The Far Side of Mobile Application Integrated Development Environments

Christos Lyvas[1(✉)], Nikolaos Pitropakis[2], and Costas Lambrinoudakis[1]

[1] Department of Digital Systems, University of Piraeus, Piraeus, Greece
{clyvas,clam}@unipi.gr
[2] School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia
pitropakis@gatech.edu

Abstract. Smart phones are, nowadays, a necessity for the vast majority of individuals around the globe. In addition to the ubiquitous computing paradigm supported by such devices, there are numerous software applications that utilize the high computational capabilities that they offer. This type of software is a vital part of what is known as e-Commerce, with a variety of business models proposed and implemented. Lately, a new era of free-ware mobile application has arisen with paid features and promoted content in them. Piracy is not only the weakest point of software's financial ecosystem for conventional computing systems but also for smartphones. Actions like replication, redistribution and licensing violations can cause financial losses of colossal extent to their creators. Mobile applications also introduce the following peculiarity: They are distributed through predefined channels (Application Stores) owned by mobile operating system vendors such as Apple, Google and Microsoft. In this research we present several scenarios where cracked and modified applications can be freely used into every non jailbroken iOS device. Moreover it is demonstrated that not even in strict mobile environments, such as Apple's, end-users should be considered as trusted entities from application developers by default.

Keywords: Application integrity · Application reverse engineer · Application security

1 Introduction

Ubiquitous computing is certainly a breakthrough. Two decades ago no one could imagine that he would be able to carry in his pocket mini computers with extremely high processing power and capable to provide internet access on demand. In a very short time smartphones have established their position in the mobile phone market and have become the accessory that almost everyone uses constantly either for work or for entertainment.

After Apple launched the first iPhone, Google and Microsoft followed, offering new smartphones and smart devices to the public. Each one of them promised to improve our living quality and has developed software that was advertised as secure and stable. During the last couple of years biometric sensors, such as fingerprint sensor and iris

S. Katsikas et al. (Eds.): TrustBus 2016, LNCS 9830, pp. 111–122, 2016.
DOI: 10.1007/978-3-319-44341-6_8

sensor, were introduced as an extra security level for the protection of the user. However in practice most mobile applications do have bugs or other vulnerabilities that can be exploited by malicious parties in order to harm the user. A very interesting debate for academics and users is the following very simple question "which mobile platform among iOS, Windows Phone and Android is more secure?" Clearly, there is not an easy answer, especially since there are a lot of similarities in terms of the security mechanisms adopted by each platform as all of them follow similar technological paths.

The main objective of this paper is to evaluate the mechanisms that the iPhone operating system features in order to check the trustworthiness of the applications. Cracked or prepackaged applications can run on Android devices by simply modifying the default configuration settings of the mobile phone. This is also true for the Windows Phone, where untrusted applications can be deployed into any developer unlocked Windows Phone using the aid of an application deployment tool running on a PC. For Apple devices the most popular method for executing untrusted applications is the Jailbreak procedure that bypasses the code signature mechanism and instantly voids the guarantee. The IOS's Mandatory Code Signature mechanism aims to ensure that an application can be executed only if its code has been signed by a trusted party [1]. Thus, prior to an application's execution, an internal kernel check verifies that the code loaded into the virtual memory contains a valid signature and can, thus, proceed with the execution [5]. Any modification of a signed executable results in the invalidation of the entire file/application. The Mandatory Code Signature mechanism can prevent cracked applications of being executed on trusted devices (not Jailbroken) while at the same time trusted malformed or malicious applications that change their executable code or behave like droppers [6] cannot execute their payload on non-modified iOS devices since the executable code does not have a valid signature. The Jailbreak procedure disables the kernel code sign check, allowing those devices to run pseudo signed code.

When a developer publishes an application, Apple ensures that the application is fully functional, bug free and that it does not violate Apple's security regulations [4]. Following the evaluation, the application is released in the iOS App Store and Mac iTunes. These applications can execute on any iDevice (iPhone, iPod, iPad) [25] since they have been signed with Apple's s private key. This mechanism, as part of the Mandatory Code Signature scheme explained before, ensures that applications with illegal content or malicious payloads will not be executed on trusted devices.

Moreover, all the executables of the applications published in the App Store are code protected with encrypted segments by Apple (connotation of ARMv7-A and ARMv8-A Mach-O compatible for both 32 and 64 Bit ARM architectures) in order to prevent any reverse engineering and replication attempts. This kind of obfuscation however is not effective during runtime dynamic analysis, and thus an attacker can obtain the unencrypted version of an executable when it is loaded into the memory [21].

In this paper we describe costless methods based on iOS Integrated Development Environment, where any user can overcome the code signature mechanism and execute cracked or prepackaged applications onto new iDevices. Moreover, the impact of this ability is highlighted as it could lead to integrity violation of legitimate applications' transactions, such as in app purchases [18], with significant financial consequences for their creators.

The rest of the paper is organized as follows. Section 2 provides an overview of the related work and a comparison with the presented approach. In Sect. 3 the anatomy of an iOS application and its embedded mechanisms is explained, while Sect. 4 describes the provisioning model for iOS devices. Section 5 introduces practical attacks on paid and free applications. Section 6 presents our thoughts for mitigating the attacks as well as pointers for future work and specifically on how the proposed method can be further extended in order to achieve a more in depth investigation of the iOS platform.

2 Related Work

This research work has emphasized into Apple's iOS security ecosystem since it is undoubtedly one of the stricter mobile platforms. Android and iOS cover 92.95 % of mobile market for the last 4 years with an average of 79.1 % and 13.8 % respectively [26]. Nonetheless, an interesting fact about those mobile platforms, is that iOS users spend much bigger amounts of money to purchase applications or features on them, in comparison to Android users [27].

Despite the fact that the huge percentage of software piracy is happening on Jailbroken devices, there are a lot of threats against applications' integrity onto new iDevices also. By combining a series of weaknesses in the development chain of iOS applications it is clearly demonstrated that the entire business model of Apple's App Store is not only threatened by Jailbreak Development but also it cannot mitigate software piracy.

The majority of research work on the iOS application security model has tried to attack the security mechanisms through remote exploits or local privilege escalation vulnerabilities, using memory corruptions and memory leaks with a variety of methods (Return Oriented Programming, Jump Oriented Programming, Heap Spraying etc.).

Wang et al. in [16] have managed to bypass Apple's App Store review process and publish vulnerable applications, while they propose ways to remotely exploit them based on iOS Framework vulnerabilities. In another paper [15] they propose ways to inject malicious developer-signed applications to non-jailbroken iOS devices by intercepting USB and Wi-Fi connection between iDevices and infected computers. Finally, they claim that infected non Jailbroken devices could act as botnets.

A survey by Zheng et al. [14] evaluates all possible ways through which an application can be distributed to a non Jailbroken iOS device signed with a variety of several different paid certificates (Developer or Enterprise). During their research they develop a framework to identify threats induced by the usage of vulnerable iOS private API (undocumented application programming interfaces) functions. They evaluated 1408 private enterprise applications and they discovered several vulnerabilities and privacy leaks in their payloads. Finally, they claim that non jailbroken iOS devices can run cracked iOS applications if the applications have been signed with valid certificates.

A methodology for repackaging iOS applications executed on new 32Bit iDevices was published by Livitt [22]. Specifically, a developer with an enrolled Developer Account, with an annual cost of $99, can generate provisioning profiles (Certificates) suitable to resign App Store Applications through Apple's Developer Portal [24]. After

performing tests with the tool proposed [23], it was concluded that it was only compatible with 32 Bit iDevices.

The novelty of the work presented in this paper (Table 1) lies on the fact that it demonstrates how someone can use any type of application (freeware or paid) freely on any kind of non Jailbroken iDevice. The above procedure is independent of the iOS version and the user only needs his/her Apple ID. Furthermore, additional ways that allow users to access premium features and bypass applications' additional security checks are discussed, while additional developer features can be unlocked and used for reversing third party applications such as automatic network monitoring, memory allocation debugging and automatic memory leak inspection. Finally, it has been demonstrated that in some app purchase cases it is feasible to bypass the payment by modifying the application's configuration files and accessing premium features by replacing legitimate with arbitrary values. This kind of access into third party application files is possible because they were supposed to run onto a new iDevice, owned by the developer who signs them (signed with developer certificate), for testing purposes.

Table 1. Method comparison

		Rethinking & Repackage iOS Apps [22]	Enpublic Apps [14]	The Far Side Of iOS IDE
Member Account	Apple ID (Free)	✗	✗	✓
	Enrolled Developer or Enterprise	✓	✓	✓
Cracked Paid	Objective-C	✗	✓	✓
	Swift	✗	?	✓
Hooked Free	Objective-C	✓	✗	✓
	Swift	✗	✗	✗
Device Architectures (Compatibility)	32 Bit	✓	?	✓
	64 Bit	?	?	✓
In App Purchases Bypass	Configuration File Modification	✗	✗	✓

3 Anatomy of iOS Application

iOS applications can be downloaded through iTunes for conventional devices (Mac, PC) and via App Store for iDevices (iPhone, iPod, iPad) with an active Apple ID account being necessary in all cases. An iOS application is a Zip archive, containing several folders and files. Every application contains a property list file with information about the downloaded ipa (Apple application archive) [21] file, such us which Apple ID was used for the purchase, the version of the application, date of creation etc. Another folder placed in every ipa archive is the Payload folder which carries the application bundle in app file extension. Every legitimate application container carries several application icons, images and files for the application's user interface. In order an application to run in a non Jailbroken iDevice it must contain a valid property list file placed inside the folder _CodeSignature. This property list contains hashes of every file inside the app container in Base64 format [21]. The property list file named "info" inside the application container carries information about the executable version, the unique name of the application (Bundle ID), URLs for the inter app communication mechanism [2] and the publisher's identifier. The executable file of an application is a connotation of ARMv7-A and ARMv8-A Mach-O executables of the production source code. Any additional extension or plugin of the application is most of the times placed inside the bundle folders. For applications developed with swift framework an additional folder exists into the app container which carries the necessary dynamic libraries for the application's execution. Figure 1 depicts the structure that has been already described.

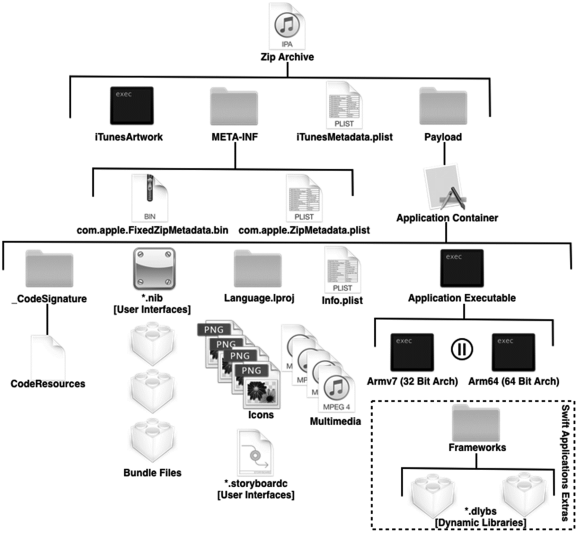

Fig. 1. IPA container

4 Provisioning Profiles

The code sign procedure is based on Public Key Infrastructure implementation which ensures the integrity of the signed objects and the identity of the parties involved. Theoretically, every developer has a pair of public and private RSA 2048 Bit key. As an authority, Apple creates developer certificates based on developers' public keys, then uses the SHA-256 hash algorithm to hash the certificate and eventually signs it with its private key. The generated developer certificate has as its only purpose to sign applications. When a developer creates an application via Apple's development tool Xcode and he/she has attached an iOS device through a USB cable, he/she is allowed to deploy the application to the iDevice [17]. Automatically after the compilation, an app container is generated containing the necessary files in order to be executed onto the iDevice. An additional file is generated with extension mobile provision. This specific file is a certificate in the form of a property list that declares the Developer ID which is the creator of the application, the Bundle ID (Unique Name) of the generated application, the target device UDID (Unique Device Identifier), the developer's public key with which the application has been signed and the permissions of the application. When a developer needs to test the application onto an iDevice he/she must first accept the developer's certificate as being legitimate through the settings of the mobile device. Using this implementation the parts of the application that have been encrypted with the developer's private key can be decrypted through the corresponding public key into the provisioned iDevice [9, 10]. The trust of this procedure is sealed with the valid certificate issued by Apple. The signed executable contains an embedded property list file, known as *entitlement*, which defines the application's Bundle ID, the Developer's ID and the permissions of the application. The entries of that file is a subset of the mobile provision's file, as explained before.

5 Attack Types

The objective of this work was to evaluate the tolerance of iOS's application code protection mechanisms. Section 5.1 demonstrates all the necessary steps to execute cracked paid applications in non jailbroken iDevices. In addition to that, we were able to extend the functionality of various applications by injecting malicious libraries as add-ons into their original bundle and deploy them also into non jailbroken iDevices. For the above purposes several 32 and 64 bit iOS devices have been used with various versions of iOS 9. Our methodology is not automated. Every step is manually driven. Automating these procedures is out of the scope of this paper.

5.1 Replication

In the experiments several legitimate paid applications, available on Apple's App Store, have been used together with several cracked application from various unofficial app stores, developed with both Objective-C and Swift programming languages. The method

for loading them onto a non jailbroken iDevice consists of the following steps (illustrated in Fig. 2):

1. User must first install any application legitimate (installed via iTunes or App Store) or cracked one (3rd party repos) into a Jailbroken iDevice.
2. After having installed a legitimate application onto a 32-bit Jailbroken iOS Device, we bypass the encryption of the application's executable by dumping the decrypted parts loaded in the virtual memory to an ARMv7-A Mach-O file. Then, we patch the decryption flag. The entire procedure has been carried out using the LLDB Debugger [20]. Application's executable decryption can also be done by automatic tools [29, 30].
3. Following the previous step, we extract the generated executable from the iDevice.
4. Then, we replace the original executable file of the app container with the cracked one. Then we modify the Bundle ID (Unique application name) of the original application listed into the info plist file inside the container of the application file, with a new name that consists of the original application's name and a random suffix. The random suffix that was utilised serves to overcome the fact that every Bundle ID is reserved and cannot be re-used. It should be stressed that the aforementioned replacement of the Bundle ID will not work for applications with iCloud or Game Center extensions.
5. Every iDevice owner is obliged to create an Apple ID account in order to have access to iTunes, App Store and iCloud. An iDevice allows a limited number of accounts per device to be created without the use of a credit card. An attacker can create as many as possible Apple accounts as he/she wants and declare them as developer accounts without paying the annual fee to activate them. As a result, the fake Apple accounts remain inactive and although they cannot be used for publishing applications to the App Store they can be used for executing application that are under development to any new iDevice. The exploited vulnerability has been based on the developers' ability to deploy their own testing applications to new iOS devices, through Xcode, without any cost but by simply using an Apple ID registered to Apple's Developer Program without enrolment. Consequently, we are able to create decoy application with the same Bundle ID as that of the modified application's (Legitimate Bundle-ID + Suffix) and bind it with the developers account. We let Xcode to automatically generate a suitable team provisioning profile in order to deploy the decoy application into a non jailbroken iDevice [12].
6. Before the user launches the decoy application for the first time, he/she must accept the developers team provisioning profile in the iDevices's Preferences.
7. At this point we are able to dump the entitlement of the generated executable and merge it with the entitlement of the original one.
8. Then the Xcode tool set [3] was employed to resign the decrypted executable with our valid developer certificate, based on the entitlement of the decoys application executable. It is clear that the bind between a valid certificate, the Developer's ID, the UDID and the Bundle ID of the application, is not enough since Apple cannot ensure that the developers actually will sign only their own legitimate applications. By resigning an application the Code Signature folder is regenerated and that allows the application to be deployed in an iDevice that has approved the developer's public

key. Finally the signed cracked application has been deployed onto a non jailbroken iDevice by cheating Xcode in the sense that the cracked application has been generated by the owner of the certificate that signs it. Due to the backward compatibility of ARM processors we were able to execute the decrypted 32 Bit armv7 executable (generated by the 32 Bit architecture of Jailbroken iPhone 5) to new iDevices with 64 and 32 Bit architectures respectively.

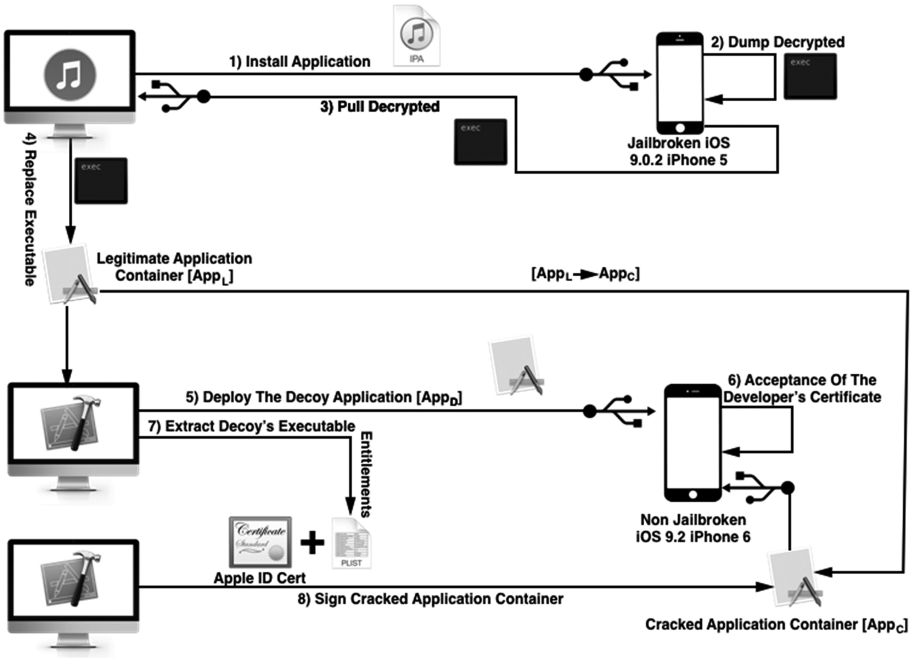

Fig. 2. Replication method

5.2 Malicious Payload Injection

Another issue that affects applications developed with the Objective-C language is the ability to hook functions of application's classes as described by Livitt [22]. After the decryption and extraction of an iOS application's executable (Step 1 Fig. 3), an attacker can reverse engineer it through static and dynamic analysis and discover the usability and functionality of its functions. Thus the attacker can take advantage of the Objective-C [13] language method calling to create dynamic libraries (Step 2 Fig. 3) and hook application's class functions and modify the passing and return values or even inject malicious payloads to them. The most suitable tool for this kind of extensions is the Theos framework [11]. This tool in combination with the iOS Software Development Kit and Cydia Substrate framework [19] is able to generate hooking dynamic libraries. For the purposes of our research we used an ARMv7-A image of Cydia Substrate suitable for both 32 and 64 Bit iOS 9 iDevices and we statically linked the Cydia Substrate to

the generated dynamic library (Step 4 Fig. 3). Then we statically linked it into the cracked executable (Step 3 Fig. 3) and place it inside the application container (Step 5 Fig. 3). Due to the additional modifications it is necessary to resign (Step 6 Fig. 3) the cracked executable and the additional dynamic libraries, with the entitlements of a decoy application as Sect. 5.1. Having the ability to hook Objective-C iOS application's class functions, an attacker can modify an application's behavior, bypass security checks, compromise application's transactions integrity and extend functionality in order to unlock premium features.

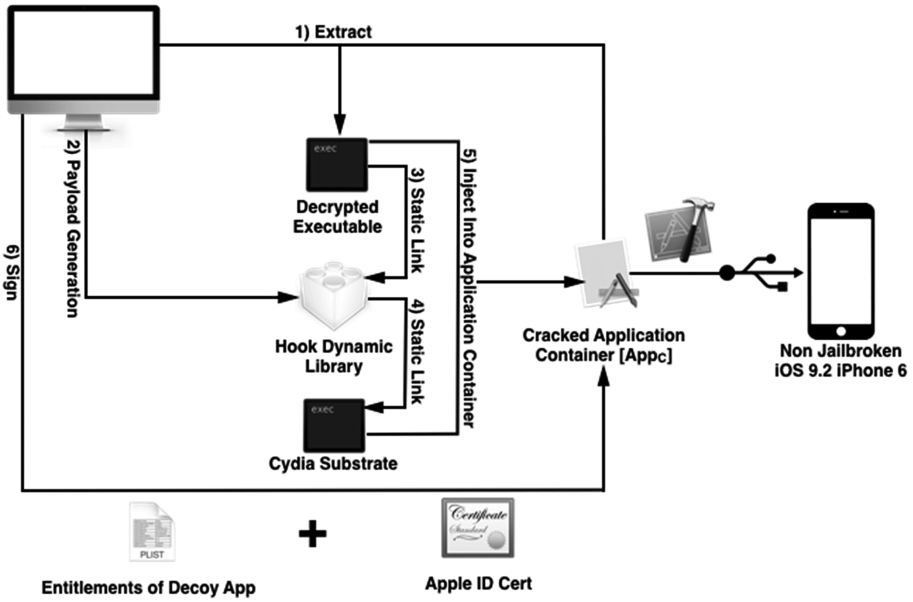

Fig. 3. Library injection

The aforementioned procedures can be performed by any owner of a non Jailbroken iOS 32 Bit or 64 bit Device with a free registration to the Apple Developer Program without enrolling his/her Apple ID and with access to a Mac or to a virtual machine of Mac OS X with Xcode installed. A further impact of signing third parties' applications as 'under test' ones, is that an attacker can unlock several developer features such as the ability to inspect memory allocations and automatically investigate and debug memory leaks throw default system tools preinstalled into any OS X.

6 Conclusions

Both cases may lead to serious financial impacts in the business model of paid and free (with in app purchases features) applications. From an economic standpoint, App Store is the largest digital distribution platform for mobile apps with the total amount of revenue from app sales since 2008 being at approximately 15 billion of United States

Dollars [7]. The use of a functional cracked application deprives the developers of the profit before taxes, which is equal to 70 % of the application's price. Also, there is a loss for Apple which amounts to the rest 30 % of the sale [28]. We were able not only to run paid and repackaged applications freely into non jailbroken iDevices but we were also able to have full access to their configuration files because we sign them as testing applications and gain paid features and bypass Apple in app purchase model by modify their data.

Running Apple ID signed applications onto not modified iDevices enlarge the attack surface of iOS platform because in combination with exploitable memory corruptions and Kernel vulnerabilities Jailbreak developers can deploy their own vulnerable apps in order to directly attack the iOS Kernel. In this paper we leverage the opportunity for unenrolled iOS developers to run freely their under developing application into their iDevices for test purposes and we prove that cracked and repackaged applications can be executed freely into every non Jailbroken devices regardless the version of the operating system.

The immediate revocation of non enrolled developer code signature certificates will only reduce the ability of iOS device owners to use cracked or malformed application to their devices, and not to eliminate that malicious activities because of the alternative equivalent methods accomplished that with enrolled developer and enterprise accounts [8]. The only way that this type of threat can be eliminated is by robust obfuscation for any generated application's executable. Another common vulnerability we faced during our research was the lack of encrypted values into applications file settings which gave us the ability to modify values related with vulnerable in app purchases implementations. Moreover it is recommended for application developers to redevelop immediately the Objective-C applications available in the App Store to their equivalent Swift editions and for Apple the design of a pure Swift framework for all the iDevices operation system. Our research is based on framework vulnerabilities and security mechanisms implemented in mobile applications. Consequently, we aim to extend our research for Android and Windows Phone applications. For Android application a malicious payload is able to be injected into a repackaged application container with a crafted C/C++ library or with Dalvik byte code injection. Similar to Android we will try to generalize those methods to Windows Phone's Applications to inject .NET assembly code into them in order to evaluate the possibility of creation prepackaged tweaked applications for this platform. Our final objective is to categorize common vulnerabilities in applications and ways they can be exploited based on the mobile platform they are implemented in order to suggest user space integrated methods for application integrity protection suitable for any mobile operating system.

Finally, the above financial and statistical data we provided are because of the seriousness of the attack and the potential losses if an escalated attack against paid or vulnerable credit based in app purchase implementations could be done.

References

1. Code Signing Guide: Code Signing Overview. Apple Inc., 23 July 2012. Web 20 June 2016
2. App Programming Guide for IOS. Inter-App Communication. Apple Inc., 16 September 2015. Web 20 June 2016
3. OS X Code Signing In Depth: Technical Note TN2206. Apple Inc., 28 July 2015. Web 08 Dec 2015
4. App Store Review Guidelines: Apple Developer. Apple Inc., 20 June 2016
5. IOS Application Security: IOS Security 2015.2, pp. 18–19. Apple Inc., 9 September 2015. Web 10 Dec 2015
6. Kwon, B.J., Mondal, J., Jang, J., Bilge, L., Dumitras, T.: The Dropper effect insights into malware distribution with downloader graph analytics. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS 2015 (2015)
7. Forbes: Forbes Magazine, 11 January 2015. http://www.forbes.com/sites/anthonykosner/2015/01/11/apple-app-store-revenue-surge-and-the-rise-of-the-freemium/. 09 Dec 2015
8. Choosing a Membership - Support. Apple Inc., 09 December 2015. https://developer.apple.com/support/compare-memberships/
9. App Distribution Guide: Exporting Your App for Testing (iOS, TvOS, WatchOS). Apple Inc., 29 May 2016. https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/TestingYouriOSApp/TestingYouriOSApp.html#//apple_ref/doc/uid/TP40012582-CH8-SW1. 20 June 2016
10. Code Signing Guide: About Code Signing. Apple Inc., 23 July 2012. https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html. 20 June 2016
11. Theos: Unified Cross-platform Makefile System. Github Repository, 7 February 2016. https://github.com/DHowett/theos. 20 June 2016
12. App Distribution Guide: Launching Your App on Devices. Apple Inc., 29 April 2016. http://developer.apple.com/library/mac/documentation/IDEs/Conceptual/AppDistributionGuide/LaunchingYourApponDevices/LaunchingYourApponDevices.html. 20 June 2016
13. Objective-C: Runtime Programming Guide. Messaging, Apple Inc., 19 October 2009. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtHowMessagingWorks.html. 20 June 2016
14. Zheng, M., Xue, H., Zhang, Y., Wei, T., Lui, J.C.S.: Enpublic Apps. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS 2015, pp. 463–474 (2015)
15. Wang, T., Jang, Y., Chen, Y., Chung, S., Lau, B., Lee, W.: On the Feasibility of Large-Scale Infections of IOS Devices. In: 23rd USENIX Security Symposium, pp. 79–93 (2014). Web 10 Dec 2015
16. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on iOS: when benign apps become evil. In: 22nd USENIX Security Symposium, pp. 559–572 (2013)
17. App Distribution Guide: Maintaining Identifiers, Devices, and Profiles. Apple Inc., 29 April 2016. https://developer.apple.com/library/mac/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingProfiles/MaintainingProfiles.html#//apple_ref/doc/uid/TP40012582-CH30-SW26. 20 June 2016
18. In-App Purchase Programming Guide: About In-App Purchase. Apple Inc., 21 October 2005. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/StoreKitGuide/Introduction.html. 20 June 2016
19. Cydia Substrate: The Powerful Code Modification Platform behind Cydia. SaurikIT LLC (2014). http://www.cydiasubstrate.com/. 20 June 2016

20. The LLDB Debugger: LLDB Homepage. LLVM Project, 20 June 2016. http://lldb.llvm.org/. 20 June 2016
21. Levin, J.: Mac OS X and IOS Internals: To the Apple's Core. Wiley, Indianapolis (2013)
22. Livitt, C.: Rethinking & Repackaging IOS Apps: Part 2. Bishop Fox, 4 May 2015. Web 2 Dec 2015
23. Theos and Cycript for Non-jailbroken IOS Devices. Github Repository, 17 August 2015. https://github.com/BishopFox/theos-jailed. 20 June 2016
24. Apple Developer: Apple Inc. (2015). (18 Dec. 2015)
25. Passary, A.: Apple IOS 9: Here's A List of Eligible Devices. TechTimes Inc., 10 June 2015. http://www.techtimes.com/articles/59076/20150610/apple-ios-9-heres-a-list-of-eligible-devices.htm. 20 June 2016
26. IDC: Smartphone OS Market Share. IDC Research, Inc., August 2015. www.idc.com. 8 Dec 2015 http://www.idc.com/prodserv/smartphone-os-market-share.jsp
27. McCracken, H.: Who's Winning, IOS or Android? All the Numbers, All in One Place | TIME.com. Time Inc., 16 April 2013. http://techland.time.com/2013/04/16/ios-vs-android. 20 June 2016
28. From Code to Customer: Apple Developer Program. Apple Inc (2016). https://developer.apple.com/programs. 20 June 2016
29. Clutch: Fast IOS Executable Dumper. Github Repository, 15 June 2016. https://github.com/KJCracks/Clutch. 20 June 2016
30. Esser, S.: Dumped Encrypted. Github Repository, 13 February 2014. https://github.com/stefanesser/dumpdecrypted. 20 June 2016