

# Prediction of SQL Injection Attacks in Web Applications

**Mote Lokeswari**

PG scholar, Department of MCA, CDNR collage, Bhimavaram, Andhra Pradesh.

**A.Naga Raju**

(Assistant Professor), Master of Computer Applications, DNR collage, Bhimavaram, Andhra Pradesh.

## Abstract

In today's present time, SQL injection has become a significant security threat over the web for diverse dynamic web applications and websites. SQL Injection may be a sort of associate injection attack that produces it doable to execute malicious SQL statements into an online application consisting of SQL information. Attackers use these SQL Injection Queries or Statements specified if an Internet site or an application hosted on web contain SQL vulnerabilities to bypass application security measures. The Attacker will even go around authentication associated with authorization of an online page or Internet application and might bypass these methods and retrieve the content of the whole SQL information of an online application. The purpose of the proposed system is to predict the occurrence of a SQL injection attack on a particular server where an application is deployed from a given supply at a particular point in time. This predictive experiment is managed using the JMeter tool. From network logs, you can now pre-measure, exclude choices, analyze, and feed machine learning models to predict SQLIA.

## Introduction

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission

and/or a fee. exploited for nefarious purposes when discovered by malicious attackers. In some cases, an attacker can crash an important running program, leading to a DoS (denial of service). In other cases, the attacker can escalate his privileges or even achieve full control over the machine. Over the years, numerous countermeasures have been implemented in both compilers and operating system to minimize the damage that malicious hackers can cause via buffer overflow attacks.

For example, DEP (data execution prevention) makes the call stack non-executable, preventing hackers from being able to execute their payloads, and ASLR (address space layout randomization), randomizes the address space layout of the process, making it more difficult for hackers to insert correct addresses into their payloads [17]. However, these techniques have proven to be little more than a nuisance to determined adversaries. Thus far, the only way to prevent hackers from successfully completing an attack is to write secure code. However, complex programs, particularly those written in a relatively low-level language like C, are difficult to scan for bugs, even when using both manual and automated techniques.

Microsoft spends roughly 100 machine years per year using automated techniques to detect bugs in their code [7], but their products often contain numerous bugs, because complex pointer arithmetic can sometimes be difficult to follow, especially when the developers are under constant time pressure to meet their deadlines. Since attackers use software to uncover security holes in programs, it is important for developers and security professionals to keep up with the latest automated vulnerability detection technologies. The contribution of this paper is a methodology for analyzing features from C

source code to classify functions as vulnerable or non-vulnerable. After finding 100 programs on GitHub, we parsed out all functions from these programs. We then extracted trivial features (function length, nesting depth, string entropy, etc) and non-trivial features (n-grams and suffix trees) from these functions. The statistics for these features were arranged in a table, which was split into training data and test data. Several different classifiers, including Naive Bayes, k nearest neighbors, k means, neural network, support vector machine, decision tree, and random forest, were used to classify the test samples. The trivial features produced the best classification result, with an accuracy of 75%, while the best n-grams result was 69% and the best suffix trees result was 60%. These results are discussed in more detail in Section 5. Section 2 discusses some background concepts, Section 3 discusses previous work, Section 4 outlines the details of the testing methodology, and Section 6 contains the conclusions.

## LITERATURE SURVEY:

### Exploration of Communication Networks from the Enron Email Corpus

The Enron email corpus is appealing to researchers because it is a) a large scale email collection from b) a real organization c) over a period of 3.5 years. In this paper we contribute to the initial investigation of the Enron email dataset from a social network analytic perspective. We report on how we enhanced and refined the Enron corpus with respect to relational data and how we extracted communication networks from it. We apply various network analytic techniques in order to explore structural properties of the networks in Enron and to identify key players across time. Our initial results indicate that during the Enron crisis the network had been denser, more centralized and more connected than during normal times. Our data also suggests that during the crisis the communication among Enron's employees had been more diverse with respect to people's formal positions, and that top executives had formed a tight clique with mutual support and highly brokered interactions with the rest of organization. The insights gained with the analyses we perform and

propose are of potential further benefit for modeling the development of crisis scenarios in organizations and the investigation of indicators of failure.

### A validation of object-oriented design metrics as quality indicators

This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design metrics introduced in (Chidamber and Kemerer, 1994). More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. This study is complementary to the work described in (Li and Henry, 1993) where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

### RICH: Automatically Protecting Against Integer-Based Vulnerabilities

We present the design and implementation of RICH (Run-time Integer CHecking), a tool for efficiently detecting integer-based attacks against C programs at run time. C integer bugs, a popular avenue of attack and frequent programming error [1–15], occur when a variable value goes out of the range of the machine word used to materialize it, e.g. when assigning a large 32-bit int to a 16-bit short. We show that safe and unsafe integer operations in C can be captured by well-known sub-typing theory. The RICH compiler extension compiles C programs to object code that monitors its own execution to detect integer-based attacks. We implemented RICH as an extension to the GCC compiler and tested it on several network

servers and UNIX utilities. Despite the ubiquity of integer operations, the performance overhead of RICH is very low, averaging about 5%. RICH found two new integer bugs and caught all but one of the previously known bugs we tested. These results show that RICH is a useful and lightweight software testing tool and run-time defense mechanism. RICH may generate false positives when programmers use integer overflows deliberately, and it can miss some integer bugs because it does not model certain C features.

## Existing METHOD

In Existing system, we are using machine learning algorithms like k-means, random forest and decision tree to develop vulnerability detection tool

## Disadvantages

1. Less accuracy.

## PROPOSED METHOD

In proposed system, we are using Ensemble Machine Learning algorithm which is combination of multiple algorithms such as SVM, KNN and Naïve Bayes.

Now-a-days Machine Learning algorithms are using everywhere from Medical disease prediction to road side traffic prediction as this algorithms prediction accuracy is more than 95%.

Above success of Machine Learning algorithms are migrating us to develop vulnerability detection tool using machine learning algorithms. Machine Learning algorithms get trained on past data and then can analyse new test data to predict it class of Normal or Vulnerability type.

In propose work we are using dataset to identify 3 different classes such as 'No Vulnerability, SQL Injection, XSS or RFI.

## Advantages

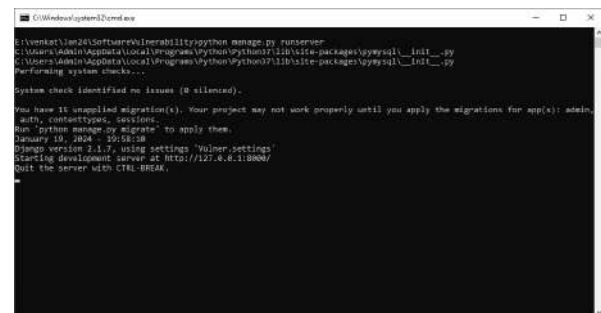
1. High Accurac

## RELUS

To run project install python 3.7 and then install MYSQL database and then copy content from DB.txt

file and paste in MYSQL to create database. Now double click on 'installNLTK.bat' file to download NLTK and once click then window will appear in that window click on "Download" button to download all packages and once downloaded then window will turn to green colour and then close the window

Now double click on 'run.bat' file to start python DJANGO web server and get below screen



```

C:\venkat\lan24\software\vulnerability>python manage.py runserver
C:\Users\Admin\AppData\Local\Programs\Python\Python37\lib\site-packages\pytz\__init__.py:
C:\Users\Admin\AppData\Local\Programs\Python\Python37\lib\site-packages\pytz\__init__.py:
Performing system checks...

System check identified no issues (0 silenced).

You have 11 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
January 19, 2024 - 10:58:18
Django version 3.1.7, using settings 'vulnerability.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

```

In above screen python web server started and now open browser and enter URL as <http://127.0.0.1:8000/index.html> and then press enter key to get below page



In above screen click on 'New User Register Here' link to get below sign up page



In above screen user is entering sign up details and then press button to get below page



In above screen user sign up completed and now click on 'User Login' link to get below page



In above screen user is login and after login will get below page



In above screen click on 'Load Dataset' link to get below page



In above screen select and upload 'dataset\_vulner.csv' file and then click on 'Open' and 'Submit' button to load dataset and then will get below output



In above screen can see dataset loaded and can see total number of records available in dataset and then can see training number of records on which Machine Learning algorithm get trained and then can see number of test records on which ML will perform prediction to calculate its prediction accuracy %. Now click on 'Run Ensemble Algorithms' link to train ensemble algorithm and then will get below output

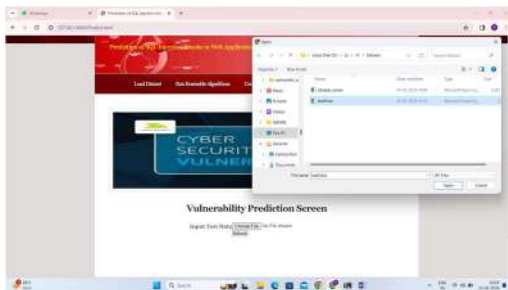


In above screen Ensemble Machine Learning algorithm training completed and can see its prediction accuracy as 95% and can see other metrics like precision, recall and FCSORE. Now click on 'Confusion Matrix Graph' link to view visually how many records ensemble predicted correctly and incorrectly





In above graph x-axis represents Predicted Labels and y-axis represents True Labels and then all different colour boxes in diagonal represents correct prediction count and remaining all blue boxes represents incorrect prediction count which are very few. Now click on ‘Predict Vulnerability’ link to upload test data and predict Vulnerability



In above screen selecting and uploading ‘testData.csv’ file which contains SQL, XSS and RFI coding commands and then click on ‘Submit’ button to get below output



Test Data	Predicted
SQL Injection: ' OR '1'='1	SQL Injection
XSS: <script>alert('XSS')</script>	XSS
RFI: http://www.example.com/../../../../etc/passwd	RFI

In above table in first column can see SQL queries, XSS and RFI coding commands and in second column can see predicted vulnerability.

So by using above tool you can easily detect all vulnerability and you can add NEW test command in ‘testData.csv’ file which is available inside ‘Dataset’ folder

## CONCLUSION:

After extensively testing function vulnerability classification using trivial features, n-grams, and suffix trees, we can draw several conclusions. First of all, we see that extracting numerous n-grams does not, thus far, seem to give good classification results, especially if we consider the 74% accuracy that we got from “character diversity “to be a baseline requirement. We also noticed that even when combinations of n-grams were manually selected (in a manner that would normally be illegal and lead to overfitting), the overall result did not improve. However, this study is a good proof-of-concept of a very important point: trivial features can tell us a lot about whether a function is vulnerable or not. There are a few directions in which this research can be taken to improve the results further. First of all, it might be possible to think of additional trivial features to investigate. Secondly, it might also make sense to test some other n-gram selection techniques, as well as some of the SciKit library’s other classification parameters (other than default settings). Third, it would be interesting to look more closely at which characters (or strings) are most important, since this would give us better insight than just “character diversity”. One way to do this would be to run the same character diversity tests after eliminating different strings (square brackets, curly brackets, ++, etc) in pre-processing. Finally, it is possible to test whether the techniques presented in this paper can be used to efficiently detect vulnerabilities in other programming languages (in addition to C).

## REFERENCES:

- [1] Enron email dataset. <https://www.cs.cmu.edu/~enron/>. Accessed: 2017-07-01.
- [2] National vulnerability database. <https://nvd.nist.gov>. Accessed: 2017-07-01.
- [3] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on software engineering, 22(10):751–761, 1996.

- [4] D. Brumley, T.-c. Chiueh, R. Johnson, H. Lin, and D. Song. Rich: Automatically protecting against integer-based vulnerabilities. Department of Electrical and Computing Engineering, page 28, 2007.
- [5] N. Dor, M. Rodeh, and M. Sagiv. Csvg: Towards a realistic tool for statically detecting all buffer overflows in c. In ACM Sigplan Notices, volume 38, pages 155–167. ACM, 2003.
- [6] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. IEEE software, 19(1):42–51, 2002.
- [7] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. Queue, 10(1):20, 2012.
- [8] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In USENIX Security Symposium, pages 49–64, 2013.
- [9] A. E. Hassan. Predicting faults using the complexity of code changes. In Proceedings of the 31st International Conference on Software Engineering, pages 78–88. IEEE Computer Society, 2009.
- [10] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In Proceedings of the 29th international conference on Software Engineering, pages 489–498. IEEE Computer Society, 2007.
- [11] D. Larochelle, D. Evans, et al. Statically detecting likely buffer overflow vulnerabilities. In USENIX Security Symposium, volume 32. Washington DC, 2001.
- [12] P. Lathar, R. Shah, and K. Srinivasa. Stacy-static code analysis for enhanced vulnerability detection. Cogent Engineering, 4(1):1335470, 2017.
- [13] R. Ma, Y. Yan, L. Wang, C. Hu, and J. Xue. Static buffer overflow detection for c/c++ source code based on abstract syntax tree. Journal of Residuals Science & Technology, 13(6), 2016.
- [14] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In Proceedings of the 30th international conference on Software engineering, pages 181–190. ACM, 2008.
- [15] R. M. Pampapathi, B. G. Mirkin, and M. Levene. A suffix tree approach to antisipam email filtering. Machine Learning, 65(1):309–338, 2006.
- [16] E. Penttilä et al. Improving c++ software quality with static code analysis. N/A, 2014.
- [17] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.
- [18] Y. Shin and L. Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, pages 315–317. ACM, 2008.
- [19] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference, pages 257–267. IEEE, 2000.
- [20] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In NDSS, pages 2000–02, 2000.