Towards Finding Mobile Code Snippets on a Question and Answer Website Causing Mobile App Vulnerabilities

Hiroki NAKANO††, Fumihiro KANEI††, Yuta TAKATA††, Nonmembers, Mitsuaki AKIYAMA††, and Katsunari YOSHIOKA††, Members

SUMMARY Android app developers sometimes copy code snippets posted on a question-and-answer (Q&A) website and use them in their apps. However, if a code snippet has vulnerabilities, Android apps containing the vulnerable snippet could also have the same vulnerabilities. Despite this, the effect of such vulnerable snippets on the Android apps has not been investigated in depth. In this paper, we investigate the correspondence between the vulnerable code snippets and vulnerable apps. We collect code snippets from a Q&A website, extract possibly vulnerable snippets, and calculate similarity between those snippets and bytecode on vulnerable apps. Our experimental results show that 15.8% of all evaluated apps that have SSL implementation vulnerabilities (Improper host name verification), 31.7% that have SSL certificate verification vulnerabilities, and 3.8% that have WEBVIEW remote code execution vulnerabilities contain possibly vulnerable code snippets from Stack Overflow. In the worst case, a single problematic snippet has caused 4,844 apps to contain a vulnerability, accounting for 31.2% of all collected apps with that vulnerability.

key words: Mobile App, Vulnerabilities, Question and Answer Website, Code Snippets

1. Introduction

The popularity and spread of mobile devices have led to a huge number of mobile apps. Various mobile app developers, from professionals to amateurs, register their apps in app markets, and those apps are download by a great number of users through the markets.

However, these developers, especially inexperienced ones, can create apps with serious vulnerabilities, for example, allowing malicious apps access to a user’s personal information. On the other hand, market providers can take countermeasures against these vulnerable apps. For example, Google Play[1] warns developers about creating vulnerable apps and may ban vulnerable apps if the developers do not take proper precautions [2]–[4]. Despite such actions, many app markets have been reported to indeed offer numerous vulnerable apps [5]–[7].

We envision that one possible source of these vulnerabilities may be community websites for software developers. These websites, also called question-and-answer (Q&A) websites, provide the developers opportunities to discuss, ask, and answer questions regarding app developments and have grown in popularity. Indeed, the rich source of information given by the public discussions on these websites often provides quick solutions to the developers. Inexperienced developers especially tend to seek direct help and advice with ready-to-use code snippets from these websites. Moreover, they may simply copy such code snippets and use them in their own apps without checking their security. Thus, vulnerable snippets on the Q&A websites may be causing some apps’ vulnerabilities.

In this paper, we investigate the reuse of code snippets from a Q&A website into the mobile apps to see if vulnerable snippets are indeed causing vulnerabilities of the apps. For the investigation, we first collected 243,589 code snippets from Stack Overflow [8], a representative Q&A website. From the collected snippets, we selected 209 vulnerable snippets that can cause app vulnerabilities. Additionally, we collected 61,910 apps from two Android app markets (Google Play and Qihoo 360 Mobile Assistant) including 47,081 free apps and 14,829 paid apps and identified 8,275,112 vulnerable classes in 48,333 apps by using a vulnerability scanner. Finally, we investigated the correspondence between the vulnerable code snippets and the collected apps by using a novel technique to compare bytecodes in each vulnerable class with the vulnerable snippets.

Our Contributions

• We collected code snippets from Stack Overflow, analyzed a connection between code snippets and vulnerable apps, and investigated vulnerabilities attributable to using code snippets.
• We found that 15.8% of all evaluated apps that have SSL implementation vulnerabilities (Improper host name verification), 31.7% that have SSL certificate verification vulnerabilities, and 3.8% that have WEBVIEW remote code execution vulnerabilities contain the possibly vulnerable code snippets from Stack Overflow.
• We designed and implemented a fully automated large-scale processing for analyzing correspondence between code snippets and Android apps.
• Our experimental results clarified that copy&paste-
based reusing of code on a Q&A website is one vector causing app vulnerabilities.

2. Background

2.1 Code Snippets on Q&A Website

A code snippet is a fragment of code that is ready-to-use. By reusing code snippets, developers can create apps efficiently. Developers often use code snippets created by others as well as themselves.

A Q&A website for developers has a lot of code examples that developers can use easily. Developers can post questions on the Q&A websites, and other developers may answer these questions with code snippets for solving the questioners’ problems. Additionally, it is possible to search already posted questions and answers by using keywords, tags, and so on. Also, users can refer to the evaluation of the answers with code snippets based on the votes by other users. Our study focused on Stack Overflow [8], which is one of the most popular Q&A websites for software developers. Code Snippets on Stack Overflow are surrounded by <code> tags and can therefore easily be crawled. Figure 1 illustrates an example of a question and a corresponding answer posted on Stack Overflow.

2.2 Android Apps Vulnerabilities

There are various categories of vulnerabilities in Android apps. Table 1 lists the Top Ten Mobile Risks in 2016 reported by the Open Web Application Security Project (OWASP) Mobile Security Project [9]. This table shows that client attacks, network attacks, and server attacks seriously threaten the security of Android apps. For example, vulnerabilities generated defects in AndroidManifest.xml, inappropriate implementations of Secure Sockets Layer (SSL) or Transport Layer Security (TLS), inadequate implementations of WebView, and so on. Almost all these vulnerabilities were the fault of careless developers. Additionally, there are a lot of existing tools and services for detecting these vulnerabilities. In this paper, we use Vulnerability Scanner AndroBugs [10] and focused on three kinds of vulnerabilities attributable to the method of implementation.

3. Method

3.1 Overview

An overview of our approach is shown in Fig. 2. In the collection phase, we collected code snippets in answer posts from Stack Overflow (Fig. 3) and downloaded apps from multiple app markets. In the analysis phase, we selected potentially vulnerable code snippets from all collected snippets and extracted features from them. Additionally, we checked
vulnerable classes in collected apps by using AndroBugs and extracted features from vulnerable classes flagged by AndroBugs. In the comparison phase, we calculated the similarity between features of code snippets and features of app and investigated code snippets that are possible causes of vulnerabilities. Output data include similarity between bytecode and code snippets and also kinds of vulnerability. We can check whether vulnerabilities are attributable to reusing vulnerable code snippets or not by analyzing output data, e.g., we can determine if codes in apps were reused from a Q&A website by setting a threshold of similarity value.

3.2 Method of Calculating Similarity between Code Snippets and Bytecode

We explain our proposed method for calculating similarity between code snippets and bytecode. It is difficult to investigate whether code snippets have vulnerabilities or not because existing vulnerability scanners can only scan compilable source codes or bytecode after compilation. Likewise, source codes on websites are small pieces in many cases, so converting source codes into bytecode automatically is also difficult. Furthermore, developers are assumed to modify code snippets and implement their apps using somewhat modified code snippets. A method of comparing and evaluating partially processing sequences or features needs to be used to specify code snippets that cause vulnerabilities of apps. Therefore, we evaluated similarities between code snippets and bytecode by using common features of method call sequences and method definitions.

We used Longest Common Subsequence (LCS) and Levenshtein Distance (LD) to evaluate method call sequences. Additionally, we also used the rate of concordance for types of the method definitions (TMD) and the similarity degree of the name of method definitions (NMD) to evaluate method definitions. Finally, we expressed the evaluation formula (1), which consists of LCS, LD, TMD, and NMD values.

The method of extracting features from code snippets is shown in Fig. 4. We extracted method calls in the method body, the name of the method, and type modifier characters in method definitions from code snippets. The method of extracting features from bytecode in apps is shown in Fig. 5. We also extracted information from bytecode in the same manner as above.

3.2.1 Comparative Approach to Information Group of Methods

The comparative approach to the information group of methods is shown in Fig. 6. In this instance, the left side is a code snippet that has two method definitions, and the right side is a bytecode that has four method definitions. We compare each method sequentially and find the most similar method pairs between a code snippet and a bytecode. In this case, ① and ② are the most similar pairs. We describe the method of calculating similarities later.

3.2.2 Evaluation Formula

We calculate similarities between code snippets and bytecode using the following Eq. (1). The number of method definitions in code snippets is \( n \) and the number of method definitions in one class of apps is \( m \). We describe each distance, LCS, LD, TMD and NMD in Sects. 3.2.3, 3.2.4, 3.2.5 and 3.2.6, respectively.

\[
Score = \sum_{i=1}^{n} \max_{0 \leq j \leq m} (LCS_j + LD_j + TMD_j + NMD_j) 
\]

(1)

Note that specific features may greatly affect the equation because no similarity is independent of others. Our future work is to define the formula while considering the dependency of these similarity definitions.
3.2.3 Longest Common Subsequence (LCS)

The LCS is an algorithm to find the longest subsequence common to all sequences in a set of sequences. It is suited to evaluate partial method call sequences and able to evaluate appropriate similarities if developers modify original code snippets, e.g., inserting other method calls or reordering call sequences.

The number of method calls in code snippets is $S_i$ and the longest common subsequence can be expressed as the following equation.

$$LCS_j = \frac{S_i}{\sum_{k=1}^{n} S_k} \times \max_{0 \leq l \leq m} LCS_i$$

3.2.4 Levenshtein Distance (LD)

The LD is a string metric for measuring the difference between two sequences. Similar to the longest common subsequence, it is also suited to evaluate partial method call sequences and able to evaluate appropriate similarities if developers modify method call sequences. The number of method calls in one class of apps is $A_i$, and the LD can be expressed as the following equation.

$$LD_j = \frac{S_i}{\sum_{k=1}^{n} S_k} \times \max \left(1 - \frac{LD_i}{S_i + A_i}\right)$$

3.2.5 Rate of Concordance for Types of Method Definitions (TMD)

We calculate similarities between code snippets and bytecode from the viewpoint of modifiers, the type of return value, the type of argument, and method definitions. In many cases, method definitions in code snippets include characteristic patterns of a type modifier. Therefore, we can capture these features by using the following equation.

$$TMD_j = \frac{S_i}{\sum_{k=1}^{n} S_k} \times \max_{0 \leq l \leq m} TMD_i$$

3.2.6 Similarity Degree of Name of Method Definitions (NMD)

Code snippets often have characteristic names of method definitions. We calculate similarities between code snippets and bytecode from the viewpoint of the NMD by using a 3-gram model that can capture features of partial match because developers may modify names of method definitions depending on their tastes.

$$NMD_j = \frac{S_i}{\sum_{k=1}^{n} S_k} \times \max_{0 \leq l \leq m} NMD_i$$

4. Experiment

In this section, we analyze the relevance between apps and code snippets by calculating similarities between them with the method proposed in Sect. 3. In Sect. 4.1, we describe our dataset. In Sect. 4.2, we calculate similarities between vulnerable code snippets and vulnerable apps using our proposed method explained in Sect. 3 to analyze their relationship.

4.1 Datasets

4.1.1 Collected Code Snippets

First, we extracted question posts tagged with “android” from all pages of Stack Overflow. Second, we collected URLs viewed $L$ or more times by users. In this paper, $L$ is 1,000. Finally, we obtained 243,589 code snippets crawled from all answer posts on the above-mentioned collected URLs.

We extracted possibly vulnerable code snippets from all collected code snippets by using the classification conditions in Table 2. Specifically, we extracted code snippets that contain specific API calls related to the focused vulnerabilities. The number of code snippets used for the further evaluation is shown in Table 2. Note that code snippets outside method definitions are not covered in this experiment.

4.1.2 Collected Android Apps and Focused Vulnerabilities

We collected apps from Google Play, the official Android market, and Qihoo 360 Mobile Assistant [11], a third party market. Datasets of Google Play include both paid apps and free apps, but datasets of Qihoo 360 Mobile Assistant include only free apps. The breakdown of these datasets is in Table 3. Additionally, Table 4 presents the results of vulnerability scanning by AndroBugs.

As was mentioned in Sect. 2.2, there are various kinds of vulnerabilities in Android apps. We focus on three critical vulnerabilities named by the vulnerability scanner AndroBugs: SSL-CN2, SSL_X509, and WEBVIEW_RCE.
SSL-CN2 and SSL-X509 are related to insecure communication by Secure Sockets Layer. That is, these vulnerabilities indicate that apps do not verify server certificates, which allows attackers to do man-in-the-middle (MITM) attacks. These vulnerabilities may leak sensitive information such as login credentials. WEBVIEW_RCE is related to WebView and Remote Code Execution. It causes apps to allow external JavaScript to control the host application. This critical vulnerability allows attackers to execute arbitrary codes by malicious JavaScript.

### 4.2 Evaluating Precision of Proposed Method

We evaluated the precision of the proposed method using apps created by ourselves using several code snippets. First, we selected five code snippets that are likely to have been reused by developers. Second, we simulated developer’s behaviors of copying and pasting code snippets into their apps by actually creating five apps by using the code snippets with IDE autocomplete. Finally, we applied the proposed method to see if the code snippets and the five created apps could be correctly matched. Table 5 presents the results of the evaluation. We show that the proposed method ensures high precision in the five code snippets.

### 4.3 Experimental Results

We calculated similarities between the collected code snippets and apps using the proposed method in Sect. 3 with a similarity threshold of 0.8. In sect. 4.2, we evaluated the precision of the proposed method. From that results, if developers copy and paste code snippets into their apps without modifications, the Score is not less than 0.9. As mentioned above, developers are assumed to modify code snippets and implement their apps using somewhat modified code snippets. Hence, we decided a similarity threshold of 0.8. considering somewhat modified code snippets.

Note that we used code snippets with two or more method calls because our method cannot handle such small snippets properly and may produce false matches. We discuss this limitation in Sect. 5.3. Table 6 presents the results. We revealed that 31.7% of all evaluated apps that have X509 vulnerabilities contain the possibly vulnerable code snippets from Stack Overflow. Especially, we confirmed that a single problematic snippet has caused 4,844 apps to contain X509 vulnerabilities, which is 31.2% of all collected apps with that vulnerability.

The above results confirm that the possibly vulnerable code snippets indeed have a high chance of being contained in the vulnerable apps. However, the results do not confirm if these snippets are indeed causing the vulnerability of the apps. To clarify that, we analyze the relationship between the snippets and vulnerable apps in more detail.

We selected the “most popular” vulnerable code snippet from each of the three vulnerabilities. That is, for each of the three vulnerabilities, we selected the code snippet that is used most frequently in the corresponding vulnerable apps. However, the results do not confirm if these snippets are indeed causing the vulnerability of the apps. To clarify that, we analyze the relationship between the snippets and vulnerable apps in more detail.

We selected the “most popular” vulnerable code snippet from each of the three vulnerabilities. That is, for each of the three vulnerabilities, we selected the code snippet that is used most frequently in the corresponding vulnerable apps. Then, we checked if these code snippets are indeed used only in the vulnerable classes in the vulnerable apps identified by AndroBugs. As a result, 95.8% or more vulnerable classes indeed contained the potentially vulnerable code snippets, as shown in Table 7. We believe that the above results show that reusing code snippets on Stack Overflow greatly contributes to vulnerabilities of apps.
5. Discussion

5.1 Countermeasures by Market and Q&A Site Operators

Our experimental results confirmed that reusing code snippets on Stack Overflow strongly increases the likelihood of three vulnerabilities identified by AndroBugs: SSL_CN2, SSL_X509, and WEBSVIEW_RCE. Qihoo360 has a higher rate of vulnerable apps than Google Play, so Google Play can be said to focus more on countermeasures against vulnerabilities. Results also confirmed that Qihoo360 has a higher rate of vulnerable apps including potentially vulnerable code snippets. This may be because more unskilled developers are reusing potentially vulnerable code snippets on Stack Overflow in the Qihoo360 market. Additionally, we found that free apps are more vulnerable than paid ones. We speculate that some developers of free apps are nonprofessional and less able to eliminate vulnerabilities from their apps.

We found the same code snippets on many different pages, suggesting users of Q&A websites also copy and paste the code snippets. It would be problematic if potentially vulnerable code snippets spread over the sites in this way. To prevent such spreads, Q&A website operators should search for and remove these potentially vulnerable code snippets.

5.2 Enhancing Awareness for Developers

Developers should try to check whether code snippets are implemented correctly or not before they reuse them and they should not use code snippets that have vulnerabilities. Software modules can sometimes contain potentially vulnerable codes, and developers may use the modules by mistake without noticing the vulnerabilities.

5.3 Limitation

The proposed similarity calculation method has limitations. It is difficult to correlate code snippets with apps if code snippets do not have specific features, e.g., a snippet contains trivial functionalities and everyone will implement it in the same manner regardless of whether they reuse code snippets or not. This problem will also occur if a code snippet is small-scale.

Moreover, code obfuscation can also be problematic for calculating similarities. However, code snippets related to Android API are unaffected by obfuscations, thus we may be able to propose a new method for specifying vulnerable code snippets with obfuscations in our future work.

6. Related Works

Android apps vulnerabilities, mobile app developers, code clone detection, and developer community websites have been studied. However, few studies have focused on the effect of real code snippets on Android app vulnerabilities. In this paper, we collected code snippets from Stack Overflow, analyzed a connection between code snippets and vulnerable apps, and investigated vulnerabilities attributable to reusing code snippets.

6.1 Vulnerability Analysis

Fahl et al. [12] introduced a tool to detect potential vulnerability against MITM attacks. They analyzed 13,500 popular free apps downloaded from Google Play and revealed that 1,074 (8.0%) of the apps examined contained SSL/TLS code that is potentially vulnerable to MITM attacks.

Egele et al. [13] developed program analysis techniques to automatically check programs on the Google Play and found that 10,327 out of 11,748 apps that use cryptographic APIs - 88% overall - make at least one mistake. They then suggested specific recommendations on the basis of their analysis for improving overall cryptographic security in Android apps.

Furukawa et al. [14] investigate 15,064 of popular Android apps to identify their library version and to reveal statistical distribution of the versions. As a result, several apps even published on Japanese Google Play turn out to be using old version libraries warned their security risks.

6.2 Research on Mobile App Developers

Acar et al. [15] contacted developers in Google Play and investigated the resources the developers reference when they create apps, reference frequencies, developers’ experience, and so on. Moreover, they performed an experiment in which their research team members develop apps by limiting referenced resources such as Stack Overflow, official documents, and books. Furthermore, they evaluated completed codes from the point of view of functional correctness and security. They also investigated how specific implemented codes are related to security in real apps. By doing these surveys, they analyzed the effects of resources referenced by developers creating apps on security problems. As a result, they clarified not only that Stack Overflow contains more handy resources than official documents do but also that a lot of resources are possible causes of vulnerabilities.

Wang et al. [16] presented a study of the mobile app ecosystem from the perspective of app developers. On the basis of over one million Android apps and 320,000 developers from Google Play, they analyzed the Android app ecosystem from different aspects. Their analysis shows that while over half of the developers have released only one app in the market, many have released hundreds of apps. Then they classified 320,000 developers into four groups on the basis of the number of apps they released and analyzed the characteristics for different developer groups from the aspects of app quality, development behaviors, and privacy behaviors. The results revealed a wide variation among app developers. In particular, highly active developers, who
have created more than 50 apps, tend to release low-quality, less popular, and high privacy risk apps.

6.3 Code Clone Detection

Various studies [17]–[20] have been reported on code clone detection in Android apps. Hanna et al. [21] investigated present situations of code reuse using similarities between Android apps by focusing on specific operation codes included bytecode. Their results found some applications with confirmed buggy code reuse of Google-provided sample code that lead to serious vulnerabilities in real-world apps, some instances of known malware and variants, and some pirated variants of a popular paid games.

Zhou et al. [22] proposed a module decoupling technique to partition an app’s code into primary and non-primary modules and developed a feature fingerprint technique to extract various semantic features. Their investigation shows that piggybacked apps are mainly used to steal ad revenue from the original developers and implant malicious payloads.

6.4 Research on Developer Community Website

Liu et al. [23] proposed a unified framework to tackle the challenge of detecting collusive spamming activities of Community Question Answering, which provides rich sources of information on a variety of topics. They also proposed a combined factor graph model to detect deceptive Q&As simultaneously by combining two independent factor graphs. Using a large-scale practical dataset, they found that their proposed framework can detect deceptive contents at an early stage and outperform a number of competitive baselines.

Fischer et al. [24] crawled Stack Overflow for code snippets and evaluated their security score using a stochastic gradient descent classifier. As a result, they revealed that 15.4% of the 1.3 million Android apps they analyzed contained security-related code snippets from Stack Overflow. Out of these, 97.9% contained at least one insecure code snippet.

7. Conclusion

We analyzed the relationship between android apps vulnerabilities and code snippets on a Q&A website. As a result, we showed that a single problematic snippet has caused 4,844 apps to contain a vulnerability, which is 31.2% of all collected apps with that vulnerability. Moreover, we found the same potentially vulnerable code snippets are on many different pages and should be removed by the site operators.

In the future, we will try to comprehensively investigate code snippets that are possible causes of vulnerabilities besides the Q&A website. Additionally, we will also analyze the mechanism of vulnerable app development using potentially vulnerable code snippets.
NAKANO et al.: TOWARDS FINDING CODE SNIPPETS ON A QUESTION AND ANSWER WEBSITE CAUSING MOBILE APP VULNERABILITIES


Hiroki Nakano received his B.E. and M.E. degree in computer science from Yokohama National University, Japan in 2016 and 2018. He has been a researcher at NTT Secure Platform Laboratories, Japan, since 2018. His research interests are mobile security and web security, especially program analysis, content analysis and code snippet analysis.

Fumihiro Kanei received the M.E. degree from Yokohama National University, Japan, in 2015. He has been a researcher at NTT Secure Platform Laboratories, Japan, since 2015. His research focuses on mobile devices and applications security, such as vulnerability assessment for mobile applications, static and dynamic analysis of mobile malware, and code obfuscation and tamper-proofing for bytecode applications.

Yuta Takata received his B.E., M.E., and Ph.D. degrees in computer science and engineering from Waseda University, Japan in 2011, 2013, and 2018. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2013, he has been engaged in research and development on cyber security, especially honeypot/honeynet, program analysis, and content analysis. He is now a researcher in the Cyber Security Project of NTT Secure Platform Laboratories.

Mitsuaki Akiyama received his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology, Japan in 2007 and 2013. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2007, he has been engaged in research and development on network security, especially honeypot and malware analysis. He is now with the Cyber Security Project of NTT Secure Platform Laboratories.

Katsunari Yoshioka received his B.E., M.E. and Ph.D. degrees in Computer Engineering from Yokohama National University in 2000, 2002, and 2005, respectively. From 2005 to 2007, he was a Researcher at the National Institute of Information and Communications Technology, Japan. Currently, he is an Associate Professor at the Graduate School of Environment and Information Sciences, Yokohama National University. His research interest covers wide range of information security, including malware analysis, network monitoring, intrusion detection, etc. He was awarded 2007 Prizes for Science and Technology by The Commendation for Science and Technology by the Minister of Education, Culture, Science and Technology.