Automated Detection System for SQL Injection Attack

Dr K.V.N.Sunitha k.v.n.sunitha@gmail.com
Professor &Head,
Department of Computer Science & Engineering,
G.Narayanamma Institute of Technology and Science
Shaikpet, Hyderabad – 500 008, A.P., India

Mrs.M. Sridevi sreetech99@gmail.com
Assoc.Professor,
Department of Computer Science & Engineering
Laqshya Institute of Technology and Science
Konijerla, Khammam – 507 305 , A.P., India

Abstract

Many software systems have evolved as Web-based t that makes them available to the public via the Internet and can expose them to a variety of Web-based attacks. One of these attacks is SQL Injection vulnerability (SQLIV), which can give attackers unrestricted access to the databases that underlie Web applications and has become increasingly frequent and serious. The intent is that Web applications will limit the kinds of queries that can be generated to a safe subset of all possible queries, regardless of what input user provides. SQL Injection attacks are possible due to the design drawbacks of the web sites, which interact with back-end databases. Successful attacks may damage more. We introduce a system that deals with new automated technique for preventing SQL Injection Attacks based on the novel concept of regular expressions is to detect SQL Injection attacks. The proposed system can detect the attacks that are from Internet and Insider Attacks, by analyzing the packets of the network servers.

Keywords—Intrusion Detection, Injection Attacks, Regular Expressions, SQL Query.

1. INTRODUCTION

Nowadays it is most common for any organization to use database and web application for maintaining their information. Security of these systems became crucial. Internet threats like SQL Injection attacks on database through web applications are more. Solutions for to avoid these attacks are 1. Placing a powerful Network SQL Injection Intrusion Detection Systems (IDS). 2. SQL Injection Insider Misuse Detection Systems(SQLIMDS).

Web applications interface with databases that contain information such as customer names, preferences, credit card numbers, purchase orders, and so on. Web applications build SQL queries to access these databases based, in part, on user-provided input. Inadequate input validation can enable attackers to gain complete access to such databases. One way in which this happens is that attackers can submit input strings that contain specially encoded database commands. When the Web application builds a query by using these strings and submits the query to its underlying database, the attacker's embedded commands are executed by the database and the attack succeeds. The results of these attacks are often disastrous and can range from leaking of sensitive data to the destruction of database contents.

The cause of SQL injection vulnerabilities are relatively simple and well understood: a kind of problem is insufficient validation of user input, developers have proposed a range of coding guidelines that promote defensive coding practices, such as encoding user input and validation. A rigorous and systematic application of these techniques is an effective solution for preventing SQL injection vulnerabilities.

However, in practice, the application of such techniques is human-based and, thus, prone to errors. Furthermore, fixing legacy code-bases that might contain SQL injection vulnerabilities can be an extremely labor-intensive task. Although recently there has been a great deal of attention to the problem of SQL injection vulnerabilities, many proposed solutions fail to address the full scope of the problem. There are many types of SQLIA and countless variations on these basic types. Researchers and practitioners are often unaware of the myriad of different techniques that can be used to perform SQLIA. Therefore, most of the solutions proposed detect or prevent only a subset of the possible SQLIA.

An SQL injection attack (SQLIA) is a type of attack on web applications that exploits the fact that input provided by web clients is directly included in the dynamically generated SQL statements. SQLIA is one of the foremost threats to web applications. According to the SQLIMDS Foundation, injection flaws, particularly SQL injection, were the second most serious web application vulnerability type in 2007. Since they are easy to find and exploit, SQL injection vulnerabilities are frequently employed by attackers.

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements. Attackers trick the SQL engine into executing unintended commands via supplying specially crafted string input, thereby gaining unauthorized access to a database in order to view or manipulate restricted data. Using SQLIAs, an attacker may be able to read, modify, or even delete database information. In many cases, this information is confidential or sensitive and its loss can lead to problems such as identity theft and fraud.

In general, SQL injection attacks are a class of code injection attacks that take advantage of the lack of validation of user input. These attacks occur when developers combine hard-coded strings with user-provided input to create dynamic queries. Intuitively, if user input is not properly validated, attackers may be able to change the developer's intended SQL command by inserting new SQL keywords or operators through specially crafted input strings.

We propose a new highly automated approach for dynamic detection of SQL injection attacks. Intuitively, our approach works by identifying "trusted" strings in an application and allowing only these trusted strings to be used to create the semantically relevant parts of a SQL query such as keywords or operators. The general mechanism that we use to implement this approach is based on dynamic tainting, which marks and tracks certain data in a program at runtime. The kind of dynamic tainting that we use gives our approach several important advantages over techniques based on other mechanisms. Our approach is highly automated, does not rely on complex static analyses and is both efficient and precise and, in most cases, requires minimal or no developer intervention. Compared to other existing techniques based on dynamic tainting, our approach makes several conceptual and practical improvements that take advantage of the specific characteristics of SQLIA. The first conceptual advantage of our approach is the use of positive tainting. The second conceptual advantage of our approach is the use of flexible syntax-aware evaluation. The practical advantages of our approach are that it imposes a low overhead on the application and it has minimal deployment requirements. Efficiency is achieved by using a specialized library, called Meta Strings, that accurately and efficiently assigns and tracks trust markings at runtime. The only deployment requirements for our approach are that the Web application must be instrumented and it must be deployed with our Meta Strings library, which is done automatically.

2. SQL INJECTION ATTACK APPROACH

Our approach makes several conceptual and practical improvements over traditional dynamic tainting ap proaches by taking advantage of the characteristics of SQLIAs and Web applications. First, unlike existing dynamic tainting techniques, our approach is based on the novel concept of positive tainting, that is, the identification and marking of trusted, instead of untrusted, data. Second, our approach performs accurate and efficient taint propagation by precisely tracking trust markings at the character level. Third, it performs syntax-aware evaluation of query strings before they are

sent to the database and blocks all queries whose non-literal parts (that is, SQL keywords and operators) contain one or more characters without trust markings. Finally, our approach has minimal deployment requirements, which makes it both practical and portable. The following sections discuss these key features of our approach in detail [8].

A. Positive Tainting VS Negative

Positive tainting differs from traditional tainting (negative tainting) because it is based on the identification, marking, and tracking of trusted, rather than untrusted, data as shown in Figure 1 . In the context of preventing SQLIAs, the conceptual advantages of positive tainting are especially significant.

With positive tainting, incompleteness may lead to false positives, but it would never result in an SQLIA escaping detection. Moreover, as explained in the following, the false positives generated by our approach, if any, are likely to be detected and easily eliminated early during prerelease testing. Positive tainting uses a white-list, rather than a block list policy.

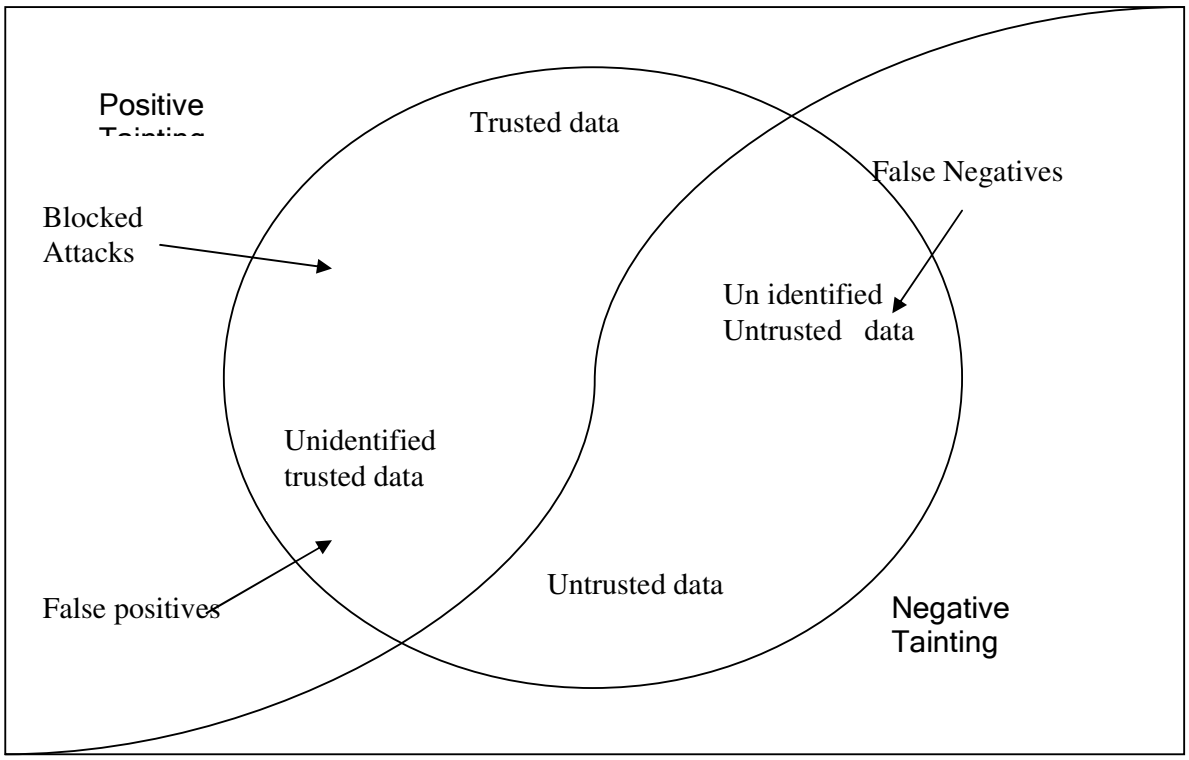

FIGURE 1: Positive Tainting vs Negative Tainting.

B. Accurate and Efficient Taint Propagation

Taint propagation consists of tracking taint markings associated with the data while the data is used and manipulated at runtime. In our approach, we provide a mechanism to accurately mark and propagate taint information by 1) tracking taint markings at the “right” level of granularity and 2) precisely accounting for the effect of functions that operate on the tainted data [7, 8, 9].

Character-level tainting. We track taint information at the character level rather than at the string level. We do this because, for building SQL queries, strings are constantly broken into sub strings, manipulated, and combined. By associating taint information to single characters, our approach can precisely model the effect of these string operations.

Accounting for string manipulations. To accurately maintain character-level taint information, we must identify all relevant string operations and account for their effect on the taint markings. Our approach achieves this goal by extending all classes and methods (that perform String manipulations), by adding functionality to update taint markings based on the methods’ semantics.

C. Syntax-aware Evaluation

Our technique performs syntax-aware evaluation of a query string immediately before the string is sent to the database to be executed. To evaluate the query string, the technique first uses a SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (i.e., sub strings) other than literals contain only trusted data. If all such tokens pass this check, the query is considered safe and is allowed to execute. If an attack is detected, a developer specified action can be invoked.

Our technique performs syntax-aware evaluation of a query string immediately before the string is sent to the database to be executed. To evaluate the query string with use of regular expressions , this technique first uses a SQL parser to break the string into a sequence of tokens that correspond to SQL keywords, operators, and literals. The technique then iterates through the tokens and checks whether tokens (that is, substrings) other than literals contain only trusted data. If all such tokens pass this check, the query is considered safe and is allowed to execute. If an attack is detected, a developer specified action can be invoked.

The proposed system is a Network Intrusion Detection System (NIDS), will be deployed in between corporate network and the internet. All the packets from corporate network to the internet and the internet to the corporate network will pass through the NIDS,. This system captures the http traffic,. And stores them in a folder called pcaps. This folder is meant for network packet files. Because of the network traffic is mix of http and non-http traffic, the proposed NIDS is meant for the SQL Injection attacks and SQL Injections are web attacks. We filter for the HTTP URLS of the pcaps and extracts them to packet.dat file.

3. DEVELOPING REGULAR EXPRESSIONS

The proposed system is a Network Intrusion Detection System (NIDS), will be deployed in between corporate network and the internet. All the packets from corporate network to the internet and the internet to the corporate network will pass through the NIDS,. This system captures the http traffic and stores them for network packet files. The Regular Expression Development Process is Gathering Attack Patterns, Common Regular Expression Development with Regex-Coach, Testing With Regex-Coach. For the attack patterns shown in Figure 2 and Figure 3, such as

‘ or 1=1 -- “
or 123=123 ‘
or ‘a’ = ‘a’ like patterns

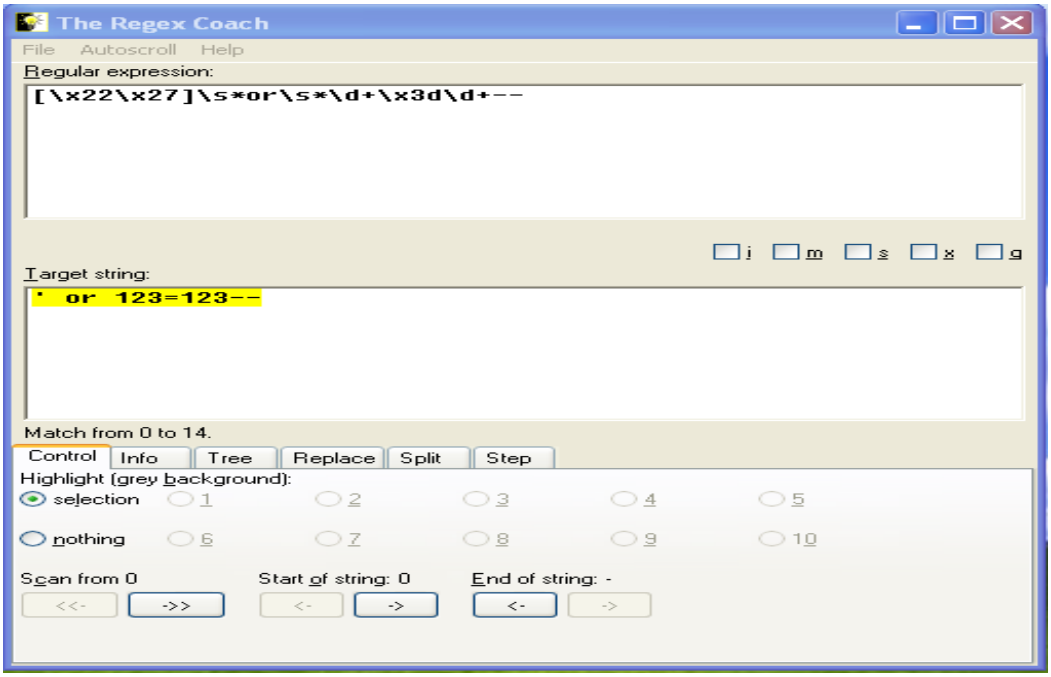

FIGURE 2: Regular Expression Will Match With 123=123 Like Patterns.

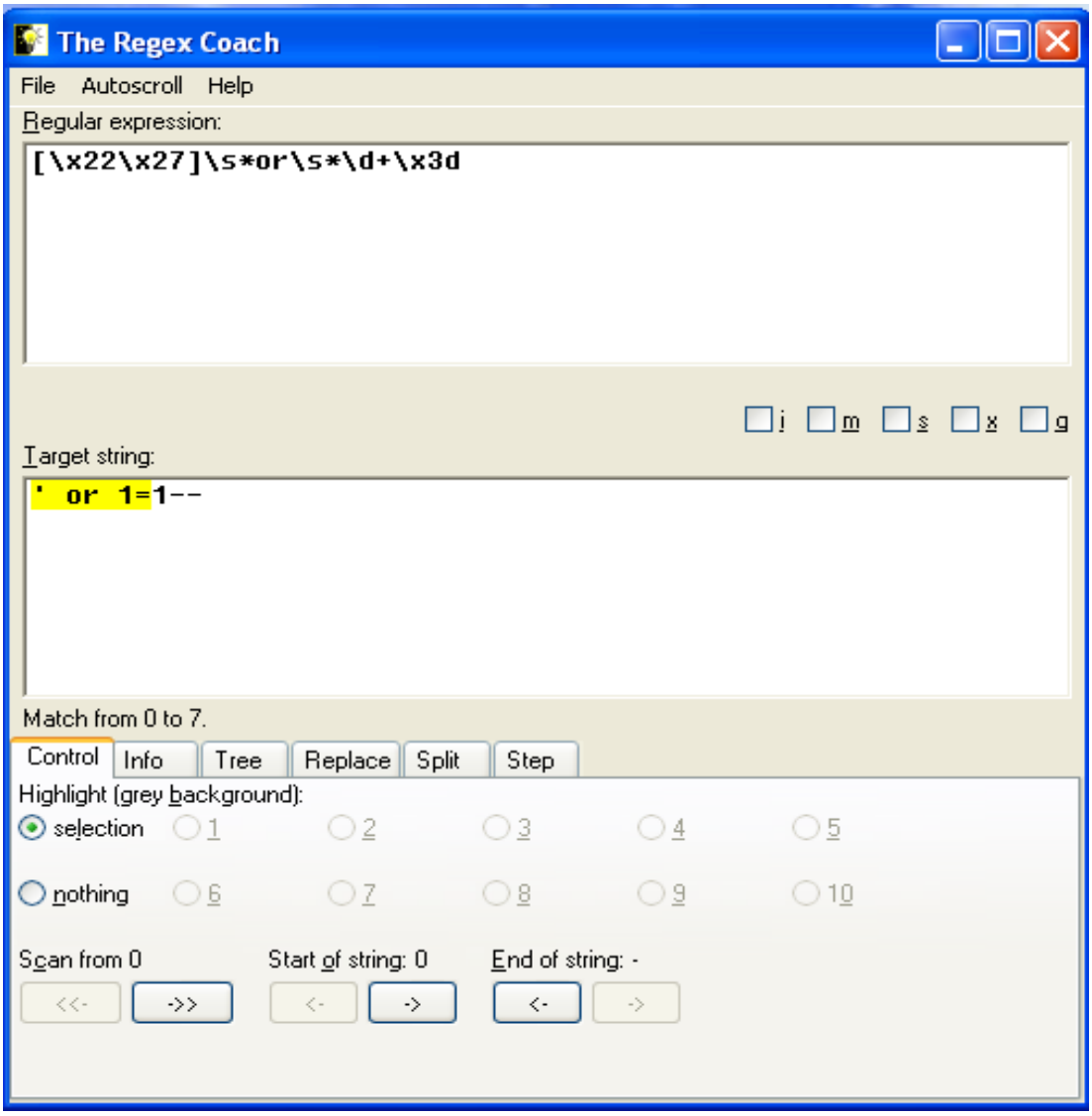

FIGURE 3: Regular Expression Will Match With 'Or 1=1 ' Like Patterns

Because of the network traffic is mix of http and non-http traffic, the proposed NIDS is meant for the SQL Injection attacks and SQL Injections are web attacks. After capturing the packets the decoded patterns will be checked against all the regular expressions. If any rule matches, then that log information will be send by the Detection System.

4. ALGORITHM DETAILS

Our system has three major steps:

- vulnerability detection,
- preparing the regular expressions,
- report generation.

As per Figure 4. first captures packets from the Ethernet , extract the decoded URL and check with regular expression which already in directory if match ok generate logs.

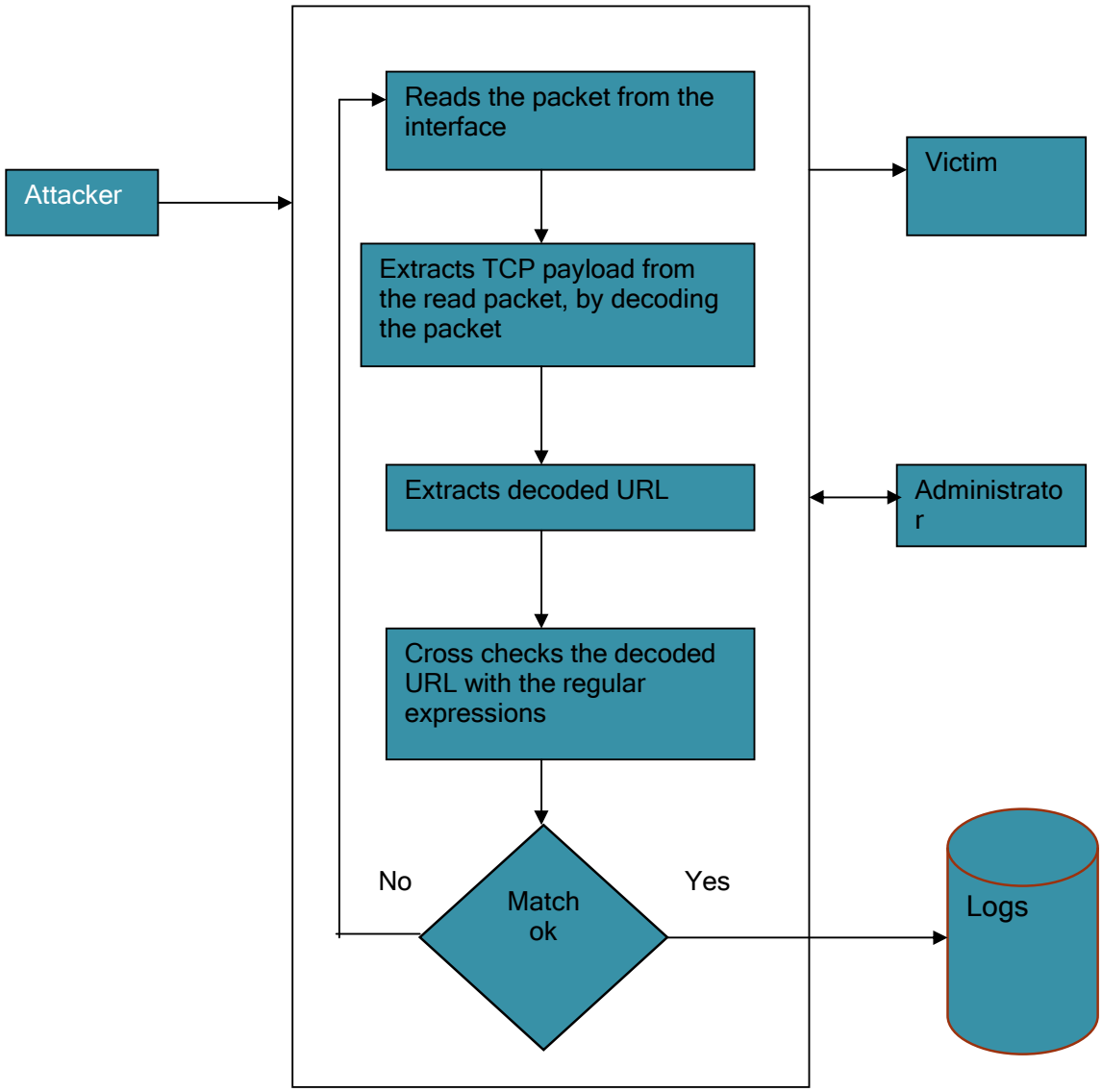

FIGURE 4: Detection Process System Design

Decoding the Encoded URLs:
In general URLs will be in the encoded format.

For example
GET http://www.google.com/hl=en&qa=sql%20injection HTTP/1.1

In the above example %20 is an encoded character of the ASCII character space.
Detection System decodes all the captured URL.
Decoded URL: http://www.google.com/hl=en&qa=sql injection

Checking against the regular expressions:
The decoded patterns will be checked against all the regular expressions. If any rule matches, then that log information will be send by the Detection System.

A.Sample Attack Patterns
Below attack Patterns are gathered from
http://www.milw0rm.com
Each attack is represented as

<Milw0rmid> ,
<Vulnerable page> ,
<Vulnerable field>,
<Attack pattern>
For example :

1. 7382
tbl_structure.php
table
TABLES%60+where+0+union+select+char%2860%2C+63%2C+112%2C+104%2C+112%2C+32%2C+101%2C+118%2C+97%2C+108%2C+40%2C+36%2C+95%2C+71%2C+69%2C+84%2C+91%2C+101%2C+93%2C+41%2C+63%2C+62%29+into+outfile+%22%2Fvar%2Fwww%2Fbackdoor.php%22+--+1
2. 7378
treplies.asp
message
20814+union+select+1,2,3,4,5,6,7,8+from+msysobjects

B. Sample Rules to Detect SQL Injection Patterns

1.Detects basic SQL authentication bypass attempts
Rule:
(?:^\s*[;>"]\s*(?:union|select|create|rename|truncate|load|alter|delete|updateiinsert|desc))|(?:(?:select|create|rename|truncate|load|alter|delete|update|insert|desc)\s+(?:concat|char|load_file)\s?\(?)|(?:end\s*\);)|("\s+regexp\W)

2. Detects conditional SQL injection attempts.
Rule :
(?:having\s+[\d\w]\s?=)|(?:if\s?\([\d\w]\s?=)

3. Detects basic SQL authentication bypass attempts 1/3
Rule:
(?:^*admin\s"|(\/*)+"+\s?(?:--|#|\/*|{)?)|(?:"\s*or[\w\s-]+\s*[+<>=(),\s*[\d"])|(?:"\s*[^\w\s]?=\s*")|(?:"\W*[+=]+\W*")|(?:"\s*[!=|][\d\s!=+-]+.*["(].*$)|(?:"\s*[!=|][\d\s!=]+.*\d+$)|(?:"\s*like[+=\s\.-]+[\d"(])|(?:\sis\s*0\W)|(?:where\s[\s\w\.,-]+\s=)

4. Detects basic SQL authentication bypass attempts 2/3
Rule:
*(?:*union\s*(?:all|distinct)?\s*[([]\s*select)|(?:like\s*"\%)|(?:"\s*like\W*["\d])|(?:"\s*(?:n?and|x?or|not|\|\||\&\&)\s+[\s\w]+=\s*\w+\s*having)|(?:"\s**\s*\w+\W+")|(?:"\s*[^?\w\s=.,;\/)(]+\s*[(@]*\s*\w+\W+\w)|(?:select\s*[\[\]()\s\w\.,-]+from)

5. Detects basic SQL authentication bypass attempts 3/3
Rule:
(?:(?:n?and|x?or|not\|\||\&\&)\s+[\s\w+]+(?:regexp\s*\(|sounds\s+like\s*"|[=\d]+x))|("\s*\d\s*(?:--|#))|(?:"[%&<>^=]+\d\s*(=|or))|(?:"\W+[\w+-]+\s*=\s*\d\W+")|(?:"\s*is\s*\d.+"?\w)|(?:"\|?[\w-]{3,}[^\w\s.]+")|(?:"\s*is\s*[\d.]+\s*\W.*")

6. Detects concatenated basic SQL injection and SQLLFI attempts
Rule :
(?:^\s*[;>"]\s*(?:union|select|create|rename|truncate|load|alter|delete|update|insert|desc))|(?:(?:select|create|rename|truncate|load|alter|delete|update|insert|desc)\s+(?:concat|char|load_file)\s?\(?)|(?:end\s*\);)|("\s+regexp\W)

7. Detects chained SQL injection attempts
Rule:
(?:\d\s+group\s+by.+\()|(?:(?:;|#|--)\s*(?:drop|alter))|(?:(?:;|#|--)\s*(?:update|insert)\s*\w{2,})|(?:[^\w]SET\s*@\w+)|(?:(?:n?and|x?or|not|\|\||\&\&)\s+\w+[!=+]+[\s\d]*["=(])

8. Detects chained SQL injection attempts 2/2
Rule:
(?:*\/from)|(?:\+\s*\d+\s*\+\s*@)|(?:\w"\s*(?:[-
+=|@]+\s*)+[\d(])|(?:coalesce\s*\(|@@\w+\s*[^\w\s])|(?:\W!+"\w)|(?:";\s*(?:if|while|begin))|(?:"[\s\d]+=\
s*\d)

9. Detects SQL benchmark and sleep injection attempts including conditional queries
Rule:
?:(select|;)\s+(?:benchmark|if|sleep)\s?\(\s?\(?\s?\w+)

5. RESULT

In our work the attack patterns are taken from milw0rm website for 45 days duration. Data patterns are stored in a file with date every day. This information is related to commercial and educational websites. Then this data is sent which is attack pattern style to the algorithms. The input given to our system is in form of packets after extracting the url information.

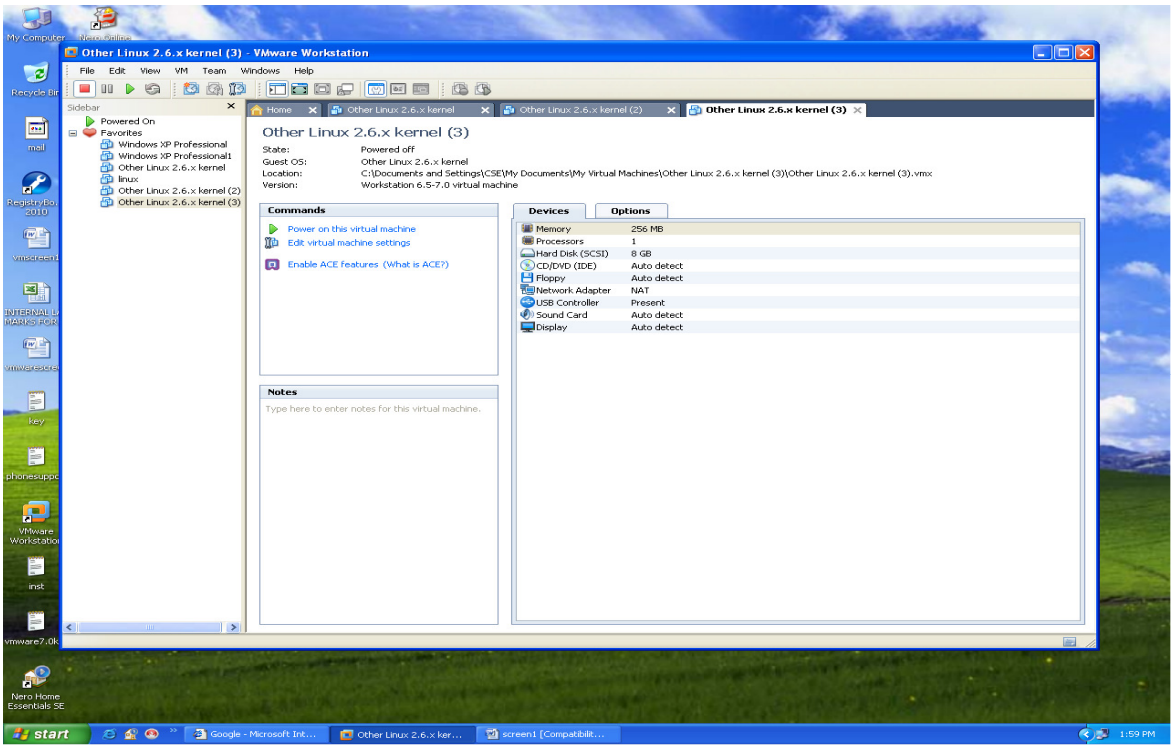

FIGURE 5: VMware with Attacker and Detection System

The result of our work is simulating the paths between two hosts and specifying the route paths. If any congestion occurs then it will split the packets and transmits it into multiple paths. Finally our system shows the result with kind of attack patterns which fails to false positives. The work is tested using VMware 7.0 as shown in Figure 5, for cloning the system with linux compatibility and coding is done in C with Perl compatibility. So we have shown our result in the form of text.

It is observed that the rules of Detects advanced XSS probings via Script(), constructors and XML namespaces,,JavaScript location/document property access, basic obfuscated JavaScript script injections, obfuscated JavaScript script injections, JavaScript cookie stealing and redirection attempts, data: URL injections and common URI schemes, possible event handlers, , possibly malicious html elements including some attributes, nullbytes and HTTP response splitting, MySQL comments, conditions and ch(a)r injections, conditional SQL injection attempts, concatenated basic SQL injection and SQLLFI attempts, code injection attempts.

6. CONCLUSION AND FUTURE WORK

Here we have developed a highly automated approach for protecting Web applications from SQL injection attacks. This application consists of 1) Using regular expression we found known attacks 2) Allowing only trusted data to form the semantically relevant parts of queries such as SQL keywords and operators. 3) Performs syntax-aware evaluation of a query string immediately before the string is sent to the database for execution. This paper also provides practical advantages over the many existing techniques whose application requires customized and complex runtime environments

To implement the functionality of query fragments that come from external sources, developer must list these sources in a configuration file that SQLIMDS processes before instrument the application. We can enhance SQLIMDS to work on web applications developed using any programming language or framework. The work can be extended by using SQLIMDS to protect actually deployed Web applications, Implementation for binary applications. for high availability on load balancing and disaster recovery. Load balancing can automatically handle failures. (bad disks, failing fans, "oops, unplugged the wrong box", ...), Make that service always work any IP addresses, Anything that has fail over or an alternate server – the IP needs to move (much faster than changing DNS).In Disaster Recovery Planning can have a status update site / weblog, Plans for getting hardware replacements, Plans for getting running temporarily on rented "dedicated servers" (ev1servers, rack space, ...)

7.REFRENCES

[1] R.Ezumalai, G. Agila, "Combinational Approach for Preventing SQL Injection Attacks," IEEE 2009-International Advance Computing Conference-2009 .

[2] S.W. Boyd and A.D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," Proc. Second Int'l Conf. Applied Cryptography and Network Security, pp. 292-302, June 2004.

[3] Sagar Joshi, " SQL Injection Attack and Defense ", white paper,2005

[4] Ke Wei, M. Muthuprasanna, Suraj Kothari, " Preventing SQL Injection Attacks in Stored Procedures ", Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)

[5] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," Proc. Int'l Symp. Software Testing and Analysis, pp. 196-206, July 2007.

[6] Xiang Fu Xin Lu Boris Peltsverger Shijun Chen , " A Static Analysis Framework For Detecting SQL Injection Vulnerabilities", 31st Annual International Computer Software and Applications Conference(COMPSAC 2007)

[7] "Top Ten Most Critical Web Application Vulnerabilities," OWASP Foundation, http://www.owasp.org/documentation/topten.html, 2005.

[8] V. Haldar, D. Chandra, and M. Franz, "Dynamic Taint Propagation for Java," Proc. 21st Ann. Computer Security Applications Conf., pp. 303-311, Dec. 2005.

[9] W. Halfond, A. Orso, and P. Manolios, "Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks," Proc. ACM SIGSOFT Symp. Foundations of Software Eng., pp. 175- 185, Nov. 2006.

[10] W.G. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks," Proc. 20th IEEE and ACM Int'l Conf. Automated Software Eng., pp. 174-183, Nov. 2005.

[11] W.G. Halfond, J. Viegas, and A. Orso, "A Classification of SQLInjection Attacks and Countermeasures," Proc. IEEE Int'l Symp. Secure Software Eng., Mar. 2006.

[12] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," Proc. 12th Ann. Network and Distributed System Security Symp., Feb. 2005.

[13] "On the stability of networks operating TCP-like congestion control," in Proc. IFAC World Congress, Barcelona, Spain, 2002.

[14] C. Anley, "Advanced SQL Injection In SQL Server Applications," white paper, Next Generation Security Software, 2002.

[15] Stuart McDonald SQL Injection: Modes of Attack, Defence, and Why It Matters , GIAC Security Essentials Certification (GSEC) Practical Assignment - Version 1.4 (amended April 8, 2002) - Option One

[16] http://nvd.nist.gov

[17] http://www.milw0rm.com

[18] http://www.securityfocus.com