3 Accepted Papers

3.1 Mathieu Renard/ Practical iOS Apps hacking

3.1.1 Mathieu Renard

@GoToHack Mathieu Renard "GoToHack" is a Senior Penetration tester, working for a French company (SOGETI-ESEC) where is leading the penetration test team. His research areas focus in Web Application Security, Embedded Systems, Hardware hacking and recently Mobile device Security. Since last year, he has focused is work (security assessments) and his research on professional iOS applications and their supporting architecture where data security is paramount.

twitter: @GoToHack

3.1.2 Practical iOS Apps hacking

This talk demonstrates how professional applications like, Mobile Device Management (MDM) Client, Confidential contents manager (Sandbox), professional media players and other applications handling sensitive data are attacked and sometimes easily breached. This talk is designed to demonstrate many of the techniques attackers use to manipulate iOS applications in order to extract confidential data from the device. In this talk, the audience will see examples of the worst practices we are dealing with every day when pentesting iOS applications and learn how to mitigate the risks and avoid common mistakes that leave applications exposed. Attendees will gain a basic understanding of how these attacks are executed, and many examples and demonstrations of how to code more securely in ways that won't leave applications exposed to such attacks. This talk will focus especially on the following features:

Secure Data Storage Secure Password Storage Secure communication Jailbreak detection Defensive tricks

- Talk and paper can be downloaded from `http://grehack.org`

Practical iOS Apps hacking

Can we trust vendors to secure our data?

Mathieu RENARD
Sogeti ESEC / GotoHack.org
Paris, FRANCE
mathieu.renard[-AT-]gotohack.org

This paper demonstrates how professional applications like, Mobile Device Management (MDM) Client, Confidential contents manager (Sandbox), professional media players and other applications handling sensitive data are attacked and sometimes easily breached.

Readers will gain a basic understanding of how these attacks are executed, and many examples of how to code more securely in ways that will not leave applications exposed to such attacks.

I. INTRODUCTION

Gone are the days when employees only used a company-issued phone for work related matters. Today, employees bring personal smart phones and tablets to the office and often have access to sensitive company information on these devices.

This paper is the result of one-year pentesting iOS application and is designed to demonstrate many of the techniques attackers use to manipulate iOS applications in order to extract confidential data from the device.

Then, Jailbreak detection features are analyzed before discussing the results of tests launched on professional applications like, Mobile Device Management (MDM) Client, Confidential contents manager (Sandbox), professional media players and other applications handling sensitive data.

Finally the author proposes mitigation techniques to implement in order to avoid common mistakes that leave applications exposed.

II. ATTACKING IOS APPLICATIONS

Most of the time attacking iOS application is synonym to jailbreak an iDevice, decrypt the application and reverse the binaries. Before developing these items there is some interesting points to linger on, especially on regular devices.

A. What attackers can do without jailbreaking the device

Without having access to the file system it is impossible de decrypt and reverse iOS applications installed from Apple App Store. Nevertheless, this section present attacks vectors that can allow retrieving confidential information stored by miss implemented iOS application.

1) *Using afc protocol to retrieve data stored on the device*
Apple File Communication Protocol (AFC) is a serial port protocol that uses a framework called MobileDevice that is installed by default with iTunes. Since 2010 this protocol is implemented in the libimobiledevice[8] open-sources project. The protocol uses the USB Port and cable when it is connected to the computer and is responsible for things such as copying music and photos and installing firmware upgrades.
AFC Clients like iTunes are allowed access to a "jailed" or limited area of the device memory. Actually, AFC clients can only access to certain files, namely those located in the Media and User installed applications folders. In other words, using AFC client a user/attacker can download the application resources and data. Including the default preferences file where sometimes credentials are stored. The only requirement is the device has to be unlocked. But this is definitively not a problem because an evil maid cans backdoor any iDevice Dock Station.

Figure 1 iPown Dock: Malicious dock station.

2) Retrieving data from backups

The main function of the backup is to permit user to restore personal data and settings to an iPhone during a Restore (during which the content on the iPhone is typically erased).

When the iPhone is connected to a computer and synced with iTunes, iTunes automatically creates a folder with device UDID (Unique device ID – 40 hexadecimal characters long) as the name and copies the device contents to the newly created folder. Most of the time this process is automatic. If the automatic sync option is turned off in iTunes, the user has to manually initiate the backup process through the iTune interface.

TABLE I. BACKUPS FILE PATH

System	Backup Path
Windows 7	C:\Users\(username)\AppData\Roaming\Apple Computer\MobileSync\Backup\
Mac OSX	/Users/(username)/Library/Application Support/MobileSync/Backup/

Since the sync option is defined on the computer side, no user interaction except unlocking the device is required. This implementation allows malicious dock station to initiate backups without user authorization. Performing such attack an attacker may retrieve personal and confidential data like copies of SMS, Call Logs, application data, default preferences and data stored in the keychain.

Keychain class keys define whether a keychain item can be migrated to other device or not. List of protection classes available for the keychain items are shown in the table bellow.

TABLE II. KEYCHAIN CLASS KEYS

Protection class	Description
kSecAttrAccessibleWhenUnlocked	Keychain item is accessible only after the device is unlocked
kSecAttrAccessibleAfterFirstUnlock	Keychain item is accessible only after the first unlock of the device to till reboot
kSecAttrAccessibleAlways	Keychain item is accessible even the device is locked
kSecAttrAccessibleWhenUnlocked ThisDeviceOnly	Keychain item is accessible only after the device is unlocked and the item cannot be migrated between devices
kSecAttrAccessibleAfterFirstUnlock ThisDeviceOnly	Keychain item is accessible after the first unlock of the device and the item cannot be migrated
kSecAttrAccessibleAlways ThisDeviceOnly	Keychain item is accessible even the device is locked and the item cannot be migrated

TABLE III. PROTECTION CLASSES FOR BUILT IN ITEM

Application & Item type	Protection class
WiFi Password	Always
IMAP/POP/SMTP accounts	AfterFirstUnlock
Exchange Accounts	Always
VPN	Always
LDAP/CalDAV/CardDAV Accounts	Always
iTunes backup password	WhenUnlocked ThisDeviceOnly
Device Certificate & private Key	AlwaysThisDeviceOnly

Using the iphonedataprotection[1] tools developed by Jean-Baptiste Bédrune and Jean Sigwald of Sogeti ESEC, it is possible to extract all data stored in the keychain. Nonetheless, only data stored without the ThisDeviceOnly protection class can be extracted without requiring any jailbreak.

Notice: Extracting data stored with the ThisDeviceOnly protection class require to previously extracting the 0x835 key the attack is detailed in the next section.

3) Monitoring communication

Monitoring communication can highlights lack of encryption allowing unsecured credential gathering.

Starting iOS 5, apple added a remote virtual interface (RVI) facility that allows capturing traces from an iOS device. On Mac OSX the virtual interface can be enabled with the rvictl command.

```
$ rvictl -s
454b673c547582234decef5ef3abce6765506af45

Starting device
454b673c547582234decef5ef3abce6765506af45
[SUCCEEDED]

$ # network interface, rvi0, added by the
previous command.
$ ifconfig -l
lo0 gif0 stf0 en0 en1 p2p0 fw0 ppp0 utun0 rvi0

$ sudo tcpdump -i rvi0 -n
listening on rvi0, link-type RAW (Raw IP),
capture size 65535 bytes
…
```

Figure 2 Enabling iOS virtual interface on OSX.

On other system this can be done using the com.apple.pcapd service through the usbmux[8] deamon.

4) Attacking secure communications to servers

Almost every application handling sensitive data will connect back to some server component. Developers are, thus, faced with the challenge of having to protect sensitive data in transit as it traverses the Internet and sometimes even insecure wireless media. This can be done using encryption but must be implemented correctly.

This is why, developers must take care when using the URL loading library. According to the security best practices, the default state of operation for the URL loading library is to fail on an invalid server certificate. However, during development it is often required to use an invalid certificate. Failure to use the libraries properly can result in weak client to server communications that attackers may compromise by setting a transparent proxy (for example on a fake Wi-Fi access point). This is why, it is really important to check this point before production launch.

Nevertheless, the default SSL Warning message can be bypassed by installing a fake certificate authority in the apple certificate store. On a regular device, this cannot be done without user interaction. However, prior iOS6, SMS applications only displayed the reply-to field. This allows attackers to send fake configuration message spoofing the reply-to field [28].

B. Installing the application on a jailbroken device

Apple designed the iPhone platform with the intent to control all software that is executed on the device. Thus, the design does not intend to give full system (or root) access to a user.
Moreover, only signed binaries can be executed. In others words, the loader will not execute either unsigned binaries or signed binary without a valid signature from Apple. This ensures that only unmodified Apple-approved applications are executed on the device.

The term jailbreaking refers to a technique where a flaw in the iOS operating system is exploited to unlock the device, thereby obtaining system-level (root) access. With such elevated privileges, it is possible to modify the system loader so that it accepts any signed binary, even if the signature is not from Apple. In others words, the loader will accept to launch every signed binaries even if it is not signed with Apple certificate.

1) Retrieving user password & keychain content

Jailbreaking also allows malicious user to retrieve application and data stored on the device. When a device is Jailbroken, the confidentiality of the data and information returned by the systems call cannot be trusted.
Moreover, jailbreaking allows users to install an SSH service, which is often left in a de facto unsecure state. Remember: Worm:iPhoneOS/Ikee the first worm which was targeting the Apple Jailbroken iPhone:

- The first version most notable action involved changing the background wallpaper on the device.
- The second version the worm was accessing user's computing device and changing their data without permission.

Running critical application on a jailbroken device may allow attackers to retrieve data such as encryption keys and credentials even when stored in the Keychain.
In the iPhoneDataProtection framework, Jean-Baptiste Bédrune and Jean Sigwald implemented a tool named "KeychainViewer"[1] allowing browsing the keychain content by directly accessing the keychain database.

Figure 3 Browsing Keychain with KechainViewer.

2) Retrieving the 0x835 Key

Browsing the keychain content on a jailbroken is not really difficult. On the opposite, extracting all data including data stored within the ThisDeviceOnly protection class form a backup require extracting the 0x835 key. The 0x835 key is generated by encrypting 01010101010101010101010101010101 with the UID-key (Hardware key). Hardware keys can only be accessed from kernel. Therefore, IOAESAccelerator kernel service has to be patched in order to allow keys access from user land.

The iPhoneDataProtection framework embeds tools allowing patching the kernel and retrieving the 0x835 key.

```
iPad:~ root# chmod +x device_infos
iPad:~ root# chmod +x kernel_patcher
iPad:~ root# ./kernel_patcher
Found IOAESAccelerator UID ptr at 80473a24
vm_write into kernel_task OK
```

Figure 4 Patching IOESAccelerator.

C. Reversing Objective-C Binaries

iOS executables are ARM binaries and use the Mach-O binary file format.

1) Faiplay encryption

The primary obstacle to overcome in reversing iOS binaries from the App Store is that all published applications are encrypted using Apple's binary encryption scheme. When an application is synchronized onto the iDevice, iTunes extracts the application folder from the archive (bundle) and stores it on the device. Furthermore, the decryption key for the application is added to the device's secure keychain. This is required because the application binaries are stored in encrypted form (When an application is encrypted the cryptid is set to 1).

Here the benefit of jailbreaking is that the user obtains immediate access to many development tools ready to be installed on iOS, such as: debugger and disassembler.

This makes the decryption step quite straightforward:

- The application is launched in the debugger.
- A breakpoint is set to the program entry point. Once this breakpoint triggers, the attacker knows that the system loader has verrified the signature and performed the decryption.
- The memory region that contains the now decrypted code is dumped.
- The binary encrypted part is replaced by the dumped one.
- The CryptID is redefined to 0.

Tools like Crackulous[2] are making this task easier since they allow decrypting / cracking iOS application in one click. Unfortunately Crackulous v1.0.0.5 does not handle the decryption of thin binary (Binary file compiled for one processor architecture only).

Figure 5 Cracking application with crackulous.

Fortunately (from the auditor point of view), Stefan Esser Aka i0nic, published a tool called: dumpdecrypted[3] producing a decrypted version of the application to analyze when loaded with. Nevertheless, in order to limit iOS application cracking, the tool does not unset the cryptic after decryption. Which mean, that the flag has do be unset manually after decryption. Since this step is not mandatory for a security analysis it will not be discussed here.

2) Objective-C & Objc_msgSend

Objective-C is the most prevalent programming language used to create applications for the iOS platform. In Objective-C methods are not called but instead a so-called message is sent to a receiver object. These messages are handled by the dynamic dispatch routine called objc_msgSend.

This dispatch routine is responsible for identifying and invoking the implementation for the method that corresponds to a message. The first argument is always a pointer to the called object. That is, the object on which the method should get invoked (for example, an instance of the class NSString). The second argument is a called selector. The selector is a string representation of the name of the method that should get invoked (for example length). All remaining arguments are passed to the target method once it is resolved.
To perform this resolution, the obj_msgSend function walks the class hierarchy starting at the receiver and searches for a method whose name corresponds to the selector. If no match is found in the receiver class, its superclasses are searched recursively. Once the corresponding method is identified, objc_msgSend invokes the method and passes along the necessary arguments.

3) Retreiving classes headers

Since many applications for iOS are developed in Objective-C, the Mach-O format supports specific sections, organized in so-called commands, to store additional meta-data about Objective-C programs.

The __objc_classlist section contains a list of all classes for which there is an implementation in the binary. The __objc_classref section, on the other hand, contains references to all classes that are used by the application including imported classes. It is the responsibility of the dynamic linker to resolve the references in this section when loading the corresponding library. Others sections include information about categories, selectors, or protocols used or referenced by the application.
Apple has been developing the Objective-C runtime as an open-source project. Thus, the specific memory layout of the involved data structures can be found in the header files of the Objective-C runtime.

One can rebuilt basic information about the implemented classes traversing these structures in the binary. Using "class-dump", the commercial version of IDA pro (starting 6.2) or using the IDA plugins like zynamics/objc-helper-plugin-ida[11] within the IDA free version it is trivial to retrieve these information.

```
@interface UserPasswordChange : XXUnknownSuperclass <XXEncryptedProtc
 {
@private
        ? userInfo;
        ? userName;
        ? password;
        ? comfirmPassword;
        ? keyboardBar;
        ? currentResponder;
}
-(?)tableView:(?)view cellForRowAtIndexPath:(?)indexPath;
-(?)tableView:(?)view numberOfRowsInSection:(?)section;
-(?)numberOfSectionsInTableView:(?)tableView;
-(?)shouldAutorotateToInterfaceOrientation:(?)interfaceOrientation;
-(?)viewDidDisappear:(?)view;
-(?)viewWillDisappear:(?)view;
-(?)viewDidAppear:(?)view;
-(?)viewWillAppear:(?)view;
```

Figure 6 Analyzing iOS binarie with classdump.

4) Where to start ?

To start the analysis in the right way we have to locate the main class. The UIApplicationDelegate protocol declares methods that are implemented by the delegate of a UIApplication object. These methods provide information about key events in an application's execution such as when it finished launching, when it is about to be terminated, when memory is low, and when important changes occur. Finding one of the following methods: ApplicationDidFinishLaunching, ApplicationDidFinishLaunchingWithOptions, Application*… is a good way to find out which view is launched first.

Regarding the views initialization, The UIViewController class provides specific methods that are called when specific events occur. When trying to follow the execution patch the main event to focus our intention is viewDidLoad that is called after views initialization.

5) Where to look ?

The list of points to focus on when reversing iOS Application is related to the features of the application to be analyzed. Here is a list of object that may have an interest regarding security matters.

TABLE IV. INTERESTING OBJECTS, CLASSES & METHODS

Use case	Objects / Classes / Methods
URL Handling	NSURL*
Socket Handling	CFSocket*
Keychain	ksecAttr*, SecKeychain*
Files Handling	NSFileManager*
Crypto	CCCrypt*

D. Dynamic analysis

There are many different approaches to dynamic analysis. In this section we will focus on the MobileSubstrate[6] framework.

1) Introducing Mobile substrate

MobileSubstrate[6] is a framework that allows developers to provide run-time patches ("MobileSubstrate extensions") to system functions. MobileSubstrate can easily install on jailbroken device through Cydia[6]. The framework consists of three major components:

- MobileHooker is used to replace system functions.
- MobileLoader is used to automatically load Mobilesubstrate extension at application launch. MobileLoader will first load itself into the run application using DYLD_INSERT_LIBRARIES environment variable. Then it looks for all dynamic libraries in the directory /Library/MobileSubstrate/DynamicLibraries/, and dlopen them.
- Safe Mode: When a extension crashed the SpringBoard, MobileLoader will catch that and put the device into safe mode menaing that all 3rd-party extensions will be disabled.

In order to define a hook the developer can use two functions:

- MSHookMessageEx() will replace the implementation of the Objective-C message by replacement, and return the original implementation. This dynamic replacement is in fact a feature of Objective-C, and can be done using method_setImplementation.
- MSHookFunction() is like MSHookMessageEx() but is for C/C++ functions. Conceptually, MSHookFunction() will write instructions at assembly level that jumps to the replacement function, and allocate some bytes on a custom memory location, which has the original cut-out instructions and a jump to the rest of the hooked function. Since on the iPhoneOS by default a memory page cannot be simultaneously writable and executable, a kernel patch is applied for MSHookFunction() to work.

Using this Framework attacker can easily trace and dynamically patch the application at runtime. Here is an example of jailbreak detection bypass:

```
static int (*old_system)(char *) = NULL;
int st_system(char * cmd){
    if (!cmd){
        return nil;
    }
  return old_system(cmd);
}
__attribute__((constructor)) static void
initialize() {
  MSHookFunction(system, st_system,&old_system);
}
```

Figure 7 Bypassing Jailbreak detection.

2) Attacking network communication

Hooking the NSURLConnection and setting a proxy on the device it is possible to silently still the credentials and all the data exchanged with the server even when transmitted through HTTPS.

iOS SSL Kill Switch[5] is a MobileSubstrate[6] extension developed by iSECPartners. This extension allows disabling certificate validation in order to facilitate black box testing of iOS Apps. Once installed on a jailbroken device, the extension patches NSURLConnection to override and disable the system's default certificate validation.

3) Stealing cryptos keys

Hooking the CCCrypt(3cc) API it is possible to silently still the crypto keys used on native iOS application. This CCCrypt(3cc) API provides access to a number of symmetric encryption algorithms. Most of the time the application are directly calling the CCCrypt() function. CCCrypt() is a stateless, one-shot encrypt or decrypt operation.

```
CCCrypt(CCOperation op, CCAlgorithm alg,
        CCOptions options, const void *key,
        size_t keyLength, const void *iv,
        const void *dataIn, size_t dataInLength,
        void *dataOut, size_t dataOutAvailable,
        size_t *dataOutMoved);
```

Figure 8 CCCrypt API definition.

III. JAILBREAK DETECTION FEATURES [THE TRUTH]

Jailbreak detection features are implanted in order to detect when an end user has compromised their device, or to detect whether an intruder has compromised a stolen device. For example, all MDM application embeds jailbreak detection features.

The following sections present common and uncommon jailbreak detection features highlighted during one year studying iOS application security.

A. Checking for jailbreak files

This is the most common check performed on the application we analyzed. Usually applications are checking for files like: "/Applications/Cydia.app", "/bin/apt","/usr/sbin/sshd"…

```
+ (BOOL)doCydia {
    if ([[NSFileManager defaultManager]
      fileExistsAtPath: @"/Applications/Cydia.app"]){
      return YES;
    }
    return NO;
}
```

Figure 9 Checking for jailbreak files.

Launching a simple "strings" on the application binary can highlight this test. Nonetheless, sometimes developers use dynamic string generation and obfuscation tricks in order hide checked files. Anyway, this test can be bypassed by hooking NSFileManager methods.

B. Checking if system partition is writable

On a regular device the system partition is mounted with the read only attribute. After jailbreaking a device with the public jailbreak tool Absinthe [4], the system partition remains writable. This changes are made by replacing the /etc/fstab file. The file is commonly 80 bytes for all iOS version, whereas the copy of the file installed by the public jailbreak tool: Absinthe is only 65 bytes.

```
+ (BOOL)doFstabSize {
    struct stat sb;
    stat("/etc/fstab", &sb);
    long long size = sb.st_size;
    if (size == 80){
       return NO;
    return YES;
}
```

Figure 10 Checking /etc/fstab size.

This test can easily be bypassed by hooking the stat system call within a mobile substrate extension.

C. Checking for shell

By default no shell is available on regular device but it comes with the public jailbreak. This is why this test aims to detect if a shell is available on the device by calling system(0). If the value of command is NULL, system() returns nonzero if the shell is available, and zero if not.

```
+ (BOOL)doShell {
        if (system(0)) {
            return YES;
        }
        return NO;
}
```

Figure 11 Checking /etc/fstab size.

This test can easily be bypassed with a mobile substrate extension (See Figure 7).

D. Checking for signer identity (Another common test)

Most of the applications cracked with “Crackulous” [2] come with a SignerIdentity key added in the Info.plist file bundled with the application.

```
+ (BOOL)doSignerIdentity {
  NSBundle *bundle = [NSBundle mainBundle];
  NSDictionary *info = [bundle infoDictionary];
  if ([info objectForKey: @"SignerIdentity"] != nil){
        NSLog(@"App have has been hacked");
        return YES;
  }
  return NO;
}
```

Figure 12 Checking for SignerIdentity.

This test is designed to overcome automated processes at best, and will probably only defeating most tutorial-followers. An attacker can hexedit the binary file and as such, could edit the string @"SignerIdentity" to read @"siNGerIDentitY" or something else which would return nil, and thus pass. This test can also by bypassed by hooking objectForKey and return nil.

E. Less commons Jailbreak checks

Less commons jailbreak checks are using system calls:

- Fork(): Documented in some books and blog posts: If the process can fork, the device is jailbroken. Except this check producing a lot of logs in the console, and most important does not work because the jailbreak does not patch this part of the sandbox. See the iPhone Wiki [12] for details about jailbreak patchs.
- Open(): Trying to open a file in write mode in a not writable path outside the sandbox: if no error the device is not jailbroken.

Like other jailbreak detection functions presented in this section system calls can easily being hooked in order to hide the jailbreak.

F. Conclusion: Jailbreak detection = Fail by design!

Despite this test are interesting and probably stops most of the script kiddies, tutorial follower and automating tools, skillful attackers can bypass them.
The thing is that Apple does not provide any API to launch action either before or after the installation. As a result attackers are able to decrypt and analyze applications before they could launch their jailbreak detection tests. Moreover after jailbreak the attackers have root access to the device, which means to control everything on the device as the opposite of iOS application.

Nevertheless, when well implemented, jailbreak detection features can discourage most of script kiddies and tutorial followers.

IV. Real world apps & Security worst practices

This section present the worst practices highlighted during 1-year pentesting iOS Application used in professional environment.

A. Unsecure password storage

Some applications are using the NSUserDefaults standardUserDefaults method in order to user credentials. The problem is that standardUserDefaults stores information in plain text in a plist file that can be downloaded trough AFC protocol.

Key	Type	Value
▼ Root	Dictionary	(1 item)
▼ Accounts	Array	(1 item)
▼ Item 1	Dictionary	(6 items)
AccountEnabled	Boolean	☑
Domain	String	example.com
FullName	String	John Smith
Realm	String	*
Username	String	1234567
Password	String	secret

Figure 13 Default plist file including clear text stored password

B. Authentication Bypass

Here the password defined by the user is stored on the file system in an encrypted database. The problem is that the application has to decrypt the database before the user being authenticated in order to check the password validity. Since the database is decrypted before the user was authenticated it is possible for an attacker, having an access to an unlocked jailbroken device to retrieve the password in the memory.

C. Unsecure data storage

This example was highlighted during the analysis of a sandbox like application. According to the documentation the application is using "high grade encryption” to secure the document.

iExplorer is an iPhone browser or iPad file explorer that runs on Mac & PC. iExplorer lets users browse the files and folders on their iDevice as if it were a normal USB flash drive or pen drive (this application does not require any jailbreak). Using this tool it is possible to download all the application resources and data. The analysis of the data highlights the lack of encryption.

In this case, when the vendor says "high grade encryption” you must read: All data are stored on the iPhone encrypted file system that provides high-grade encryption.

D. Extraction data from log

This application embeds a secure web browser, according to the best practices; redirect all the network traffic through an SSL Tunnel (even HTTP traffic is redirected trough this tunnel). However, all the cookies used during the user navigation are exported in the application logs. Application logs are available to any applications installed on the device. A malicious application could use these cookies in order to impersonate user sessions and access/steal confidential data.

E. Hardcoded encryption key – Common!

This application is secure media player allowing to play / view protected content. Here is the pseudo code of the images decryption routine.

```
Base64Key = (int)objc_msgSend(
                    *classRef_NSString_Ptr,
                    *selRef_stringWithFormat_Ptr,
                     CFSTR("HiddenTreasures"));
                     NSData = &classRef_NSData;

Key = objc_msgSend(&OBJC_CLASS___NSData,
                    "dataFromBase64String:",
                     Base64Key);

BundlePath = objc_msgSend(&OBJC_CLASS__NSBundle,
                    "mainBundle");

cpngPath = (int)objc_msgSend(BundlePath,
                     "pathForResource:ofType:"
                     filename,
                     CFSTR("cpng"),
                     1063452672);
Data = *NSData;
cpngFileContent = (int)objc_msgSend(Data,
                     dataWithContentsOfFile:",
                     cpngPath);

decyptedContent = (int)objc_msgSend(
                    &OBJC_CLASS___FBEncryptorAES,
                    "decryptData:key:iv:",
                     cpngFileContent,
                     Key,
                     0);
```

Figure 14 Image decryption function pseudo code

Here the Key/Password: “Hiddentreasures” is Base64 decoded before being used as a key for the AES decryption algorithm. Moreover the IV used by the AES crypto function is fixed to “0”. With this information an attacker can easily be re-implement the algorithm and decrypt the data.

F. Playing DRM video with MPMoviePlayerControler

This application was using the apple “MPMoviePlayerControler” API to play encrypted content stored on the device. In other words the application was localy streaming the files.

The “MPMoviePlayerControler” is a part of Apple API’s. Apple developer’s documentation says:

- A movie player (of type MPMoviePlayerController) manages the playback of a movie from a file or a network stream.
- When encryption is employed, references to the corresponding key files appear in the index file so that the client can retrieve the keys for decryption.
- When a key file is listed in the index file, the key file contains a cipher key that must be used to decrypt subsequent media files listed in the index file.
- Currently HTTP Live Streaming supports AES-128 encryption using 16-bytes keys. The format of the key file is a packed array of these 16 bytes in binary format”.

All an attacker needs to play the video on another device is the index file; the key and the encrypted video, which in this case are stored in, clear text on the file system and can be retrieved through AFC protocols.

```
#EXTM3U
#EXT-X-TARGETDURATION:63
#EXT-X-VERSION:2
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:63,
#EXT-X-KEY:METHOD=AES-
128,URI="http://localhost:12345/crypt5.key",IV=0cd463
4ed46bbc1e8235e21b23dc6792e3

http://localhost:12345/fileSequence5.ts

#EXT-X-ENDLIST
```

Figure 15 MPMoviePlayerControler index file

This allows an attacker to develop its own movie player to read the videos files extracted/dumped from the Ipad.

V. Defending iOS application

In matter of security the iOS system is not perfect. Even if Apple increases the security level of its mobile operating system, for each new release comes a new jailbreak. Jailbreaking is a process that allows users to gain the root access to the command line, decrypt, analyze and crack iOS application.

In this section we will present some defensives tricks, which can be use to the aim to slow down skilful attackers, discourage script kiddies and defeat automatic tools.

A. Anti-analysis part 1

It is possible to add an anti-debugging feature by sending a non-standard ptrace value named PT_DENY_ATTACH. Setting this value allows a process that is not currently being traced to deny future traces by its parent. All others arguments are ignored. An attempt by the parent to trace a process, which has set this flag, will result in a segmentation violation in the parent.

```
#import <dlfcn.h>
#import <sys/types.h>
#define PT_DENY_ATTACH 31

typedef int (*ptrace_ptr_t)(int _request, pid_t _pid,
caddr_t _addr, int _data);

void disable_gdb() {
  void* handle = dlopen(0, RTLD_GLOBAL | RTLD_NOW);
  ptrace_ptr_t ptrace_ptr = dlsym(handle, "ptrace");
  ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);
  dlclose(handle);
}
```

Figure 16 Disabling GDB with PTRACE_DENY_ATTACH

It is useful for defeating most tutorial-followers but this is no guarantee that your application cannot be debugged, and in fact there are ways around this. An attacker, can set a breakpoint within the application prior to issuing a run from within a debugger, and specify that the debugger run any commands he wants when ptrace starts and before the application can shut down.

Here is an example of a GDB script to bypass PT_DENY_ATTACH system call:

```
break ptrace
commands 1
   return
   continue
end
```

Figure 17 Bypassing PTRACE_DENY_ATTACH

It is useful for defeating most tutorial-followers but this is no guarantee that your application cannot be debugged, and in fact there are ways around this.

Nevertheless, since the ptrace function is built inside the kernel, the user space interface only performs syscall 26 (ptrace). If the anti-debugging function is inlined like the example bellow the PT_DENY_ATTACH will be installed and there is no way .

```
mov r0, #31
mov r1, #0
mov r2, #0
mov r3, #0
mov ip, #26
svc #0x80
```

Figure 18 Inline vesion of the PTRACE_DENY_ATTACH test

Anyway a dedicated and skillful attacker can patch the kernel/application.

B. Anti-analysis 2

When an application is being debugged, the kernel sets the P_TRACED flag for the process signifying that the process is being traced. Applications can monitor the state of this flag. If this flag is set, the application knows that it was either started with a debugger, or a debugger was later attached to it.

When the application detect it is being debugged, the program should silently wipe all confidential data and encryption keys and then inform the user.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/sysctl.h>
#include <string.h>
#define P_TRACED 0x00000800

static int checkGDB() __attribute__((always_inline));

int checkGDB()
{
  size_t size = sizeof(struct kinfo_proc);
  struct kinfo_proc info;
  memset(&info, 0, sizeof(struct kinfo_proc));

  int ret, name[4];
  name[0] = CTL_KERN;
  name[1] = KERN_PROC;
  name[2] = KERN_PROC_PID;
  name[3] = getpid();

  if (ret = (sysctl(name, 4, &info, &size, NULL, 0)))
      return ret;
  return (info.kp_proc.p_flag & P_TRACED) ? 1 : 0;
}
```

Figure 19 Checking the P_TRACED flag

This technique will only allow the application to detect when gdb, or another debugger, is attached to the process, but will not detect when malicious code is injected, or when other tools that do not trace are attached to the process. Implementing this in your code will only force an attacker to either avoid using a debugger (which will further complicate things for him), or to locate and patch the debugging checks. Moreover a skillful attacker could also patch out the invocation of sysctl itself. This is why simple's sanity checks should be done to ensure that sysctl can return other data, and to ensure that the call does not fail. This will help further complicate the attack and require the attacker to properly populate the kinfo_proc structure with valid information.

C. Preventing Hooking

Hooking allowing attackers to alter or augment the behaviour of applications. By implementing the following defensives measures allowing ensuring that called function are the ones implemented in the application.

1) Validating Address Space

Any time malicious code is injected into an application, it is loaded into the application address space. Validating the address space for critical methods used by the application force the attacker to find ways to inject his code into the existing address space.

The dynamic linker library includes a function named dladdr. The function dladdr() takes a function pointer and tries to resolve name and file where it is located. Information is stored in the Dl_info structure.

```
typedef struct {
  const char *dli_fname;  /* Pathname of shared
                             object that
                             contains address */
  void *dli_fbase;   /* Shared object Address */
  const char *dli_sname; /* Name of nearest
                            symbol with address
                            lower than addr */
  void *dli_saddr;       /* Exact address of
                            symbol named in
                            dli_sname */
} Dl_info;
```

Figure 20 Dl_info structure

By providing the structure with the function pointer of a class's method implementation, its origins can be verified.

```
#include <dlfcn.h>
#include <objc/objc.h>
#include <objc/runtime.h>
#include <stdio.h>
#include <string.h>

static int checkAddressSpace
__attribute__((always_inline));

int checkAddressSpace(NString MyCriticalClass ,
                      NSString MyCriticalMethod){
  Dl_info info;
  IMP imp = class_getMethodImplementation(
            objc_getClass(MyCriticalClass),
            sel_registerName(MyCriticalMethod));

if (dladdr(imp, &info)){
      /* Do some additional tests:
         Pathname of shared object … */
      return 1;
  } else {
      NSLog("Error: cannot find %@ symbol",
            MyCriticalMethod);
      return 0;
  }
}
```

Figure 21 Checking address space

2) Inlining

iOS offers a way to override functions in a shared library with DYLD_INSERT_LIBRARIES environment variable (which is similar to LD_PRELOAD on Linux). On a jailbroken device the MobileSubstrate framework simplify this task and allows developers to easily load libraries at application launch.

Inline functions are functions in which the compiler expands a function body to be inserted within the code every time it is called. In other words, there is no longer a function: the code gets pasted into the machine code whenever it is called. Turning the critical functions into inline ones will cause it to be repeated throughout the application every time it is called. This rise up attacks complexity by forcing an attacker to hunt down every occurrence of code and patch it.

To be inlined a function must be declared within the attribute __attribute__((always_inline));

```
static int isPasswordValid(char * pwd)
__attribute__((always_inline));
int isPasswordValid(char * pwd) {
    //Function body
}
```

Figure 22 Defining inline attribute

In addition of this attribute the following two compilations flags should be enabled:

-finline-functions
-Winline

D. Others binary protection

iOS Applications are not exempt of overflow vulnerabilities this is why the following mitigating technics should be implemented in every application.

1) Stack smashing protection

It is possible to activate stack-mashing protection at compilation time. This can be achieved by specifying the –fstack-protector-all compiler flag.
When an application is compiled with this protection, a known value called "canary" is placed on the stack before the local variables to protect the saved base pointer, saved instruction pointer and function arguments. The value of the canary is verified upon the function return to see if it has been overwritten.

One can identify the presence of stack canaries examining the symbol table of the binary, if stack-smashing protection is compiled in to the application, two undefined symbols will be present:

- ___stack_chk_fail
- ___stack_chk_guard

2) Automatic Reference Counting

Automatic Reference Counting (ARC) was introduced in iOS SDK version 5.0 to move the responsibility of memory management from the developer to the compiler. Consequently, ARC also offers some security benefits as it reduces the likelihood of developers introducing memory corruption (specifically object use - after-free and double free) vulnerabilities in to applications. ARC can be enabled in an application within XCode by setting the compiler option "Objective-C Automatic Reference Counting" to "yes". This option is automatically check statring XCode 4.3.

3) Binary obfuscation

Main purpose of code obfuscation and other protections applied to source code or resulting binaries is to prevent reverse engineering and cracking. Unfortunately, there are no popular, well-known tools for Objective C code obfuscation. Objective C is a dynamic language, based on message passing paradigm, where most of bindings are resolved run time. Therefore it is always possible for attacker to track, intercept and reroute calls, even with obfuscated names.

Nevertheless, adding some obfuscation to the binaries will slow down the analysis. Since no open source tool exists to perform this task automatically the developer has to implement the obfuscation himself.
Symbol stripping and dynamic string generation should be implemented. Especially if the application is checking for Jailbreak files and if it informs the user they're using a cracked version. Actually, when strings are stored in plain text, the crackers can quickly track down where the view is generated with strings and disable the check.

For example the dynamic strings generation can easily being implemented by using a crypto algorithm. In this case strings are decrypted on the fly just before being used and cleared from memory after use. In addition of these basic tricks it may be interesting to rename classes and methods with random names.
Using basic blocks cloning technics and inserting opaque predicate will also increase the binary obfuscation level.
Basic blocs' cloning allows spreading the execution across the cloned basic blocks.
Inserting opaque predicate add extra tests which cannot be easily proved to conditionals.

The obfuscation should be performed semi-automatic, over source code copy, with tool custom developed for such task. Obfuscated code is by definition hard to read before compilation as same as after decompiling from binaries.

Anyway, it important to keep in mind that obfuscation wills only slowdown attackers performing static analysis only. It is only a matter of time before an attacker mixing static and dynamic analysis will be able to reverse your application.

E. Security of running memory

The following guidelines can help to improve the security of running memory:

- Never store anything in memory until the user has authenticated and data has been decrypted. It should not even be possible to store passwords, credentials, or other information in memory before a user has entered their passphrase; if it is, the application is not properly implementing encryption.
- Do not store encryption keys or other critical data inside Objective-C instance variables, as they can be easily referenced. Instead, manually allocate memory for these. This will not stop an attacker from hooking into your application with a debugger, but will up the ante for an attacker. Typically, if a device is compromised while the user is using it, the attack is automated malware rather than an active human.

VI. Conclusion

Regarding security most of iOS applications are not mature!

Developers should apply the following recommendation in order to mitigate the risks.

- Do not relay only on iOS security,
- Do not store credential using standardUserDefaults method.
- Encrypt your data even when stored in the keychain,
- Do not store crypto keys on the device,
- Check your code, classes, functions, methods integrity,
- Detect the jailbreak,
- Properly implement cryptography in applications (simple implementation are the most secure),
- Remove all debug information from the final release,
- Minimize use of Objective-C for critical functions & security features.

Users and companies should not blindly thrust iOS application vendors when talking about security.

References

[1] iPhoneDataProtection - Jean-Baptiste Bédrune and Jean Sigwald,

[2] Crakulous - Angel, http://hackulo.us

[3] Dumpdecrypted – Stefan Esser – i0n1c, https://github.com/stefanesser/dumpdecrypted

[4] Absinthe - Chronic-Dev Team and iPhone Dev Teams (Jailbreak Dream Team), http://greenpois0n.com

[5] iOS SSL Kill Switch – iSECPartners, https://github.com/iSECPartners

[6] MobileSubstrate, Cydia – Sauric, http://iphonedevwiki.net/index.php/MobileSubstrate, http://cydia.saurik.com/

[7] iExplorer - Macroplatant, http://www.macroplant.com/iexplorer/

[8] libimobiledevice & usbmuxd - Nikias, http://www.libimobiledevice.org/

[9] Gutmann method, http://en.wikipedia.org/wiki/Gutmann_method

[10] iPhone security model & vulnerabilities : http://esec-lab.sogeti.com/dotclear/public/publications/10-hitbkl-iphone.pdf

[11] zynamics/objc-helper-plugin-ida - https://github.com/zynamics/objc-helper-plugin-ida

[12] Sandbox patch, http://theiphonewiki.com/wiki/index.php?title=Sandbox_Patch

[13] Evolution of iOS Data Protection and iPhone Forensics: from iPhone OS to iOS 5: https://media.blackhat.com/bh-ad-11/Belenko/bh-ad-11-Belenko-iOS_Data_Protection.pdf

[14] Overcoming iOS data protection to re-enable iPhone Forensics: https://media.blackhat.com/bh-us-11/Belenko/BH_US_11_Belenko_iOS_Forensics_Slides.pdf

[15] Apple iOS Security Evaluation: http://hakim.ws/BHUS2011/materials/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf

[16] New age application attacks against Apple iOS and countermeasures: https://media.blackhat.com/bh-eu-11/Nitesh_Dhanjani/BlackHat_EU_2011_Dhanjani_Attacks_Against_Apples_iOS-WP.pdf

[17] Hacking and Securing Next Generation iPhone and iPad Apps: http://software-security.sans.org/downloads/appsec-2011-files/dhanjani-hacking-securing-next-gen.pdf

[18] Secure Development on iOS – Advice for developers and penetration testers: http://www.isecpartners.com/storage/docs/presentations/iOS_Secure_Development_SOURCE_Boston_2011.pdf

[19] Pentesting iPhone & iPad Apps: http://www.hackinparis.com/slides/hip2k11/07-Pentesting_iPhone_iPad.pdf

[20] Penetration testing of iPhone/iPad applications: http://www.mcafee.com/us/resources/white-papers/foundstone/wp-pen-testing-iphone-ipad-apps.pdf

[21] Practical Consideration of iOS Device Encryption Security: http://sit.sit.fraunhofer.de/studies/en/sc-iphone-passwords.pdf

[22] iPhone 3GS Forensics: Logical analysis using Apple iTunes Backup Utility: http://www.ssddfj.org/papers/SSDDFJ_V4_1_Bader_Bagilli.pdf

[23] iOS Security by Apple: http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf

[24] Corona Jailbreak for iOS 5.0.1 by Dream team: http://conference.hitb.org/hitbsecconf2012ams/materials/D2T2 – Jailbreak Dream Team – CoronaJailbreak for iOS 5.0.1.pdf

[25] Absinthe Jailbreak for iOS 5.0.1 by Dream team: http://conference.hitb.org/hitbsecconf2012ams/materials/D2T2%20-%20Jailbreak%20Dream%20Team%20-%20Absinthe%20Jailbreak%20for%20iOS%205.0.1.pdf

[26] iOS Application Security : http://www.exploit-db.com/wp-content/themes/exploit/docs/18831.pdf

[27] Breaking iOS code signing: http://reverse.put.as/wp-content/uploads/2011/06/syscan11_breaking_ios_code_signing.pdf

[28] Never trust SMS: iOS text spoofing - http://www.pod2g.org/2012/08/never-trust-sms-ios-text-spoofing.html

[29] Mobile certificate pinning (iOS SSL kill switch): http://cloud.github.com/downloads/iSECPartners/ios-ssl-kill-switch/BH2012_MobileCertificatePinning_short.pdf

[30] Overview on Apple iOS Security : http://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/LectureSlides/ESS-SS2012/9_iOS_-_hand-out.pdf

FURTHER READING

[31] Hacking and Securing iOS Applications: Jonathan Zdziarski

[32] iOS Hacker's Handbook : Charlie Miller , Dion Blazakis , Dino Dai Zovi , Stefan Esser , Vincenzo Iozzo , Ralf-Phillip Weinmann

[33] iOS kernel exploitation IOKit edition: http://reverse.put.as/wp-content/uploads/2011/06/SyScanTaipei2011_StefanEsser_iOS_Kernel_Exploitation_IOKit_Edition.pdf

[34] iOS 5 – an exploitation night mare

http://antid0te.com/CSW2012_StefanEsser_iOS5_An_Exploitation_Nightmare_FINAL.pdf

[35] Evolution of iPhone Baseband and unlocks: http://conference.hitb.org/hitbsecconf2012ams/materials/D1T2 – MuscleNerd – Evolution of iPhone Baseband and Unlocks.pdf