Multiple Device Login Attacks and Countermeasures of Mobile VoIP Apps on Android

Su Wan Park and Jeong Hyun Yi*
School of Compuer Science and Engineering, Soongsil University
Seoul, 156-743, Korea
{skyhwen, jhyi}@ssu.ac.kr

Abstract

Because Android apps are structurally easy to decompile, attackers may, using reverse engineering, modify the source code or inject some code of his choice. If a mobile messenger app were to be attacked in the same nature, the attacker can bypass the authentication mechanism applied on the app to not only view past conversations and Time line records of a particular user but to also receive and view real time conversations. In addition, there are widespread attacks on the apps' weak points depending on the app such as wiretapping VoIP and other voice messages or illegally use of pay items. Therefore, in this paper, we analyze the security weak points of app A and app B, two representative Android message apps, and propose effective solutions.

Keywords: android, repackaging, multi device login, voip

1 Introduction

With the sudden widespread penetration of the Smartphone, messenger apps quickly replaced the role of SMS (Short Message Services). These apps, by not only providing simple message sending services but also sending media contents, audio files, and contacts, has established themselves as another platform to generate and reinforce social networks. The prime messenger app A and app B, as of December 2013, have over few hundred millions users.

But despite this importance, messenger apps are vulnerable to various attack, and only few has implemented mitigation to defend it. In this paper, we analyze the app A and app B which are protected in part by security techniques to show that bypassing of current protective mechanisms is possible and, ultimately, that tapping into others' conversations and leaking of personal information using multi log-ins from two different devices is possible. Furthermore, we propose countermeasures to supplement these weaknesses.

This paper is organized as follows. In section 2, we delineate relevant research to the topic. Section 3 explains security attacks to app A and app B while Section 4 looks into the results of the experiment where we carried out such security attacks. In section 5, we propose countermeasures to remedy the apps' vulnerabilities and, finally, in section 6, we make our conclusion.

2 Related Work

Due to the structural characteristics of Java code and self-signing, repackaging Android apps is an easy task [8] [9]. Furthermore, bypassing the protective mechanisms within the app is feasible if an attacker

Journal of Internet Services and Information Security (JISIS), volume: 4, number: 4 (November 2014), pp. 115-126
*Corresponding author: 369 Sangdo-ro, Dongjak-gu, Soongsil University, Information Science Bldg. 409, Seoul 156-743, Korea, Tel: +82-2-821-0914, Web: http://msec.ssu.ac.kr/

pinpoints the location of the code, while detecting the code location is uncomplicated. That is, if an attacker only determines the location of the protective mechanism routine of typical apps, one can manipulate the control flow through repackaging attack. The attacker can bypass the entire tamper detection part and also insert any faked value into the appropriate parameter.

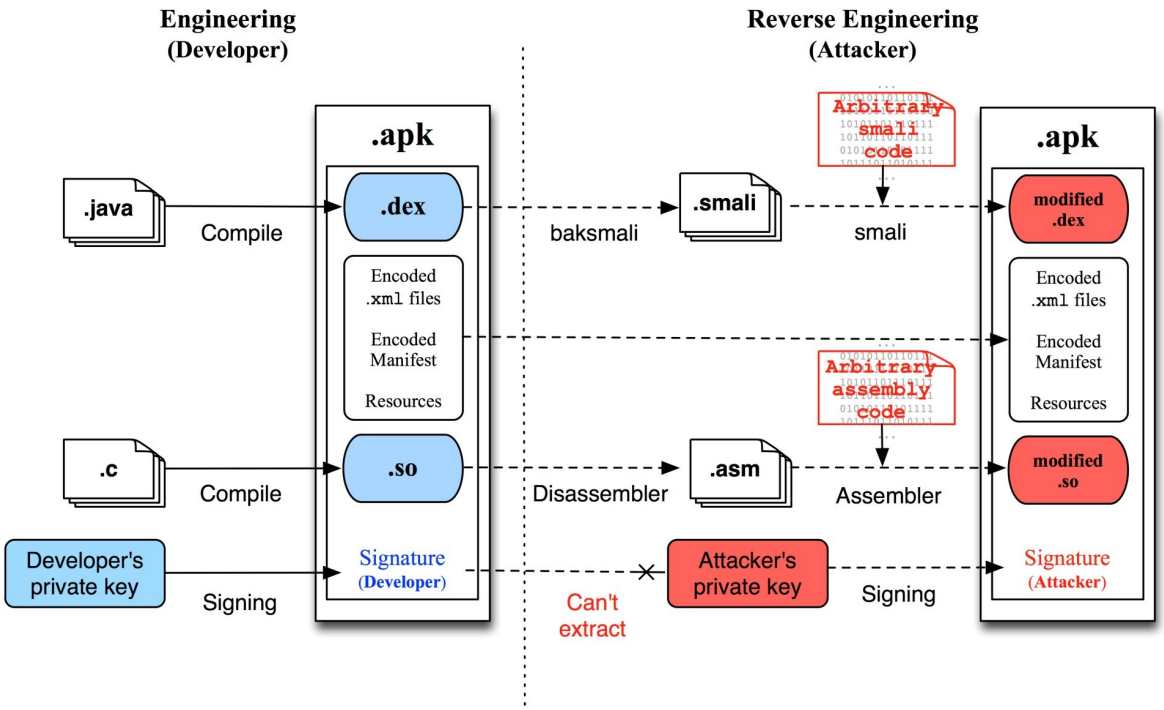


Figure 1: Overview of Android repackaging attack

As shown in Figure 1, the attacker may, using baksmali [1] and disassembler like IDA [5], easily identify the control flow of the source code and, using this information, may inject code into whatever desired location. Because extracting anything from a signed file being impossible, the attacker lastly assembles a new APK and signs it with a personal private key before proceeding to either personally use or even distribute the new APK.

3 Security Analysis

Table 1 shows the different security techniques applied to the major mobile messenger apps. Note that we analyze the latest version of app at that time. The results of reverse engineering on nine major apps, showed that the app A was the only one with a tamper detection feature and app A and app B were the two that checked if a previously authenticated device is the current device in use, disallowing multi-logins.

Before we describe couple of attack vectors, need to address about analytical method. As said above, arbitrary code can be inserted to application. Main obstacle is how to find the code area we want to modify. To solve this our analysis framework, which put particular code alerting monitoring tool to first and end point of every function in appointed module, is used. So, we can build call graph of target app, and furthermore specific code location is able to be found by combining with string information. Detail methods using the obtained data will be introduced in each subsection.

	app A	app B	Others
Obfuscation	O	O	O
Tamper detection	O	X	X
Device authentication	O	O	X

Table 1: Security techniques applied to each app

3.1 Multi Device Login

This attack needs the victim's credential files which is stored on the path:*/data/data/appname*, because application save the chat and connection information to those. Using repackaged or 3rd party app, extract the file, and put the data to the device of the attacker. Almost of messenger, don't apply any security techniques to protect from this attack, but app A and app B implement some functions. To bypass them, we perform the several thing as follows.

3.1.1 Bypassing Tamper Detection

We will begin by talking about the app A. The app A puts relatively much consideration into security with very minimal empty space in its log and providing a tamper detection feature. However, once we decompile the app and search the string, one can see that there exist functions that leave a log and that the functions that have, as one of theirs parameters, a string that appears as log is invoked. To analyze this part, after causing all logs to be outputted, we used the collected information to custom build a tamper detection section. This is made possible if afterwards one creates a section that checks whether the authentication information within the file aligns with the current device information after one manipulates the return value and bypasses the area in concern while adding its credential into the app folder. Figure 2 shows how to bypass the tamper detection methods for app A.

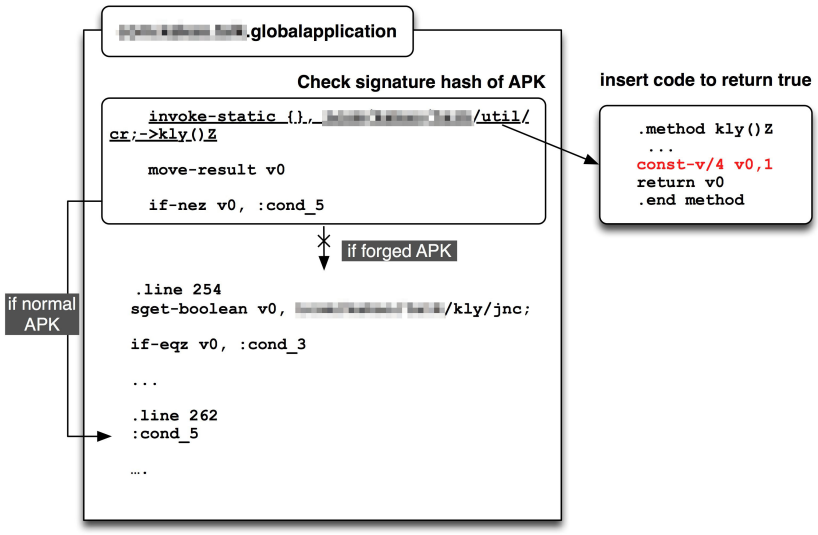


Figure 2: Bypassing app A tamper detection

For the app A app, while the kly function of the util/cr class executes tamper detection, if one inserts a code to ensure that this function returns only the true value, one can easily manipulate app A to run like a typical app.

It is not with just these two methods that one can bypass detection. The majority of functions that inspect particular sections are in the form shown in below.

boolean CheckIntegrity() { ... }

Although the function is structurally problem-free, the function will only return either a true or false return value despite how complicated the internal logic. Due to this, the attacker only needs to find the location of the function using string or log and can disregard the complicated internal calculations to bypass detection without difficulty.

3.1.2 Bypassing Device Authentication

Likewise, bypassing the authentication is possible using a similar method. But, mechanism of app B is slightly different with app A. For this app, the device ID is used in decrypting data so a specific value must be inserted.

Figure 3 shows tampering to enable normal data decryption and duplicate access when one the device ID of the login device is inserted even if the actual device in use differs.

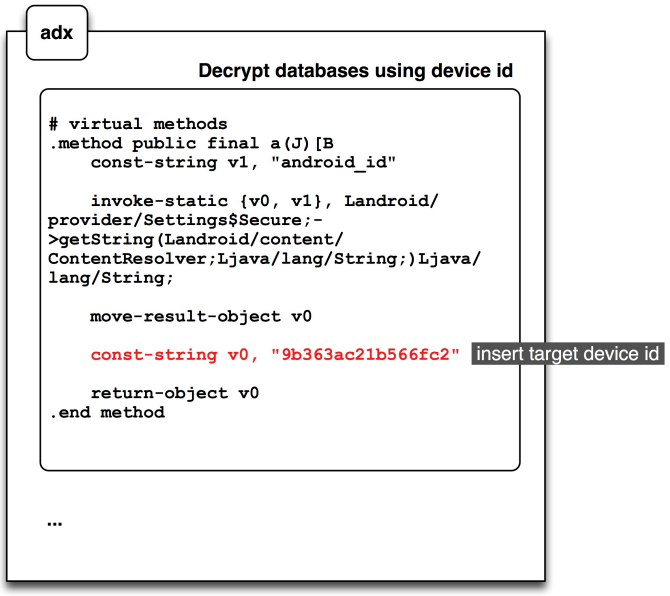


Figure 3: Modifying decryption routine on the app B

3.2 Bypassing Item Payment

Using above method, Figure 4 shows the method for bypassing the section that checks whether the item is paid for or not. The app features can also be manipulated just as easily.

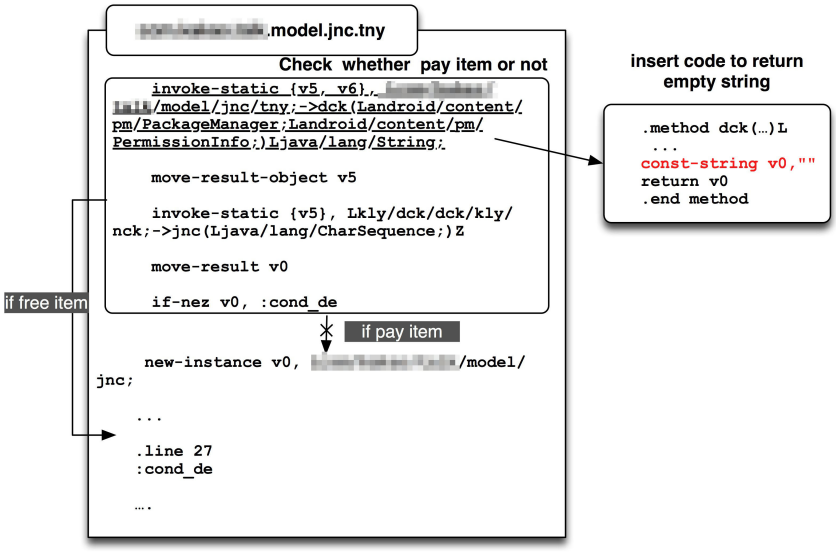


Figure 4: Bypassing item payment checking for app A

3.3 VoIP Hooking

Next, let us examine VoIP hooking. Interestingly, the VoIP module does not exist on the smali code level but is located at *.so* library. Although the fact that one must deal with assembly code instead of smali poses some difficulty, modifying the library is by no means impossible. In fact, because the native code in *.so* library is executed internally inside Dalvik VM [2] rather than externally, dynamic debugging, which makes following the flow of code and memory, becomes easier using the tools like gdb [3] in android ndk [6] and IDA [5].

Because app A encrypts the moment it connects to VoIP, to gain an audio buffer that can play back, app A must hook with a function that uses the corresponding memory address of the particular function as a factor. Although there are multiple possible hooking points, the play or record functions, as functions provided by Android, are easy to pry into. Because the register value can change during function execution, hooking occurs immediately before the play function or immediately after the record function. Instead of executing the original code for the function, this will execute the code that diverges the function containing the desired assembly. The corresponding custom function must, immediately before and after executing its code, return to standard routine after summoning the originally called upon function while, also, save and restore the registers. Furthermore, because VoIP executes the receiving and transmitting sections separately, it is in need of a process that corresponds to both functions. For the text section, because it lacked space to inject code, we created and summoned a new section. Figure 5, shown below, explains the aforementioned VoIP hooking process.

Insert the modified library file into APK file or replace original file which is stored data path. Or this can be performed in different way like binder hook or memory hacking from 3rd party app. If app updates library file using insecure connection, this attack will be implemented more smoothly by performing MITM attack.

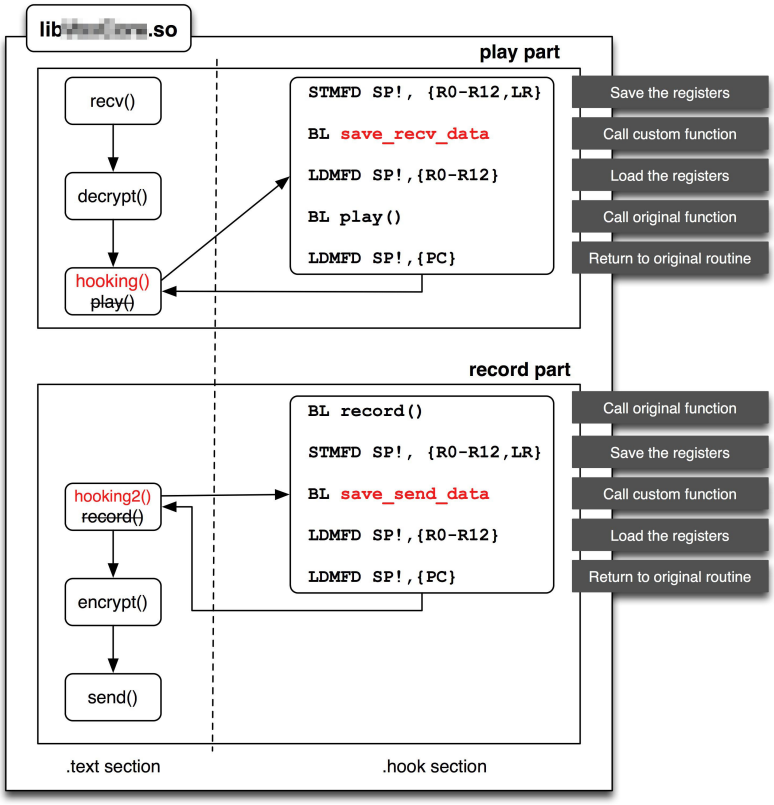


Figure 5: VoIP hooking mechanism

4 Experimental Results

After bypassing tamper detection and the device authentication, one can use the credential files of a different user to allow multi-device login. If the multi-login is successful, the hacker is able to not only view the past user's conversation records and list of friends as shown in Figure 6 below but can also send or receive messages in real time.

We confirmed this attack is possible not only app A and app B implementing security mechanism, but also the others. Of course, the latter apps are no trouble to do, because they don't have to repackage for control flow modification, and just need information files of victim device. Even some programs allow other applications to access their own data file. It makes remote attack using 3rd party app possible in non root device.X

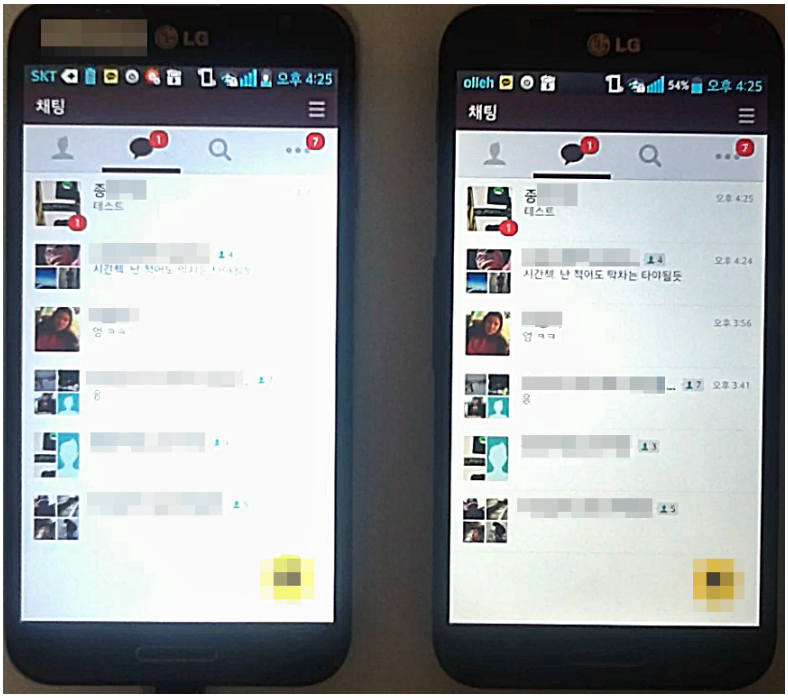


Figure 6: Result from multi-device login with repackaging

The attackers can have unauthorized use of existing pay items, as shown in Figure 7, if one bypasses the tamper detection and pay item inspection routine. In addition, the exposed control flow reveals the URL from which one can directly download pay items and the file containing authentication information, allowing downloading and even free distribution of pay items.

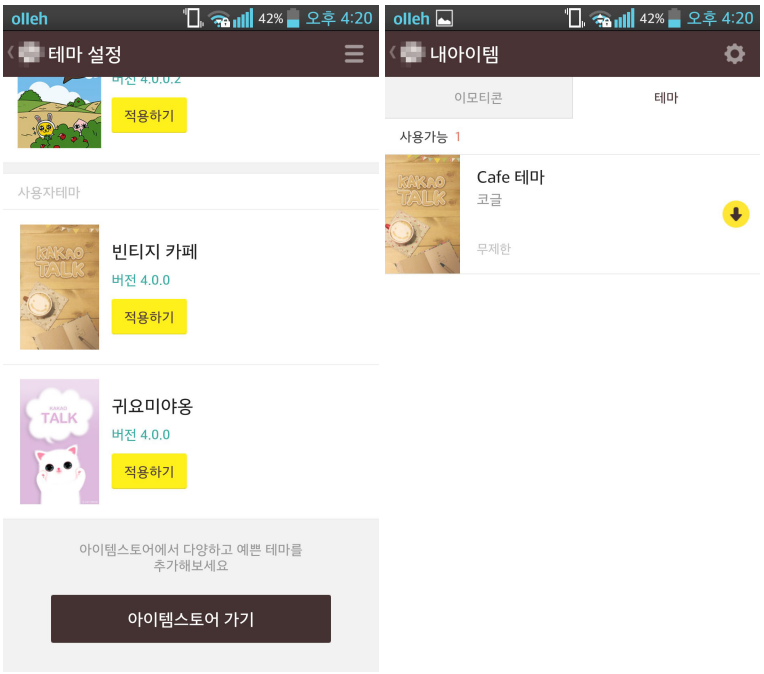


Figure 7: Unauthorized use of pay items for app A

When such code injection is applied to VoIP, an attacker is able to record, save, and send the contents of a different user's conversation. Figure 8, shown below, explains this process.

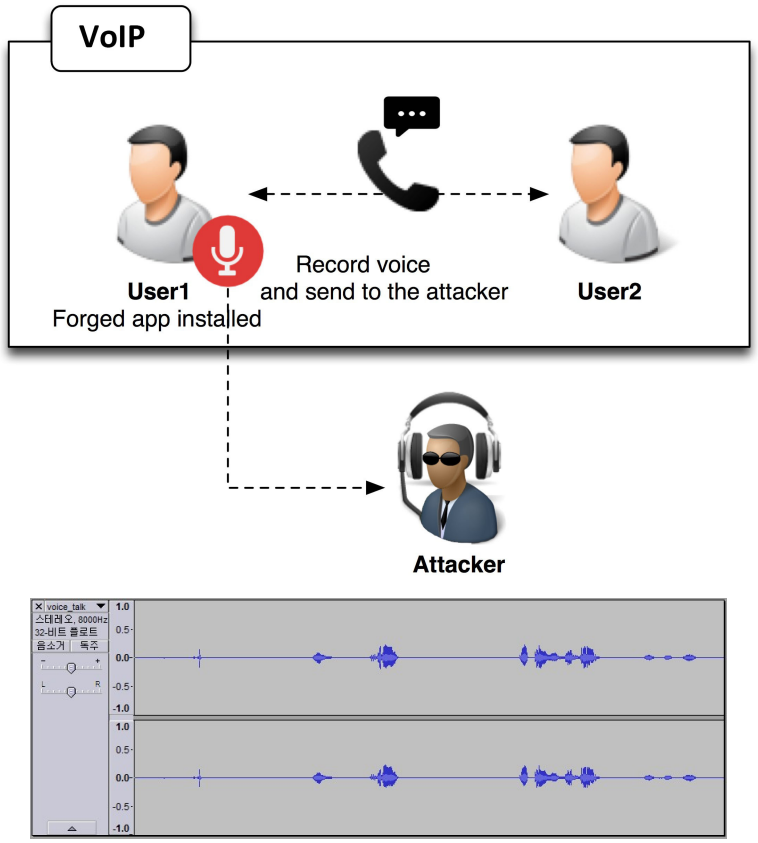


Figure 8: VoIP hooking results

5 Countermeasures

The tamper detection feature, device authentication, and other such features on the app A or app B are easily bypassed. To supplement for these weak points, the following countermeasures can be considered.

5.1 Code Integrity Check

First, there is the server-based integrity checking method. Most commonly used for integrity checking is the hash authentication method in the signature. The hash value is useful in verifying the integrity of falsified areas of a code that an attacker has repackaged. However, if this routine exists within the app's internals, the attacker can also tamper with this method as well, making this method insecure. For this reason, we must send this signature value using the server without having the client internals verify the signature. Because the transmission value may be exposed, SSL communications must fundamentally be used and to prevent attacks through memory dump, the memory area containing the hash string should be immediately deallocated.

The second method is providing necessary information to the registered client only if initial verification was successful. Currently, most apps use the method of running the integrity checking routine only

once and allowing the app to run normally if the return value from the checking is correct. This method is very vulnerable to reverse engineering and leaves an identification flag within the server. It is only good to use this information sending method only when this flag is normal. If one uses this method, even if an attacker were to repackage the app to make it appear as a normal app, one cannot gain information from the server. Figure 9 shows the two proposed method used together.

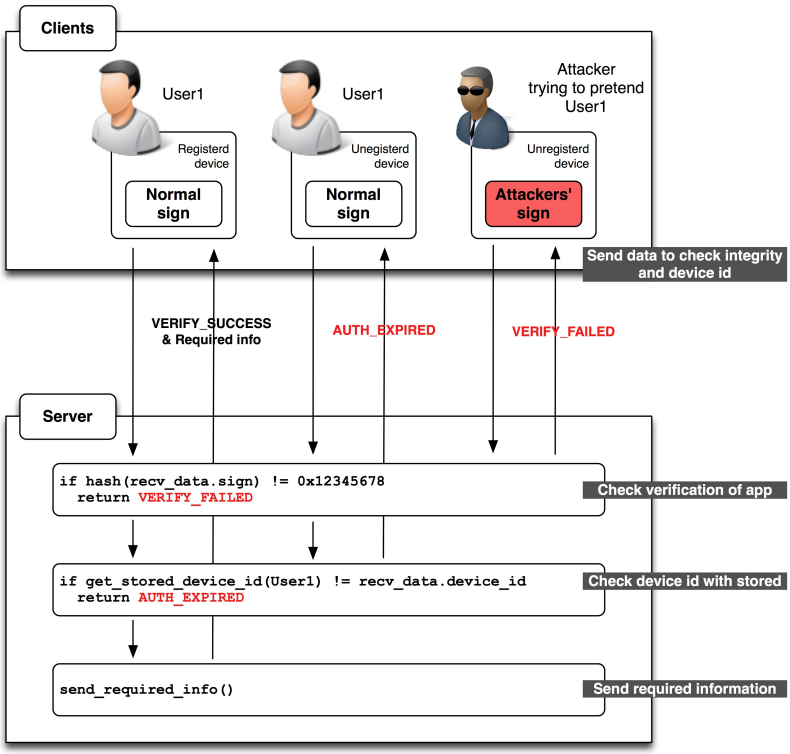


Figure 9: Server based integrity checking mechanism

For this to happen on an actual server, more elaborate verification must take place. We must keep in mind that because VoIP along with the library file could be falsified, simply using a signature value test does not suffice.

5.2 Dynamic Code Injection

Because repackaging attacks occur statically, use of the dynamic code injection method hides the control flow and makes code modification more difficult. The method of injecting code after receiving, through communication with the server, the code to be rewritten minimizes the section made vulnerable to attackers. If one injects only a part of the contents of the function that executes the authentication process, executes that code, and then overwrites it back to the original code, attackers will have difficulty identifying the code using static analysis. Figure 10 gives a rough outline of the mechanisms of code injection.

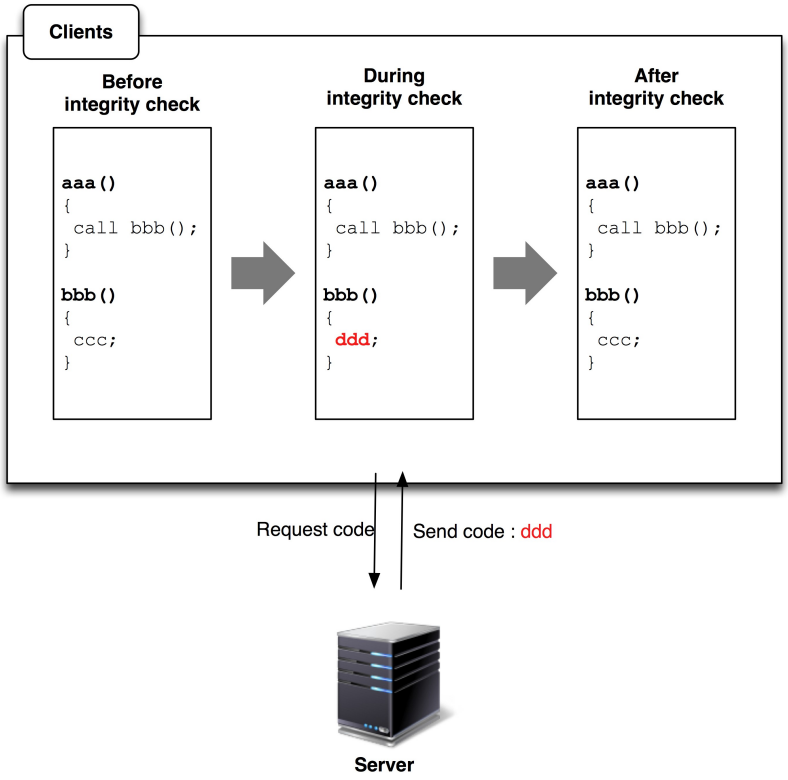


Figure 10: Dynamic code injection mechanism

5.3 Code Obfuscation

Many apps currently do not use obfuscation and, even if they do, they use the Android provided Proguard [7]. As much as Proguard is provided for free, there are many known methods of bypassing and the level of security is very low. As a result, applying the more highly secure obfuscation technique is good for hiding the API name, function name, and variable identifier. As the location of the security mechanism routine can be easily inferred through the log or string variable, it is important to encode the string variable and discard the log.

5.4 Code Attestation

Although obfuscation can run an analysis — with some difficulties due to the attacker — using the protective mechanism of the application layer with static code protection techniques, when running the app, it is difficult to protect the code. Supplementing this, to dynamically protect the code during the app runtime, code attestation technology that uses protective hardware such as TEE (Trusted Execution Environment) [4] as the trusted point is needed. Using this, not only protecting the smartphone platform but, by verifying the presence of tamper in the app, a more reliable protective service can be provided.

6 Conclusion

Until now, we have investigated how easily repackaging attacks on Android can occur. Although app repackaging attacks can happen on all types of apps, the experiment confirmed that if such attack occurs

on a mobile messenger app, security vulnerabilities such as exposure of real time messages using multi login, free use of pay items, wiretapping of VoIP, etc. may arise.

Because the concerns discussed in this paper can occur by simply installing an app even on devices that cannot be rooted, we recommend that all apps be downloaded and installed from the standard market. Also, even if a device has been rooted, one must always be mindful because, during the installation of a different app, the switching of the library file or a leak of the credential files of an unrelated package can occur.

For app developers, applying security methods such as code injection, code obfuscation, code attestation will prevent exposing the control flow, and continuous inspection of the integrity of the app will minimize repackaging attacks.

Acknowledgements

This research was supported in part by Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (NRF-2014K1A1A2043029), and in part by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Education (NRF-2013R1A1A2013041).

References

[1] Baksmali. `https://code.google.com/p/smali/`.

[2] Dalvik vm. `https://source.android.com/devices/tech/dalvik/index.html`.

[3] gdb. `http://www.gnu.org/software/gdb/`.

[4] Global platform device specifications. `http://www.globalplatform.org/specificationsdevice.asp`.

[5] Ida pro. `https://www.hex-rays.com/products/ida/`.

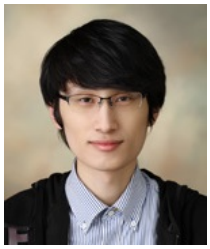
[6] ndk. `https://developer.android.com/tools/sdk/ndk/index.html`.

[7] Proguard. `http://proguard.sourceforge.net`.

[8] J. H. Jung, J. Y. Kim, H. C. Lee, and J. H. Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communication*, 73(1):342–351, June 2013.

[9] Arxan Technologies. State of security in the app economy: Mobile apps under attack, 2013. `https://www.arxan.com/assets/1/7/State_of_Security_in_the_App_Economy_Report_Vol._2.pdf`.

Author Biography



Su wan Park received his B.S. degrees in Computer Science from Soongsil University, Seoul, Korea in 2013. He is a M.S student in the Graduate School of Information Security at KAIST. His research interests include mobile application and platform security.

Jeong Hyun Yi is an Assistant Professor in the School of Computer Science and Engineering at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, Korea, from 2005 to 2008, and a member of research staff at Electronics and Telecommunications Research Institute (ETRI), Korea, from 1995 to 2001. Between 2000 and 2001, he was a guest researcher at National Institute of Standards and Technology (NIST), Maryland, U.S. His research interests include mobile security and privacy, network security, cloud computing security, and applied cryptography. Some of his notable research contributions include Certificate Management Protocol (CMP) for Korean PKI Standards and integration of Korea PKI and U.S. Federal PKI.