iOS Encryption Systems
Deploying iOS Devices in Security-critical Environments

Peter Teufl, Thomas Zefferer, Christof Stromberger and Christoph Hechenblaikner
Institute for Applied Information Processing and Communications,
Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria

Keywords: Smartphone Security, iOS, Encryption, Bring-Your-Own-Device, Mobile Device Management, Mobile Devices, Risk Analysis, Security Analysis.

Abstract: The high usability of smartphones and tablets is embraced by consumers as well as the private and public sector. However, especially in the non-consumer area the factor security plays a decisive role for the platform selection process. All of the current companies within the mobile device sector added a wide range of security features to the initially consumer-oriented devices (Apple, Google, Microsoft), or have dealt with security as a core feature from the beginning (RIM, now Blackerry). One of the key security features for protecting data on the device or in device backups are the encryption systems, which are deployed in most current devices. However, even under the assumption that the systems are implemented correctly, there is a wide range of parameters, specific use cases, and weaknesses that need to be considered by the security officer. As the first part in a series of papers, this work analyzes the deployment of the iOS platform and its encryption systems within a security-critical context from a security officer's perspective. Thereby, the different sub-systems, the influence of the developer, the applied configuration, and the susceptibility to various attacks are analyzed in detail. Based on these results we present a workflow that supports the security officer in analyzing the security of an iOS device and the installed applications within a security-critical context. This workflow is supported by various tools that were either developed by ourselves or are available from other sources.

1 INTRODUCTION

The recent success story of smartphones and tablets – further referred to as mobile devices – was mainly fueled by user-friendly and consumer-oriented devices introduced by Apple and Google in 2007 and 2008. For the past years, Apple's iOS and Google's Android have been dominating the market. Both companies have introduced a wide range of security related features to improve the device security for consumer and business applications. Recently, the market power of iOS and Android has been challenged by a new Windows Phone 8 release and a new version of the BlackBerry platform. For all major smartphone platforms, encryption represents a core feature that is advertised for its strong security. However, the deployed encryption systems differ in various security related aspects. For instance, different platforms rely on different approaches to encrypt data (file based encryption vs. file-system based encryption) and implement different methods to derive required encryption keys from user input (e.g. PIN or passcodes[1]). Furthermore, different platforms offer developers, administrators and end users different options to use and configure provided encryption features.

Thus – if the platform should be deployed in a security-critical context – a security analysis for a given mobile device platform must consider a wide range of different security related aspects. Even under the assumption that the encryption systems and algorithms are implemented correctly, there are many other aspects that can render the systems ineffective. Examples for such higher level aspects are parameters related to system configuration, application development or the utilization of backup systems. In fact those higher level aspects play a vital role in a security analysis that needs to be conducted by a security officer prior to deploying a platform within a security-critical context.

[1]Subsequently, we will refer to PIN and passcodes only with the term passcode.

170 Teufl P., Zefferer T., Stromberger C. and Hechenblaikner C..
iOS Encryption Systems - Deploying iOS Devices in Security-critical Environments.
DOI: 10.5220/0004526201700182
In *Proceedings of the 10th International Conference on Security and Cryptography* (SECRYPT-2013), pages 170-182
ISBN: 978-989-8565-73-0
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

As the first part in a series of security analyses emphasizing those high-level aspects, this work analyzes the iOS encryption and backup systems. After giving a detailed description of the respective systems and the possible attacks, we present a workflow that can be followed by a security officer in order to ensure that the various encryption systems provide the intended protection for security-critical data. This workflow is supported by several tools that were either developed by ourselves or other parties.

2 RELATED WORK

The details on the encryption systems presented in this work, the conducted security analysis and the proposed workflow are based on a wide range of information sources and projects that were started in late 2011. These projects are related to the authors' mobile security consulting work within the public sector and the development of the *Secure Send*[2] application which relies on the iOS encryption systems to securely store data, and exchange this data via CMS-based containers[3] via email or Dropbox. The details about the deployed encryption systems have been extracted from official documents, from sources related to the iOS jailbreak and forensics communities and scientific publications. Especially within the consulting context the authors have extracted many practical requirements in security-critical environments. These requirements and the thereby generated questions form the basis for the proposed workflow that is presented in this work. The development of the *Secure Send* application helped to understand the iOS security mechanisms and the influence of the developer within a security-critical context. The remainder of this section will refer to the information sources that helped us to gain an in-depth understanding of the iOS encryption systems.

For our contribution, related work dealing with encryption systems for mobile devices is of special interest. Indeed, various authors have approached this topic from different perspectives so far. The relevance of encryption solutions on mobile devices and possible implications on jurisdiction have been discussed in (Paul et al., 2011). Proprietary encryption solutions for smartphone platforms have for instance been proposed in (Shurui et al., 2010) and (Chen and Ku, 2009). Interestingly, most related work on encryption systems on mobile devices does not focus on the platform's encryption systems, but on the development of proprietary solutions. Such solutions are often utilized within container applications that play an important role within the currently popular Bring-Your-Own-Device (BYOD) scenario. Due to the private ownership of the devices in such a scenario, those devices and their configuration typically cannot be controlled via mobile-device-management (MDM) solutions that activate and configure the platform's security mechanisms. Due to the lack of these management facilities and the thereby associated activation and configuration of the platform's security mechanisms, unmanaged devices typically cannot be deployed within a security-critical context.

There is only one white paper available from Apple that addresses the iOS encryption systems (Apple, 2012). Unfortunately, this document only presents a superficial overview and lacks many important details. The most detailed analyses of the iOS encryption and backup systems are available from the forensics community, where the recovery of the stored data plays a crucial role. The presentation by (Belenko and Sklyarov, 2011) describes the evolution of forensics from the first version of iOS to iOS 5.0 and thereby highlights the weaknesses of the initial versions that were later addressed by introducing various protection systems. The same topic is also addressed by others, such as (Hoog and Strzempka, 2011). Apart from the forensic oriented approach, many sources cover the details and the weaknesses of the iOS encryption systems, and iOS security in general. Early work on the security of the iOS encryption systems is presented by (Pandya, 2008). The iOS encryption systems, the involved keys and concepts are addressed by the following sources: A project[4] dedicated to the iOS data protection system describes the keys involved in the encryption systems (Bedrune and Sigwald, 2011). A general security analysis for iOS 4 is presented by (Zovi, 2011). Although, this iOS version is outdated, the analysis still provides important information, due to the introduction of the *Data Protection* system in iOS 4. A very important aspect of this *Data Protection* system is discussed in the iOS *KeyChain* FAQ (Heider and Khayari, 2012). This document addresses the problem of choosing protection classes that do not adequately protect *KeyChain* entries such as passwords or symmetric and asymmetric key material. Another relevant source is the iPhone Wiki[5] covers a wide range of iOS security related aspects.

When analyzing the security of the stored data on iOS devices, one also needs to consider the iTunes and iCloud backup systems that are used to backup application data on a local device or within the Ap-

[2]https://itunes.apple.com/app/id560086616
[3]http://tools.ietf.org/html/rfc5652
[4]http://code.google.com/p/iphone-dataprotection/
[5]http://theiphonewiki.com

ple iCloud solution. Details about the information stored in iTunes backups can be found on a dedicated webpage on the previously mentioned iPhone Wiki[6]. The aspects related to forensics are covered by various other sources that describe the backup process, the difference between standard and encrypted backups and the implications for a forensic analysis (Infosec Institute, 2012b), (Infosec Institute, 2012a).

3 THREATS AND ASSUMPTIONS

The presented security analysis is based on the scenario that a security officer within the public or private sector is confronted with the task of deploying the iOS platform within a security-critical context. Thereby, mobile devices and the required applications are utilized to process and store security-critical data.

The main threats in relation to mobile device security are related to the possibility of an attacker gaining physical access to the device (**theft**), or installing **malware** on the device. The main emphasis of this paper is placed on the first threat – **theft**. Malware is only considered in the context of jailbreaking where a piece of software that gains root access to the device by exploiting a security flaw, is deliberately installed by the user or attacker.

Theft. The following assumptions for the conducted analysis are defined within the context of an attacker stealing a device containing security-critical data. **First**, our assessments are based on the assumption that the iOS encryption systems are implemented correctly. The goal of this assessment is to analyze weaknesses located on a higher level, such as misconfigurations, weak passwords, limits of key derivation functions or wrong assumptions in relation to the encryption scope (e.g., files vs. file-system). **Second**, we assume that a passcode-locked iOS device is stolen by an attacker who is an expert with in-depth knowledge about the deployed encryption systems and their weaknesses. This scenario is similar to the one faced by a forensic expert who needs to analyze the data stored on an iOS device. In this context we also assume that the attacker employs jailbreaking tools. This is the only type of malware[7] that will be considered in the conducted analysis. **Finally**, the analysis is mainly focused on managed devices that are configured via mobile-device-management solutions. The rationale behind this assumption is the security-critical deployment scenario that forms the basis for this analysis. In such an environment, the currently popular Bring-Your-Own-Device scenario can only play a minor role due to the user's sole control of the security related device configuration (setting passcodes, activating/deactivating encryption systems, backups). In such a scenario, the developers/administrators cannot rely on platform security and encryption systems, but must implement their own protection mechanisms (e.g., container applications[8]), which typically cannot achieve the same level of security as it is provided by system functionality.

Malware. Although malware related attacks with the exception of jailbreaks are not considered in the remainder of this work, a short overview on the different malware categories and their implications are discussed in this section. Also, the reasons for excluding these kind of attacks in this analysis are explained.

We refer to malware as a piece of software that is installed on the user's device, even if the attacker does not have physical access to the device. The installation could either be initiated by the user who intentionally installs an application that contains hidden malicious code, or by an action that triggers the injection of malicious code via a security flaw in the device's operating system or applications. An example would be a critical browser vulnerability that is exploited when the user visits the attacker's webpage hosting the exploit code and the malicious payload injected via the security flaw.

We need to differentiate between two types of malicious applications: **First**, malware that exploits critical system vulnerabilities to gain root access to the device, and **second**, malware that only relies on the standard platform APIs. This malware could either extract the protected data from the victim's phone, or gain additional information that helps the attacker to break the encryption systems, when physical access to the device is gained at a later time.

Unfortunately, when a malicious application of the first category gains root privileges on the targeted device, none of the existing encryption systems can protect the stored data. Malicious code executed with root privileges can inspect or modify arbitrary aspects of the device's operating system and thus in one way or another gain access to the protected data. Examples for such attacks would be the deployment of keyloggers, the extraction of encryption keys from the

[6]http://theiphonewiki.com/wiki/ITunes_Backup

[7]Jailbreak software might not be considered as malicous, when deliberately installed by the user. Still, this type of software needs to exploit a security flaw in order to gain root access. Therefore, there is a high similarity to malware.

[8]Such as the solutions by Good Technology and Excitor. http://www1.good.com/mobility-management-solutions/bring-your-own-device and http://www.excitor.com

device's memory, or accessing the data while it is decrypted (e.g. when the user unlocks the phone). This does not mean that such attacks do not need to be considered. However, the defence against such attacks is already partly covered by a tight security configuration of the device. Also, the protection mechanisms for such attacks need to be implemented on a much lower level that is out of scope of a security officer's sphere of influence.

The second malware-category relies on the standard functionality provided by the platform's APIs and thus the capabilities can be compared to those of standard applications. Although, such malware is not capable of gaining direct access to the encrypted data, it could gather information for a later carried out physical attack. Examples are phishing passcodes, extracting information about passcode complexity, or any other side-channel information that might help the attacker to gain access to the protected information. Due to wide range of possible ways to gain additional information, this second malware category is considered as out of scope here, but will be addressed in future work.

4 iOS ENCRYPTION SYSTEMS

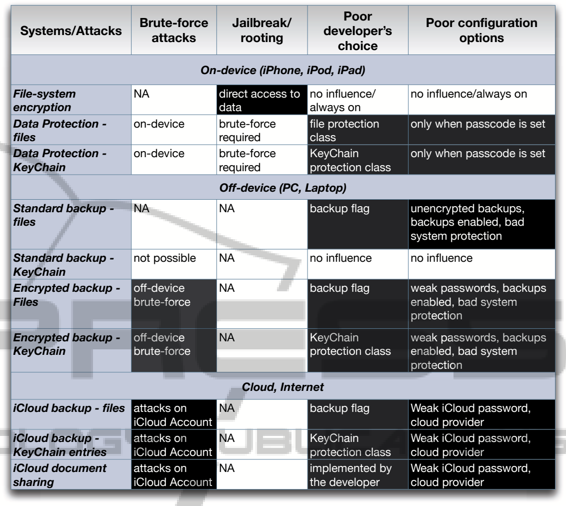
The iOS encryption system is based on multiple on-device, backup and cloud related sub-systems that have different purposes and protection mechanisms: the always-on *file-system encryption* system, the *Data Protection* system for files and credentials (*KeyChain*), the encryption systems for iTunes/iCloud backups, and finally the iCloud document-sharing mechanism. Although, these systems offer a reasonable level of protection, they still need to be deployed and configured correctly. Mistakes made by the developer or the security officer can easily make the systems ineffective.

In the following analysis the respective system details, the involved key derivation functions and issues in relation to configuration and development are considered for each iOS sub-system. The gained results are summarized in Table 1.

4.1 File-system Encryption

The file-system encryption system is available since iOS 3 and the iPhone 3GS. The encryption system relies on a hardware AES 256 crypto engine (Apple, 2012) that is present on every iOS device – further referred to as *secure element*. This chip securely stores a unique device (*UID*) key that is used to generate and protect all further encryption keys. The file-system

Table 1: Attacks on the various encryption systems: *White* indicates no or only minor problems that can be mitigated. *Dark-grey* represents more critical issues that definitely need to be considered. *Black* represents critical security problems, that are not solved by the respective system and need to be dealt with, either by deactivating the system or relying on other systems. *NA* indicates that an attack is not applicable/relevant for the respective subsystem.



Systems/Attacks	Brute-force attacks	Jailbreak/ rooting	Poor developer's choice	Poor configuration options
On-device (iPhone, iPod, iPad)				
File-system encryption	NA	direct access to data	no influence/ always on	no influence/always on
Data Protection - files	on-device	brute-force required	file protection class	only when passcode is set
Data Protection - KeyChain	on-device	brute-force required	KeyChain protection class	only when passcode is set
Off-device (PC, Laptop)				
Standard backup - files	NA	NA	backup flag	unencrypted backups, backups enabled, bad system protection
Standard backup - KeyChain	not possible	NA	no influence	no influence
Encrypted backup - Files	off-device brute-force	NA	backup flag	weak passwords, backups enabled, bad system protection
Encrypted backup - KeyChain	off-device brute-force	NA	KeyChain protection class	weak passwords, backups enabled, bad system protection
Cloud, Internet				
iCloud backup - files	attacks on iCloud Account	NA	backup flag	Weak iCloud password, cloud provider
iCloud backup - KeyChain entries	attacks on iCloud Account	NA	KeyChain protection class	Weak iCloud password, cloud provider
iCloud document sharing	attacks on iCloud Account	NA	implemented by the developer	Weak iCloud password, cloud provider

based encryption system is always-on and cannot be configured neither by the developer nor the security officer/user. Unfortunately, not including the user's passcode in the protection system is a major weakness that can be exploited by an attacker. Therefore, the file-system encryption system is primarily used to provide basic protection and to implement the quick-remote wipe functionality. Instead of going to the lengthy process of deleting the stored data, only the associated encryption keys are wiped, which cryptographically erases the stored data.

4.1.1 Details

An overview about the AES keys involved in this system and their protection is provided in Figure 1. Each iOS device has a unique AES key (*UID* key) that is stored within the *secure element*. This key is used to derive[9] the AES keys labelled as 0x89B and 0x835. These keys are used to wrap the *EMF* (file-system master encryption key) and *Dkey* (device key) keys, which are stored in the *effaceable storage* area of the flash memory. This flash memory area can be wiped very quickly, and thus offers a robust remote-wiping functionality which deletes the file-system encryption keys instead of the whole file-system data. The *EMF* key is used as a master key for the file-system encryp-

[9]Both keys are generated by encrypting different constants with the UID key.

tion process. The *Dkey* plays a role within the later explained *Data Protection* system. For now, it is only relevant that both keys are indirectly protected with the *UID* key without involving the user's passcode. The employment of a *secure element* eliminates the possibility to access the data within file-system images that are gained by cloning or directly extracting the storage device. In other words, any attack on the file-system must involve physical access to the device.

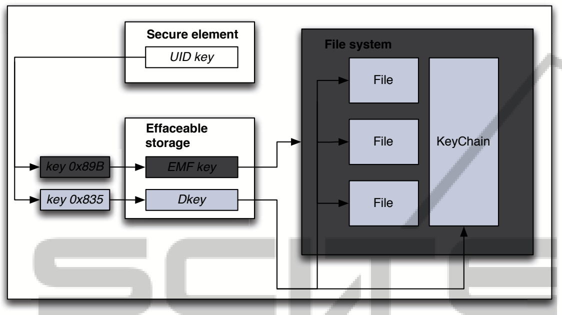


Figure 1: A simplified representation of the AES encryption keys that are involved in the file-system encryption. The most important aspect is that only the *secure element* is involved in the protection of these keys.

4.1.2 Key Derivation

Strictly speaking the notation *"key derivation"* is not adequate for an description on how the system generates and protects the described keys. The reason is, that the *UID* key within the *secure element* is used to create the *0x89B* and *0x835* keys, which are then used to protect the *EMF* and *Dkey* keys. This protection hierarchy does not involve the user's passcode and its security is only based on the UID key that is stored in the *secure element* and cannot be extracted. The aspect of not including a user supplied passcode is considered as the main weakness of the file-system based encryption system.

4.1.3 Configuration/Developer

The developers, users or administrators cannot influence this encryption system, since it is activated per default on every device and cannot be deactivated. However, all of the involved developers, users and administrators need to be aware of the limited level of protection this system provides when deploying security-critical applications.

4.1.4 Attacks

The system can easily be attacked via **Jailbreaking/Rooting**, which concentrates on the weakest point of the file-system based encryption system: The protection of the *EMF* and *Dkey* keys for encrypting the file-system is not based on the *passcode* of the user, but relies only on the integrated *secure element*. Thus, even when a passcode is set on the device, the application of a jailbreak enables the attacker to gain root access to the operating system, which decrypts the data with the *EMF* and *Dkey* keys. Thus, an attacker gains access to all data without knowing any of the deployed encryption keys. Jailbreaks are available for almost all iOS versions and devices[10]. However, under our assumption that an attacker gains access to a passcode-locked device, only those jailbreaks that do not rely on a task carried out by the user (e.g. accessing a webpage that exploits a critical security vulnerability in the Web browser, or installing a malicious application), can be used for the attack[11]. These jailbreaks are referred to as tethered jailbreaks and typically rely on security flaws in the iOS boot process.

Since the encryption system is always-on and does not provide any configuration options, it is not susceptible to any misconfigurations made by the developer (**poor developer's choice**) or the user/administrator (**poor configuration option**). Any attacks on the **key derivation function** itself are not considered, due to the fact that access to the data is already gained when a jailbreak is successfully applied to the system.

4.2 Data Protection System – Files

The *Data Protection* system was introduced with iOS 4 and offers an additional layer of protection for files and *KeyChain* entries stored on an iOS device. The main advantage of this system – when compared to the file-system based encryption system – is the *key derivation function* (*KDF*) that provides the functionality of deriving an encryption key based on the user's *passcode* and the *UID* key stored in the *secure element*. The encryption keys gained by the *KDF* are then used to wrap further keys that are used to protect single files and credentials according to the protection class assigned by the developer. Although, the system offers a quite good level of protection for the respective application files, it is also quite complex and requires in-depth knowledge of the developer.

This section describes the *Data Protection* system in the context of file protection. The protection of credentials stored in the iOS *KeyChain* is another feature of the *Data Protection* system. However, due to some important differences the *KeyChain* protection will be explained separately in Section 4.3.

[10]A good overview is given here: http://www.apfelzone.at/jailbreak-ubersicht/

[11]Here, malware that exploits critical security flaws might play a role without the need for the attacker to have physical access to the device.

4.2.1 Details

A detailed overview over the *Data Protection* system is given in Figure 2. The system is based on various protection classes that need to be defined by the developer for stored files and credentials. Each protection class has a specific class key that is used to encrypt/decrypt the specific file encryption keys. Thereby, each file has a unique file-encryption key (indicated as *cprotect key* in Figure 2). The protection classes define when the class keys are kept in memory for encrypting and decrypting the file encryption keys and thus the respective files. Thereby, each file is protected by a unique file encryption key. The four available data (file) protection classes are *NSFileProtection*{*None, Complete, UntilFirstUserAuthentication, CompleteUnlessOpen*}. Thereby, *None* indicates that the file is only protected via the file-system based encryption (keys based on the *EMF* key) and the *Dkey* key, which acts as class key for this protection class.

Complete means that the class key and the file-encryption keys of the protected files are removed from memory whenever the device is locked. *UntilFirstUserAuthentication* decrypts the respective class key when the passcode is entered the first time after booting the device. The decrypted keys are then kept in device memory until the next shutdown. Finally, *CompleteUnlessOpen* is used to write data to files that are open when the device is locked, and in addition offers a system based on asymmetric cryptography that is used for encrypting data which is received while the device is locked (e.g. emails).

4.2.2 Key Derivation

The *Data Protection* system employs the standardized *Password Based Key Derivation Function 2* (PBKDF2) (Apple, 2012), which is specified in PKCS#5 (Kaliski, 2000). The user's passcode is tangled with the *UID* key stored in the *secure element* and combined with a salt (not shown in Figure 2). The resulting value is then used as input for the key derivation function with a high iteration count. The gained key (*passcode key*) is used to encrypt and decrypt the aforementioned protection class keys, which are in turn used to protect the actual file encryption keys.

4.2.3 Developer

The *Data Protection* system is only effective when the developer has detailed knowledge about iOS security and carefully selects the appropriate protection classes for the data that needs to be protected. Especially, the protection class *NSFileProtectionNone* does not offer any protection beyond the standard file-system based encryption system and thus must *NOT* be considered as an option for security-critical applications.

4.2.4 Configuration

The administrator's primary influence on this system is the enforcement of specific *passcode* properties that are mandatory for the user. By setting a *passcode*, the *Data Protection* system is activated and protects the files marked with the appropriate protection classes. For managed devices, there exists a wide range of rules that define the password complexity requirements.

One of the major problem for expert users or administrators is that there is no simple way to determine the protection classes that were selected by the developer for the application files. However, this information is vital for selecting applications that are appropriate for handling and storing security-critical data.

4.2.5 Attacks

When **Rooting/Jailbreak** an iOS device, which is protected by the *Data Protection* system, the attacker still has access to the file-system. However, the files and credentials that use the correct protection classes (other than *None* for files) cannot be decrypted without knowing the *passcode*. Under the assumption, that this *passcode* is not known to the attacker the only remaining option for the attacker is the application of a **brute-force attack**. However, the presence of the *secure element* and the PBKDF2 key derivation function significantly slows down the brute-force attack. Due to the high iteration count of PBKDF2, the derivation of the *passcode key* from the *passcode* takes roughly 80 ms (Apple, 2012). This delay can be used to calculate the worst-case brute-force attack times when choosing a password. Also, and probably even more important, the brute-force attack must be carried out on the device, because the *secure element* is also involved in the key derivation process. This eliminates the possibility to speed up the attack by deploying an off-device attack capable of using a high level of computational resources. The described attack is implemented by a forensic toolkit offered by the UK based company Elcomsoft[12].

Although, the combination of the *secure element* and the *passcode* for key derivation offers a reasonable level of security, the overall level of security strongly depends on the developer as well as the

[12]http://www.elcomsoft.co.uk/eift.html

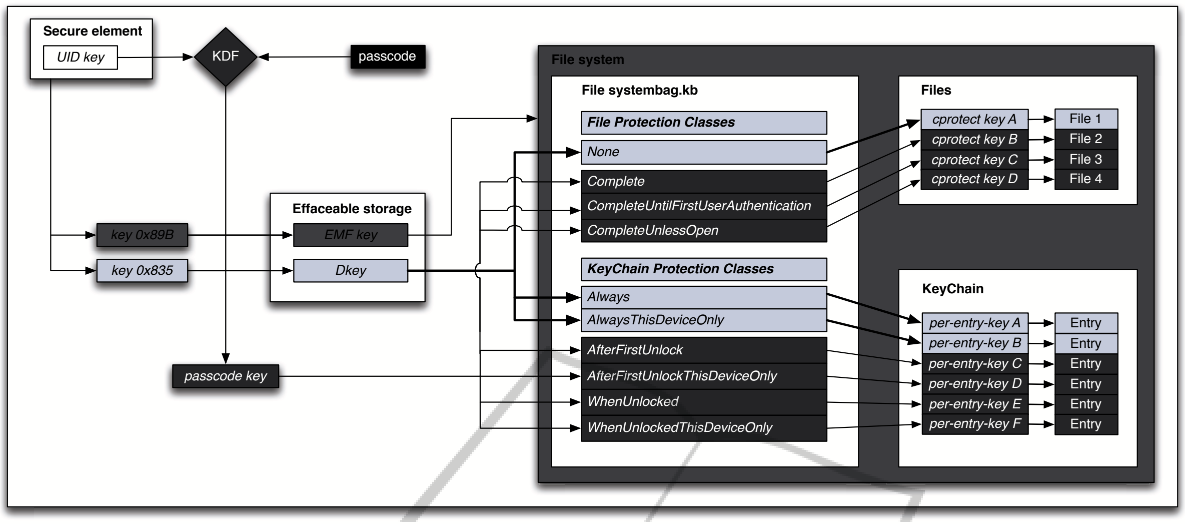


Figure 2: iOS *Data Protection* system for the protection of individual files and credentials. The encryption behaviour is defined by the developer via different protection classes. The most important aspect is that the key derivation function (*KDF*) that generates the *Passcode key* for the protection of the protection class keys depends on the *UID* key within the *secure element* as well the *Passcode* of the user.

user/administrator. **Poor configuration options** are especially critical in the unmanaged scenario where the user selects the passcode. In this case, the attacker can base an attack on the *Data Protection* system on the following assumptions: **(1)** Due to usability issues related to passcode length and complexity, the user will typically not use a passcode or select a rather short one. **(2)** Since the default setting for passcode locks on an iOS device uses 4-character numerical PIN codes (also indicated by the type of deployed lock-screen) a successful brute-force attack is quite likely.

Apart from the configuration issues, **poor developer choices** have a major influence on the security of the application files. When the wrong classes (especially *NSFileProtectionNone*) are chosen by the developer, the security is reduced to that of the file-system based encryption, which can be circumvented by applying **Jailbreaking/Rooting**. This opens a critical security issue, that could easily be exploited by an attacker. Assuming an application (e.g. the Apple mail) application uses the right protection classes, further assuming a user opens an attachment (e.g. a PDF file) in an external application that employs the *NSFileProtectionNone* protection class for storing the file: Then an attacker does not need to apply a brute-force attack on the passcode in order to get access to the email with the PDF document. Instead, the attacker can simply apply a jailbreak and extract the file form the external application that was used to view/edit the PDF file. Unfortunately, iOS does not provide any means for the user/administrator to inspect the protection classes used for the application files.

4.3 Data Protection System – KeyChain

The *Data Protection* system is also available to protect data such as passwords or key material stored in the iOS *KeyChain*. The main differences to the file-based *Data Protection* system are the available protection classes and the possibility to use protection classes that allow or disallow the transfer to other devices.

4.3.1 Details

iOS provides the functionality of storing credentials such as private keys, passwords and certificates in the iOS *KeyChain*. The functionality of the offered protection classes *kSecAttrAccessible*{*Always, WhenUnlocked, AfterFirstUnlock*} corresponds to the first three file *Data Protection* classes (*None*, *Complete*, *UntilFirstUserAuthentication*). However, there is no class that is equivalent to the *CompleteUnlessOpen* file protection class. An important difference to the file-based classes is that the *KeyChain* protection classes also exist in a *ThisDeviceOnly* version which indicates that the *KeyChain* data cannot be transferred off-device via iTunes or iCloud backups (covered in detail in Sections 4.4 and 4.5).

4.3.2 Key Derivation

The *KDF* for the key used to protect the respective class keys is based on the same function as deployed within the file-based *Data Protection* system (Section 4.2.2 and Figure 2).

4.3.3 Configuration/Developer

The influence of the configuration options are equal to those of the file-based protection classes. Except for some slight differences, this also applies to the developer: For the *KeyChain* protection classes the *Always, AlwaysThisDeviceOnly* classes are considered as dangerous, because their deployment has the same implications as the deployment of the *None* protection class. There is no class, which is equivalent to the *CompleteUnlessOpen* file protection class. One difference that needs to be considered is that the developer indicates whether a protected credential is transferable to another device. This option is not available for the file protection classes. For further details and implications the reader is referred to the backup-related Sections 4.4 and 4.5.

4.3.4 Attacks

Regarding the attacks, very similar conclusions as for the file-based *Data Protection* system can be drawn: The classes *Always* and *AlwaysThisDeviceOnly* are susceptible to **Rooting/Jailbreaking** attacks, which corresponds to the implications of the *None* file protection class. The problem is extensively discussed in (Heider and Khayari, 2012). The considerations for attacks based on **poor configuration/developer choices** are equivalent to those of the file protection classes.

4.4 Backup on iTunes

Although the *backup encryption system* is technically not a part of the mobile device, it still plays an important role for the security of data and credentials. On iOS, the backup can either be made via an iTunes installation, or the Apple based iCloud solution. Thereby, the iTunes backup can either be stored in plain text or be encrypted with a key derived from a *password*. The differences are depicted in Figure 3.

4.4.1 Details – Standard Backups

The files on an iOS device, that are marked by the developer for backup (which is the default setting), are stored by iTunes (via USB cable or WiFi) in plain text. Since the files on the device are encrypted via the *Data Protection* system, the iTunes access to those files is not possible without the respective keys. On the iOS device the *system keybag* contains all the protection class keys, that are required to wrap the respective file/credential encryption keys. When iTunes is connected to the iOS device for the first time, an *escrow keybag* is generated on the backup device. This keybag contains the protection class keys of the *system keybag* and is encrypted via a randomly generated key that is stored on the iOS device. This key is stored in the *KeyChain* with the attribute *kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly*, which means that the key is only available when the *passcode* is entered correctly after system boot up, and the *ThisDeviceOnly* part indicates that it cannot be transferred to other devices. Whenever iTunes copies the files from the iOS device, this key is used to decrypt the *escrow keybag* in iTunes and thereby the protection class keys. iTunes is now capable of decrypting the files on the iOS device and transferring them to the backup location.

The *KeyChain* entries of the iOS device are also transferred to the iTunes backup. However, and this is the important part, they remain encrypted with the device's protection class keys, and thus cannot be decrypted without physical access to the device. This leads to the somehow confusing aspect that the *KeyChain* entries are better protected in standard iTunes backups than they are in encrypted iTunes backups.

4.4.2 Details – Encrypted Backups

Encrypted backups require the user to define a *backup password* that is used to derive an encryption key, which is then used to encrypt the files and the iOS *KeyChain*. There are two important aspects that need to be considered here: **(1)** the files are stored encrypted, which offers a much higher protection level than within the standard backups. **(2)** the *system keybag* is encrypted with the *backup key* that is derived from the user's *backup password*. This enables the transfer of *KeyChain* entries to other devices, which is not possible in standard backups due to the encryption with a device-bound key.

4.4.3 Key Derivation – Standard Backups

Since the files are stored in plain text on the device where iTunes is installed, there is no key derivation function involved. The *KeyChain* entries are encrypted via device-specific keys, which are not available off-device.

4.4.4 Key Derivation – Encrypted Backups

This key derivation is also based on the PBKDF2 function. However – and this is the most important

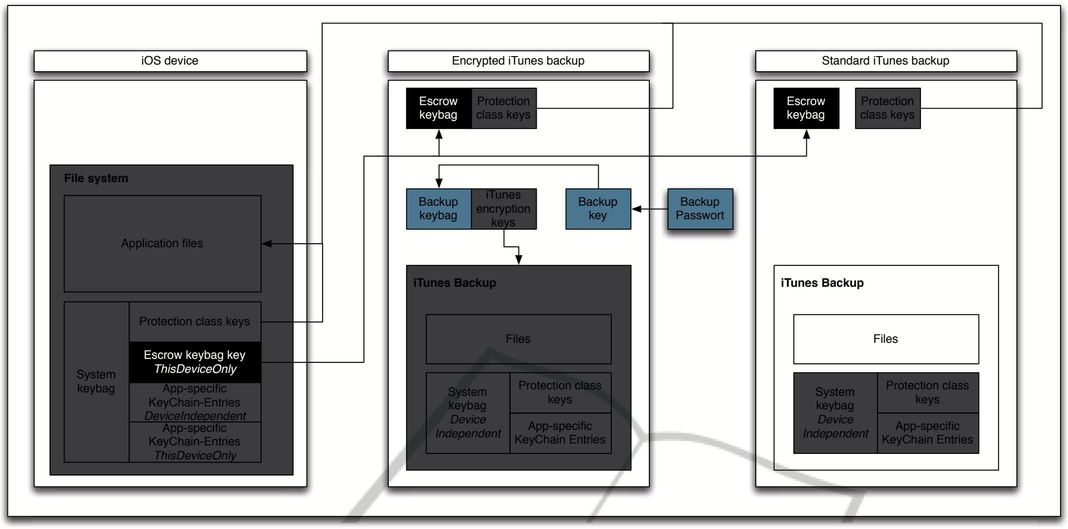


Figure 3: iTunes backup systems: Standard backups store the files in plain on the backup device. The credentials cannot be decrypted without physical access to the iOS device. Encrypted backups encrypt the files and the credentials with a password derived key.

difference to the *Data Protection* system *KDF* – due to the lack of a *secure element* on the PC/laptop where iTunes is installed, the key derivation subsystem is only based on the *backup password*, which allows for faster brute-force attacks.

4.4.5 Configuration

iTunes standard and encrypted backups can be activated and configured within the iTunes configuration. For managed devices, the administrator cannot deactivate iTunes backups if the user decides to use them. However, the administrator can enforce encrypted iTunes backups, which forces the user to define a backup password. Unfortunately, no rules regarding the password complexity can be defined, which leaves the password selection to the user.

4.4.6 Development

Files that are used within applications are marked for backup per default. The developer can exclude files that should not be included in iTunes/iCloud backups via the flags *NSURLIsExcludedFromBackupKey* or *kCFURLIsExcludedFromBackupKey*. Those files will never be included in unencrypted/encrypted and iTunes/iCloud backups. For security-critical data, the developer should typically deactivate the backup of the associates files.

4.4.7 Attacks

The backup on an iTunes device can be attacked via different techniques. **(1)** The developer might chose to include security-critical data within backups (**poor developer's choice**). If this is the case a wide range of attacks can be derived depending on the administrator's and user's choices regarding the available backup systems and their protection mechanism: **(2)** When the administrator allows non-encrypted standard backups (**poor configuration options**), the attacker can just copy the whole backup and get access to all of the files that are marked for backup by the installed iOS applications. **(3)** When encrypted backups are utilized, the attacker can carry out an **off-device-brute-force attack** on the encryption password. Due to the lack of a *secure element*, the attacker can use external processing resources to speed up this attack. Unfortunately, the administrator cannot define passcode policies that are mandatory for the user. This also applies for iCloud account passwords. **(4)** The user/administrator also decides whether the iCloud-based system is allowed. In case of its availability security issues with the iCloud account can be used for an attack path by an attacker to gain access to the backups (**poor configuration options**).

Two interesting aspects of the iTunes backup system are related to the credentials backup: **(1)** Only those credentials that are not marked via the *ThisDeviceOnly* protection classes are stored in the backup. **(2)** The standard backup offers a better protection for the stored credentials than the encrypted backup. In the first case, the encrypted iOS *KeyChain* is directly stored in the backup. Thereby, the encryption keys required to decrypt the *KeyChain* are only available on the iOS device itself. Thus, an attacker who has access to the unencrypted backup, can extract the stored files, but not the credentials. This also means, that the stored credentials cannot be transferred to another iOS device, due to the lack of the encryption keys required to decrypt the backed up *KeyChain*. In case of an encrypted backup the *KeyChain* is encrypted with encryption keys that are derived from the backup password. Thus, those credentials are susceptible to the described brute-force attacks.

4.5 iCloud

Two main iCloud features need to be considered – iCloud backups and iCloud document sharing. While the first feature is strongly related to iTunes backups and is activated by the respective backup flags, the second feature is used to share application data over different iOS and OS X devices. In contrast to the backup functionality, which needs to be explicitly deactivated by the developer via specific flags, the second feature needs to be deliberately implemented by the developer. In both cases, the security of the data stored on iCloud highly depends on the security of the associated iCloud/iTunes accounts. When an attacker gains access to an iCloud account the stored backup files and the shared documents can easily be restored on the attacker's device who then gains access to the security-critical data.

It is highly recommended that the iCloud functionality is deactivated by the MDM administrator. This recommendation is based on a wide range of security issues: A secure iCloud password needs to be chosen by the user. Here, similar issues as for the encrypted backups need to be considered. The quality of the iCloud password cannot be enforced by the administrator via MDM rules. Furthermore, iCloud and iTunes account security has been brought into the news in 2012, when an account of a journalist was successfully attacked by exploiting weaknesses in the call center policies for gaining access to accounts where the password was lost[13]. In a recent report various security issues in relation to the transferal of iTunes/iCloud credentials over unencrypted HTTP connections were highlighted (Goodin, 2013). Although these flaws have been fixed, iCloud and iTunes account security clearly not only depends on the chosen password, but on a wide range of other factors: Examples are social engineering issues, phishing attacks on passwords, the tight interconnection of different accounts, possible security issues within the iCloud service, and the issues associated to the fact that the cloud provider has access to the stored data (Foresman, 2012). Including these factors within a detailed risk-analysis and their possible mitigation is not possible due to the lack of information and the wide range of issues that need to be considered. Therefore, iCloud backups and iCloud document sharing should not be considered within security-critical applications.

[13]http://www.wired.com/gadgetlab/2012/08/apple-amazon-mat-honan-hacking/

5 SECURITY ANALYSIS – WORKFLOW

The presented workflow is designed under the assumption that a thorough risk analysis within the context of the deployment scenario has already been conducted, and the critical assets, threats and risk factors are already known. Also the mobile device platform – in this case iOS – has already been selected. Subsequently, based on the risk analysis and the existing policies, the iOS platform needs to be configured properly and the appropriate applications for handling the critical data need to be chosen. After selecting those applications and setting up a demonstration environment that already contains data to be protected[14], the proposed workflow is applied.

A simple example can be outlined as follows: Given a scenario, where a certain organisation needs to handle critical internal email communication on the mobile device platform: Then, for processing the attached files within these emails several applications are required by the users. Examples could be various document readers, PDF annotators or any other application that handles those attachments or the information contained in the emails. Based on the selection of third-party applications or even in-house developments, the security officer needs to analyze the overall security of the whole setup. The then applied workflow is shown in Figure 4, where the first step (*System Security Analysis*) is applied to the whole system and the subsequent steps need to be considered for the installed applications[15].

5.1 System Security Analysis

Although the overall security related configuration of the device is not covered in this workflow, there is one system security related aspect that needs to be addressed within the context of the *Data Protection* system for credentials. When communication systems, such as MS Exchange connections, VPN connections etc. are utilized in the deployment scenario,

[14]Meaning, that the applications have been used on the device and application files are already present.

[15]The various procedures only need to be applied once to cover all the installed applications.

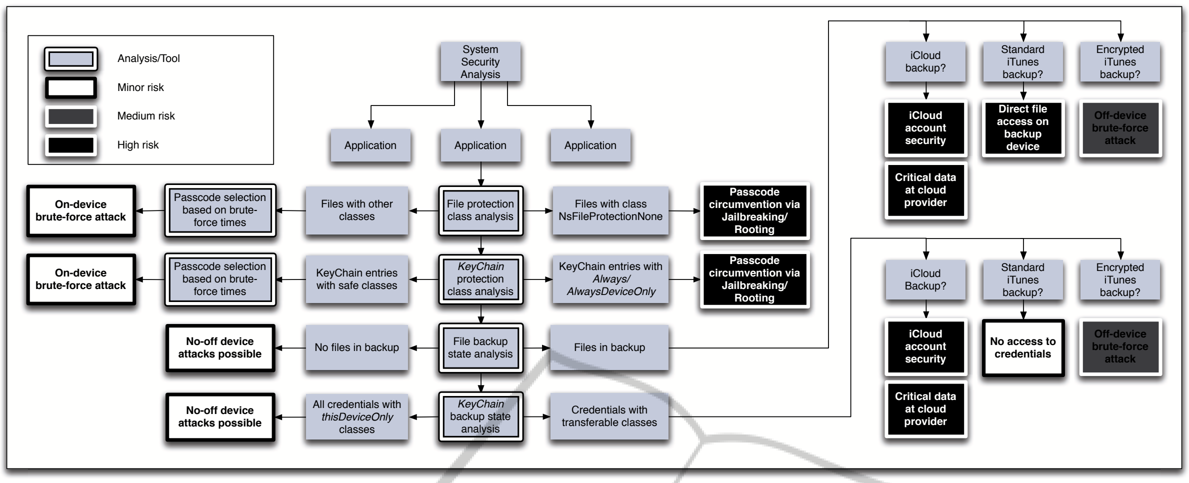

Analysis/Tool
Minor risk
Medium risk
High risk
System Security Analysis
Application
Application
Application
On-device brute-force attack
Passcode selection based on brute-force times
Files with other classes
File protection class analysis
Files with class NsFileProtectionNone
Passcode circumvention via Jailbreaking/ Rooting
On-device brute-force attack
Passcode selection based on brute-force times
KeyChain entries with safe classes
KeyChain protection class analysis
KeyChain entries with Always/ AlwaysDeviceOnly
Passcode circumvention via Jailbreaking/ Rooting
No-off device attacks possible
No files in backup
File backup state analysis
Files in backup
No-off device attacks possible
All credentials with thisDeviceOnly classes
KeyChain backup state analysis
Credentials with transferable classes
iCloud backup?
Standard iTunes backup?
Encrypted iTunes backup?
iCloud account security
Direct file access on backup device
Off-device brute-force attack
Critical data at cloud provider
iCloud Backup?
Standard iTunes backup?
Encrypted iTunes backup?
iCloud account security
No access to credentials
Off-device brute-force attack
Critical data at cloud provider

Figure 4: Workflow for analyzing the security of application files in the context of the iOS encryption systems.

their respective *KeyChain* entries (private keys, passwords etc.) need to be considered. As described in detail by (Heider and Khayari, 2012), several of these entries were marked by Apple via the problematic *Always, AlwaysThisDeviceOnly* classes and thus can be accessed after successfully applying a jailbreak (Section 4.3).

5.2 File Protection Class Analysis

In the first application-oriented step the protection classes of all files are analyzed. This is considered as the most important step, since the wrong selection of protection classes (especially *None*) defeats the purpose of the *Data Protection* system. Any application using this class for storing security-critical data must be rejected. Otherwise the **Jailbreaking/Rooting** attack can be applied by an attacker to gain physical access to the device and retrieve the desired data.

Tool – File Protection Class Analysis. To gather the file protection class information we present a simple Java based *Data Protection* tool[16] that analyzes the unencrypted metadata[17] of standard and encrypted iTunes backups. The protection class information of the files within these backups can be extracted from the gained metadata. The main limitation of the current version is that the protection classes of files, which are not contained in the backups, cannot be analyzed.

Tool – Passcode Selection based on Brute-force Times. If the application files use the appropriate protection classes, then the passcode is highly relevant for the security of the system. The length and the complexity of this passcode can be chosen according to the time necessary to carry out a brute-force attack. Due to the presence of the *secure element* and the high iteration count of the PBKDF2 key derivation function, the time to derive the key from a given password is known. Given the length of the password and the allowed characters, which can both be configured via various MDM policies, the worst-case time for a brute-force attack on the password can be calculated[18]. Based on these times, the risk analysis and the deployed security policy the appropriate password length and complexity can be selected and modelled via the respective MDM policies.

5.3 Keychain Protection Class Analysis

The extraction of the application-specific *KeyChain* entry protection classes has a similar relevance as for the file *Data Protection* classes. Within the *KeyChain* context the protection classes *Always* and *AlwaysThisDeviceOnly* need to be considered as dangerous due to the jailbreaking scenario. Apart from the classes of the third-party application credentials, one needs to consider the pre-defined classes of system applications (Section 5.1). Furthermore, the implications for iCloud and iTunes backups need to be considered for the transferable classes and device-only classes (*ThisDeviceOnly*). This will be discussed in the backup-related sections of this workflow.

[16]https://github.com/ciso/ios-dataprotection/

[17]By using and extending parts of the source code of the iPhoneStalker tool: http:// code.google.com/ p/ iphonestalker/

[18]A simple online spreadsheet is available at http:// goo.gl/pGijv

Tool – *KeyChain* Protection Class Analysis. Similar to the file protection classes, the protection classes of the *KeyChain* entries can also be extracted from the backups. This functionality is provided by the iphone-dataprotection backup script[19]. Since only the non-*ThisDeviceOnly* classes are transferred to the backup, the other *KeyChain* entries store on the iOS device cannot be analyzed with this tool.

Tool – Passcode Selection based on Brute-force Times. The same conclusions for the brute-force times in case of successful **jailbreaking/rooting** attacks can be drawn as for the file protection classes.

5.4 File Backup State Analysis

In the next step, the backup flags of the application files are evaluated. By knowing which application files are included in possible iTunes and iCloud backups, conclusions about the required iTunes device security or iCloud usage can be drawn. The information on the files within the backup is extracted via the aforementioned *Data Protection* tool.

Implications for Unencrypted iTunes Backups. An attacker who gains access to the iTunes device that stores the backup, can directly access all files that are included in the backup (**Direct file access on backup device**). When critical files are marked via the backup flags the unencrypted iTunes backup should be avoided, especially when the backup device might be exposed to non-secure environments.

Implications for Encrypted iTunes Backups. An attacker who gains access to the iTunes device that stores the backup cannot access the files directly due to their encrypted nature. However, an attacker can still mount a brute-force attack on the iTunes encryption password (**Off-device brute-force attack**). Since the involved key derivation function is not bound to a device, the attacker can use arbitrary external resources to speed up this process. Thus, a strong password needs to be selected to adequately protect the backup data. Although, encrypted backups can be enforced via MDM policies, it is not possible for the MDM administrator to define password properties, such as length, complexity etc. As soon as the iOS device is allowed to backup data via iTunes it is the user's responsibility to set an adequate password.

Implications for iCloud Backups. The files marked with the backup flags are also stored on iCloud backups, when this functionality is enabled by the MDM administrator. Due to a wide range of possible security issues and the fact that the MDM administrator cannot define security properties for the iCloud account password, it is highly recommended to deactivate iCloud backups in security-critical environments (**iCloud account security**, **Critical data at cloud provider**).

5.5 KeyChain Backup State Analysis

In contrast to the specific backup flags required for file backups, the credentials protection classes themselves indicate whether they can be transferred to a different device via an iCloud or iTunes backup. All *KeyChain* entry protection classes that do not contain the *ThisDeviceOnly* identifier, are included in backups. The protection classes of the *KeyChain* entries marked for backup can be extracted with the aforementioned iphone-dataprotection backup script.

Implications for Standard iTunes Backups. Credentials that are stored within standard iTunes backups are protected by an iOS device-bound key and thus cannot be decrypted without physical access to this device. This implies that none of these credentials can be transferred to other devices when restoring a standard iTunes backup. Interestingly, this also means that credentials in such a backup are better protected than those stored in an encrypted iTunes backup (**No access to credentials**).

Implications for Encrypted iTunes Backups. Within an encrypted iTunes backup the non-*ThisDeviceOnly KeyChain* entries are stored and encrypted via the backup password chosen by the user. Similar to the stored files these *KeyChain* entries are susceptible to **off-device brute-force attack**.

Implications for iCloud Backups. The same conclusions as for the files can be drawn. All credentials that are marked with the transferable protection classes, are included in the iCloud backup and thus can be accessed when the iCloud account password is known (**iCloud account security**, **Critical data at cloud provider**).

[19]http://code.google.com/p/iphone-dataprotection/source/browse/python_scripts/backup_tool.py. Usage: http://stackoverflow.com/questions/1498342/how-to-decrypt-an-encrypted-apple-itunes-iphone-backup

6 CONCLUSIONS AND FUTURE WORK

The existing heterogeneity in mobile device platforms and the associated security features present a challenge for administrators or security officers that need to deploy such devices in a security-critical context. Thereby, the encryption systems of a mobile device platform play a vital role in the protection of critical data, when an attacker gains physical access to the device. However, even under the assumption that the encryption systems are implemented correctly, their complexity within the high-level application creates many pitfalls that need to be considered by developers and administrators as well. In this paper, we have analyzed the highly sophisticated and complex iOS encryption systems from a security officer's perspective. As a result of this analysis a workflow is presented that analyzes the different aspects of the iOS encryption systems. To support this workflow various tools are presented that help the security officer to gain critical information which is not available within the standard iOS configuration tools. As future work, we intend to conduct the same analysis for the other major mobile device platforms Android, Blackberry and Windows Phone/RT. Furthermore, we intend to combine the various tools into an application that can easily be utilized and does not require root access to the iOS devices.

REFERENCES

Apple (2012). iOS Security. Technical Report May, Apple Inc.

Bedrune, J.-B. and Sigwald, J. (2011). iPhone data protection in depth. Technical report, Sogeti / ESEC.

Belenko, A. and Sklyarov, D. (2011). Evolution of iOS Data Protection and iPhone Forensics : from iPhone OS to iOS 5.

Chen, Y. C. Y. and Ku, W.-S. K. W.-S. (2009). Self-Encryption Scheme for Data Security in Mobile Devices.

Foresman, C. (2012). Apple holds the master decryption key when it comes to iCloud security, privacy, http://arstechnica.com/apple/2012/04/apple-holds-the-master-key-when-it-comes-to-icloud-security-privacy/.

Goodin, D. (2013). After leaving users exposed, Apple fully HTTPS-protects iOS App Store, http://arstechnica.com/security/2013/03/after-leaving-users-exposed-apple-finally-https-protects-ios-app-store/.

Heider, J. and Khayari, R. E. (2012). iOS Keychain Weakness FAQ - Further Information on iOS Password Protection.

Hoog, A. and Strzempka, K. (2011). *iPhone and iOS Forensics: Investigation, Analysis and Mobile Security for Apple iPhone, iPad and iOS Devices*. Syngress.

Infosec Institute (2012a). Forensic analysis of iPhone backups, http://www.exploit-db.com/wp-content/themes/exploit/docs/19767.pdf. Technical report.

Infosec Institute (2012b). iPhone Forensics Analysis of iOS 5 backups, http://resources.infosecinstitute.com/ios-5-backups-part-1/. Technical report, Infosec Institute.

Kaliski, B. (2000). PKCS #5: Password-Based Cryptography Specification Version 2.0.

Pandya, V. R. (2008). IPHONE SECURITY ANALYSIS. *Journal of Information Security*, 1(May):74–87.

Paul, M., Chauhan, N. S., and Saxena, A. (2011). A security analysis of smartphone data flow and feasible solutions for lawful interception.

Shurui, L. S. L., Jie, L. J. L., Ru, Z. R. Z., and Cong, W. C. W. (2010). A Modified AES Algorithm for the Platform of Smartphone.

Zovi, D. A. D. (2011). Apple iOS 4 Security Evaluation.