SQL Injection Attack Prediction in Web Applications Using SVM, KNN, and Naïve Bayes

Dr. Vishnu Kumar Misra [1], G. Sushmitha [2] , J. Sharon David [3] , K. Harani [4]

[1] Professor, Department of CSE, Malla Reddy Engineering College for Women, Autonomous, Hyderabad

[2],[3],[4]Student, Department of CSE, Malla Reddy Engineering College for Women, Autonomous, Hyderabad

ABSTRACT:

Various dynamic web apps and websites are now vulnerable to SQL injection, a major online security risk. One kind of injection attack is SQL Injection, which allows malicious SQL queries to be executed into a digital application that contains SQL information. In order to circumvent application security measures, attackers utilize SQL injection requests or statements that are supplied when a website or online application has SQL vulnerabilities. An attacker may even find ways to circumvent authentication processes linked with authorizing a web page or Internet application, allowing them to access all of the SQL data stored on that platform. The suggested system's goal is to foretell when an SQL injection scheme would hit a certain server, assuming that an application is installed from a specific source at a specific moment. I am using the JMeter application to handle this prediction experiment. The ability to pre-measure, eliminate options, evaluate, and feed deep learning models using network logs to forecast SQLIA is now at your fingertips.

INTRODUCTION

already operating. Sometimes the hacker might even get complete control of the system by escalating his privileges. To lessen the impact that hostile hackers may have with buffer overflow attacks, several safeguards have been included into compilers and operating systems over the years. Consider two common security measures: data execution prevention (DEP) and address space layout randomization (ASLR). DEP renders the call stack non-executable, thereby preventing hackers from executing their payloads. ASLR further makes it more difficult for hackers to get the correct addresses within their payloads by randomly arranging the process's address space. On the other hand, persistent enemies have shown that these tactics are useless. As of right now, writing secure code is your one option for keeping hackers at bay. But even with automatic and manual methods, it is difficult to scan complicated programs for defects, especially those written in a low-level language like C. Even though Microsoft invests around 100 machine years annually into automated bug detection techniques, their products frequently have multiple bugs due to the complexity of pointer arithmetic and the developers' relentless focus on meeting deadlines. Keeping up with the newest automated vulnerability detection technologies is crucial for developers and security experts, since attackers use software to find program security gaps. A approach for assessing C source code attributes to identify functions as susceptible or non-vulnerable is the contribution of this paper. We extracted all functions from 100 applications that we found on GitHub. From these functions, we retrieved both simple and non-trivial properties, such as function length of sentence, nesting depth, string entropy, and suffix trees. A table containing the feature statistics was organized, with the data being divided into training and test sets. The test samples were classified using a variety of classifiers, such as Naive Bayes, k-nearest neighbors, k-means, neural network, assistance vector machine, decision tree, and random forest. The most effective classification result was 75% for the trivial features, 69% for the n-grams, and 60% for the suffix trees. In Section 5, we go into further depth about these findings. Section 2 covers some introductory ideas, Section 3 reviews the relevant literature, Section 4 describes the testing procedures in depth, and Section 6 offers the final thoughts.

RELATED WORK

Exploration of Communication Networks from the Enron Email Corpus

Researchers are interested in the Enron emails corpus for three reasons: (a) it is a massive collection of emails from a genuine company; (b) it spans three and a half years; and (c) it is a large-scale collection. Our study in this article adds to the preliminary social network analytics examination of the Houston email dataset. In this paper, we detail our efforts to improve the Enron corpus by adding relational data and removing communication networks. In order to discover important participants across time and investigate the structural features of Enron's networks, we use a number of network analytic methods. The network was denser, more centralized, and more linked throughout the Enron crisis than it is during normal times, according to our early data. According to our data, there was greater cross-functional contact among Enron workers throughout the crisis, regardless of employees' official roles. However, the top executives remained close-knit, supported each other, and interacted regularly the rest of the business via extensive brokering. Organizational crisis scenario modeling and failure indicator research may both benefit from the insights obtained via the studies we conduct and suggest.

The reliability of metrics for object-oriented design as measures of quality

We conducted an empirical investigation of the set of object-oriented (OO) design metrics proposed in (Chidamber and Kemerer, 1994) and reported our findings in this article. Our main objective is to validate these measures for their ability to identify classes prone to errors and, by extension, for their potential utility as early quality gauges. This research supplements that of Li and Henry (1993), who also utilized the same set of criteria to evaluate the frequency of class maintenance modifications. For the purpose of our validation, we gathered information on the creation of eight information management systems for medium-sized businesses that met the same criteria. The eight projects were created in C++ using the sequential developmental model, a popular object-oriented analysis and design paradigm. Here we examine the benefits and downsides of certain OO measures using data from empirical and quantitative studies. It seems that certain of the OO metrics proposed by Chidamber and Kemerer may be used to foretell which classes will be more prone to errors in their early stages of development. Not only that, but they outperform "traditional" code metrics—which are only

obtainable later in the software development lifecycle—as predictors on our dataset.

RICH: Secure By Design Against Integer-Based Attacks

In this paper, we detail the architecture and implementation of RICH, an efficient tool for identifying integer-based attacks on C programs during runtime. When a variable's value exceeds the tolerance of the machine word that was utilized to materialize it, for as when assigning a huge 32-bit int to a 16-bit short, a common programming mistake known as a C integer bug occurs [1–15]. We prove that the well-known sub-typing theory captures both safe and hazardous integer operations in C. To protect against integer-based assaults, the RICH compiler extension converts C programs to object programs that executes self-monitoring. After integrating RICH into the GCC compiler, we ran tests on several servers in the network and UNIX utilities. The performance expense of RICH is quite modest, averaging approximately 5%, even though integer operations are ubiquitous. In addition to catching nearly every one of the known issues, RICH discovered two additional integer bugs. Based on these findings, RICH is an effective and lightweight tool for testing software and a defense mechanism for runtime. Due to its lack of modeling of some C features, RICH has the potential to overlook certain integer problems and produce error messages when programmers intentionally employ integer overflows.

Achieving a Practical Method for Statically Identifying All C Buffer Overflows using CSSV

One common cause of software faults in C programs is incorrect string manipulation, which may lead to vulnerabilities that viruses can exploit. Introducing C String Static Verifier (CSSV), a program that can statically detect and fix any issue related to string manipulation. As a cautious tool, it discloses all such mistakes, even if it sometimes triggers false alarms. The tiny number of reported false alarms demonstrates that software vulnerability may be significantly reduced, which is a relief. By dissecting each operation independently, CSSV is able to manage big applications. In order to achieve this goal, the technology permits procedural contracts that are confirmed. To ensure that the actual EADS Airbus code was error-free, we built a CSSV replica and tested it extensively. Applying CSSV to another popular string-intensive app revealed actual issues with few false positives. Enhancing safety with lightweight, extendable static analysis

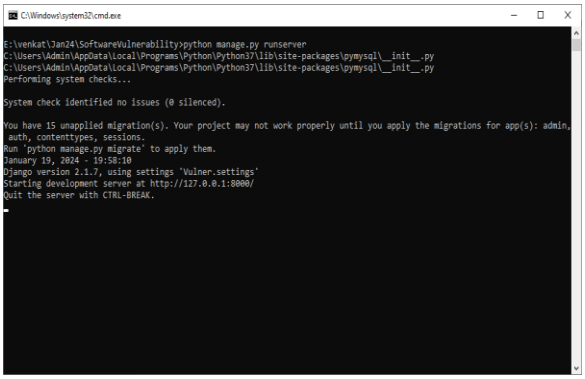
Common types of implementation problems

are often the target of security attacks. Although developers have the ability to identify and fix many of these issues prior to software deployment, these problems occur with alarming regularity. This is not due to a lack of understanding within the security community, but rather to the fact that methods for avoiding them have not been incorporated into software development. In order to identify typical security flaws, such as format string vulnerabilities and buffer overflows, this paper details an extendable tool that use lightweight static analysis.

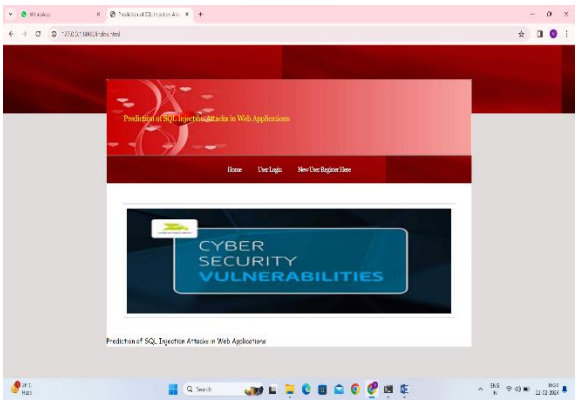
METHODOLOGY

1) The first step is for new users to register with the app.
2) After signing up, users may access the application using the login page.
3) Load Dataset: Once the user logs in, they can upload a dataset to the program. From there, they can extract all the labels and queries. After that, they may remove stop words include "and," "or," and "what" from all the searches. The application will contain core query terms when stop words are removed. By using the Natural Language Processing Toolkit, the core word dataset will be processed.
4) Execute Ensemble Algorithms: The processed dataset will be fed into the Ensemble machine learning technique for training a model. Then, to determine accuracy and other metrics, this model's predictions will be applied to test data.
5) The fifth module is the confusion matrix graph, which will show the algorithm's prediction capabilities.
6) Vulnerability Prediction: This module allows users to contribute new TEST data queries, which are subsequently analyzed by a machine learning system to determine the kind of vulnerability.

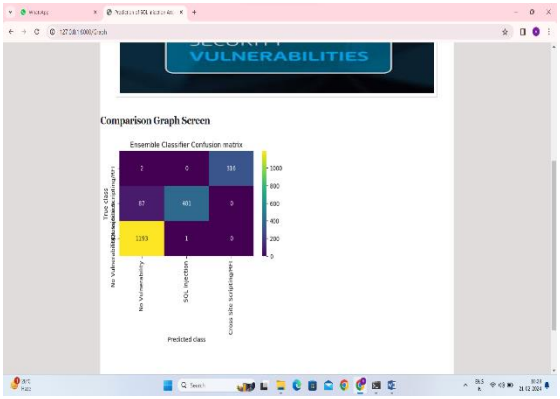
RESULT AND DISCUSSION



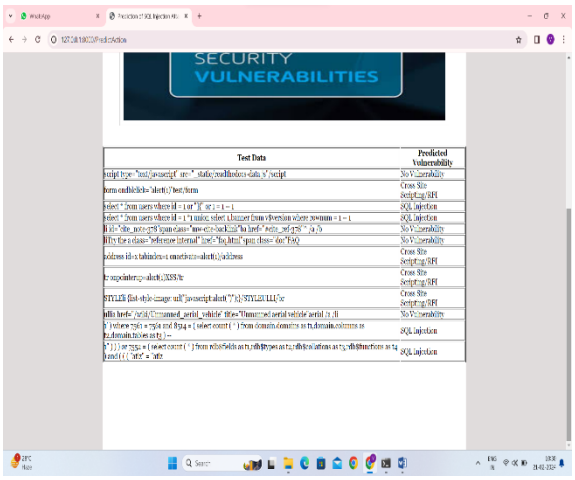
In above result python web server started and now open browser and enter URL as http://127.0.0.1:8000/index.html and then press enter key to get below page



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



In above graph x-axis represents Predicted Labels and y-axis represents True Labels and then all different colour boxes in diagnol 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 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

Several conclusions may be drawn from the thorough testing of function vulnerability categorization using n-grams, suffix trees, and trivial characteristics. First, using the 74% accuracy we achieved from "character diversity "as a baseline criterion, it is clear that extracting multiple n-grams does not seem to provide strong classification results at this time. We also found that the overall outcome remained unchanged even when n-gram combinations were manually picked (in a way that would often be considered

unlawful and result in overfitting). This research does, however, provide solid proof-of-concept for a crucial point: seemingly insignificant traits may reveal a great deal about a function's vulnerability. To make the outcomes even better, this study may be done in a few different ways. To start, it may be easy to come up with more insignificant qualities to look into. The second thing to consider is trying out other n-gram selection methods and other classification parameters (beyond the defaults) in the SciKit package. Third, although "character diversity" is intriguing, it would be much more illuminating to focus on the most crucial characters (or strings). One approach would be to do the character variety tests again after pre-processing removes certain strings, such as square brackets, curly brackets, ++, etc. Lastly, it is feasible to evaluate whether the methods discussed in this article may effectively identify security flaws in languages other than 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. Cssv: 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 antispam 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 overruns vulnerabilities. In NDSS, pages 2000–02, 2000.