hgi EMMA

RUHR UNIVERSITÄT BOCHUM RUB

iOS Application Security

Diploma Thesis

Student:	Manuel Binna Email: manuel@binna.de Matriculation Number: 108004202162
Supervisor:	Prof. Dr. Thorsten Holz
Period:	From 2010-12-06 till 2011-06-06

Embedded Malware Research Group
Prof. Dr. Thorsten Holz
Faculty of Electrical Engineering and Information Technology
Ruhr University Bochum

Abstract

This thesis provides insight into the various aspects of mobile application security on the iOS platform and shows the pitfalls of secure application development for iPhone, iPod touch, and iPad. It discusses the possibilities of evil third-party developers who want to steal the user's private data. The focus is on the official, i.e., non-jailbroken, version of iOS.

Declaration

I hereby declare that the content of this thesis is a work of my own and that it is original to the best of my knowledge, except where indicated by references to other sources.

_____ _____

Location, Date Signature

Acknowledgements

I would like to thank Prof. Dr. Thorsten Holz for providing me the opportunity to write my diploma thesis in his research group. His supervision and feedback were invaluable to me.

Thanks go to René Korthaus, Thomas Fischer, and Marcel Winandy for the discussions about iOS security during their preparation of a practical course on this topic. I would like to thank Tim Kornau for helping me disassembling ARM binaries and Stefan Heyse for providing answers about cryptographic keys in hardware.

I am very grateful to Raphael Sobik and Markus Rudel who reviewed the final draft and provided feedback to improve the quality and technical accuracy of this work.

My sincere thanks go to Mira. She corrected most of my English spelling mistakes and motivated me when I was lacking in drive.

This thesis would not have been possible without the support of my family who enabled me to study and earn a diploma degree in engineering. Thank you so much for your support during this long and demanding undertaking.

Table of Contents

1. Introduction

1.1. Motivation

In recent years, smartphones have evolved into powerful, tiny computers which empower sophisticated operating systems. With their large touch displays and the possibility to easily install and run native applications they experience unprecedented acceptance by their users. Apple's App Store currently distributes more than 350,000 native applications (apps) to iOS devices [APPL_PRESS_IOS43]. There exists a huge variety of apps, e.g., RSS feed readers, social network apps, instant messaging apps, to-do management apps, online banking apps, audio and video streaming apps, navigation apps, and a wide variety of games.

Users of mobile devices integrate them more and more into their everyday life, carrying more sensitive personal data than ever with them in their pockets - data such as private or corporate e-mails, contacts, calendar events, photos, but also credentials, e.g., to log into websites or access corporate networks. Businesses have begun developing and deploying iOS devices. These devices often run applications which access, process, and store sensitive corporate data. Thus, mobile devices with all the data on them are a worthwhile target for attackers.

In recent years, users of iOS devices have been the target of several attacks. In 2008, Apple pulled the game Aurora Feint from the App Store, because it transferred the contact list to a game server - unencrypted [AURORA_FEINT]. In 2009, users of the application mogoRoad received calls from the developer after installing the free version asking them to buy the full version [MOGOROAD]. The developer accessed the phone numbers of the devices on which the application was installed. Games from the developer Storm8 also sent the phone number to a server without the user's knowledge. Storm8 updated the corresponding games when this became public [STORM8]. With twelve games being in the App Store's top 100, over 20 million users installed games from Storm8 at that time. Flurry (former Pinch Media) and other analytics frameworks track the behavior and usage of iOS applications on the users device. When a developer uses an analytics framework, the application often transmits the unique device identifier (UDID) and even location data among other information to a server that collects the analytics data. Because the UDID is unique even across applications, the analytics companies are able to create fine-grained user profiles. The malware Ikee [IKEE], iPhone/Privacy.A [PRIVACY_A], and Ikee.B [IKEE_B] target jailbroken devices on which the user did not change the default SSH password. Although, malware on non-jailbroken devices has not been observed until the time of this writing.

In their Black Hat USA 2009 talk *Fuzzing the Phone in your Phone*, Collin Mulliner and Charlie Miller presented an approach for fuzzing SMS messages [SMS_FUZZING] to discover bugs on mobile devices. They found several bugs on iOS that could be used for Denial-of-Service attacks.
The Pwn2Own contest [PWN2OWN] at the annual conference CanSecWest, mobile devices are challenged to withstand the attacks of the attending security researchers. If a researcher or a team of researchers manage to successfully attack a device they get it and can take it home. Although the security of iOS-based devices has been improved constantly by Apple, the researchers managed to successfully hack the iPhone on all preceding contests.

Because iOS devices face an ever-growing threat, applications running on them have to be developed with security in mind. They have to leverage all the protection mechanisms of iOS as best as possible to protect their data. The scope of this work is to evaluate the field of mobile application security on the iOS operating system, which powers Apple's iPhone, iPod touch, and iPad. It analyzes what attackers can do to obtain personal data from iOS devices and evaluates the mechanisms third-party developers can use to protect their applications.

1.2. Objective

Work related to iOS security is most often about jailbreaking [WHY_JAILB] or gaining full access to the device by exploiting a vulnerability in the operating system [PWN2OWN]. Few work is dedicated to secure application development. The iOS Developer Library [IOS_REF_LIB], Apple's official reference documentation of the iOS SDK, contains a huge amount of documentation of the iOS SDK. However, documentation related to security topics is sometimes incomplete and often lacks important background information about the cryptographic mechanisms behind a specific API.

The objective of this work is to provide a deep insight into secure mobile application development for the iOS platform. This work explains the execution environment of third-party applications and discusses the mechanisms and APIs a developer can leverage to protect application data. It also provides missing background information about the cryptographic mechanisms behind some of the APIs.

The thesis shows the possibilities of a third-party developer who wants to steal the user's personal data from the system or other applications. It presents how an application can access certain sources of personal information on iOS (first shown by [SPYPHONE]) and discusses other relevant sources.
Application attacks like the ones presented by [DHANJANI_BHEU2011] are also discussed. The offensive part of this work concludes with a security analysis of selected iOS applications from the App Store.

The focus is on the official, i.e., non-jailbroken, version of iOS. This has the following reasons:

- iOS-based devices find their way more and more into the corporate world. Businesses have high security requirements. Therefore, most corporate policies prohibit employees from jailbreaking their device.
- Most people obtain applications from the App Store, because it is the only official source for applications. Applications distributed in the App Store are subject to the App Store Review Process and are signed by Apple.

Overall, this work aims to provide third-party iOS developers knowledge about secure iOS application development to leverage the security features and APIs of the platform in order to secure their applications and data as good as possible. Evaluating the security of iOS applications on condition of the device not being jailbroken is crucial during the inevitable adoption of iOS devices by the corporate world.

1.3. Related Work

Professional Cocoa Application Security by Graham J. Lee [P_C_A_S] discusses secure application development for Mac OS X and iOS. The focus of the book is mainly on Mac OS X with hints and references to iOS. It provides a good starting point for an experienced Mac or iOS developer who wants to know more about developing secure applications for those two operating systems. However, the book lacks detailed security-related information about often-used APIs such as the authentication methods of the URL Loading System and CFNetwork, Keychain access groups, and Data Protection. Background information about mechanisms like, e.g. PKCS #1 padding during RSA encryption and digital signature creation, is not provided as well. The book does not cover vulnerabilities and attack vectors of iOS applications.

Nicolas Seriot showed in his BlackHat DC 2010 talk *iPhone Privacy* [IPHONE_PIRACY] that a malicious third-party developer can access personal user data, e.g., Safari and YouTube searches, phone and e-mail account data, contacts, the content of the keyboard cache, locations of photos taken with the phone, GPS and Wi-Fi locations, without exploiting a vulnerability or using other offensive techniques to gain access to information. He developed the application SpyPhone [SPYPHONE] to demonstrate what he discovered. SpyPhone does not require the device to be jailbroken and could potentially pass the App Store review process because it only uses benign public APIs available to any third-party developer to harvest personal data.

Nitesh Dhanjani discusses in his BlackHat Europe 2011 talk *New Age Application Attacks Against Apple's iOS (and Countermeasures)* [DHANJANI_BHEU2011] some recently found attacks against iOS applications. He explains URL scheme

vulnerabilities, interface spoofing attacks, implementation details and deployment setups that lead to insecure usage of the Apple Push Notification Service, man-in-the-middle attacks made possible through insufficient trust evaluation, and discusses the new Data Protection API introduced by iOS 4.

The Mac Hacker's Handbook by Charlie Miller and Dino Dai Zovi [MAC_HACKER] explains the tools and strategies of attackers who target the Mac OS X platform. Because Mac OS X and iOS share many of their underlying mechanisms, the approaches in this book are interesting for the iOS platform as well.

Objective-C is the main programming language used for Mac and iOS application development. *The Objective-C Runtime: Understanding and Abusing* [PHRACK_OBJC] provides deep technical insight into the Objective-C runtime and how it can be abused by an attacker.

iPhone Forensics - Recovering Evidence, Personal Data, and Corporate Assets by Jonathan Zdziarski [ZDZIARSKI_FORENSIC] and *iOS Forensic Analysis* by Sean Morrissey [MORRISSEY_FORENSIC] discuss the forensic analysis of iOS devices. [ZDZIARSKI_FORENSIC] is slightly outdated because it was published in December 2008 when the latest mobile device from Apple was the iPhone 3G with iOS 2. [MORRISSEY_FORENSIC] was published in December 2010 and explores the forensic analysis of the current mobile devices from Apple.

The paper *PiOS: Detecting Privacy Leaks in iOS Applications* [PIOS] by Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna presents a way to detect if an iOS application leaks private data to third parties. The authors developed a plugin for the disassembler IDA Pro, called PiOS, that uses static binary analysis to examine an iOS application binary for privacy leaks. More than 1,400 applications from Apple's official App Store and Cydia, that is only available on jailbroken iOS devices, were analyzed. They found out that more than half the applications leak the Unique Device Identifier (UDID). Another result is that applications that are distributed through the App Store and are thus subject to the App Store Review Guidelines leak as much data as applications that are distributed through Cydia.

Jens Heider and Matthias Boll from Fraunhofer Institute for Secure Information Technology demonstrate in their paper *Lost iPhone? Lost Passwords!* [LOST_IPHONE] that an attacker with physical control over the device (e.g., the device was found or stolen) is able to obtain the plaintext Keychain data of certain Keychain items. The presented attack boots a jailbroken version of iOS, runs a script that extracts the Keychain data from the Keychain's SQLite database in the file system, and uses system functions of iOS to decrypt the sensitive data.

1.4. Outline

Chapter 2 presents security mechanisms of iOS devices: hardware security features used by iOS, the authenticated boot process, the mandatory code signing of applications, and the sandboxing of third-party applications. Chapter 3 explains secure development of iOS applications and how applications protect their data as effectively as possible. It discusses the security of Objective-C and Cocoa Touch as well as the usage of security-related APIs available to iOS developers. Chapter 4 discusses application attacks on non-jailbroken devices. Chapter 5 contains experimental results of this work: iOS applications that demonstrate the proper usage of security-related APIs and applications that implement specific attacks. Chapter 6 provides a conclusion and outlines possible future work in this area.

2. Overview: iOS Security Mechanisms

2.1. Security Features of iOS Devices

2.1.1. ARM TrustZone

TrustZone is a feature of certain ARM processors that enables them to simulate two virtual processors, called Normal World and Secure World, to separate security-critical instructions from regular instructions. The physical processor executes only instructions from one world at a given time. Hardware-based access control is used to manage the access from one world to the other. The Monitor Mode enables the processor to switch between the two worlds (Figure 2-1). The Secure Monitor Call instruction `SMC` triggers the processor's entry to Monitor Mode [TRUSTZONE].

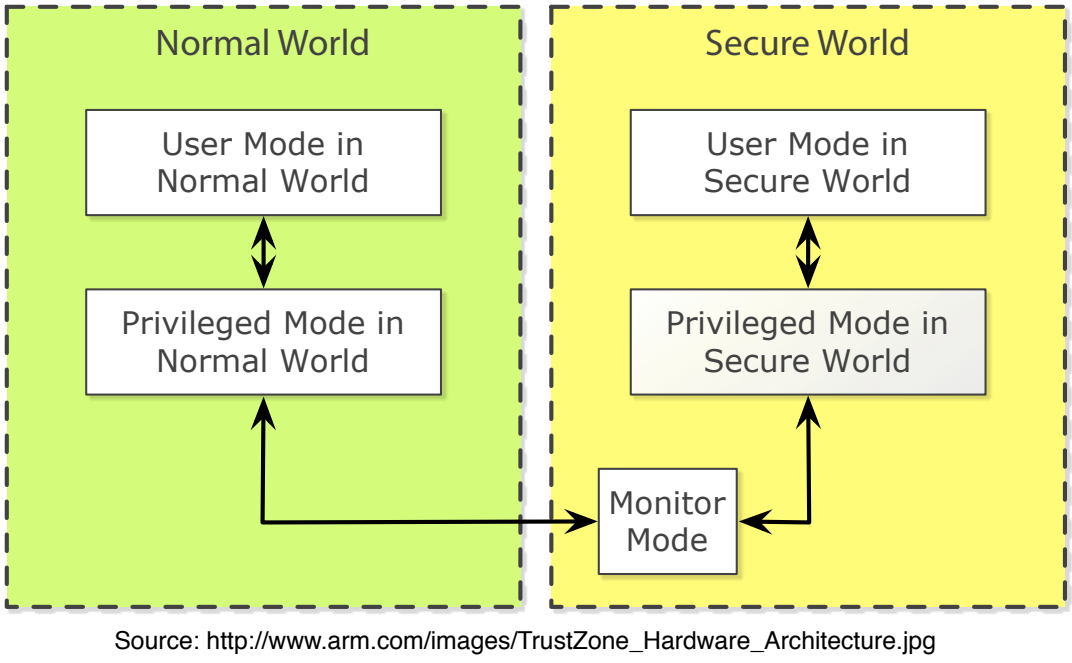

Figure 2-1: ARM TrustZone

The separation into two distinct execution environments can prevent the leakage of information from the Secure World to the Normal World. It can also prevent the introduction of malicious instructions from an exploited vulnerability in the Normal World into the Secure World.

iOS devices have had support for ARM TrustZone from the beginning [MORRISSEY_FORENSIC]. However, it was found that iOS does not use TrustZone. The disassembly of an iOS application binary, libSystem, and the binary of the Security framework did not yield a single call site where the instruction occurs. Listing 2-1

shows the commands used for searching the SMC instruction in the disassembled binaries.

```
$ otool -tv MyApp.app/MyApp | grep -i smc

$ otool -tv /Developer/Platforms/iPhoneOS.platform/Developer/SDKs/
iPhoneOS4.3.sdk/usr/lib/libSystem.dylib | grep -i smc

$ otool -tv /Developer/Platforms/iPhoneOS.platform/Developer/SDKs/
iPhoneOS4.3.sdk/System/Library/Frameworks/Security.framework/
Security | grep -i smc
```

Listing 2-1: Search for the usage of the SMC instruction

It would be interesting to know if Apple has any plans to use TrustZone in the future. If the necessity emerges, it can be used without breaking backwards-compatibility because all devices support it.

2.1.2. Data Execution Prevention

Data execution prevention (DEP) is the capability of the operating system to mark certain pages in memory as non-executable. On iOS devices, this is achieved with the *eXecute Never* (XN) bit which is supported by ARM processors as of the ARM architecture version ARMv6. iOS sets the XN bit on memory pages of the stack segment and the heap segment. Instructions located in memory pages which are configured as non-executable are not executed by the processor. DEP can prevent certain types of attacks where an attacker successfully exploits a vulnerability and writes malicious code onto the stack or heap. The processor refuses to execute that code, because the memory pages that contain it are marked as non-executable.

But that does not mean that an attacker cannot circumvent DEP to mount an attack. Return-oriented programming [ROP1], [ROP2], [ROP3] is an offensive technique that re-uses code already located in executable memory pages, e. g., from libraries linked into the attacked program. A short instruction sequence that performs a specific basic operation is called a *gadget*. An attacker can manipulate the stack in a way that the processor executes a chain of gadgets in succession, thereby executing the attacker's arbitrary code. Return-oriented programming relies on the knowledge of the addresses of the gadgets used for the attack. Before address space layout randomization was introduced in iOS 4.3, gadgets were always located at the same addresses in the virtual memory of the process, making a return-oriented programming attack very effective.

2.1.3. Address Space Layout Randomization

Address space layout randomization (ASLR) is a mechanism to randomize addresses of areas in the virtual address space of a program. The ASLR implementation in iOS randomizes the addresses of linked libraries every time the device restarts.

Prior to iOS 4.3, when ASLR was not available, libraries were always located at the same position in the process's address space. Thus, linked libraries always remained at the same address. Listing 2-2 shows the address of the linked Security framework library in an application running on an iPod touch 2nd Generation with iOS 4.2.1. The command `info shared` lists the shared libraries in the program being debugged. Libraries other than the Security framework are omitted in the output, because they show the same behavior discussed here.

```
(gdb) info shared
The DYLD shared library state has been initialized from the
executable's shared library information.  All symbols should be
present, but the addresses of some symbols may move when the program
is executed, as DYLD may relocate library load addresses if
necessary.

...

3 Security                      F 0x3564b000        dyld E E /
Developer/Platforms/iPhoneOS.platform/DeviceSupport/4.2.1 (8C148)/
Symbols/System/Library/Frameworks/Security.framework/Security at
0x3564b000 (offset 0x0)

...
```

Listing 2-2: iPod touch 2nd Generation with iOS 4.2.1 (no ASLR)

The offset of all libraries in the debugged process is always `0x0` bytes. On this device and with this iOS version, the Security framework dynamic library is always located at the address `0x3564b000`. Different libraries are located at different, fixed addresses. An attacker can utilize this knowledge to her advantage by invoking known library functions to execute malicious code. The attack is also known as *return-to-libc* attack [RET_LIBC_1], [RET_LIBC_2]. DEP does not prevent *return-to-libc* attacks, because the invoked library functions are contained in executable memory pages. DEP only prevents code from being executed if it is contained in non-executable memory pages.

Apple introduced ASLR in iOS 4.3. The address of a function that should be invoked during a *return-to-libc* attack has to be guessed. This mechanism makes it more difficult for the attack to succeed. Listing 2-3, Listing 2-4, and Listing 2-5 show the offset of the linked Security framework library when the application runs on devices which have ASLR enabled.

```
(gdb) info shared
The DYLD shared library state has been initialized from the
executable's shared library information.  All symbols should be
present, but the addresses of some symbols may move when the program
is executed, as DYLD may relocate library load addresses if
necessary.

...

3 Security                      F 0x3076c000       dyld E E /
Developer/Platforms/iPhoneOS.platform/DeviceSupport/4.3.1 (8G4)/
Symbols/System/Library/Frameworks/Security.framework/Security at
0x3076c000 (offset -0x5e45000)

...
```

Listing 2-3: iPhone 4 (1st device) with iOS 4.3.1 (ASLR)

```
(gdb) info shared
The DYLD shared library state has been initialized from the
executable's shared library information.  All symbols should be
present, but the addresses of some symbols may move when the program
is executed, as DYLD may relocate library load addresses if
necessary.

...

3 Security                      F 0x309b1000       dyld E E /
Developer/Platforms/iPhoneOS.platform/DeviceSupport/4.3.1 (8G4)/
Symbols/System/Library/Frameworks/Security.framework/Security at
0x309b1000 (offset -0x5c00000)

...
```

Listing 2-4: iPhone 4 (2nd device) with iOS 4.3.1 (ASLR)

```
(gdb) info shared
The DYLD shared library state has been initialized from the
executable's shared library information.  All symbols should be
present, but the addresses of some symbols may move when the program
is executed, as DYLD may relocate library load addresses if
necessary.

...

3 Security                      F 0x36c25000       dyld E E /
Developer/Platforms/iPhoneOS.platform/DeviceSupport/4.3.1 (8G4)/
Symbols/System/Library/Frameworks/Security.framework/Security at
0x36c25000 (offset 0x674000)

...
```

Listing 2-5: iPad 1 with iOS 4.3.1 (ASLR)

When different applications are debugged several times on the same device, it is observed that the offsets of the libraries are always the same. A reboot of one of the devices reveals that the offsets change after a reboot. Listing 2-6 shows the GDB output for the rebooted first iPhone 4.

```
(gdb) info shared
The DYLD shared library state has been initialized from the
executable's shared library information.  All symbols should be
present, but the addresses of some symbols may move when the program
is executed, as DYLD may relocate library load addresses if
necessary.

...

3 Security                      F 0x30557000        dyld E E /
Developer/Platforms/iPhoneOS.platform/DeviceSupport/4.3.1 (8G4)/
Symbols/System/Library/Frameworks/Security.framework/Security at
0x30557000 (offset -0x605a000)

...
```

Listing 2-6: iPhone 4 (1st device) with iOS 4.3.1 (ASLR) after reboot

2.2. Authenticated Boot Process

In their presentation *Hacking the iPhone* at the 25th Chaos Communication Congress (25C3) [HACK_IPHONE], pytey, planetbeing, and MuscleNerd demonstrated what they discovered about the boot process and security measures of the iPhone and iOS architecture. The talk was held in December 2008 when iOS 2 was the current version of iOS. Figure 2-2 shows the boot process presented in the talk.

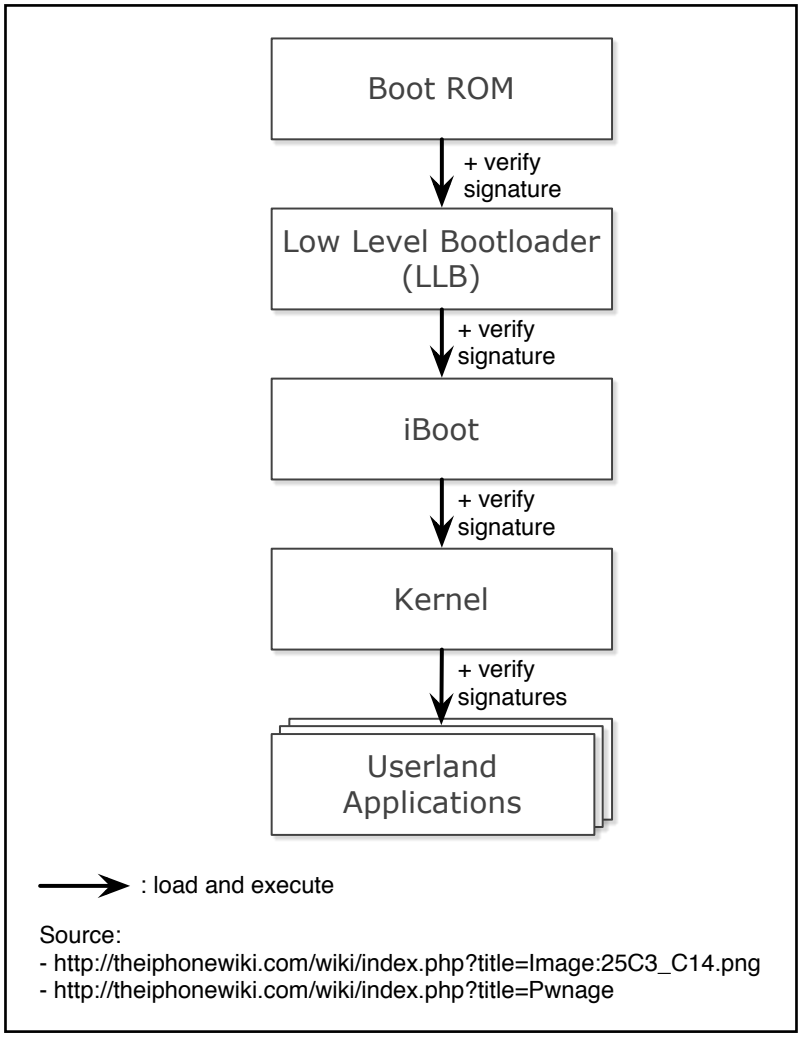

Figure 2-2: The authenticated boot process

From the Low Level Bootloader (LLB) to the userland applications, every program is digitally signed. A program that loads and executes another program first verifies the signature and only continues when the signature is valid.

- **Boot ROM:** The device's boot begins with the Boot ROM after a power on. The Boot ROM loads and executes the LLB, if the signature verification was successful.
- **LLB:** The LLB verifies the signature of the next stage boot loader iBoot. It loads and executes iBoot if the signature verification was successful.
- **iBoot:** iBoot loads a device tree and populates it with device-specific data (e.g., serial numbers, touch display initialization sequences, WiFi calibration data). Among other things, it verifies the signature of the device tree and the operating system's kernel. It loads and executes the kernel if the verifications were successful.
- **Kernel:** The kernel verifies a userland application's signature during the `execve()` system call. It loads and executes the application if the verification was successful.
- **Userland Applications:** All userland applications must be signed in order to be run by the operating system. The next chapter "Mandatory Code Signing" explains the code signing of userland applications in greater detail.

2.3. Mandatory Code Signing

Code signing is the process of digitally signing an application. The Code Signing Guide [CS_G] from Apple explains code signing as follows:

> *"Code signing is a technology [...] that ensures the integrity of code and allows the system to recognize updated versions of code as the same program as the original. [...] When code is signed, it is possible to determine reliably whether that code has been modified by someone other than the signer, no matter whether the modification was intentional (by a hacker, for example) or accidental (as when a file gets corrupted). In addition, by adding a code signature, a developer can ensure that updates to a program are valid and can be treated by the system as the same program as the previous version."*
>
> *"[...] a code signature consists of three parts:*
> - *A unique identifier, which can be used to identify the code or to determine to which groups or categories the code belongs. This identifier can be derived from the contents of the Info.plist for the program, or can be provided explicitly by the signer.*
> - *A seal, which is a collection of checksums or hashes of the various parts of the program, such as the identifier, the Info.plist, the main executable, the resource files, and so on. The seal can be used to detect alterations to the code and to the program identifier.*
> - *A digital signature, which signs the seal to guarantee its integrity. The signature includes information that can be used to determine who signed the code and whether the signature is valid."*

Only applications that are digitally signed are runnable on iOS. The loader in the operating system refuses to load applications with an invalid or missing signature. Furthermore, it only loads applications that are signed for one of the following distribution methods:

- **App Store:** The user downloads the application from Apple's official App Store. This is by far the most common way of installing applications. Before uploading an application to the App Store for review, it has to be signed by the developer. The Code Signing Guide points out that "[...] Apple does not sign applications that have not been signed by the developer, and applications not signed by Apple simply will not run" [S_O]. Apple replaces the signature of the developer with its own signature if the application successfully passes the review process. The reviewed and re-signed application is then distributed through the App Store.
- **Enterprise:** Companies with more than 500 employees are eligible to enroll in the iOS Developer Enterprise Program. This allows them to sign applications that can be distributed in-house without going through the App Store review process. Enterprise applications can be distributed via e-mail or one of the methods

described in the guide Distributing Enterprise Apps for iOS 4 Devices [DIST_ENT_IOS4].

- **Ad-hoc:** Builds of applications distributed to beta testers are signed by the developer. They can only run on at most 100 devices configured for this application in the iOS Provisioning Portal of the developer. Beta builds can be distributed via e-mail or one of the methods described in Distributing Enterprise Apps for iOS 4 Devices [DIST_ENT_IOS4].
- **Development:** A developer often needs to test an application's behavior on an actual device during development. In this scenario, apps are signed by the developer and have the same limitations regarding the devices that can run the application are the same as with applications distributed via Ad-Hoc Distribution.

Signing the application is the last step during the build process in Xcode (see Figure 2-3). This step invokes the commands shown in Listing 2-7. The first command sets the environment variable PATH to include the directory with the command line utilities of the iOS SDK. This ensures that the correct version of `codesign` is used if it also exists in other directories not related to the iOS SDK. The second step sets the path to the command-line utility `codesign_allocate` to be used during code signing. The man page of `codesign_allocate` describes it as a tool to *"add code signing data to a Mach-O file"* [CS_ALLOC_MAN]. The third command runs the `codesign` utility with a set of parameters:

- `-f`: If an existing signature is present, forces the replacement thereof.
- `-s "iPhone Developer: Manuel Binna (XXXXXXXXXX)"`: Signs the application with the private key that corresponds to the certificate in the computer's Keychain whose subject contains the given string.
- `--resource-rules=/path/to/ResourceRules.plist`: Defines "rules for collecting bundle resources to be sealed into the signature" [CS_MAN]. The product of the build process during the development of an iOS applications is a bundle that contains the executable together with its related resources like images and property list files. The signature creation signs the complete bundle. The rules that define how to perform this are provided by this parameter.
- `--entitlements /path/to/DataProtectionDemo.xcent`: Copies the content of the file DataProtectionDemo.xcent to the application bundle before creating the signature. Listing 2-8 shows the structure of a code signing entitlements file. It contains the application identifier and a list of Keychain access groups to which the application belongs. The chapter on Keychain Services explains the meaning of these values.
- `DataProtectionDemo.app`: The last parameter is the application bundle, a directory in the computer's filesystem.

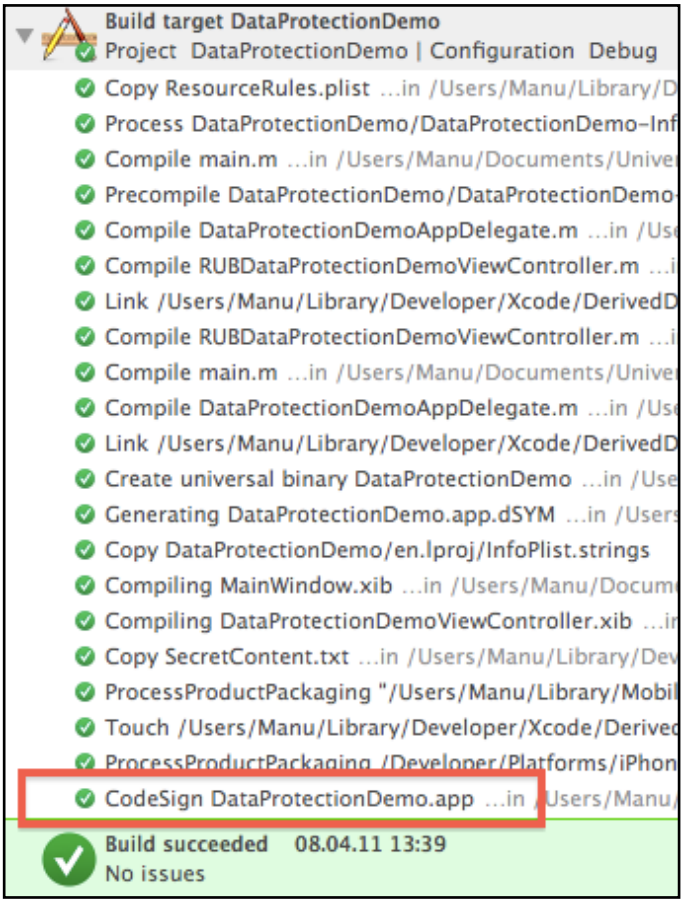

Figure 2-3: Code signing during Xcode's build process

```
$ setenv PATH "/Developer/Platforms/iPhoneOS.platform/Developer/usr/
bin:/Developer/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin"

$ setenv _CODESIGN_ALLOCATE_ /Developer/Platforms/iPhoneOS.platform/
Developer/usr/bin/codesign_allocate

$ /usr/bin/codesign -f -s "iPhone Developer: Manuel Binna
(XXXXXXXXXX)" --resource-rules=/path/to/ResourceRules.plist
--entitlements /path/to/DataProtectionDemo.xcent
DataProtectionDemo.app
```

Listing 2-7: Invocation of the codesign utility during Xcode's build process

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
      "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
      <key>application-identifier</key>
      <string>XXXXXXXXXX.de.rub.DataProtectionDemo</string>
      <key>get-task-allow</key>
      <true/>
      <key>keychain-access-groups</key>
      <array>
            <string>XXXXXXXXXX.de.rub.DataProtectionDemo</string>
      </array>
</dict>
</plist>
```

Listing 2-8: Code signing entitlements

Code signing adds extended attributes to the executable and creates new files in the application bundle. The application bundle is the folder that contains the application binary together with the resources of the application like, e.g., images, XIB files, the Info.plist. However, it does not modify existing files other than the executable. As mentioned in Professional Cocoa Application Security, the "signing process embeds the certificate associated with the signing identity in the signature attached to the signed application, along with any certificates needed to validate the identity of the signer" [P_C_A_S]. The Code Signing Guide explains that signing code is "fast, requires few resources, and increases the size of your deliverable by less than 1%" [CS_G].

The developer sets the requirements used to verify the signature of an application when she signs it. The designated requirement of a code signature is a special criterion. The verifier uses it to assert that the application is the same application that was verified before and should be treated as such.

Universal Binaries

The older devices iPhone 2G, iPhone 3G, and iPod touch (1st and 2nd Generation) are based on the ARMv6 architecture. The newer devices iPhone 3GS, iPhone 4, iPad 1, and iPad 2 also support the ARMv7 architecture. An iOS executable can be compiled for both architectures and merged into a single Mach-O binary. Such a Mach-O binary is called a Universal Binary.

The Code Signing Guide points out that "[...] the object code for each architecture is signed separately [...]" and "[...] usually only the native architecture on the end user's system is verified." [CS_G].

2.4. Application Sandboxing

iOS restricts applications by enforcing mandatory access controls (MAC) to the processes with a kernel extension (kext) called *Seatbelt* [SEATBELT]. iOS limits the files, directories, and system resources an application can access and enforces the rules that define how the application can access these items. This mechanism is called *sandboxing*. Apple's Security Overview [S_O] explains the function of the sandbox as follows:

> *"[...] every application is sandboxed during installation. The application, its preferences, and its data are restricted to a unique location in the file system and no application can access another application's preferences or data. In addition, an application running in iOS can see only its own keychain items."*
>
> *"[...] an application can call the public networking APIs to communicate over a network, but has no direct access to the communications or networking hardware."*

If an application with a vulnerability processes data that it received from an untrusted network without validating, an attacker can exploit the vulnerability and access the resources the application has access to. The objective of MAC in general is not to prevent an attack, but to limit the possibilities of attackers after a successful attack. Because MAC limits the access to the filesystem and resources, an attacker cannot access the resources of the operating system or other applications.

Apple provides no documentation about the architecture of the sandbox and how it is used to enforce mandatory access controls in the operating system. The paper *The Apple Sandbox* by Dionysus Blazakis [APPLE_SANDBOX] provides highly technical details to better understand the inner workings of the sandbox:

> *"The Sandbox framework, previously codenamed Seatbelt, provides fine-grained access control via Scheme policy definitions. The Sandbox is implemented as a policy module for the TrustedBSD mandatory access control (MAC) framework. The Sandbox framework adds significant value by providing a user-space configurable, per-process policy on top of the TrustedBSD system call hooking and policy management engine."*
>
> *"Once the sandbox is initialized, function calls hooked by the TrustedBSD layer will pass through Sandbox.kext for policy enforcement. Depending on the system call, the extension will consult the list of rules for the current process. Some rules [...] will require pattern matching support. Sandbox.kext imports functions from AppleMatch.kext to perform regular expression matching on the system call argument and the policy rule that is being checked."*

On Mac OS X, application processes can put themselves into a sandbox [SANDBOX_INIT_MAN]. The process can choose between the different profiles:

- **kSBXProfileNoInternet**: The application is prohibited from using TCP/IP networking.
- **kSBXProfileNoNetwork**: The application is prohibited from using socket-based network connections.
- **kSBXProfileNoWrite**: The application is prohibited from writing to the filesystem.
- **kSBXProfileNoWriteExceptTemporary**: The application can only write to temporary directories.
- **kSBXProfilePureComputation**: The application is prohibited from using all operating system services.

On iOS, the sandbox has no public API. In fact, there is also *"no API for changing the policy that applies to an application."* [P_C_A_S]. Seatbelt policies are written in the Scheme programing language. The page about Seatbelt in the iPhone Wiki [SEATBELT] shows a template of the sandbox policy that is applied to an application on iOS 2.

An application is sandboxed to a random directory in the directory `/var/mobile/Applications/` during its installation process. The directory name follows the convention `XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX` where every character `X` is in the range between 0 and 9 or between A and F. It could not be determined how iOS computes this name. For example, an iOS device can contain the following application directories:

- `317310A4-0765-42A9-975B-43E3309FDFDB`
- `C307E8E4-95F3-4CFE-B150-7BFE09FE5EDA`

iOS randomly generates these names upon installation and update of an application. The update process performs the following steps:

- Install the updated version of the application into a new random directory in `/var/mobile/Applications/`.
- Copy the resources from the old version's directory to the new version's directory .
- Remove the directory of the old application.

The application FSWalker [FSWALKER] demonstrates that the sandbox rules are relatively lax. An application can access many parts of the filesystem without using private APIs or requiring the device to be jailbroken. However, the App Store Review Guidelines explain that *"Apps that read or write data outside its designated container area will be rejected"* [ASRG].

2.5. Summary

With the authenticated boot process, the enforced signature checking of third-party applications, the application sandbox that restricts access of applications to the other parts of the system, DEP, and ASLR, iOS devices have many security features in place to protect the operating system and its applications against attacks. Although iOS has never used TrustZone, Apple can use it in future versions of iOS because all devices contain an ARM processor with TrustZone support.

3. Defense: Developing Secure iOS Applications

This chapter discusses the development of secure iOS applications. It shows how developers protect their data as effectively as possible by the usage of the APIs provided in the iOS SDK (Software Development Kit). After a discussion about the security of the Objective-C programming language and the Foundation framework, it demonstrates the usage of APIs that are related to security: Common Crypto, Keychain Services, Certificate, Key, and Trust Services, the class `UIWebView`, the URL Loading System, CFNetwork, and Data Protection.

3.1. Objective-C

Objective-C is the main programming language that is used to implement iOS applications. It is an object-oriented extension of the ANSI C programming language. The language is dynamic, meaning that many decisions are made at runtime and not at compile time or link time. Because of its dynamic nature, Objective-C needs a runtime environment for the execution of the compiled code [OBJC_RUNTIME_P_G].

Instance Variable Scopes

Instance variables (ivars) of Objective-C classes can have one of several scopes that specify their visibility within the program [OBJC_PROG_LANG].

- **`@private`**: The ivar is only visible to the class in which it is declared.
- **`@protected`**: The ivar is visible to the class in which it is declared and to all classes that inherit from this class. This is the default scope.
- **`@package`**: This scope is often used in frameworks, where a specific ivar is visible globally within the framework binary but not visible to other parts of the program the framework is linked into.
- **`@public`**: The ivar is visible from everywhere in the program.

Accessing a `@private` ivar with the struct dereference operator –> causes a compiler error as Listing 3-1 shows.

```
// IvarAccess.h
@interface IvarAccess : NSObject {
    @private
    NSUInteger x;
}

@property(nonatomic, readonly) NSUInteger x;

@end

// IvarAccess.m
@implementation IvarAccess

@synthesize x;

- (id)init {
    if ((self = [super init])) {
        x = 3;
    }
    return self;
}

@end

// main.c
int main(int argc, char *argv[]) {
  // ...

  IvarAccess *ivarAccess = [[[IvarAccess alloc] init] autorelease];
  ivarAccess->x = 1337;  // This line causes a compiler error

  // ...
}
```

Listing 3-1: Accessing a `@private` instance variable

However, this does not mean that code from other parts of the program cannot access it [P_C_A_S]. Key-Value Coding (KVC) is a mechanism that enables applications to access ivars and properties by name instead of accessing ivars directly or invoking accessor methods [KVC_PROG_G]. KVC tries to access the property that matches the provided key by invoking the appropriate accessor method. If the KVC mechanism cannot find an accessor method, it tries to access the instance variable directly. Listing 3-2 shows that KVC can modify the value of an instance variable with `@private` scope that corresponds to a read-only property.

```
// main.c
int main(int argc, char *argv[]) {
  // ...

  IvarAccess *ivarAccess = [[[IvarAccess alloc] init] autorelease];

  // Modify private ivar x via KVC
  [ivarAccess setValue:[NSNumber numberWithUnsignedInteger:1337]
                forKey:@"x"];
  // ivarAccess.x is now 1337

  // ...
}
```

Listing 3-2: Accessing a `@private` instance variable with KVC

This behavior can be modified by implementing the class method `accessInstanceVariablesDirectly` defined by the informal protocol `NSKeyValueCoding` and returning `NO`. The default implementation of that method returns `YES`. Listing 3-3 shows how to restrict the KVC mechanism from accessing the instance variables directly if it does not find a suitable accessor method.

An informal protocol is a set of methods that a class can implement in order to modify the behavior of another class. There is no language support for informal protocols in Objective-C. One possible way to implement an informal protocol is the definition of a category on `NSObject`. `NSKeyValueCoding` is actually a category on `NSObject`. Informal protocols are the only way to declare interfaces of a class in Objective-C prior to version 2.0 of the language specification. Formal protocols are available since Objective-C 2.0. A formal protocol is declared with the `@protocol` keyword. It is equivalent to an interface in the Java programming language. Formal protocols can declare required and optional methods. A class which conforms to a formal protocol has to explicitly list the protocol's name in its `@class` definition.

```
// IvarAccess.h
@interface IvarAccess : NSObject {
    @private
    NSUInteger x;
}

@property(nonatomic, readonly) NSUInteger x;

@end

// IvarAccess.m
@implementation IvarAccess

@synthesize x;

- (id)init {
    if ((self = [super init])) {
        x = 3;
    }
    return self;
}

// Restrict KVO from accessing ivars directly
+ (BOOL)accessInstanceVariablesDirectly {
    return NO;
}

@end

// main.c
int main(int argc, char *argv[]) {
  // ...

  IvarAccess *ivarAccess = [[[IvarAccess alloc] init] autorelease];

  // This would cause the program to crash, because the property is
  // declared as read-only and KVO cannot access the ivar directly.
  [ivarAccess setValue:[NSNumber numberWithUnsignedInteger:1337]
                forKey:@"x"];

  // ...
}
```

Listing 3-3: Restricting KVO from accessing ivars directly

Even if KVO is restricted from accessing instance variables directly, a category can still access `@private` instance variables. This should be kept in mind when including open source code from other developers into an application. This code could introduce categories on classes or superclasses from the application that accesses `@private` instance variables. Listing 3-4 shows how a category accesses a `@private` instance variable of another class.

```
// EvilCategory.h
@interface IvarAccess (EvilCategory)

- (void)accessPrivateIvar;

@end

// EvilCategory.m
@implementation IvarAccess (EvilCategory)

- (void)accessPrivateIvar {
    self->x = 1337;
}

@end

// main.c
int main(int argc, char *argv[]) {
  // ...

  IvarAccess *ivarAccess = [[[IvarAccess alloc] init] autorelease];

  // Access private ivar in the category method
  [ivarAccess accessPrivateIvar];
  // ivarAccess.x is now 1337

  // ...
}
```

Listing 3-4: Accessing a `@private` ivar in a category

Methods

In Objective-C, a method is invoked by sending a message to the receiver (either a class or an instance of a class):

```
[receiver message];
```

In this expression, the message `message` is sent to `receiver`. The Objective-C compiler converts the expression into a call to the Objective-C Runtime function `objc_msgSend()` with the receiver as the first parameter and the selector of the message as the second parameter:

```
objc_msgSend(receiver, @selector(message));
```

A selector is the number that represents a method in the selector table of a class. A selector is represented by the type `SEL`. A selector corresponds to an `IMP`, a function pointer in the selector table that references the actual code of the method. The `IMP` is defined as a function pointer to a function that takes an object of type `id` as the first parameter, a selector of type `SEL` as the second parameter, and returns an object of type `id`:

```
typedef id (*IMP)(id self, SEL _cmd, ...);
```

When an instance is sent a message, the Objective-C Runtime looks up the selector for this message and obtains the `IMP` for this selector. It then dereferences the function pointer, thus invoking the function.

The Objective-C compiler creates an entry for every method of the class in the selector table which contains a selector (`SEL`) and a function pointer (`IMP`) to the function that implements the code of the method. The same applies to methods in categories of this class. All methods, private and public, are invokable through the Objective-C Runtime in this way. Therefore, all Objective-C methods are invokable from everywhere in the program, regardless if the method is declared in a public header file or not. Private methods are not declared in the public header of a class or in a category. But this does not stop other code from invoking them. `NSObject` is the base class in Cocoa Touch applications for iOS. The method `performSelector:` of `NSObject` expects a selector (`SEL`) as its first parameter and causes the Objective-C Runtime to invoke the corresponding method implementation. The following expression allows other code of the program to invoke the private method `privateMethod` of `securityCriticalObject` without even causing the compiler to emit a warning:

```
[securityCriticalObject performSelector:@selector(privateMethod)];
```

A possible solution is to implement the actual code of the private method in a static function within the implementation file of the class. C functions are not added to the Objective-C Runtime and therefore cannot be called with the message sending mechanism explained earlier. Static functions are only visible within the file in which they are implemented. Listing 3-5 shows how to implement a static function and access it from within a private method of the class. The function IvarAccess_privateCalculation() cannot be invoked from outside the class because it is a static function.

```
// IvarAccess.h

@interface IvarAccess : NSObject
{

}

@end


// IvarAccess.m

static int IvarAccess_privateCalculation(int x, int y)
{
  // Perform a private calculation with x and y
  int calculatedValue = ...;

  return calculatedValue;
}

@implementation IvarAccess

- (void)privateMethod
{
  int x = ...;
  int y = ...;

  int secret = IvarAccess_privateCalculation(x, y);
  // Do something with secret value
}

@end
```

Listing 3-5: Access a static function within the class implementation

3.2. Foundation

The Foundation framework is the collection of Objective-C classes that is used by all iOS applications. Foundation is the layer on which UIKit and many other frameworks of Cocoa Touch are built upon. It provides classes and mechanisms that are useful in almost every computer program, among others:

- Collections / data structures
- Strings with Unicode support
- Memory management
- Persistence
- Concurrency
- Date and time
- Network communication
- Files

Strings

The class `NSString` and its public subclass `NSMutableString` provide methods to initialize a new instance with a format string that is processed during the initialization. Cocoa strings are instances or literals of such subclasses. Format strings are used in many places in iOS applications. The function `NSLog()` expects a format string that is printed to the console after being processed (see Listing 3-6). The developer has to be aware of the danger of format strings and use them in a secure manner. The code in Listing 3-6 shows the secure usage of format strings in Cocoa. The format string should always be provided explicitly.

```
NSString *username = @“Manuel”;

// Result: @“Hello Manuel, welcome back!”
NSString *message = [NSString stringWithFormat:
                     @”Hello %@, welcome back!”, username];

// Result: @”Hello, Manuel”
NSLog(@”Hello, %@”, username);
```

Listing 3-6: Secure format strings in Foundation

Listing 3-7 shows the instantiation of a string without explicitly providing the format string. If the attacker controls the value of the instance `username`, she can use this vulnerability to mount a format string attack.

```
NSString *username = ...;     // Controlled by the attacker.

// Attacker has full control over the format string.
NSString *message = [NSString stringWithFormat:username];
```

Listing 3-7: Insecure format strings in Foundation

The Foundation framework allows the conversion between C strings and Cocoa strings (see Listing 3-8). C strings are character pointers of type `char *` that are explicitly terminated by a `'\0'` character. Cocoa strings are instances of `NSString` or a subclass of `NSString`. In addition to the array of Unicode characters, Cocoa strings store the count of their characters. This allows functions and methods to determine their length without relying on the presence of the termination character `'\0'`. This approach is more secure compared to the approach that C strings use [P_C_A_S].

```
NSString *cocoaString = @"Hello World";

// Create a C string from the Cocoa string
const char *cString = [cocoaString UTF8String];

// Create a Cocoa string from the C string
NSString *anotherCococaString =
  [NSString stringWithCString:cString
                     encoding:NSUTF8StringEncoding];
```

Listing 3-8: C strings and Cocoa strings

Security issues can arise when C strings are used [STACK_SMASH]. If not terminated correctly, they might enable an attacker to mount a buffer overflow attack. DEP and ASLR limit the effects of such an attack. The code injected by the attacker is not executable, because it resides in non-executable memory pages (the stack and heap segments are marked as non-executable by DEP). The locations of linked libraries and thus gadgets used for return-oriented programming are randomized by ASLR. Nevertheless, exploiting a buffer overflow vulnerability might be the first step of a sequence of several other steps to execute arbitrary malicious code. Developers should therefore take great care when dealing with C strings and use `NSString` or `NSMutableString` whenever possible.

Collections

Foundation contains different collection classes like `NSArray`, `NSDictionary`, `NSSet`, and their mutable subclasses. These classes provide variadic methods to initialize the collection with existing objects. Variadic methods are methods with a variable number of parameters. The end of the parameter list is defined by the sentinel `nil`. If the developer does not add the sentinel as the last parameter of the invocation of the variadic method, the method continues reading after the end of the parameter list. Listing 3-9 shows the appropriate and inappropriate use of one of the initializers of `NSArray`.

```
// Correct
NSArray *names = [NSArray arrayWithObjects:@"Manuel", @"Raphael",
                  @"Felix", @"Mira", nil];

// Wrong
NSArray *names = [NSArray arrayWithObjects:@"Manuel", @"Raphael",
                  @"Felix", @"Mira"];
```

Listing 3-9: Initializing an NSArray

Data

NSData and its subclass NSMutableData are wrappers around byte buffers. They provide an object-oriented interface for working with binary data and manage the automatic allocation and deallocation of the memory for the internal byte buffer. During initialization, an instance copies bytes from the byte buffer provided as a parameter to the internal byte buffer. The automatic management of the underlying byte buffer ensures that it is always available during the lifetime of the NSData instance and not freed prematurely (see Listing 3-10).

```
char *buffer = (char *) calloc(3, sizeof(*buffer));
buffer[0] = 'H';
buffer[1] = 'i';
buffer[2] = '\0';

// 'data' stores the characters of 'buffer' in its internal buffer
NSData *data = [NSData dataWithBytes:buffer
                              length:strlen(buffer)];

// Calling free() here is ok.
free(buffer);
```

Listing 3-10: Secure use of NSData

However, this behavior is optional. Instances can also use the byte buffer that is provided to the initializer as the storage for their data. If the buffer is freed during the lifetime of the instance and the memory re-used for other data, the instance can potentially read and write to memory not intended by the developer (see Listing 3-11).

```
char *buffer = (char *) calloc(3, sizeof(*buffer));
buffer[0] = 'H';
buffer[1] = 'i';
buffer[2] = '\0';

// 'data' takes ownership of 'buffer' and releases it on
// deallocation. 'NoCopy' means that the bytes of 'buffer' are not
// copied to an internal buffer.
NSData *data = [NSData dataWithBytesNoCopy:buffer
                                    length:strlen(buffer)];

// Calling free() here can cause a vulnerability.
free(buffer);
```

Listing 3-11: Insecure use of `NSData`

NSCoding and Property List Serialization

Foundation provides mechanisms for object persistency. The formal protocol `NSCoding` allows the persistence of custom objects and Property List Serialization allows the serialization of the classes `NSArray`, `NSDictionary`, `NSData`, `NSDate`, `NSString`, and `NSNumber` as a property list file. Both mechanisms allow the serialization of objects into a byte stream and the storage of the stream as a file in the file system. At a later time, the application can read the file and deserialize the contained objects. When serialized objects are stored on disk, an attacker can potentially manipulate them. The deserialized objects have to be validated before they are used, e.g., if an instance is really a kind of an assumed class and if its values are valid.

Files

Working with temporary files can be subject to race conditions. If an application creates a temporary file and opens it in a second step, an attacker might be able to remove the file and replace it with a file that is under the attacker's control. If the original application writes sensitive data to the file, the attacker can read it. If the attacker replaced the original temporary file with a symbolic link to another file and the original application writes data to the symbolic link, it might inadvertently overwrite existing data. One possibility to mitigate this problem is to obtain the file descriptor by creating and opening the file in one step and using the file descriptor instead the path to reference the file. Another improvement is to use a temporary directory that is neither known nor accessible by the attacker. The functions `NSDocumentDirectory()`, `NSCachesDirectory()`, and `NSTemporaryDirectory()` return paths to the appropriate directories within the calling application's home directory. On non-jailbroken devices, applications cannot access the home directory of another application. Files created in the home directory or any of its subdirectories are therefore not at risk from being manipulated by other third-party applications.

3.3. Security-related APIs

3.3.1. Randomization Services

Many cryptographic functions need random numbers. For example, a block cipher operating in CBC mode needs random initialization vectors to protect the first ciphertext block and cryptographic protocols use random numbers in their key agreement to obtain session keys between two parties. It is mandatory that random numbers used for cryptographic purposes are cryptographically secure, i.e. not predictable by an adversary. Not using cryptographically secure random numbers would allow the adversary to mount a successful attack with a higher probability. Randomization Services is the API to obtain cryptographically secure pseudo-random numbers on iOS devices.

Currently, the API consists of the function `SecRandomCopyBytes()` and the constant `kSecRandomDefault` with the value `NULL`. `kSecRandomDefault` is the system's default random number generator. Randomization Services obtain the amount of requested random bytes and return them indirectly as a byte array to the caller (see Listing 3-12).

```
size_t numberOfBytes = 1024;
uint8_t initializationVector[numberOfBytes];
SecRandomCopyBytes(kSecRandomDefault,
                   (size_t) sizeof(initializationVector),
                   initializationVector);
```

Listing 3-12: Obtaining 1024 random bytes

3.3.2. Common Crypto

Common Crypto is an API in the Core OS layer and is part of libSystem, which itself is a library that is dynamically linked into every iOS application. It provides symmetric encryption and decryption, hash functions, and hashed message authentication codes (HMACs). The security-related APIs in the higher iOS platform layers (discussed below) use these underlying cryptographic primitives to accomplish their tasks. Common Crypto is open source [CC_SRC].

Several different cryptographic primitives compose Common Crypto. It contains the block ciphers AES, DES, 3DES, CAST-128, and RC2. AES can be used with the key sizes 128 bit, 192 bit, and 256 bit. DES uses the key size 64 bit, whereas 3DES uses 192 bit. The key size of CAST-128 ranges from 40 bit to 128 bit but can only be incremented by a multiple of 8 bits. RC2 uses key sizes between 8 bit and 1024 bit. RC4 is the only available stream cipher. Common Crypto's supported hash functions are MD2, MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. Common Crypto can use the

hash functions MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 to derive an HMAC.

A block cipher operates in one of several different modes of operation. The mode of operation denotes the way in which a block cipher processes consecutive blocks of input when the input as a whole is larger than one block. The block ciphers in Common Crypto support the electronic code book (ECB) mode and the cipher block chaining (CBC) mode. The stream cipher RC4 inherently uses the ECB mode.

Figure 3-1 and 3-2 illustrate the encryption and decryption with the ECB mode. The input (plaintext) is processed block by block. The block size depends on the algorithm. AES uses a block size of 128 bits. DES, 3DES, CAST, and RC2 use a block size of 64 bits. In this mode the block cipher encrypts each block independently from the other blocks.

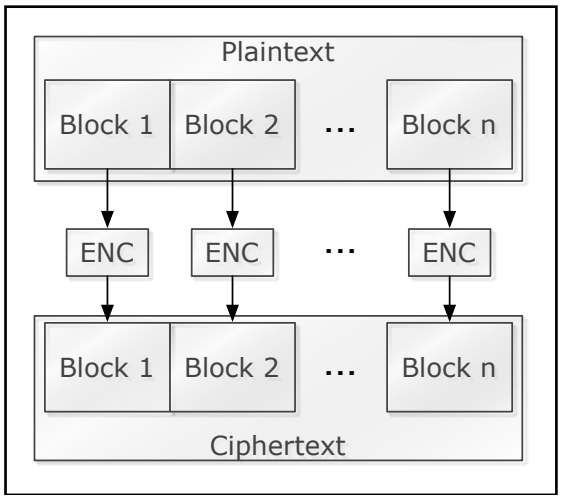

Figure 3-1: ECB mode (encryption)

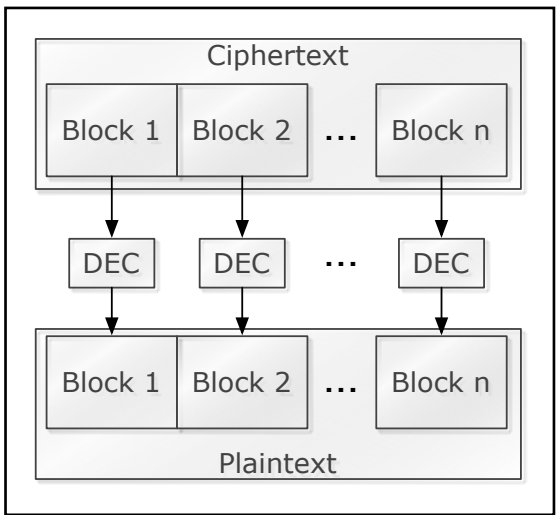

Figure 3-2: ECB mode (decryption)

The disadvantages of the ECB mode are [HAC]:

- If two distinct blocks contain the same plaintext data, the two ciphertexts are also the same. The header of a file is often public information. If an attacker knows the file type of the encrypted file, she has some plaintext/ciphertext pairs and is able to learn something about the key.
- An attacker can replace one or more blocks of the ciphertext without affecting the decryption of the remaining ciphertext blocks.
- Patterns in the plaintext are not hidden in the corresponding ciphertext, because the ECB mode does not perturb these patterns.

A more secure mode of operation is the CBC mode (Figure 3-3 and 3-4). Every ciphertext block directly affects the encryption of the next block. Before a plaintext block is actually encrypted it is XORed with the previously encrypted block. The first block has no preceding encrypted block with which it could be XORed, therefore a (usually random) initialization vector (IV) is used for that purpose. The size of the IV is the block size. The block cipher performs this procedure in reverse order when decrypting a ciphertext in CBC mode.

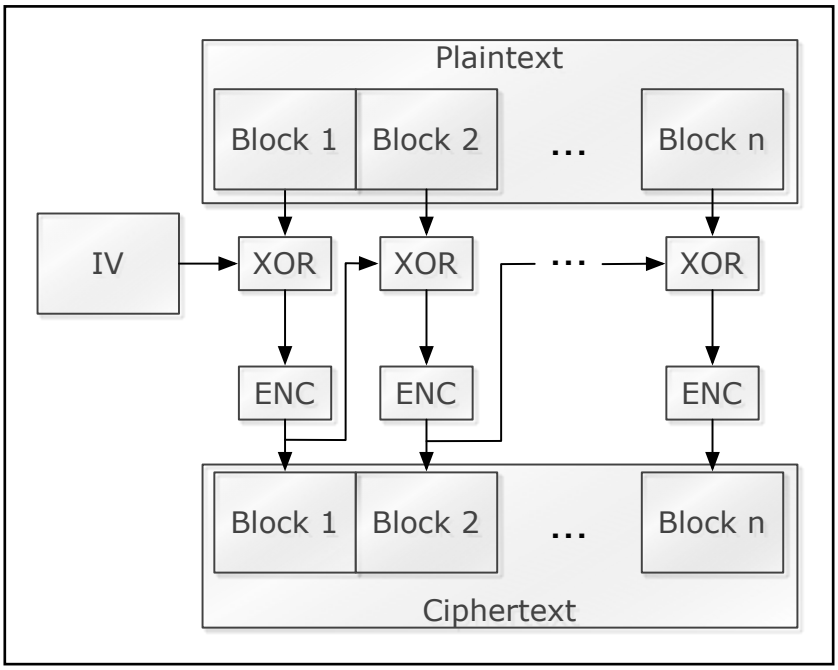

Figure 3-3: CBC mode (encryption)

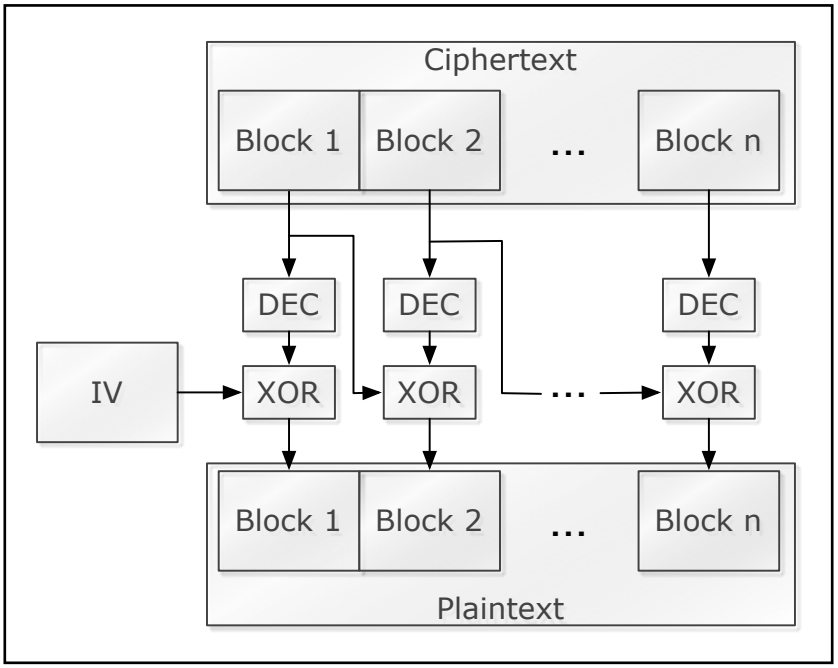

Figure 3-4: CBC mode (decryption)

Advantages of the CBC mode compared to the ECB mode are [HAC]:

- When using a random IV, two plaintext blocks with the same data result in different ciphertext blocks.
- The CBC mode perturbs patterns in the plaintext through the chaining of blocks.
- An attacker cannot replace one or more ciphertext blocks because every block depends on all previous blocks. Replacing one or more blocks of the ciphertext will yield a different plaintext after decryption.

A random IV that is only used once for encryption ensures that the ciphertext is always different, even if the same plaintext is encrypted twice. It is important that the integrity of the IV is protected [HAC]. If an attacker manages to convince the receiver of the encrypted message to use a manipulated IV, he/she is able to control certain bits in the first block of the plaintext. One method to protect the IV is to encrypt it with the ECB mode. Because the IV's length is exactly the length of one block, the CBC mode does not enhance the protection in this situation.

Padding

If the length of the plaintext is not an integral multiple of the block size, the plaintext must be padded up to the next integral multiple of the block size. Hence, padding only affects the last block (see Figure 3-5).

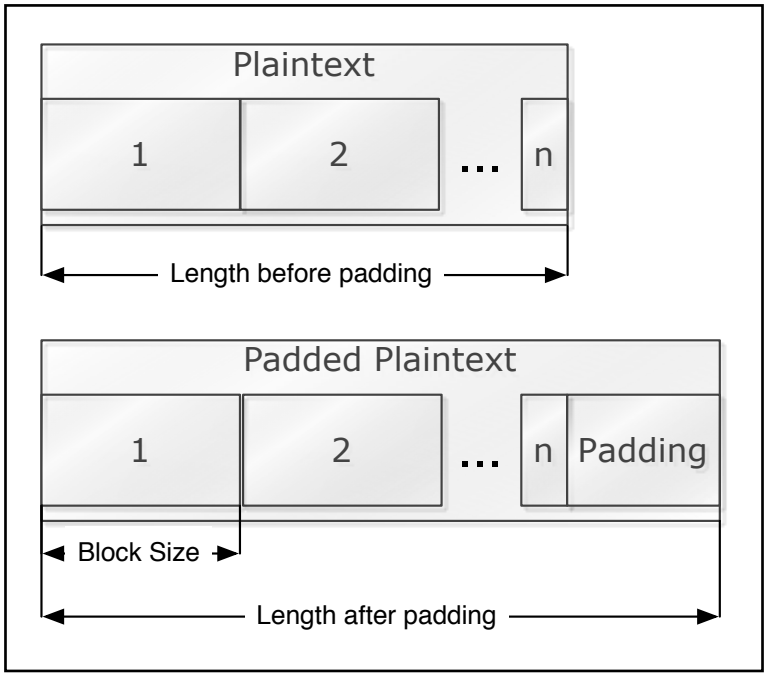

Figure 3-5: Padding

Padding is only necessary when using block ciphers since stream ciphers inherently operate on single bits instead of blocks. Common Crypto uses the padding described by PKCS #7 [PKCS7]. Before actually padding the last block, the count of byes to be padded is determined. The padding algorithm writes this count value exactly count times to the end of the last block. After the padding, the size of the last block is the block size of the cipher. If the size of the last block, e.g., block n in Figure 3-5, is 3 bytes and the block size of the cipher is 8 bytes, the padding algorithm will write the value 0x05 exactly 5 times.

3.3.2.1. Deriving a Digest

For every supported hash function Common Crypto provides four methods to derive a digest [CC_MANPAGES] (In Listing 3-13, HASH is a placeholder for one of the following: MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512):

```
// Process input data in one chunk
extern unsigned char *
CC_HASH(CC_HASH_CTC *c);

// Initialize context
extern int
CC_HASH_INIT(CC_HASH_CTX *c);

// Process current chunk
extern int
CC_HASH_UPDATE(CC_HASH_CTX *c, const void *data, CC_LONG len);

// Finalize derivation
extern int
CC_HASH_FINAL(unsigned char *md, CC_HASH_CTX *c);
```

Listing 3-13: Function prototypes for digest derivation

Listing 3-14 shows how to derive a digest from a single chunk of data with the function `CC_HASH()`. First, memory to store the digest is allocated. Common Crypto defines preprocessor macros for the different digest lengths of the hash functions. The second and final step is to call one of the functions to derive a digest from the given input. A pointer to the input data, the length of the input data, and a pointer to the allocated memory is provided to the function as parameters.

```
// 'data' is an NSData which contains the input data
unsigned char digest[CC_SHA256_DIGEST_LENGTH];
CC_SHA256([data bytes], [data length], digest);
// Buffer 'digest' now contains the hash digest
```

Listing 3-14: Derive a SHA-256 digest from the input data with a single invocation

Besides the functions which derive the digest with a single invocation, Common Crypto also provides a mechanism which allows to process the input data in chunks (see Listing 3-15). Processing large input data in smaller chunks is often the preferred way, because processing the data in one single chunk (like in Listing 3-14) may negatively affect the application's performance by blocking the current thread and taking up more memory. An instance of the opaque type `CC_HASH_CTX` keeps the state between the multiple invocations of the `CC_HASH_Update()` functions. An opaque type is a C struct with private implementation details. There exist init, update, and final functions for each of the supported hash functions. The name of the hash function is a part of the function name. To refer to a specific function without referring to a specific hash function, this paper uses the placeholder HASH instead of one of the algorithm names. Another scenario is the digest derivation of a very large file which does not fit into memory. The application can read the data from the file chunk-wise with `NSInputStream` or `CFReadStream` while only storing the current chunk in memory and processing it with Common Crypto.

```
// 'chunkData' is an NSData which contains the current chunk of
// input data
unsigned char digest[CC_SHA256_DIGEST_LENGTH];
CC_SHA256_CTX context;
CC_SHA256_Init(&context);

// Other processing ...

// Provide the current chunk of data to Common Crypto
CC_SHA256_Update(&context, [chunkData bytes], [chunkData length]);

// Other processing ...

CC_SHA256_Final(digest, &context);
// Buffer 'digest' now contains the hash digest
```

Listing 3-15: Derive a SHA-256 digest by processing the input data in chunks

3.3.2.2. Deriving an HMAC

The functions to derive an HMAC work similarly to those to derive a hash digest. The only difference is that the HMAC functions expect two additional parameters: a pointer to the key data and the key length.

3.3.2.3. Encrypting and Decrypting Data

The functions to perform encryption and decryption of data work similar to those to derive a digest or an HMAC [CC_MANPAGES]. `CCCrypt()` processes the input in a single function call (see Listing 3-16).

```
// Process input data in one chunk
CCCryptorStatus
CCCrypt(CCOperation op,
        CCAlgorithm alg,
        CCOptions options,
        const void *key,
        size_t keyLength,
        const void *iv,
        const void *dataIn,
        size_t dataInLength,
        void *dataOut,
        size_t dataOutAvailable,
        size_t *dataOutMoved);
```

Listing 3-16: `CCCrypt()` provides processing the input data via one function call

The functions in Listing 3-17 allow the chunk-wise processing of data. An opaque reference of type `CCCryptorRef` keeps the state between the different function

invocations. An opaque reference is a pointer to a C struct with private implementation details. Only one thread at a time can use a given `CCCryptorRef`, but different threads may use different `CCCryptorRefs` at the same time.

`CCCryptorRelease()` releases the memory of a `CCCryptorRef` after automatically overwriting it with zeros. Alternatively, a `CCCryptorRef` can be re-used for another operation with the same key (encryption or decryption, depending on which was set when `CCCryptorCreate()` was called for that `CCCryptoRef`). `CCCryptorReset()` has to be called before actually re-using the `CCCryptorRef`. This function sets a new initialization vector to be used for the next operation.

```
// Create context
CCCryptorStatus
CCCryptorCreate(CCOperation op,
                CCAlgorithm alg,
                CCOptions options,
                const void *key,
                size_t keyLength,
                const void *iv,
                CCCryptorRef *cryptorRef);

// Get required buffer length to store output
size_t
CCCryptorGetOutputLength(CCCryptorRef cryptorRef,
                         size_t inputLength,
                         bool final);

// Process current chunk
CCCryptorStatus
CCCryptorUpdate(CCCryptorRef cryptorRef,
                const void *dataIn,
                size_t dataInLength,
                void *dataOut,
                size_t dataOutAvailable,
                size_t *dataOutMoved);

// Finalize
CCCryptorStatus
CCCryptorFinal(CCCryptorRef cryptorRef,
               void *dataOut,
               size_t dataOutAvailable,
               size_t *dataOutMoved);

// Release memory (after overwriting it with zeros)
CCCryptorStatus
CCCryptorRelease(CCCryptorRef cryptorRef);

// Prepare 'cryptorRef' for re-usage (by setting a new IV)
CCCryptorStatus
CCCryptorReset(CCCryptorRef cryptorRef, const void *iv);
```

Listing 3-17: Function prototypes for chunk-wise processing of input data

Listing 3-18 shows how to encrypt data with AES-256, CBC mode, and PKCS #7 padding. In this sample listing, the cryptographic key consists only of zeros. In a real-world application the key is likely to be derived from a password with, e.g., PBKDF2 (Password-Based Key Derivation Function 2 (see [RFC2898] chapter 5.2)), or randomly generated with `SecRandomCopyBytes()` when used in combination with an asymmetric cryptosystem.

```
// 'plainData' is an instance of class NSData

// Create a cryptographic key
// ATTENTION: 'secretKey' is filled with ones and is for
// demonstration purposes only!!!
char secretKey[kCCKeySizeAES256];
memset((void *) secretKey, 0x1, (size_t) sizeof(secretKey));

// Obtain a new random initialization vector
uint8_t initializationVector[kCCBlockSizeAES128];
SecRandomCopyBytes(kSecRandomDefault,
                   sizeof(initializationVector),
                   initializationVector);

// Create storage for output
NSUInteger outputLength = [plainData length] + kCCBlockSizeAES128;
NSMutableData *cipherMutableData =
      [NSMutableData dataWithLength:outputLength];

// CCCrypt() indirectly returns number of encrypted bytes
size_t numBytesEncrypted = 0;

// Encrypt 'plainData' with 'secretKey' and 'initializationVector'
CCCryptorStatus status = CCCrypt(kCCEncrypt,
                                 kCCAlgorithmAES128,
                                 kCCOptionPKCS7Padding,
                                 secretKey,
                                 kCCKeySizeAES256,
                                 initializationVector,
                                 [plainData bytes],
                                 [plainData length],
                                 [cipherMutableData mutableBytes],
                                 [cipherMutableData length],
                                 &numBytesEncrypted);

// 'cipherMutableData' contains encrypted data
```

Listing 3-18: Encrypting data with AES-256, CBC mode, and PKCS #7 padding

3.3.2.4. Summary

Common Crypto provides a simple API to symmetrically encrypt/decrypt data and to derive digests and HMACs. Both a chunk-wise and single invocation approach are available. The most important symmetric encryption algorithms (e.g. AES, 3DES) and hash functions (e.g. SHA1, SHA-256) currently used are supported. However,

Common Crypto does not support many encryption algorithms. E.g., Blowfish, Twofish, Serpent, IDEA, or Camellia are not supported. Common Crypto cannot be used if an application needs to use one of those algorithms. In this case, applications have to use implementations provided by other libraries not being part of iOS.

3.3.3. Keychain Services

3.3.3.1. Introduction

The Keychain is Apple's implementation of a secure storage for sensitive information such as login credentials, passwords, cryptographic keys, and digital certificates. Keychain Services is the public API to interact with the Keychain in order to create, find, update, and delete items in the Keychain. Together with Certificate, Key, and Trust Services (explained in the next chapter), it is the only interface to access the Keychain on iOS.

Keychain Items

A Keychain item is a record which comprises related non-sensitive information such as, e.g., a class, a username, a URL, or a service together with sensitive information, e.g., a password. Within an application a Keychain item is a dictionary (`NSDictionary` or `CFDictionary`) that stores values for certain keys defined in [KCS_REF]. The term "attribute" refers to the combination of a key and its corresponding value that describe the properties of the Keychain item. The sensitive information (the value of the key `kSecValueData`) is no attribute and is encrypted when stored to disk and decrypted when an application accesses the item. Attributes are stored unencrypted. The only mandatory entry of a Keychain item is the entry with the key `kSecClass`. All the other entries are optional and may be left empty. Figure 3-6 shows the structure of an item. Listing 3-19 explains how a new item is constructed in code. All available attributes are explained in [KCS_REF].

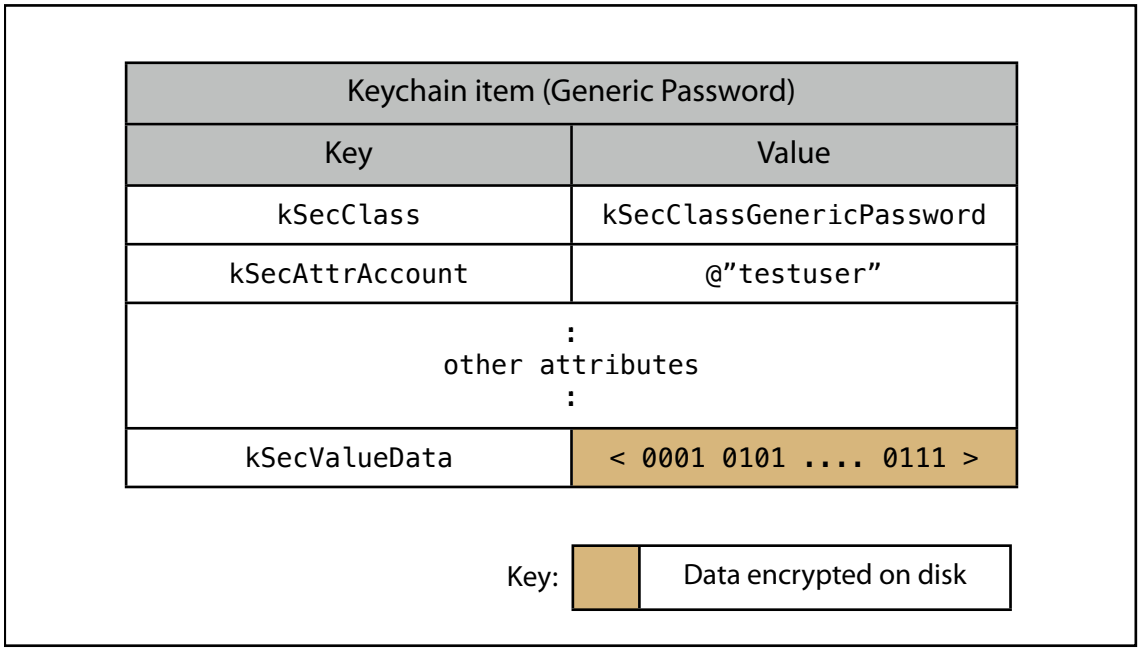

Figure 3-6: Structure of an example Keychain item

```
// Application-specific values
NSString *username = @"testuser";
NSString *password = @"let_me_in";

// Store item's properties in a dictionary
NSDictionary *item = [NSDictionary dictionaryWithObjectsAndKeys:
  kSecClassGenericPassword, kSecClass,
  username, kSecAttrAccount,
  [password dataUsingEncoding:NSUTF8StringEncoding], kSecValueData,
  nil];

// For the developer, a Keychain item is just a dictionary which
// contains the item's information as its entries.
```

Listing 3-19: Construction of a Keychain item

The class of a Keychain item is specified by the value of the key `kSecClass`. Each Keychain item belongs to exactly one of five classes:

- *Generic Password* (`kSecClassGenericPassword`) stores a password together with attributes like the corresponding account (username) and a service name. This item is designed for storing generic account information.
- *Internet Password* (`kSecClassInternetPassword`) stores a password designed to be used as a login credential for a website or a service. Compared to an item of class *Generic Password* it contains additional attributes that describe where this item is applicable: the server's IP or domain name, protocol, authentication type, port, and path.
- *Certificate* (`kSecClassCertificate`) stores a digital certificate.
- *Key* (`kSecClassKey`) stores a private key corresponding to a digital certificate.

- *Identity* (`kSecClassIdentity`) is the combination of a private key and a digital certificate.

The item classes *Certificate*, *Key*, and *Identity* are explained in more detail in the chapter "Certificate, Key, and Trust Services: Certificates, Asymmetric Encryption, Digital Signatures, and Trust Policies".

3.3.3.2. Architecture

The components involved in providing the functionality of the Keychain to third-party applications are illustrated in Figure 3-7. A third-party application uses Keychain Services to access its items in the Keychain. Keychain Services communicate with the Security Server which uses Common Crypto to encrypt and decrypt the sensitive information in the items. It accesses the Keychain database, a SQLite database file on the filesystem, directly (see [S_O] "Security Architecture"). SQLite is a database management system designed to be embedded in applications. Third-party applications cannot access the database file directly, because it is stored in a directory not accessible by them.

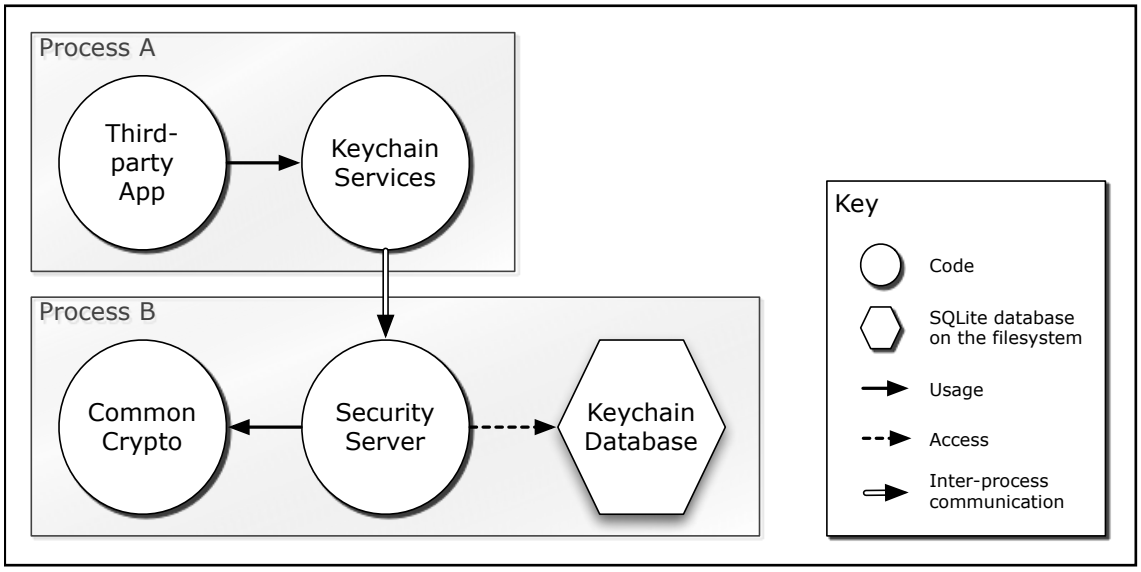

Figure 3-7: Keychain Architecture

The Security Server is a separate process that handles security-related tasks such as encryption and decryption. Cryptographic keys and other secrets can remain in the address space of the Security Server without making them available in the address space of a third-party application (iOS uses virtual memory to separate the address spaces of processes from each other). This enhances the security of the keys. The sensitive information of Keychain items is "encrypted using hardware keys" [ELCOM_PPB_FAQ] before being stored in the database. Applications do not have access to those keys ([KCS_PROG_G] "iPhone Keychain Backups"). Because every

device has its own unique set of hardware keys the encrypted data on the Keychain is tightly coupled to the device on which it was encrypted. This is important for the security of Keychain data in device backups with iTunes (discussed later in this chapter).

3.3.3.3. Access to Keychain Items

iOS maintains one Keychain to store all items from all applications. However, because "Keychain rights depend on the provisioning profile used to sign your application [...] an application can always access its own keychain items, but not items created by any other application" [KCS_PROG_G]. Figure 3-8 illustrates applications accessing the Keychain. All applications are code-signed with a Provisioning Profile issued by Apple. The Keychain uses this profile to decide which items the application can access.

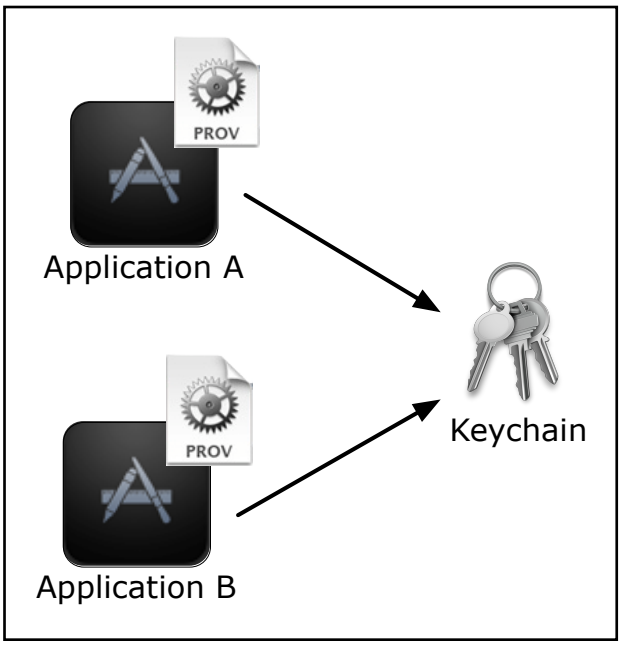

Figure 3-8: Keychain rights depend on their Provisioning Profile

Access Groups

By default, an application only belongs to its own access group and is therefore only able to access its own Keychain items. However, an application can belong to other access groups. A Keychain item that is stored in a shared access group is accessible by all applications which belong to that access group. In this way applications can intentionally share Keychain items with other applications.

The Code Signing Entitlements, a property list file in the application's resources, configures the access groups an application belongs to. In the Target settings, the file name of that file has to be set (see Figure 3-9). A property list file is either an XML or

binary file that stores serialized objects of type `NSString`, `NSNumber`, `NSDate`, `NSData`, `NSArray`, `NSDictionary`, and their Core Foundation counter-parts `CFString`, `CFNumber` & `CFBoolean`, `CFDate`, `CFData`, `CFArray`, and `CFDictionary`. Property list files are used to store user preferences and meta information about an iOS application.

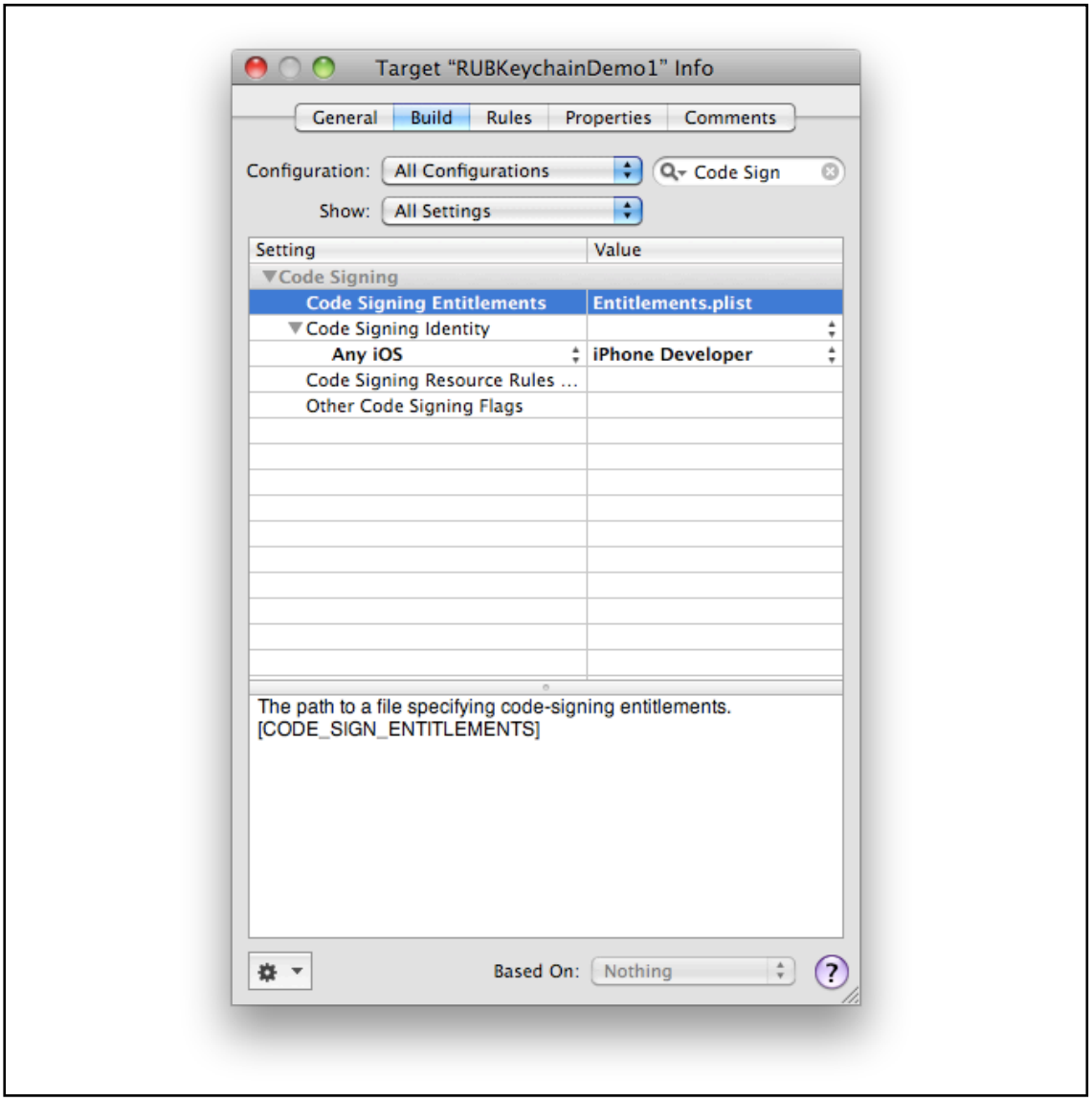

Figure 3-9: Configure the code signing entitlements file

The Code Signing Entitlements, in Figure 3-10 the file is named "Entitlements.plist", contain the key *keychain-access-groups* whose value is an array of strings, the access groups.

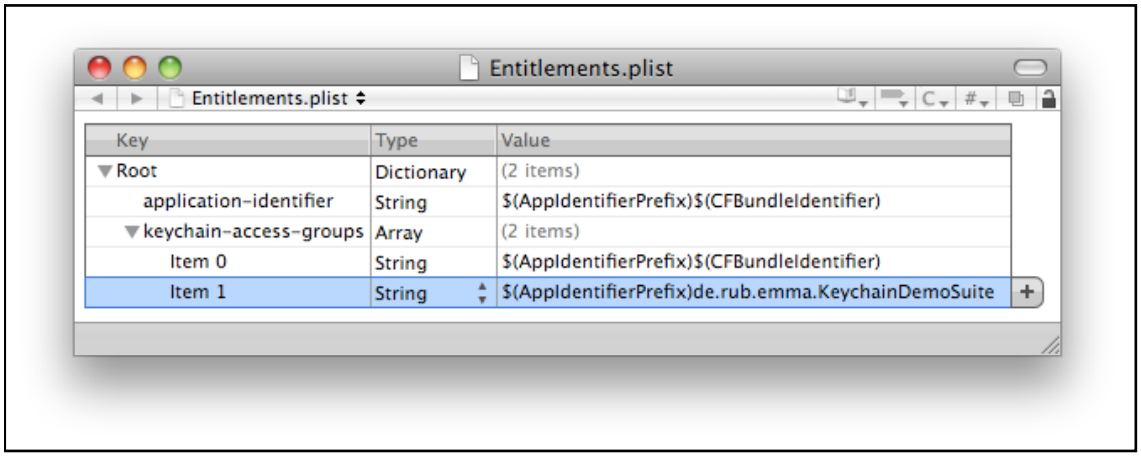

Figure 3-10: Configure the Keychain access groups in Entitlements.plist

The basic structure is generated automatically by Xcode when creating a new file with File → New File... → iOS → Code Signing → Entitlements [QA1710]. The first access group in Figure 3-10 (at index 0) is the application's own access group. Only the application itself can access items in that access group. When the application is built by Xcode the environment variable *AppIdentifierPrefix* is replaced by the Bundle Seed ID and *CFBundleIdentifier* is replaced by the Bundle Identifier.

Every application must be signed with a valid Provisioning Profile. A Provisioning Profile is tightly coupled to an App ID that is configured in the iOS Provisioning Portal. The App ID consists of the Bundle Seed ID and the Bundle Identifier. When a developer configures a new App ID in the iOS Provisioning Portal, two options are available to configure the Bundle Seed ID for the new App ID: the first option generates a random new Bundle Seed ID. The second option reuses an existing Bundle Seed ID from that same iOS Developer Program Account. The second option allows the developer to create application suites that can share Keychain items. The creation of Provisioning Profiles is limited to App IDs of the same iOS Developer Program Account. It is therefore not possible to create a valid Provisioning Profile with a Bundle Seed ID that does not belong to his own account. Other developers cannot gain access to the own Keychain items, even when they are stored in a shared access group, because they do not possess the appropriate Provisioning Profile.

The resulting access group at item index 0 in Figure 3-10 is the string `A2935A6MY3.de.rub.emma.KeychainDemo1`, where `A2935A6MY3` is the Bundle Seed ID and `de.rub.emma.KeychainDemo1` is the Bundle Identifier. The Bundle Seed ID is contained in the Provisioning Profile which is used during code signing the application. The Bundle Identifier is configured in the project's Info.plist. The Info.plist file is a property list file that configures the application's meta data such as e.g. the filename of the application icon, supported device capabilities, Bundle Identifier, and supported devices.

The item at index 1 is a shared access group. It uses the same Bundle Seed ID as the first access group but a different Bundle Identifier. In this case, the access group is `A2935A6MY3.de.rub.emma.KeychainDemoSuite`. Items added to this access group can be accessed by other applications if they also contain that access group name in their Code Signing Entitlements.

The access group which contains non-shared items is not accessible by other applications. Shared access groups are only accessible by applications that are members of that access group. Even if an evil third-party developer knows the name of an access group he wants to gain access to, he cannot access the items in it.

3.3.3.4. Create Keychain Items

An application creates a Keychain item with `SecItemAdd()`. The item's properties are passed as a parameter of type `CFDictionaryRef`. Because of the *toll-free bridging* support, an object of type `NSDictionary` can also be passed as a parameter (see Listing 3-20). Toll-free bridging allows the interchangeable usage of Core Foundation types and Foundation objects where an object of the other type is expected. For example, a Foundation object of type NSDictionary can be used where a Core Foundation object of type CFDictionary is expected and vice versa. A `CFDictionaryRef` is a pointer to a `CFDictionary` Core Foundation object. Foundation objects are cast to their corresponding Core Foundation type in the examples of this thesis to satisfy the compiler. Otherwise the compiler outputs warnings that indicate that the provided parameter types do not match the expected types.

```
// Application-specific values
NSString *itemID = @"de.rub.emma.KeychainDemo.item";
NSString *username = @"testuser";
NSString *password = @"let_me_in";
NSString *accessGroup = @"A2935A6MY3.de.rub.emma.KeychainDemoSuite";

// Store item's properties in a dictionary
NSMutableDictionary *attrs = [NSMutableDictionary dictionary];
[attrs setObject:kSecClassGenericPassword
          forKey:kSecClass];
[attrs setObject:accessGroup
          forKey:kSecAttrAccessGroup];
[attrs setObject:[itemID dataUsingEncoding:NSUTF8StringEncoding]
          forKey:kSecAttrGeneric];
[attrs setObject:[itemID dataUsingEncoding:NSUTF8StringEncoding]
          forKey:kSecAttrService];
[attrs setObject:username
          forKey:kSecAttrAccount];
[attrs setObject:[password dataUsingEncoding:NSUTF8StringEncoding]
          forKey:kSecValueData];

// Add item to Keychain
OSStatus status = SecItemAdd((CFDictionaryRef)attrs,
                             NULL);
if (status == noErr) {
  // Item was added to the Keychain
}
else if (status == errSecDuplicateItem) {
  // Item is already in the Keychain...
  // ... use SecItemUpdate() to update its contents
}
else {
  // Item could not be added to the Keychain
}
```

Listing 3-20: Add an item to the Keychain

The code in Listing 3-20 adds an item of the class *Generic Password* (`kSecClassGenericPassword`) to the shared access group (`kSecAttrAccessGroup`) `A2935A6MY3.de.rub.emma.KeychainDemoSuite`. This allows other applications to access the item if they are also a member of that access group. The string `itemID` is used as an identifier to find the item later (see section "Find Keychain Items"). An item can only be added to exactly one access group, either explicitly by specifying the access group in the attributes dictionary or implicitly by obtaining the order of access groups listed in the Code Signing Entitlements and adding the item to the first access group.

The entry with the key `kSecValueData` holds the raw data of the unencrypted password. However, this value gets encrypted before stored in the Keychain's database on the filesystem [KCS_REF].

Trying to add an item to an access group not listed in the application's Code Signing Entitlements fails and Keychain Services return the error code -25243. Unfortunately,

no description of the code is found in the Keychain Services Reference or Certificate, Key, and Trust Services Reference of the iOS Developer Library. But the Certificate, Key, and Trust Services Reference of the Mac OS X Developer Library contains the constant `errSecNoAccessForItem` for that error code number together with the description "The specified item has no access control".

3.3.3.5. Find Keychain Items

The function used to find Keychain items with Keychain Services is `SecItemCopyMatching()`. It expects a parameter of type `CFDictionaryRef` (the so-called search dictionary) containing the search criteria used to match existing Keychain items against it. Keychain Services search dictionaries are explained in detail in [KCS_REF]. `SecItemAdd()` and `SecItemCopyMatching()` allow the developer to provide a return type in the search dictionary. If a return type is provided, the developer must additionally provide a pointer to a result of type `CFTypeRef` that allows the function to return the desired result indirectly.

`SecItemCopyMatching()` returns all items of the application that match the search criteria. If the application does not specify a specific access group in the search dictionary explicitly, the access groups are searched in order of their appearance in the Code Signing Entitlements. If an access group is defined as s search criterion via the key `kSecAttrAccessGroup` only that access group is searched. By defining the value `kSecMatchLimitOne` for the key `kSecMatchLimit` Keychain Services return only the first item if more than one item matches the search criteria. Listing 3-21 shows how to use Keychain Services to find an item in the Keychain. The example creates a dictionary that contains the search criteria and defines a pointer to the result dictionary. It invokes `SecItemCopyMatching()` with the pointer to the query dictionary and the pointer to the pointer to the result dictionary.

```
// Application-specific values
NSString *itemID = @"de.rub.emma.KeychainDemo.item";

// Construct a query to find the item
NSDictionary *query =
  [NSDictionary dictionaryWithObjectsAndKeys:
    kSecClassGenericPassword, kSecClass,
    [itemID dataUsingEncoding:NSUTF8StringEncoding],
      kSecAttrGeneric,
    [itemID dataUsingEncoding:NSUTF8StringEncoding],
      kSecAttrService,
    kSecMatchLimitOne, kSecMatchLimit,
    (id)kCFBooleanTrue, kSecReturnAttributes,
    nil];

// Add item to Keychain
NSDictionary *result = nil;
OSStatus status = SecItemCopyMatching((CFDictionaryRef)query,
                                      (CFTypeRef *)&result);
if (status == noErr) {
  // Found item(s) in the Keychain
} else if (status == errSecItemNotFound) {
  // No item found
} else {
  // Other status
}
```

Listing 3-21: Find an item in the Keychain

An application can only find items of access groups of which it is a member of. `SecItemCopyMatching()` does not return items of other access groups, even if the access group is explicitly provided in the search dictionary.

3.3.3.6. Update Keychain Items

`SecItemUpdate()` updates items in the Keychain which match the provided search dictionary. The first parameter is the search dictionary and the second parameter is a dictionary which contains the values to be updated on the matching items (see Listing 3-22).

```
// Application-specific values
NSString *itemID = @"de.rub.emma.KeychainDemo.item";

// Construct a query to find the item
NSDictionary *query =
  [NSDictionary dictionaryWithObjectsAndKeys:
    kSecClassGenericPassword, kSecClass,
    [itemID dataUsingEncoding:NSUTF8StringEncoding],
      kSecAttrGeneric,
    [itemID dataUsingEncoding:NSUTF8StringEncoding],
      kSecAttrService,
    nil];

// Assemble new attribute values
NSString *updatedPassword = @"31337";
NSDictionary *updatedAttributes =
  [NSDictionary dictionaryWithObjectsAndKeys:
    [updatedPassword dataUsingEncoding:NSUTF8StringEncoding],
      kSecValueData,
    nil];

// Add item to Keychain
OSStatus status = SecItemUpdate((CFDictionaryRef)query,
                                (CFDictionaryRef)updatedAttributes);
if (status == noErr) {
  // Matching Keychain item(s) updated successfully.
}
else {
  // Keychain item(s) not updated.
}
```

Listing 3-22: Update a Keychain item

3.3.3.7. Delete Keychain Items

`SecItemDelete()` deletes items of the application that match the provided search dictionary (see Listing 3-23).

```
// Application-specific values
NSString *itemID = @"de.rub.emma.KeychainDemo.item";

// Construct a query to find the item(s)
NSDictionary *query =
  [NSDictionary dictionaryWithObjectsAndKeys:
    kSecClassGenericPassword, kSecClass,
    [itemID dataUsingEncoding:NSUTF8StringEncoding],
      kSecAttrGeneric,
    [itemID dataUsingEncoding:NSUTF8StringEncoding],
      kSecAttrService,
    nil];

// Delete item(s) on the Keychain
OSStatus status = SecItemDelete((CFDictionaryRef)query);
if (status == noErr) {
  // Keychain item(s) successfully deleted.
}
else {
  // Keychain item(s) not deleted.
}
```

Listing 3-23: Delete a Keychain item

3.3.3.8. Persistent Keychain References

`kSecReturnPersistentRef` is the return type key that indicates that `SecItemAdd()` or `SecItemCopyMatching()` should return a persistent Keychain reference. A persistent Keychain reference is a `CFDataRef` that uniquely identifies an item in the Keychain. Because items remain in the Keychain when an application is uninstalled, the persistent Keychain reference stays valid even when the application is no longer installed. An item is only removed from the Keychain if the application deletes it with `SecItemDelete()` or the device, and as a result the Keychain, is reset. An application can find an item by providing just the persistent Keychain reference in the search dictionary.

A persistent Keychain reference can be shared with other applications on the device that belong to the same access group in which the item is stored. For example, application *A* is a member of the access group *G*. It stores a Keychain item in *G* and obtains a persistent Keychain reference *R* to it. It then sends *R* to application *B* which is also a member of access group *G*. Application *B* is now able to access the shared Keychain item by using *R* in the search dictionary for the call to `SecItemCopyMatching()`.

3.3.3.9. Keychain Data in iTunes Backups

Every time iTunes synchronizes a device it creates a backup that is stored on the computer's hard disk. According to [ITUNES_BKP], iTunes backups include (among

other things): contacts, application settings, calendar events, call history, photos, videos, the Keychain, Mail accounts, network settings, bookmarks, cookies, and SMS messages.

The content of a backup is unencrypted by default. However, as mentioned earlier, the sensitive information of Keychain items is encrypted on the device by the use of hardware keys before the item is stored on disk. The device's hardware keys are not included in the backup. Therefore, the secrets in the Keychain database remain protected in the backup.

iTunes allows to optionally encrypt device backups with a password supplied by the user (see Figure 3-11). When enabled, iTunes encrypts the files stored on the computer during the synchronization. The encryption of backups enables to migrate the Keychain when the backup is restored to another device. Keychain data is decrypted on the device with the hardware keys and stored in the encrypted backup. This step is necessary, as Keychain data normally cannot be decrypted on other devices because they lack the hardware keys originally used to encrypt the data.

Figure 3-11: Enable/disable encrypted backups in iTunes

The problem with this approach is the strength of the user-provided password to encrypt the backups. iTunes does neither enforce a minimum password length nor indicate to the user the quality of the password. In the current version of iTunes, version 10.1.1, it is possible to use a password with only one character. If an attacker

gains access to the encrypted backups, the only defense is a secure backup password that protects the content, especially the Keychain data.

A good way to indicate the strength of a password can be found in the Keychain Access application on Mac OS X. The application shows a form with a colored bar when a new Keychain item is created (see Figure 3-12). If the password is secure the bar grows to the right and changes its color to green. If the password is insecure the bar shrinks to the left and changes its color to red. This approach aids the user who does not know anything about password strength with her decision to use a certain password. The combination of an enforced minimum password length of eight characters together with an indication of the password's strength would therefore improve the security of iTunes backups.

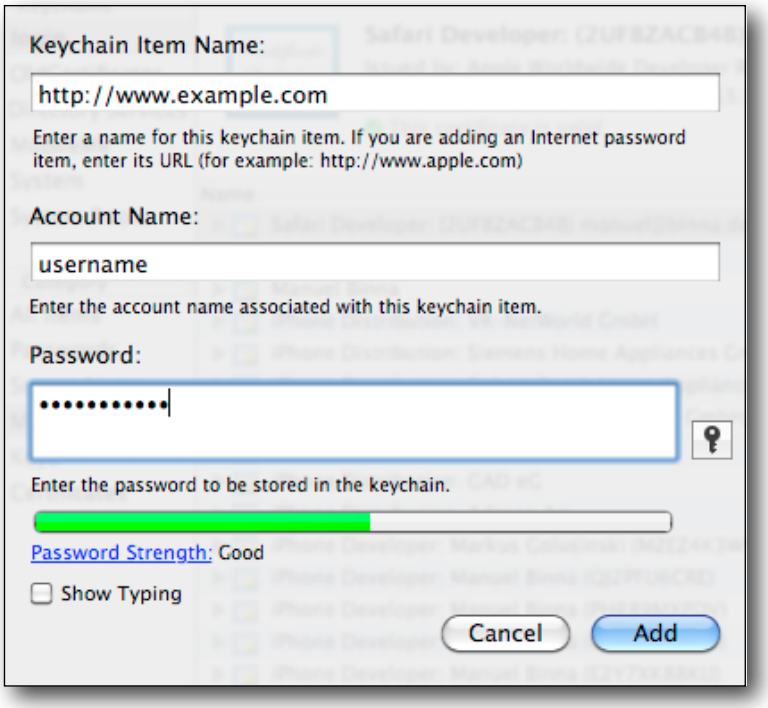

Figure 3-12: Creating a new item with Keychain Access on Mac OS X

3.3.3.10."Lost iPhone? Lost Passwords!"

In a recent attack, Jens Heider and Matthias Boll from Fraunhofer Institute for Secure Information Technology showed that an attacker who has physical control over the device (e.g., the device was found or stolen) is able to obtain the plaintext Keychain data of certain Keychain items [LOST_IPHONE]. The attack first performs a jailbreak of the device and boots the jailbroken device. After that, the attacker has full access to the device. A script then extracts the secret Keychain data from the Keychain's SQLite database in the file system, and uses internal system functions of iOS to decrypt the sensitive data.

This attack works for Keychain items that belong to the protection class `kSecAttrAccessibleAlways` or `kSecAttrAccessibleAlwaysThisDeviceOnly`. Because the user's passcode was not involved in the encryption, items with these classes are transparently decrypted by the hardware device key when the appropriate system function is called. This behavior originates from iOS versions prior to iOS 4, where Data Protection is not available and all data on the device is always encrypted and decrypted transparently by the hardware device key. Even when the device is locked, some files and Keychain items must be available to provide certain functionality, e.g., Exchange account credentials for pushing emails, contacts, and calendar events to the device.

Data Protection, a new mechanism introduced in iOS 4, changes this behavior. Developers can assign a Keychain item a protection class that protects the item with the user's passcode. The attack from Heider and Boll is not able to obtain the plaintext data of those items, because it assumes that the user's passcode is unknown to the attacker and cannot be obtained by brute-force. Data Protection is discussed in in detail in a later chapter.

3.3.3.11.Summary

Keychain Services provides a convenient API to create, find, update, and delete items in the Keychain. The sensitive data from Keychain items is automatically encrypted when the item is stored and decrypted when it is retrieved from the Keychain's SQLite database file in the filesystem. Applications from the same iOS developer can share their Keychain items through the use of access groups.

Secret Keychain data remains encrypted in iTunes backups. However, this protection is weakened by insecure backup passwords.

Keychain Services is missing an Objective-C interface. Many developers, especially those new to the platform, struggle with the lower-level C API and the need to use Core Foundation types or toll-free bridging. This may lead to slow adoption rate and thus potentially more insecure applications. An official Objective-C API for accessing Keychain items could mitigate this shortcoming.

3.3.4. Certificate, Key, and Trust Services

3.3.4.1. Introduction

Certificate, Key, and Trust Services provide the functionality to work with certificates and asymmetric keys on iOS. The API provides functions to import certificates and their private keys from PKCS #12 containers. A certificate together with its corresponding private key is called an identity. Functions to evaluate the trust of certificates, generate new asymmetric key pairs, and encrypt, decrypt, sign and verify data with asymmetric ciphers are also available. This chapter shows how to leverage the API and provides background information about the underlying cryptographic mechanisms.

Applications can import certificates, private keys, and identities from a PKCS #12 file obtained over one of the following channels:

- The file is packaged in the application's main bundle as one of its resources. The main bundle of an iOS application is the collection of the application binary together with the resources like images, XIB (Interface Builder document) files, and other files that the developer distributes with the application.
- The file is received over a network connection.
- The file is received from another application on the device.
- The file is received via the document sharing mechanism of iTunes.

Certificate, Key, and Trust Services is missing a function that provides the creation of certificate signing requests (CSR). A mechanism to construct a CSR from an asymmetric key pair must be implemented by a third-party developer.

Certificates obtained through a secure network connection (e.g. over SSL/TLS) can be evaluated, stored in the Keychain, and used in an application.

3.3.4.2. Import Certificates and Private Keys from a PKCS #12 File

Certificates and corresponding private keys can be imported from a PKCS #12 file with `SecPKCS12Import()`. It expects the PKCS #12 data as one of its parameters and returns the extracted identities (certificates with corresponding private keys) and their associated trust management object indirectly as an array. A trust management object is of type `SecTrustRef`. It is a certificate chain that can be evaluated. PKCS #12 allows more than one identity to be embedded into one file. Therefore the function returns an array instead of a single identity. Information in the PKCS #12 file is symmetrically encrypted with a password. This password must be provided to `SecPKCS12Import()` as its second parameter, the options dictionary. Listing 3-24 shows how to use `SecPKCS12Import()` to import a certificate and corresponding private key from a PKCS #12 file located in the application's main bundle.

```
// Read the PKCS #12 file from the main bundle
NSString *thePath = [[NSBundle mainBundle]
                     pathForResource:@"ExportedIdentity"
                     ofType:@"p12"];
NSData *PKCS12Data = [NSData dataWithContentsOfFile:thePath];

// The password is needed to decrypt the information in the
// PKCS #12 data
NSDictionary *options = [NSDictionary
                         dictionaryWithObject:@"THE_PASSWORD"
                         forKey:(id)kSecImportExportPassphrase];

NSArray *items = nil;
OSStatus importStatus = SecPKCS12Import((CFDataRef)PKCS12Data,
                                        (CFDictionaryRef)options,
                                        (CFArrayRef *)&items);
if (importStatus != errSecSuccess) {
  // Error
}

// SecPKCS12Import() returns one dictionary for each item (identity
// or certificate) in the PKCS #12 data. Use the first item here.
NSDictionary *identityAndTrust = [items objectAtIndex:0];

SecIdentityRef identity = (SecIdentityRef)
  [identityAndTrust objectForKey:(id)kSecImportItemIdentity];
SecTrustRef trust = (SecTrustRef)
  [identityAndTrust objectForKey:(id)kSecImportItemTrust];

// Obtain the certificate from the identity
SecCertificateRef certificate = NULL;
OSStatus certStatus = SecIdentityCopyCertificate(identity,
                                                 &certificate);
if (certStatus != errSecSuccess) {
  // Error
}

// Obtain the public key from the certificate
// !!! Real-world applications should validate the certificate
// before using the public key !!!
SecKeyRef publicKey = SecTrustCopyPublicKey(trust, &publicKey);
if (publicKey == NULL) {
  // Error
}

// Obtain the private key from the identity
SecKeyRef privateKey = NULL;
OSStatus privateKeyStatus = SecIdentityCopyPrivateKey(identity,
                                                      &privateKey);
if (privateKeyStatus != errSecSuccess) {
  // Error
}

// Do something with the items obtained from the PKCS #12 file.
// ...
```

Listing 3-24: Import from a PKCS #12 file

After obtaining a reference to the identity and the trust management object with `SecPKCS12Import()`, a reference to the certificate, the public key, and the private key can be obtained very easily with the functions `SecIdentityCopyCertificate()`, `SecTrustCopyPublicKey()`, and `SecIdentityCopyPrivateKey()`. It is important to validate the certificate before establishing trust in it because the certificate may not be the intended one. If the certificate is not validated before its public key is used, an adversary could mount a man-in-the-middle attack. Validating a certificate ensures that it actually is from the intended peer, that it is valid, and that it is trusted for the intended usage.

3.3.4.3. Evaluate the Trust of a Certificate

`SecTrustEvaluate()` examines a certificate that is provided within a trust management object (an object of type `SecTrustRef`) if it is trusted for, e.g., validating digital signatures or establishing secure network connections. It validates the certificate chain starting with the leaf certificate and continuing up until it successfully verifies the anchor certificate. A list of available trusted root certificates on iOS is available at [IOS4_ROOT_CERTS]. Only then is the leaf certificate considered to be trustworthy. A `SecTrustRef` object contains a leaf certificate, optionally one or more anchor certificates, and one or more security policies (object of type `SecPolicyRef`) that are used during the evaluation of the leaf certificate (see Figure 3-13).

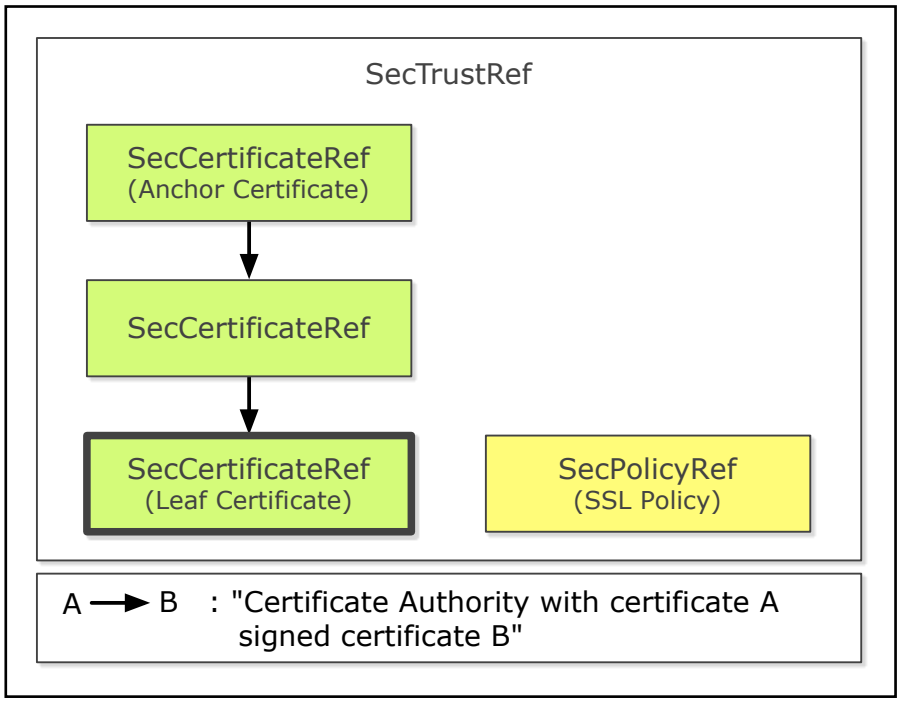

Figure 3-13: A Trust Management Object (`SecTrustRef`)

An anchor certificate is a certificate that the user and/or the operating system trusts. A given certificate is trusted when it was signed directly or indirectly by a trusted anchor certificate. If the trust management object does not contain all certificates needed to verify the leaf certificate, it searches in the Keychain for matching certificates. The optional anchor certificates provided to `SecTrustEvaluate()` can improve the speed of the evaluation when all necessary certificates are contained. Certificates that are not needed during the evaluation are ignored.

Listing 3-25 shows how `SecTrustEvaluate()` is used. It also explains the different evaluation results and their implication on the use of the certificate being evaluated.

```
SecTrustResultType trustResult = kSecTrustResultInvalid;
OSStatus trustStatus = SecTrustEvaluate(trust, &trustResult);
if (trustStatus == errSecSuccess) {
  // Trust evaluation performed successfully. Check the result.
  switch (trustResult) {
    case kSecTrustResultConfirm:
      // The user is required to confirm before proceeding.
      break;
    case kSecTrustResultRecoverableTrustFailure:
      // It is possible to recover from the failure by modifying
      // the evaluation conditions.
      break;
    case kSecTrustResultProceed:
      // The certificate is trusted. It is safe to use it.
      break;
    case kSecTrustResultUnspecified:
      // No specific trust was set by the user. It is safe to use
      // the certificate.
      break;
    case kSecTrustResultDeny:
      // The user does not trust the certificate. Do not use it and
      // fail gracefully.
      break;
    case kSecTrustResultInvalid:
      // A setting or result is invalid. Do not use the certificate
      // and fail gracefully.
      break;
    case kSecTrustResultFatalTrustFailure:
      // The certificate is not trusted. Do not use it and fail
      // gracefully.
      break;
    case kSecTrustResultOtherError:
      // Non-trust-evaluation failure. Do not use the certificate
      // and fail gracefully.
      break;
  }
}
```

Listing 3-25: Evaluating a certificate

Trust Policies

Every `SecTrustRef` contains one or more policies that are used by `SecTrustEvaluate()`. The contained leaf certificate is only considered trustworthy if all certificates in the certificate chain are validated successfully for every policy contained in the trust management object. A policy is represented by an object of type `SecPolicyRef`. Application developers can use two policies: the default X.509 policy (created by `SecPolicyCreateBasicX509()`) and an SSL policy (created by `SecPolicyCreateSSL()`) suitable for establishing secure network connections.

Recoverable Trust Failure

A recoverable trust failure indicates that the evaluation of a certificate was not successful, but may be successful if the parameters of the evaluation are changed. Consider the following situation: an application uses a private key to digitally sign a document and sends the signed document together with the private key's corresponding certificate to another application. However, before the other application verifies the document's digital signature the certificate expires. The certificate was valid when the signature was generated but is now invalid because it expired. In this situation `SecTrustEvaluate()` would return the result `kSecTrustResultRecoverableTrustFailure`.
`SecTrustGetVerifyTime()` and `SecTrustSetVerifyDate()` can be used to query and edit the date and time used during the validation of the certificate. A new verify date set with `SecTrustSetVerifyDate()` is stored in the trust management object. The application could set the verify date to the date the signature was created. The following invocation of `SecTrustEvaluate()` would then indicate that the certificate was valid when the signature was created, given that there are no other issues.

Exceptions to Trust Policies

iOS 4 introduces the functions `SecTrustCopyExceptions()` and `SecTrustSetExceptions()` to explicitly set exceptions in the trust policies used during the evaluation of a certificate. If the user wants to explicitly trust a certificate although `SecTrustEvaluate()` determines that it cannot be trusted, an application can use these two methods to set the current exceptions in the trust policy. `SecTrustEvaluate()` ignores those exceptions in future evaluations and returns `kSecTrustResultProceed` if no new error occurs between the invocations of `SecTrustCopyExceptions()` and `SecTrustEvaluate()`.

3.3.4.4. Store Certificates, Keys, and Identities in the Keychain

The previous chapter "Keychain Services: Secure Storage for Sensitive Information" already described how to create, find, update, and delete items in the Keychain with the functions provided by the Keychain Services API. When working with certificates, keys, or identities, it is often required to store them in the Keychain. This section explains the usage of Keychain Services with a focus on certificates, keys, and identities.

In order to create, find, update, and delete certificates, keys, or identities in the Keychain, the functions `SecItemAdd()`, `SecItemCopyMatching()`, `SecItemUpdate()`, and `SecItemDelete()` are used. Those functions have already been introduced in the last chapter. Listing 3-26 explains how to add an identity to the Keychain.

```
NSDictionary *attributes =
  [NSDictionary dictionaryWithObject:(id)identity
                              forKey:(id)kSecValueRef];
OSStatus itemAddStatus = SecItemAdd((CFDictionaryRef)attributes,
                                    NULL);
if (itemAddStatus != errSecSuccess) {
  // Error
} else {
  // Item successfully added.
}
```

Listing 3-26: Adding an identity to the Keychain

Obtain Information About an Identity

When a certificate is imported from a PKCS #12 file, there is only few information available that can be extracted from the `SecCertificateRef`. `SecCertificateCopySubjectSummary()` returns a string with information about the provided `SecCertificateRef`, but that is the only available information at this point. The item's attributes can only be obtained after the certificate was added to the Keychain. Listing 3-27 shows how `SecItemCopyMatching()` returns the attributes of a certificate stored in the Keychain.

```
// Obtain the attributes and value of a Keychain item with
// SecItemCopyMatching(). The item is returned indirectly.
NSDictionary *query = ...; // Search for an identity.
NSDictionary *keychainItem = nil; // The identity's information.
OSStatus status = SecItemCopyMatching(query,
                                      (CFTypeRef *)&keychainItem);
if (status == errSecSuccess) {
  CFTypeRef accessibility =
    [keychainItem objectForKey:kSecAttrAccessible];
  NSString *accessGroup =
    [keychainItem objectForKey:kSecAttrAccessGroup];
  CFTypeRef keyClass =
    [keychainItem objectForKey:kSecAttrKeyClass];
  NSString *label =
    [keychainItem objectForKey:kSecAttrLabel];
  NSString *applicationLabel =
    [keychainItem objectForKey:kSecAttrApplicationLabel];
  NSNumber *isPermanent =
    [keychainItem objectForKey:kSecAttrIsPermanent];
  NSData *applicationTag =
    [keychainItem objectForKey:kSecAttrApplicationTag];
  NSNumber *keyType =
    [keychainItem objectForKey:kSecAttrKeyType];
  NSNumber* keySizeInBits =
    [keychainItem objectForKey:kSecAttrKeySizeInBits];
  NSNumber *effectiveKeySize =
    [keychainItem objectForKey:kSecAttrEffectiveKeySize];
  NSNumber *canEncrypt =
    [keychainItem objectForKey:kSecAttrCanEncrypt];
  NSNumber *canDecrypt =
    [keychainItem objectForKey:kSecAttrCanDecrypt];
  NSNumber *canDerive =
    [keychainItem objectForKey:kSecAttrCanDerive];
  NSNumber *canSign =
    [keychainItem objectForKey:kSecAttrCanSign];
  NSNumber *canVerify =
    [keychainItem objectForKey:kSecAttrCanVerify];
  NSNumber *canWrap =
    [keychainItem objectForKey:kSecAttrCanWrap];
  NSNumber *canUnwrap =
    [keychainItem objectForKey:kSecAttrCanUnwrap];
}
```

Listing 3-27: Obtaining information from an identity in the Keychain

3.3.4.5. Generate an Asymmetric Key Pair

Certificate, Key, and Trust Services provide the function `SecKeyGeneratePair()` to generate an asymmetric key pair. Asymmetric key pairs are used for public key cryptography, like cryptographic protocols, digital signatures, or to wrap a symmetric key to be sent from one party to another. `SecKeyGeneratePair()` expects two pointers to `SecKeyRefs`, one to return the generated public key indirectly and one to return the generated private key indirectly. Furthermore, it expects a dictionary with options that configure the keys that are going to be generated (see Listing 3-28).

```
#define kPrivateKeyTag @"de.rub.emma.CertKeyTrustDemo.PrivateKey"
#define kPublicKeyTag  @"de.rub.emma.CertKeyTrustDemo.PublicKey"

// Configure the public key
NSMutableDictionary *publicKeyAttributes =
  [[[NSMutableDictionary alloc] init] autorelease];
NSData *publicKeyTagData =
  [kPublicKeyTag dataUsingEncoding:NSUTF8StringEncoding];
[publicKeyAttributes setObject:publicKeyTagData
                        forKey:kSecAttrApplicationTag];
[publicKeyAttributes setObject:[NSNumber numberWithBool:YES]
                        forKey:kSecAttrIsPermanent];

// Configure private key
NSMutableDictionary *privateKeyAttributes =
  [[[NSMutableDictionary alloc] init] autorelease];
NSData *privateKeyTagData =
  [kPrivateKeyTag dataUsingEncoding:NSUTF8StringEncoding];
[privateKeyAttributes setObject:privateKeyTagData
                         forKey:kSecAttrApplicationTag];
[privateKeyAttributes setObject:[NSNumber numberWithBool:YES]
                         forKey:kSecAttrIsPermanent];

// Configure key pair
NSMutableDictionary *keyPairAttributes =
  [[[NSMutableDictionary alloc] init] autorelease];
[keyPairAttributes setObject:kSecAttrKeyTypeRSA
                      forKey:kSecAttrKeyType];
[keyPairAttributes setObject:[NSNumber numberWithInteger:2048]
                      forKey:kSecAttrKeySizeInBits];
[keyPairAttributes setObject:publicKeyAttributes
                      forKey:kSecPublicKeyAttrs];
[keyPairAttributes setObject:privateKeyAttributes
                      forKey:kSecPrivateKeyAttrs];

// Generate key pair
SecKeyRef thePublicKey = NULL;
SecKeyRef thePrivateKey = NULL;
OSStatus generatePairStatus =
  SecKeyGeneratePair((CFDictionaryRef)keyPairAttributes,
                     &thePublicKey,
                     &thePrivateKey);
if (generatePairStatus != errSecSuccess) {
  // Error
} else {
  // Success. Use the generated keys...
}
```

Listing 3-28: Generating an asymmetric key pair

The strings `kPrivateKeyTag` and `kPublicKeyTag` annotate the generated keys with a unique identifier. The tags can be used later in a search dictionary to find the keys with `SecItemCopyMatching()`.

Among others, the following options define the properties of the key pair to be generated in Listing 3-28:

- `kSecAttrKeyType` specifies the cryptographic algorithm for which the key pair is generated. The two supported algorithms are RSA (`kSecAttrKeyTypeRSA`) and elliptic curve (`kSecAttrKeyTypeEC`).
- `kSecAttrKeySizeInBits` specifies the key pair's size in bits.
- `kSecAttrIsPermanent` specifies if the public and/or private key should be automatically added to the Keychain.

iOS supports RSA key sizes up to 4096 bits. Trying to generate an RSA key pair with a key size greater than 4096 bits resulted in the status -9809 which is not documented in the iOS Developer Library. However, the header file SecureTransport.h on Mac OS X reveals the constant `errSSLCrypto` for this error code. The description of this error code is as follows: "underlying cryptographic error". On iOS, Secure Transport is a private API and thus is not accessible by third-party developers.

[KCS_REF] indicates that iOS supports elliptic curve cryptography via the constant `kSecAttrKeyTypeEC`. However, generating a key pair of type `kSecAttrKeyTypeEC` was not possible. When building the application in Xcode the linker could not find the symbol `kSecAttrKeyTypeEC` although being defined in the header file SecItem.h (see Listing 3-29).

```
extern CFTypeRef kSecAttrKeyTypeEC
    __OSX_AVAILABLE_STARTING(__MAC_NA, __IPHONE_4_0);
```

Listing 3-29: Definition of `kSecAttrKeyTypeEC` in SecItem.h

A search for "kSecAttrKeyTypeEC" on Stack Overflow, in the Apple Developer Forums, and on Google yields no results at the time of writing. A discussion on the Apple CDSA mailing list [CDSA_ML_EC_KEY] does not yield useful results, too. An Apple Developer Technical Support (ADTS) incident clarified that elliptic curve cryptography is only used by Secure Transport (a private API not publicly documented that provides the primary SSL/TLS layer on iOS) to establish an SSL/TLS session if the device uses a certificate with an elliptic curve public key. ADTS mentions that `kSecAttrKeyTypeEC` is not meant to be public. ADTS filed an internal bug report asking that this discrepancy be corrected. ADTS also mentions that Secure Transport is not based on OpenSSL or on another implementation, but details about that are not publicly documented. The email conversation with ADTS is contained on the enclosed disc in the folder "References/Apple Developer Technical Support".

The question if there is information available about the elliptic curve(s) being used by iOS was negated. The enhancement request 9195235 ("Provide information about elliptic curves in reference documentation") was filed thereafter via the Apple Bug Reporter.

Figure 3-14 shows the duration to generate different RSA key pairs on an iPhone 4 with iOS version 4.2.1. Ten samples were taken for every key size. The results show that the duration of the generation varies considerably between the different key sizes but also between different generations of the same key size.

Key Size (bits)	1024	2048	3072	4096
Average Duration (s)	2	38	140	467
Maximum Duration (s)	3	76	226	953
Minimum Duration (s)	1	16	60	106

Figure 3-14: Duration of RSA Key Pair Generations

A `SecKeyRef` has many other attributes to work with. A complete list can be found in [KCS_REF] in the section "Attribute Item Keys and Values".

3.3.4.6. Asymmetrically Encrypt and Decrypt Data

iOS applications use asymmetric encryption and decryption through the functions `SecKeyEncrypt()` and `SecKeyDecrypt()` (see Listing 3-30 and 3-31). Both functions expect an input and an output buffer and the buffers' corresponding lengths. If necessary, the input buffer can also be used as the output buffer to save memory. The maximum length of the input buffer is restricted to the modulus of the RSA key. `SecKeyEncrypt()` expects a public key, `SecKeyDecrypt()` expects a private key.

```
SecKeyRef publicKey = ...;    // A given public key

// PKCS #1 padding.
// Can only encrypt SecKeyGetBlockSize() - 11 bytes.
if ([dataToEncrypt length] > (SecKeyGetBlockSize(publicKey) - 11)) {
  // Encryption not possible
  return;
}

// Encrypt data with public key
size_t encryptedDataLength = SecKeyGetBlockSize(publicKey);
NSMutableData *encryptedData = [[[NSMutableData alloc]
                                 initWithLength:encryptedDataLength]
                                 autorelease];
OSStatus encryptStatus = SecKeyEncrypt(publicKey,
                                       kSecPaddingPKCS1,
                                       [dataToEncrypt bytes],
                                       [dataToEncrypt length],
                                       [encryptedData mutableBytes],
                                       &encryptedDataLength);

// Adjust length to actual length of encrypted data
[encryptedData setLength:encryptedDataLength];

// Check for error
if (encryptStatus != errSecSuccess) {
  return;
}

// Do something with encrypted data...
```

Listing 3-30: Encrypting data with a public key

```
SecKeyRef privateKey = ...;  // A given private key

// Decrypt data with private key
size_t decryptedDataLength = SecKeyGetBlockSize(privateKey);
NSMutableData *decryptedData = [[[NSMutableData alloc]
                                 initWithLength:decryptedDataLength]
                                 autorelease];
OSStatus decryptStatus = SecKeyDecrypt(privateKey,
                                       kSecPaddingPKCS1,
                                       [dataToDecrypt bytes],
                                       [dataToDecrypt length],
                                       [decryptedData mutableBytes],
                                       &decryptedDataLength);

// Adjust length to actual length of decrypted data
[decryptedData setLength:decryptedDataLength];

// Check for error
if (decryptStatus != errSecSuccess) {
  return;
}

// Do somthing with the decrypted data...
```

Listing 3-31: Decrypting data with a private key

PKCS #1 Padding and OAEP

`SecKeyEncrypt()` optionally pads the input before the actual encryption and `SecKeyDecrypt()` can remove that padding after the actual decryption. The available paddings are:

- `kSecPaddingNone` - no padding
- `kSecPaddingPKCS1` - PKCS #1 padding
- `kSecPadding0AEP` - Optimal Asymmetric Encryption Padding (OAEP)

PKCS #1 padding and OAEP are defined by [RFC3447]. OAEP is supported since iOS 4.1. However, both Keychain Services Reference [KCS_REF] and Certificate, Key, and Trust Services Reference [CKTS_REF] do not mention the availability of this option. Solely the iOS 4.0 to iOS 4.1 API Differences [41_API_DIFFS] and the header file SecKey.h of the Security framework itself list the constant `kSecPadding0AEP`. The enhancement request 9203545 ("Provide more details about PKCS #1 padding in reference documentation") was filed via the Apple Bug Reporter. The bugs filed in the Apple Bug Reporter are not publicly available. Thus, developers can use the platform Open Radar to publish the requests they submitted to the Apple Bug Reporter. The enhancement request 9203545 is available at [RDAR_1199409]. The request also suggests to consider that MD2, MD5, and SHA1 are less secure than hash functions of the SHA-2 family (e.g. SHA-256 or SHA-512) and to provide information about how to use those with PKCS #1 padding appropriately.

Figure 3-15 shows the encryption and decryption scheme of RSA with PKCS #1 padding defined by [RFC3447]. The operations which perform the actual padding are highlighted in a dark color.

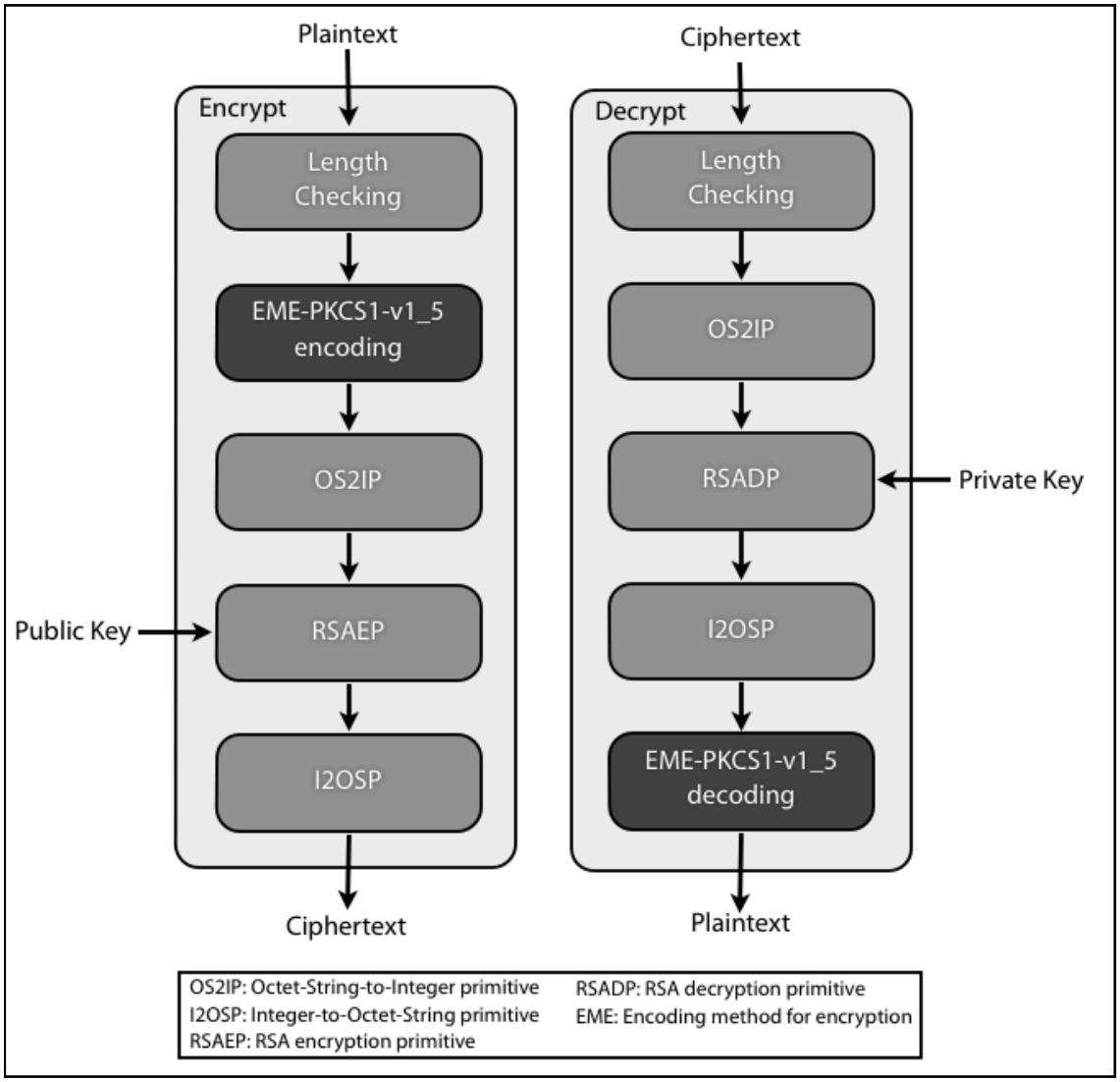

Figure 3-15: RSA Encryption and Decryption with PKCS #1 Padding

Figure 3-16 shows the encryption and decryption scheme of RSA with OAEP defined by [RFC3347].

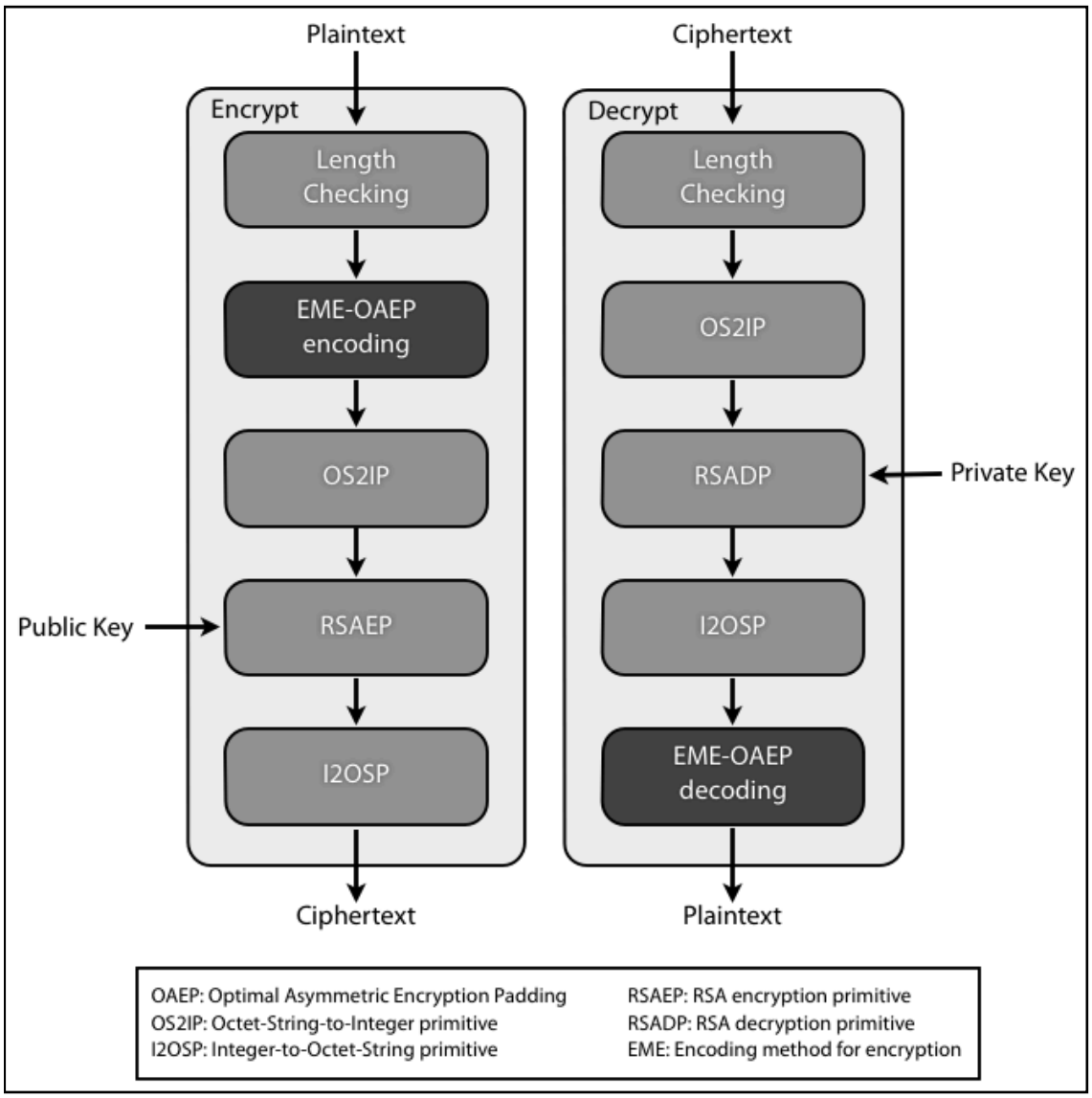

Figure 3-16: RSA Encryption and Decryption with OAEP

3.3.4.7. Sign a Document and Verify the Signature

Digital signatures are created with the function `SecKeyRawSign()`. The function `SecKeyRawVerify()` verifies a given signature. In practice, creating a digital signature consists of calculating a digest of the document to be signed with a secure hash function and signing that digest. Calculating digests from arbitrary data is explained in the chapter "Common Crypto: Symmetric Encryption, Hash Functions, and HMACs" earlier in this thesis.

During the creation of a digital signature, the data to be signed is typically padded. Certificate, Key, and Trust Services support the following paddings:

- `kSecPaddingNone` - No padding is applied.
- `kSecPaddingPKCS1` - PKCS #1 padding (also see [PKCS1_PAD_ML])
- `kSecPaddingPKCS1MD2` - When the data to sign is an MD2 digest.
- `kSecPaddingPKCS1MD5` - When the data to sign is an MD5 digest.
- `kSecPaddingPKCS1SHA1` - When the data to sign is a SHA1 digest.

PKCS #1 v1.5 requires two different paddings when signing a digest. The first padding concatenates a value that is specific to the hash function to the digest `H` [RFC3447]. The result is called `T`:

```
MD2:      T = 0x3020300c06082a864886f70d020205000410   || H
MD5:      T = 0x3020300c06082a864886f70d020505000410   || H
SHA-1:    T = 0x3021300906052b0e03021a05000414         || H
SHA-256:  T = 0x3031300d060960864801650304020105000420 || H
SHA-384:  T = 0x3041300d060960864801650304020205000430 || H
SHA-512:  T = 0x3051300d060960864801650304020305000440 || H
```

The second padding yields the encoded message (`EM`) that is to be signed in a later step.

```
EM = 0x00 || 0x01 || PS || 0x00 || T
```

The padding string (`PS`) consists of exactly that many bytes `0xFF` that the length of `EM` is the length of the RSA modulus.

`SecKeyRawSign()` performs both paddings when the developer specifies either `kSecPaddingPKCS1MD2`, `kSecPaddingPKCS1MD5`, or `kSecPaddingPKCS1SHA1`. However, if `SecKeyRawSign()` is invoked with `kSecPaddingPKCS1`, the function expects that the first padding was already applied to the provided input. To use SHA-512 the developer concatenates the value `0x3051300d060960864801650304020305000440` with the SHA-512 digest `H`. The result together with `kSecPaddingPKCS1` is the input for `SecKeyRawSign()` [SO_PADDING_TYPES].

The enhancement request 9203545 ("Provide more details about PKCS #1 padding in reference documentation") also covers the missing distinction between `kSecPaddingPKCS1` and `kSecPaddingPKCS1SHA1`, `kSecPaddingPKCS1MD2`, and `kSecPaddingPKCS1MD5`.

Figure 3-17 shows the signature generation and verification scheme with appendix and PKCS #1 v1.5 padding defined by [RFC3447]. The two dark blocks indicate operations that perform padding.

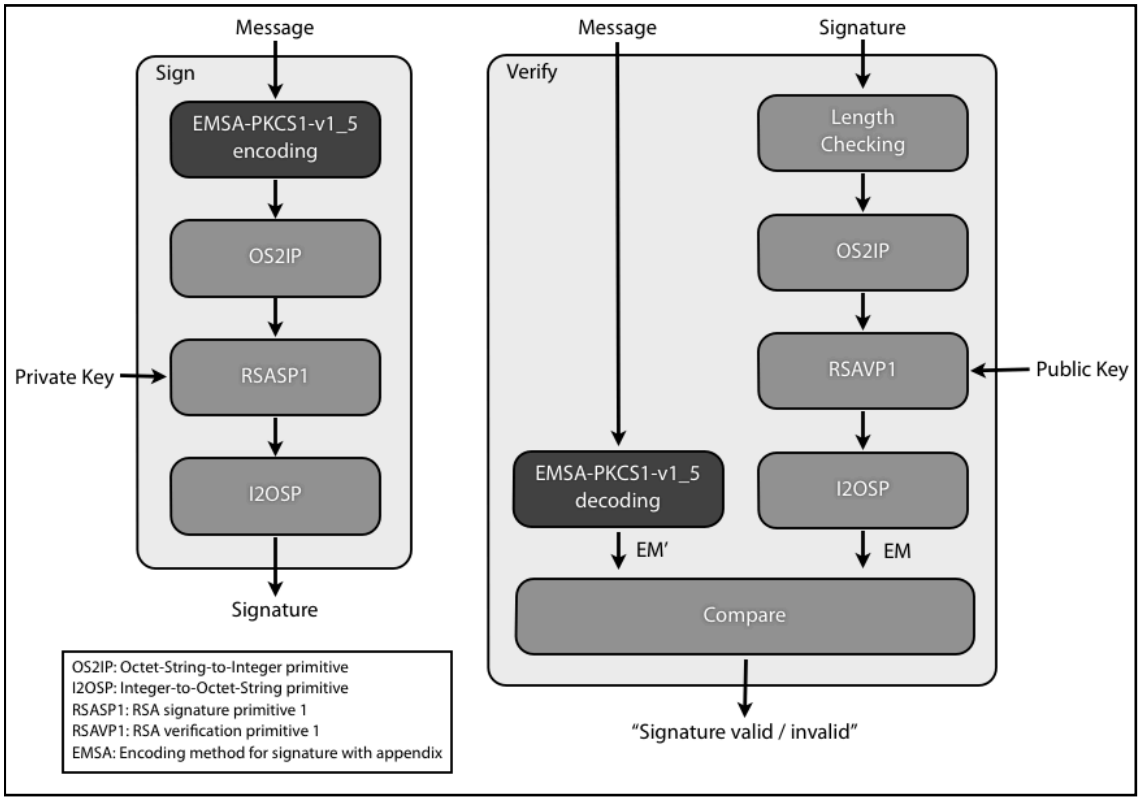

Figure 3-17: RSA Signature and Verification with PKCS #1 Padding

Listing 3-32 explains how to use `SecKeyRawSign()` to create a digital signature using the hash function SHA1.

```
NSData *dataToSign = ...;    // Actual document to sign
SecKeyRef privateKey = ...;  // Private key of signer
NSData *digest = ...;        // SHA1 digest of 'dataToSign'

// Sign digest
size_t signatureLength = SecKeyGetBlockSize(aPrivateKey);
NSMutableData *signature = [[NSMutableData alloc]
                            initWithLength:signatureLength];
OSStatus signStatus = SecKeyRawSign(aPrivateKey,
                                    kSecPaddingPKCS1SHA1,
                                    [digest bytes],
                                    [digest length],
                                    [signature mutableBytes],
                                    &signatureLength);

// Just in case SecKeyRawSign() modifies that value internally
[signature setLength:signatureLength];

if (signStatus != errSecSuccess) {
  // Error
} else {
  // Signature was created
}
```

Listing 3-32: Signing a Document

Listing 3-33 shows the verification of the digital signature of a given document and signature.

```
NSData *dataToVerify = ...;    // Received document
NSData *signature = ...;       // Signature to verify
SecKeyRef publicKey = ...;     // Public key of signer

// Calculate SHA1 digest from received data
NSData *digestToVerify = ...;  // SHA1 digest of 'dataToVerify'

// Verify digest
OSStatus verifyStatus = SecKeyRawVerify(publicKey,
                                        kSecPaddingPKCS1SHA1,
                                        [digestToVerify bytes],
                                        [digestToVerify length],
                                        [signature bytes],
                                        [signature length]);

if (verifyStatus != errSecSuccess) {
  // Signature is invalid
} else {
  // Signature is valid
}
```

Listing 3-33: Verifying the Signature

3.3.4.8. Summary

Certificate, Key, and Trust Services is an API to generate asymmetric key pairs, perform asymmetric encryption and decryption, create and verify digital signatures, import keys and certificates from PKCS #12 files, obtain information from certificates, and evaluate certificate chains with policies (combined into a trust management object). However, the API lacks in the functionality to create certificate signing requests from generated asymmetric key pairs.

The constant `kSecAttrKeyTypeEC` is defined in a public header albeit iOS only supports elliptic curves internally through the private API *Secure Transport*. Only after filing an Apple Developer Technical Support incident and sending several e-mails did Apple provide sufficient information about the current state of elliptic curve support on iOS and clarified that *Secure Transport* is not based on OpenSSL or on another implementation. This emphasizes the requirement that Apple has to provide more detailed documentation about security features and technologies used in iOS.

At the moment, the constants related to PKCS #1 padding are documented sparsely. Optimal Asymmetric Encryption Padding (OAEP) via `kSecPadding0AEP` is not documented at all although being a public API since iOS 4.1 [41_API_DIFFS]. Background information about the padding schemes PKCS #1 v1.5 and OAEP is missing as well as documentation about the difference between `kSecPaddingPKCS1`

and `kSecPaddingPKCS1SHA1`, `kSecPaddingPKCS1MD2`, or `kSecPaddingPKCS1MD5`. The documentation should also consider that MD2, MD5, and SHA1 are less secure than hash functions of the SHA-2 family (e.g. SHA-256 or SHA-512) and provide information about how to use those with PKCS #1 padding appropriately.

3.3.5. URL Loading System, `UIWebView` and CFNetwork

3.3.5.1. Introduction

iOS contains different APIs to implement networking functionality in an application: the class `UIWebView`, the URL Loading System, CFNetwork, and BSD sockets. The APIs differ from each other both in the level of abstraction and control they provide (see Figure 3-18 [CFN_PROG_G]).

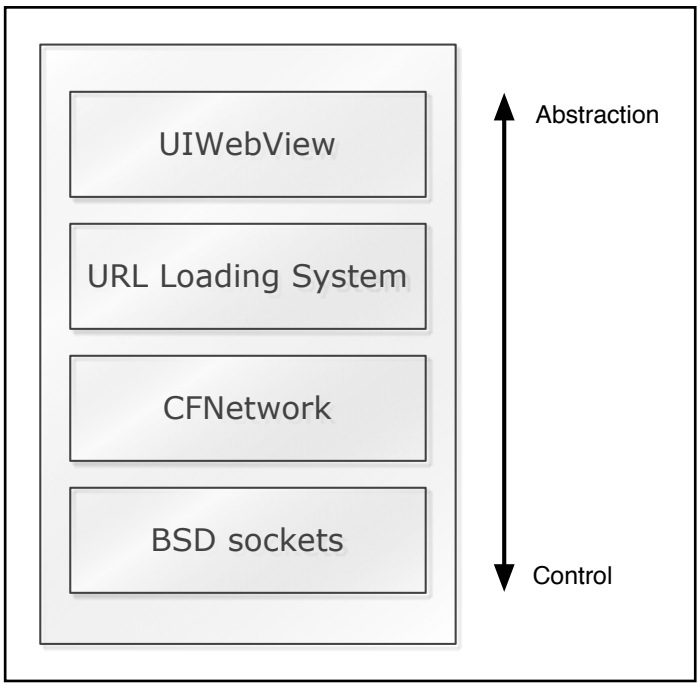

Figure 3-18: Networking APIs in iOS

The class `UIWebView` implements a view that is capable of loading and displaying HTML content. It is often used by developers to implement Web browser functionality within an application so that the user does not have to switch to Safari to view a Web page. The developer can register a class as the delegate of the `UIWebView` and receive callbacks while the view loads the requested resource. `UIWebView` loads resources from the local filesystem or from remote servers. The developer creates and configures an `NSURLRequest` and tells the `UIWebView` to load that request.
A class that conforms to the protocol `UIWebViewDelegate` can receive messages declared by the `UIWebViewDelegate` protocol in the header file UIWebView.h of

`UIWebView` (see Listing 3-34). Besides the initial configuration of the request to load the resource, the developer has only very limited control over the actual loading process.

```
@protocol UIWebViewDelegate <NSObject>

@optional
- (BOOL)webView:(UIWebView *)webView
        shouldStartLoadWithRequest:(NSURLRequest *)request
        navigationType:(UIWebViewNavigationType)navigationType;
- (void)webViewDidStartLoad:(UIWebView *)webView;
- (void)webViewDidFinishLoad:(UIWebView *)webView;
- (void)webView:(UIWebView *)webView
        didFailLoadWithError:(NSError *)error;

@end
```

Listing 3-34: `UIWebViewDelegate` Protocol

The URL Loading System [URLLS_P_G] provides an Objective-C API to perform asynchronous network communication on iOS. The class `NSURLRequest` encapsulates the request for a specific resource (addressed by an instance of `NSURL`) that is sent via an instance of `NSURLConnection`. The `NSURLConnection` object asynchronously sends its delegate methods corresponding to network events, e.g., when the connection received a response, when the connection did fail, or when the connection did finish loading (see Figure 3-19). The delegate is implemented by the developer who uses the URL Loading System.

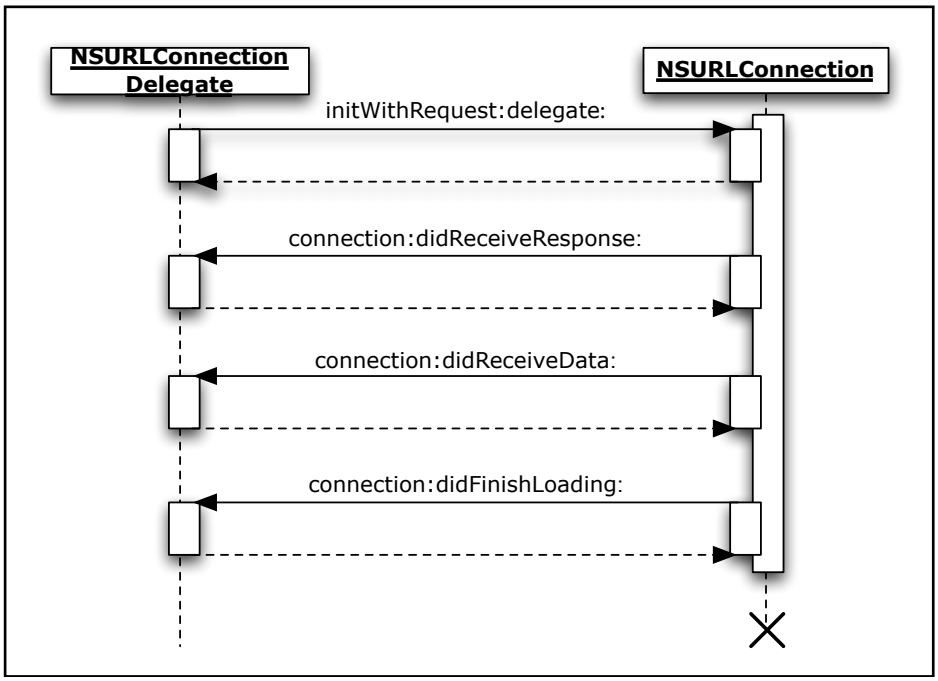

Figure 3-19: Functionality of URL Loading System

CFNetwork is a C-based API that is architecturally located between the low-level BSD sockets and the high-level URL Loading System. It provides the developer much more control over the protocol than the URL Loading System, but it is also more complex to use. CFNetwork is used by the all the higher-level frameworks on iOS.

Of the layers shown in Figure 3-18, the BSD socket layer provides the most control, flexibility, and performance. All network frameworks in iOS are based on this "de-facto standard for UNIX network programming" [IOS_NET_INET]. A BSD socket is an "abstraction of a communication endpoint" [APUE]. In UNIX operating systems, applications use file descriptors to access sockets. Listing 3-35 gives on overview of the different functions of the BSD socket API. This thesis focusses only on the APIs specific to iOS, hence it does not discuss BSD sockets further.

```
// Create a socket (returns a file descriptor)
int socket(int domain, int type, int protocol);

// Assign an address to the socket
int bind(int socket, const struct sockaddr *address,
         socklen_t address_len);

// Get the address bound to the socket
int getsockname(int socket, struct sockaddr *restrict address,
                socklen_t *restrict address_len);

// Get the peer's address
int getpeername(int socket, struct sockaddr *restrict address,
                socklen_t *restrict address_len);

// Set an option on the socket
int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);

// Get an option on the socket
int getsockopt(int socket, int level, int option_name,
               void *restrict option_value,
               socklen_t *restrict option_len);

// Create a connection between client and server
int connect(int socket, const struct sockaddr *address,
            socklen_t address_len);

// Begin listening for incoming connections (only server)
int listen(int socket, int backlog);

// Retrieve a connect request and convert it into a connection
int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);

// Send data
ssize_t send(int socket, const void *buffer, size_t length,
             int flags);
ssize_t sendto(int socket, const void *buffer, size_t length,
               int flags, const struct sockaddr *dest_addr,
               socklen_t dest_len);
ssize_t sendmsg(int socket, const struct msghdr *message,
                int flags);

// Receive data
ssize_t recv(int socket, void *buffer, size_t length, int flags);
ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
                 int flags, struct sockaddr *restrict address,
                 socklen_t *restrict address_len);
ssize_t recvmsg(int socket, struct msghdr *message, int flags);

// Disable I/O on the socket
int shutdown(int socket, int how);

// Close the file (or socket) specified by the file descriptor
int close(int fildes);
```

Listing 3-35: Overview of the BSD Socket API

3.3.5.2. Secure Network Connections with the URL Loading System

A requested resource may need authentication before being delivered to the device. If the delegate wants to gain full control over the authentication procedure, it can implement the method `connectionShouldUseCredentialStorage:` to state that the `NSURLConnection` instance should not use the `NSURLCredentialStorage`. The `NSURLCredentialStorage` is a singleton object that provides access to the `NSURLCredentials` stored in the system that are available to all applications.

`NSURLCredential` encapsulates a credential used for authentication. It contains the authentication information like, e.g., username and password, a client certificate, or a server certificate. The `NSURLCredential` also contains a persistence property that states how long the credential will be stored

- `NSURLCredentialPersistenceNone` - The credential will not be stored.
- `NSURLCredentialPersistenceForSession` - The credential will be stored for the current session.
- `NSURLCredentialPersistencePermanent` - The credential will be stored in the Keychain. Other applications can access the credential. It will be accessible by other applications through the shared `NSURLCredentialStorage`.

An `NSURLProtectionSpace` "represents a server or an area on a server that requires authentication" before delivering the requested resource [NSURLPS_C_REF]. It contains properties like, e.g., the authentication method, host, port, protocol, or the server trust as a SecTrustRef object. When stored in the `NSURLCredentialStorage`, an `NSURLCredential` is associated with an `NSURLProtectionSpace`. The credential applies to all requests within that protection space. The different methods of authentication within an `NSURLProtectionSpace` are:

- `NSURLAuthenticationMethodDefault`: Default authentication for the given protocol. The default for HTTP is `NSURLAuthenticationMethodBasic`.
- `NSURLAuthenticationMethodBasic`: HTTP Basic Access Authentication [RFC2617]
- `NSURLAuthenticationMethodDigest`: HTTP Digest Access Authentication [RFC2617]
- `NSURLAuthenticationMethodHTMLForm`: HTML form authentication
- `NSURLAuthenticationMethodNTLM`: NT LAN Manager (NTLM) authentication [NTLM]
- `NSURLAuthenticationMethodNegotiate`: Use negotiate authentication.
- `NSURLAuthenticationMethodClientCertificate`: SSL/TLS client certificate authentication
- `NSURLAuthenticationMethodServerTrust`: SSL/TLS server trust authentication

`NSURLConnection` sends its delegate the message `connection:canAuthenticateAgainstProtectionSpace:` to check if the delegate can perform the authentication against a given protection space. If the delegate responds with the boolean value NO, the `NSURLConnection` tries to obtain

credentials from the Keychain to fulfill the authentication challenges for that protection space. If the delegate returns YES, the delegate is sent the message `connection:didReceiveAuthenticationChallenge:` in which it is responsible to respond to the authentication challenge. The second parameter of `connection:didReceiveAuthenticationChallenge:` is an instance of the class `NSURLAuthenticationChallenge` which represents the authentication challenge. Among other properties, this object contains the number of previous authentication failures, the protection space the client wants to access, and the instance that performs the actual loading (called the "sender" of the authentication challenge). Typically, the delegate provides the sender a credential that should be used for the challenge and tells it to continue loading with this credential, or it tells the sender to continue without a credential.

The following chapters explain how to implement HTTP Basic and Digest Access Authentication, server authentication via TLS, and mutual authentication of client and server via TLS.

HTTP Basic and Digest Access Authentication

In order to respond to an HTTP Basic or Digest Access Authentication challenge [RFC2617], the delegate creates an instance of `NSURLCredential` with a username and a password. It then obtains the sender from the `NSURLAuthenticationChallenge` and tells it to continue with the credential for the current challenge.

Listing 3-36 depicts the implementation of HTTP basic and digest authentication. The methods are implemented by the `NSURLConnection`'s delegate.

```
- (BOOL)connection:(NSURLConnection *)connection
  canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)
  protectionSpace
{
  // The delegate can handle HTTP basic and digest authentication
  return ([[protectionSpace authenticationMethod]
           isEqualToString:NSURLAuthenticationMethodHTTPBasic] ||
          [[protectionSpace authenticationMethod]
           isEqualToString:NSURLAuthenticationMethodHTTPDigest]);
}

- (void)connection:(NSURLConnection *)connection
  didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)
  challenge
{
  if ([[[challenge protectionSpace] authenticationMethod]
       isEqualToString:NSURLAuthenticationMethodHTTPBasic] ||
      [[[challenge protectionSpace] authenticationMethod]
       isEqualToString:NSURLAuthenticationMethodHTTPDigest])
  {
    // Create a credential containing the username and password
    NSURLCredential *credential = [NSURLCredential
        credentialWithUser:@"USERNAME"
                  password:@"SECRET_PASSWORD"
               persistence:NSURLCredentialPersistenceForSession];

    // Use the credential for the authentication challenge
    [[challenge sender] useCredential:credential
           forAuthenticationChallenge:challenge];
  }
  else
  {
    // Continue without credential for other authentication methods
    [[challenge sender]
     continueWithoutCredentialForAuthenticationChallenge:challenge];
  }
}
```

Listing 3-36: HTTP Basic and Digest Access Authentication

SSL/TLS with Server Authentication

Transport Layer Security (TLS) [TLSv1] and its predecessor Secure Sockets Layer (SSL) [SSLv2, SSLv3] are often used in conjunction with HTTP and called HTTPS. In the most common use case of an HTTPS connection only the server authenticates towards the client. The server owns a certificate and the corresponding private key. During the establishment of the SSL/TLS connection, the server sends its certificate to the client who evaluates the certificate to determine if it is valid or not. If the certificate is not valid, the client may cancel its attempt to establish the connection, because the server is possibly an adversary that pretends to be the server (a scenario called *man-in-the-middle attack*). However, if the certificate is valid, the client can be sure that it establishes a secure connection with the intended server.

The previous chapter "Certificate, Key, and Trust Services: Certificates, Asymmetric Encryption, and Digital Signatures" discussed the APIs to work with certificates, private keys, policies, and trust management objects. Listing 3-37 therefore focusses on the aspects of the URL Loading System.

```
- (BOOL)connection:(NSURLConnection *)connection
  canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)
  protectionSpace
{
  // The delegate can handle server authentication
  return ([[protectionSpace authenticationMethod]
           isEqualToString:NSURLAuthenticationMethodServerTrust]);
}

- (void)connection:(NSURLConnection *)connection
  didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)
  challenge
{
  if ([[[challenge protectionSpace] authenticationMethod]
        isEqualToString:NSURLAuthenticationMethodServerTrust]) {
    // Obtain trust management object from authentication challenge
    SecTrustRef serverTrust = [[challenge protectionSpace]
                               serverTrust];

    // Evaluate trust with Certificate, Key, and Trust Services
    BOOL trusted = ...;
    if (!trusted) {
      // Certificate not trustworthy. Ask the user if she trusts the
      // certificate or fail gracefully.
    } else {
      // Create a credential with the trusted server certificate
      NSURLCredential *credential =
        [NSURLCredential credentialForTrust:serverTrust];

      // Use the credential for the authentication challenge
      [[challenge sender] useCredential:credential
             forAuthenticationChallenge:challenge];
    }
  } else {
    // Continue without credential for other authentication methods
    [[challenge sender]
     continueWithoutCredentialForAuthenticationChallenge:challenge];
  }
}
```

Listing 3-37: Server Authentication with SSL/TLS

TLS with Mutual Authentication of Client and Server

The URL Loading System also supports mutual authentication of both client and server with TLS. The authentication of the server towards the client was described in the section above ("TLS with Server Authentication").

The client authenticates towards the server by responding to an authentication challenge of authentication type `NSURLAuthenticationMethodClientCertificate` within the method `connection:didReceiveAuthenticationChallenge:`. The client selects an arbitrary `SecIdentityRef` obtained via Keychain Services or Certificate, Key, and Trust Services and creates an `NSURLCredential` with it. It then tells the sender of the challenge to use the credential for the current challenge (see Listing 3-38).

```
- (BOOL)connection:(NSURLConnection *)connection
  canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)
  protectionSpace
{
  // The delegate can provide a client certificate and is able to
  // authenticate the server
  return (
    [[protectionSpace authenticationMethod]
      isEqualToString:NSURLAuthenticationMethodServerTrust] ||
    [[protectionSpace authenticationMethod]
      isEqualToString:NSURLAuthenticationMethodClientCertificate]
  );
}

- (void)connection:(NSURLConnection *)connection
  didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)
  challenge
{
  if ([[[challenge protectionSpace] authenticationMethod]
        isEqualToString:NSURLAuthenticationMethodServerTrust]) {
    // As described in Listing 3-37 above
  } else if ([[[challenge protectionSpace] authenticationMethod]
     isEqualToString:NSURLAuthenticationMethodClientCertificate]) {
    // Provide client identity (obtained via, e.g., Certificate,
    // Key, and Trust Services)
    SecIdentityRef identity = ...;

    NSURLCredential *clientCertificateCredential =
      [NSURLCredential
       credentialWithIdentity:identity
                 certificates:nil
                  persistence:NSURLCredentialPersistenceForSession];

    [[challenge sender] useCredential:clientCertificateCredential
           forAuthenticationChallenge:challenge];
  } else {
    // As described in Listing 3-37 above
  }
}
```

Listing 3-38: Mutual Authentication of Client and Server with SSL/TLS

3.3.5.3. Secure Network Connections with `UIWebView`

A shortcoming of `UIWebView` is the missing capability to let its delegate respond to authentication challenges. If a needed credential is stored in the global credential storage (accessibly via `NSCredentialStorage`) for a particular protection space or the server owns a certificate that is signed by a trusted third party, the `UIWebView` is able to load the requested resource. However, if a credential must be provided by the user, the server only has a self-signed certificate and/or the server requires the client to authenticate with a client certificate over TLS, `UIWebView` fails to load the requested resource. Because, unlike `NSURLConnection`, `UIWebView` does not expose the authentication challenges to its delegate, the delegate cannot influence the response to those challenges.

A possible solution to circumvent this shortcoming is to use the URL Loading System to authenticate the client prior to loading the resource with `UIWebView` [SO_WEBVIEW]. When the `NSURLConnection`'s delegate receives the message `connection:didReceiveResponse:`, the delegate cancels the loading with the URL Loading System and resends the request via `UIWebView`. Listing 3-39 demonstrates this approach.

```
- (void)startLoadingURL:(NSURL *)URL
{
  [self setURL:URL];  // Property of class
  [self setAuthenticated:NO];  // Property of class
  [[self webView] loadRequest:[NSURLRequest requestWithURL:URL]];
}

#pragma mark UIWebViewDelegate methods

- (BOOL)webView:(UIWebView *)webView
   shouldStartLoadWithRequest:(NSURLRequest *)request
   navigationType:(UIWebViewNavigationType)navigationType
{
  if (![self authenticated]) {
      // Perform authentication with URL Loading System
      NSURLConnection *connection = [[NSURLConnection alloc]
                                     initWithRequest:request
                                     delegate:self];
      return NO;
  }
  return YES;
}

#pragma mark NSURLConnection delegate methods

- (BOOL)connection:(NSURLConnection *)connection
  canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)
  protectionSpace
{
  // See the section about the URL Loading System.
}

- (void)connection:(NSURLConnection *)connection
  didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)
  challenge
{
  // See the section about the URL Loading System.
}

- (void)connection:(NSURLConnection *)connection
  didReceiveResponse:(NSURLResponse *)response
{
    [connection cancel];

    [self setAuthenticated:YES];

    NSURLRequest *originalRequest =
        [NSURLRequest requestWithURL:[self URL]];
    [[self webView] loadRequest:originalRequest];
}
```

Listing 3-39: Mixing *URL Loading System* and `UIWebView` Loading

3.3.5.4. Secure Network Connections with CFNetwork

CFNetwork is built upon `CFSocket` and `CFStream`, two classes of the Core Foundation framework. `CFSocket` is a wrapper around BSD sockets that provides almost all their functionality [CFN_PROG_G]. A developer can create a `CFSocket` object from a BSD socket and obtain the raw socket from a `CFSocket` object that was created by the framework. Therefore, CFNetwork allows a developer to work with the provided higher-level constructs, but at the same time also use the BSD socket directly.

`CFStream` provides streaming-based read and write access to `CFSocket`. A stream is either an instance of type `CFReadStream` or `CFWriteStream`, both are subtypes of `CFStream`. The functions that read from a read stream block until the stream has bytes to read. Likewise, functions that write to a write stream block until there is space available in the stream. In order to prevent blocking the current thread, the stream can be scheduled on the threads run loop such that the stream's callbacks are called asynchronously when the (read) stream has bytes available to be read or the (write) stream has space available to be written to.

A `CFStream` object is toll-free bridged with `NSStream`, `CFReadStream` is toll-free bridged with `NSInputStream` and `CFWriteStream` is toll-free bridged with `NSOutputStream` [TOLL_FREE_BRIDGING]. This combines a fine-grained configuration of streams at the CFNetwork layer and the more comfortable usage as Objective-C objects. A class that conforms to the `NSStreamDelegate` protocol can be configured as the delegate of an `NSStream` to receive messages corresponding to stream events asynchronously.

CFNetwork consists of multiple APIs: CFSocketStream, CFHTTP, CFHTTPAuthentication, CFFTP, CFHost, CFNetServices, and CFNetDiagnostics (see Figure 3-20 [CFN_PROG_G]). CFSocket and CFStream are actually part of Core Foundation, not CFNetwork.

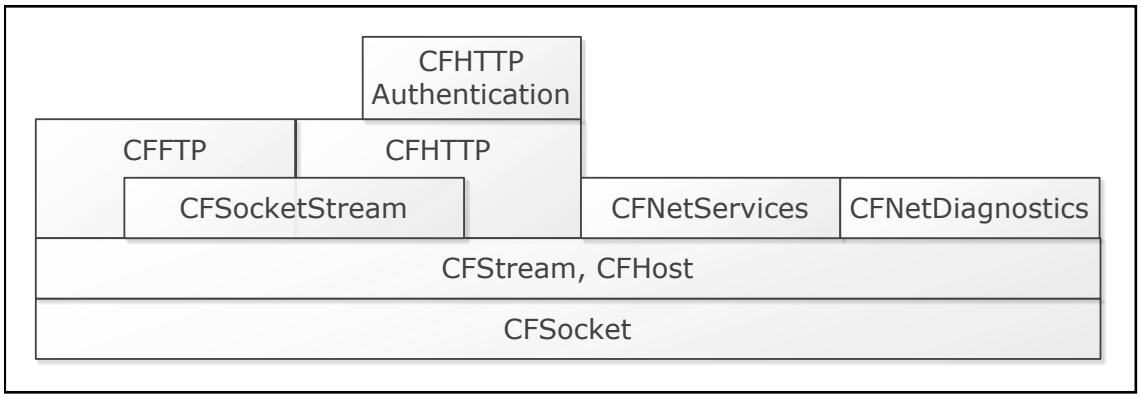

Figure 3-20: CFNetwork

- CFSocketStream is a collection of properties to configure SSL / TLS and proxy settings of socket streams. CFHTTP implements functions for constructing and sending HTTP requests and retrieving HTTP responses.
- CFHTTPAuthentication can apply authentication credentials to HTTP requests. It supports HTTP Basic and Digest Access Authentication, NTLM, and Simple and Protected GSS-API Negotiation Mechanism [RFC4178].
- CFFTP provides functions to create read streams to download files and write streams to upload files with the FTP protocol. It is also possible to get directory listings and to create new directories.
- CFHost provides resolution of hosts in order to retrieve the name, address, and information about the reachability of a particular host.
- CFNetServices enables the developer to use Bonjour to register new services and discover existing services on the local network. Bonjour is Apple's implementation of Zero Configuration Networking (zeroconf).
- CFNetDiagnostics can extract network status information from a connection.

CFNetwork and SSL/TLS

`CFStream` inherently supports the creation of SSL/TLS streams. The property `kCFStreamPropertySSLSettings` allows the developer to provide a dictionary with options that configure the SSL/TLS stream via `CFReadStreamSetProperty()` and `CFWriteStreamSetProperty()` before opening the stream. The supported options regarding SSL/TLS are `kCFStreamPropertySocketSecurityLevel` and `kCFStreamPropertySSLSettings`:

The stream's security level property **`kCFStreamPropertySocketSecurityLevel`** can be set to one of the following values [CF_SS_REF]:
- `kCFStreamSocketSecurityLevelNone`: Do not use SSL/TLS.
- `kCFStreamSocketSecurityLevelSSLv2`: Use SSL version 2.
- `kCFStreamSocketSecurityLevelSSLv3`: Use SSL version 3. Use SSL version 2 if SSL version 3 is not available.
- `kCFStreamSocketSecurityLevelTLSv1`: Use TLS version 1. Only TLS version 1.0 is supported [IPHONE_SEC_O].
- `kCFStreamSocketSecurityLevelNegotiatedSSL`: Use the highest security level that can be negotiated with the peer. If no security level can be negotiated, the connection establishment is cancelled.

If the developer wants to have finer-grained control over the SSL/TLS stream settings, she can provide a dictionary for the key **`kCFStreamPropertySSLSettings`** which can contain entries with the following keys [CF_SS_REF]:
- `kCFStreamSSLLevel`: Equivalent to configuring the security level of the stream with a value for the key `kCFStreamPropertySocketSecurityLevel` mentioned above. The default security level is `kCFStreamSocketSecurityLevelNegotiatedSSL`.

- `kCFStreamSSLAllowsExpiredCertificates`: Indicates if expired certificates are allowed. The value for this property is either `kCFBooleanTrue` or `kCFBooleanFalse`. By default, expired certificates are not allowed.
- `kCFStreamSSLAllowsExpiredRoots`: Determines if expired root certificates are allowed. The value for this property is either `kCFBooleanTrue` or `kCFBooleanFalse`. By default, expired root certificates are not allowed.
- `kCFStreamSSLAllowsAnyRoot`: Determines if root certificates are allowed as server certificates. The value for this property is either `kCFBooleanTrue` or `kCFBooleanFalse`. By default, root certificates are not allowed.
- `kCFStreamSSLValidatesCertificateChain`: Determines if the peer's certificate chain is validated during the connection establishment. The value for this property is either `kCFBooleanTrue` or `kCFBooleanFalse`. By default, the certificate chain is validated.
- `kCFStreamSSLPeerName`: Overwrites the name used for the validation of the peer's certificate. The default peer name is the hostname used for creating the stream. No peer name will be used if the stream was created without providing a hostname. A value of `kCFNull` prevents peer name verification.
- `kCFStreamSSLCertificates`: An array (`CFArrayRef`) with an identity (`SecIdentityRef`) at index 0 and optionally one or more intermediate certificates (`SecCertificateRef`). This entry configures the identity that is used for the authentication towards the peer.
- `kCFStreamSSLIsServer`: Determines if the stream should act as the server in the SSL/TLS protocol. The value for this property is either `kCFBooleanTrue` or `kCFBooleanFalse`. By default, the stream acts as the client in the SSL/TLS protocol.

The security level of a socket stream is also configurable on `NSStream` instances at the Objective-C layer with the key `NSStreamSocketSecurityLevelKey` (Listing 3-40):

```
// 'inputStream' is of class NSInputStream
[inputStream setProperty:NSStreamSocketSecurityLevelTLSv1
                  forKey:NSStreamSocketSecurityLevelKey];
```

Listing 3-40: Setting the Security Level of a SSL / TLS Stream

The meaning of each value for **`NSStreamSocketSecurityLevelKey`** is equivalent to the corresponding value for `kCFStreamPropertySocketSecurityLevel` mentioned above:

- `NSStreamSocketSecurityLevelNone`: Do not use SSL/TLS.
- `NSStreamSocketSecurityLevelSSLv2`: Use SSL version 2.
- `NSStreamSocketSecurityLevelSSLv3`: Use SSL version 3. Use SSL version 2 if SSL version 3 is not available.
- `NSStreamSocketSecurityLevelTLSv1`: Use TLS version 1. Only TLS version 1.0 is supported [IPHONE_SEC_O].

- `NSStreamSocketSecurityLevelNegotiatedSSL`: Use the highest security level that can be negotiated with the peer. If no security level can be negotiated, the connection establishment is cancelled.

However, the settings configurable through `kCFStreamPropertySSLSettings` must be set at the C layer on `CFStream` instances.

After opening the stream, the peers perform the SSL/TLS handshake protocol to establish a secure connection. The handshake fails if the security levels of the peers are not compatible with each other. In this case, the stream cancels the connection establishment and sets an error code that can be used for further evaluation. The security level becomes the highest level that is available on both sides if the stream's configured security level is `NSStreamSocketSecurityLevelNegotiatedSSL` (or `kCFStreamSocketSecurityLevelNegotiatedSSL`). Nonetheless, if the peer does not provide any security, the connection establishment is cancelled and an error is set for further evaluation.

If the SSL/TLS handshake succeeds, a trust management object containing the peer's certificate, its anchor certificates, and an SSL security policy object can be obtained with `CFReadStreamGetProperty()` and `CFWriteStreamGetProperty()` for evaluation of the certificate chain. The property `kCFStreamPropertySSLPeerTrust` contains the trust management object. `kCFStreamPropertySSLPeerCertificates` contains an array with the certificates of the certificate chain.

TLS with Server Authentication

Listing 3-41 shows how to configure a stream for SSL/TLS. First, the initial HTTP request is created. `CFReadStreamCreateForHTTPRequest()` creates a read stream for the HTTP request. A property of a stream can be configured with `CFReadStreamSetProperty()`. The `CFReadStream` is cast to an `NSInputStream` in order to leverage the comfort of an Objective-C API. The delegate is set and the stream is scheduled in the current Run Loop. The stream sends messages to its delegate asynchronously whenever a new event occurs.

```
NSURL *theURL = [NSURL URLWithString:@"https://www.example.com"];

CFHTTPMessageRef httpRequest = CFHTTPMessageCreateRequest(
                                  kCFAllocatorDefault,
                                  CFSTR("GET"),
                                  (CFURLRef)theURL,
                                  kCFHTTPVersion1_1);

// Create stream to retrieve HTTP response
CFReadStreamRef readStream = CFReadStreamCreateForHTTPRequest(
                                 kCFAllocatorDefault,
                                 httpRequest);

// Configure stream to follow autoredirections
CFReadStreamSetProperty(readStream,
                        kCFStreamPropertyHTTPShouldAutoredirect,
                        kCFBooleanTrue);

// Configure SSL/TLS security settings
NSDictionary *tlsSettings =
  [NSDictionary dictionaryWithObjectsAndKeys:
    (id)kCFStreamSocketSecurityLevelNegotiatedSSL,
    (id)kCFStreamSSLLevel,
    (id)kCFBooleanFalse,
    (id)kCFStreamSSLValidatesCertificateChain,
    nil];
CFReadStreamSetProperty(readStream,
                        kCFStreamPropertySSLSettings,
                        tlsSettings);

// Use toll-free bridging between CFReadStream and NSInputStream
NSInputStream *inputStream = (NSInputStream *)readStream;
[inputStream setDelegate:self];

// Schedule stream in Run Loop
[inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                       forMode:NSDefaultRunLoopMode];

[inputStream open];

// NSStreamDelegate methods are invoked asynchronously...
```

Listing 3-41: Configure and Open a HTTPS Connection

Whenever the stream encounters an event (`NSStreamEvent`), it sends the message `stream:handleEvent:` to its delegate (see Listing 3-42). When the stream has bytes available, the delegate retrieves the server's trust management object and evaluates it.

```
- (void)stream:(NSStream *)stream
   handleEvent:(NSStreamEvent)streamEvent
{
  NSInputStream *inputStream = (NSInputStream *)stream;
  switch (streamEvent)
  {
    case NSStreamEventHasBytesAvailable:
    {
      SecTrustRef serverTrust =
        [inputStream propertyForKey:(NSString *)
                     kCFStreamPropertySSLPeerCertificates];

      // Evaluate server trust with Certificate, Key, and Trust
      // Services API ...

      // Append available bytes to property 'receivedData'
      // which is of class NSMutableData ...

      break;
    }

    case NSStreamEventEndEncountered:
    {
      // 'receivedData' is a property of the current instance
      NSString *htmlBody = [[NSString alloc]
                            initWithData:[self receivedData]
                            encoding:NSUTF8StringEncoding];

      [inputStream close];

      // Do something with response data ...

      break;
    }

    default:
      break;
}
```

Listing 3-42: Processing the HTTPS Response

3.3.5.5. Summary

iOS provides several APIs to implement network functionality in applications. The more abstraction an API provides the less control it gives a developer over the actual network connection.

`UIWebView` is a view class that loads and displays Web content. If the content is located on a remote server, `UIWebView` sends its delegate messages while loading the content. However, delegate messages corresponding to authentication requests are missing and as a result a delegate cannot control the authentication process. If the `UIWebView` is not able to perform an authentication request on its own, it fails.

The URL Loading System provides an Objective-C API that enables a delegate to response to authentication requests made by the server. The delegate is sent messages for HTTP Basic Authentication, HTTP Digest Authentication, SSL/TLS server certificate authentication, and SSL/TLS client certificate authentication challenges. Regarding the SSL/TLS session properties only the security level can be configured in the URL Loading System.

CFNetwork is a C API that allows finer-grained control over the network connection as `UIWebView` or the URL Loading System. Many SSL/TLS properties can be configured: if root certificates are allowed as server certificates, if expired root certificates are allowed, if expired certificates are allowed, the name used for the validation of the peer's certificate, if the stream should act as the server in the SSL/TLS protocol, if the peer's certificate chain is validated during the connection establishment, and the identity that is used for the authentication towards the peer. Developers have to implement more by themselves when using CFNetwork because it provides less abstraction than the other two networking APIs mentioned above.

BSD sockets are not specific to iOS. They are available in all BSD-like operating systems. They provide the most control but also the least abstraction available to implement networking functions in an iOS application.

3.3.6. Data Protection

The original iPhone, iPhone 3G, iPod touch (1st generation), and iPod touch (2nd generation) do not contain specific encryption hardware. Files in the filesystem are stored unencrypted on these devices. Only the sensitive information of Keychain items are transparently encrypted (see chapter "Keychain Services: Secure Storage for Sensitive Information") by a device-specific hardware key. Remote wipe is the capability of an iOS device to receive a command over the Internet to securely delete the contents of the hard disk. The conventional way is to overwrite every single bit. This approach is used by the mentioned iOS devices. It takes a long time to complete, because the size of the hard disk is between 4GB and 32GB and writing that amount of data to the hard disk is a long-running process.

Starting with iPhone 3GS, iPod touch (3rd generation), and iPad, every device contains specific hardware to perform symmetric AES-256 full disk encryption [IPHONE_SEC_O]. All files on these devices are transparently encrypted with a device-specific key. This allows a remote wipe to be performed much faster, because the device can cryptographically delete all the files on the hard disk by securely deleting the encryption key. The length of an AES-256 encryption key is 256 bits and overwriting 256 bits is very fast. The purpose of this mechanism is to provide an efficient way to remotely wipe an iOS device and not to provide confidentiality of the hard disk's contents against local attacks in which the attacker has unrestricted physical access to the device.

iOS 4 introduces Data Protection, a completely revised architecture to protect files and Keychain items. This architecture is not publicly documented at the moment. Registered members of the iOS Developer Program [IOS_DEV_PROG] can access the conference videos and slides from Apple's Worldwide Developers Conference (WWDC) 2010 [WWDC2010]. Session 209 ("Securing Application Data") provides an overview of Data Protection and shows how developers can leverage the Data Protection API to protect their application's assets. Unfortunately, this information is currently protected by a non-disclosure agreement (NDA) and is only accessible by members of the iOS Developer Program. Apple has not responded to a request made on February 20, 2011, to use the information of session 209 in this thesis.

The problem with the transparent encryption and decryption mentioned earlier is that an attacker can obtain the contents of the hard disk if she has physical access to the device [3GS_ENCRYPTION]. The goal of Data Protection is to minimize the information disclosure to a local attacker by tying the confidentiality of protected files and Keychain items to the availability of the user's passcode. When the user enters the passcode to unlock the device, protected files and Keychain items become available and are transparently decrypted on access. When the user locks the device, protected data becomes unavailable after the passcode grace period. The passcode grace period is the time between turning off the display and the device requiring the

user's passcode for the unlock. Attempts to access data on a locked device after the grace period fails.

[DATA_PROT] describes the function of Data Protection as follows:

> *"Data protection has the primary advantage of using the user's passcode or password to derive a key that is used to encrypt data on the device. When the phone is locked or turned off, the key is immediately erased, making data secured on the device inaccessible. [...] to mitigate the threat of a brute force attack, the file encryption requires a key generated by the device itself, in addition to the key derived from the user's password. This slows brute forcing because the encryption key generation process is slow by design: the iPhone 4 takes about 50 milliseconds to derive the key once the user submits a password. This means an attacker can guess only about 20 passwords per second."*

An attack that is performed off-device is infeasible. To the best of the author's knowledge, the device key is only available in the hardware of the device and a method which easily extracts it without breaking the device is not publicly known. Off-device attacks would therefore have to brute-force an AES-256 encryption key. On-device attacks, however, could perform a "smart" brute-force attack by intelligently guessing the user's passcode. This could possibly considerably reduce the amount of time needed to break the system. By making the encryption key derivation process slow by design, a brute-force attack is limited to guessing about twenty passwords per second. This is very slow and renders on-device brute-force attacks infeasible, too.

When upgrading a device with hardware encryption capabilities from iOS 3 to iOS 4, the filesystem has to be re-formatted before iOS can use Data Protection [APPLE_DATA_PROT]. If iOS is able to use Data Protection, it is automatically enabled when the user configures a passcode. If no passcode is set, Data Protection is disabled. The Passcode Lock screen in the Settings application shows whether Data Protection is enabled or disabled (see Figure 3-21).

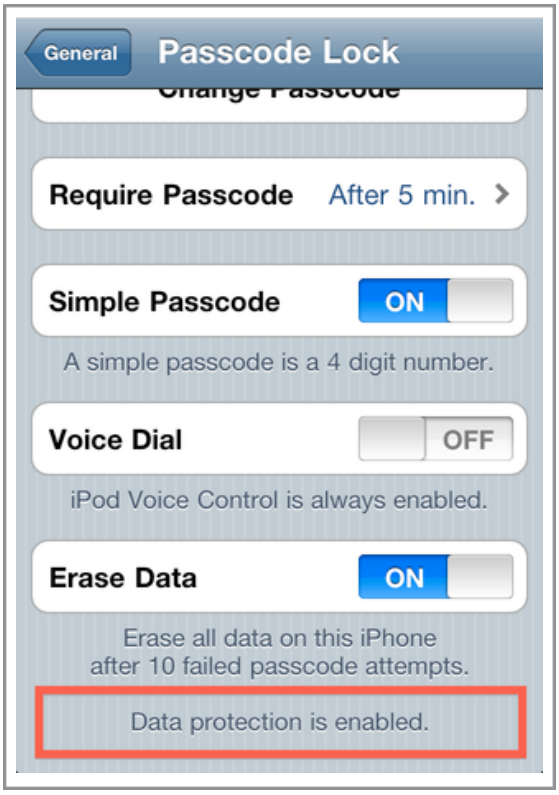

Figure 3-21: Settings application -> Passcode Lock

Apple currently uses Data Protection only in its application Mail to provide *"an additional layer of protection for [..] email messages and attachments"* [APPLE_DATA_PROT]. How third-party applications can adopt Data Protection to protect their data is explained later in this chapter.

[DATA_PROT] points out that *"a weakness in the data protection system is something called the "Escrow Keybag", which is a collection of keys necessary to decrypt every file on the device without requiring the user's password. This was done to allow computers to sync with the iPhone without asking the user for the password. [...] Apple's rationale was that if the PC containing the escrow keybag was obtained, an attacker most likely already had the user's important data anyway".*
Data Protection is useless if an attacker has access to the computer which syncs with the device, even if iTunes backups are encrypted. On a Mac, the directory `~/Library/Application Support/MobileSync/Backup/` contains the backups of the synced devices. The backup of a device is stored in a distinct directory. Listing 3-43 shows the contents of the backup directory on the author's computer.

```
$ pwd
/Users/mbinna/Library/Application Support/MobileSync/Backup

$ ls
0ae33a34825f1d8c88b4cb8a8d8d865b7528f9dc
4f7b2c888d2d80f9638cdc1faf3d0d89fa63b67d
be7963cddb76f08502c0b38abb66585ba7002343
```

Listing 3-43: Contents of ~/Library/Application Support/MobileSync/Backup/

Listing 3-43 shows that this computer synchronizes with three devices. The directory name in which the latest backup of a synchronized device is stored is the hexadecimal representation of a SHA-1 hash (40 byte). The directory /private/var/db/lockdown contains property list files which contain the credentials to "[...] allow the investigator to bypass the passcode on the device" [MORRISSEY_FORENSIC] and access the files without entering the passcode. For every directory name in ~/Library/Application Support/MobileSync/Backup/ there exists a property list file with the same name in /private/var/db/lockdown. Listing 3-44 shows the contents of the directory on the author's computer.

```
$ pwd
/private/var/db/lockdown

$ ls
0ae33a34825f1d8c88b4cb8a8d8d865b7528f9dc.plist
4f7b2c888d2d80f9638cdc1faf3d0d89fa63b67d.plist
be7963cddb76f08502c0b38abb66585ba7002343.plist
```

Listing 3-44: Contents of /private/var/db/lockdown

The files in /private/var/db/lockdown are XML property list files named by the UDID of the device they correspond to. Listing 3-45 shows the structure of a lockdown property list file. According to [MORRISSEY_FORENSIC] the Windows version of iTunes stores similar artifacts on Windows computers.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
      <key>DeviceCertificate</key>
      <data>
      ...
      </data>
      <key>DevicePublicKey</key>
      <data>
      ...
      </data>
      <key>EscrowBag</key>
      <data>
      ...
      </data>
      <key>HostCertificate</key>
      <data>
      ...
      </data>
      <key>HostID</key>
      <string>...</string>
      <key>HostPrivateKey</key>
      <data>
      ...
      </data>
      <key>RootCertificate</key>
      <data>
      ...
      </data>
      <key>RootPrivateKey</key>
      <data>
      ...
      </data>
      <key>SystemBUID</key>
      <string>...</string>
</dict>
</plist>
```

Listing 3-45: Contents of a lockdown property list file

Synchronizing a locked iOS device with a new computer without unlocking it is not possible. Otherwise, an attacker could gain access to all files on the device by just synchronizing it with her own computer. Figure 3-22 shows the warning that is displayed by iTunes in such a situation.

Figure 3-22: Trying to sync a locked iPhone with a new computer

The device has to be unlocked to perform the initial sync process. During that process, iTunes creates the lockdown property list file on the computer. From that time on, the user does not have to enter the passcode to sync this device.

A locked (and thus protected) iOS device synchronizes with a new computer without requiring the passcode if an appropriate lockdown property list file is available in the folder `/private/var/db/lockdown` on the attacker's computer. In this scenario, the attacker has access to both the device and the computer of the victim. She copies the lockdown property list file from the victim's to her computer and syncs the device. However, because the attacker has access to the device backup on the victim's computer, no additional data is disclosed to the attacker.

3.3.6.1. Protection Classes

iOS devices can protect files and Keychain items with the user's passcode when they are not needed. Additionally, Keychain items can be restricted to the current device. When a backup is restored to another device, those items will not migrate to the other device. This is useful for credentials that are specific to the device instead of the user of the device, such as device identities for VPN. The developer defines when the application needs to have access to a file or Keychain item by using one of the following protection classes:

- **Always accessible (lowest protection)**
  - Data in this class is not protected by the user's passcode. It is transparently encrypted and decrypted like in iOS 3 when Data Protection was not available. This data is always accessible and therefore vulnerable to local attacks - even if the user sets a secure passcode and a short passcode grace period. An attacker

who has physical control over the device (e.g., the device is found or stolen by the attacker) can perform a jailbreak, boot the jailbroken device, access the Keychain's SQLite database in the file system, and directly use system functions for decrypting Keychain data [LOST_IPHONE].

- Keychain item: When creating an item, the value of the attribute `kSecAttrAccessible` has to be set to `kSecAttrAcessibleAlways` or `kSecAttrAccessibleAlwaysThisDeviceOnly` (if the item should not migrate to another device when a backup is restored).
- File: After creating a file, the method `setAttributes:ofItemAtPath:error:` of NSFileManager is used to set the attribute `NSFileProtectionKey` to the value `NSFileProtectionNone`. When writing an `NSData` instance to a file with the method `writeToFile:options:error:`, the option `NSDataWritingFileProtectionNone` has to be used.

- **Accessible after first unlock of device**
  - Data in this class is protected by the user's passcode. The data becomes available after the user unlocks the device for the first time after a restart. It remains available until the device is powered off or restarted. The data is protected from attacks that require a restart of the device. A restart renders the data protected by this protection class unavailable. The encrypted data cannot be decrypted with system functions, because the user's passcode is not available and therefore the encryption keys, which are derived from the passcode, are missing.
  - Keychain item: When creating the item, the value of the attribute `kSecAttrAccessible` has to be set to `kSecAttrAccessibleAfterFirstUnlock` or `kSecAttrAfterFirstUnlockThisDeviceOnly` (if the item should not migrate to another device when a backup is restored).
  - File: Files on this level are not supported by the API.

- **Accessible when device unlocked (highest protection)**
  - Data in this class is protected by the user's passcode. It becomes available every time the user unlocks the device by entering the correct passcode. It becomes unavailable when the passcode grace period is over after the user locked the device. This class offers the highest protection among the available protection classes. Even if a local attacker can access the filesystem and execute her own code without rebooting the device, she will not be able to access the protected data.
  - Keychain item: When creating the item, the value of the attribute `kSecAttrAccessible` has to be to `kSecAttrAccessibleWhenUnlocked` or `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` (if the item should not migrate to another device when a backup is restored).
  - File: After creating a file, the method `setAttributes:ofItemAtPath:error:` of `NSFileManager` is used to set the attribute `NSFileProtectionKey` to the value `NSFileProtectionComplete`. When writing an `NSData` instance to a file with the

method `writeToFile:options:error:`, the option `NSDataWritingFileProtectionComplete` has to be used.

3.3.6.2. Writing Data from Memory to a New Protected File

When the data is already in memory (represented by an instance of class `NSData`), it can be written to a file with `writeToFile:options:error:` and providing the option `NSDataWritingFileProtectionComplete` (see Listing 3-46). This creates a protected file in the filesystem and then writes the data into that file.

```
NSData *data = ...; // Data to be stored in the file
NSString *path = ...;  // Path of file in filesystem

NSError *error = nil;
BOOL successful = [data writeToFile:path
                        options:NSDataWritingFileProtectionComplete
                        error:&error];

if (!successful) {
  // An error occured.
}
```

Listing 3-46: Writing data to a protected file

3.3.6.3. Protecting a File

A regular file can be protected by setting an extended attribute in the file's metadata in the filesystem. This is accomplished with the method `setAttributes:ofItemAtPath:error:` of `NSFileManager` (Listing 3-47). Protecting a file after its creation is necessary in situations where an application uses Data Protection in a later version or when a pre-iOS-4 backup is restored and the application files are not protected (because this feature was not available prior to iOS 4). An can application change the protection class of these files after the first launch of the application.

```
// Path to a regular (unprotected) file
NSString *path = ...;

// Extended attribute to protect the file
NSDictionary *protectionAttribute =
  [NSDictionary dictionaryWithObject:NSFileProtectionComplete
                              forKey:NSFileProtectionKey];

NSFileManager *fileManager =
  [[[NSFileManager alloc] init] autorelease];
NSError *error = nil;
BOOL successful = [fileManager setAttributes:protectionAttribute
                                ofItemAtPath:path
                                       error:&error];
if (!successful) {
  // An error occured.
}
```

Listing 3-47: Change the protection class of an existing file

The file protection can also be removed as Listing 3-48 demonstrates. After removing the extended attribute the contents of the file are always available, even when the device is locked.

```
// Path to a protected file
NSString *protectedFilePath = ...;

// Set extended attribute to unprotect the file
NSDictionary *attributes =
  [NSDictionary dictionaryWithObject:NSFileProtectionNone
                              forKey:NSFileProtectionKey];
NSFileManager *fileManager =
  [[[NSFileManager alloc] init] autorelease];
NSError *error = nil;
BOOL successful = [fileManager setAttributes:attributes
                                ofItemAtPath:protectedFilePath
                                       error:&error];
if (!successful) {
  // An error occured.
}
```

Listing 3-48: Remove the protection attribute from a protected file

When protecting new files with `NSFileProtectionComplete`, it is suggested to first create the file, add the protection attribute, and only then write data to it. This ensures that the written data is always protected [IOS_A_PROG_G].

3.3.6.4. Detecting the Availability of Protected Data

Applications that use Data Protection have to deal with situations in which protected data (files and/or keychain items) may not be available because the device is locked.

Protected data may only be available when the device is unlocked. iOS provides three mechanisms for an application to detect the availability of protected data. The application can detect when protected data did become available or will become unavailable by

- implementing the methods `applicationProtectedDataDidBecomeAvailable:` and `applicationProtectedDataWillBecomeUnavailable:` in the application delegate (Listing 3-49).

- registering a class as observer for the notifications `UIApplicationProtectedDataDidBecomeAvailable` and `UIApplicationProtectedDataWillBecomeUnavailable` (Listing 3-50).

- invoking the method `isProtectedDataAvailable` on the shared `UIApplication` instance (Listing 3-51). This tells the application if protected data is currently available.

```
//
// DataProtectionDemoAppDelegate.h
//
@interface DataProtectionDemoAppDelegate : NSObject
    <UIApplicationDelegate>
{
  // Instance variables...
}

// Properties...

@end


//
// DataProtectionDemoAppDelegate.m
//
@implementation DataProtectionDemoAppDelegate

- (void)applicationProtectedDataDidBecomeAvailable:
  (UIApplication *)application
{
  // Do something after protected data did become available
}

- (void)applicationProtectedDataWillBecomeUnavailable:
  (UIApplication *)application
{
  // Do something before protected data will become unavailable
}

// Other methods...

@end
```

Listing 3-49: Implementing the `UIApplicationDelegate` Data Protection methods

```
- (void)registerForDataProtectionNotifications
{
  // Register for UIApplicationProtectedDataDidBecomeAvailable
  [[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(protectedDataDidBecomeAvailable:)
    name:UIApplicationProtectedDataDidBecomeAvailable
    object:nil];

  // Register for UIApplicationProtectedDataWillBecomeUnavailable
  [[NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(protectedDataWillBecomeUnavailable:)
    name:UIApplicationProtectedDataWillBecomeUnavailable
    object:nil];
}

- (void)unregisterFromDataProtectionNotifications
{
  // Unregister from UIApplicationProtectedDataDidBecomeAvailable
  [[NSNotificationCenter defaultCenter]
    removeObserver:self
    name:UIApplicationProtectedDataDidBecomeAvailable
    object:nil];

  // Unregister from UIApplicationProtectedDataWillBecomeUnavailable
  [[NSNotificationCenter defaultCenter]
    removeObserver:self
    name:UIApplicationProtectedDataWillBecomeUnavailable
    object:nil];
}

- (void)protectedDataDidBecomeAvailable:(NSNotification *)
notification
{
    // Do something with available protected data...
}

- (void)protectedDataWillBecomeUnavailable:(NSNotification *)
notification
{
    // Do something before protected data will become unavailable...
}
```

Listing 3-50: Registering and responding to the Data Protection notifications

```
BOOL protectedDataAvailable =
  [[UIApplication sharedApplication] isProtectedDataAvailable];

if (protectedDataAvailable) {
 // Do something with available protected data...
} else {
 // Protected data is not available. Do something else...
}
```

Listing 3-51: Detecting if protected data is currently available

When protected data becomes unavailable, an application should close all protected files and purge protected information that is no longer needed from memory.

3.3.6.5. Protecting a Keychain Item

The protection class of a Keychain item is set with the attribute `kSecAttrAccessible`. Listing 3-51 shows how to construct a new Keychain item with the protection class `kSecAttrAccessibleAfterFirstUnlock`.

```
NSData *encodedIdentifier = [@"de.rub.emma.DataProtectionDemo"
   dataUsingEncoding:NSUTF8StringEncoding];

NSData *encodedPassword = [@"S3Cret_P4ssC0de!"
   dataUsingEncoding:NSUTF8StringEncoding];

// Construct a Keychain item
NSDictionary *keychainItem =
  [NSDictionary dictionaryWithObjectsAndKeys:
     kSecClassGenericPassword, kSecClass,
     encodedIdentifier, kSecAttrGeneric,
     encodedIdentifier, kSecAttrService,
     @"USERNAME", kSecAttrAccount,
     kSecAttrAccessibleAfterFirstUnlock, kSecAttrAccessible,
     encodedPassword, kSecValueData
     nil];

// Add the item to the keychain
OSStatus status = SecItemAdd((CFDictionaryRef)attributes, NULL);
if (status != errSecSuccess) {
  // An error occured.
}
```

Listing 3-52: Adding a Keychain item with protection class

New items that are added to the Keychain without specifying an explicit protection class have the protection class `kSecAttrAccessibleWhenUnlocked` by default. This is a secure default because it is the most protective class available on iOS 4.

After upgrading the device to iOS 4, Keychain items that have already existed prior to the upgrade (when Data Protection was not available) belong to the protection class `kSecAttrAccessibleAlways`. This behavior provides backwards compatibility. Keychain items with the protection class `kSecAttrAccessibleAlways` have the same level of protection like they had in iOS 3. In this case, `kSecAttrAccessibleAlways` does not weaken the security of the Keychain item. Developers who want to better protect their Keychain items have to update old items to a more protective class by themselves.

3.3.6.6. Updating the Protection Class of a Keychain Item

After an upgrade to iOS 4, all Keychain items have the protection class `kSecAttrAccessibleAlways`. A developer may want to update the protection class of an existing Keychain item to a higher protection class. For example, the application wants to update the item to `kSecAttrAccessibleWhenUnlocked`. The item is identified by the Bundle Identifier of the application. Listing 3-53 describes this scenario.

```
// Construct a query to find the Keychain item
NSData *encodedIdentifier =
  [@"BUNDLE_IDENTIFIER" dataUsingEncoding:NSUTF8StringEncoding];
NSDictionary *query = [NSDictionary dictionaryWithObjectsAndKeys:
                       kSecClassGenericPassword, kSecClass,
                       encodedIdentifier, kSecAttrGeneric,
                       encodedIdentifier, kSecAttrService,
                       nil];

// The new protection class
NSDictionary *updatedAttributes = [NSDictionary
  dictionaryWithObject:kSecAttrAccessibleWhenUnlocked
                forKey:kSecAttrAccessible];

// Perform the update
OSStatus updateItemStatus =
  SecItemUpdate((CFDictionaryRef)query,
                (CFDictionaryRef)updatedAttributes);
```

Listing 3-53: Updating the protection class of a Keychain item without data fails

This sample code yields the value `errSecUnimplemented` for the variable `updateItemStatus` in a demo project implemented for this thesis. It seems that the protection class of a Keychain item cannot be changed after it was added to the Keychain. [IOS_A_PROG_G] explains that "Adding protection to a file is a one-way operation [...], you cannot remove the attribute". However, it does not mention the same limitation for Keychain items. In fact, every Keychain item has a protection class, whether intended or unintended by the developer.

An Apple Developer Technical Support (ADTS) incident clarified this issue. ADTS explains that `SecItemUpdate()` requires the Keychain item's data as the attribute `kSecValueData` to perform the update of the attribute `kSecAttrAccessible`. This constraint is currently not documented in the reference documentation. Therefore, ADTS filed a bug internally to add this information to the reference documentation. The email conversation with ADTS is contained on the enclosed disc in the folder "References/Apple Developer Technical Support". Listing 3-54 shows how to correctly update the protection class of a Keychain item.

```
// Construct a query to find the Keychain item
NSData *encodedIdentifier =
  [@"BUNDLE_IDENTIFIER" dataUsingEncoding:NSUTF8StringEncoding];
NSDictionary *query = [NSDictionary dictionaryWithObjectsAndKeys:
                       kSecClassGenericPassword, kSecClass,
                       encodedIdentifier, kSecAttrGeneric,
                       encodedIdentifier, kSecAttrService,
                       nil];

// Obtain the Keychain item's data via SecItemCopyMatching()
NSData *encodedPassword = ...;

NSDictionary *updatedAttributes =
  [NSDictionary dictionaryWithObjectsAndKeys:
    kSecAttrAccessibleWhenUnlocked, kSecAttrAccessible,
    (CFDataRef)encodedPassword, kSecValueData,
    nil];

// Perform the update
OSStatus updateItemStatus =
  SecItemUpdate((CFDictionaryRef)query,
                (CFDictionaryRef)updatedAttributes);

// If the update was successful, 'updateItemStatus' has the value
// errSecSuccess
```

Listing 3-54: Updating the protection class of a Keychain item with data succeeds

3.3.6.7. Multitasking

When an application runs in the background it could try to access a protected file although the passcode grace period is over and the device is locked. The code in Listing 3-55 dispatches a block with Grand Central Dispatch (GCD) that accesses a protected file five seconds after the device was locked. Because the file was written to disk with either `NSDataWritingFileProtectionComplete` or its extended attribute was changed to `NSFileProtectionComplete`, it cannot be accessed. Therefore, the invocation of `dataWithContentsOfFile:` returns `nil` instead of a pointer to instance of class NSData that contains the contents of the file.

```
- (void)protectedDataWillBecomeUnavailable:(NSNotification *)
notification
{
  // Access a protected file (device is currently unlocked)
  NSString *protectedFilePath = ...;
  NSData *protectedData =
    [NSData dataWithContentsOfFile:protectedFilePath];

  // Dispatch a Block that accesses a protected file 5s after the
  // device is locked
  double delayInNanoSeconds = 5.0 * NSEC_PER_SEC;
  dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW,
                                          delayInNanoSeconds);
  dispatch_after(popTime,
                 dispatch_get_global_queue(
                   DISPATCH_QUEUE_PRIORITY_DEFAULT,
                   0),
                 ^{

    // Access protected data (device is currently locked)
    NSData *protectedData =
      [NSData dataWithContentsOfFile:protectedFilePath];

    // 'protectedData' is nil because the file is not accessible.
  });
}
```

Listing 3-55: Accessing a protected file when the device is locked

Accessing a Keychain item with protection class `kSecAttrAccessibleWhenUnlocked` on a locked device causes the function `SecItemCopyMatching()` to return the status `errSecInteractionNotAllowed`. The security server prohibits access to this item. A Keychain item with protection class `kSecAttrAccessibleAfterFirstUnlock` or `kSecAttrAccessibleAlways` is returned successfully when the device is locked.

3.3.6.8. Older Devices

Data Protection is not supported on the original iPhone, iPhone 3G, iPod touch (1st Generation), and iPod touch (2nd Generation). This section explains how an application that uses Data Protection behaves on these devices.

Older iOS devices do not support multitasking. The events "data did become available" and "data will become unavailable" only occur when the device is locked while the application is running in the foreground and the passcode grace period is over. The `UIApplication` instance method `isProtectedDataAvailable` always returns `YES` because the code is always performed by the application running in the foreground.

The term "protected data" is misleading when the application is installed on older iOS devices that lack support of hardware-based full disk encryption. Actually, data is not

protected against a local attacker at all. Assigning a file or Keychain item one of the protection classes that provide better protection does not improve the security of the data on these devices. The local attack described in [LOST_IPHONE] would therefore expose "protected" Keychain item data and "protected" file contents to the attacker. The solution to mitigate this, is to only use newer devices that perform hardware-based full disk encryption, set a secure passcode and a short passcode grace time.

3.3.6.9. iPhone Data Protection in Depth

The slides of the talk *"iPhone data protection in depth"* by Jean-Baptiste Bédrune and Jean Sigwald at the conference Hack In The Box 2011 in Amsterdam provide a detailed explanation about the architecture and mechanisms of Data Protection at the operating system level [HITB2011]. To the best of the author's knowledge, this level of detail about Data Protection has not been publicly available before their presentation in May 2011.

3.3.6.10.Elcomsoft Forensic Toolkit

On 23 May, 2011, Elcomsoft published two posts on their blog which state that they developed a forensic toolkit that is able to circumvent Data Protection of iOS [ELCOMSOFT_DATA_PROTECTION_1], [ELCOMSOFT_DATA_PROTECTION_2]. The toolkit is only available to law enforcement, forensic and intelligence agencies. According to the posts, the toolkit extracts the unencrypted contents of a device on a separate computer offline using Elcomsoft's Phone Password Breaker [ELCOMSOFT_PPB] after performing the following tasks on the acquired device:

- **Obtain a raw copy of the encrypted filesystem image:** This step can use a non-permanent jailbreak and a tool such as the command-line utility `dd` to create a bitwise copy of the filesystem image. The jailbreak can be performed without modifying data on the device [ELCOMSOFT_DATA_PROTECTION_2].
- **Obtain the required keys:** *"Keys computed from the unique device key (UID), which is believed to be embedded in the hardware and is not extractable (so-called keys 0×835 and 0x89B)"*, the *"[..] passcode key which is derived from users' passcode using the unique device key (UID)"*, *"Escrow key(s) which are derived from escrow pairing records using the unique device key (UID)"*, and *"Effaceable storage area which stores number of encryption keys"* [ELCOMSOFT_DATA_PROTECTION_1].
- **Obtain the user's passcode with a brute-force attack:** iOS provides the option to erase all data on the device after 10 failed passcode attempts. *"When brute-forcing the passcode, we don't use API, we call the required functions directly, so these attempts are not counted"*, comments the author Vladimir Katalov [ELCOMSOFT_DATA_PROTECTION_1]. In iOS, the default passcode option "Simple Passcode" restricts the passcode to a 4 digit number. Although the passcode must

be brute-forced on the device (because it is assumed that the unique device key is not extractable), the attack needs at the maximum 40 minutes to succeed. The author suggests to protect the device *"[...] with the good passcode (disabling "simple passcode" option), and provide physical security to both the device and the computer it was connected to"*.

3.3.6.11.Summary

Data Protection is a powerful new security feature of iOS 4. Developers are able to better protect their application's Keychain items and files by tying their availability to the user's passcode. The passcode is then used during the encryption of the data. An attacker with physical access to the device cannot decrypt data of the protection classes "accessible after first unlock of device" and "accessible when device unlocked" without knowing the passcode. A disadvantage for the adoption of Data Protection is that developers have to modify their existing applications and develop new applications with Data Protection in mind.

When upgrading an older device with hardware encryption capabilities that shipped with iOS 3, such as an iPhone 3GS, iPod touch (3. generation), or iPad 1), it is necessary to restore the device to use Data Protection. This additional step increases the risk for users who are not aware of this requirement and use their devices without Data Protection.

The decision to make `kSecAttrAccessibleWhenUnlocked` the default protection class is a good choice for the security of Keychain items. Even when a developer does not adopt Data Protection in her applications, items that are added to the Keychain without specifying a protection class automatically receive the highest protection class available.

Elcomsoft recently announced a forensic toolkit that circumvents Data Protection and is able to extract all contents if it successfully brute-forces the user's passcode on an acquired device. According to their announcement, a brute-force attack on a passcode that consists of a 4 digit number takes 40 minutes at the maximum and 20 minutes on average. Users of iOS devices should therefore configure a strong passcode in order to render the brute-force attack infeasible.

3.4. Summary

By design, the Objective-C programming language lacks good security properties. If a developer knows the name of a private method, it can be invoked. Accessing a private instance variable from outside the class causes a compiler error. Nevertheless,

Key-Value Coding (KVC) enables the access if the developer does not prevent it by implementing the method `accessInstanceVariablesDirectly`.

The Foundation framework provides many classes and mechanisms that are used in almost every iOS application, e.g., the string classes `NSString` and `NSMutableString`. Instances of these classes are often initialized with format strings which provide the risk of introducing format string vulnerabilities if the developer does not use them in a secure manner. In addition to the more secure Foundation class `NSString` (and its subclass `NSMutableString`), Objective-C also allows the usage of C strings which provide the risk of introducing buffer overflow vulnerabilities when the string's termination character `'\0'` is not handled correctly.

A developer can use several security-related APIs to protect application data. Common Crypto provides functions to perform symmetric encryption and decryption as well as to derive digests and HMACs. Keychain Services provides access to the Keychain, a secure storage for credentials, certificates, and private keys. The sensitive information of Keychain items is stored encrypted on disk. Every application can only obtain its own Keychain items. However, applications from the same developer can use access groups to share Keychain items among themselves. Certificate, Key, and Trust Services provides an API to extract certificates and private keys from PKCS#12 files, to validate certificates, to perform asymmetric encryption and decryption as well as to create and verify digital signatures. The URL Loading System is a high-level mechanism to load a remote resource securely over the network. It supports HTTP Basic Access Authentication, HTTP Digest Access Authentication, SSL/TLS server authentication, and SSL/TLS authentication of both server and client. CFNetwork is a C API which allows finer-grained control over the SSL/TLS stream properties than the URL Loading System. Data Protection ties the confidentiality of protected files and Keychain items to the availability of the user's passcode. The API provides protection classes for files and Keychain items that define when data is available to the application. A local attacker that gains access to a lost or stolen iOS device with enabled Data Protection can only access the data that is currently available, depending on the protection class of files and Keychain items and the current state of the device (locked, unlocked, restarted but before first unlock).
Developers can protect their application's data by using these APIs correctly. However, this statement only holds true as long as no jailbreak exists. On a jailbroken device, attackers are able to bypass the security mechanisms and, according to recent research, may even be able to obtain the unencrypted content of files and Keychain items that are protected by Data Protection by performing a brute-force attack on the user's passcode.

4. Offense: Attacking iOS Applications

This chapter discusses several attack vectors at the application level on non-jailbroken iOS devices. It presents several mechanisms that allow apps to perform inter-process communication (IPC), methods to get access to private information of the user, certificate validation and network communication weaknesses, and the security implications of using `UIWebView` in an application. The chapter closes with a security evaluation of some popular iOS apps from the App Store.

4.1. Interprocess Communication Between Applications

4.1.1. URL Schemes

URL schemes are a mechanism to launch other applications and provide information to them. When an application is installed, iOS registers the URL schemes listed under the key `CFBundleURLTypes` in the application's Info.plist. The entry contains two sub-entries: `CFBundleURLName` (the name) and `CFBundleURLSchemes` (the actual URL schemes). An application opens a URL with `-[UIApplication openURL:]`. Applications can check if the URL can be opened by another application by invoking `-[UIApplication canOpenURL:]`. This method does not determine which application will open the URL but rather if it can be opened at all. iOS provides a set of default URL schemes to open some of the pre-installed applications like Phone, Mail, Messages, Maps, YouTube, and iTunes. Listing 4-1 shows how an application can open another application by opening a URL.

```
// iOS registerd the URL scheme 'other-app' during the installation
// of the other application.
NSURL *applicationURL = [NSURL URLWithString:
    @"other-app://action?parameter1=value1&parameter2=value2"];
UIApplication *application = [UIApplication sharedApplication];
if ([application canOpenURL:applicationURL]) {
  [application openURL:applicationURL];
}
```

Listing 4-1: Launch another application by opening a URL

iOS launches the application that is registered to handle URLs with the given scheme and passes it the URL as an argument in the method `-[UIApplicationDelegate application:openURL:sourceApplication:annotation:]`. The application parses the URL and obtains the embedded information (see Listing 4-2).

```
- (BOOL)application:(UIApplication *)application
            openURL:(NSURL *)url
  sourceApplication:(NSString *)sourceApplication
         annotation:(id)annotation
{
  // Parse and validate the URL.
  // ...

  // Trigger an action based on the parsed contents of the URL.
  // ...

  // Return YES if the app could open the URL successfully,
  // otherwhise return NO.
  return YES;
}
```

Listing 4-2: `UIApplicationDelegate` method invoked during opening a URL

The URL passed to the second application can contain arbitrary information as long as it conforms to RFC 3986 [RFC3986]. Applications can use URL schemes as an IPC mechanism to transfer data to another application. The user recognizes the opening of the URL only indirectly because the other application launches and switches to the foreground. An application can invoke `–[UIApplication openURL:]` without asking the user. The user has no control over the information being passed to the other application.

PiOS [PIOS] could detect information leakage that results by opening a URL. It could taint the method `–[UIApplication openURL:]` as a sink and trace the flow of data to this sink.

4.1.2. Keychain Access Groups

Keychain access groups can also be used as a way to communicate data from one application to another. Consider the following scenario: application A and B belong to a shared Keychain access group. Application A stores a Keychain item with a shared access group in the Keychain. The item contains information intended for application B. Application B retrieves the item and obtains the information from it.

This approach of IPC only functions between applications from the same developer, because the configuration of a Keychain access group is only possible for apps of the same iOS Developer Program account.

PiOS [PIOS] could detect the information leakage that results by storing information in a Keychain item with a shared access group. It could taint the functions `SecItemAdd()` and `SecItemUpdate()` of Keychain Services as sinks and trace the flow of data to these sinks.

4.1.3. Pasteboards

`UIPasteboard` is a class of the UIKit framework which provides access to pasteboards for sharing data between parts of the same application or between applications. Applications can create their own pasteboards (see Listing 4-3). Another application can read the pasteboard contents if it knows the name of the pasteboard (see Listing 4-4).

```
UIPasteboard *sharedPasteboard =
  [UIPasteboard pasteboardWithName:@"SharedPasteboard" create:YES];
sharedPasteboard.persistent = YES;
sharedPasteboard.string = @"Hello, world";
```

Listing 4-3: Store a string in an application pasteboard

```
UIPasteboard *sharedPasteboard =
  [UIPasteboard pasteboardWithName:@"SharedPasteboard" create:NO];
NSString *receivedString = sharedPasteboard.string;
NSLog(@"Received the string '%@' from the other application.",
      receivedString);
```

Listing 4-4: Retrieve a string from an application pasteboard

4.1.4. Symbolic and Hard Links in the Filesystem

Every application is installed into its own, random home directory in the filesystem. Applications store files either in the Documents directory, the Library/Caches directory, or the tmp directory within the application's home directory [IOS_A_PROG_G]. The App Store Review Guidelines prohibit developers from reading or writing files outside the application's home directory: *"Apps that read or write data outside its designated container area will be rejected"* [ASRG]. However, that does not mean that an application cannot access these areas. Many directories are not accessible because the application does not have sufficient rights to access it. Some directories outside the application's designated home directory are accessible by all applications. These directories can be used as sources and destinations for symbolic and hard links. The iPhone Dev Wiki points out that an *"[..] AppStore app itself can also read/write to these folders, even without jailbreaking (in principle), but they must be accessed via a symlink. Doing so allows easy cross-app data sharing, but that may violate the AppStore rules."* [SEATBELT]

Symbolic Links

A symbolic link is a file that references a filename of another file. Accessing a symbolic link results in an access of the file the symbolic link references via its destination filename. The resolution of the symbolic link is automatically performed

by the operating system. The *Foundation* class `NSFileManager` provides the method `createSymbolicLinkAtPath:withDestinationPath:error:` to create a symbolic link. An application can use this method to create a symbolic link in a directory accessible to other applications that references a file in the application's home directory. A possible location for storing links is the directory `/private/var/mobile/Library/AddressBook/`. Applications can create files as well as read and write to the files in this directory (see Listing 4-5).

```
// Establish a symbolic link to the file. 'documentsDirectoryPath'
// is the path to a file in the Documents directory within the
// application's home directory.
NSString *symbolicLinkPath =
  @"/var/mobile/Library/AddressBook/TEST_SOFT_LINKED.txt";
NSFileManager *fileManager =
  [[NSFileManager alloc] init] autorelease];
NSError *symbolicLinkError = nil;
if ([fileManager createSymbolicLinkAtPath:symbolicLinkPath
                      withDestinationPath:documentsDirectoryPath
                                    error:&symbolicLinkError]) {
  NSLog(@"Created a symbolic link at path: %@", symbolicLinkPath);
}
```

Listing 4-5: Creating a symbolic link with `NSFileManager`

Other applications can then access the symbolic link (because it is located in a directory accessible to all applications) and obtain the destination of the link. This reveals the destination path to the file in the random home directory of the application that created the symbolic link (see Listing 4-6).

```
// Determine the symbolic link's destination.
NSString *symbolicLinkPath =
  @"/var/mobile/Library/AddressBook/TEST_SOFT_LINKED.txt";
NSError *symbolicLinkError = nil;
NSString *destinationPath =
  [fileManager destinationOfSymbolicLinkAtPath:symbolicLinkPath
                                         error:&symbolicLinkError];
if (!symbolicLinkError) {
  NSLog(@"Destintation of symbolic link is: \"%@\"",
        destinationPath);
} else {
  NSLog(@"%@", symbolicLinkError.localizedDescription);
}
```

Listing 4-6: Obtaining the destination path of the symbolic link

If an application tries to read the contents of the destination file (by reading from the symbolic link), the operation fails with the error message "The operation couldn't be completed. Operation not permitted" (see Listing 4-7).

```
// Try to obtain the contents via the symbolic link.
NSString *fileContents = [[[NSString alloc]
                           initWithContentsOfFile:symbolicLinkPath
                           encoding:NSUTF8StringEncoding
                           error:&symbolicLinkError]
                          autorelease];
if (!symbolicLinkError) {
  NSLog(@"Content of symbolically linked file: \"%@\"",
        fileContents);
} else {
  NSLog(@"%@", symbolicLinkError.localizedDescription);
}
```

Listing 4-7: Trying to read the contents of the file referenced by the symbolic link

Hard Links

A hard link is a direct reference to a file in the filesystem. Just as with symbolic links, a file can have many hard links that reference it. An application can accesses a single file through many different hard links. The `NSFileManager` method `linkItemAtPath:toPath:error:` creates a hard link to the file at the path provided as the first argument. An application can use this method to create a hard link in a directory accessible to other applications (e.g., `/private/var/mobile/Library/AddressBook/`) that references a file that is otherwise only referenced from within the application's home directory (see Listing 4-8). Other applications can access the contents of the file referenced by the hard link (see Listing 4-9).

```
// Hard-link the file.'documentsDirectoryPath'
// is the path to a file in the Documents directory within the
// application's home directory.
NSString *hardLinkPath =
  @"/var/mobile/Library/AddressBook/TEST_HARD_LINKED.txt";
NSError *hardLinkError = nil;
if ([fileManager linkItemAtPath:documentsDirectoryPath
                         toPath:hardLinkPath
                          error:&hardLinkError]) {
  NSLog(@"Created a hard link at path %@", hardLinkPath);
}
```

Listing 4-8: Creating a hard link with `NSFileManager`

```
// Read the hard-linked file.
NSString *hardLinkPath =
  @"/var/mobile/Library/AddressBook/TEST_HARD_LINKED.txt";
NSError *hardLinkError = nil;
NSString *hardLinkedContents = [[[NSString alloc]
                                  initWithContentsOfFile:hardLinkPath
                                  encoding:NSUTF8StringEncoding
                                  error:&hardLinkError]
                                 autorelease];
if (!hardLinkError) {
  NSLog(@"Content of hard-linked file: %@", hardLinkedContents);
} else {
  NSLog(@"%@", hardLinkError.localizedDescription);
}
```

Listing 4-9: Reading the contents of the file referenced by the hard link

Applications can use symbolic links tell other applications the random path to their home directory in the filesystem. They can use hard links to give other applications access to files within their home directory, which is normally prevented by the sandboxing mechanism.

4.1.5. UNIX Domain Sockets

Applications can use UNIX domain sockets to perform interprocess communication. The only limitation is that fact that the socket has to be created in a directory that is accessible by both communicating applications, such as `/private/var/mobile/Library/AddressBook/`. Listing 4-10 shows how an application creates a UNIX domain socket. In this example the application responds with the string "`This is a message from the other program.`".

```
#define SOCKET_PATH "/var/mobile/Library/AddressBook/SOCKET"

int serverSocket = socket(AF_UNIX, SOCK_STREAM, 0);
if (serverSocket < 0) {
  perror("socket");
  exit(EXIT_FAILURE);
}

// Ensure that SOCKET_PATH does not exist
unlink(SOCKET_PATH);

struct sockaddr_un address;
address.sun_family = AF_UNIX;
strcpy(address.sun_path, SOCKET_PATH);
socklen_t addressLength = SUN_LEN(&address);

if (bind(serverSocket,
         (struct sockaddr *)&address,
         addressLength) != 0) {

  perror("bind");
  exit(EXIT_FAILURE);
}

if (listen(serverSocket, 5) != 0) {
  perror("listen");
  exit(EXIT_FAILURE);
}

int connection = 0;
while ((connection = accept(serverSocket,
                            (struct sockaddr *)&address,
                            &addressLength)) >= 0) {

  NSString *message = @"This is a message from the other program.";
  NSMutableData *messageData =
    [[[message dataUsingEncoding:NSUTF8StringEncoding] mutableCopy]
     autorelease];
  write(connection,
        [messageData mutableBytes],
        [messageData length]);
  close(connection);
}
close(serverSocket);

// Remove socket at SOCKET_PATH
unlink(SOCKET_PATH);
```

Listing 4-10: Creating and opening a UNIX domain socket

In iOS, only one application runs in the foreground. The other applications run in the background or are not running at all. Background processing in third-party applications is very limited in iOS. By default, iOS suspend the application when it transitions to the background. However, the application can use one of the multitasking services of iOS to either schedule a long-running task, continue audio playback, receive location updates, or retain an active VoIP call.

When two applications perform interprocess communication via a UNIX domain socket, at least one of them runs in the background. For example, the application that creates and opens the socket can begin a long-running task by invoking the message `-[UIApplication beginBackgroundTaskWithExpirationHandler:]`. iOS continues executing the block provided as the parameter when the application transitions to the background on the event that another applications transitions to the foreground (see Listing 4-11).

```
// Begin async exection in background. 'socketTaskID' (of type
// UIBackgroundTaskIdentifier) is an instance variable of the
// current class.
UIApplication *sharedApp = [UIApplication sharedApplication];
socketTaskID = [sharedApp
                beginBackgroundTaskWithExpirationHandler:^(void) {

  [sharedApp endBackgroundTask:socketTaskID];
   socketTaskID = UIBackgroundTaskInvalid;
}];

dispatch_async(
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

  // Listen for incoming connections -> See Listing 4-10
  // ...

  [sharedApp endBackgroundTask:socketTaskID];
  socketTaskID = UIBackgroundTaskInvalid;
});
```

Listing 4-11: Begin a long running task to perform socket IPC in background

Another application opens the socket and receives the string sent by the first application that runs in the background. Listing 4-12 shows how the second application receives the string from the first application via the socket.

```
#define SOCKET_PATH "/var/mobile/Library/AddressBook/SOCKET"

int clientSocket = socket(AF_UNIX, SOCK_STREAM, 0);
if (clientSocket < 0) {
  perror("socket");
  exit(EXIT_FAILURE);
}

struct sockaddr_un address;
address.sun_family = AF_UNIX;
strcpy(address.sun_path, SOCKET_PATH);
socklen_t addressLength = SUN_LEN(&address);

if (connect(clientSocket,
            (struct sockaddr *)&address,
            addressLength) != 0 ) {

  perror("connect");
  exit(EXIT_FAILURE);
}

// Receive the string.
NSMutableData *receivedData = [[[NSMutableData alloc]
                                initWithLength:100]
                               autorelease];
read(clientSocket,
     [receivedData mutableBytes],
     [receivedData length]);
NSString *receivedMessage = [[[[NSString alloc]
                               initWithData:receivedData
                               encoding:NSUTF8StringEncoding]
                              autorelease];
NSLog(@"Received: %@", receivedMessage);
[receivedData release];

// Close the connection.
close(clientSocket);
```

Listing 4-12: Open the socket and receive the string from the first application

4.2. Privacy Leaks

In his talk *iPhone Privacy* at BlackHat DC 2010, Nicolas Seriot showed that a third-party developer can access personal information by only using public APIs that are available to every iOS developer and without exploiting the device or requiring the device to be jailbroken [IPHONE_PIRACY]. He developed the application SpyPhone [SPYPHONE] to demonstrate what he discovered. An application that uses the techniques presented by SpyPhone could potentially pass the App Store review process because they only use benign public APIs to harvest the following personal data:

- Phone number of the device, Unique Device Identifier, SIM card serial number, International Mobile Subscriber Identity (IMSI)
- Recent MobileSafari and YouTube searches
- Recent calls
- E-mail account data (full name of user, e-mail address, host, login)
- Contact information from the Address Book
- Contents of the keyboard cache
- Photos and the locations they are taken
- GPS and Wi-Fi locations
- Cities configured in the Weather application

This chapter discusses additional privacy issues in iOS.

4.2.1. Voice Chat Services

The Game Kit framework introduced in iOS 3.0 allows developers to incorporate voice chat functionality in their applications [GK_PROG_G]. The framework automatically adjusts the microphone and mixes audio playback. It uses a previously established network connection to another iOS device to establish its own direct real-time connection. The Game Kit Programming Guide mentions two kinds of discovery an application can use to establish the voice channel: peer-to-peer-based discovery and server-based discovery [GK_PROG_G]. Peer-to-peer-based discovery uses an already existing peer-to-peer connection established with the class `GKSession` (Bluetooth or local Wi-Fi). The server-based discovery uses a server that distributes the participant IDs that are needed by the devices to establish the voice channel. When the framework's attempt to establish the voice channel fails, the application is given a chance to send the data by itself. Voice Chat Services require a Wi-Fi connection.

When an application establishes a voice chat connection with another device, it can do so without notifying the user. This can be abused by an evil third-party developer to intercept the ambient sounds of a device while the user uses the application. The voice communication channel is only active as long as the application runs in the foreground. Game Kit does not automatically re-establish the connection when the application transitions from the background to the foreground.

The sample application AmbientSounds (available on the enclosed disc) demonstrates how an application creates an undetected voice channel between the device and a peer from the local Wi-Fi network or with a Bluetooth enabled device that also runs the application. AmbientSounds is explained in greater detail in the chapter "Experimental Results".

4.2.2. NSUserDefaults

The user defaults is a mechanism to persistently store instances of `NSArray`, `NSDictionary`, `NSString`, `NSDate`, `NSData`, and `NSNumber`. It is a convenient way to store application preferences. When an application uses the pre-installed Settings application to configure its preferences, it obtains these preferences through `NSUserDefaults`. Listing 4-13 shows how an instance of `NSString` is stored in the user defaults and then obtained.

```
// Store the author's name
NSString *authorName = @"Manuel Binna";
[[NSUserDefaults standardUserDefaults] setObject:authorName
                                          forKey:@"AuthorOfThesis"];

// Retrieve the author's name
NSString *myName = [[NSUserDefaults standardUserDefaults]
                    objectForKey:@"AuthorOfThesis"];
```

Listing 4-13: Storing and retrieving a string from the user defaults

The user defaults are stored in the binary property list file `$BUNDLE_IDENTIFIER.plist` in the folder `Library/Preferences/` of the application's home directory. `$BUNDLE_IDENTIFIER` is the value of the entry `CFBundleIdentifier` in the application's `Info.plist` file. Every application uses its own distinct set of user defaults which are stored in the application's home directory. Although other third-party applications cannot accesses the home directory of other applications, the values are not encrypted before being written to the file and the file itself is not protected by Data Protection (the extended attribute `NSFileProtectionKey` of the file has the value `NSFileProtectionNone`). Credentials stored in the user defaults are therefore vulnerable to disclosure. Developers should use the Keychain instead of the user defaults to store sensitive information. The Keychain provides a secure storage for, e.g., passwords, certificates, private keys, and authentication tokens. Items in the Keychain are automatically encrypted and protected by the Data Protection architecture.

4.2.3. UIPasteboard

The *Cut, Copy, and Paste* functionality of iOS is based on the general pasteboard returned by the method `+[UIPasteboard generalPasteboard]`. The general pasteboard is a system pasteboard that is shared between all applications. System pasteboards are always persistent. The contents of these pasteboards is stored in the filesystem and endures device reboots. The binary property list file `/private/var/mobile/Library/Caches/com.apple.UIKit.pboard/pasteboardDB` stores the contents of persistent pasteboards. Standard application pasteboards are non-persistent by default, but they can also be configured to be persistent. Values added

to persistent pasteboards are usually not directly stored in the file mentioned above. iOS caches the values and writes them to the file before the system reboots.

When the user copies several texts from different applications, the general pasteboard only stores the latest item added to it. A value stored in the general pasteboard can be read by all other applications in the system by using the public method `-[UIPasteboard items]`. Accessing the file `pasteboardDB` directly or with a symlink or hard link is not permitted on non-jailbroken devices. Access is permitted on jailbroken devices, but does not yield more items than using the public API.

Listing 4-14 shows a sample `pasteboardDB` file extracted from a jailbroken device and converted with the command `plutil -convert xml1 pasteboardDB` to the XML property list file format. The base64-decoded value of the key `NeXT smart paste pasteboard type` is the string "NeXT smart paste pasteboard type".

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <integer>1</integer>
  <dict>
    <key>bundle</key>
    <string>com.apple.UIKit.pboard</string>
    <key>items</key>
    <array>
      <dict>
        <key>NeXT smart paste pasteboard type</key>
        <data>
        TmVYVCBzbWFydCBwYXN0ZSBwYXN0ZWJvYXJkIHR5cGU=
        </data>
        <key>public.utf8-plain-text</key>
        <data>
        TWFudWVs
        </data>
      </dict>
    </array>
    <key>name</key>
    <string>com.apple.UIKit.pboard.general</string>
    <key>persistent</key>
    <true/>
  </dict>
</array>
</plist
```

Listing 4-14: Sample `pasteboardDB` file

4.2.4. Screenshot during App-Enters-Background Transition

When an application changes its state from the foreground to the background, iOS creates a screenshot of the application's window and animates the transition to the

background [DHANJANI_BHEU2011]. It stores the screenshot in the folder `Library/Caches/Snapshots/$BUNDLE_IDENTIFIER/` in the application's home directory where `$BUNDLE_IDENTIFIER` is the value of the `CFBundleIdentifier` key in the file Info.plist of the application's main bundle. During an application's transition from the foreground to the background or during its termination, iOS removes the directory `Library/Caches/Snapshots/$BUNDLE_IDENTIFIER/` together with its contents and creates a new directory with the same name. The new directory contains the new screenshot which remains in the directory, even after the transition completed. The filename depends on the interface orientation of the device when the screenshot was made and whether the device has a Retina display or not.

The possible filenames of the screenshot on devices without a Retina display are:
- `UIApplicationAutomaticSnapshotDefault-Portrait.jpg`
- `UIApplicationAutomaticSnapshotDefault-LandscapeLeft.jpg`
- `UIApplicationAutomaticSnapshotDefault-LandscapeRight.jpg`
- `UIApplicationAutomaticSnapshotDefault-PortraitUpsideDown.jpg`

The possible filenames of the screenshot on devices with Retina display are:
- `UIApplicationAutomaticSnapshotDefault-Portrait@2x.jpg`
- `UIApplicationAutomaticSnapshotDefault-LandscapeLeft@2x.jpg`
- `UIApplicationAutomaticSnapshotDefault-LandscapeRight@2x.jpg`
- `UIApplicationAutomaticSnapshotDefault-PortraitUpsideDown@2x.jpg`

iOS stores the screenshot of pre-installed applications in the directory `/private/var/mobile/Library/Caches/Snapshots/$BUNDLE_IDENTIFIER/` where `$BUNDLE_IDENTIFIER` is the value of the `CFBundleIdentifier` key in the file Info.plist of the pre-installed application's main bundle. Only devices that support Multitasking store screenshots of third-party applications. However, all devices store screenshots of built-in applications like Mobile Mail, Mobile Safari, and Maps. The screenshot is not removed after the transition completed but only when a new screenshot is created.

If an application displays sensitive information when it begins transitioning from the foreground to the background, the newly created screenshot also contains the sensitive information. For example, viewing a received e-mail and then switching to another application creates a screenshot in the filesystem that contains the contents of the e-mail. On devices that support Data Protection, the screenshot is protected with `NSFileProtectionComplete`. Although the screenshots exist in the filesystem, they are protected with the user's passcode if the device is locked. On devices that do not support Data Protection, the screenshot is easily accessible when an attacker has physical access to the device.

Developers can prevent sensitive information from appearing in the screenshot. An application observes the `UIApplicationDidEnterBackgroundNotification` and `UIApplicationWillTerminateNotification` notifications which are posted just before the application transitions to the background and iOS creates the screenshot.

The handler for the observed notification can then remove the sensitive information from the displayed view and therefore prevent their appearance in the screenshot. The application should also observe the notification `UIApplicationWillEnterForegroundNotification` which is posted before the application is active and displays its contents. The handler for the observed notification adds the sensitive information to the view that is to be displayed.

Listing 4-15 shows how a `UIViewController` instance hides its view when it observes a `UIApplicationDidEnterBackgroundNotification` or a `UIApplicationWillTerminateNotification` notification and un-hides its view when it observes a `UIApplicationWillEnterForegroundNotification` notification.

```
[[NSNotificationCenter defaultCenter]
  addObserverForName:UIApplicationDidEnterBackgroundNotification
              object:nil
               queue:[NSOperationQueue mainQueue]
          usingBlock:^(NSNotification *notification) {

  view.hidden = YES;  // view is an ivar of UIViewController
}];

[[NSNotificationCenter defaultCenter]
  addObserverForName:UIApplicationWillTerminateNotification
              object:nil
               queue:[NSOperationQueue mainQueue]
          usingBlock:^(NSNotification *notification) {

  view.hidden = YES;  // view is an ivar of UIViewController
}];

[[NSNotificationCenter defaultCenter]
  addObserverForName:UIApplicationWillEnterForegroundNotification
              object:nil
               queue:[NSOperationQueue mainQueue]
          usingBlock:^(NSNotification *notification) {

  view.hidden = NO;  // view is an ivar of UIViewController
}];
```

Listing 4-15: Prevent sensitive information from appearing in the screenshot

4.3. Network and PKI Attacks

4.3.1. Shortcoming of Certificate Detail View

When MobileSafari attempts to establish an SSL/TLS connection and cannot successfully verify the server identity, it prompts the user to cancel or continue with the establishment of the secure connection (see Figure 4-1). Before the user chooses to continue, she may optionally review the details of the server certificate. However, the view that presents the certificate details to the user truncates the values and does not present them in full length, especially in portrait orientation (see Figure 4-2). If the user browses to the URL in question in portrait mode, the certificate detail view is presented in portrait mode. The view does not automatically rotate in order to reveal more characters of the certificate details that fit into one line of the view. The device presents the certificate details in landscape mode only if the user browses to the URL in landscape mode. When presented in landscape mode the certificate details view can display more characters of the certificate values before truncating them (see Figure 4-3).

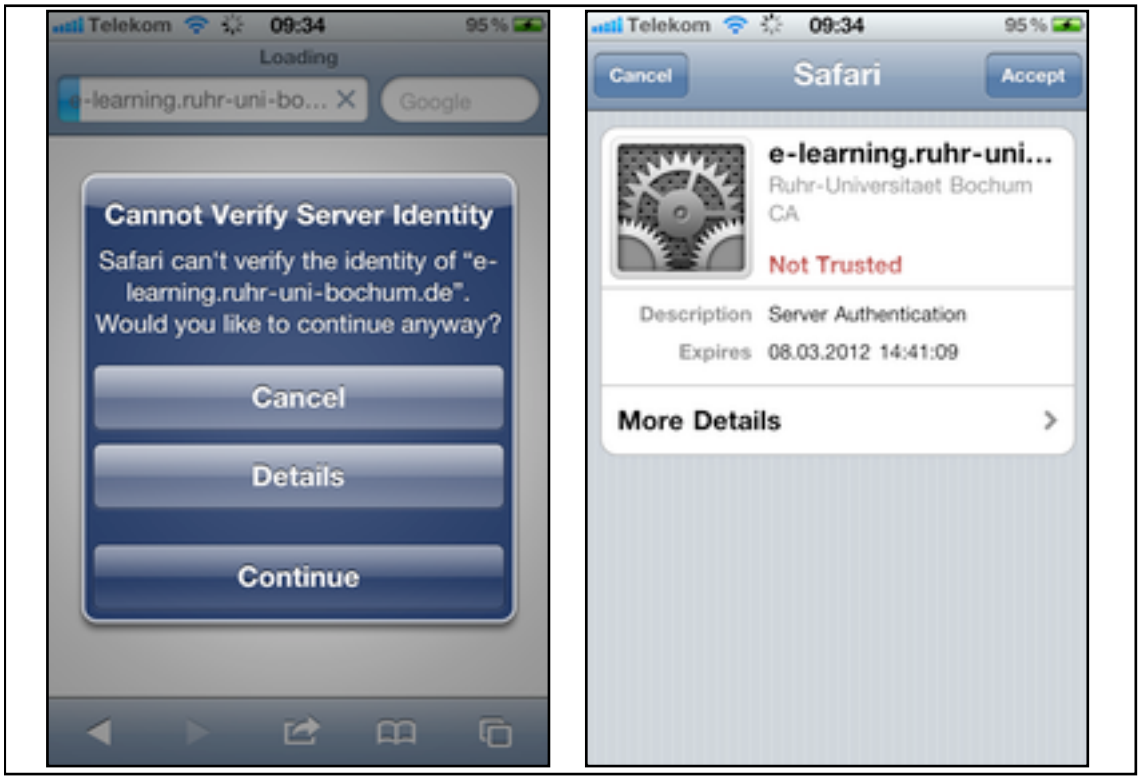

Figure 4-1: Dialogs leading to certificate detail view (portrait)

Listing 4-2: Certificate detail view (portrait)

Listing 4-3: Certificate detail view (landscape)

The shortcoming of the view to present truncated certificate details to the user could be used by an adversary to create a certificate for a common name that begins with the common name of the certificate she wants to forge. However, the forged certificate's common name ends with a domain the attacker owns. The attacker then performs a man-in-the-middle attack and sends the victim the forged certificate. The truncation of the certificate details on the victim's iOS device could mislead her in believing that she connects to the correct host.

4.3.2. Certificates with Null Prefix

In his paper *"Null Prefix Attacks Against SSL/TLS Certificates"*, Moxie Marlinspike describes an attack that uses the zero byte character `'\0'` in the common name of a certificate [NULL_PREFIX]. The certificate authority (CA) issues a certificate with a null prefix common name, e.g., `apple.com\0evil.com`, to the owner of the domain evil.com. Some applications only use the characters up to the termination character

'\0' during the validation of the certificate. When the attacker performs a man-in-the-middle attack, the certificate validation mechanism on the victim's computer may potentially use only the substring "apple.com" from the certificate's common name. This can be used to spoof a certificate for a domain which the attacker does not own.

Figure 4-4 shows a wildcard certificate with a null character between the domain * and the domain thoughtcrime.noisebridge.net. This certificate can be effectively used for any URL with the domain which the victim requests. This certificate was posted by Jacob Applebaum on the Noisebridge-discuss mailing list [NOISEBRIDGE_DISCUSS]. He describes that it is a *"fully valid, signed certificate"*. However, the certificate of the issuer, ipsCA CLASEA1 Certification Authority, is not contained in the list of root certificates in iOS 4 [IOS4_ROOT_CERTS]. Therefore, the certificate validation with SecTrustEvaluate(), as discussed in the chapter on Certificate, Key, and Trust Services, fails.

Figure 4-4: A wildcard certificate with null prefix

The function SecCertificateCopySubjectSummary() of the Certificate, Key, and Trust Services API returns the common name of the certificate. The function is used for populating a dialog during certificate validation if user interaction is necessary to confirm the use of a certificate that cannot be validated automatically. When presented a certificate with with null character in the common name, SecCertificateCopySubjectSummary() returns a string that only contains the characters up to the null character (see Listing 4-16).

```
// 'certificate' has the common name apple.com\0evil.com
SecCertificateRef certificate = ...;
NSString *commonName =
  (NSString *) SecCertificateCopySubjectSummary(certificate);

// 'commonName' has the value @"apple.com"
```

Listing 4-16: SecCertificateCopySubjectSummary() with a null prefix certificate

An attacker can use this issue to create a certificate with two domains separated by a null character in the common name. Although the verifying mechanism might inform the user that the certificate cannot be validated successfully and prompt if it should continue or cancel. If the subject summary contains exactly the domain of the

requested URL, the user might be mislead into thinking that the certificate is a valid certificate for this domain.

4.4. Security Evaluation of Particular Applications

This chapter provides a security evaluation of popular iOS applications. A jailbroken iPod touch (4. generation) with iOS 4.3.2 was used to obtain some information not accessible on non-jailbroken iOS devices. The following aspects are evaluated:

Is the application able to open URLs with a certain URL scheme?

Every installed application contains a file with the name Info.plist which stores general information about the application. If the application is able to open URLs with a certain URL scheme, the Info.plist file must list the URL scheme. Using SSH to log in to the jailbroken iOS device, the Info.plist files of the evaluated application is searched for registered URL schemes.

Does the application provide URL handlers that another application can invoke?

The command-line utility `strings` was used to find strings that indicate URL handlers in an iOS application binary file. The Mach-O binary of an iOS application is encrypted by default. The loader in the operating system decrypts the binary during the launch. The GNU Debugger (GDB) on the jailbroken devices was used to launch the evaluated application and dump the decrypted image to disk [DECRYPT_APP]. The steps necessary to dump the decrypted binary are:

- Use the command-line utility `otool` with the parameter `-l` to output the load commands. The output contains information about the load command responsible for the decryption.

```
$ otool -l application
Load command 10
          cmd LC_ENCRYPTION_INFO
      cmdsize 20
 cryptoff  4096
 cryptsize 270336
 cryptid   1
```

  The value `cryptoff` determines the offset of the encrypted part of the application binary, `cryptsize` determines its size. `cryptid` value 1 determines that the application is encrypted. 0 determines that the application is not encrypted.
- Launch the application with GDB:

```
$ gdb application
```

- Obtain the program's entry point using the command:

```
(gdb) info files
```

- Set a breakpoint at the program's entry point. When this breakpoint is triggered, the application is decrypted in the main memory.

```
(gdb) break *0xENTRYPOINT
```

- Launch the application:

```
(gdb) r
```

- Dump the memory region from the offset of the text segment in the program's virtual address space (`0x2000`) to the address at (`0x2000+cryptsize`).

```
(gdb) dump binary memory /path/to/file 0x2000 (0x2000+cryptsize)
```

- The dumped memory region is used as the input for `strings` to obtain knowledge about possible URL handlers in the application.

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

The class `NSUserDefaults` stores application data in a property list file within the directory `Library/Preferences` in the application's home directory. If an application stores any data with NSUserDefaults, it is stored in this file. Using SSH to log in to the jailbroken iOS device, the contents of the file was inspected by first using the command-line utility `plutil` with the parameter `–convert xml1` to convert the binary property list file to an XML property list file and then using `cat` to print the contents of the file to the command line.

Does the application protect its files with `NSFileProtectionComplete` or `NSDataWritingFileProtectionComplete`?

The open source application fswalker [FSWALKER] provides a way to navigate the filesystem hierarchy of the iOS device. The application was modified to display the extended attribute `NSFileProtectionKey` of a selected file and therefore check if the file is protected by the Data Protection architecture [FSWALKER_MBINNA]. This evaluation determines if the files in the directories `Documents`, `Library/Caches`, and `tmp` within the application's home directory are protected or not.

Does the application use SSL/TLS to secure network connections?

A MacBook Pro with Mac OS X Snow Leopard 10.6.7 was used as the access point for the jailbroken iPod touch (see Figure 4-5). The HTTP proxy *mitmproxy* [MITMPROXY], which acts as a man-in-the-middle between the iOS device and remote servers, allowed to inspect the requests and responses.

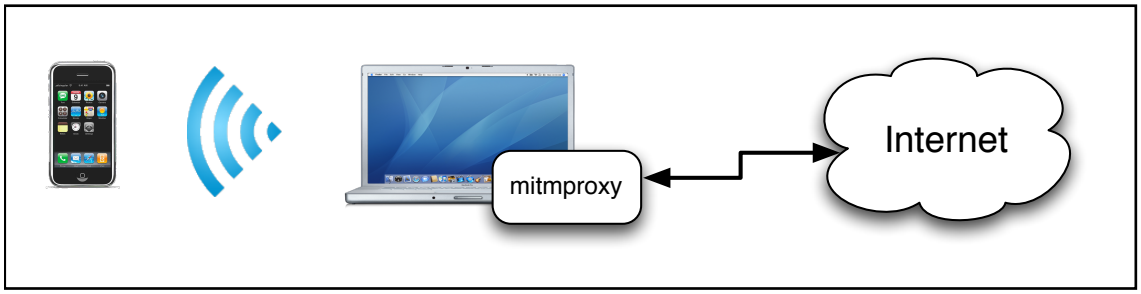

Figure 4-5: Setup for evaluating network connections

4.4.1. Blackboard Mobile Learn v2.1

Is the application able to open URLs with a certain URL scheme?

The application does not register any custom URL schemes.

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

No sensitive information in `NSUserDefaults` is found.

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

Blackboard Mobile Learn does not protect its files. All inspected files have the extended attribute `NSFileProtectionNone`.

Does the application use SSL/TLS to secure network connections?

The application uses HTTP over SSL/TLS for some requests, but it also uses plain HTTP for some requests. The request "directUserLogin" is a plain (insecure) HTTP GET request that transmits, among others, the username, password, and UDID.

4.4.2. iOutbank v2.8.9.6677

Is the application able to open URLs with a certain URL scheme?

The application registers the URL schemes `bank`, `ioutbank`, and `outbank` (see Listing 4-17).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>de.stoegerit.iOutBank</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>bank</string>
      <string>ioutbank</string>
      <string>outbank</string>
    </array>
  </dict>
</array>
...
```

Listing 4-17: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening either `bank://`, `ioutbank://`, or `outbank://`, the application provides the following URL handlers:
- `bank://paymentcollection`

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

No sensitive information in `NSUserDefaults` is found.

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

iOutbank protects the following files with Data Protection:
- `Documents/importexport/backup/*`
- `Documents/outbankdbv2.sql`

The other files are not protected by Data Protection.

Does the application use SSL/TLS to secure network connections?

The application uses HTTP over SSL/TLS to securely transmit data over the network.

4.4.3. Facebook v3420

Is the application able to open URLs with a certain URL scheme?

The application registers the URL schemes `fbauth`, `fbauth2`, and `fb` (see Listing 4-18).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.facebook</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>fbauth</string>
      <string>fbauth2</string>
      <string>fb</string>
    </array>
  </dict>
</array>
...
```

Listing 4-18: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening either `fbauth://`, `fbauth2://`, or `fb://`, the application provides the following URL handlers:

- `fbauth://authorize/dummy_path?`
- `fbauth://authorize/(initWithURL:params:)`
- `fbauth2://authorize/(initWithURL:params:)`
- `fb://mailbox`
- `fb://friends`
- `fb://places`
- `fb://launcher`
- `fb://root`
- `fb://feed`
- `fb://profile`
- `fb://pages`
- `fb://messaging`
- `fb://online`
- `fb://requests`
- `fb://events`
- `fb://birthdays`
- `fb://photosapp`
- `fb://albums`
- `fb://notes`
- `fb://groups`
- `fb://contactimporter`
- `fb://contactimporter/modal`
- `fb://contactimporter/legalese`
- `fb://findfriends`
- `fb://findfriends/modal`
- `fb://findfriends/legalese`
- `fb://notifications`
- `fb://feed/(initWithFilterKey:)`
- `fb://feedfilters`
- `fb://post/(initWithPostId:)`
- `fb://post/(initWithPostId:)/tagged`
- `fb://post/(untagSelfFromPostWithId:)/untagSelf`
- `fb://profile/(initWithUID:)`
- `fb://profile/(initWithWallUID:)/wall`
- `fb://profile/(initWithInfoUID:)/info`
- `fb://profile/(initWithPhotosUID:)/photos`
- `fb://profile/(initWithMutualFriendsUID:)/mutualfriends`
- `fb://profile/(initWithFansUID:)/fans`
- `fb://profile/(initWithFriendsUID:)/friends`
- `fb://profile/(initWithUID:)/fanpages`
- `fb://mailbox/(initWithFolder:)`
- `fb://messaging/(initWithFolder:)`
- `fb://messaging/(initWithFolder:)/(tid:)/participants`
- `fb://messaging/original_message?mid=(initWithMessageId:)`
- `fb://mailbox/(initWithFolder:)/(tid:)`
- `fb://messaging/(initWithFolder:)/thread?tid=(tid:)`
- `fb://chat/(initWithUID:)`
- `fb://photo/(initWithUID:)/(aid:)/(pid:)`
- `fb://photo/(initWithUID:)/(aid:)/(pid:)/feedback`

- fb://photo/(initWithProfilePicturesUID:)/profilepic
- fb://album/(initWithAID:)
- fb://album/(initWithAID:)/cover
- fb://event/(initWithEventId:)
- fb://event/(initWithEventId:)/rsvp
- fb://event/(initWithEventId:)/members/(rsvpStatus:)
- fb://group/(initWithGroupId:)/members
- fb://note/(initWithNoteId:)
- fb://online#online
- fb://online#offline
- fb://birthdays/(initWithMonth:)/(year:)
- fb://userset
- fb://nearby
- fb://place/(initWithPageId:)
- fb://place/addfriends
- fb://places/(initWithCheckinAtPlace:)/(byUser:)
- fb://places/legalese/tagged/(initWithTaggedAtPlace:)/(byUser:)
- fb://publish
- fb://publish/profile/(initWithUID:)
- fb://publish/post/(initWithPostId:)
- fb://publish/photo/(initWithUID:)/(aid:)/(pid:)
- fb://publish/mailbox/(initWithFolder:)/(tid:)
- fb://place/create
- fb://map
- fb://upload
- fb://upload/checkin/(showUploadMenuWithCheckinID:)
- fb://upload/profile/(showUploadMenuWithUID:)
- fb://upload/album/(showUploadMenuWithUID:)/(aid:)
- fb://upload/actions
- fb://upload/actions/newalbum
- fb://upload/actions/profile/(initWithUID:)
- fb://upload/actions/album/(initWithUID:)/(aid:)
- fb://upload/actions/checkin/(initWithCheckinId:)
- fb://upload/actions/resume
- fb://upload/(initWithSource:)/profile/(uid:)
- fb://upload/(initWithSource:)/album/(uid:)/(aid:)
- fb://upload/(initWithSource:)/checkin/(checkinId:)
- fb://upload/resume
- fb://upload/discard
- fb://mailbox/compose/(initWithUID:)
- fb://messaging/compose/(initWithUID:)
- fb://mailbox/compose
- fb://messaging/compose
- fb://note/compose
- fb://note/(initWithNoteId:)/edit
- fb://album/new
- fb://friends/picker
- fb://friends/sync
- fb://friends/sync/legalese
- fb://friends/sync/(removeData:)
- fb://profile/(initWithUID:)/poke
- fb://profile/(initWithUID:)/addfriend
- fb://profile/(initWithUID:)/removefriend
- fb://profile/(addFan:)/addfan
- fb://profile/(toggleFavorite:)/favorite

- fb://video/(playVideoWithId:)
- fb://profile/(initWithFBID:)/menu
- fb://faceweb/(initWithURL:)
- fb://facewebmodal/(initWithURL:)
- fb://profile/(fbid)
- fb://photo/(album.user.fbid)/(album.aid)/(pid)
- fb://album/(aid)
- fb://chat/(user.fbid)
- fb://feed/(filter.filterKey)
- fb://post/(postId)
- fb://mailbox/(folder)
- fb://messaging/(folder)
- fb://mailbox/(mailbox.folder)/(tid)
- fb://messaging/(mailbox.folder)/thread?tid=(urlEscapedTid)
- fb://event/(fbid)
- fb://note/(noteId)
- fb://video/(videoId)
- fb://place/(targetId)
- fb://place/addphoto
- fb://profile/(fbid)/wall
- fb://profile/(fbid)/info
- fb://profile/(fbid)/photos
- fb://profile/(fbid)/menu
- fb://chat/(fbid)
- fb://profile/(fbid)/poke
- fb://profile/(fbid)/addfriend
- fb://profile/(fbid)/removefriend
- fb://profile/(fbid)/addfan
- fb://profile/(fbid)/favorite
- fb://mailbox/compose/(fbid)
- fb://messaging/compose/(fbid)
- fb://messaging/original_message?mid=(commentId)
- fb://profile/(fbid)/fanpages
- fb://photo/(fbid)/profilepic
- fb://photo/(album.user.fbid)/(album.aid)/(pid)/feedback
- fb://note/(noteId)/edit
- fb://event/(fbid)/rsvp
- fb://publish/profile/(owner.fbid)
- fb://publish/profile/(fbid)
- fb://publish/post/(postId)
- fb://post/(postId)/tagged
- fb://post/(postId)/untagSelf
- fb://publish/photo/(album.user.fbid)/(album.aid)/(pid)
- fb://publish/mailbox/(mailbox.folder)/(tid)
- fb://album/(aid)/cover
- fb://birthdays/(month)/(year)
- fb://profile/(fbid)/friends
- fb://profile/(fbid)/mutualfriends
- fb://profile/(fbid)/fans
- fb://group/(fbid)/members
- fb://profile/(profile.fbid)/friends
- fb://profile/(profile.fbid)/mutualfriends
- fb://profile/(profile.fbid)/fans
- fb://event/(fbid)/members/attending
- fb://event/(fbid)/members/unsure

- fb://event/(fbid)/members/declined
- fb://event/(fbid)/members/not_replied
- fb://event/(event.fbid)/members/(rsvpStatus)
- fb://upload/profile/(owner.fbid)
- fb://upload/profile/(fbid)
- fb://upload/album/(user.fbid)/(aid)
- fb://upload/checkin/(checkinId)
- fb://upload/actions/profile/(fbid)
- fb://upload/actions/album/(user.fbid)/(aid)
- fb://upload/actions/checkin/(checkinId)/
- fb://messaging/(mailbox.folder)/(urlEscapedTid)/participants
- fb://post/%@_%@
- fb://profile/%@
- fb://photo/%@/0/%@
- fb://album/%@
- fb://event/%@
- fb://events/
- fb://video/%@
- fb://note/%@
- fb://place/%@
- fb://places/legalese/tagged/%lld/%lld
- fb://upload/%@/album/%lld/%@
- fb://upload/%@/profile/%lld
- fb://upload/%@/checkin/%lld
- fb://feed/%@
- fb://profile/0
- fb://profile/%lld
- fb://profile/
- fb://friends/sync/disconnect
- fb://post/%@
- fb://event/%llu
- fb://places/%lld/%lld
- fb://contactimporter/invites

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

The values for the keys `FBSessionAccessToken`, `FBSessionKey`, and `FBSessionSecret` are especially interesting (see Listing 4-19). A more in-depth analysis of these values is necessary to determine if they are sensitive information and should therefore not be stored in `NSUserDefaults`.

```
...
<key>FBSessionAccessToken</key>
<string>6628568379|65616214b4ce98b0ed41b07d.0-1813115193|
A1ACfaS_MOGBK9AIZaNDJ8ru8q4</string>
<key>FBSessionKey</key>
<string>65616214b4ce98b0ed41b07d.0-1813115193</string>
<key>FBSessionSecret</key>
<string>2ed962fe68fa65a07d76026af60c28d4</string>
...
```

Listing 4-19: `NSUserDefaults` file `com.facebook.Facebook.plist`

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

Facebook does not protect its files with Data Protection. All inspected files have the extended attribute `NSFileProtectionNone`.

Does the application use SSL/TLS to secure network connections? Does the app validate the certificate during the establishment of a SSL/TLS connection?

The application performs the login via HTTPS, but uses plain HTTP to transmit everything else.

4.4.4. Skype v3.0.1

Is the application able to open URLs with a certain URL scheme?

The application registers the URL scheme `skype` (see Listing 4-20).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.skype.skype</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>skype</string>
    </array>
  </dict>
</array>
...
```

Listing 4-20: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening `skype://`, the application provides the following URL handlers [DHANJANI_BHEU2011], where NUMBER is the Skype username or phone number of the recipient:

- `skype://NUMBER?call`
- `skype://NUMBER?chat`

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

Skype stores the username of the last logged in user under the key `SkypePrefsLastLoggedInSkypeName` (see Listing 4-21). The password is not stored in `NSUserDefaults`.

```
...
<key>SkypePrefsLastLoggedInSkypeName</key>
<string>XXXXX</string>
...
```

Listing 4-21: NSUserDefaults file com.skype.skype.plist

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

Skype does not protect its files with Data Protection. All inspected files have the extended attribute NSFileProtectionNone.

Does the application use SSL/TLS to secure network connections?

Skype uses plain HTTP to communicate with http://data.flurry.com. Flurry is a service that provides analytics and advertising for mobile applications. The actual application data is transmitted over Skype's own proprietary protocol.

4.4.5. VZ-Netzwerke v2.8.1

Is the application able to open URLs with a certain URL scheme?

The application does not register any custom URL schemes.

Does the application store sensitive information such as, e.g., user credentials in NSUserDefaults instead of in the *Keychain*?

Interesting values are the values of the keys deviceToken and userIds (see Listing 4-22). A more in-depth analysis of these values is necessary to determine if they are sensitive information and should therefore not be stored in NSUserDefaults.

```
...
<key>deviceToken</key>
<string>a1eed9b19eca69e335192d58b4f9f58267c2d98aa9bc7cf1381c291c407a
05d3</string>
...
<key>userIds</key>
<string>hPHScfCpFfAu639GGOIE-SQOSXmfpuKGgEYf5hSAl90</string>
...
```

Listing 4-22: NSUserDefaults file net.vz.vznetworks.iphone.plist

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

VZ-Netzwerke does not protect its files with Data Protection. All inspected files have the extended attribute `NSFileProtectionNone`.

Does the application use SSL/TLS to secure network connections?

The application transmits all data over plain HTTP.

4.4.6. Reeder v2.3.1

Is the application able to open URLs with a certain URL scheme?

The application registers the URL schemes `reeder` and `fb140883085961607` (see Listing 4-23).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.reederapp</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>reeder</string>
      <string>fb140883085961607</string>
    </array>
  </dict>
</array>
...
```

Listing 4-23: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening `reeder://` and `fb140883085961607://`, the application provides the following URL handlers:

- `reeder://oauth/delicious`

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

Reeder stores the Google Reader username under the key `GoogleLogin` (see Listing 4-24). The password could not be obtained from the file.

```
...
<key>GoogleLogin</key>
<string>XXXXX</string>
...
```

Listing 4-24: NSUserDefaults file ch.reeder.plist

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

Reeder does not protect its files with Data Protection. All inspected files have the extended attribute NSFileProtectionNone.

Does the application use SSL/TLS to secure network connections?

The application uses HTTPS to communicate with Google Reader and HTTP to fetch favicons (favorite icons of the websites) and images .

4.4.7. Twitter v3.3.4

Is the application able to open URLs with a certain URL scheme?

The application registers the URL schemes twitter, com.twitter.twitter-iphone, com.twitter.twitter-iphone+1.0.0, tweetie, com.atebits.Tweetie2, com.atebits.Tweetie2+2.0.0, com.atebits.Tweetie2+2.1.0, com.atebits.Tweetie2+2.1.1, and com.atebits.Tweetie2+3.0.0 (see Listing 4-25).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.twitter.twitter-iphone</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>twitter</string>
      <string>com.twitter.twitter-iphone</string>
      <string>com.twitter.twitter-iphone+1.0.0</string>
      <string>tweetie</string>
      <string>com.atebits.Tweetie2</string>
      <string>com.atebits.Tweetie2+2.0.0</string>
      <string>com.atebits.Tweetie2+2.1.0</string>
      <string>com.atebits.Tweetie2+2.1.1</string>
      <string>com.atebits.Tweetie2+3.0.0</string>
    </array>
  </dict>
</array>
...
```

Listing 4-25: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening `twitter://`, `com.twitter.twitter-iphone://`, `com.twitter.twitter-iphone+1.0.0://`, `tweetie://`, `com.atebits.Tweetie2://`, `com.atebits.Tweetie2+2.0.0://`, `com.atebits.Tweetie2+2.1.0://`, `com.atebits.Tweetie2+2.1.1://`, or `com.atebits.Tweetie2+3.0.0://`, the application provides the following URL handlers:

- `twitter://user`
- `twitter://post`
- `twitter://messages?account=%@&screen_name=%@&show_loader=true`
- `twitter://status?account=%@&id=%@`
- `twitter://status?account=%@&id=%@&type=mention`
- `twitter://timeline?account=%@`

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

No sensitive information in `NSUserDefaults` is found.

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

Twitter does not protect its files with Data Protection. All inspected files have the extended attribute `NSFileProtectionNone`.

Does the application use SSL/TLS to secure network connections?

The application uses exclusively HTTPS to communicate with the Twitter API at `https://api.twitter.com`.

4.4.8. XING v3.1.1

Is the application able to open URLs with a certain URL scheme?

The application registers the URL scheme `xing2` (see Listing 4-26).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.xing.XINGLink</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>xing2</string>
    </array>
  </dict>
</array>
...
```

Listing 4-26: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening `xing2://`, the application provides no URL handler.

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

No sensitive information in `NSUserDefaults` is found.

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

XING does not protect its files. All inspected files have the extended attribute `NSFileProtectionNone`.

Does the application use SSL/TLS to secure network connections?

The application uses exclusively HTTPS to communicate with the backend of XING.

4.4.9. 1Password v359001

Is the application able to open URLs with a certain URL scheme?

The application registers the URL scheme `onepassword`, `1password`, `1p`, and `onep` (see Listing 4-27).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.1password.open</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>onepassword</string>
      <string>1password</string>
      <string>1p</string>
      <string>onep</string>
    </array>
  </dict>
</array>
...
```

Listing 4-27: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening `onepassword://`, `1password://`, `1p://`, or `onep://`, the application provides the following URL handlers:

- `1p://dropbox_authorized`

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

1Password does not use `NSUserDefaults`, because an `NSUserDefaults` file cannot be found in Library/Preferences.

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

1Password does not protect its files with Data Protection. All inspected files have the extended attribute `NSFileProtectionNone`. However, 1Password protects its data by its own mechanisms called the Agile Keychain [AGILE_KEYCHAIN].

Does the application use SSL/TLS to secure network connections?

1Password only creates network connections when the user syncs the 1Password database with Dropbox or Bonjour. When syncing with Dropbox, 1Password uses HTTPS exclusively to communicate with Dropbox.

4.4.10. Telekom Kundencenter v1.0

Is the application able to open URLs with a certain URL scheme?

The application does not register any custom URL schemes.

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

Kundencenter stores the username in `NSUserDefaults` (see Listing 4-28). The password cannot be obtained from the file.

```
...
<key>NSUserDefaultsUserKey</key>
<string>XXXXX</string>
...
```

Listing 4-28: `NSUserDefaults` file `de.telekom.Kundencenter.plist`

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

Kundencenter does not create files in the inspected directories. Thus, it does not need to protect files.

Does the application use SSL/TLS to secure network connections?

The application uses HTTPS to transfer sensitive information over the network. Non-sensitive information is sent with plain HTTP requests.

4.4.11. eBay v2.2.0

Is the application able to open URLs with a certain URL scheme?

The application registers the URL scheme `ebay` (see Listing 4-29).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>ebay</string>
    </array>
  </dict>
</array>
...
```

Listing 4-29: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening `ebay://`, the application provides the following URL handlers:

- `ebay://launch?itm=%@`

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

eBay stores the username under the key `userid` in NSUserDefaults (see Listing 4-30). The password could not be obtained from the file.

```
...
<key>userid</key>
<string>XXXXX</string>
...
```

Listing 4-30: `NSUserDefaults` file `com.ebay.iphone.plist`

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

eBay protects the files in the following directories:

- `Library/Caches/FavoriteSearchCache/`
- `Library/Caches/Settings/`

The other files are unprotected.

Does the application use SSL/TLS to secure network connections?

This evaluation does not evaluate the offer process. It only covers the login process as well as searching and viewing articles. The application uses HTTPS to transfer the credentials during the login process. Plain HTTP is used for requests that do not contain sensitive information.

4.4.12. Amazon.de v1.4.0

Is the application able to open URLs with a certain URL scheme?

The application registers the URL scheme `amazonde` (see Listing 4-31).

```
...
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.amazon</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>amazonde</string>
    </array>
  </dict>
</array>
...
```

Listing 4-31: URL schemes in Info.plist

Does the application provide URL handlers that another application can invoke?

Besides switching to the application by opening `amazonde://`, the application provides no URL handler.

Does the application store sensitive information such as, e.g., user credentials in `NSUserDefaults` instead of in the *Keychain*?

Amazon.de stores the username and the user's full name (see Listing 4-32). Also, it stores potentially security-related binary data under the key `secureCookieJar:msh.amazon.de`.

```
...
<key>AWUserKeyEmail</key>
<string>manuel@binna.de</string>
<key>AWUserKeyFullName</key>
<string>mbinna</string>
...
<key>secureCookieJar:msh.amazon.de</key>
<data>
MXwX8gc5T+ZvQmY4pF81IFXnykrPeahtB60ac33x+B6OvorG4wLux6i7f62jDhJ6Lt2h
82Qxg3U7QTi3//fXNdP5XVJsS3tqqzOW4Bak/6e7NVvmkCIz14D4lCalpWioARDyzy1k
EyU2GgN/HdWhP1GgyFwdqz3TdB+N/5j9czD8os0YMVi8IxHeVH2+45b3nuonxlVvbLX4
RJ8lkD4vaedDjAoFce/uB6gav3ZQMyHegaIN4znkZvQN4g/sGdO3KlYybUZVCvR4Qf7Q
kJhv8OWZtnjv
</data>
...
```

Listing 4-32: `NSUserDefaults` file `de.amazon.AmazonDE.plist`

Does the application protect its files with NSFileProtectionComplete or NSDataWritingFileProtectionComplete?

Amazon.de does not create files in the inspected directories. Thus, it does not need to protect files.

Does the application use SSL/TLS to secure network connections?

The application uses HTTPS for requests of the login and purchase process. Other data is transmitted with plain HTTP which also contain the (unencrypted) UDID.

4.5. Summary

The sandbox tightly controls what an application can and cannot do. However, two applications can use several mechanisms to perform interprocess communication between themselves: URL schemes, Keychain access groups, pasteboards, links in the filesystem, and UNIX domain sockets. In these cases, the user has no control over the information being shared and cannot prevent leaking information from one of the two applications to the other.

Voice Chat Services enable a developer to integrate the functionality to record the ambient sounds of a device that currently runs the developer's application. Voice Chat Services do not require the user's consent before recording and transmitting any data.

`NSUserDefaults` provides a mechanism to persistently store application settings in a property list file in the application's home directory. The file is not protected by the user's passcode (Data Protection). Also, the values are stored in the file unencrypted.

The general pasteboard provided by the class `UIPasteboard` enables the *Cut, Copy, and Paste* functionality of iOS. All applications are able to obtain the current value stored in the general pasteboard. Sensitive information like passwords that an application stores in the general pasteboard may leak to other applications.

When an application changes its state from running in the foreground to running in the background, iOS creates a screenshot of the application's window and animates the transition to the background. If the application displays sensitive information while the screenshot is made, the information is available in the screenshot file. Although the screenshot file inside the application's home directory is protected by Data Protection, applications which display sensitive information may want to prevent the information from appearing in the screenshot.

The certificate detail view in iOS has the shortcoming that certificate details are truncated, especially when the system presents the view in portrait mode. An attacker can use this to create a certificate for a common name that begins with the common name of the certificate she wants to forge. The truncation of the certificate details could mislead the user in believing that she connects to the correct host.

Certificates with a common name that consists of two domains separated by a null character in the common name can also mislead a user into thinking that the certificate is a valid certificate for the requested domain. When presented a certificate with with null character in the common name, the function `SecCertificateCopySubjectSummary()` returns a string that contains only the characters up to the null character.

The security evaluation of particular applications from the App Store reveals that these applications do not store passwords in `NSUserDefaults`. Most of them do not protect their data with Data Protection. Most of the evaluated applications transfer application data securely over the network by using SSL/TLS. However, Facebook only uses SSL/TLS for the login mechanism. It transfers all other application data unencrypted via HTTP. The application VZ-Netzwerke transfers all application data over plain HTTP.

5. Experimental Results

5.1. Defense: Sample Applications with Security-related APIs

This chapter provides an overview of the iOS applications developed to showcase the use of particular security-related APIs. The applications use these APIs to protect their data from attacks. They demonstrate the use of Common Crypto, Randomization Services, Keychain Services, Certificate, Key, and Trust Services, Secure Networking with `UIWebView`, the URL Loading System, and CFNetwork, and they show how to use the new Data Protection API introduced with iOS 4.

5.1.1. CryptoMailer

CryptoMailer demonstrates the usage of the APIs Randomization Services and Common Crypto. The application allows the user to send encrypted messages via email to other users of the application.

The user launches CryptoMailer and enters a message of arbitrary length into a text field. After tapping the button "Back", the application derives the digest for the message with the the hash function configured in the Settings application. CryptoMailer supports the hash functions that are supported by Common Crypto: MD2, MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. The main menu shows the first few characters of the entered message and the hexadecimal representation of the derived digest (see Figure 4-6).

Figure 4-6: Entering a message and deriving the digest

After setting a password, tapping the button "Encrypt" causes the application to encrypt the message with the algorithm configured in the Settings application. The supported algorithms are DES, 3DES, CAST, AES-128, AES-192, and AES-256. CryptoMailer derives the encryption key from the bytes of the UTF-8 encoded password and pads it with ones if the password is too short to provide all necessary bytes for the key. If the password is too long, only the first bytes up to key's length are used. CryptoMailer uses Randomization Services to obtain a random sequence of bytes for the initialization vector (IV). The symmetric encryption uses the cipher block chaining (CBC) mode for the message and the electronic code book (ECB) mode for the IV. The application applies PKCS #7 padding before the actual encryption of the message and the IV.

The encrypted message and the IV are sent as URL parameters in a hyperlink within an HTML email. Because the characters in URL parameters are restricted [RFC3986], the base64-encoded version of the encrypted message and encrypted IV are embedded in the URL. Figure 4-7 demonstrates how the user interface displays the encrypted message after applying the base64 encoding. The second picture in Figure 4-7 shows the message composer view with the HTML message and the embedded hyperlink.

Figure 4-7: Encrypted message after base64 encoding and before sending

The application sends the encrypted message via email to another person. The body of the email is a HTML document with an embedded hyperlink. CryptoMailer registers itself to handle URLs with the scheme `cryptomailer` during installation. This behavior is configured with the `CFBundleURLTypes` property in the application's Info.plist file. Every time an application opens a URL with this scheme, iOS opens the application that registered itself to handle URLs with this scheme. CryptoMailer uses this mechanism to send encrypted messages via email to recipients. The URL has the form:

```
cryptomailer://open?message=ENCRYPTED_MESSAGE&iv=ENCRYPTED_IV
```

A recipient receives the email with the application Mail on the device. When the user taps the hyperlink in the HTML email, iOS detects that CryptoMailer can handle the URL and launches the application. CryptoMailer receives the tapped URL and parses the GET parameters to extract the base64-encoded encrypted message and the base64-encoded encrypted IV. The application populates the user interface with the base64-encoded encrypted message. After entering the password and tapping the button "Decrypt", CryptoMailer first base64-decodes and then decrypts the IV. It then base64-decodes and decrypts the message with the IV. If the whole process is successful, the application displays the decrypted message in the user interface.

5.1.2. KeychainServicesDemo

KeychainServicesDemo shows the usage of the Keychain Services API. It explains how to create, retrieve, update, and delegate Keychains items. It also shows how persistent references are used and demonstrates how to share items between applications with access groups.

Every time the application starts, it checks if it already obtained a persistent reference. If not, the application adds a new item with an access group to the Keychain and retrieves a persistent reference for that item. KeychainServicesDemo persistently stores the obtained persistent reference with `NSUserDefaults`. The first view of the application is a login view that enables the user to enter a username and a password. When the user taps the button "Login", the username and password of the Keychain item that the persistent reference references are updated with the values from the text fields. The login view also offers to reset the username and the password of the Keychain item to empty strings by tapping the button "Reset login data" (Figure 4-8).

Figure 4-8: KeychainServicesDemo

The Xcode project is configured to produce two different versions of the application. Each version has its own App ID in the iOS Provisioning Portal and is signed with its own Provisioning Profile created for this App ID. Keychain items are by default only accessible to the application that created them. However, both versions of the application are a member of a shared Keychain access group and automatically share

all items that have the configured access group. The membership in access groups is configured in the file Entitlements.plist in the application bundle. The entry `keychain-access-groups` defines an array with access groups. The access group at index 0 is the application's own access group. The access group `de.rub.emma.KeychainDemoSuite` at index 1 is the shared access group.

When run on the device, the applications set the access group of the item that is initially added to the Keychain to be the shared access group. When the user enters a username and a password in one version of the application and taps the button "Login", the other version of the application displays the credentials entered in the first application.

Applications built for the iOS Simulator are not signed. Keychain access groups are therefore not available on the iOS Simulator. Trying to add an item with an access group results in an error. The Xcode project anticipates this by setting the access group of the initial Keychain item only if the product is built for an actual device.

5.1.3. CertKeyTrustServicesDemo

The CertKeyTrustServicesDemo application demonstrates the use of the Certificate, Key, and Trust Services API. It explains how to extract an identity (a certificate and the corresponding private key) from an encrypted PKCS #12 file, evaluate the trust of the certificate, add the identity to the Keychain, and retrieve the identity from the Keychain. It also explains how to generate an RSA asymmetric key pair, encrypt a text with a public key, decrypt the ciphertext with the corresponding private key, sign a text with a private key, and verify the signature with the corresponding public key. Figure 4-9 shows the two views of the application.

Figure 4-9: The sample application CertKeyTrustServicesDemo

After the application launch, the tab "Certificate" is the active tab in the tab bar at the bottom of the screen. The application's main bundle contains an encrypted PKCS #12 file with an identity. CertKeyTrustServicesDemo reads the file from disk and extracts the embedded identity with `SecPKCS12Import()`. The function returns a `SecIdentityRef` and a `SecTrustRef` for every identity embedded in the file. The `SecTrustRef` is used to evaluate the trust of the identity with `SecTrustEvaluate()`. The first evaluation fails because the embedded identity is not signed by a trusted third party. `SecTrustCopyExceptions()` and `SecTrustSetExceptions()` mark the identity as trusted so that future evaluations will not fail when one of the exceptions occurs. The second evaluation of the `SecTrustRef` succeeds. The application adds the identity with `SecItemAdd()` to the Keychain and obtains a persistent reference for it. `SecIdentityCopyCertificate()` is used to obtain the certificate (`SecCertificateRef`) from the identity. The view displays the certificate subject summary obtained via `SecCertificateCopySubjectSummary()`. The persistent reference is used to obtain the identity from the Keychain and display the attributes of the returned item in the user interface.

When the user taps the tab "Key" in the tab bar at the bottom, the application generates a new RSA key pair by invoking `SecKeyGeneratePair()`. Tapping the button "Encrypt & Decrypt" causes the application to first encrypt a string with `SecKeyEncrypt()` by using the generated public key. Next the ciphertext is decrypted with `SecKeyDecrypt()` and the generated private key. The resulting plaintext is checked if it matches the original text. Tapping the button "Sign & Verify"

causes the application to sign a string and then verify the signature. The application derives the SHA-1 digest from the text with `CC_SHA1()` of the Common Crypto API and creates a signature with `SecKeyRawSign()`. It then verifies the signature with `SecKeyRawVerify()` and displays the result of the verification.

5.1.4. SecureNetworkingDemo

The application SecureNetworkingDemo showcases the usage of `UIWebView`, the URL Loading System, and CFNetwork to demonstrate how secure network connections can be implemented in iOS applications. Every mode of the application (represented by the tabs in the tab bar at the bottom of the screen) is a browser that loads the URL entered in the search bar and displays the loaded contents with an instance of the class `UIWebView`. The first mode uses the class `UIWebView` to load the contents. The second mode uses the URL Loading System. The third mode uses CFNetwork. Figure 4-10 shows the application in each of the three modes.

Figure 4-10: The sample application SecureNetworkingDemo

Establishing a secure network connection usually involves properly responding to authentication challenges. SecureNetworkingDemo can respond to authentication challenges for the following mechanisms:

- HTTP Basic Access Authentication
- HTTP Digest Access Authentication
- SSL/TLS server authentication
- SSL/TLS client authentication

Normal SSL/TLS connections only provide server authentication. The server sends the client its certificate which the client validates. In such cases the application only receives the server authentication challenge with the server's trust management object. If the server is configured to require SSL/TLS client certificates, the server authentication challenge is followed by a client authentication challenge to establish a mutually authenticated connection. In the client authentication challenge, the application provides the client certificate that is sent to the server. To develop and test SecureNetworkingDemo, an instance of the Apache 2.2 web server is used to host different locations which require the mentioned authentication mechanisms. In this scenario, the certificates used for SSL/TLS authentication are self-signed and thus not issued by a trusted third party. The application nevertheless evaluates certificates received during SSL/TLS handshakes and decides if a given certificate is valid before it is used to establish the secure connection. The server configuration files, web sites, certificates, and private keys are available in the project directory on the bundled disc.

The mode "UIWebView" loads the URL entered in the search bar with the class `UIWebView`. The instance responsible for the load conforms to the `UIWebViewDelegate` protocol, is set as the delegate of the `UIWebView` instance, and receives the messages the `UIWebView` instance sends to it. The `UIWebViewDelegate` protocol does not declare delegate methods related to authentication. Therefore, the `UIWebView`'s delegate cannot handle authentication challenges and, as a result, the `UIWebView` instance is not able to access locations that require authentication it cannot perform by itself. To circumvent this shortcoming, the application implements the suggestion made in "How to display the Authentication Challenge in UIWebView" on Stack Overflow [SO_WEBVIEW]. The delegate of `UIWebView` uses the URL Loading System to handle authentication challenges. Listing 3-39 shows this approach in detail. The mode "UIWebView" handles the authentication challenges exactly like the mode "URL Loading System" explained next.

The mode "URL Loading System" loads the entered URL with the URL Loading System [URLLS_P_G]. The `UIWebView` is only used to display the resulting contents. The controller that is responsible for the load implements the `NSURLConnection` delegate methods. The methods related to authentication are:

- `connectionShouldUseCredentialStorage:` determines if the loader should use credentials from the credential storage or if it should rely exclusively on the delegate.
- `connection:canAuthenticateAgainstProtectionSpace:` states which authentication challenges the delegate can respond to.
- `connection:didReceiveAuthenticationChallenge:` provides the delegate the authentication challenge it needs to respond to.
- `connection:didCancelAuthenticationChallenge:` states that the connection did cancel the authentication challenge and provides the delegate the chance to fail gracefully.

`connectionShouldUseCredentialStorage:` returns `NO` to indicate that all credentials are supplied by the NSURLConnection delegate. Otherwise iOS would use valid credentials stored in the global credential storage.

`connection:canAuthenticateAgainstProtectionSpace:` returns YES for the following authentication methods:

- `NSURLAuthenticationMethodHTTPBasic` - the delegate can provide HTTP Basic Access Authentication credentials for the protection space.
- `NSURLAuthenticationMethodHTTPDigest` - the delegate can provide HTTP Digest Access Authentication credentials for the protection space.
- `NSURLAuthenticationMethodServerTrust` - the delegate can evaluate the server's trust management object during SSL/TLS connection establishment.
- `NSURLAuthenticationMethodClientCertificate` - the delegate is able to provide a client identity during SSL/TLS connection establishment.

The method `connection:didReceiveAuthenticationChallenge:` implements the logic necessary to properly respond to authentication challenges. For HTTP Basic Access Authentication and HTTP Digest Access Authentication, it creates an `NSURLCredential` with a username and a password. The credential is valid until the session ends. It then tells the sender of the challenge to use the credential for the current authentication challenge.
For SSL/TLS server authentication the controller accesses the trust management object (`SecTrustRef`) from the challenge's associated `NSURLProtectionSpace` instance. `SecTrustEvaluate()` is used to validate the certificate received from the server. The application copies and sets the evaluation exceptions with `SecTrustCopyExceptions()` and `SecTrustSetExceptions()`. A real application must carefully decide what to do before doing this! Next a new `NSURLCredential` is created for the trust management object. The sender is then told to use this credential for the current authentication challenge.
SSL/TLS client authentication is handled by first importing an identity (`SecIdentityRef`) from a PKCS #12 file in the application's main bundle with `SecPKCS12Import()`. The method creates a new `NSURLCredential` with this identity. The credential is valid until the end of the session. It then tells the sender to use this credential for the current authentication challenge.

The mode "CFNetwork" uses CFNetwork to load the contents of the entered URL. This mode only implements SSL/TLS server authentication. The controller creates a new HTTP request with `CFHTTPMessageCreateRequest()` for the entered URL and a read stream (`CFReadStreamRef`) with `CFReadStreamCreateForHTTPRequest()`. It configures the stream to automatically follow HTTP redirections (`kCFStreamPropertyHTTPShouldAutoredirect`) and to use the security level (`kCFStreamSSLLevel`) TLS version 1 (`kCFStreamSocketSecurityLevelTLSv1`). The stream does not validate the certificate chain of the server certificate

(kCFStreamSSLValidatesCertificateChain). The read stream is toll-free bridged to NSInputStream, scheduled in the default run loop mode (NSDefaultRunLoopMode), and then opened. The controller implements the optional method stream:handleEvent: of the NSStreamDelegate protocol. iOS performs the SSL/TLS handshake and invokes this method when the stream has bytes available to be read (NSStreamEventHasBytesAvailable). The controller obtains the trust management object with the server certificate and evaluates it with SecTrustEvaluate(). The application copies and sets the evaluation exceptions with SecTrustCopyExceptions() and SecTrustSetExceptions(). A real-world application must carefully decide what to do before doing this! If the evaluation fails, the controller closes the stream. If the evaluation succeeds the controller reads from the secure stream.

5.1.5. DataProtectionDemo

The application DataProtectionDemo explains the use of the Data Protection API introduced with iOS 4. It shows how to detect the availability of protected data and how an application can protect files and Keychain items.

After the application launches, it detects if protected data is available by sending the shared UIApplication instance the message isProtectedDataAvailable. The application delegate conforms to the protocol UIApplicationDelegate and implements the two methods applicationProtectedDataDidBecomeAvailable: and applicationProtectedDataWillBecomeUnavailable:. The single view controller observes the UIApplicationProtectedDataDidBecomeAvailable and UIApplicationProtectedDataWillBecomeUnavailable notifications.

DataProtectionDemo reads the contents of a text file in the application's main bundle with +[NSData dataWithContentsOfFile:] and writes the data with -[NSData writeToFile:options:error:] and NSDataWritingFileProtectionComplete to a protected new file. The application also shows the alternative method to protect a file: use -[NSFileManager setAttributes:ofItemAtPath:error:] to set the attribute NSFileProtectionKey of an unprotected existing file to NSFileProtectionComplete. The application removes the protection of the first protected file by setting its attribute NSFileProtectionKey to NSFileProtectionNone. The protection class of a file can only be changed if protected data is available.

The application adds a new item with the default protection class kSecAttrAccessibleWhenUnlocked to the Keychain by using SecItemAdd(). It then retrieves that item with SecItemCopyMatching() to inspect the item's values. Next, it updates the protection class with SecItemUpdate() and retrieves the item again to verify that the protection class is updated successfully.

5.2. Offense: Sample Applications with Offensive Techniques

5.2.1. HeartLinker and HeartReader

HeartLinker copies the file SecretText.txt from the main bundle to the application's Documents directory, and creates both a symbolic link to this file at the path `/var/mobile/Library/AddressBook/TEST_SOFT_LINKED.txt` and a hard link at the path `/var/mobile/Library/AddressBook/TEST_HARD_LINKED.txt`.

HeartReader accesses the symbolic link stored at the path `/var/mobile/Library/AddressBook/TEST_SOFT_LINKED.txt` and the hard link at the path `/var/mobile/Library/AddressBook/TEST_HARD_LINKED.txt` and tries to read the contents of the referenced files. By reading the contents from the referenced file, HeartReader accesses files originally stored in the home directory of another application.

HeartReader is able to obtain the destination of the symbolic link retrieving the random path to HeartLinker's home directory. However, it cannot read the file's contents. The attempt fails with the error message "The operation couldn't be completed. Operation not permitted". The access to the contents of the hard link is successful. HeartReader can access the contents of the hard link which HeartLinker created.

5.2.2. PasteboardIPCServer and PasteboardIPCClient

PasteboardIPCServer creates a new persistent application pasteboard with the name "SharedPasteboardName" and stores the text "Hello, world" in the pasteboard. PasteboardIPCClient obtains the pasteboard with the name "SharedPasteboardName" and reads the text. Hereby, it obtains the text stored by PasteboardIPCServer.

5.2.3. SocketIPCServer and SocketIPCClient

SocketIPCServer listens for incoming connections on the UNIX domain socket at the path `/var/mobile/Library/AddressBook/SOCKET`. When it receives an incoming connection, it writes the UTF-8 encoded bytes of the `NSString` "This is a message from the other program." to the connection.

SocketIPCClient connects to the UNIX domain socket at the path `/var/mobile/Library/AddressBook/SOCKET` and prints the received bytes as UTF-8-decoded value of the class `NSString`.

5.2.4. AmbientSounds

The sample application AmbientSounds is a stripped-down version of the sample code GKRocket from the iOS Developer Library [GKROCKET]. AmbientSounds shows a list of devices either connected to the local WiFi network or with Bluetooth enabled that also run the application (see Figure 4-11). When the user taps an entry in the list, AmbientSounds establishes a voice chat connection between the two devices. The owner of the other device may not recognize the connection establishment and the interception of its ambient sounds.

Figure 4-11: AmbientSounds

AmbientSounds uses peer-to-peer-based discovery with the class GKSession to find devices with which it can establish a voice communication channel. It is therefore limited to establish voice chats only between devices from the same local WiFi network or over Bluetooth. It would also be possible to implement a version of AmbientSounds which uses a server-based directory and could therefore discover peers globally.

5.2.5. Pasteboarder

Pasteboarder is an application that tries to access the contents of the file `/private/var/mobile/Library/Caches/com.apple.UIKit.pboard/pasteboardDB` in which the system stores the contents of the persistent pasteboards. The first attempt accesses `/private/var/mobile/Library/Caches/com.apple.UIKit.pboard/pasteboardDB` directly. The second attempt creates a symbolic link to the file pasteboardDB in the application's temporary folder and tries to access the contents through that link. the third attempt creates a hard link in the application's temporary directory and tries to access its contents. All these three attempts fail due to missing permissions in the filesystem. The fourth attempt uses the public API provided by the class `UIPasteboard` to access the contents of the general pasteboard. This attempt (logically) succeeds. Pasteboarder prints the text that another application previously stored in the general pasteboard.

5.2.6. ScreenBlank

The application ScreenBlank shows how an application can protect its sensitive information from appearing in the screenshot that is automatically taken by iOS when the application transitions from the foreground to the background, e.g., when the user presses the home button. The screenshot is used by iOS to animate the transition. Before the system creates the screenshot (after the application did enter the background and additionally before it will terminate), the application can manipulate its view hierarchy to remove sensitive information from the user interface. ScreenBlank implements a category on `UIViewController`, named `MBSensitiveInformationInScreenshotPrevention`. Every instance of class `UIViewController` can prevent sensitive information from appearing in the screenshot by invoking the category method `mb_startPreventingSensitiveInformationFromAppearingInScreenshot`, which registers observers for the notifications `UIApplicationDidEnterBackgroundNotification`, `UIApplicationWillTerminateNotification`, and `UIApplicationWillEnterForegroundNotification` in the default notification center. The handler for `UIApplicationDidEnterBackgroundNotification` and `UIApplicationWillTerminateNotification` hides the view controller's root view

and thus all of its subviews. The handler for `UIApplicationWillEnterForegroundNotification` unhides the view controller's root view and thus all of its subviews. `mb_stopPreventingSensitiveInformationFromAppearingInScreenshot` removes the observers from the default notification center.

The image file is stored in the directory Caches/Snapshots within the application's home directory. After the application did become active, ScreenBlank retrieves the extended attributes of the file `UIApplicationAutomaticSnapshotDefault-Portrait@2x.jpg` and prints the value for the key `NSFileProtectionKey` to the console.

5.3. Summary

The defensive sample applications presented in this chapter show that iOS provides an extensive set of tools to perform security-related tasks, such as symmetric and asymmetric encryption and decryption, digital signatures, certificates, hash functions, MACs, secure network connections via SSL/TLS, and Data Protection. The offensive sample applications show that there are unresolved issues with e.g., the application sandbox that should control IPC between applications more tightly, the screenshot that is automatically created for the "app enters background" animation, and the possibility to establish voice chats in an application without the user's consent.

6. Conclusion and Future Work

The focus of this work is the development of secure applications for non-jailbroken iOS devices. iOS provides third-party applications a restricted and tightly controlled environment. The authenticated boot process and mandatory code signing ensures that only applications with valid code signatures can run on the device. The application sandbox restricts what applications are allowed to do in the system. Although not used by the current version, ARM TrustZone has been available on all iOS devices and may be used in future versions of iOS. DEP and ASLR make it more difficult to exploit vulnerabilities.

By design, the Objective-C programming language lacks good security properties. The language is based on the idea that as many decisions as possible are made at runtime instead of at compile-time. In combination with the Objective-C Runtime, this enables the invocation of every method of every class. Private methods are methods that are not declared in a public header file. However, if a developer knows the name of the private method, it can be invoked. A solutions to prevent code outside the class from invoking private behavior, a class can implement the private code in a static C function and call this function from within a private method. Static functions are only visible within the file in which they are implemented and are also not known to the Objective-C Runtime.

Instance variables of a class can be declared private with the keyword `@private`. Accessing a private instance variable from outside the class causes a compiler error. Nevertheless, Key-Value Coding (KVC) allows to access a private instance variable from outside the class if the developer does not implement the class method `accessInstanceVariablesDirectly` and always return the value `NO`.

The Foundation framework provides classes and mechanisms that are used in almost every iOS application, among others: collections and data structures: strings with Unicode support, memory management, persistence, concurrency, date and time, network communication, and files. Strings are often initialized with format strings which provide the risk of introducing format string vulnerabilities if the developer does not use them in a secure manner. Objective-C is a superset of C. In addition to the more secure Foundation class `NSString` (and its subclass `NSMutableString`), Objective-C also allows the usage of C strings which provide the risk of introducing buffer overflow vulnerabilities when the string's termination character `'\0'` is not handled correctly.

Except for the lack of security properties in the Objective-C programming language and the danger of misusing classes or mechanisms of the Foundation framework, iOS provides a well-designed and feature-rich environment for developing secure mobile applications. There exist many security-related APIs a developer can use to

protect application data. Common Crypto provides functions to perform symmetric encryption and decryption (with the algorithms AES, DES, 3DES, CAST-128, RC2, RC4), and derive digests and HMACs (with the hash functions MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512).

Keychain Services provides access to the Keychain, a secure storage for credentials, certificates, and private keys. The sensitive information of Keychain items is stored encrypted on disk. Every application can only obtain its own Keychain items. However, applications from the same developer can use access groups to share Keychain items among themselves.

Certificate, Key, and Trust Services provides an API to extract certificates and private keys from PKCS#12 files, to validate certificates, to perform asymmetric encryption and decryption (with RSA) as well as to create and verify digital signatures (with RSA).
The URL Loading System is a high-level mechanism to load a remote resource securely over the network. It supports HTTP Basic Access Authentication, HTTP Digest Access Authentication, SSL/TLS server authentication, and SSL/TLS authentication of both server and client. During the establishment of a secure SSL/TLS connection, the developer uses Certificate, Key, and Trust Services to validate the server certificate and provide the client identity to the loading system.

CFNetwork is a networking API in the C programming language which allows finer-grained control over the SSL/TLS stream properties than the URL Loading System. The developer can explicitly configure the socket security level (SSLv2, SSLv3, TLSv1, or none) and configure if expired certificates are allowed, if expired root certificates are allowed, if root certificates are allowed as server certificates, if the peer's certificate chain is validated during the connection establishment, the name used for the validation of the peer's certificate, the identity that is used for the authentication towards the peer, and if the stream should act as the server in the SSL/TLS protocol.

Data Protection is both an architecture and an API which is available since iOS 4. Data Protection ties the confidentiality of protected files and Keychain items to the availability of the user's passcode. The API provides protection classes for files and Keychain items that define when data is available to the application. The protection classes are grouped into the three classes "always available", "available after first unlock", and "available after unlock". When the user locks the device, the passcode is securely deleted and data in the class "available after unlock" becomes unavailable. Before unlocking the device for the first time after a restart, the data of the classes "available after first unlock" and "available after unlock" is not available. The data of the class "always available" is always available, regardless if the device is locked or unlocked. A local attacker that gains access to a lost or stolen iOS device with enabled Data Protection can only access the data of the classes "always available" and "available after first unlock" (if the device is on) or only data of the class "available after unlock" (if the device is off or needs to be restarted).

Developers can protect their application's data by using these APIs correctly. However, this statement only holds true as long as no jailbreak exists. On a jailbroken device, attackers are able to bypass the security mechanisms and, according to recent research, may even be able to obtain the unencrypted content of files and Keychain items that are protected by Data Protection by performing a brute-force attack on the user's passcode.

Although the sandbox tightly controls what an application can and cannot do, two applications can use several mechanisms to perform interprocess communication between themselves: URL schemes, Keychain access groups, pasteboards, links in the filesystem, and UNIX domain sockets. In these cases, the user has no control over the information being shared and cannot prevent leaking information from one of the two applications to the other.

Voice Chat Services of the Game Kit framework enable a developer to integrate the functionality to record the ambient sounds of a device that currently runs the developer's application. Voice Chat Services do not require the user's consent before recording and transmitting any data.

`NSUserDefaults` provides a mechanism to persistently store application settings in a property list file in the application's home directory. The file is not protected by the user's passcode (Data Protection). Also, the values are stored in the file unencrypted. Sensitive information should therefore never be stored in `NSUserDefaults`. The Keychain provides a secure place to store such information.

The general pasteboard provided by the class `UIPasteboard` enables the *Cut, Copy, and Paste* functionality of iOS. All applications are able to obtain the current value stored in the general pasteboard. Sensitive information like passwords that an application stores in the general pasteboard may leak to other applications.

When an application changes its state from running in the foreground to running in the background, iOS creates a screenshot of the application's window and animates the transition to the background. If the application displays sensitive information while the screenshot is made, the information is available in the screenshot file. Although the screenshot file inside the application's home directory is protected by Data Protection, applications which display sensitive information may want to prevent the information from appearing in the screenshot. An application can register as an observer for the notifications that indicate that the application enters the background or is being suspended and remove the sensitive information from the displayed view.

The certificate detail view in iOS has the shortcoming that certificate details are truncated, especially when the system presents the view in portrait mode. An attacker can use this to create a certificate for a common name that begins with the

common name of the certificate she wants to forge. The truncation of the certificate details could mislead the user in believing that she connects to the correct host.

Certificates with a common name that consists of two domains separated by a null character in the common name can also mislead a user into thinking that the certificate is a valid certificate for the requested domain. When presented a certificate with with null character in the common name, the function `SecCertificateCopySubjectSummary()` returns a string that contains only the characters up to the null character.

The security evaluation of particular applications from the App Store reveals that these applications do not store passwords in `NSUserDefaults`. Most of them do not protect their data with Data Protection. Most of the evaluated applications transfer application data securely over the network by using SSL/TLS. However, Facebook only uses SSL/TLS for the login mechanism. It transfers all other application data unencrypted via HTTP. The application VZ-Netzwerke transfers all application data over plain HTTP.

The sample applications, which were developed during the work on this thesis, document the practical applicability of the discussed approaches to develop secure mobile applications for iOS. They also show that the offensive techniques work as described on actual devices that run the latest version of iOS, iOS 4.3.3.

Future work in this field may involve the following topics:

- Comparison of the security between jailbroken vs. non-jailbroken iOS devices
- Security evaluation of the enterprise features of iOS
  - Mobile device management
  - Configuration profiles
  - Over-the-Air enrollment with the Simple Certificate Enrollment Protocol
  - Wireless app distribution
- In-depth security evaluation of existing iOS applications
- Security evaluation of the App Store application distribution and In App Purchase system

A. References

[3GS_ENCRYPTION]
Hacker Says iPhone 3GS Encryption Is 'Useless' for Businesses
Wired Gadget Lab
2009-07-23
http://www.wired.com/gadgetlab/2009/07/iphone-encryption/
(retrieved 2011-03-21 at 22:33)

[41_API_DIFFS]
iOS 4.0 to iOS 4.1 API Differences
iOS Developer Library
http://developer.apple.com/library/ios/#releasenotes/General/iOS41APIDiffs/
(retrieved 2011-06-02 at 12:11)

[AGILE_KEYCHAIN]
Agile Keychain Design
http://help.agilebits.com/1Password3/agile_keychain_design.html
(retrieved 2011-06-03 at 03:35)

[APPL_PRESS_IOS43]
Apple Introduces iOS 4.3
http://www.apple.com/pr/library/2011/03/02ios.html
(retrieved 2011-03-25 at 11:29)

[APPLE_DATA_PROT]
iOS4: Understanding data protection
http://support.apple.com/kb/HT4175
(retrieved 2011-03-23 at 11:52)

[APPLE_SANDBOX]
The Apple Sandbox
Dionysus Blazakis
Black Hat DC 2011
http://www.semantiscope.com/research/BHDC2011/BHDC2011-Paper.pdf
(retrieved 2011-04-13 at 11:21)

[APUE]
Advanced Programming in the UNIX Environment (Second Edition)
W. Richard Stevens, Stephen A. Rago
Addison Wesley, 2005

[ASRG]
App Store Review Guidelines
http://developer.apple.com/appstore/guidelines.html
(retrieved 2011-06-02 at 12:09)

[AURORA_FEINT]
First iPhone App Pulled from Apple App Store
http://www.ghacks.net/2008/07/28/first-iphone-app-pulled-from-apple-app-store/
(retrieved 2011-04-05 at 13:34)

[CC_MANPAGES]
Common Crypto man page CC_crypto
http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man3/Common%20Crypto.3cc.html
(retrieved 2011-06-02 at 12:14)

[CC_SRC]
Common Crypto Source Code
http://www.opensource.apple.com/source/CommonCrypto/
(retrieved 2011-06-02 at 12:06)

[CF_SS_REF]
CFStream Socket Additions
Revision 2010-03-22
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/CoreFoundation/Reference/CFSocketStreamRef/Reference/reference.html
(retrieved 2011-03-16 at 17:21)

[CDSA_ML_EC_KEY]
Elliptic Curve Crypto on iOS
Apple CDSA Mailing List
http://lists.apple.com/archives/Apple-cdsa/2011/Mar/msg00007.html
(retrieved 2011-03-16 at 11:15)

[CFN_PROG_G]
CFNetwork Programming Guide
Revision 2009-05-06
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Networking/Conceptual/CFNetwork/Introduction/Introduction.html
(retrieved 2011-06-22 at 12:16)

[CKTS_PROG_G]
Certificate, Key, and Trust Services Programming Guide
Revision 2010-07-09
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Security/Conceptual/CertKeyTrustProgGuide/01introduction/introduction.html
(retrieved 2011-06-22 at 12:18)

[CKTS_REF]
Certificate, Key, and Trust Services Reference
Revision 2010-09-01
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Security/Reference/certifkeytrustservices/Reference/reference.html
(retrieved 2011-06-02 at 12:21)

[CS_ALLOC_MAN]
codesign_allocate(1) Mac OS X Manual Page
http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/codesign_allocate.1.html
(retrieved 2011-04-05 at 14:25)

[CS_G]
Code Signing Guide
Revision 2009-10-13
Mac OS X Developer Library
http://developer.apple.com/library/mac/#documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html
(retrieved 2011-06-02 at 12:22)

[CS_MAN]
codesign(1) Mac OS X Manual Page
http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/codesign.1.html
(retrieved 2011-04-05 at 14:30)

[DATA_PROT]
Limitations of Data Protection in iOS 4
http://anthonyvance.com/blog/forensics/ios4_data_protection/
(retrieved 2011-03-20 at 19:48)

[DECRYPT_APP]
Reverse Engineering iPhone AppStore Binaries
Pedram Amini
2009-03-06 , TippingPoint Digital Vaccine Laboratories
http://dvlabs.tippingpoint.com/blog/2009/03/06/reverse-engineering-iphone-appstore-binaries
(retrieved 2011-05-25 at 14:53)

[DHANJANI_BHEU2011]
New Age Application Attacks Against Apple's iOS [and Countermeasures]
Nitesh Dhanjani
Black Hat Europe 2011
https://media.blackhat.com/bh-eu-11/Nitesh_Dhanjani/BlackHat_EU_2011_Dhanjani_Attacks_Against_Apples_iOS-WP.pdf
(retrieved 2011-03-29 at 18:05)

[DIST_ENT_IOS4]
Distributing Enterprise Apps for iOS 4 Devices
Revision 2011-03-04
iOS Developer Library
http://developer.apple.com/library/ios/#featuredarticles/FA_Wireless_Enterprise_App_Distribution/Introduction/Introduction.html
(retrieved 2011-04-04 at 18:03)

[ELCOM_PPB_FAQ]
Phone Password Breaker Frequently Asked Questions
Elcomsoft
http://www.elcomsoft.com/iphone_password_recovery.html
(retrieved 2011-01-25 at 13:18)

[ELCOMSOFT_DATA_PROTECTION_1]
ElcomSoft Breaks iPhone Encryption, Offers Forensic Access to File System Dumps
Vladimir Katalov
Elcomsoft, 2011-05-23
http://blog.crackpassword.com/2011/05/elcomsoft-breaks-iphone-encryption-offers-forensic-access-to-file-system-dumps/
(retrieved 2011-05-29 at 15:10)

[ELCOMSOFT_DATA_PROTECTION_2]
Extracting the File System from iPhone/iPad/iPod Touch Devices
Andrey Belenko
Elcomsoft, 2011-05-23
http://blog.crackpassword.com/2011/05/extracting-the-file-system-from-iphone-ipad-ipod-devices/
(retrieved 2011-05-29 at 15:19)

[ELCOMSOFT_PPB]
Elcomsoft Phone Password Breaker
http://www.elcomsoft.com/eppb.html
(retrieved 2011-05-29 at 16:11)

[FSWALKER]
FSWalker - File System Browser for iPhone OS
http://code.google.com/p/fswalker/
(retrieved 2011-04-13 at 15:41)

[FSWALKER_MBINNA]
FSWalker with modifications to display Data Protection class
https://github.com/mbinna/FSWalker
(retrieved 2011-06-01 at 11:56)

[GK_PROG_G]
Game Kit Programming Guide
Revision 2011-03-08
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/GameKit_Guide/Introduction/Introduction.html
(retrieved 2011-05-18 at 8:54)

[GKROCKET]
GKRocket
Revision 2011-03-15
iOS Developer Library
http://developer.apple.com/library/ios/#samplecode/GKRocket/Introduction/Intro.html
(retrieved 2011-05-18 at 11:46)

[HAC]
Handbook of Applied Cryptography
Menezes, Oorschot, Vanstone
CRC Press, October 1997

[HACK_IPHONE]
25C3 presentation "Hacking the iPhone"
The iPhone Wiki
http://theiphonewiki.com/wiki/index.php?title=25C3
(retrieved 2011-04-13 at 10:38)

[HITB2011]
iPhone data protection in depth
Jean-Baptiste Bédrune and Jean Sigwald
Hack In The Box 2011 Conference, Amsterdam, 17 - 20 May, 2011
http://conference.hackinthebox.org/hitbsecconf2011ams/materials/D2T2%20-%20Jean-Baptiste%20Be%CC%81drune%20&%20Jean%20Sigwald%20-%20iPhone%20Data%20Protection%20in%20Depth.pdf
(retrieved 2011-05-29 at 16:55)

[IKEE]
First iPhone worm spreading in the wild, Sophos reports
http://www.sophos.com/pressoffice/news/articles/2009/11/iphone-worm.html
(retrieved 2011-04-05 at 16:28)

[IKEE_B]
An Analysis of the Ikee.B (Duh) iPhone Botnet
Phillip Porras, Hassen Saidi, and Vinod Yegneswaran
http://mtc.sri.com/iPhone/
(retrieved 2011-04-05 at 16:26)

[IOS_A_PROG_G]
iOS Application Programming Guide
Revision 2010-08-20
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html
(retrieved 2011-06-02 at 13:08)

[IOS_DEV_PROG]
iOS Developer Program
http://developer.apple.com/programs/ios/
(retrieved 2011-03-20 at 18:23)

[IOS_DPLA]
iOS Developer Program License Agreement
http://developer.apple.com/appstore/guidelines.html
(retrieved 2011-06-02 at 13:13)

[IOS_NET_INET]
iOS Networking & Internet
http://developer.apple.com/technologies/ios/networking.html
(retrieved 2011-03-17 at 15:06)

[IOS_REF_LIB]
iOS Developer Library
http://developer.apple.com/library/ios/navigation/
(retrieved 2011-03-25 at 12:37)

[IOS4_ROOT_CERTS]
iOS 4.x: List of available trusted root certificates
Last changed 08 November, 2010
http://support.apple.com/kb/HT4415
(retrieved 2011-05-22 at 22:47)

[IPHONE_PIRACY]
iPhone Piracy
Nicolas Seriot
Black Hat DC 2010
https://www.blackhat.com/presentations/bh-dc-10/Seriot_Nicolas/BlackHat-DC-2010-Seriot-iPhone-Privacy-wp.pdf
(retrieved 2011-03-28 at 11:02)

[IPHONE_SEC_O]
iPhone in Business
Security Overview
http://images.apple.com/iphone/business/docs/iPhone_Security.pdf
(retrieved 2011-03-21 at 11:15)

[ITUNES_BKP]
iPhone and iPod touch: About backups
http://support.apple.com/kb/HT1766
(retrieved 2011-01-25 at 15:27)

[KCS_PROG_G]
Keychain Services Programming Guide
Revision 2009-10-19
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html
(retrieved 2011-06-02 at 13:15)

[KCS_REF]
Keychain Services Reference
Revision 2010-09-01
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Security/Reference/keychainservices/Reference/reference.html
(retrieved 2011-06-02 at 13:16)

[KVC_PROG_G]
Key-Value Coding Programming Guide
Revision 2011-03-08
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html
(retrieved 2011-04-15 at 11:03)

[LOST_IPHONE]
Lost iPhone? Lost Passwords!
Practical Consideration of iOS Device Encryption Security
Jens Heider, Matthias Boll
2011-02-09
Fraunhofer Institute for Secure Information Technology
http://www.sit.fraunhofer.de/forschungsbereiche/projekte/Lost_iPhone.jsp
(retrieved 2011-03-24 at 11:13)

[MAC_HACKER]
The Mac Hacker's Handbook
Charlie Miller, Dino Dai Zovi
Wiley Publishing, Inc., 2009

[MITMPROXY]
mitmproxy - an SSL-capable intercepting proxy
http://mitmproxy.org/
(retrieved 2011-05-27 at 16:11)

[MOGOROAD]
Retrievable iPhone numbers mean potential privacy issues
http://www.macworld.com/article/143047/2009/09/phone_hole.html
(retrieved 2011-04-05 at 13:35)

[MORRISSEY_FORENSIC]
iOS Forensic Analysis for iPhone, iPad, and iPod touch
Sean Morrissey
Appress, 2010

[NOISEBRIDGE_DISCUSS]
Merry Certmas! CN=*\x00thoughtcrime.noisebridge.net
Jacob Applebaum
2009-09-29
https://www.noisebridge.net/pipermail/noisebridge-discuss/2009-September/008400.html
(retrieved 2011-05-22 at 22:42)

[NSURLPS_C_REF]
NSURLProtectionSpace Class Reference
Revision 2010-03-25
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/Foundation/Classes/NSURLProtectionSpace_Class/Reference/Reference.html
(retrieved 2011-03-16 at 20:50)

[NTLM]
NT LAN Manager (NTLM) Authentication Protocol Specification
MSDN Library
http://msdn.microsoft.com/en-us/library/cc236621(v=PROT.10).aspx
(retrieved 2011-03-16 at 21:16)

[NULL_PREFIX]
Null Prefix Attacks Against SSL/TLS Certificates
Moxie Marlinspike
2009-07-29
http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf
(retrieved 2011-05-22 at 20:30)

[OBJC_PROG_LANG]
The Objective-C Programming Language
Revision 2010-12-08
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html
(retrieved 2011-04-15 at 10:25)

[OBJC_RUNTIME_P_G]
Objective-C Runtime Programming Guide
Revision 2009-10-19
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html
(retrieved 2011-04-15 at 10:10)

[P_C_A_S]
Professional Cocoa Application Security
Graham J. Lee
John Wiley & Sons; 1st Edition (4 June, 2010)

[PHRACK_OBJC]
The Objective-C Runtime: Understanding and Abusing
nemo
Phrack Inc., Volume 0x0d, Issue 0x42, Phile #0x04 of 0x11
http://www.phrack.org/issues.html?issue=66&id=4
(retrieved 2011-03-28 at 12:08)

[PIOS]
PiOS: Detecting Privacy Leaks in iOS Applications
http://iseclab.org/papers/egele-ndss11.pdf
(retrieved 2011-01-19 at 12:06)

[PKCS1_PAD_ML]
Encrypting on iPhone using RSA for decryption in C#
Apple CDSA Mailing List
http://lists.apple.com/archives/Apple-cdsa/2009/Jul/msg00027.html
(retrieved 2011-02-15 at 18:28)

[PKCS7]
PKCS #7: Cryptographic Message Syntax Version 1.5
http://tools.ietf.org/rfc/rfc2315.txt
(retrieved 2010-12-16 at 00:11)

[PRIVACY_A]
Intego Security Memo: Hacker Tool Copies Personal Info from iPhones
http://blog.intego.com/2009/11/11/intego-security-memo-hacker-tool-copies-personal-info-from-iphones/
(retrieved 2011-04-05 at 16:28)

[PWN2OWN]
Announcing Pwn2Own 2011
http://dvlabs.tippingpoint.com/blog/2011/02/02/pwn2own-2011
(retrieved 2011-03-25 at 12:23)

[QA1710]
Technical Q&A QA1710
iOS Developer Library
http://developer.apple.com/library/ios/#qa/qa1710/_index.html
(retrieved 2011-01-21 at 15:03)

[RDAR_1199409]
Provide more details about PKCS #1 padding in reference documentation
Open Radar
http://openradar.appspot.com/radar?id=1199409
(retrieved 2011-06-02 at 14:49)

[RET_LIBC_1]
Getting around non-executable stack (and fix)
Solar Designer, 1997
http://seclists.org/bugtraq/1997/Aug/63
(retrieved 2011-05-15 at 18:15)

[RET_LIBC_2]
The advanced return-into-lib(c) exploits: PaX case study
Nergal
http://www.phrack.org/issues.html?issue=58&id=4&mode=txt
(retrieved 2011-05-15 at 18:16)

[RFC2617]
HTTP Authentication: Basic and Digest Access Authentication
http://www.ietf.org/rfc/rfc2617.txt
(retrieved 2011-03-08 at 21:11)

[RFC2898]
PKCS #5: Password-Based Cryptography Specification
Version 2.0
http://www.ietf.org/rfc/rfc2898.txt
(retrieved 2010-12-20 at 16:06)

[RFC3347]
Public-Key Cryptography Standards (PKCS) #1:
RSA Cryptography Specifications Version 2.1
http://tools.ietf.org/html/rfc3447
(retrieved 2011-02-16 at 16:53)

[RFC3986]
Uniform Resource Identifier (URI): Generic Syntax
http://tools.ietf.org/rfc/rfc3986.txt
(retrieved 2011-04-19 at 12:35)

[RFC4178]
The Simple and Protected Generic Security Service
Application Program Interface (GSS-API) Negotiation Mechanism
http://tools.ietf.org/html/rfc4178
(retrieved 2011-03-11 at 14:13)

[ROP1]
The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)
Hovav Shacham
Proceedings of CCS 2007, pages 552-561
ACM Press, 2007
http://cseweb.ucsd.edu/~hovav/papers/s07.html
(retrieved 2011-05-16 at 13:05)

[ROP2]
When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC
Erik Buchanan, Ryan Roemer, Hovav Shacham, Stefan Savage
Proceedings of CCS 2008, pages 27-38
ACM Press, 2008
http://cseweb.ucsd.edu/~hovav/papers/brss08.html
(retrieved 2011-05-16 at 13:07)

[ROP3]
Return Oriented Programming for the ARM Architecture
Tim Kornau
Diplomarbeit
Ruhr-Universität Bochum, 2009
http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf
(retrieved 2011-05-16 at 13:12)

[RS_REF]
Randomization Services Reference
Revision 2008-03-12
iOS Developer Library

[S_A_D]
Securing Application Data
WWDC 2010 Session 209
https://developer.apple.com/videos/wwdc/2010/index.php
(retrieved 2011-06-02 at 13:25)

[SANDBOX_INIT_MAN]
sandbox_init(3) Mac OS X Manual Page
http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man3/sandbox_init.3.html
(retrieved 2011-04-13 at 13:25)

[SEATBELT]
Seatbelt
The iPhone Wiki
http://iphonedevwiki.net/index.php/Seatbelt
(retrieved 2011-04-13 at 12:11)

[S_O]
Security Overview
Revision 2010-07-13
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Security/Conceptual/Security_Overview/Introduction/Introduction.html
(retrieved 2011-06-02 at 13:27)

[S_PROG_G]
Stream Programming Guide
Revision 2009-12-16
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Streams/Streams.html
(retrieved 2011-06-02 at 13:28)

[SC_G]
Secure Coding Guide
Revision 2010-02-12
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Security/Conceptual/SecureCodingGuide/Introduction.html
(retrieved 2011-06-02 at 13:29)

[SMS_FUZZING]
Fuzzing the Phone in your Phone
Collin Mulliner, Charlie Miller
Black Hat USA 2009
https://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf
(retrieved 2011-04-05 at 13:11)

[SO_PADDING_TYPES]
What is the difference between the different padding types on iOS?
Stack Overflow
http://stackoverflow.com/questions/5054036/what-is-the-difference-between-the-different-padding-types-on-ios
(retrieved 2011-03-16 at 13:16)

[SO_PROT_CLASS_KC]
Is it possible to update a Keychain item's kSecAttrAccessible value?
Stack Overflow
http://stackoverflow.com/questions/5369238/is-it-possible-to-update-a-keychain-items-ksecattraccessible-value
(retrieved 2011-03-21 at 10:25)

[SO_WEBVIEW]
How to display the Authentication Challenge in UIWebView?
http://stackoverflow.com/questions/1769888/how-to-display-the-authentication-challenge-in-uiwebview
(retrieved 2011-03-09 at 12:31)

[SPYPHONE]
SpyPhone
Source code on GitHub
https://github.com/nst/SpyPhone
(retrieved 2011-03-28 at 11:11)

[SSLv2]
SSL 0.2 Protocol Specification
http://www.mozilla.org/projects/security/pki/nss/ssl/draft02.html
(retrieved 2011-03-16 at 15:30)

[SSLv3]
SSL 3.0 Specification
http://www.freesoft.org/CIE/Topics/ssl-draft/3-SPEC.HTM
(retrieved 2011-03-16 at 15:28)

[ST_REF]
Secure Transport Reference
Revision 2004-08-31
Mac OS X Developer Library
http://developer.apple.com/library/mac/#documentation/Security/Reference/secureTransportRef/Reference/reference.html
(retrieved 2011-04-21 at 16:03)

[STACK_SMASH]
Smashing The Stack For Fun And Profit
Aleph One
Phrack, Volume 7, Issue 49, File 14 of 16
http://www.phrack.org/issues.html?issue=49&id=14&mode=txt
(retrieved 2011-05-16 at 21:10)

[STORM8]
Backdoor in top iPhone games stole user data, suit claims Storm8's iSpy
http://www.theregister.co.uk/2009/11/06/iphone_games_storm8_lawsuit//
(retrieved 2011-04-05 at 15:00)

[TLSv1]
The Transport Layer Security (TLS) Protocol Version 1.2
http://tools.ietf.org/rfc/rfc5246.txt
(retrieved 2011-03-16 at 15:19)

[TOLL_FREE_BRIDGING]
Friday Q&A 2010-01-22: Toll Free Bridging Internals
http://www.mikeash.com/pyblog/friday-qa-2010-01-22-toll-free-bridging-internals.html
(retrieved 2011-03-16 at 15:57)

[TRUSTZONE]
ARM TrustZone
http://www.arm.com/products/processors/technologies/trustzone.php
(retrieved 2011-04-05 at 11:50)

[URLLS_P_G]
URL Loading System Programming Guide
Revision 2010-09-01
iOS Developer Library
http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/URLLoadingSystem/URLLoadingSystem.html
(retrieved 2011-04-21 at 14:10)

[WHY_JAILB]
Why Jailbreak iPhone
http://thebigboss.org/why-jailbreak-iphone
(retrieved 2011-03-25 at 12:31)

[WWDC2010]
WWDC 2010 Session Videos
http://developer.apple.com/videos/wwdc/2010/
(retrieved 2011-03-20 at 18:36)

[ZDZIARSKI_FORENSIC]
iPhone Forensics - Recovering Evidence, Personal Data, and Corporate Assets
Jonathan Zdziarski
O'Reilly Media, 2008