iOS Hacker's Handbook

Charlie Miller · Dionysus Blazakis · Dino Dai Zovi
Stefan Esser · Vincenzo Iozzo · Ralf-Philipp Weinmann

Table of Contents

Introduction

Chapter 1

iOS Security Basics

If you're like us, every time you get your hands on a new device you wonder how secure it is. The iPhone was no exception. Here was a device that had jumped across the threshold from being a phone that might have a small web browser to a device that was more like your computer than your old phone. Surely there were going to be similar security issues in these (and future) devices to the issues that were already occurring on desktop computers. What precautions and security mechanisms had Apple built into these devices to prevent compromises? Here was a chance to start a whole new branch of computing, from the beginning. How important was security going to be for these emerging smart devices?

This chapter answers these questions for iOS devices. It begins by looking at the hardware seen for various iOS devices and then quickly moves into describing the security architecture of iOS 5. This includes highlighting the many layers of defense built into current devices to make attacks by malware and exploitation by attackers difficult. It then illustrates how these defenses have held up (or not) in the real world by showing some attacks that have occurred against iOS devices. This section on iOS attacks takes a historical approach starting from attacks against the very first iPhone and ending with attacks against iOS 5 devices. Along the way you will notice how much the security of iOS devices has improved. Whereas the very first versions of iOS had almost no security, the most recent versions of iOS 5 have quite a strong and robust security posture.

iOS Hardware/Device Types

As iOS evolved during the years, so did the hardware inside the various Apple devices. When smartphones and tablets became widespread among users, people started to feel the need to have powerful devices at their disposal. In a way, the expectation was to have a computer in their pocket.

The first step in that direction was the creation of the iPad. The original iPad had an ARM Cortex-A8 CPU, which, compared to the CPU present on the original iPhone, was roughly twice as fast.

Another big step forward was the iPad 2 and the iPhone 4S. They both feature ARM Cortex-A9 dual-core processors, which are 20 percent faster compared to the A8 in terms of CPU operations. Even more astonishing is the fact that the GPU of the A9 is nine times faster compared to the A8.

From a security perspective, the biggest hardware differences came with the iPhone 3GS and the iPad 2. The iPhone 3GS was the first one to support the Thumb2 instruction set. The new instruction set changed the way ROP payloads needed to be created. Most code sequences present in previous versions of the device were suddenly different on the 3GS.

The iPad 2, on the other hand, introduced dual-core processors, which in turn enabled the iOS allocator to work in full swing. This has had a huge impact on exploit development because exploits become much less reliable in a multi-processor environment.

Another relevant hardware component from a security point of view is the baseband. In fact, in most countries the iDevices are bound to a carrier (locked).

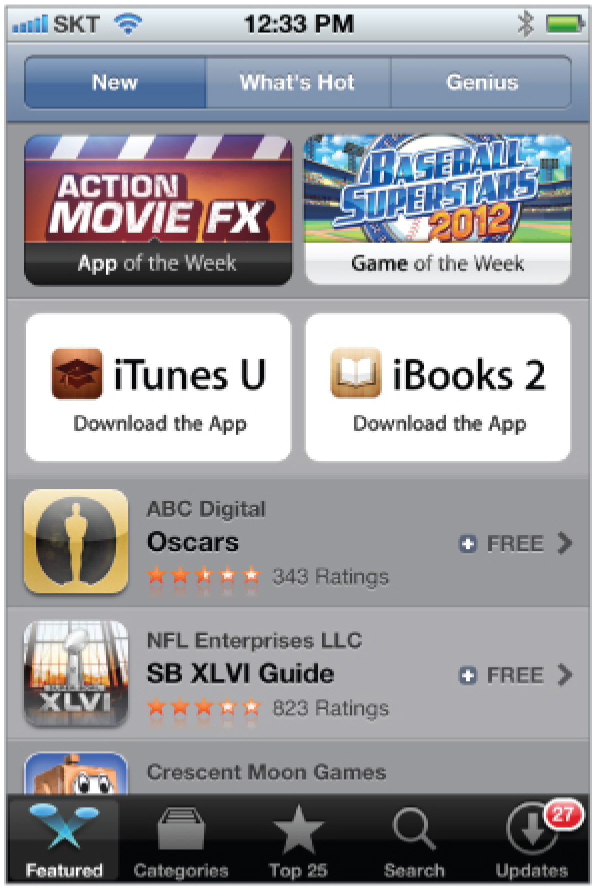
To unlock iPhones, most exploits use bugs in the baseband component inside the phone. Both devices have historically used Infineon baseband firmwares. Only recently with the iPhone 4 CDMA and iPhone 4S has Apple moved to Qualcomm.

A number of exploits have been published on the various Infineon firmwares, but none yet on the Qualcomm ones.

How Apple Protects the App Store

One of the things that makes iOS devices so great is the number of applications, or apps, that are available to run on them. These apps can be found in Apple's App Store. There have been more than 18 billion downloads from the App Store, and at least 500,000 different apps are available (see Figure 1.1).

Figure 1.1 A users' view of the App Store.



Apps are developed using Xcode and the iOS SDK on Mac OS X computers. The built apps can run in an iOS simulator or can be put on real devices for testing. The apps are then sent to Apple for review. If approved, they are signed by Apple's private key and pushed out to the App Store for download. Apps must be signed by a trusted party, such as Apple, or they will not run on the devices because of the Mandatory Code-Signing requirement in iOS (see Chapter 4 for more details). Enterprises can also distribute apps to their employees using a similar system, but the employees' phones must be configured to accept apps that are signed by the enterprise as well as by Apple.

Of course, once you could download new apps to iOS devices, it opened up the possibility for malware. Apple has tried to reduce this risk with code signing and the App Store review process. Additionally, App Store apps run in a sandbox at a low privilege level to reduce the damage they can cause. You see more on this in a bit.

Understanding Security Threats

This book is about iOS security — how it works and how to break it. To fully understand the decisions made by Apple in trying to secure its devices, it is first necessary to think about the different types of threats that the device might face.

At a high level, iOS devices face many of the same types of attacks that any desktop computer faces. These types of attacks can be split into two broad categories: malware and exploits. Malware has been around for decades on personal computers and is starting to become a menace for mobile devices as well. In general, malware is any software that does something "bad" when it is installed and run on a device. This malware might be bundled with software the user wants, or it might disguise itself as something the user wants. In either case, the user downloads and installs the malware and when the malware is executed, it performs its malicious actions. These actions might include sending e-mails, allowing remote access to an attacker, installing a keylogger, and so on. All general-purpose computing devices are susceptible at some level to malware. Computers are designed to run software; they do what they are told. If the user tells it to run something that turns out to be malicious, the computing device will happily comply. There is no real vulnerability with the computer; it is just not in a position to know which programs it should run and which it should not. The typical way to protect devices from malware is with antivirus (AV) software. It is the AV's job to determine which software is safe to run and which is not safe to run.

On the other hand, exploits take advantage of some underlying defect of the software on the device to run its code. A user might be innocently surfing a web page, reading an e-mail, or doing absolutely nothing at all, when all of a sudden some malicious code (perhaps in the form of a web page, e-mail, or text message) takes advantage of a vulnerability to run code on the device. Such attacks are sometimes called *drive-by-downloads* because, unlike the malware example, the user is mostly an innocent victim and wasn't trying to install any code, but just trying to use his or her device! The exploit might run some code inside the compromised process, or it might download some software, install it, and run it. The victim might have no idea that anything out of the ordinary has happened.

Exploitation such as this requires two ingredients. The first is a flaw or vulnerability in the software on the device. The second is a way to leverage this vulnerability to get attacker-controlled code to run on the device. Because of this two-step process, you have two main ways to protect against this kind of attack. The first involves making it harder to find vulnerabilities. This might mean exposing less code to the attacker (reducing the attack surface) or cleaning up and removing as many flaws as possible in the code. The problem with this approach is that some code must always be exposed to the attacker or the device cannot interact with the outside world. Furthermore, it is very difficult to find all (or even most) of the vulnerabilities lurking deep in a code base. If it were easy, there would be no book like this one — or any jailbreaks, for that matter!

The second approach to protecting against exploitation is to make the process of going from vulnerability to performing a malicious action more difficult. This involves a lot of engineering technologies such as data execution prevention, and memory randomization, which are discussed throughout this book. Continuing with this line of reasoning, if you concede that an attacker will eventually find a bug in your code and might get it running, you can at least limit the damage that code might do. This involves using privilege separation or sandboxing to keep sensitive data from some processes. For example, your web browser probably doesn't need the capability to make videos or send text messages.

So far, the discussion has centered on security threats for all devices. Next, you examine how attacking an iOS device might differ from attacking a personal computer. In many respects, it is very similar. iOS is a stripped-down version of Mac OS X, and so many of the vulnerabilities and attacks are shared between the two or are at least very similar. The differences that do exist basically boil down to the attack surface. The attack surface is the portion of code that is accessible to an attacker and that processes attacker-supplied input.

In some respects, the attack surface of iOS devices is smaller than a corresponding Mac OS X desktop computer. Certain applications, such as iChat, are not installed in iOS. Other applications, such as QuickTime, are greatly reduced in their capabilities. Likewise, certain file types are rejected by MobileSafari but are parsed by Safari. So in these ways iOS has a smaller attack surface. On the other hand, certain features are present only on iOS devices, particularly the iPhone. One such example is SMS messages. The fact that iPhones parse these messages but you don't have corresponding code in Mac OS X demonstrates that in some regards, iOS has a larger attack surface. Another example of the expanded attack surface of iOS includes the code running on the baseband processor of the iPhone. We talk about these two iOS-specific attack vectors later in this book in Chapters 6 and 12, respectively.

Understanding the iOS Security Architecture

You can imagine some of the nasty attacks that await an iOS device; this section discusses how the device is engineered to withstand these kinds of attacks. Here we describe iOS 5, which as you'll see, is pretty secure. In a later section we show you the evolution of how iOS got here, which was a bit of a bumpy ride.

The Reduced Attack Surface

The attack surface is the code that processes attacker-supplied input. If Apple has a vulnerability in some code, and either the attacker can't reach it or Apple doesn't ship the code at all in iOS, an attacker cannot base an exploit on this vulnerability. Therefore, a key practice is minimizing the amount of code an attacker can access, especially remotely.

In the ways that were possible, Apple reduced the attack surface of iOS compared to Mac OS X (or other smartphones). For example, love it or hate it, Java and Flash are unavailable on iOS. These two applications have a history of security vulnerabilities and not including them makes it harder for an attacker to find a flaw to leverage. Likewise, iOS will not process certain files, but Mac OS X will. One example is `.psd` files. This file type is handled happily in Safari, but not in MobileSafari, and importantly, nobody would likely notice the lack of support for this obscure file format. Likewise, one of Apple's own formats, `.mov`, is only partially supported, and many `.mov` files that play on Mac OS X won't play in iOS. Finally, even though iOS renders `.pdf` files natively, only some features of the file format are parsed. Just to see some numbers on the subject, Charlie Miller once fuzzed Preview (the native Mac OS X PDF viewer) and found well over a hundred crashes. When he tried these same files against iOS, only 7 percent of them caused a problem in iOS. This means that just by reducing the PDF features that iOS handled, it reduced the number of potential vulnerabilities by more than 90 percent in this case. Fewer flaws mean fewer opportunities for exploitation.

The Stripped-Down iOS

Beyond just reducing the potential code an attacker might exploit, Apple also stripped down the number of useful applications an attacker might want to use during and after exploitation. The most obvious example is that there is no shell (`/bin/sh`) on an iOS device. In Mac OS X exploits, the main goal is to try to execute a shell in “shellcode.” Because there is no shell at all in iOS, some other end goal must be developed for iOS exploits. But even if there were a shell in iOS, it wouldn't be useful because an attacker would not be able to execute other utilities from a shell, such as `rm`, `ls`, `ps`, and so on. Therefore, attackers who get code running will have to either perform all of their actions within the context of the exploited process, or bring along all the tools they want to use. Neither or these options are particularly easy to pull off.

Privilege Separation

iOS separates processes using users, groups, and other traditional UNIX file permission mechanisms. For example, many of the applications to which the user has direct access, such as the web browser, mail client, or third-party apps, run as the user `mobile`. The most important system processes run as the privileged user `root`. Other system processes run as other users such as `_wireless` or `_mdnsresponder`. By using this model, an attacker who gets full control of a process such as the web browser will be constrained by the fact the code she is executing will be running as user `mobile`. There are limits to what such an exploit can do; for example, the exploit will not be able to make system-level configuration changes. Likewise, apps from the App Store are limited in what they can do because they will be executed as user `mobile` as well.

Code Signing

One of the most important security mechanisms in iOS is code signing. All binaries and libraries must be signed by a trusted authority (such as Apple) before the kernel will allow them to be executed. Furthermore, only pages in memory that come from signed sources will be executed. This means apps cannot change their behavior dynamically or upgrade themselves. Together, these actions prevent users from downloading and executing random files from the Internet. All apps must come from the Apple App Store (unless the device is configured to accept other sources). Apple has the ultimate approval and inspects applications before they can be hosted at the App Store. In this way, Apple plays the role of an antivirus for iOS devices. It inspects each app and determines if it is okay to run on iOS devices. This protection makes it very hard to get infected with malware. In fact, only a few instances of malware have ever been found for iOS.

The other impact of code signing is that it complicates exploitation. Once an exploit is executing code in memory, it might want to download, install, and execute additional malicious applications. This will be denied because anything it tries to install will not be signed. Therefore, exploits will be restricted to the process they originally exploit, unless it goes on to attack other features of the device.

This code signing protection is, of course, the reason people jailbreak their phones. Once jailbroken, unsigned applications can be executed on the device. Jailbreaking also turns off other features (more on that later).

Data Execution Prevention

Normally, data execution prevention (DEP) is a mechanism whereas a processor can distinguish which portions of memory are executable code and which portions are data; DEP will not allow the execution of data, only code. This is important because when an exploit is trying to run a payload, it would like to inject the payload into the process and execute it. DEP makes this impossible because the payload is recognized as data and not code. The way attackers normally try to bypass DEP is to use return-oriented programming (ROP), which is discussed in Chapter 8. ROP is a procedure in which the attacker reuses existing valid code snippets, typically in a way never intended by the process, to carry out the desired actions.

The code-signing mechanism in iOS acts like DEP but is even stronger. Typical attacks against DEP-enabled systems use ROP briefly to create a section of memory that is writable and executable (and hence where DEP is not enforced). Then they can write their payload there and execute it. However, code signing requires that no page may be executed unless it originates from code signed by a trusted authority. Therefore, when performing ROP in iOS, it is not possible to turn off DEP like an attacker normally would. Combined with the fact that the exploit cannot execute applications that they may have written to disk, this means that exploits must only perform ROP. They may not execute any other kinds of payloads such as shellcode or other binaries. Writing large payloads in ROP is very time-consuming and complex. This makes exploitation of iOS more difficult than just about any other platform.

Address Space Layout Randomization

As discussed in the previous section, the way attackers try to bypass DEP is to reuse existing code snippets (ROP). However, to do this, they need to know where the code segments they want to reuse are located. Address space layout randomization (ASLR) makes this difficult by randomizing the location of objects in memory. In iOS, the location of the binary, libraries, dynamic linker, stack, and heap memory addresses are all randomized. When systems have both DEP and ASLR, there is no generic way to write an exploit for it. In practice, this usually means an attacker needs two vulnerabilities — one to obtain code execution and one to leak a memory address in order to perform ROP — or the attacker may be able to get by with having only one very special vulnerability.

Sandboxing

The final piece of the iOS defense is sandboxing. Sandboxing allows even finer-grained control over the actions that processes can perform than the UNIX permission system mentioned earlier. For example, both the SMS application and the web browser run as user `mobile`, but perform very different actions. The SMS application probably doesn't need access to your web browser cookies and the web browser doesn't need access to your text messages. Third-party apps from the App Store shouldn't have access to either of these things. Sandboxing solves this problem by allowing Apple to specify exactly what permissions are necessary for apps. (See Chapter 5 for more details.)

Sandboxing has two effects. First, it limits the damage malware can do to the device. If you imagine a piece of malware being able to get through the App Store review process and being downloaded and executed on a device, the app will still be limited by the sandbox rules. It may be able to steal all your photos and your address book, but it won't be able to send text messages or make phone calls, which might directly cost you money. Sandboxing also makes exploitation harder. If an attacker finds a vulnerability in the reduced attack surface, manages to get code executing despite the ASLR and DEP, and writes a productive payload entirely in ROP, the payload will still be confined to what is accessible within the sandbox. Together, all of these protections make malware and exploitation difficult, although not impossible.

A Brief History of iOS Attacks

You now have a basic understanding of the defensive capabilities of iOS devices. This section discusses some successful attacks against these devices to see how their security holds up in the real world. This discussion also demonstrates how the security of the device has evolved to keep up with real-world attacks.

Libtiff

When the original iPhone came out in 2007, people were lining up to get one. Perhaps in an effort to get it out the door as quickly as possible, the device did not ship in a very secure state. You've seen how iOS 5 looks, but compare it to “iOS 1” in the original iPhone:

- There was a reduced attack surface.
- There was a stripped-down OS.
- There was no privilege separation: All processes ran as root.

- There was no code-signing enforcement.
- There was no DEP.
- There was no ASLR.
- There was no sandboxing.

So, if you could find a vulnerability in the device, it was very easy to exploit it. The exploit was free to run shellcode or download files and execute them. Even finding vulnerabilities was rather easy because the original iPhone software was shipped with known flaws. Every attack gave you instant root access.

Tavis Ormandy first pointed out that the version of Libtiff, used to process TIFF images, had a known vulnerability in it. Chris Wade actually wrote a working exploit for this vulnerability. Therefore, it was possible to surf to a malicious website and have the site get remote root access to your device. This flaw was patched in iPhone OS 1.1.2.

Compare the Libtiff exploit at that time with what would have to happen for a similar vulnerability in the Libtiff library found today. The original exploit filled heap memory with executable code and then redirected execution to it. This would fail now because of the presence of DEP. Therefore, the exploit would have to use ROP and somehow defeat the ASLR. This would probably require an additional vulnerability. Furthermore, even if the attacker were to get an exploit working, the attacker would only have the permissions of the user `mobile` and would be sandboxed as well. This is in stark contrast to having unfettered root access.

While we're on the topic of iOS 1, it should be pointed out that malware wasn't much of a problem for it. This is because, with what seems unbelievable now, the original iPhone had no official way to download third-party apps. That didn't come along until iOS version 2.

Fun with SMS

In 2009, researchers Collin Mulliner and Charlie Miller found a vulnerability in the way the iPhone parsed SMS messages. By this time, iOS 2 was in use. iOS 2 featured almost all of the security mechanisms present in iOS 5 with the exception of ASLR. The problem was that while most processes ran as an unprivileged, sandboxed user, the particular process that handled SMS messages did not. The responsible program, CommCenter, happened to run as `root` with no sandboxing.

The problem with not implementing ASLR is that DEP really works only in conjunction with ASLR. That is, if memory is not randomized and an attacker knows exactly where all executable code is located, performing ROP is rather easy.

Besides being a powerful way into the system, SMS makes a great attack vector for a number of other reasons. For one, it requires no user interaction. Instead of trying to get a victim to visit a malicious website, an attacker only has to know the victim's phone number and send the attack. Additionally, the victim cannot prevent the attack from occurring. There is no way to disable SMS on a default phone. Finally, the attack is silent and is possible even when a device is powered off. If an attacker sends the malicious SMS messages while a device is off, the carrier will conveniently queue them up and deliver them as soon as the device powers up.

This flaw was patched in version 3.0.1. Today, things would be more difficult because not only would the exploit have to deal with ASLR, but now the CommCenter process runs as user `_wireless` instead of `root`.

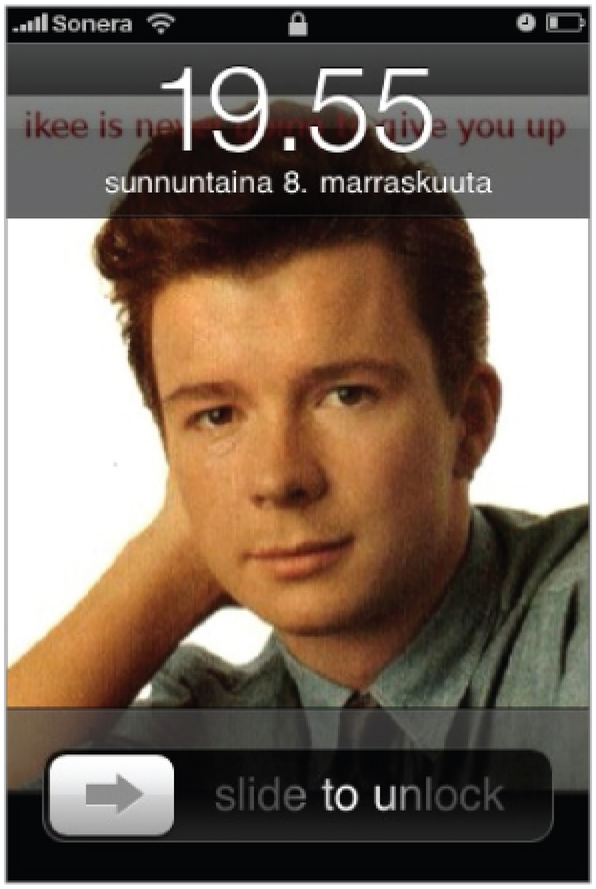
The Ikee Worm

By the time iOS 2 came out, the device was in pretty good shape. However, it turns out that jailbreaking your device breaks the whole security architecture of the device. Sure, it disables code signing, but it does much more. It increases the attack surface by adding software (after all, the whole point is to run unsigned code). It adds a bunch of system utilities, such as a shell. It can install things that run as the `root` user. By turning off code signing, you also turn off the strong form of DEP. That is, ROP payloads can disable DEP and write and execute shellcode on jailbroken devices. Finally, the new unsigned apps are not sandboxed. So, yes, jailbreaking pretty much turns off all the security of the device, not just the code signing.

Therefore, it shouldn't come as a shock that jailbroken phones were targeted for exploitation. The Ikee worm (also known by a variety of other names like Dutch ransom, iPhone/Privacy.A, or Duh/Ikee.B) took advantage of the fact that many people who jailbroke their phones installed an SSH server and didn't bother to change the widely-known default root password This meant anybody could connect to their device and remotely control it with root privileges. It is hardly a challenge to write a worm given these conditions. Additionally, the SSH server was in no way sandboxed.

The worm did various things at different stages of its lifetime. Initially, it just changed the wallpaper of the device (see Figure 1.2). Later, it was changed to perform malicious actions such as locking the phone for ransom, stealing content, or even enrolling it to become part of a botnet.

Figure 1.2 Rick Astley is never gonna give you up.


Sonera
19.55
ikee is never going to give you up
sunnuntaina 8. marraskuuta
slide to unlock

Obviously, none of this could have happened prior to the victims jailbreaking their devices.

Storm8

In 2009, games developed by popular developer Storm8 were collecting the cell phone numbers of the devices on which they were playing. The games would then send this information to Storm8 servers. Some of the affected apps included "Vampires Live," "Zombies Live," and "Rockstars Live" (see Figure 1.3). A class action suit was filed against Storm8, which claimed the data collection feature of the apps was a simple mistake. There were approximately 20 million downloads of Storm8 apps during the time in question.

Figure 1.3 Vampires Live brought more than rampaging vampires to iOS.

SpyPhone

SpyPhone was a proof of concept app written by Seriot Nicolas that exercised the limits of the iOS sandbox for third-party apps. It tried to access every conceivable piece of information and perform any actions allowed by the sandbox. One thing to notice about the iOS sandbox is that every third party app from the App Store has the same sandbox rules. That means that if Apple thinks one app should have a certain capability, all apps must have that capability. This differs, for example, from the Android sandbox where every app can have different capabilities assigned to it based on its needs. One of the weaknesses of the iOS model is that it may be too permissive. For example, by using public APIs in entirely legitimate ways (despite the fact the app was in a sandbox), SpyPhone was able to access the following data:

- Cell phone number
- Read/write access to address book
- Safari/YouTube search terms
- E-mail account information
- Keyboard cache
- Geotagged photos
- GPS information
- WiFi access point names

This app demonstrated that even inside a sandbox, a malicious program could extract a frightening amount of information from the device.

Pwn2Own 2010

Two of the authors of this book, Vincenzo Iozzo and Ralf-Philip Weinmann, won the Pwn2Own hacking competition against the iPhone 3GS in 2010. They found a vulnerability in MobileSafari that allowed them to execute code remotely. This was in iOS version 3 before ASLR was introduced. Their entire payload was written in ROP due to the code-signing mechanisms in place. Using ROP, they were able to open up the SMS database, which stored all the text messages, and send them off to a remote server they controlled. They were, however, limited to the user `mobile` and the MobileSafari sandbox. It would have taken some more work to do more damage. For their effort they won $15,000 and the phone. The next year two different authors of this book won the same competition.

Jailbreakme.com 2 (“Star”)

So far we've talked about all the limits that something like iOS 5 puts on a remote attacker. This makes attacks very difficult, but not impossible. An example of this was shown in August 2010 by comex's infamous jailbreakme.com website. (The first jailbreakme.com worked against the original iPhone and so was rather easy in comparison.) This second jailbreakme.com site performed a series of actions that eventually led to jailbreaking the iOS device that visited it. This means it must obtain remote root access, similar to the iOS 1.0 days. In this case, however, it was against iOS 4.0.1, which had all the security mechanisms except ASLR (which hadn't been added yet). So how did it work? First, it took advantage of a stack overflow in the way a particular type of font was handled by MobileSafari. This allowed the exploit to begin its ROP payload within MobileSafari. Then, instead of just shipping off the SMS database, this sophisticated payload proceeded to exploit another vulnerability to increase its level of access to the device. This second vulnerability was an integer overflow in an `IOSurface` property in IOKit. This second attack allowed code execution by the attacker inside the kernel. From the kernel, it disabled code signing, then the ROP downloaded an unsigned dynamic library that jailbroke the phone and loaded it. Apple quickly patched it because while the jailbreakme.com site simply jailbroke your phone, it could have been easily modified to perform any actions on the device it wanted.

Jailbreakme.com 3 (“Saffron”)

One thing all the examples have had in common so far is that they have been against iOS versions before 4.3. This is when ASLR was introduced. Once that final obstacle is added, perhaps it is too difficult to exploit the device? Well, comex again showed this is not the case with the third incarnation of the jailbreakme.com site targeting iOS versions up to 4.3.3. Again, this required two exploits, one to get code execution and one to disable the code signing.

But what about the ASLR? You learn more about this exploit in Chapter 8, but for now it is enough to know that the particular vulnerability exploited allowed the attacker to both read and write memory. With that, it was possible for it to figure out where it was located in memory by reading the values of some nearby pointers. After that it was able to corrupt memory and get control of the process by writing to memory. Like we said before, defeating ASLR usually requires either two vulnerabilities or one really special one. In this case, the exploit took advantage of a single, but very powerful, vulnerability.

Summary

This chapter began by introducing iOS devices, including the hardware and how they've changed since their introduction. You then learned some basic information about security topics, including the types of threats that are faced by iOS devices. The chapter then introduced many of the concepts of this book at a high level. It discussed the security design of iOS; many of whose layers will be highlighted in their own chapters later on. Finally, it walked through some of the attacks that have succeeded against iOS in the past, right up to ones that bypass all the security of even iOS 5.

Chapter 2

iOS in the Enterprise

As Apple's iOS-based devices have gained popularity among consumers, more and more enterprises have begun to allow employees to access and store enterprise data on these devices. Typically, the enterprise purchases and fully manages the smartphones or other devices that may be used to access sensitive enterprise data. In some cases (and in what is becoming increasingly common), enterprises may allow employees to access enterprise data from devices they personally own. In either case, the benefits of using these mobile devices to access enterprise data must be weighed against the new security risks that they present.

Any mobile device can get misplaced, lost, or stolen. If the mobile device stores or is capable of accessing sensitive enterprise data, there is a risk that this data may be recoverable and fall into the wrong hands. For these reasons, it is important that access to the physical device be restricted by requiring a strong passcode, and that it can be remotely locked or wiped if it is lost. This chapter describes how to use Apple's iPhone Configuration Utility and Lion Server's Profile Manager to create and apply configuration profiles to iOS devices. These profiles can be used to ensure that the devices enforce your organization's security policy, including requiring a strong device passcode, for example. As a Mobile Device Management (MDM) service, Profile Manager can also be used to remotely lock or wipe a lost device.

iOS Configuration Management

iOS-based devices are managed through the creation and installation of configuration profiles. These profiles contain settings configured by an administrator for installation on a user's device. Most of these settings correspond to configuration options visible in the iOS Settings app, but some settings are available only through a configuration profile and some are available only in the iOS Settings app. The settings available in configuration profiles are the only ones that can be centrally managed.

The simplest way to create and manage configuration profiles is by using Apple's iPhone Configuration Utility for Mac or Windows. This graphical utility lets administrators create and manage configuration profiles. These profiles can be installed onto iOS devices over a USB connection, by attaching them to an e-mail message sent to the device's owner, or by hosting them on a web server.

To manage a larger number of devices, enterprises should use a Mobile Device Management (MDM) system. Apple provides one in Lion Server through the Profile Manager service. This service works well for workgroups and small- to medium-sized organizations. For larger enterprises, however, a commercial third-party MDM solution would likely work best.

This section covers the basics of configuration profiles and how to use the iPhone Configuration Utility and Lion Server's Profile Manager to create and install a simple configuration profile.

Mobile Configuration Profiles

A configuration profile is structured as an XML property list file (referred to as a plist) with data values stored in Base64. The plist data may optionally be signed and encrypted, in which case the file will be structured according to RFC 3852 Cryptographic Message Syntax (CMS). Because configuration profiles may contain sensitive information, such as user passwords and Wi-Fi network passwords, they should be encrypted if the profile is going to be sent over a network. The use of a Mobile Device Management (MDM) server automates this, which is recommended for any enterprise managing iOS devices.

The configuration profile consists of some basic metadata and zero or more configuration payloads. The configuration profile metadata includes the human-readable name, description, and creating organization of the profile, as well as some other fields that are used only under the hood. The configuration payloads are the most important portions of the profile, because they implement the configuration options specified in the profile. The available configuration payload types in iOS 5 are listed and described in Table 2.1.

Table 2.1 Configuration Profile Payload Types

Payload	Description
Removal Password	Specifies a password that users must enter to remove a locked profile from the device
Passcode Policy	Defines whether a passcode is required to unlock the device and how complex this passcode must be
E-mail	Configures the user's e-mail account
Web Clip	Places a web clipping on the user's home screen
Restrictions	Restricts the user of the device from performing certain actions, such as using the camera, iTunes App Store, Siri, YouTube, Safari, and so on
LDAP	Configures an LDAP server to use
CalDAV	Configures a user's network calendar account using CalDAV
Calendar Subscription	Subscribes the user to a shared CalDAV calendar
SCEP	Associates the device with a Simple Certificate Enrollment

	Protocol server
APN	Configures an iOS device with a cellular baseband (iPhone or iPad) to use a specific mobile carrier
Exchange	Configures a user's Microsoft Exchange e-mail account
VPN	Specifies a Virtual Private Network (VPN) configuration for the device to use
Wi-Fi	Configures the device to use the specified 802.11 network

Each payload type has a set of property list keys and values that define the supported set of configuration settings. The full list of these keys and their available values for each payload are listed in Apple's iOS Configuration Profile Reference in the iOS Developer Library. Although you can create the configuration profile manually using this specification, only Mobile Device Management product developers are likely to do so. Apple recommends that most users rely on Apple's iPhone Configuration Utility or a Mobile Device Management product to create, manage, and deploy their configuration profiles. Enterprises with a small number of iOS-based devices are likely to configure them using the iPhone Configuration Utility, which is described next.

iPhone Configuration Utility

Apple's iPhone Configuration Utility is a graphical utility for Mac OS X and Windows that helps users create, manage, and install configuration profiles on iOS devices. At the time of writing, the latest available version is 3.4, which has just been updated to support the new configuration options in iOS 5.0.

The iPhone Configuration Utility automatically creates a root certificate authority (CA) certificate in the user's keychain the first time it is run. This CA certificate is used to sign the certificate that is automatically created for each device connected over USB to the host running the iPhone Configuration Utility. These certificates are used to sign and encrypt configuration profiles for secure transmission to the