An analysis on iOS Jailbreak

1. Introduction

• *What is iOS Jailbreak*

iOS is Apple's mobile operating system, which is derived from Mac OS X, with which it shares the Darwin foundation, and is therefore a Unix-like operating system. Being developed originally for the iPhone, it then has been used on the iPod Touch, iPad and Apple TV as well. So in this report iOS is specifically refer to the mini-operation system that run on the iDevices (iPhone, iPod, iPad and Apple TV. In this little apple operation system, there are four abstraction layers: the Core OS layer, the Core Services layer, the Media layer, and the Cocoa Touch layer, which in total will roughly use 500 megabytes of the devices' storage.

For security and commercial reasons and considerations, Apple does not permit the OS to run on third-party hardware and also has a limitation on the usage of iOS on these iDevices. Therefore iOS has been subject to a variety of different hacking methods focusing on attaching functionality not supported by Apple. This hacking procedure is called iOS Jailbreak.

• *Why to Jailbreak—a Self-redemption*

Jailbreaking is a process that allows these iDevices users to gain the root access to the command line of the iOS operating system, in order to remove usage and access limitations imposed by Apple. Once jailbroken, iPhone users are able to download extensions and themes that are unavailable through the App Store (via installers such as Cydia) and perform other tasks that are not possible on store-bought devices, including installing non-Apple operating systems such as Linux, running multi-task on old version of iDevices (the new Generation of store-bought devices includes this function). Through the authentication server developed by Aurik (a Ph.d student from UCSB) built up to sign old firmware of iOS (which will be later addressed in this report), Cydia creator Jay Freeman estimates that over 10% of all iPhones are jailbroken.

Jailbreaking an iPod or iPhone in the United States is legal 'fair use', and does not violate copyright laws defined by the Digital Millennium Copyright Act. However some jailbreakers also attempt to illegally share paid App Store applications, which has caused some strife within the jailbreaking community, as it was not the original motivation of jailbreaking and is obviously illegal.

2. 'Identifying' yourself after Jailbreaking

Smart phones all have their own mechanism on preventing arbitrary code running on the little machine. For security and commercial reasons, the operation system and high-level application codes all require official signature before plug in to the machine. iPhone from Apple also has several mechanism for this consideration. Below is a detail explanation on how iPhone organize its iOS firmware authentication and the tricks that the hackers implementing to by pass this remote authentication mechanism.

• *iOS firmware authentication mechanism*

SHSH Blob is short for **S**ignature **H**a**SH**. SHSH Blob is a 128-byte RSA signature used to verify the validity of firmware on newer Apple iOS devices. SHSH Blobs is based on a challenge-response authentication mechanism to verify the validity the old firmware, whenever the iDevices want to upgrade or downgrade its version of iOS.

Exclusive Chip ID, or **ECID** is a 16-digit hexadecimal number used to uniquely identify Apple iDevices. The unique chip identification ECID for each iDevice is now widely considered as a new security feature from Apple, implemented to stop Jailbreaking of future firmwares.

How can it stop the Jailbreaking? Combined with the signature hash of the firmware, SHSH, the ECID of a device is used by Apple, as a challenge key in its challenge-response authentication protocol. This authentication is done in order to perform digital signing on iOS software. The challenge key comes in a combination of a hash of the firmware and the Exclusive Chip ID of the device. Then the response from Apple is the SHSH itself, the digital signature required to validate the firmware.

In a practical view, during the restore process, users see "Verifying restore with Apple...", during which period a challenge/response protocol is launched between the iDevice and Apple authentication server: a "partial digest" of the firmware files being used is sent to a server, which can then decide to sign off on the result or not.

iTunes in the authentication process is taken as the signing software whenever the iOS software upgrade or restore. When you restore your iDevice, iTunes contacts the Apple servers, to generate signatures, just for your device, based on the ECID and the SHSH, with which the iTunes is only authorized to restore the firmware version that addressed in SHSH. Therefore the user will be temporarily out of exploits since it is the most recently patched firmware and at the same time no Jailbreak work can be done as well. Because Jailbreak will work only on those firmware versions that have certain types of exploits being found already. As the Jailbreaking groups need time to dig new holes in the recently updated firmware, the firmware version is somehow safe for a period (unjailbreakable). However, this sort of save is obviously the Jailbreaking user unwilling to see. Some detail exploits will be introduced in later sections.

- ***By pass the firmware authentication and identify yourself***

Because the challenge key (ECID+SHSH) is static, a saved copy of the signature can be used in a replay attack to trick the signer iTunes into validating an old firmware. Why we are requiring the validity of old firmware? Since Apple will only sign the most currently published version of firmware that is it stops signing any lower version of iOS on corresponding iDevice. Whenever an iDevice requires a restore officially, it can only restore the more updated firmware version. However as addressed in 2.1 the Jailbreak will work only on the old unpatched firmware versions. This official firmware update is initially considered as Jailbroken iDevices' nightmare.

However, Aurik is working in one of the Jailbreaking group has developed a server that user can point the iTunes to and it will not only authenticate firmware versions that Apple no longer signs. It also saves information during the authentication and will allow you to downgrade later if user's firmware is updated accidentally by the iTune, or the user has no knowledge of Jailbreaking before and naturally update to the latest version.

Aurik has constructed a server, which mimics the functionality of Apple's signature server. The main difference is that this server is using "on file" results rather than apple's live challenge and response. This self-constructed server will play the man in the middle (MiM). It replays the saved key file for challenge and response and then trick iTunes onto believing that he is talking to the official Apple authentication server.

However, how to make iTunes into believing in the self-constructed server is still a problem. Actually, the hosts file being kept in several systems, like the file C:\Windows\System32\...\hosts\etc\hosts

(Windows) or /etc/hosts (Mac OS X) renders a perfect solution to help playing the trick by a simple local manipulation. Only an attachment of on entry like 111.222.333.111 gs.apple.com to the hosts file is enough to let iTunes redirect the requests to the constructed server with the address 111.222.333.111.

The hosts file is one of several system facilities to assist in addressing network nodes in a computer network. It is a common part in an operating system's Internet Protocol (IP) implementation, and serves the function of translating hostnames into numeric protocol addresses, namely IP addresses, that identify and locate a host in an IP network. It is similar to DNS, yet unlike the DNS, the hosts file is under the direct control of the local computer's administrator. Therefore, mechanism behind this MiM is similar to DNS poisoning, but it is done by the user himself. It is unusual to poisoning one's local hosts file by oneself, but now it is doing the attack in a benign way.

- ***Save the SHSH for later authentication (reply attack)***

ECID is considered as a security measure for Apple to protect its iDevices from being manipulated and almost every upcoming iDevice is affected by this security. Now what this security does is when someone updates his iDevice to a newer firmware without saving ECID SHSH blobs, they not only lose their jailbreak but also the ability to downgrade their iDevice to an older firmware.

Since Jailbreaking is doing something that is not officially considered right by Apple, the whole process of Jailbreaking is considered as a self-redemption. When iTunes thinks it is talking to Apple's authentication server, it actually is talking to the constructed server instead. This will allow iTunes to access signatures already stored "on file" in the server. This server will also act as a cache for any SHSH blobs it hasn't seen, acting as an intermediary to Apple's server. This effectively registers the users iDevice with the "on file" mechanism, which means users can now enjoy the protections of being able to downgrade your firmware whenever in the future. After one has just stored a copy of Apple's sign off and then returns it at a later point, a replay attack can be sufficient any time a downgrade is required by the user.

Another meaningful point should be addressed is: even if the user who don't playing a jailbreak, and even never intend to jailbreak, he/she should consider using the new "on file" service. There is a case that Apple releases an OS upgrade in the future, a user takes it, and then accidentally break something important, like the e-mail account, or todo list. The user could never downgrade and restore the former system without saving the old signature file on the other server like Cydia.

3. Digging holes on iDevices and building up your own house

Why Jailbreaking downgrades and backup the old versions of firmwares for different iDevices is so important? It is due to the Jailbreaking works are based on different exploits appear on several old firmwares. When never an exploit is published, apple will take action to patch it and publish related new firmware version and stop signing the old exploitable ones. This official firmware update is initially considered as Jailbroken iDevices' nightmare. However, Jailbreak community is always digging new holes in the newly-published firmware or holding some great exploits for newly published firmware. This section, exploits are classified according to the level they appeared in the Bootchain. Then explanation of the pros and cons of each type of exploits are rendered, and difference are provided by comparisons.

- ***iOS exploits classification and mechanism***

1) userland Exploits

A userland exploit is one found in the system itself. It uses a hole in one of the application built in the system before, like safari or mail receiver to get root access. To explain this, a understand of UNIX-like system's execution layer is necessary. Actually, there are usually two fundamental levels of execution in a UNIX-like environment, one is kernel mode, kernel hooks and kernel space memory access and the other is user mode, user space memory access. The term *userland* refers to all the code that is not running in kernel space but in user space.

Since a userland jailbreaking exploits a vulnerability belonging to some code running in user space, it is untethered Jailbreak because nothing can cause a recovery mode loop in the iBoot which is related to the kernel mode. However, drawbacks reside in this type of jailbreaking: these exploits provide filesystem access only, no very low level control of the kernel. Moreover a userland Jailbreak can be fixed very easily since it exploits the vulnerability of some non-vital code built in the applications. However, this type of exploits is more user-friendly and platform independent, as it is related to the application layer but not the iBoot layer.

2) iBoot Exploits

Different from the userland exploit the iBoot exploits can render extremely low level control to the iOS. These exploits provide filesystem and iBoot access.

An iBoot exploit is found in the iDevice's third bootloader, called iBoot, the SecureROM and Low Level Bootloader(LLB) are 1st and 2nd, which will be mentioned later. It uses a hole in iBoot in order to turn off code signing mechanism, and then user can install and run program that does all the 'evil'. This kind of exploit can be tethered if the device refresh its bootrom, since the bootrom checks the LLB which checks the iBoot (which is modified), and results in a recovery screen, and the user has to re-exploit it to get out of that recovery screen, which is called a tethered exploit. Since the bootrom , is truly read-only. In the iDevices prior to iPod Touch 2G, the read-only nature of the bootrom does benefit the jailbreak community. It means Apple can never add signature checking of the LLB by refreshing the bootrom in those devices. And it means they can never fix certain types of old exploits in those old iDevice.

An iBoot jailbreak is much more valuable, as it is at a deeper level type of exploits. With this sort of exploit, user could enable the device to accept custom firmware and probably jailbreak forever more.

3) Bootrom Exploits

Due to the new refreshable bootrom introduced in Apples' new iDevice, the jailbreaking community is considering new ways of exploits to totally bypass the authentication chain from the lowest level. That is the Bootrom exploits. These exploits provide filesystem, iBoot, and NOR access (custom boot logos). A bootrom exploit is something found in the iDevice's 1st bootloader, the SecureROM. It digs a hole to disable signature checks, which can be used to load patched NOR firmware. This kind of exploit cannot be tethered, there's nothing to check the bootrom, as it is the lowest level in the whole code signing chain for authentication. These cannot be patched easily by Apple, but need new chip.

On Februrary 27th , 2008, the iPhone Dev Team demonstrated that they had the ability to load a custom recovery logo (NOR) to the iPhone, bypassing signature checks. They name the exploit "Pwnage". It was the earliest version of Bootrom exploits. It is an amazing exploit, since it comes in the lowest level (code in flashed in the hardware) so that it could not be patched. The mechanism of "Pwnage" exploit is it depends on the fact that Bootrom does not signature check LLB, breaking the chain of authentication trust. That is to say user could patch the LLB signature check arbitrarily via by passing the Bootrom check, so that LLB would accept a patched iBoot, and that iBoot will accept a patched kernel, and so on so forth.

- ***iPhoneLinux program—building users' own land***

Based on the layered exploits classification and corresponding mechanism and working environment introducing, it is nature to render this Bootchain below:

SecurityROM ->LowLevelBootloader(LLB)->iBoot->Kernel->System Software

Bootrom Exploits->iBoot Exploits ->Userlands Exploits

Each of high layer checks the signature of the next lower layer before loading into it. While search for the Jailbreak community, I found a very interesting project is going on, the iPhonelinux, which is aiming at porting linux on the iPhone and make a Free (free software) OS alternative. One of the goals for this project is to replace part of the BootChain that is after iBoot:

SecurityROM->OpeniBoot->Linux Kernel->X Server->Window Manager

On April 21, 2010, an Android distribution based on iPhoneLinux called iDroid was released. Up till now, the iDroid has not fully port the Linux kernel and the Google Android OS to Apple's iDevices. But based on the implementation of the 'OpeniBoot' bootloader, the team has finish the work to boot the Linux Kernel on iDevice, which enables users to boot Google Android & any other Linux based operating system easily. Video and simple tutorial is appeared to guide users doing this. In this way, users can build up their own playground on the iDevices, without restrictions from Apple.

Appendix collected examples of classified exploits

1) Userland exploits

❖ Symlinks - Works up to iOS 1.1.1
❖ LibTiff - Works up to iOS 1.1.1
❖ Mknod - Works up to iOS 1.1.2
❖ Dual Boot Exploit - Works up to iOS 2.0 beta 3
❖ MobileBackup Copy Exploit - Works up to iOS 3.1.3
❖ Malformed CFF Vulnerability - Works up to iOS 4.0, 4.0.1

2) iBoot exploits

❖ Restore Mode - Works up to iOS 1.0.2
❖ Ramdisk Hack - Works up to iOS 2.0 beta 3
❖ diags - Works up to iOS 2.0 beta 5
❖ iBoot Environment Variable Overflow - Works up to iOS 3.1 beta 3
❖ usb_control_msg(0x21, 2) Exploit - Works up to iOS 3.1.2
❖ ARM7 Go - Works on iOS 2.1.1

3) Bootrom exploits

❖ Pwnage 1.0 (Ramdisk + AppleImage2NORAccess)

- Pwnage 2.0 (DFU + Malformed Certificate)
- 0x24000 Segment Overflow - only in iBoot-240.4 (old bootrom)
- usb_control_msg(0xA1, 1) Exploit - in iBoot-240.4 and iBoot-240.5.1
- 0x24000 Segment Overflow - only in iBoot-359.3