International Journal of Security and Its Applications
Vol. 10, No. 7 (2016) pp.259-268
http://dx.doi.org/10.14257/ijsia.2016.10.7.23

Securing Android In-app Billing Service against Automated Attacks

Heeyoul Kim[1*] and Sang-won Kim[1]

[1]*Department of Computer science, Kyonggi Univ, Republic of Korea*
heeyoul.kim@kgu.ac.kr

Abstract

In-app Billing service in Android enables the application developers sell digital content from inside the applications. To enhancing security of these applications, there have been presented security guidelines and various security techniques. However, recent automated attacks on every application in a device make the user of the device to get valuable content without paying for them. In this paper we present a novel approach to secure Android In-app Billing service against such automated attacks by detecting these attacks. Before completing the purchase, our approach performs a test to check whether there is an attempt to bypass a legitimate payment process. It is simple so as to be applied easily, and it effectively detects attacks by testing the signature verification process. With this approach, known automated attacks could be detected successfully.

Keywords*: Android, in-app Billing, security, automated attack*

1. Introduction

With the great increase of using mobile applications for smartphones, the markets for mobile applications have changed the way people purchase applications they are interested in [1]. Users can easily find, install, and update useful applications through Apple's App Store or Google's Play Store. One of important business models for these markets is encouraging users to pay for various content including game items, premium features, and subscriptions within applications themselves. Apple introduced an In-app Purchase service for this in 2010 [2], and soon Google provided a similar In-app Billing service in 2011 [3]. Users first try out an application with limited functionality and check whether it meets their expectations. If satisfied, they can purchase additional virtual goods such as premium items or in-app currency, and buy additional functionalities such as extra content or removing. They also can subscribe to the regular delivered content from inside the application. This model is very profitable and thus preferable to the application developers.

Security is very important in mobile ecosystems, and lots of research have been presented to prevent applications from various attackers [4-5, 14-16]. However, the attacks against the In-app Billing service have a different tendency from other attacks. They mainly focus on bypassing legitimate payment process to purchase valuable items. And these attacks are actively attempted by the user owning the mobile device with full control rights, which makes it more vulnerable than other traditional systems. The well-known attack in [6] provides an alternate black market containing cracked applications by disassembling, modifying, and re-packaging each application manually. Google provides security and design guidelines [7] for application developers using the In-app Billing service, and these are concentrated on preventing applications from being tampered directly through reverse engineering. The guidelines recommend to verify signature on a remote server and to obfuscate the codes and important data.

[*] Corresponding Author
Heeyoul Kim

ISSN: 1738-9976 IJSIA
Copyright ⓒ 2016 SERSC

However, recently several automated attacks such as VirtualSwindle [8] have appeared to enable the attackers to bypass the payment process of every application on the device. These attacks subvert the signature verification process to get digital content and services without paying for them. The attacker's code is dynamically added to the application by using of library injection [12], and the method java.security. Signature.verify() is replaced to the attacker's function returning true always. With these attacks, the applications employing on-device signature verification were easily cracked.

In this paper, we present a novel approach to prevent this kind of attacks by detecting the attempts to bypass the signature verification process. Before calling the verification method, our approach tests whether the method works correctly or not by intentionally modifying the data to be verified or the application's public key used for verification. It is simple because a few codes for test are added to the applications and there is no need to modify Android kernel codes or to obtain additional permissions. Furthermore, it is very effective because it successfully detects known automated attacks.

The paper is organized as follows. In Section 2, the In-app Billing service of Android is briefly explained. In Section 3, the security of it and provided guidelines are discussed. In Section 4, the mechanism of automated attacks is analyzed. Section 5 provides our simple but effective approach to detect automated attacks. Finally, Section 6 concludes the paper.

2. Android In-app Billing Service

The In-app Billing service in Android platform lets a developer sell digital content from inside his applications in the Android platform. This service consists of three components as shown in *Figure* 1. The Google Play server is responsible for performing the actual financial transactions remotely requested, and the checkout backend service is used as for application purchases. The Google Play app installed on the user's device conveys billing requests and responses between the application and the Google Play server. The application on the device accesses the In-app Billing service using the IInAppBillingService interface exposed by the Google Play app. The application never directly communicates with the Google Play server. Instead, it communicates with the Google Play app over interprocess communication (IPC) for billing.

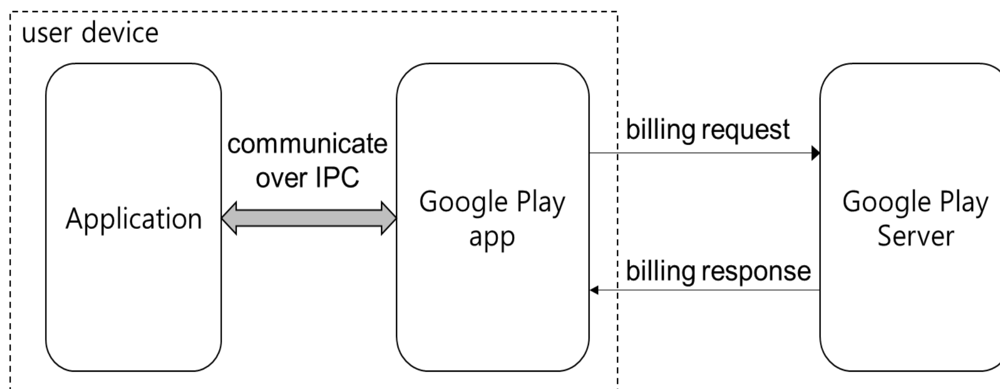


Figure 1. Android In-app Billing Architecture

Before publishing an application using the In-app Billing service, a developer needs to perform some steps in advance. The code to communicate with Google Play app must be implemented with appropriate APIs according to the guideline. The digital goods for purchase are to be registered in the Google Play Developer Console. A cryptographic public/private key pair has to be generated for the application. The private key is then registered in the Google Play server for signing purchase transactions. The corresponding public key is used to verify the integrity of the purchase transactions through the In-app Billing service.

 Copyright ⓒ 2016 SERSC

A simplified purchase flow between the application and the Google Play app is as *Figure* 2. The application sends an isBillingSupported request for supported version check. Then, to start a purchase request, the application sends a getBuyIntent request including the specific product ID of the item to purchase. The Google Play app returns a Bundle containing a PendingIntent used to start the checkout UI. The application launches the intent by calling the startIntentSenderForResult method. After the payment process finishes, the Google Play app sends a response Intent containing the detailed information about both the purchased item and the purchase transaction. The Intent also contains the signature of the purchase data, signed by the application's private key. The application should verify the signature with the corresponding public key to check whether a valid payment has been made. In fact, this verification process is the key to ensure a legitimate purchase has been completed and to filter out malicious attempts. In addition, the In-app Billing service provides another APIs for querying product details or consuming purchased items.

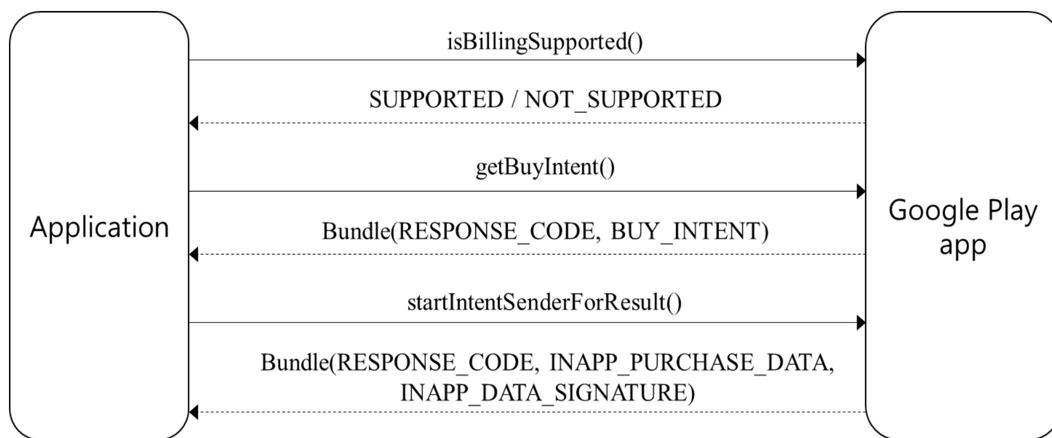


Figure 2. Purchase Flow for In-app Billing

3. Security Guidelines for Developers using In-app Billing Service

The goal of attacks on the In-app Billing service is to bypass payment process for purchasing items. Obtaining valuable items without any payment is very attractive, and the success of automated and repeated attack will give financial benefits to the attacker. For example, the expensive items obtained by the attacks can be traded in the black markets to make money.

The big difference of this kind of attacks is that the user owning his mobile device actively performs attacks on the applications. In general, users are assumed in a defensive position to prevent their data from malicious attackers. However, in this case, the user may try various attacks to find vulnerability of the In-app Billing service with full control over his own device. In the android platform, the process called as rooting allows the user to attain root access on the device. Rooting is sometimes useful to overcome limitations that manufacturers set, and many vendors such as HTC [9] and Google explicitly provide the ability to unlock devices. Moreover, there are a lot of documentations and tools such as SuperOneClick [10] to help unsophisticated users to gain root access easily. Therefore, in this paper the attacker is assumed to have full control with root privilege on the device.

Securing In-app Billing service against malicious attacks is important because most mobile application developers worry about losing revenue due to bypassing payment. Google has already recognized the significance of this, and provides security and design guidelines for In-app Billing service. The essence of the guidelines are as follows.

Copyright ⓒ 2016 SERSC

- If practical, the developer should perform signature verification on his remote server and not on a device itself. This implementation makes it difficult for attackers to break the verification process by reverse engineering.
- The developer should obfuscate the codes for In-app Billing so it is difficult for attackers to reverse engineer security protocols and other application components. Using obfuscation tools such as Proguard [11] is recommended. Furthermore, using method inlining, constructing strings on the fly instead of constants, and using Java reflection to call methods are recommended for further obfuscation.
- If the developer reuses the sample code provided by Google, it should be modified to prevent easy reverse engineering.
- The developer should set the developer payload string when making purchase request to identify the user who made the purchase.
- The application's public key to be embedded in the code should be obfuscated by some techniques such as runtime construction or bit manipulation. This makes it difficult for attackers to discover and manipulate the public key.

4. Analyzing Automated Attacks on In-app Billing Service

The guidelines above mainly focus on the defense against tampering directly with the code of a target application through reverse engineering or manual analysis. Obfuscation is one of good countermeasures because it imposes time-consuming burdens on the attacker. It is considered the attacker would not gain great benefits in comparison to the time and effort consumed to analyze and reverse engineer a specific target application. However, recently some automated attacks on the In-app Billing service has appeared. Instead of taking the manual approach of trying to reverse engineer individual applications, these attacks automatically try to bypass all the payment transactions of every application using the In-app Billing service on the device.

The VirtualSwindle [8] is the first and excellent automated attack in the literature against the In-app Billing service on Android. The attack application runs in the background, and when invoked, attacks every application that performs signature verification on the device itself (not on the remote server). It allows the attacker to access digital content and services without paying for them by subverting the signature verification process. According to the authors, among the 85 popular applications using the In-app Billing, 60% of them employed on-device signature verification, and therefore they were easily attacked successfully. Here we analyze the detailed mechanism of the VirtualSwindle and also discuss why this kind of attack successes in most applications.

The attack flow of the VirtualSwindle is shown in *Figure* 3. The attack emulates and subverts the part responsible for the IInAppBillingService interface in the Google Play app on the device. It works like a proxy between the application using In-app Billing and the Google Play app, and it generates a fake response Intent that makes the calling application believe the payment process was made successfully. In the victim application side, to bypass the signature verification of the purchase data, the attack replaces the standard Dalvik library method java.securty.Signature.verify() with its own method. The replaced verification method returns true indicating success if the input is the fake signature, and otherwise it redirects the call to the original method in order to not break the verification for the rest of the device.

The attack shares a dynamic Dalvik instrumentation approach [13], which is implemented as a *libddi* library, to replace any Dalvik method to an alternative native function provided by the attacker, by abusing the Java Native Interfaces (JNI) layer. In the first step, the com.android.vending process responsible for the Google Play app is hijacked by injecting *libddi*. Then the prepared Dalvik classes for acting as a proxy are loaded into the vending process.

 Copyright ⓒ 2016 SERSC

When the victim application sends a purchase request through the getBuyIntent call, the proxy intercepts and redirects it to the original implementation for the getBuyIntent to get a valid PendingIntent. The answer generated by the Google Play app is then passed back to the victim application. The victim application launches this PendingIntent for checkout, and the proxy also intercepts this request. Because the actual payment is not made at this time, the proxy itself generates a JSON object containing purchase information such as the order ID, instead of receiving it from the remote Google Play server. The proxy then returns the JSON object with a fake signature chosen by the attacker.

After receiving them, the victim application performs signature verification for checking valid payment and, as mentioned above, most of them call the java.security.Signature.verify method for on-device verification. Because the fake signature is not generated with the application's private key, it is necessary to bypass method call. For this purpose, the attack injects a native library into the zygote process beforehand. The library then manipulates the *method struct* of the java.security.Signature.verify so that a native dummy function is called through JNI mechanism instead of the original method. The dummy function is also implemented by the attacker and it always returns true if the input is the fake signature. Once the library is injected into the zygote, it is automatically propagated to every process running on the system.

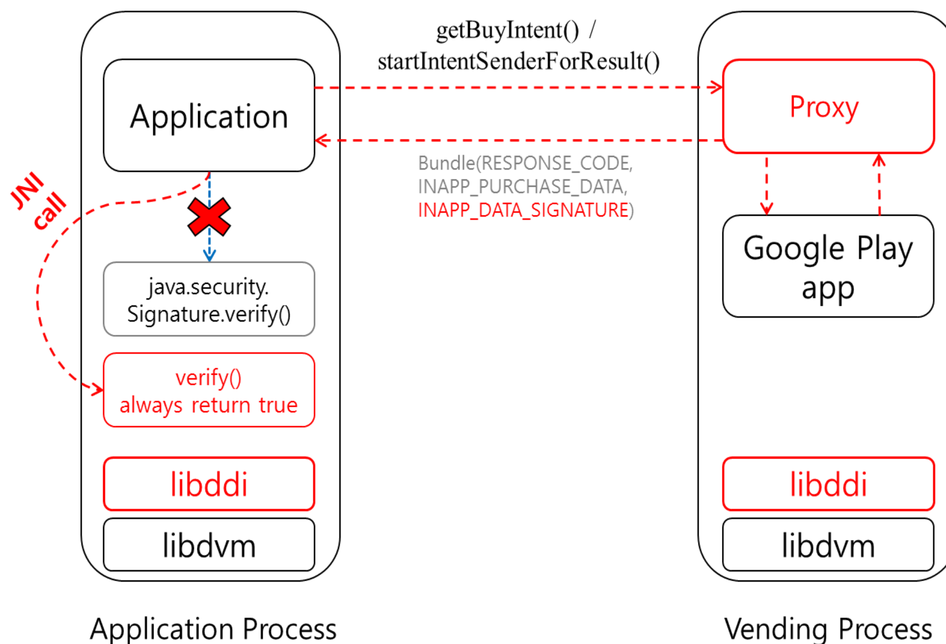


Figure 3. The Attack Flow of the VirtualSwindle

There is another automated attack tool called Freedom [17], which is very popular among the black market users. It is known a lot of game applications has been hacked by this tool. Although the detailed mechanism of the Freedom has not been discovered, our analysis shows that it is very similar to the VirtualSwindle, at least in the way to bypass the signature verification. Like the VirtualSwindle, it can only hack the applications using on-device signature verification. It also performs an automated attack on every application on the device, and it also needs to run a specific launcher before starting attack. Crucially, it fails to attack if the method presented in this paper is applied as you can see in *Figure* 6.

Copyright ⓒ 2016 SERSC

5. Detecting the Bypass of Signature Verification for In-app Billing

The key functionality of the attacks above is to bypass the signature verification process. Because a legitimate payment is not made and the attacker cannot discover the victim application's private key in any way, a valid signature for purchase cannot be obtained. Instead, the call for java.security.Signature.verify method is redirected to the attacker's code returning true though the signature is fake. In other words, if there is a way to detect the bypass trial, the whole attack can be detected and thus prevented.

There are several approaches to detect the bypass trial. Because the *method struct* of the verify method must be modified for the call to be redirected, monitoring of it at runtime can detect the trial. Because some library like the *libddi* must be dynamically injected in the process for the attacks, monitoring and checking the integrity of the process also can detect the trial. However, this kind of runtime monitoring imposes a heavy burden on the system and requires modification of android kernel codes. On the other hand, here we present simple but very effective approaches to detect the bypass.

Let us assume the call for java.security.Signature.verify method is redirected to the attacker's function fake_verify. Signature verification requires three kinds of input parameters: the data to be verified, the victim application's public key, and the signature. Among them, the signature is already replaced to a fake signature and the fake_verify function can easily check whether the input signature is the fake signature. But the other two parameters are not known to the function at runtime. Based on this insight, the main idea of our approach is to test whether the verification works correctly or not after modifying these parameters by intent. The original verify method will judge the signature is invalid because the parameters have been modified, however, the fake_verify function will judge it is valid as usual. Therefore, if the return of the test call is true, we can recognize the signature verification has been bypassed.

The first approach is modifying the data to be verified, and the flowchart for the test is shown in *Figure* 4. The application receives a response Intent containing both the purchase data and the signature after payment process. We chose the orderId field in INAPP_PURCHASE_DATA to be modified because it is uniquely generated per each purchase. A random number is generated and the orderId is replaced to the number. The application then calls the verify method with the modified data. If true is returned, it means the verification was bypassed by the attack and thus the application stops providing item. Otherwise, the application calls again the verify method with the original data. If true is returned at this time, it means a legitimate payment has been made and thus the application provides corresponding item.

Copyright ⓒ 2016 SERSC

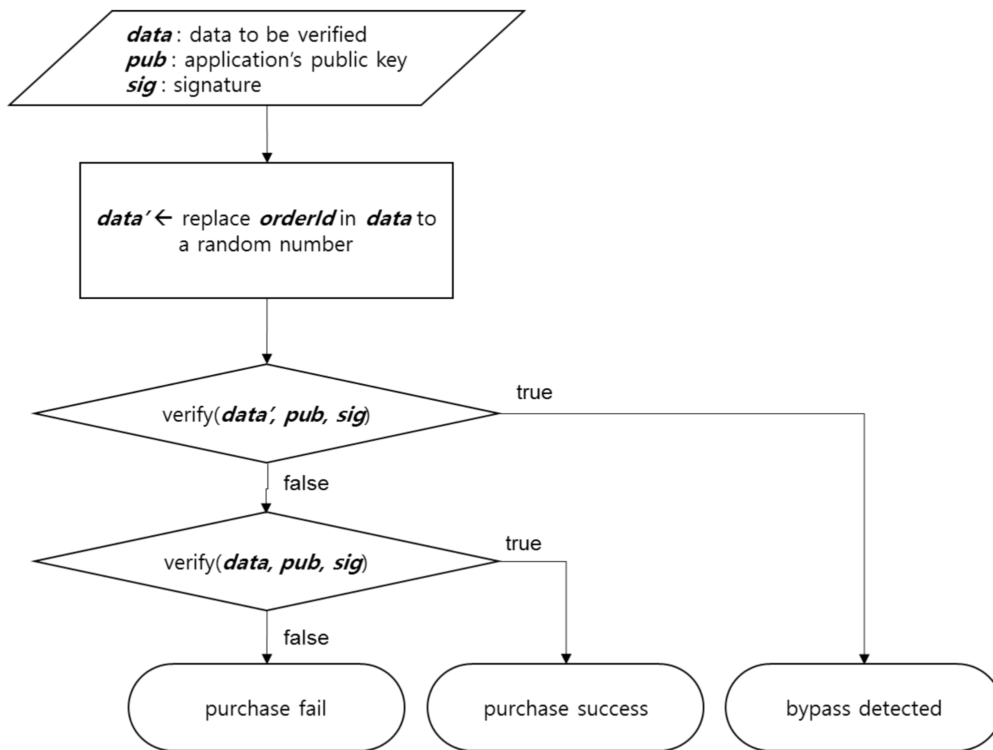


Figure 4. Flowchart for Detection using Modified Purchase Data

This approach assumes the fake_verify function cannot know the original orderId. However, if the attacker improves the attack later, it may be possible the orderId is informed by the proxy through a covert channel. Then the function can pass the test by comparing the two orderIds. The second approach is presented here to detect the bypass even in this case. In this approach, the application generates a fake public key after receiving a response Intent. Then this key is used for the test, instead of the application's public key. The original key is not known to the proxy, and thus the proxy cannot inform it to the fake_verify function. And Google recommends to hide the public key from the attackers. Of course, it may be possible the attacker discovers the public keys of some specific applications by reverse engineering manually per each application. However, we are dealing with automated attacks on every application and there is no way known to find every application's public key automatically. For the test, the application calls the verify method with the fake public key. If true is returned although a fake key is provided, it means the verification was bypassed by the attack and thus the application stops providing item.

The developers can easily apply these approaches in their applications by adding a few codes. The main advantage is that the developers do not need additional permissions and complicated techniques. Furthermore, the Android kernel does not need to be modified for securing the In-app Billing service. We applied our approaches to the sample application TrivialDrive provided by Google to show the correctness of the approaches. We tried to purchase an item after launching the Freedom, and consequently the application could successfully detect that there was an attempt to bypass the payment as can be seen in *Figure* 6.

Copyright ⓒ 2016 SERSC

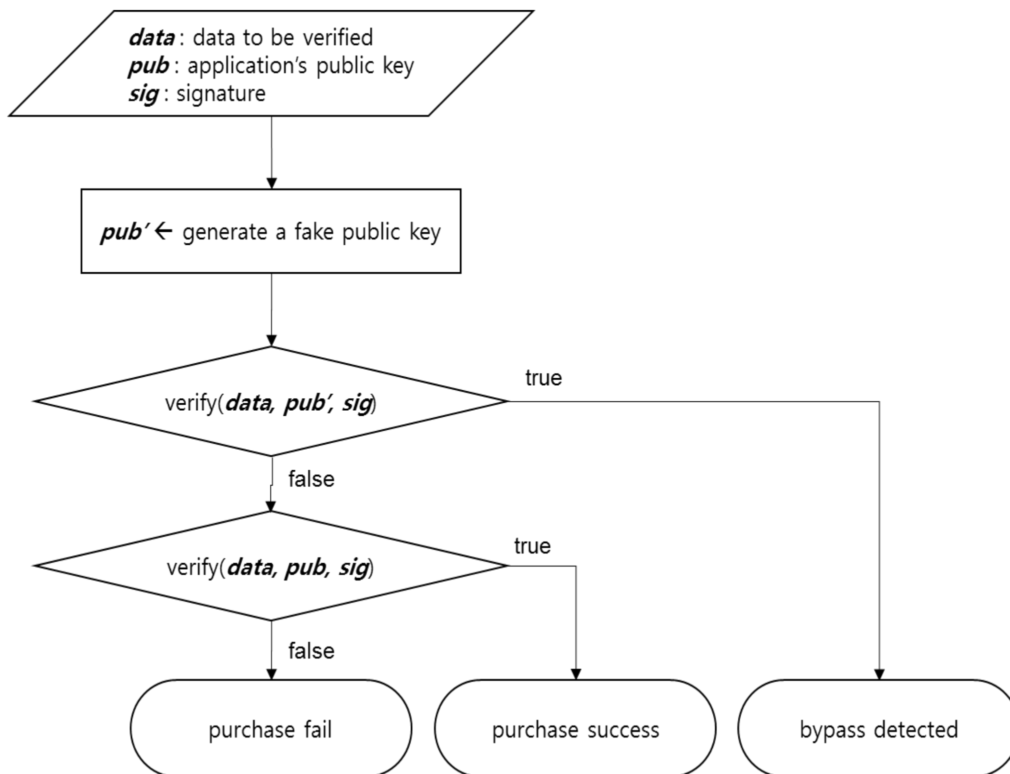


Figure 5. Flowchart for Detection using a Fake Public Key

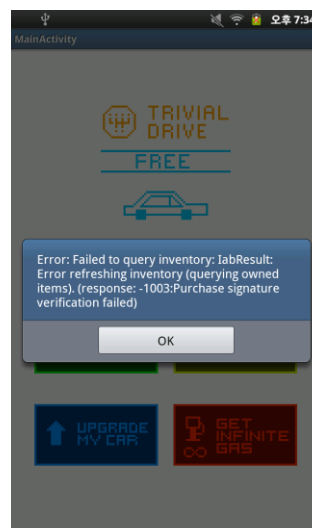


Figure 6. A Screenshot Showing Detection of the Freedom Attack

6. Conclusion

Recent automated attacks on Android In-app Billing service enable the attackers to get valuable contents and services without paying for them. In this paper, we presented a novel approach to prevent this kind of attacks by detecting the attempt to bypass signature verification process. This approach is so simple that it can be applied easily to the applications without modifying the kernel codes. Furthermore, it was shown that this approach prevents existing automated attack Freedom effectively.

Copyright ⓒ 2016 SERSC

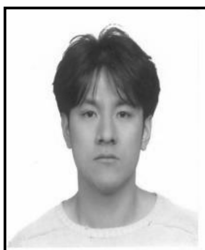
Acknowledgment

This work was supported by Kyonggi University Research Grant 2014.

References

[1] L. Ma, L. Gu and J. Wang, “Research and Development of Mobile Application for Android Platform”, International Journal of Multimedia and Ubiquitous Engineering, vol. 9, no. 4, **(2014)**, pp. 187-198.
[2] Apple, “In-App Purchase for Developers”, https://developer.apple.com/in-app-purchase/, **(2010)**.
[3] Google, “In-app Billing”, http://developer.android.com/google/play/billing/index.html, **(2011)**.
[4] A. Kurniawan, Doni Nathaniel Pranama, Junius, and M. Megasari, “Droidglance: Network Topology Generator and Device Security Assessment Application on Android Mobile Device”, International Journal of Software Engineering and Its Applications, vol. 8, no. 5, **(2014)**, pp. 189-204.
[5] S. Biswas, W. Haipeng and J. Rashid, “Android Permissions Management at App Installing”, International Journal of Security and Its Applications, vol. 10, no. 3, **(2016)**, pp. 223-232.
[6] D. Reynaud, D. Song, E. W. Tom Magrino and R. Shin, “FreeMarket: Shopping for free in Android applications”, ISOC Network and Distributed System Security Symposium (NDSS), February, **(2012)**.
[7] Google, “In-app Billing Security and Design”, http://developer.android.com/google/play/billing/billing_best_practices.html, **(2016)**.
[8] C. Mulliner, R. William and E. Kirda, “VirtualSwindle: an automated attack against in-app billing on android”, Proceedings of the 9th ACM symposium on Information, computer and communications security, Kyoto, Japan, **(2014)** June 4-6.
[9] HTC, “Unlock Bootloader”, http://htcdev:com/bootloader/, **(2012)** November.
[10] CLShortFuse, “SuperOneClick”, http://forum:xda-developers:com/showthread:php?t=803682, November, **(2012)**.
[11] E. Lafortune, “ProGuard”, http://proguard.sourceforge.net, **(2004)**.
[12] S. Clowes, “Injectso - Modifying and Spying on running processes under Linux and Solaris”, Balckhat Europe 2001, http://www:blackhat:com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes:ppt, **(2001)**.
[13] J. Freeman, “Substrate for Android”, http://www:cydiasubstrate:com/.
[14] B. Davis, B. Sanders, A. Khodaverdian and H. Chen, “I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications”, In Workshop on Mobile Security Technologies MoST), **(2012)** May.
[15] R. Xu, H. Saidi and R. Anderson, “Aurasium: Practical Policy Enforcement for Android Applications”, In USENIX Security Symposium, **(2012)** August.
[16] L. Ki. Yan and H. Yin, “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis”, In USENIX Security Symposium, **(2012)** August.
[17] Unknown. Freedom. https://in-appstore.com/?page_id=12

Authors



Heeyoul Kim, received the B.E. degree in Computer Science from KAIST, Korea, in 2000, the M.S. degree in Computer Science from KAIST in 2002, and the Ph.D. degree in computer science from KAIST in 2007. From 2007 to 2008, with the Samsung Electronics as a senior engineer. Since 2009 he has been a faculty member of Department of Computer Science at Kyonggi University. His main research interests include cryptography & security such as secure group communication and clod computing security.

Sang-won Kim, received the B.E. degree in Computer Science from Kyonggi University, Korea, in 2012, the M.S. degree in Computer Science from Kyonggi University, Korea, in 2014. His main research interests include mobile security and in-app billing security.

 Copyright ⓒ 2016 SERSC