nccgroup

Whitepaper

Blackbox iOS App Assessments Using idb

June 7, 2015 – Version 1.0

Prepared by
Daniel A. Mayer — Senior Security Consultant

Abstract
More than ever, mobile apps are used to manage and store sensitive data by both corporations and individuals. In this paper, we review common iOS mobile app flaws involving data storage, inter-process communication, network communications, and user input handling as seen in real-world applications. To assist the community in assessing security risks of mobile apps, we introduce our recent tool called *idb* and show how it can be used to efficiently test for a range of iOS app flaws.

We will explore a number of vulnerability classes. Each class will first be introduced and discussed before demonstrating how idb can enhance the testing for instances of it. With this we illustrate how apps commonly fail at safeguarding sensitive data and demonstrate how idb can arm security professionals and developers with the means necessary to uncover these flaws from a black-box perspective. Furthermore, we will provide illustration of how to mitigate each flaw. idb is open source and available to the public.

Table of Contents

1 Introduction

More than ever, businesses and individuals use mobile applications to manage and store sensitive data [Sta09]. When analyzing how users interact with their mobile devices, studies show that it is mostly via mobile apps and not via the mobile web (see Figure 1). Compared to traditional applications where data is stored on a tightly controlled server, mobile applications typically cache data locally for performance and availability reasons. This leaves confidential data outside of the control of the corresponding service which provided the data. Moreover, since mobile devices are by definition portable, they are prone to theft and loss [Smi13], [BSM12] which opens up additional avenues for compromising data stored on the device.

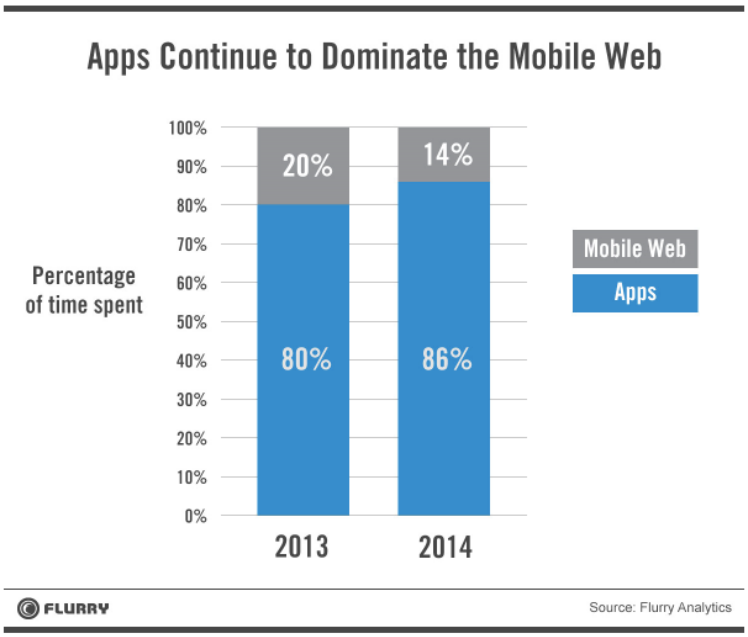

Figure 1: Research shows that users mostly use apps on mobile devices.

Besides data storage flaws, mobile applications are prone to a number of different attack vectors. Compared to platforms such as Android (Java-based) or Windows Phone (.NET-based), the assessment of iOS applications poses a unique challenge in that these applications are compiled to native code and source code is not readily available through decompilation or reflection. However, it is still possible to learn a significant amount about iOS applications from a black-box perspective.

In the following, we review common iOS application flaws (see Section section 2) and then show how our tool *idb* can be used to efficiently test for those flaws in a black-box manner (see Section section 3).

2 Common iOS App Flaws

The attack surface of mobile applications tends to be complex and vulnerabilities typically arise at trust boundaries where data is exchanged with other components. Those include user input, communication with backend services, on-device inter-process communication with other applications, and interactions with the iOS operating system (see Figure 2). Below, we visit some of the more prevalent types of vulnerabilities in these areas.

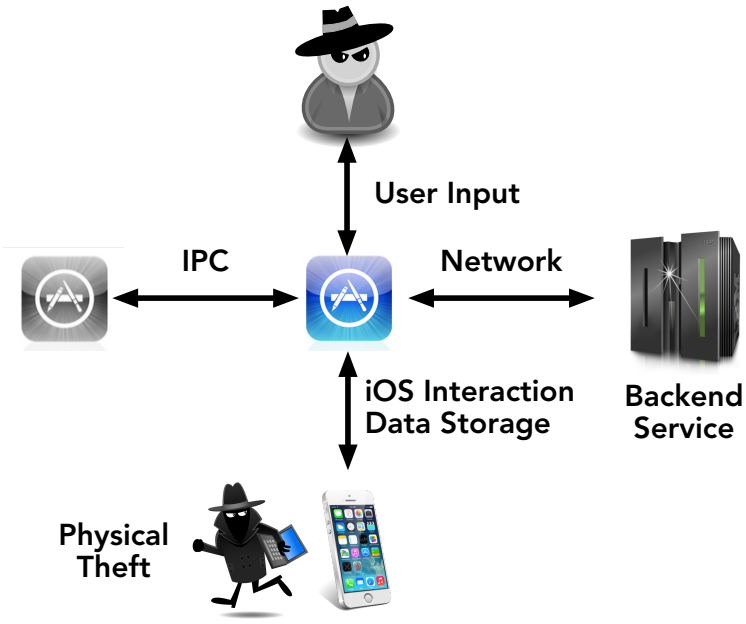

Figure 2: Overview of the attack surface of iOS applications

2.1 Local Storage

Should an attacker gain physical access to an iOS device, it is typically possible to jailbreak it and subsequently access any inadequately protected data on the device. Similar access may be achieved by reading device memory directly or by remotely exploiting an appropriate flaw in iOS (should one exist). The description in this section is kept brief and to the minimum required for understanding idb's features. See our upcoming whitepaper and BlackHat USA 2015 talk titled ''Faux Disk Encryption: Realities of Secure Storage on Mobile Devices'' for an in-depth discussion of this topic.

2.1.1 Storing Data in the File System

Sqlite databases and `NSUserDefaults` (plist files) are common ways for applications to store data on the device. Both of these use the default protection provided by iOS which, as we will see below, is limited.

In order to store data such that it cannot be recovered by an attacker who has full system-level access to the device, there must be some information that is inaccessible to the attacker. If one were simply to encrypt data, the encryption key would need to be stored securely which leaves us with the same problem we started with. In order to truly protect data, iOS incorporates the iOS passcode into a complex encryption hierarchy [App15c]. This hierarchy allows different protection levels, referred to as *Data Protection Classes* to be used by app developers (see Table 1) [App15a].

Protection Class	Meaning
`NSFileProtectionComplete`	Protected when device is locked.
`NSFileProtectionCompleteUnlessOpen`	If open, file can be read when locked.
`NSFileProtectionCompleteUntilFirstUserAuthentication`	Protected from boot until user unlocks.
`NSFileProtectionNone`	No effective protection.

Table 1: Overview of iOS file protection classes.

As the default, iOS automatically encrypts the entire user file system using a device and file-specific key, but it also transparently decrypts files for any read operation (`NSFileProtectionNone`). Since iOS 8, the default protection was changed such that the decryption key is only available as long as the device is turned on and the user unlocked the device at least once since boot-up (`NSFileProtectionCompleteUntilFirstUserAuthentication`) [App15c]. On reboot or shutdown, the key is deleted and will only be available once the user unlocks the device again. Note that since the key is derived from the user's passcode, this only provides additional protection if the user has a passcode set. Application developers can opt to use a more tight protection class which uses a key that is also derived from the passcode but which is removed from the device as soon as the user locks the device (`NSFileProtectionComplete`). Additional details on iOS data protection can be found in [MBD+12] and [App15c] as well as in our upcoming whitepaper mentioned above.

Best Practices: In order to access protected files on the device, an attacker would generally jailbreak the device or otherwise gain read-access to the file system. Most public jailbreaks require the device to be restarted in the process and the default protection class `NSFileProtectionCompleteUntilFirstUserAuthentication` may provide adequate protection in this attack scenario. However, there is no inherent need for a reboot in the jailbreak process and previous attacks have demonstrated that this is not generally required. Therefore, in order to protect sensitive data, developers should enable the `NSDataProtectionComplete` flag. Note that in order for this setting to be effective, a strong device passcode must be set by the user.

2.1.2 Storing Data in the Keychain

Data stored using `NSUserDefaults` resides in a Property List file (`.plist`) [Wik14]. This kind of storage mechanism is often used to persist settings, cryptographic keys, and credentials, making it a high-value target for an attacker. Data that is structured and small in size is a good candidate for being stored in the iOS keychain instead of a file.

The iOS keychain provides similar protections options as the ones available for the file system discussed in the previous section (see Table 2) [App15d]. The additional class `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` can be used to ensure that sensitive data is only stored on the device when a passcode is set. As mentioned previously, a passcode is required in order for `kSecAttrAccessibleWhenUnlocked` and `kSecAttrAccessibleAfterFirstUnlock` to be effective. This new class ensures that data is not left unprotected should the user choose to not have a passcode.

Protection Class	Meaning
`kSecAttrAccessibleWhenUnlocked`	Protected when device is locked.
`kSecAttrAccessibleAfterFirstUnlock`	Protected from boot until user unlocks.
`kSecAttrAccessibleAlways`	No protection.
`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly`	Store data only when passcode set.

Table 2: Overview of iOS Keychain protection classes.

Best-Practices: Developers should store all sensitive data of reasonable size in the iOS keychain using `kSecAttrAccessibleWhenUnlocked` or `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly`.

2.2 Interaction with iOS

When applications call iOS library or framework functions, there exist several side channels and developers may not be aware of the resulting data disclosure.

2.2.1 Background Screenshot

One notable quirk of iOS is that it stores an unencrypted screenshot when an application is backgrounded. Any content displayed on the screen when the application is placed in the background is thus accessible by an attacker.

Best Practices: Developers should remove or hide any sensitive content on the screen before the application backgrounds by using the `applicationDidEnterBackround` delegate of `UIApplication` [App15f, App14b].

2.2.2 Cached Requests and Responses

Another data leak is the automatic request and response caching performed by `NSURLConnection`. Specifically, it stores request and response data in `Cache.db`, an unencrypted sqlite database. This database is easily accessed and queried by an attacker.

Best Practices: In order to prevent caching of sensitive data, developers should, e.g., implement the `willCacheResponse` delegate of `NSURLConnection` to disable caching [App15e].

2.3 Inter-Process Communication (IPC)

Because iOS does not provide a proper IPC mechanism, applications rely on other iOS features for this purpose.

2.3.1 iOS Pasteboard

Some applications exchange data through `NSPasteboard`. Since data stored in the pasteboard (including private pasteboards) is accessible by any application on the device, it may be intercepted or modified by an attacker.

Best Practices: The pasteboard should never be used to exchange any kind of sensitive data. Moreover, consider deactivating the copy and paste mechanism for any fields that may contain sensitive data.

2.3.2 URL handlers

More commonly, URL handlers (schemes) are used to facilitate IPC. In this scenario, one application registers a custom URL scheme and receives data when a second application opens a corresponding URL. However, any application (even Safari) can call any URL scheme which means that data received via a URL handler should be considered untrusted. In addition, the same URL scheme may be registered by more than one application and Apple states that ''If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme." [App15b]. As a result, data passed via URL handlers can be hijacked by an attacker and researchers at FireEye have demonstrated that this is in-fact exploitable in practice [XWZ+15].

Best Practices: Do not send sensitive data via URL schemes as it may be intercepted by a malicious application. In addition, the receiving application has to check whether the calling application is trusted and perform strict data validation on the URL data.

2.4 Binary Protection

Since iOS applications are compiled to native code, memory corruption flaws may be present which can disclose any data accessible to the application. iOS provides advanced protections that make exploitation of memory corruption flaws more difficult (e.g., Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR)) [App15b, pp. 7-8]. However, in order to take full advantage of these features, some flags have to be set when building the application.

Best Practices: In order to take advantage of ASLR, the developer must compile the application as a Position Independent Executable (PIE) [App14a]. In addition, the developer should enable stack canaries [Wik15b] and Automatic Reference Counting (ARC) [Wik15a]. The former provides some degree of buffer overflow protection and the latter will help avoid use-after-free or double-free vulnerabilities.

2.5 Network / API

Finally, if the application communicates with backend systems (such as HTTP APIs), vulnerabilities similar to those found in web applications may be present in the back-end. While most of these vulnerabilities are not specific to iOS applications, SSL/TLS certificate validation deserves special attention. In general, all applications should use SSL/TLS to protect data in transit between the application and the server.

Best Practices: In order for SSL/TLS to be effective, Man-In-The-Middle (MITM) attacks must be prevented through strict SSL/TLS certificate validation (i.e., only establish a connection if the server's SSL certificate is trusted). This is the default on iOS and should not be bypassed. For additional security, the validation can be *pinned* to a single certificate or trusted Certificate Authority (CA) which prevents a malicious or compromised CA from issuing certificates that may be used in a MITM attack. The resources at [iSE15] and [OWA15] can aid in implementing certificate pinning in a proper manner.

3 Simplified Pentesting with idb

With the goal of making black-box testing for the above and other vulnerabilities easier and more efficient, we developed a new tool called *idb* available at http://www.idbtool.com [May15b, May15a]. Where possible, idb leverages existing tools and provides a unified interface (see Figure 3) to make them easier to use. In addition, idb provides functionality that was not publicly available previously. idb's features developed over time and it remains under active development with the latest version (2.8) having approximately 5,500 lines of ruby code. In an attempt to remain platform-independent, idb is written in ruby using Qt for the GUI front-end. In the following, we give an overview of idb's features and relate them to the common iOS vulnerabilities discussed in the previous section.

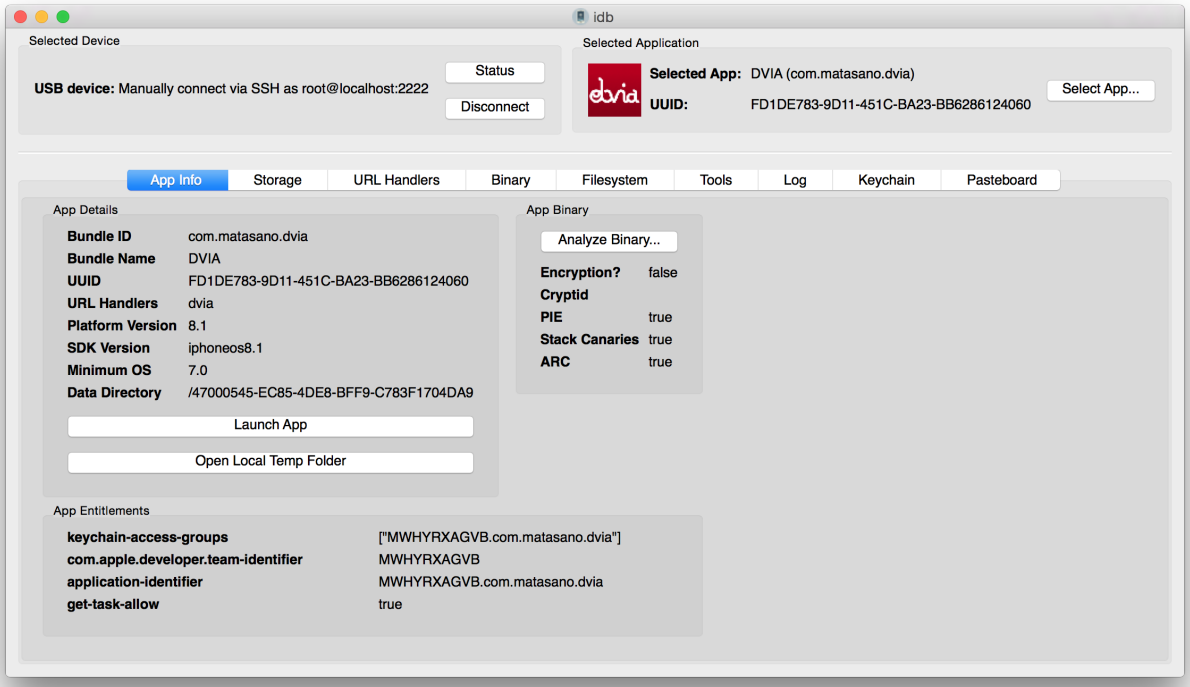

Figure 3: Overview of idb's user interface.

3.1 Pentesting Setup

idb requires a jailbroken iDevice and communicates with it via SSH. One can either establish this connection directly with the device's SSH server or use *usbmuxd* [usb] to tunnel SSH over USB. Any required usbmuxd connections are established automatically and transparently.

3.1.1 Port Forwarding

idb provides optional port forwarding between the host running idb (*client*) and the iDevice (*server*). The forwarding settings follow the SSH naming conventions where *Remote* forwarding opens up a port on the server and forwards it to the given host/port originating at the client. Conversely, *Local* forwarding opens up a port on the client and forwards incoming connection to the specified host/port on the server side. One can specify an arbitrary number of each forward type. This function is particularly useful for investigating traffic from the device using an intercepting HTTP proxy. For example, assume your proxy is running on the client and listens on port 8080 then you can define

```
remote:8080 -> localhost:8080
```

This remote forward will open port 8080 on the device and forwards all incoming connections to port 8080 on the host running idb. With this in place, one can configure the proxy server on the device as localhost:8080 and the traffic will be proxied via SSH (and USB).

3.2 Basic Application Information

After connecting to the device and selecting an application to analyze, idb immediately displays some basic information on the application. This data is extracted from its `Info.plist` and related files and displayed in the *App Info* tab:

1. Bundle ID
2. Bundle Name
3. UUID
4. Registered URL Schemes
5. Platform Version
6. SDK Version
7. Minimum OS Version
8. The folder where app data is stored (iOS 8+)

In addition, all of the app's entitlements are displayed as well.

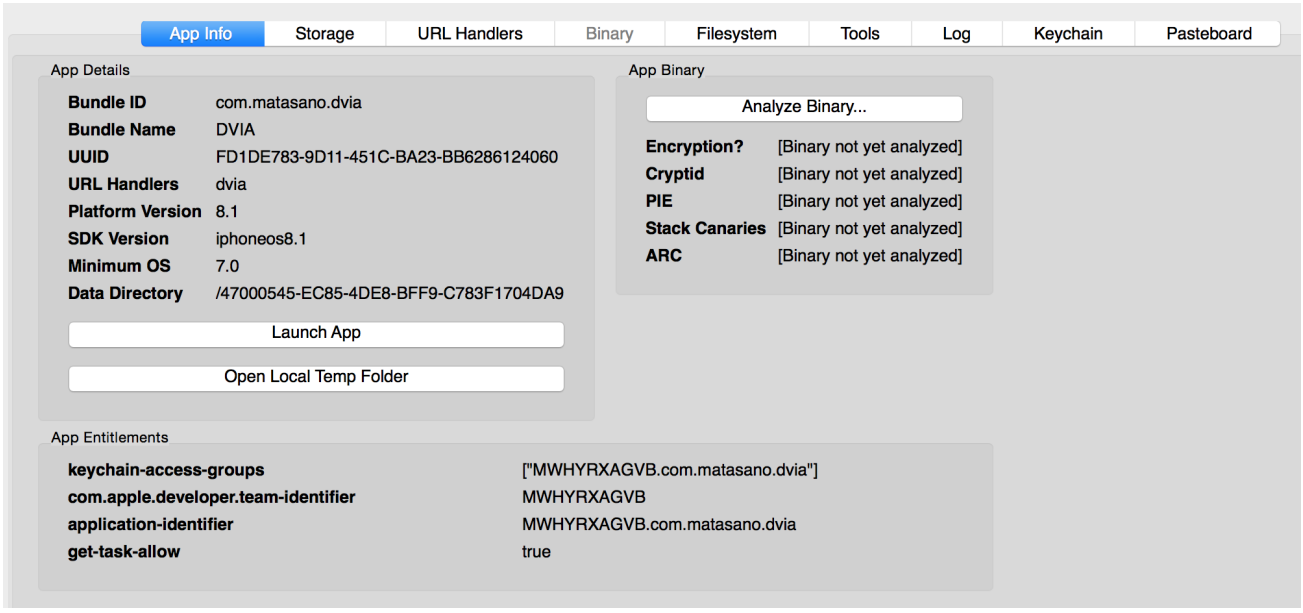

Figure 4: idb's App Info tab.

3.3 Local Storage

idb has a number of functions related to local storage. All of them are geared to get a quick overview on which data is stored on the device and whether it is adequately protected.

3.3.1 Special File Types

As a new function, idb provides convenient searching for sensitive files such as `.sqlite`, `.plist`, and `Cache.db` files. For any discovered file, the content and Data Protection Class can be viewed. For iOS 8+, Each storage related tab also displays the *Default Data Protection* of the app which is set via app entitlements. This class defines the protection used for files created by the app if the developer does not explicitly specify a different class. In order to determine the protection class for each file, idb uses a small helper utility which is available at: https://github.com/dmayer/protectionclassviewer

3.3.2 File System Browser

In addition, idb includes a full file system browser. It provides an easy to use interface for browsing the app's sandbox. All directories and files are listed in a familiar tree and list structure. A details pane displays file permissions and the DataProtection class which applies to the file. This way one can easily verify that sensitive files are properly protected. Moreover, double-clicking on any file will automatically download and open it on the idb host with the application that is associated with the file extension.

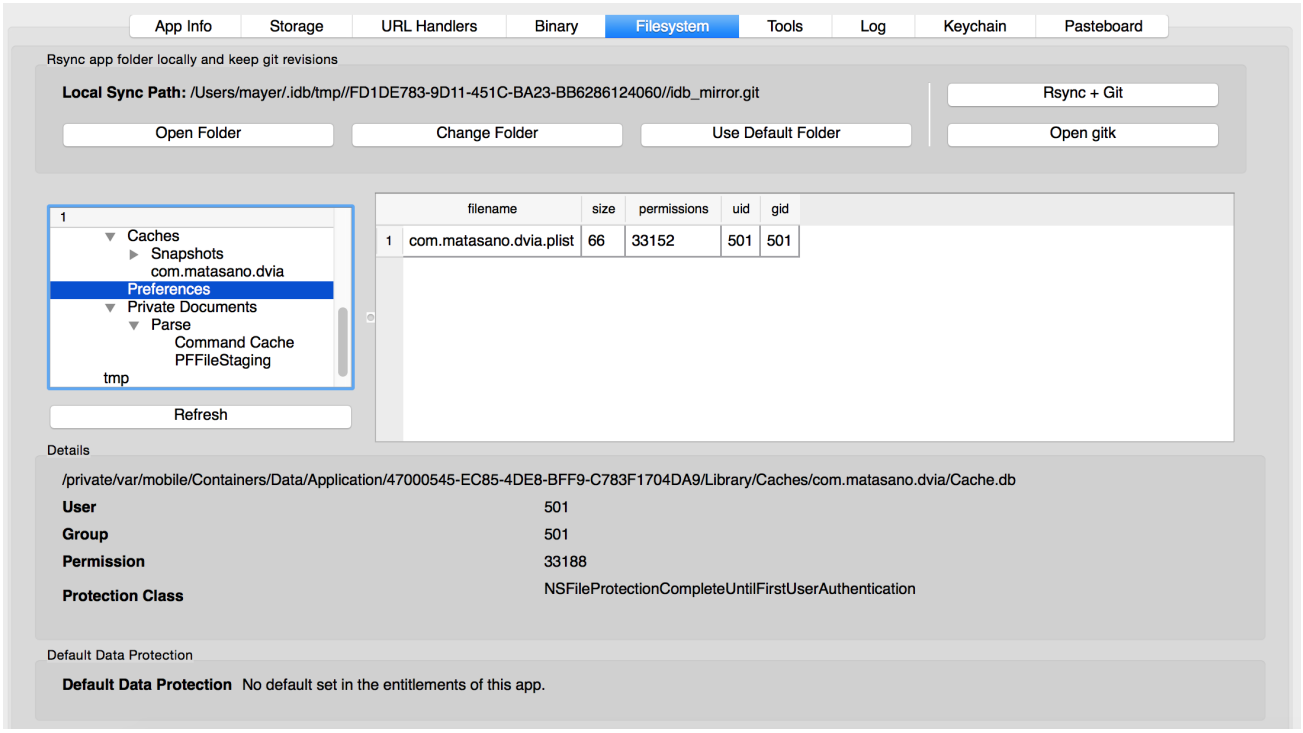

Figure 5: idb's filesystem browser.

Sync+Git: In order to get a full mirror of the current application sandbox (and the data directory on iOS 8+), the Git+Rsync function can be used. It uses rsync to create an exact mirror of the remote file system on the idb host. In addition, the entire directory structure is checked into a dedicated git repository. When performing subsequent syncs using Git+Rsync, new revisions are created in the git repo and thus changes in the application directory can be tracked. The directory where the git repository is held can easily be changed

in the user interface. This can, e.g., be used to compare or keep track of installations on different devices.

3.3.3 iOS Keychain

idb provides a convenient way for dumping, editing, and deleting the content of the iOS Keychain (adding new keychain items is currently a work in progress). Figure 6 shows an overview screenshot of the keychain tab. This tab allows easy dumping of the entire keychain content. The returned data includes Entitlement Groups, Account, Service, Protection class, User Presence requirement, create and modification date. User presence is a new feature in iOS which requires a user to authenticate either via TouchID or by entering their passcode in order to access a keychain item. Note that there is no known way for bypassing this check since data is in fact encrypted under the passcode and this processing happens in hardware through the Secure Enclave.

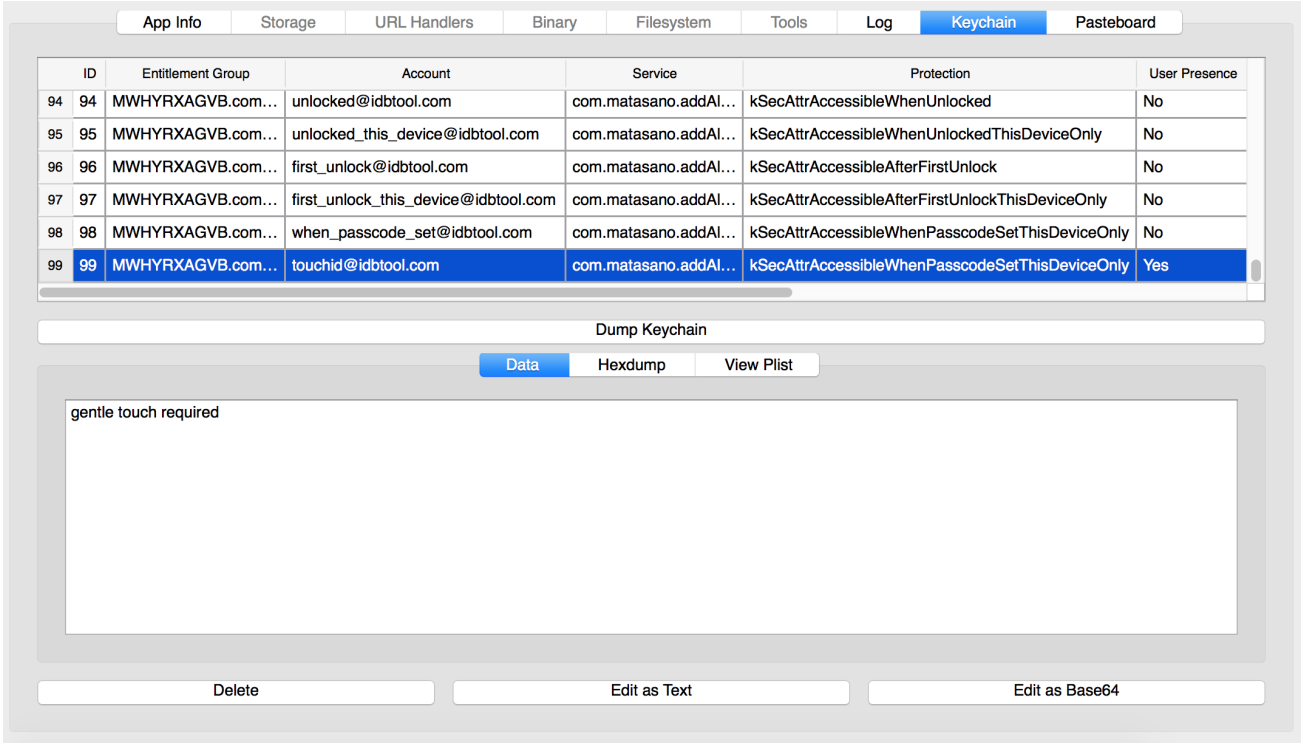

Figure 6: idb's keychain editor.

When selecting an entry in the table shown above, the data can be viewed as text, hexdump, or parsed as XML in the case of binary plist data. In addition, each entry can be deleted and edited.

3.4 Binary Protection

To check the binary protections used by the application, idb displays basic information on the application binary. This includes binary encryption, PIE, use of stack canaries, and ARC. If the application binary is encrypted, idb uses Stefan Esser's dumpdecrypted (https://github.com/stefanesser/dumpdecrypted) to decrypt the application before downloading and analyzing it.

3.4.1 Shared Libraries

In addition, the *Shared Libraries* tab provides a listing of all the external libraries the application references. This allows for the easy discovery of any suspicious frameworks or vulnerable libraries that may be in use by the application. Internally, idb uses `otool` to analyze the binary.

3.4.2 Strings

Application binaries frequently include data of interest such as API keys, credentials, encryption keys, URLs, etc. The strings tab extracts all strings in the (decrypted!) application binary and displays them right in the UI.

3.4.3 Class and Method Signature Dumping

When reversing, instrumenting (e.g., using Cycript (http://www.cycript.org/) or Mobile Substrate (http://www.cydiasubstrate.com/)), or simply trying to understand an application, knowing all of the classes and method signatures of the application is of great help. idb provides a convenient way for obtaining these from compiled iOS applications. Under the hood, this function uses `cycript` and the *weak_classdump* script by Elias Limneos (https://github.com/limneos/weak_classdump).

3.5 Inter-Process Communication

idb provides functions for monitoring both the iOS pasteboard as well as URL schemes.

3.5.1 Pasteboard

idb's integrated pasteboard monitor displays any content copied to the general and private pasteboards in near real time. In order to monitor the pasteboard, idb uses a small helper utility which is available at: https://github.com/dmayer/pbwatcher

3.5.2 URL Handlers

For URL handler testing, a list of the registered schemes is provided and any URL scheme can directly be launched from idb. Since input received via URL schemes is often used in unsafe ways, which can lead to vulnerabilities including logic flaws and memory corruption, idb includes a basic URL scheme fuzzer. On the *Fuzzer* tab, one can enter a number of fuzz strings as well as a fuzz template. In the template, `$@$` is used to mark potential injection points. For example, if a valid URL is

```
dvia://configure?input1=hello&input2=woot
```

one could specify

```
dvia://configure?input1=$@$&input2=$@$
```

as the template to fuzz both of the intended inputs. For each position, idb will cycle through all the possible fuzz inputs and launch the URL handler (which launches the app) and then wait for several seconds before killing it. In order to detect a crash, idb monitors the `/var/mobile/Library/Logs/CrashReporter` folder for new crash reports for the application in question.

3.6 Other Tools

3.6.1 Certificate Manager

idb can be used to easily view, install, and remove SSL (CA) certificates on the device and the simulator. This simplifies the setup for interception of HTTPS / SSL connections. In particular, if the device is proxied through *Burp Suite*, one can install its CA certificate with a single click in idb.

3.6.2 Screenshot Tool

The screenshot utility is a simple wizard that can be used to test whether an app is disclosing sensitive data in the automatic backgrounding screenshots taken by iOS. After starting the wizard, the *Launch Application* button can be used to launch the app under investigation. After following the instructions of the wizard, idb will download the screenshot and allow you to open it in the default image viewer.

3.6.3 Hosts File Editor

The `/etc/hosts` file editor provides a simple way to modify the hosts that applications connect to. In order to intercept traffic for an app, one would typically use a tool such as *Burp Suite* and set the iOS system proxy to forward application traffic to it. However, when the app does not respect proxy settings or communicates via non-HTTP protocols, this may fail. Under these circumstances modifying the `/etc/hosts` may help in overriding the DNS entry for a host and pointing the app at a running proxy instance which then forwards traffic to the actual server expected by the app.

3.6.4 iOS Log

The Log tab can be used to view the syslog of the iDevice. Besides system messages, it also includes any log statements that apps produce using `NSLog` which often disclose sensitive data. Internally, the log view uses `idevicesyslog` which is part of libmobiledevice [lib].

4 Summary

Flaws in iOS applications arise in a variety of ways and by releasing our tool idb [May15b, May15a] we aim at making iOS application assessments easier and more effective.

References

[App14a] Apple Inc. Technical Q&A QA1788 - Building a Position Independent Executable. https://developer.apple.com/library/ios/qa/qa1788/_index.html, 2014. 7

[App14b] Apple Inc. Technical Q&A QA1838 - Preventing Sensitive Information From Appearing In The Task Switcher. https://developer.apple.com/library/ios/qa/qa1838/_index.html, 2014. 6

[App15a] Apple Inc. File Protection Values. https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSFileManager_Class/index.html#//apple_ref/doc/constant_group/File_Protection_Values, 2015. 4

[App15b] Apple Inc. Inter-App Communication. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html, 2015. 7

[App15c] Apple Inc. iOS Security. https://www.apple.com/business/docs/iOS_Security_Guide.pdf, April 2015. 4, 5

[App15d] Apple Inc. Keychain Services Reference. https://developer.apple.com/library/ios/documentation/Security/Reference/keychainservices/index.html, 2015. 5

[App15e] Apple Inc. NSURLConnectionDataDelegate Protocol Reference. https://developer.apple.com/library/mac/documentation/Foundation/Reference/NSURLConnectionDataDelegate_protocol/index.html#//apple_ref/occ/intfm/NSURLConnectionDataDelegate/connection:willCacheResponse:, 2015. 6

[App15f] Apple Inc. UIApplicationDelegate Protocol Reference. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIApplicationDelegate_Protocol/#//apple_ref/occ/intfm/UIApplicationDelegate/applicationDidEnterBackground:, 2015. 6

[BSM12] Jan Lauren Boyles, Aaron Smith, and Mary Madden. Privacy and Data Management on Mobile Devices. http://www.pewinternet.org/2012/09/05/main-findings-7/, 2012. 3

[iSE15] iSEC Partners. SSL Conservatory. https://github.com/iSECPartners/ssl-conservatory, 2015. 7

[lib] libimobiledevice - A cross-platform software protocol library and tools to communicate with iOS® devices natively. http://www.libimobiledevice.org/. 13

[May15a] Daniel A. Mayer. idb - Github. https://github.com/dmayer/idb, 2015. 8, 14

[May15b] Daniel A. Mayer. idb - iOS App Security Assessment Tool. http://www.idbtool.com, 2015. 8, 14

[MBD+12] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.P. Weinmann. *iOS Hacker's Handbook*. ITPro collection. Wiley, 2012. 5

[OWA15] OWASP. Certificate and Public Key Pinning. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#iOS, 03 2015. 7

[Smi13] Gerry Smith. Apple, Samsung Face Grilling Over Stolen Smartphone Epidemic. http://www.huffingtonpost.com/2013/06/05/apple-samsung-thefts_n_3383407.html, 2013. 3

[Sta09] Morgan Stanely. The Mobile Internet Report, 2009. 3

[usb] usbmuxd - A socket daemon to multiplex connections from and to iOS devices. http://cgit.sukimashita.com/usbmuxd.git/. 8

[Wik14] Wikipedia. Property list — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Property_list&oldid=610572011, 2014. [Online; accessed 17-May-2015]. 5

[Wik15a] Wikipedia. Automatic Reference Counting — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Automatic_Reference_Counting&oldid=654718993, 2015. [Online; accessed 17-May-2015]. 7

[Wik15b] Wikipedia. Buffer overflow protection — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Buffer_overflow_protection&oldid=658859945#Canaries, 2015. [Online; accessed 17-May-2015]. 7

[XWZ+15] Hui Xue, Tao Wei, Yulong Zhang, Song Jin, and Zhaofeng Chen. iOS Masque Attack Revived: Bypassing Prompt for Trust and App URL Scheme Hijacking. https://www.fireeye.com/blog/threat-research/2015/02/ios_masque_attackre.html, 02 2015. 7