Making codes more secure and less vulnerable*

Using GITHUB

Balaji Mohan
Computer Science
University of Texas at Arlington
USA
balaji.paiyurmohan@mavs.uta.edu

ABSTRACT

Many of the programmers when developing their codes tend to ignore implementing security mechanisms in their code which leads to several vulnerabilities. Many of these vulnerabilities form attack surfaces for attackers to gain unauthorized access to their code. So, it is very important to study the most frequently occurring security vulnerabilities and how to avoid them. It is also important to understand why these security vulnerabilities occur and how to encourage secure coding practices among software developers so that such mistakes will be reduced in the future.

In this project, we try to analyze several top-rated projects on GitHub to find details such as number of followers, stargazers count, number of forks, open issues and GitHub score which a metric score given to a project by GitHub. We rely on static analysis of codes to give us the idea of all vulnerabilities present in the code. We try to find if there is any correlation between these features with the presence of absence of a vulnerability in the code. We also try to understand if factors such as experience level of a programmer or developer, exposure to multiple languages and some kind of knowledge regrading secure coding practices affect the style of coding in such a way that vulnerabilities are reduced in their code. We initially start by concentrating mainly on the OWASP Top 10 Security risks to perform our analysis but later move on to more frequently occurring vulnerabilities.

KEYWORDS
Vulnerabilities, OWASP, GitHub

1 INTRODUCTION

Secure coding practices are something that has increasingly been encouraged for programmers and developers to inherit in their codes and coding styles. The reason for this is the increasing number of vulnerabilities being explored in the wild and more proof surfacing via proof of concept and exploits done by malicious parties that most of the vulnerabilities are exploitable. The presence of vulnerabilities in a code or software opens up an attack surface in the product which can be used to exploit the product by malicious users. So, it has become an increasingly important research are to try and understand how and why vulnerabilities are committed in the first place and are there ways to mitigate them or to reduce them by performing analysis on raw features of the code and the programmer.

The examination of diverse software characteristics in relation to security has been a constant topic of interest to the research community. In this paper, we try to analyze the top projects on GitHub for security vulnerabilities and try to form correlation between factors such as the authors experience, knowledge of security concerns or lack of exposure to security concerns and how frequent the author maybe committing the same security vulnerabilities.

"Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency" [1]. It is used to keep track of modifications or updates made to a file and also for sharing among developers. "GitHub is a web-based hosting service for version control using Git which is mostly used for computer code" [2]. GitHub adds its own features along with all the features of Git and its open source making it one of the world's leading software development platform. Due to its efficiency and ease of use, it has gained a large user base especially among aspiring developers as a platform to get their work published for other users to see and also is a major player in the corporate industries where it is used by the programmers to perform coordinated work amongst themselves on the same project updating and incorporating useful ideas. At the university level, it is a great source for students for referencing software projects and their code solutions.

GitHub has become a great source for finding codes from any programming language. It is fair to say that any type and every type of code can be found on GitHub and this was one of the main reasons we choose to perform our analysis based on GitHub as a source.

Along our way, we came across LGTM which is security analysis tool free for open source projects powered by Semmle and uses Semmle QL to perform queries. It performs security analysis of codes for projects that are integrated with it. It is not language specific which makes it really useful for perform analysis on codes of any programming language. To date, they have managed to disclose 57 CVEs and have 135,821 open source repositories on their site [3]. We choose LGTM has our analysis tool due to the fact that the analysis is automatically performed by LGTM which helps us focus more on our research findings rather than performing static analysis for each project. GitHub also has the LGTM app in its marketplace where programmers can integrate their code with LGTM and it will automatically perform security analysis for every commit and notify the user about any vulnerabilities and how to remove them if there is a way to. The fact that LGTM is endorsed by GitHub itself makes it a reliable tool for us to base our analysis on.

In our research, we initially obtain projects from GitHub using a scrapper along with features such as number of followers, number of forks, GitHub score etc. We then find these projects on LGTM and obtain the result of the static analysis done by the tool. We extract important features namely type of vulnerabilities and total number of distinct vulnerabilities in the project. Using these features, we try to perform statistical analysis to find correlation between different features. We mainly focus on trying to answer the following research questions:

RQ1 - Is there a relation between the experience level of programmer and the vulnerabilities committed by them? The main idea behind this is to find out if concern for secure codes increases as programmers gain more experience with their coding practices.

RQ2 - Does exposure to some sort of security related courses help developers avoid making basic security vulnerabilities? We tried to figure out if developers who had taken security related courses such as Secure Programming are less likely to commit a security vulnerability that someone who has had no such formal lessons in that domain.

RQ3 – How frequently does vulnerabilities among the OWASP Top 10 list occur? We ask this question because: "The OWASP Top 10 is a powerful awareness document for web application security. It represents a broad consensus about the most critical security risks to web applications. Project members include a variety of security experts from around the world who have shared their expertise to produce the list" [4]. We wanted to find out if indeed these vulnerabilities are being made by the programmers on a frequent basis or not.

RQ4 - What are the most frequently occurring security vulnerabilities and can they be avoided? Some of the common vulnerabilities can be avoided but at some cost to performance of the code. We wanted to explore if such avoidable codes are being committed just because the programmer wants to meet the pragmatic goal of making the code to run.

RQ5 - What are the most critical vulnerabilities occurring in the codes? Which vulnerabilities being committed can seriously compromise the program altogether?

RQ6 – Can we build a classifier to detect if a code is vulnerable or not just using its raw features instead of performing the actual analysis on the code? (This is a very hypothetical solution to the vulnerability problem, but it is worth exploring)

The paper mainly focuses on answering these research question. The first two questions are answered based on assumption as of now and requires actual programmers' response to perform a solid analysis. We find answers to questions 3-5 using the data collected from our sources. We build a model to answer question 6.

With our work, we were able to conclude that:

- The most popular codes on GitHub did not pass a score of 50% in terms of secure coding and vulnerabilities, which shows that even experienced coders lack secure practices.
- Most popular authors do have considerable amount of vulnerabilities in their code but were not useful in predicting how misleading the codes were which has been starred and forked by thousands of followers.
- The most commonly occurring vulnerabilities were:
    1. Cross-Site Scripting
    2. Cross-Site Request Forgery
    3. SQL Injection
- Some vulnerabilities such as buffer overflow and integer overflow can be avoided using dynamic allocation, but this will lead to cache miss which might affect performance of the code, but it is tradeoff that must be considered.

The remainder of the paper constitutes of the related work being done, describe our data collection method along with its challenges, what methods we used to evaluate on the data obtained, the results after performing our evaluation, the limitations to our research work, how we plan to extend our work beyond the contents of this paper and a discussion section pointing out practical need for secure coding practices and secure codes cannot be guaranteed but can be achieved with suitable safeguards.

2 RELATED WORK

A lot of work has been put into understanding various aspects of vulnerabilities in code, what is the root cause, how do they impact the system, are certain vulnerabilities severe than others and so on. Much of the prior work has concentrated on specific vulnerabilities such as misuse of secure socket layers (SSL) and cryptographic keys. Lazar et al. showed that 83% of the cryptographic vulnerabilities analyzed by them were caused due to misuse of cryptographic API [5]. Nadi et al. analyzed how the misuse of JAVA cryptographic APIs caused problems [6]. Georgiev et al. demonstrated how misuse of SSL APIs leads to man-in-the-middle attack in android applications [7].

Apart from this, there have been programming language specific research in terms of vulnerabilities. Rahaman and Yao introduced an analysis scheme called CPA (Cryptographic Program Analysis) [8] which uses taint analysis on C/C++ codes to detect cryptographic errors. Na Meng et al. analyzed several posts on Stack Overflow relating to JAVA and how use of its third-party libraries introduces vulnerabilities in the code [9].

Anne Edmundson et al. conducted a study where they asked developers to perform a manual code review and prepare a vulnerability report. They found out less than twenty percent of the participants found no vulnerabilities and no developer was able to find all the seven vulnerabilities in the code [10]. The closest work related to our research is the work done by Baca et al. who performed an industry experiment on the assumption that all users can identify the warning by a static analysis tool as a security threat. They were able to show that only users with specific experiences had a better chance at detecting the security vulnerability than an average developer. They were also able to show that combing static analysis tool experience along with security experience increases the number of correct answers by three-fold [11].

3 METHODOLOGY

We used two online platforms as our source. We used an existing Machine Learning algorithm to build our binary classifier that will determine if a code is vulnerable or not based only on raw features without performing any static analysis on the code.

a) Phase I - GitHub

For retrieving information from GitHub, we build a scrapper that will scrap the GitHub webpage and retrieve all the features required such as name of the project, number of forks, stargazers count, number of followers, open issues and GitHub scores. We used the scrappy framework provided by Python to design our scrapper. It is nearly impossible to run automated code on GitHub website due to restrictions on the website. So, to overcome this, the GitHub developer community introduced the clone version of stand-alone GitHub restful API which allows the developers to utilize the web API for research purpose. We used the PyGitHub framework to pass through the two-stage authentication (OAuth).

We then use our scrapper to scrape the data from this developers API. We search the repositories in decreasing order of number of stars (the idea behind this is that most popular projects will have more stars which implies projects are assumed to be reliable) and scrap all the information needed and store it in a JSON file. Then, we retrieve all the required fields and generate a CSV file which will be fed to a JAVA program.

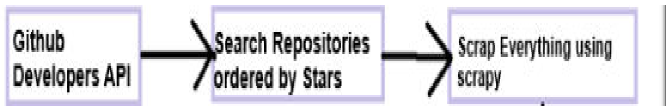

Github Developers API
Search Repositories ordered by Stars
Scrap Everything using scrapy

Figure 1: Phase I is takes care of extracting all required features from GitHub and is carried out entirely in Python

b) Phase II - LGTM

Once we get the CSV file containing the features, we extract the project name and search for it in LGTM. Due to some policy restrictions, there was no way to manually download contents from LGTM. So, we had to manually check for presence or absence of the project in LGTM and manually download all the alerts.

The downloaded file is in .sarif (Static Analysis Results Exchange Format) format which “defines a standard format for the output of static analysis tool” [12]. We use the JAVA code to extract the useful features mainly, the names of different vulnerabilities in the project and the number of vulnerabilities in the project. Once we extract these features, we combine these features with the GitHub features to form our final dataset.

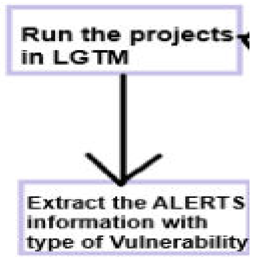


Figure 2: This phase of the project deals with extracting LGTM features and is carried out using JAVA

Once we complete Phase II, we have our dataset required to perform our analysis. We try to find correlation between various features and try to find which raw features affect the presence of vulnerabilities in a code/project. We also analyze the data to observe the frequency with which OWASP Top 10 vulnerabilities occur along with the most commonly occurring vulnerabilities in our dataset.

c) A Classifier

We then proceed towards building a classifier which aims to classify codes as either codes with vulnerabilities or codes without vulnerabilities. We use all the features obtained from GitHub such as experience level (obtained by calculating difference between date of first commit for a project and the date on which we perform the analysis; though is not a good assumption, we believe it will give us a general idea about experience level of the programmer), number of lines in code, number of followers, number of forks and GitHub score. We then sample projects obtained from our GitHub scrapper to form our ground truth for not vulnerable class. We assume that projects that are not present in LGTM, do not contain any vulnerability (this again is a weak assumption; but we do not have sufficient data to obtain a good ground truth for codes without any vulnerabilities). We manually obtain data for a certain number of projects from LGTM and use this for our ground truth for the vulnerable class. We make sure we choose equal number of projects for each class i.e. vulnerable or not vulnerable. We use the remaining projects returned from scrapper to test our classifier and evaluate its accuracy to correctly classify a code into the corresponding class.

We use an in-built library provided by Python to design our classifier and test it using various features using varying combinations.

4 DATA COLLECTION

Using our initial approach, we could not scrap any data from GitHub's website. We made use of the GitHub's Restful API for obtaining our data. We modelled our scrapper as per the requirements needed and ran it iteratively to obtain data. We started of with 3 keywords (in our case, the keywords were the programming languages we were interested in i.e. Java, JavaScript, Python) based on which the scrapper would search for the contents i.e. projects and repositories. We were not able to get enough information, so we expanded the number of keywords to 10 to incorporate different programming languages.

On a single run, the scrapper was not able to return more than 30 projects which was a very low number. We decided to run the scrapper iteratively and store the contents in a CSV file. After a few runs, we were able to get a total of 850 projects. But, most of the projects were returned multiple times by different iterative runs. On removing the duplicates, we were able to obtain a total of 316 distinct projects along with all their GitHub features. We then queried these projects on LGTM (which was done manually) and were able to find only 38 projects on LGTM which was <15% of our total projects. The amount of data was significantly small to perform any kind of analysis.

So, we came up with an alternative approach to obtain more data. We manually obtained data from LGTM for more than 150 projects. Extracted the required features namely number and names of vulnerabilities. We stored this in a CSV file along with the project name. We then used the project names to iteratively scrap GitHub using our scrapper (so, instead of scrapping for different projects based on keywords, we made it scrap for a particular project using the project/repository URL) and were able to extract features for 100 such projects.

For performing our numerical analysis for answering RQ3 and RQ4, we were able to collect a total of 138 projects. These projects contained 158 unique vulnerabilities ranging in values for severity. These 158 vulnerabilities accounted for a total of 4411 vulnerabilities present in the 138 projects for which we collected the data which included the result of static analysis performed on the codes.

The main limitation to our data collection was the fact that we could not automate our process for collecting data from LGTM due to lack of resources and restrictions imposed by the tool. We had to perform this process manually and took us a lot of time to perform this step. Also, it took us some while to figure out about the GitHub's Restful API and to configure our scrapper accordingly.

5 EVALUATION and RESULTS

- **RQ1**

For answering our first RQ, we assumed that the difference between the current date and date of creation for the project as the experience value for the programmer. We then compared it against the number of vulnerabilities being committed users with varying level of experience. We were not able to obtain any reportable results for this question and is something we would like to extend our research on.

- **RQ2**

We were not able to get any useful information regarding the exposure to security related courses for each programmer. We leave this as future work where we would be conducting programmer surveys and asking them about their experience level and exposure to security related topics and practices.

- **RQ3**

To answer our question, we first collected the OWASP Top 10 security risky published for the year 2019 which were:

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities
5. Broken Access Control
6. Security misconfigurations
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using components with known vulnerabilities
10. Insufficient logging and monitoring [13]

We observed that:

I. Path-injection (1 on the above list) occurred a total of 4 times.
II. Database resource leak (3 on list) occurred a total of 13 times.
III. Mismatch access (5 on list) occurred a total of 8 times.
IV. We observed XSS 2 times.
V. Insecure vulnerabilities occurred 4 times.

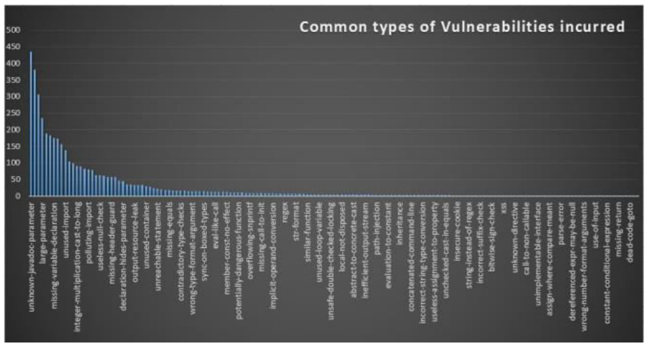
We did not observe any other vulnerabilities on the list.

- **RQ4**

For answering this question, we created a CSV file with vulnerabilities and total occurrence. Sorted the vulnerabilities according to decreasing number in total occurrence. We observed that some of the entries were just warnings without any major ramifications and hence we ignored those entries while populating this list. We have listed the top 10 vulnerabilities (which we considered severe) according to the frequency among the vulnerabilities encountered:

i. Non-sync Override
ii. Input Resource Leak
iii. Inconsistent Hashcode
iv. Implicit assignment
v. Index out of bound
vi. Output Resource Leak
vii. XXE
viii. Database Resource Leak
ix. Dangerous Function Use
x. Stack Trace Exposure

Figure 3: Shows all the vulnerabilities and their frequency.



- **RQ5**

Resource leak (190 total; 156 – Input and 34 – Output) and Hashcode (57 total) violation were the only vulnerabilities that we found which had very high severity and occurred multiple times.

- **RQ6**

We used 56 records as our training data set and 461 records for testing as there was limited information available in the GitHub API. We modelled a logistic regression classifier that would take as input raw features of a program i.e. the GitHub features extracted and predict what kind of vulnerability the program could probably have based on its popularity (forks, watchings, subscribers).

We were able to achieve an accuracy of 10.909 % based on unsupervised learning which can be further put into research on LGTM analyzer.

6 DISCUSSION

There are more than 10000 new vulnerabilities being disclosed every year. These vulnerabilities mainly occur due to lack of knowledge about security mechanisms to be implemented while developing a software or writing a program. Most of the common vulnerabilities have been known to be exploitable and can be used to compromise the code by malicious attackers.

One approach towards avoiding vulnerabilities is Secure Coding. “Secure coding is the practice of developing computer software in a way that guards against the accidental introduction of security vulnerabilities” [14]. Security professionals managed to identify the common errors that lead to most of the vulnerabilities. They were able to show that by education users of secure coding methods, they can significantly reduce vulnerabilities. Let us have a look at some of the common errors and alternative coding methods that can help prevent those errors leading to vulnerabilities.

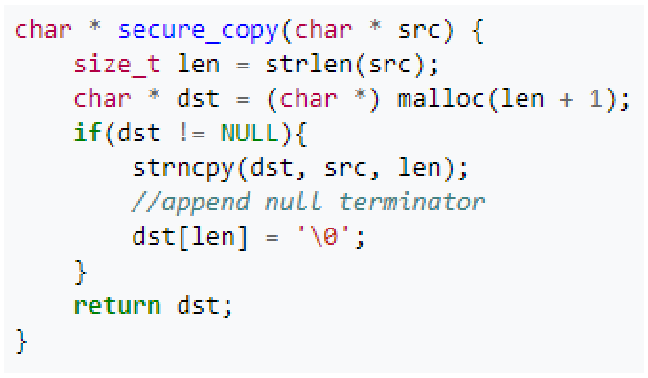
- BUFFER OVERFLOW

Can occur “when more data is put into a fixed length buffer and this extra information can corrupt or overwrite held in the overflown space” [15].

```
int vulnerable_function(char * large_user_input) {
    char dst[SMALL];
    strcpy(dst, large_user_input);
}
```

Figure 4: A piece of C code prone to buffer overflow [14]

There are different ways to avoid this: using strncpy instead of strcpy or using dynamic allocation using malloc. Below figure shows how this can be avoided using dynamic memory allocation.

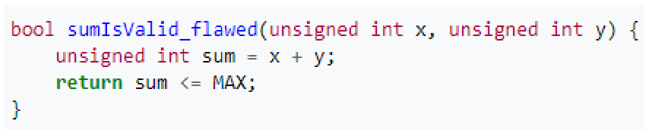


```
char * secure_copy(char * src) {
    size_t len = strlen(src);
    char * dst = (char *) malloc(len + 1);
    if(dst != NULL){
        strncpy(dst, src, len);
        //append null terminator
        dst[len] = '\0';
    }
    return dst;
}
```

Figure 5: this makes sure only required memory was allocated [14].

- INTEGER OVERFLOW

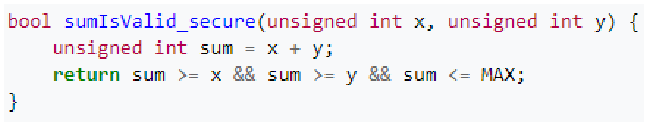
"An integer overflow is condition that occurs when result of an arithmetic operation exceed maximum size of integer type used to store it" [16].



```
bool sumIsValid_flawed(unsigned int x, unsigned int y) {
    unsigned int sum = x + y;
    return sum <= MAX;
}
```

Figure 6: This is a C++ code. It does not check for possible integer overflow.

This can be resolved by code shown in Figure 6.



```
bool sumIsValid_secure(unsigned int x, unsigned int y) {
    unsigned int sum = x + y;
    return sum >= x && sum >= y && sum <= MAX;
}
```

Figure 7: This code makes sure it checks for any possibility of integer overflow.

From the above cases, it can be understood that major vulnerabilities are caused by small errors occurring in codes. By making programmers aware of secure coding and security alternatives these errors can be avoided.

We should start encouraging programmers to start using security mechanisms and follow secure coding practices to reduce the risk of causing severe vulnerabilities in their code.

7 CONCLUSION

There were a lot of limitations while performing our study which restricted the findings to a minimal quantity.

We perform 10 epochs on different set of repositories and we were able to achieve a maximum of 10.9% accuracy in finding the vulnerabilities based on GitHub repository features. We were able to conclude that 90% of the codes in the collected repositories contained unknown java doc parameter as one of the most prevalent vulnerability. We were able to identify 158 kinds of vulnerabilities across 137 scrapped projects. We were also able to find that OWASP Top 10 vulnerabilities do occur on a fairly basis across projects.

8 FUTURE WORK

There is a lot of potential in this research area. We would like to perform solid base surveys to find answers to RQ1 and RQ2. Also, we want to explore how we can improve the performance of our classifier. We believe building a good classifier for finding out if a code is vulnerable or not will help a lot of programmers tackle vulnerability issues in their code.

REFERENCES

[1] https://git-scm.com/

[2] https://en.wikipedia.org/wiki/GitHub

[3] https://lgtm.com/

[4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[5] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail? A case study and open problems. In Proceedings of 5th Asia-Pacific Workshop on Systems, APSys '14, pages 7:1–7:7, New York, NY, USA, 2014. ACM.

[6] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In Proceedings of the 38th International Conference on Software Engineering, ICSE, pages 935–946, New York, NY, USA, 2016. ACM

[7] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In Proceedings of the ACM Conference on Computer and Communications Security, CCS, pages 38–49, New York, NY, USA, 2012. ACM

[8] S. Rahaman and D. Yao. Program analysis of cryptographic implementations for security. In IEEE Security Development Conference (SecDev), pages 61–68, 2017

[9] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang and Gustavo Arango Argoty. Secure Coding Practices in Java: Challenges and Vulnerabilities. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 2018. ACM/IEEE

[10] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler and David Wagner. An Empirical study on the Effectiveness of Security Code Review.

[11] Dejan Baca, Kai Petersen, Bengt Carlsson and Lars Lundberg. Static code analysis to detect software vulnerabilities-does experience matter? In 2009 International Conference on Availability, Reliability and Security, pages 804-810

[12] https://sarifweb.azurewebsites.net/

[13] https://blog.sucuri.net/2019/01/owasp-top-10-security-risks-part-iv.html

[14] https://en.wikipedia.org/wiki/Secure_coding

[15] https://www.veracode.com/security/buffer-overflow