Reverse Engineering iOS Applications

Drew Branch, Independent Security Evaluators, Associate Security Analyst

ABSTRACT

Mobile applications are a part of nearly everyone's life, and most use multiple mobile applications on a day-to-day basis. Mobile applications are widespread and have a plethora of purposes—including, but not limited to, banking and budgeting, social media, sending money, and playing games. With all of these capabilities, one must ponder whether or not these applications are securing sensitive user information at rest, as well as in transit. While Apple provides an API for developers to secure data, developers may not be utilizing these controls in a secure manner. This paper describes common mistake security developers make, methods to test for those mistakes, and problems encountered when testing for them. We will also explore reverse engineering techniques to analyze iPhone operating system (iOS) applications.

INTRODUCTION

When developing iOS applications, there are several ways to secure sensitive data that an application may handle. These measures may or may not be secure when the device is lost or stolen, which could lead to the loss in integrity of the sensitive data. Even when utilizing Apple's provided security controls (e.g., keychain) for secure storage, data is still at risk for exposure.

iOS applications have their own sandboxed folders, which cannot be accessed by any other application. Although every iOS application has its own sandboxed folder, the data within those folders could be accessible by readily available free applications. While Apple's security model is a comprehensive one, it relies on the fact that users do not have file system root-level access. Developers must take extra steps to ensure sensitive data is secure from adversaries even when they have root access to the file system.

The Damn Vulnerable iOS Application[1] (DVIA) will be used to simulate common mistakes that developers make. This application was developed to provide people with an application to gain or test iOS application reverse engineering skills.

This whitepaper is geared toward those who want to gain knowledge about assessing iOS applications and/or developers who want to know how to develop more security sound applications.

COMMON SECURITY MISTAKES

Developers make security-compromising mistakes when developing applications due to the implied secureness of an operating system. While developers may be using a secure space (e.g., keychain) for storing sensitive information, this space is only secure when a device is not jailbroken. Other mistakes arise when developers

[1] http://damnvulnerableiosapp.com/

assume that only their applications have sufficient privileges to access their sandboxed application folders and corresponding data. Many tools, which are readily available, have the capabilities to bypass the security model of the iPhone operating system, as well as access iOS backups that contain sensitive user data (e.g., text messages, notes, and voicemails). These tools make accessing applications' data, which are supposed to be secure and sandboxed, extremely easy. In the subsequent sections, common security mistakes, the problems encountered while testing for said mistakes, and how to overcome those problems will be discussed.

Storing Sensitive Data on the File System

According to the iOS security model, an application does not have access to another application's sandboxed folder and data. Further, users do not have direct access to the file system, which would allow users to browse and extract files from the file system. Developers use knowledge of iOS's security model to store sensitive information, such as: credit card information, passwords, and personally identifiable information (PII) on the file system. This information should not be stored on a device that could be lost or stolen because the iOS security model could be bypassed with relative ease. This will lead to the exposure of the sensitive information stored on the file system.

Mobile operating systems do not provide built-in browser utilities, which is a problem that a penetration tester may encounter during an assessment. iExplorer[2] will aid in overcoming this problem on iOS devices. This application is offers a free trial and is available on Windows and Mac OS X operating systems. Many of iExplorer's features do not require the iOS device to be jailbroken. iExplorer provides a clean user interface, where one could traverse an application's bundle folder, as well as its data folder, view files, and extract files. iExplorer has the capabilities to open several file types such as images, databases, and text files.

Using DVIA, "testuser" and "secretpassword" are stored in a .plist file with the application's bundle folder. Figure 1 and Figure 2 display the sensitive information stored by the application in a .plist and .plist file opened using iExplorer.

---

[2] https://www.macroplant.com/iexplorer/

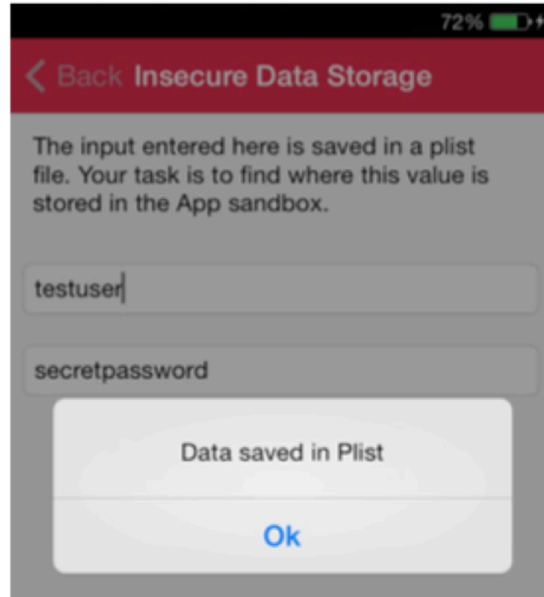

Figure 1. Sensitive data stored in keychain.

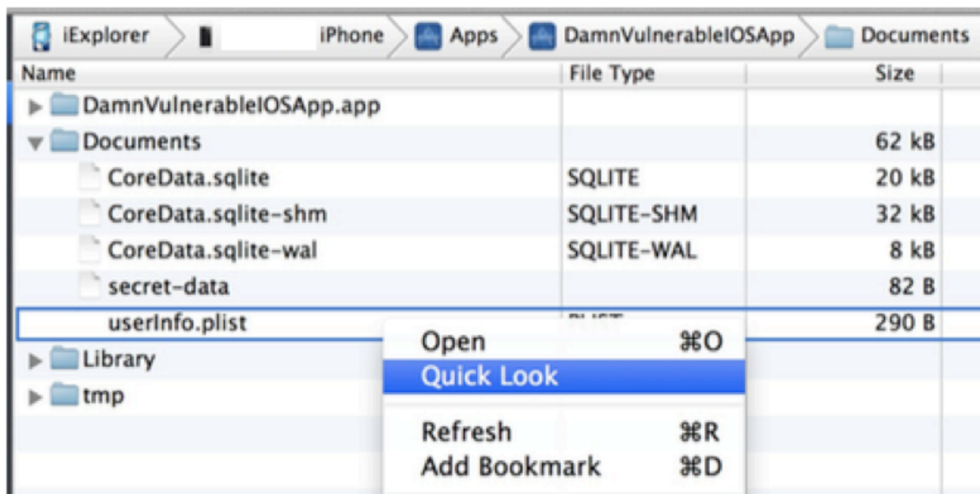

Figure 2. DVIA sandboxed folder contents.

Once the option to open the .plist file within iExplorer is available, execute an alternate click and select "Quick Look," then the file will open and display the username and password in plaintext. Figure 3 displays the results of opening the .plist file.

Figure 3. Sensitive information exposed within a .plist file.

Sending Sensitive Data over Unencrypted HTTP

In addition to storing sensitive information in an insecure manner, some developers make the mistake of sending sensitive data over an unencrypted HTTP channel without sufficient protection (i.e., encryption). When attempting to analyze network traffic, penetration testers cannot run packet sniffer applications on mobile devices, such as Wireshark.[3] To bypass this barrier, a proxy server, such as Burp Suite,[4] could be used to capture network traffic for analysis.

To proxy mobile device network traffic:

1. Start a proxy listener within Burp Suite or related application.
2. Edit the mobile device's network configuration to utilize a HTTP Proxy server.
3. Enter the IP address and port number of the proxy server within the mobile device's network configuration.

When testing for this type of information leakage, key information to look for when analyzing network traffic is user credentials, credit card information, etc. Figure 4 displays credit card information (Credit Card number, CVV number, and Name) sent from the DVIA to a bogus server over HTTP. Figure 5 displays the HTTP POST request captured by Burp Suite.

---

[3] https://www.wireshark.org/
[4] https://portswigger.net/burp/

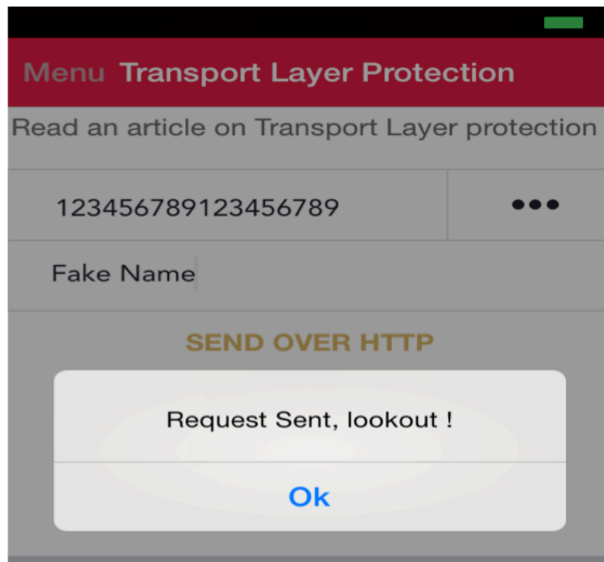

Figure 4. Credit card information sent within a HTTP POST request.

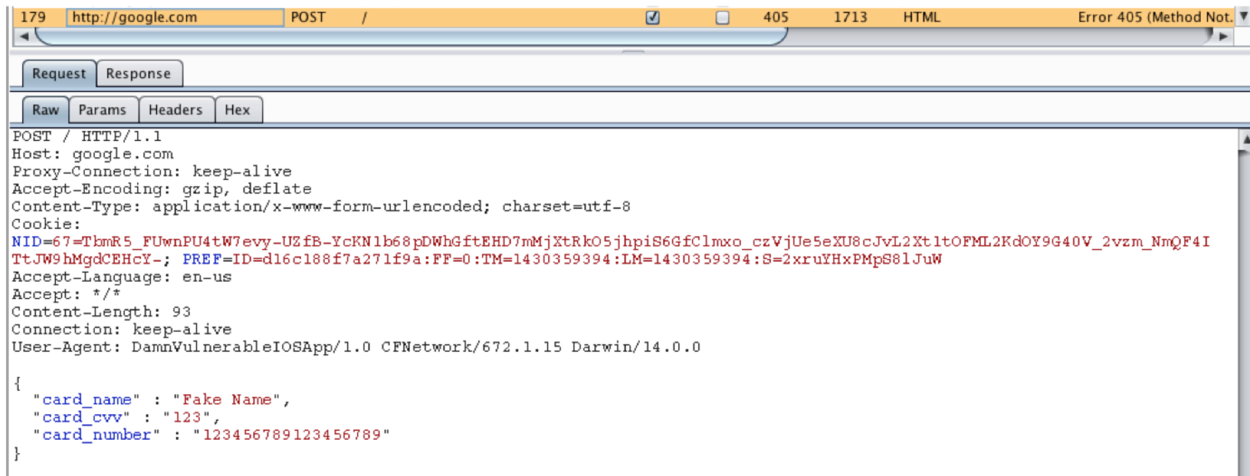

Figure 5. HTTP POST request captured with Burp Suite.

Security Bugs in Server APIs

Many mistakes occur when implementing server APIs. Most times, these mistake include missing authentication in RESTful APIs, broken authorization checks, and sending sensitive data to servers. Mobile applications do not have built-in developer tools like web browsers, so developers sometimes overlook easy problems. Further, developers believe that HTTPS is sufficient enough to stop reverse engineering/testing of an API.

When testing server APIs, if an application is using HTTPS and checking certifications properly, assessors cannot simply perform a man-in-the-middle attack. To overcome this issue, a tester can use interception software such as Burp Suite. iOS keeps a list of trusted certificate authorities (CAs), and many applications rely on this list provided by iOS. Interception software has the capabilities to create a bogus CA. Once a bogus CA has been created, load the CA into the device to be able to analyze HTTPS traffic.

Some developers may take an additional step when protecting their APIs by utilizing a custom trusted CA list or have the CA built into the application. Having the CA built into the application is referred to as cert pinning and is defined as the process of associating a host with its expected certificate or public key. This will negate the bogus CA created by the interception software, and the interception software will not capture requests.

The application SSL Killswitch[5] bypasses cert pinning, which will allow tools such as Burp to capture HTTPS requests. SSL Killswitch attaches to the application and suppresses typical implementations of cert pinning. To install this application, the iOS device used for testing must be jailbroken. Once installed, activate SSL Killswitch and monitor HTTP requests as stated in the section above.

Storing Sensitive Information in the Keychain

As described by Apple,[6] the keychain is provided to store sensitive information that an application may need to store. Sensitive information, such as passwords and encryption/decryption keys, is often found within the keychain. It should be noted that the iOS keychain security model relies on the user not having root permission level access to the filesystem. That said, the keychain is not accessible without jailbreaking the test device. Once the device is jailbroken, a program called Keychain_dumper[7] could be transferred to the device via sftp to dump the contents of the keychain.

---

[5] https://github.com/iSECPartners/ios-ssl-kill-switch

[6] https://developer.apple.com/library/mac/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html

[7] https://github.com/ptoomey3/Keychain-Dumper

To demonstrate how easily one could gain access to the keychain, “this is a secret” was stored in the keychain via the DVIA. To access the newly created keychain entry:

1. SSH into the jailbroken iOS device.
2. Traverse the filesystem to the directory where the keychain_dumper application resides.
3. Make the keychain_dumper application executable using `chmod +x keychain_dumper`.
4. Then execute the binary using `./keychain_dumper`
5. Locate the DVIA entry.

Figure 6 shows the text that was stored into the keychain, and Figure 7 shows the entry within the keychain after being dumped.

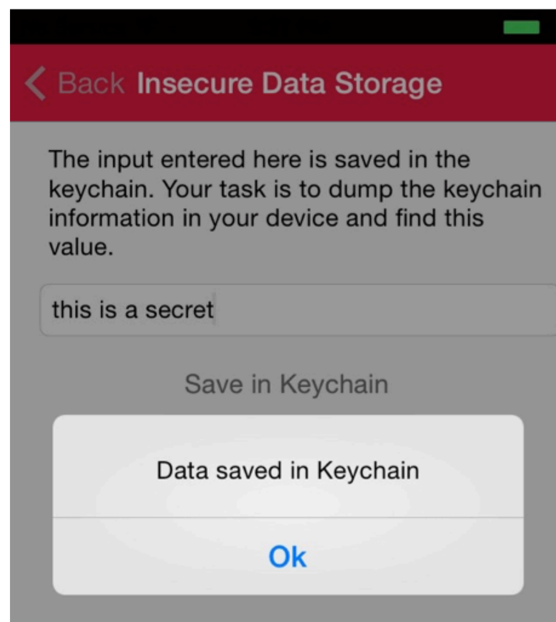

Figure 6. Sensitive data stored within the keychain.

```
iPhone:/ root# ./keychain_dumper
Generic Password
----------------
Service: com.highaltitudehacks.dvia
Account: keychainValue
Entitlement Group: 5SN4U5A564.com.highaltitudehacks.dvia
Label: (null)
Generic Field: (null)
Keychain Data: this is a secret
```

Figure 7. Sensitive data exposed from keychain.

Hardcoding Sensitive Information inside Application Binary

Hardcoding any type of sensitive information within an application binary is against best practices. Mobile application developers believe that mobile applications are extremely difficult to analyze, which is not completely the case. The same tools used to analyze native applications could be used to analyze mobile applications.

One issue users can encounter when attempting to analyze an application is that iOS applications from the App Store are encrypted by default and do not contain any debugging information. To overcome this problem, Clutch[8] is used to decrypt iOS applications from the App Store. This will allow the application to be extracted from the filesystem and loaded into a disassembler (e.g., IDA Pro).

To install Clutch:

1. Open Cydia.
2. Add cydia.iphonecake.com to the source list.
3. Search and install Clutch.

Once Clutch is installed, establish a secure shell (SSH) connection between a computer system and the iOS device and type "Clutch;" a list of applications will appear that are installed on the device that could be decrypted. Following the onscreen instructions, select the application to be analyzed. Once the decryption process is complete, extract the "cracked" binary from the file system and load it into a disassembler/decompiler of choice to analyze the application for sensitive hardcoded information.

---

[8] https://github.com/KJCracks/Clutch

REVERSE ENGINEERING TECHNIQUES

Depending on the scope of the mobile application assessment, a penetration tester may need to reverse engineer an iOS application using static and dynamic analysis. While Apple makes an effort to discourage reverse engineering, it is still very possible to carry out. In the succeeding sections, types of reverse engineering, methods, and tools will be explored.

Extracting Implementation Details

iOS applications are coded in the Objective-C language. Objective-C and C/C++ compilers are different. Message passing and late binding are attributes of Objective-C. Message passing is a form of communication where objects send each other messages; late binding allows the receiving class to handle the same message differently than other classes. These attributes lead to the inclusion of embedded strings, as well as classes, method, and global variable names.

Classes, methods, variables, and properties of an application could be displayed using class-dump.[9] Class-dump is very similar to otool,[10] with `-ov` flags, but the output of class-dump is more user friendly and readable. Figure 8 displays a snippet of the output from class-dump of the DVIA application.

[9] http://stevenygard.com/projects/class-dump/
[10] http://www.unix.com/man-page/osx/1/otool/

```
iPhone:/var/mobile/Applications/DamnVulnerableIOSApp.app root# class-dump DamnVulnerableIOSApp
/*
 *     Generated by class-dump 3.1.2.
 *
 *     class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2007 by Steve Nygard.
 */

struct CGAffineTransform {
    float a;
    float b;
    float c;
    float d;
    float tx;
    float ty;
};

struct CGContext;

struct CGPoint {
    float _field1;
    float _field2;
};

struct CGRect {
    struct CGPoint _field1;
    struct CGSize _field2;
};

struct CGSize {
    float _field1;
    float _field2;
};

struct _CCCryptor;

struct _NSZone;

struct _RNCryptorKeyDerivationSettings {
    unsigned int keySize;
    unsigned int saltSize;
    unsigned int PBKDFAlgorithm;
    unsigned int PRF;
    unsigned int rounds;
    char hasV2Password;
};

struct _RNCryptorSettings {
    unsigned int algorithm;
    unsigned int blockSize;
    unsigned int IVSize;
    unsigned int options;
    unsigned int HMACAlgorithm;
    unsigned int HMACLength;
```

Figure 8.Class-dump of DVIA.

Easy Dynamic Analysis

Snoop-it[11] is a dynamic analysis application that watches for common events important to reverse engineers, such as keychain access, file system access, network communication, and cryptography activity. Snoop-it also has more advanced features, such as method tracing and method invoking. This application makes analyzing an application easier and more efficient. Snoop-it is available by adding its source to Cydia. Figure 9 depicts the UI of Snoop-it and sample output when analyzing keychain actions from the DVIA.

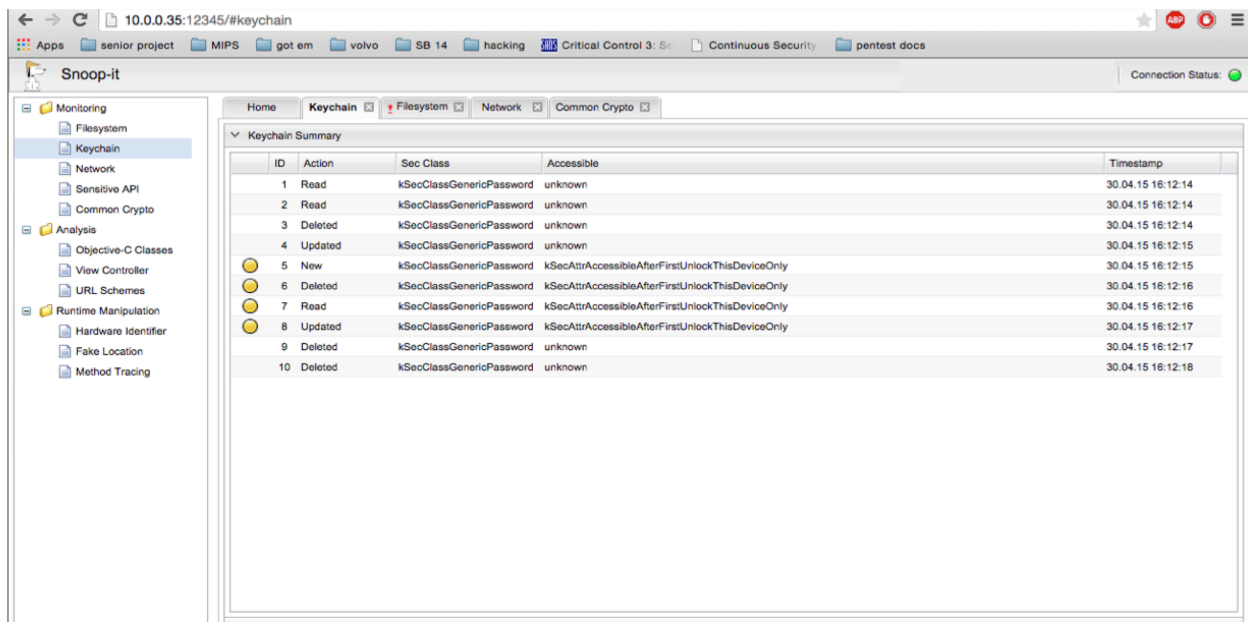

Figure 9. Snoop-it UI displaying recorded keychain actions.

Advanced Dynamic Analysis

Information obtained from static analysis (e.g., class-dump, disassembly analysis) should be used to identify key functions for break points to be set on for further analysis.

Recent versions of iOS require the use of LLDB[12] to perform dynamic analysis. LLDB is a debugger developed by Apple that was made to replace GDB[13] and provide a faster, more reliable experience. Debugging iOS applications remotely requires a Mac system and Xcode with command line tools installed. To provide an iOS device with the

---

[11] https://code.google.com/p/snoop-it/
[12] http://lldb.llvm.org/
[13] https://www.gnu.org/software/gdb/

capability to allow remote debugging connections, debugserver[14] must be extracted from the iOS developer image, correct entitlements must be added, and then loaded onto the iOS device.

lldb Usage

Once the debugserver binary has been transferred to the iOS device, run the application to be analyzed, then determine the application's process id (pid) via the ps aux | grep <application name> command. Once the application's pid has been determined, execute the debugserver via the following command:

```
debugserver \*:port number -a pid
```

The command above executes the debugserver binary, listens for connections on the selected port from all IP addresses, and attaches to the application with the supplied pid number. Figure 10 depicts the process of determining the pid number of an application and using debugserver to attach to the application.

```
    :~ root# ps aux | grep Damn
mobile     368   0.0  4.3   605084  22056   ??  Ss    4:20PM   0:01.97 /Applications
/DamnVulnerableIOSApp.app/DamnVulnerableIOSApp
root       497   0.0  0.1   536256    432 s000  R+    4:22PM   0:00.01 grep Damn
    :~ root# debugserver *:1234 -a 368
debugserver-300.2 for armv7.
Attaching to process 368...
Spawning general listening thread.
Spawning kqueue listening thread.
Listening to port 1234 for a connection from *...
Waiting for debugger instructions for process 368.
```

Figure 10. Starting debugserver for remote debugging.

On the host Mac machine, open a terminal window and type "lldb." If the Xcode command line tools are not installed, a prompt will appear requesting to install them. Once LLDB is loaded, enter "platform select remote-ios." Once the device support requirements have been loaded, enter the following:

```
process connect connect://ip_adress_of_iOS_device:port
```

This command will connect to the remote debugserver on the iOS device and allow users to debug an iOS application. LLDB commands are similar to gdbs, and a comparison list can be found at http://lldb.llvm.org/lldb-gdb.html.

---

[14] http://iphonedevwiki.net/index.php/Debugserver

CONCLUSION

Developers can make a number of mistakes to place sensitive data at risk of exposure; these include storing sensitive data on the file system, sending sensitive data over an unencrypted channel, and storing sensitive information in the keychain. Plus, security bugs in server APIs put data at risk. While developers may rely on iOS' security model to securely store sensitive data, there are techniques and readily available tools to circumvent Apple's provided security controls available to adversaries. To ensure sensitive data is at a reduced risk of exposure, developers should avoid using Apple's iOS security model and security controls and rely on information security best practices to secure sensitive data.

By using the common security mistakes identified above along with the reverse engineering techniques mentioned, more secure iOS applications should be developed and tested thoroughly from a security standpoint. Following the steps listed in this report can help ensure that user information and other sensitive data that is stored and transmitted through iOS applications will be accomplished in a secure manner.