the database. There is much existing research that attempts to examine and mitigate SQL injection attacks. One technique in use involves static analysis of code to form a model that can be compared to dynamic queries at runtime to detect SQL injection attacks. In Lee et al, the authors propose a system that performs analysis of PHP-based websites to evaluate the SQL queries, then processes these queries by removing parameters to form general query structures, a process they refer to as query transformation. These are then compared to live SQL queries generated dynamically and a query that has a different structure will be flagged as an attack. In addition to removing parameters, Kar et al. generalize SQL queries into structural elements, and were able to achieve a 100% detection rate with their technique.

1. **ARCHITECTURE:**

Many different architectures have been explored for Intrusion Detection Systems (IDS). In Djanali et al., the authors create a clustered architecture using Raspberry Pi computing devices. This project uses the HIHAT honeypot system and the SQL injection detection technique proposed in Lee et al. The system uses a load-balancing server to route computation to the Raspberry Pi cluster. 9 Sadasivam et al.propose a distributed, multi-honeypot architecture. They use several different honeypot systems connected to a front-end server, and the system is modular to accommodate the addition or removal of honeypot systems. This distributed architecture has the advantage that it's capable of gathering more data than any one of the honeypot systems, and the authors mention that the attacks they detected were primarily targeting SSH, as well as MySQL, MSSQL, and telnet, and originate primarily in China and the US.

Tawari and Jane propose a complicated architecture they refer to as a "virtual honeynet" designed to maximize the time spent by an attacker in the system and thus the amount of data gathered about attackers. This system uses the SNORT IDS to detect malicious traffic which is then diverted to the honeynet. The virtual honeypots use the HIHAT system as well as the Sebek data capture tool as components, and combine these into a complicated architecture designed to confuse and delay attackers. In the Honeydoop system, the authors propose a dynamic architecture using Hadoop for data collection. This system uses dynamic allocation based on currently observed network conditions, with the goal of efficient utilization of resources and increased security. The authors mention that in their experience these dynamically created honeypots receive much more traffic than those that are statically generated. The authors use Snort for traffic capture, but the architecture is modular so any IDS system could be used.



In the diagram, the attacker sends an SQL query to the web application that contains malicious code. The web application then executes the query without properly sanitizing it, which allows the attacker to gain access to the database.

For example, the attacker might send the following query:

SELECT \* FROM accounts WHERE username= 'Ahmad' OR 1=1--

The OR 1=1 part of the query is malicious code. It will always evaluate to true, which means that the query will return all rows in the accounts table, regardless of the value of the username column.

If the web application does not properly sanitize the query, it will execute it as is and return all rows in the accounts table to the attacker. This could give the attacker access to sensitive information such as usernames, passwords, and credit card numbers.

SQL injection attacks can be very dangerous, and they can be used to steal data, modify data, or even take down entire websites. It is important for web developers to take steps to prevent SQL injection attacks, such as sanitizing all user input and using prepared statements.

Here are some tips for protecting your website from SQL injection attacks:

* Sanitize all user input. This means validating and escaping all user input before using it in SQL queries.
* Use prepared statements. Prepared statements allow you to create SQL queries with parameters that can be bound to values later. This prevents SQL injection attacks by preventing attackers from inserting malicious code into the query.
* Keep your database software up to date. Database software vendors regularly release security patches that fix vulnerabilities that could be exploited by attackers.
* Use a web application firewall (WAF). A WAF can help to protect your website from a variety of attacks, including SQL injection attacks.

If you are concerned about the security of your website, you should contact a qualified web security professional for assistance.

1. **ALGORITHMS AND METHODS:**

**Logistic Regression:**

Logistic Regression is like a detective trying to find clues to solve a mystery. In our case, it's trying to predict whether a given input (like user data) belongs to a certain category (like vulnerable or not vulnerable to SQL injection). It examines the relationship between the input variables and uses this information to make a probabilistic prediction. Think of it as a detective analysing different pieces of evidence to decide if a person is a suspect or not.

**Random Forest:**

Random Forest is like a group of detectives, each with their own expertise. In this case, these detectives are decision trees. A decision tree is like a flowchart that asks a series of questions about the input data, leading to a decision. Now, imagine a forest where each tree (detective) has its unique set of questions. Random Forest combines the answers from all these detectives to make a more informed and accurate decision. It's like having a team of detectives working together, each contributing their insights to solve the mystery.

**Decision Tree:**

A Decision Tree is a single detective. It works by asking a series of questions about the input data. For instance, in the context of SQL injection vulnerability detection, it might ask questions like "Does the input contain certain SQL keywords?" or "Is the input properly validated?" Based on the answers, it makes a decision about whether the input is vulnerable to SQL injection or not. Think of it as a detective following a trail of clues, making decisions at each step until a conclusion is reached.

**Fuzz Technique (Fuzz Testing):**

The concept of fuzz testing is implemented through the fuzz\_test\_cases function. Fuzz testing involves generating random or mutated inputs to test how a system handles unexpected data. In the context of SQL injection detection, the code creates mutated SQL query test cases by performing random mutations on the user input. These mutations simulate potential attack scenarios, allowing the system to assess its ability to detect and defend against various SQL injection attempts. Fuzz testing helps ensure the system's robustness by exploring different input possibilities and uncovering potential vulnerabilities.

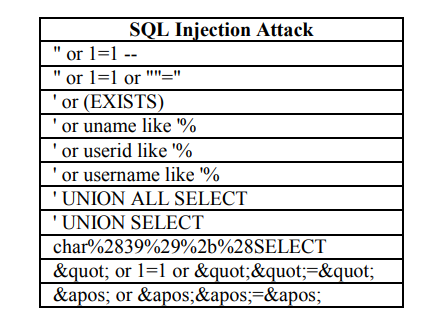
**Model Checking:**

Model checking is not directly implemented in the provided code. However, the underlying machine learning model can be considered a form of model checking. The machine learning model is trained to recognize patterns within SQL queries, essentially creating a model of safe and malicious query behaviors. During the evaluation phase (classifier.predict), the system checks the user input against this learned model. If the input query matches a known malicious pattern, the model raises an alert, indicating a potential SQL injection attempt. While it's not traditional model checking, the process involves comparing the user input against a learned model to make a decision about its safety.

**Pattern Matching:**

Pattern matching is implemented through the pattern\_matching function in the provided code. The function uses regular expressions (regex patterns) to identify known SQL injection patterns within the user input. By comparing the input query against a list of predefined regex patterns (sql\_injection\_patterns), the system can detect specific malicious patterns commonly associated with SQL injection attacks. If a match is found, the system categorizes the input as a potential SQL injection attempt. Pattern matching provides a rule-based approach to identifying specific attack patterns, complementing the machine learning-based detection in the code.

1. **CHEAT CODES:**



1. **PROPOSED METHODS:**

**1. Adaptive Machine Learning Enhancement:**

One proposed method involves refining the machine learning model’s adaptability. By incorporating continuous learning techniques, the model evolves in real-time, assimilating new data and evolving attack strategies. This adaptive learning ensures that the system remains up-to-date with emerging threats, enhancing its detection accuracy and resilience.

**2. Behavioural Fingerprinting:**

Introducing the concept of behavioural fingerprinting, this method focuses on analysing user input patterns. By studying the unique ways users interact with applications, deviations from normal behaviour can be detected. Unusual input sequences or excessive query length, for

instance, can raise suspicion, prompting further scrutiny and potential threat identification.

**3. Semantic Analysis and Contextual Understanding:**

Delving deeper into query semantics, this method proposes a contextual understanding of user intent. By analyzing the meaning behind queries, the system can distinguish between harmless requests and potentially malicious injections. Semantic analysis techniques, combined with context-aware algorithms, provide a nuanced approach to identifying SQL injection attempts, enhancing the system’s accuracy in differentiating between benign and harmful queries.

**4. Multi-Source Data Fusion:**

Expanding on the concept of multi-source data analysis, this method integrates diverse data streams. By merging information from various sources, such as network logs, user behavior data, and system activity logs, a comprehensive view of application interactions is obtained. Analyzing this amalgamated data allows for a holistic understanding of user behavior, enabling the system to spot inconsistencies and detect suspicious SQL injection attempts more effectively.

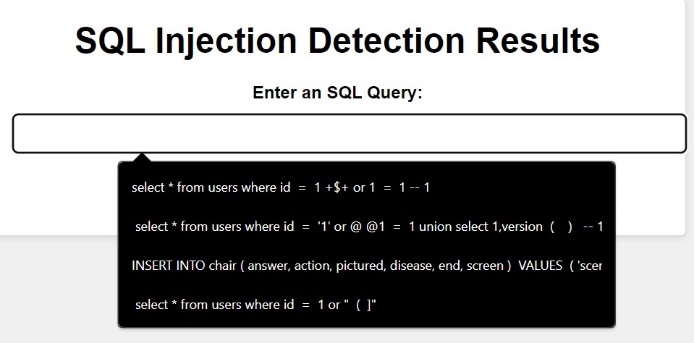
**5. Natural Language Processing (NLP) Integration:**

Leveraging the power of natural language processing, this method explores the linguistic nuances within user queries. By understanding the semantics of language, including synonyms, metaphors, and contextual cues, the system gains a deeper insight into user intentions. NLP algorithms can identify subtle manipulations within queries, making the detection process more sophisticated and accurate.

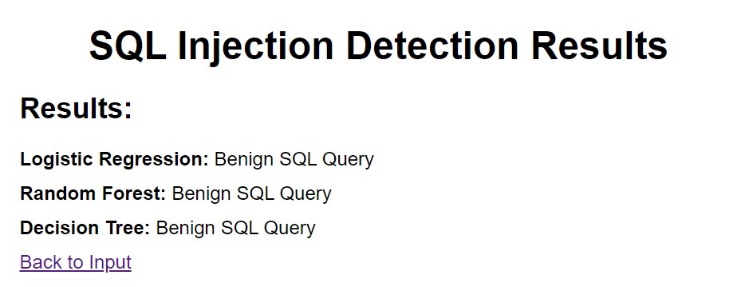
1. **RESULTS AND DISCUSSION:**



The above picture is the front end part of our project which we have developed using Html, Css, JS. We have connected the front end and back end by using the python library Flask.

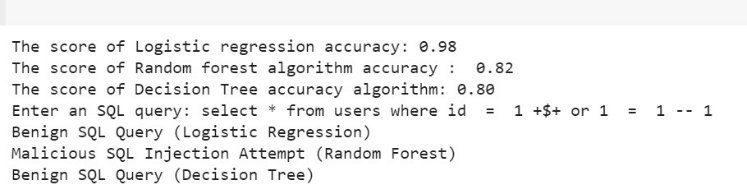


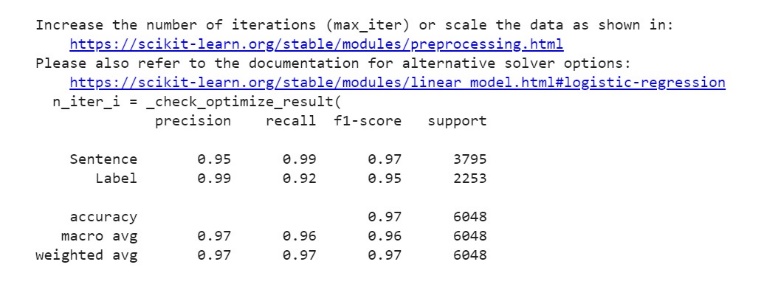
The above picture is about giving the sql query from which we will be getting whether the given sql query is good or bad query. This page appears after the main page which we have above.



The above picture is the results which we got from the given sql query. We are using three algorithms that are logistic regression, random forest, decision tree. From the three algorithms we are displaying whether the given sql query is

benign or not.





The above two pictures show the brief results of the three algorithms.

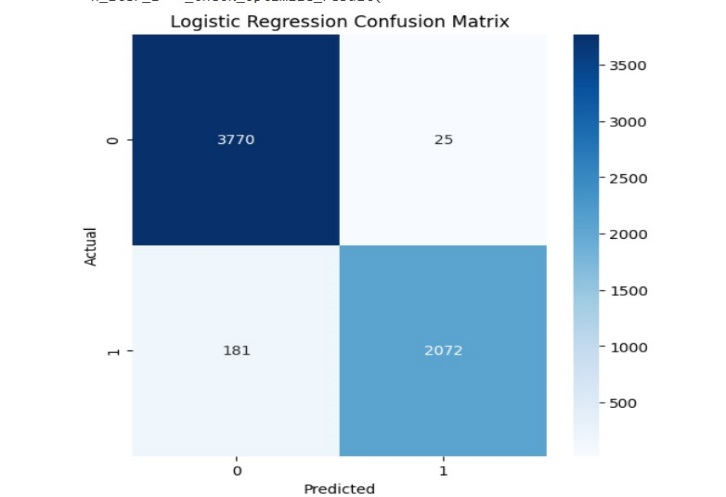
Here's how the confusion matrix terms relate to the SQL injection detection scenario in the given code:

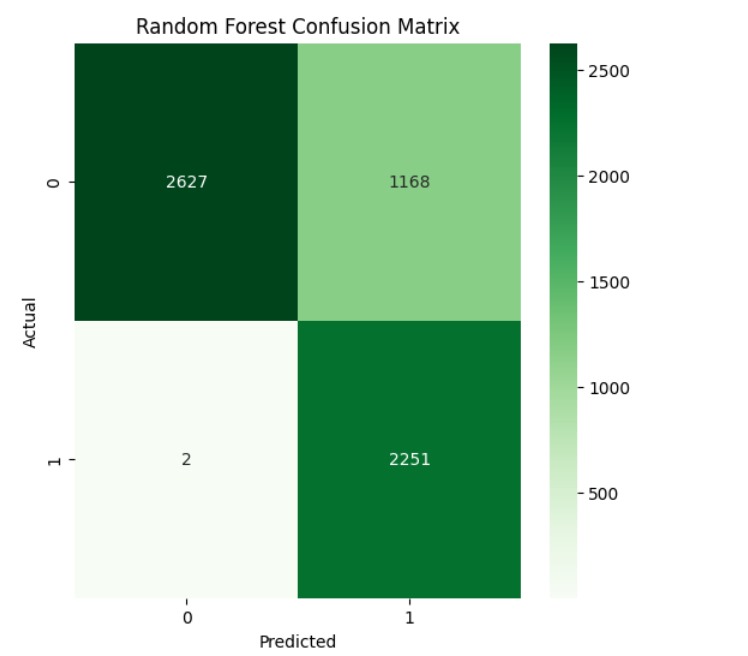
True Positive (TP): This represents the cases where the model correctly predicts a malicious SQL injection attempt. In the context of the code, it indicates the number of actual SQL injection attempts that were correctly identified by the model as malicious.

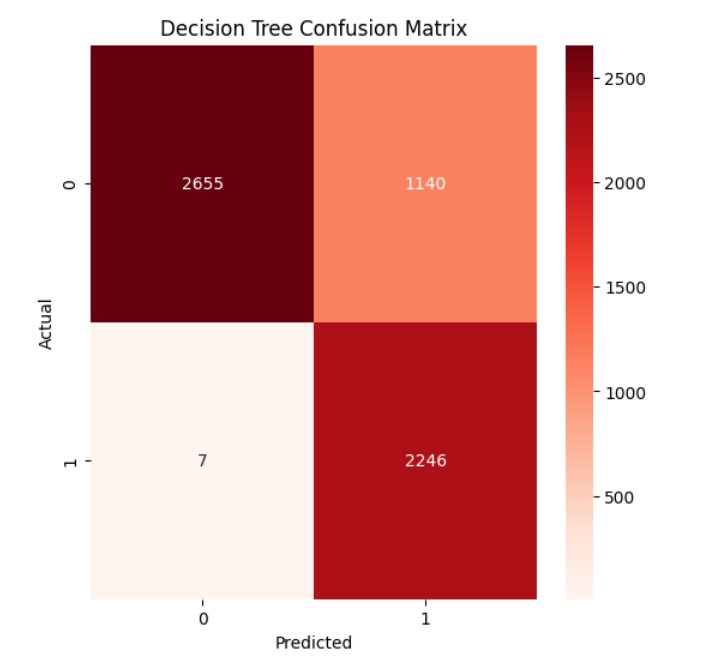
True Negative (TN): This represents the cases where the model correctly predicts a benign SQL query. In the code, it indicates the number of actual harmless SQL queries that were correctly identified as benign.

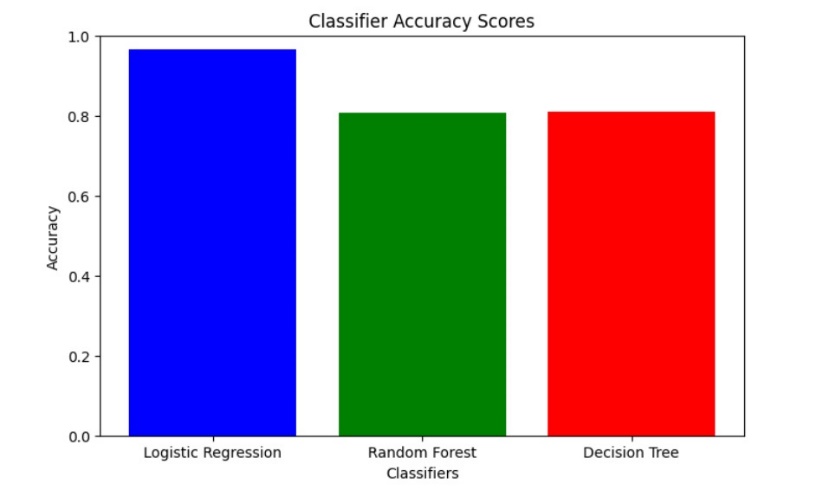
False Positive (FP): Also known as Type I error, this occurs when the model incorrectly predicts a benign SQL query as malicious. In the context of the code, it means the number of harmless SQL queries that were wrongly flagged as SQL injection attempts.

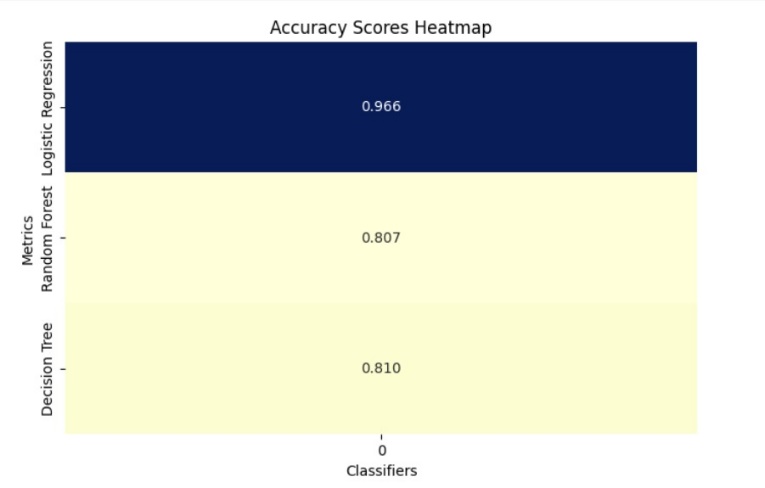
False Negative (FN): Also known as Type II error, this happens when the model incorrectly predicts a malicious SQL injection attempt as benign. In the code, it represents the number of actual SQL injection attempts that were missed and not detected by the model.











The heatmap color-codes these values, making it easy to spot patterns and areas where the model might need improvement. Generally, a confusion matrix heatmap with bright colors in the diagonal (representing correct predictions) and darker colors off the diagonal (representing incorrect predictions) indicates a well-performing model.

**Limitations:**

1. Insufficient Coverage; The techniques used to detect SQL injection may not cover all attack vectors or vulnerabilities, in an application. As new attack methods and vulnerabilities arise it can take time for detection methods to adapt.
2. False. Missed Threats; The detection techniques can sometimes produce alarms flagging legitimate SQL queries as suspicious or fail to identify actual attacks. Striking a balance between detecting threats and avoiding false alarms is a challenging task.
3. Reliance on Established Patterns; Many detection methods rely on recognizing patterns of SQL injection attacks. However they may not be effective against discovered vulnerabilities or customized attack payloads that do not match these established patterns.
4. Code Adjustments; Some detection techniques require developers to modify their code or adopt coding practices. This process can be time consuming. May not be feasible for systems.
5. Complexity of Applications; Modern web applications are often complex with layers and components. Detecting SQL injection vulnerabilities in environments poses challenges. The likelihood of false alarms increases when dealing with intricate codebases.
6. Technology Limitations; Certain detection techniques are tailored to programming languages, frameworks or database management systems. This restricts their applicability in environments or, for applications built using common technologies.
7. Some detection methods, like runtime monitoring and security gateways can have an impact, on performance. This might influence how quickly the application responds and its ability to scale.
8. **CONCLUSION:**

In this paper we propose an approach to evaluate and compare web application vulnerability scanners. It is based on the injection of realistic software faults in web applications in order to compare the efficiency of the different tools in the detection of the possible vulnerabilities caused by the injected bugs. The results of the evaluation of three leading web application vulnerability scanners show that different scanners produce quite different results and that all of them leave a considerable percentage of vulnerabilities undetected. The percentage of false positives is very high, ranging from 20% to 77% in the experiments performed. The results obtained also show that the proposed approach allows easy comparison of coverage and false positives of the web vulnerability scanners. In addition to the evaluation and comparison of vulnerability scanners, the proposed approach also can be used to improve the quality of vulnerability scanners, as it easily shows their limitations. For some critical web applications several scanners should be used and a hand scan should not be discarded from the process.

1. **REFERENCES:**

* [1] OWASP Top 10, 2013, "Top 10 2013-A1-Injection",

https://www.owasp.org/index.php/Top\_10\_2013- A1-Injection

* [2] Application Defense Center (ADC), 2015, "2015 Web Application Attack Report (WAAR)", https://www.imperva.com/docs/HII\_Web\_Application\_Attack\_Report\_Ed6.pdf
* [3] M. Moh, S. Pininti, S. Doddapaneni, T-S. Moh, Detecting Web Attacks Using Multi-Stage Log Analysis, 2016 IEEE 6th International Conference on Advanced Computing (IACC), Bhimavaram, 2016, pp. 733-738.
* [4] K. Ross, M. Moh, T-S. Moh, J. Yao, Poster: Multi-Source Data Analysis For SQL Injection Detection, 38th IEEE Symposium on Security and Privacy (IEEE S&P), San Jo-se, CA, 2017.
* [5] S. Djanali, F.X. Arunanto, B.A. Pratomo, H. Studiawan, S.G. Nugraha, "SQL injection detection and prevention system with raspberry Pi honeypot cluster for trapping attacker," in Technology Management and Emerging Technologies (ISTMET), 2014 International Symposium on, vol., no., pp.163-166, 27-29 May 2014
* [6] Anamika Joshi, Geetha V “SQL Injection Detection using Machine Learning” in International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2014.
* [7] Kevin Ross “SQL Injection Detection Using Machine Learning Techniques and Multiple Data Sources” in San Jose State University, 2018.
* [8] Maha Alghawazi, Daniyal Alghazzawi, Suaad Alarifi “Detection of SQL Injection Attack Using Machine Learning Techniques: A Systematic Literature Review” in Information Systems Department, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 80200, Saudi Arabia 2022.
* [9] Atefeh Tajpour, Mohammad JorJor zade Shooshtari “Evaluation of SQL Injection Detection and Preventation Techniques” in 2010 Second International Conference on Computational Intelligence, Communication Systems and Networks.
* [10] MeiJunjin “An approach for SQL injection vulnerability detection”, Huangshi Institute of Technololgy, The north street of Guilin ,Huangshi,Hubei province,China,435003.
* [11] P.Grazie., PhD SQLPrevent thesis. University of British Columbia (UBC) Vancouver, Canada.2008.
* [12] C. Gould, Z. Su, and P. Devanbu. JDBC Checker:A Static Analysis Tool for SQL/JDBC Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04) Formal Demos, pp 697–698, 2004.
* [13] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In Proceedings of the 26th International Conference on Software Engineering (ICSE 04).
* [14] M. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In Proceedings of the 20th Annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2005), pp 365–383, 2005.
* [15] Y. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 12th International World Wide Web Conference (WWW 04), May 2004.
* [16] W. G. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA, Nov 2005.
* [17] W. G. Halfond and A. Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005), pp 22–28, St. Louis, MO, USA, May 2005.
* [18] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In Proceedings of the 11th International World Wide Web Conference (WWW 03), May 2003.
* [19] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks through Context-Sensitive String Evaluation. In Proceedings of Recent Advances in Intrusion Detection (RAID2005), 2005.
* [20] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting Information. In Twentieth IFIP International Information Security Conference (SEC 2005), May 2005.
* [21] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In Proceedings 21st Annual Computer Security Applications Conference, Dec. 2005.
* [22] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference, pp 292–302. June 2004.