A generic process to identify vulnerabilities and design weaknesses in iOS healthcare apps

Christian D'Orazio Kim-Kwang Raymond Choo
Information Assurance Research Group, University of South Australia
christian.dorazio@mymail.unisa.edu.au raymond.choo@unisa.edu.au

Abstract

Due to the capability of mobile applications (or apps, as they are commonly known) to access sensitive data and personally identifiable information (PII) such as medical history and electronic health transactions, they present a genuine security and privacy threat to their users. In this paper, we propose a generic process to identify vulnerabilities and design weaknesses in apps for iOS devices. We validate our process with a widely used Australian Government Healthcare app and revealed previously unknown / unpublished vulnerability that consequently exposes the user's sensitive data and PII stored on the device. We then propose several recommendations with the hope that similar structural mistakes can be avoided in future app design.

1. Introduction

Healthcare is increasingly digitised as patients and healthcare professionals turn to mobile devices and mobile apps for a wide variety of medical uses [13]; [20]. For example, according to the Research2Guidance's Mobile Health Market Report, there are more than 97,000 apps listed on 62 app stores in the calendar year 2013 [21].

This is not surprising as mobile devices (e.g. iOS and Android devices) and mobile apps (e.g. healthcare apps, social networking apps, and VoIP apps) are rapidly become part of everyday life in both developed and developing countries. While there is a wide range of mobile devices, various statistics have suggested that Apple iOS and Google Android are the two dominant platforms for mobile apps [8] [21]. Of these two platforms, iOS is widely regarded to be more secure – the platform this research proposes to address.

In the rush to attract new consumers and accelerate the product's time-to-market, many mobile apps including healthcare apps were not designed with user security and privacy in mind. The situation is somewhat similar to twenty or thirty years ago when cryptographic protocols were routinely published without a rigorous security analysis and, subsequently, found to be insecure [9][10].

A recent study conducted by Hewlett Packard Security Research [14], for example, revealed that 90% of the 2,107 mobile apps examined were vulnerable to attacks, and 97% accessed sensitive data and PII and 86% had privacy-related risks. Lack of binary code protection was identified as a potential vulnerability affecting 86% of the apps examined. Another major vulnerability was weak or inappropriate implementation of cryptographic schemes to secure sensitive data stored by the apps on mobile devices – revealed in 75% of the apps examined [14].

Despite the growing importance of healthcare apps, our survey of the literature suggested that security and privacy of user data in this area is understudied and not widely understood. In recent years, a number of technologically developed countries such as United States have recognised the need to introduce regulatory requirements pertaining to healthcare apps. For example, the U.S. Food and Drug Administration is currently focusing on 'mobile apps [that] meet the definition of medical device in the FD&C Act and their functionality poses a risk to a patient's safety if the mobile app were to not function as intended' [22].

Considering that a user's medical and other sensitive information can be stored and accessed by healthcare apps and on mobile devices, ensuring the security and privacy of such data is of paramount importance (e.g. accidental or intentional leakage of one's medical history can result in the individual being embarrassed or subject to extortion).

Contribution 1: Adopting the approach common in computer security research [10], we propose a generic process to analyse mobile apps for iOS devices (one of the most popular and secure mobile operating systems). The aim is to provide a consistent and rigorous approach in identifying vulnerabilities and design weaknesses in apps that could potentially put sensitive data and PII stored on iOS devices at risk.

1530-1605/15 $31.00 © 2015 IEEE
DOI 10.1109/HICSS.2015.611

Contribution 2: Using our proposed iOS app analysis process (iOSAAP – see Figure 1), we study a widely used Australian Government healthcare app and identify a previously unknown / unpublished vulnerability, which would expose a user's sensitive data and PII such as claim history and electronic health transactions stored on the iOS device. The vulnerabilities were reported to the Australian Government Department of Human Services on 13 May 2014. The reported vulnerabilities were fixed in version 1.0.9 of the app released on 22 May 2014.

The outline of this paper is as follows. The next two sections outline our research motivations and the proposed iOSAAP. In Section 4, we demonstrate how iOSAAP can be applied and our findings on the case study app are revealed. Section 5 concludes our paper.

2. Research motivations

In recent years, there has been increased interest in mobile apps, including healthcare apps and their benefits in mobile healthcare. For example, some healthcare apps allow users and their healthcare providers to access health record systems to view or download the user's personal / electronic health record data, and some apps use a mobile camera to document or transmit pictures to communicate potential medical conditions.

A number of studies have, however, raised concerns about the potential security and privacy risks associated with healthcare apps due to inadequate security policies, lack of experience in mobile app security design and the absence of self-protection (see [23]). For example, in the USN Remote Patient Monitoring System (u-RPMS) [16] designed to provide integrated medical and patient monitoring remote services, a test app was implemented in order to demonstrate its potential for commercialisation. However, security of the user's sensitive data and PII (e.g. biometric information and patients' location) were neither discussed nor considered.

An analysis of over 800,000 apps for iOS and Android platforms also highlighted the various security and privacy implications in poorly designed healthcare and medical apps, as well as the need to regulate the development of medical apps [6].

User's sensitive data and PII may not be stored securely on the mobile devices, as demonstrated by Mitchel et al. [18]. The latter study revealed that a significant number of mobile health apps store unencrypted personal information on mobile devices. In another independent study, Craig [11] also pointed out that user data stored on mobile devices or transmitted on communication networks may be collected by a third party.

Although there has been academic interest in mobile apps as well as mobile devices (see [12][17]), previous studies tend to adopt a break-and-fix approach. To the best of our knowledge, there is no one user-friendly process that will assist app developers, store operators and users to examine apps for vulnerabilities and design weaknesses, particularly in apps designed for iOS devices. This is the focus of the paper.

3. Our proposed iOS app analysis process

The iOSAAP comprises four main stages, namely

1. *Analysis environment set-up,*
2. *App decryption,*
3. *App disassembly,* and
4. *App debugging*

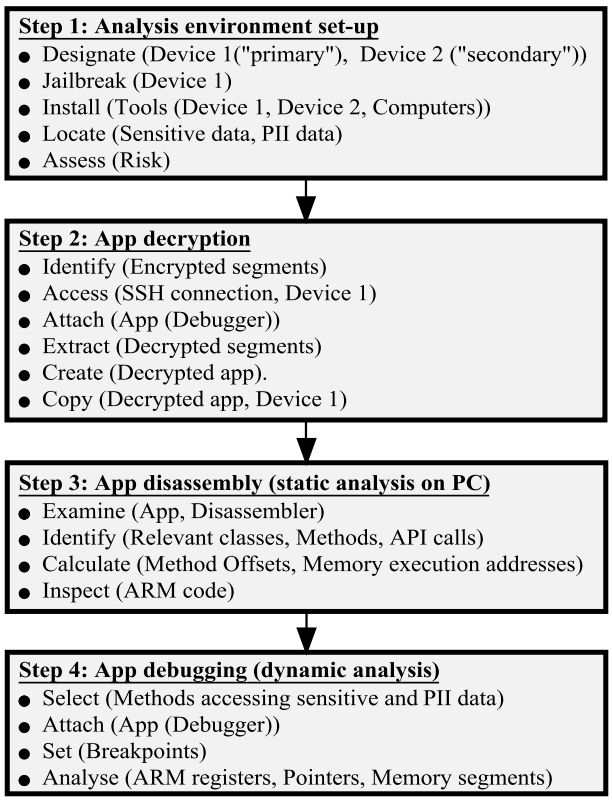

Figure 1. iOS app analysis process (iOSAAP)

Although our proposed process is designed mainly for app developers and app store operators who have to either evaluate risks associated with third-party applications or discover their vulnerabilities, it could be of potential interest to the iOS app development community if this is integrated into the final stages of

the Software Development Life Cycle to detect security vulnerabilities before any app is published.

3.1. Analysis environment set-up

Apple has significantly improved the security architecture of iOS devices in recent years, and, consequently, the number of vulnerabilities that can be exploited is reduced. However, this complicates and hinders law enforcement digital forensic investigations as existing digital forensic tools do not have root access on newer iOS devices.

In order to conduct a detailed examination of an iOS app, we need to install the app on a device that has been jailbroken[1]. Otherwise, we would not be able to install the tools (e.g. *OpenSSH* and *gdb*) to examine the app. This is a widely accepted approach in analysing iOS devices (including in digital forensics investigations). To jailbreak the device, we use *evasi0n (www.evasi0n.com)* which is the only jailbreak program supporting both A4 and A5 chip-based devices running the newer iOS versions (i.e. iOS 7.0.x) at the time when this research was conducted.

The aim of this stage is to prepare the necessary equipment. We recommend having one iOS device designated as the "primary" device (the attacker's device) and the victim's device(s) designated as the "secondary" device(s). The app is then installed on both the "primary" and "secondary" devices.

In this stage, we will also identify the type of sensitive data and PII by running the app on the secondary devices (for the evaluation of data at rest) and the primary device (for the evaluation of data in transit) and using the research tools (see Table 2) to determine the data that could potentially be at risk.

3.2. App decryption

If an existing app installed directly from the Apple's App Store is the subject of analysis, we would have to first decrypt the executable file prior to conducting static and dynamic code analysis (also known as App disassembly and App debugging – see Sections 3.3 and 3.4 respectively). This would allow us to obtain an executable file that contains no encrypted segment for detailed analysis.

The app decryption process can be either automatic or manual. For the former, a decryption tool such as Clutch or the iNalyzer framework can be used to generate a package containing a decrypted version of the app for installation on jailbroken devices. Although the automatic process is faster and more flexible, we will demonstrate how to perform a manual decryption of iOS apps in Section 4.2 for completeness. A manual app decryption process can also facilitate one's understanding of iOS apps and its intricate workings.

3.3. App disassembly (Static analysis)

The decrypted app is disassembled and analysed using tools, such as *Hopper Disassembler*, capable of processing the binary code of the executable file to generate ARM assembly code (an assembly language) and pseudo code (a human-readable form of code, which closely resembles the code using the programming language Objective C).

The static analysis of both assembly and pseudo codes is of vital importance in determining the relevant app components (e.g. methods and procedures) used to process sensitive data and PII and to identify the memory addresses where these methods are executed. Information obtained is then used as input for the dynamic analysis.

3.4. App debugging (Dynamic analysis)

The app is dynamically analysed while it is executed on the 'primary' device using a debugging tool. The memory addresses identified during the previous stage will be set as the breakpoints (i.e. locations where the app is programmed to pause). Next, a step-by-step execution of ARM instructions is performed in order to inspect the CPU registers, memory content and values returned from the methods and procedures (identified in the static analysis stage). Since the app is analysed at runtime on the 'primary' device, we will be able to analyse sensitive data and PII, both at rest and in transit.

Based on the findings from this stage, the app examiner would be able to determine whether there are design and/or implementation weaknesses that need further investigation (e.g. weakness that could potentially lead to a compromise of the user's data).

4. A case study app

In this paper, we present our findings on the analysis of *Medicare Express Plus (MEP)*. The app allows users to use their iOS and Android devices to view their medical history, access health-related documents saved in the vault (e.g. medical transactions and lodged claims), update their personal information (e.g. residential address, telephone numbers and e-mail, contact numbers, e-mail and bank account information), find the nearest service centres, etc. In

[1] Jailbreaking is the process of gaining root access on iOS devices by removing limitations on the operating system. It is a form of privilege escalation necessary for the installation of analysis and monitoring tools.

order to access their accounts, users must first register with the agency to receive their username. The username is an 8-character field, in the following format: Two capital letters followed by six digits (in this paper, one of our usernames was used, and denoted as 'ZZ999999'). Users will then choose their password, a 4-digit PIN number and three (pre-defined or user-generated) security questions. The 4-digit PIN number is designed to replace the need for user to enter his/her username and password when using the app on the mobile device.

We now demonstrate how iOSAAP is used to analyse the case study app.

4.1. Analysis environment set-up

We installed the two most recent versions of the case study app (i.e. version 1.0.8 last updated on 5 May 2014, and version 1.0.7 last updated on 18 Nov 2013) prior to the fix on seven different devices running different iOS versions – see Table 1. The vulnerabilities described in this paper have been fixed in version 1.0.9 of the app released on 22 May 2014 due to our responsible disclosure.

Table 1. iOS devices used in our research

iOS device	iOS version
iPhone 3G (8 GB)	6.1.2
** iPhone 4 (16 GB)	6.1.2
iPhone 4S (16 GB)	6.1.3
iPhone 4S (32 GB)	7.1
iPhone 5S (16 GB)	7.0.5
** iPhone 5S (16 GB)	7.0.5
iPhone 5S (16 GB)	7.1

** Devices selected as the "primary" devices.

We selected the iPhone 4 and one of the iPhone 5S as the "primary" devices to verify that our process is backward compatible, which is confirmed by our findings in this section.

We proceeded with the jailbreaking of both "primary" devices and the installation of the research tools outlined in Table 2. We connected the remaining five devices to a personal computer (PC) running Windows 7 (see Table 2) and searched for sensitive data and PII in the application folder using *DiskAid*. We found documents in PDF format saved in the app vault as well as an associated database named VaultDataModel.sqlite.

As this database was unencrypted, we examined it using SQLittle DB Browser and found records that linked to the stored documents containing sensitive data and PII such as the user (patient)'s name, date of birth, healthcare card number, types of healthcare treatment received, date and cost of the healthcare services, and information about the healthcare provider. Surprisingly, these files could be trivially copied from devices running iOS versions prior to 7 (e.g. iOS version 6.1.6), even when the devices were locked with a passcode or password (unknown to us / the attacker). Although the documents were encrypted, once copied to another external device, the attacker could conduct an offline brute force attack as we demonstrate in Section 4.5.

Table 2. Tools used in our research

Device	Tools installed
iPhone 4 ("primary" device 1)	OpenSSH, gdb, ldid, tcpdump, keychain_dumper
iPhone 5S ("primary" device 2)	OpenSSH, gdb, ldid, tcpdump, keychain_dumper
Macbook Pro	MachOView
PC (running Windows 7)	Hopper Disassembler, SQLite DB Browser, Hexadecimal Editor, MobaXterm, iPhone Backup Extractor, DiskAid, Wireshark

We were also able to locate documents in non-password protected iTunes backups by using *iPhone Backup Extractor*, regardless of the device type/model and iOS version used. In addition, it was discovered that documents saved on a device running iOS version 7 and above could also be copied when the device was passcode/password-locked if it is connected to a trusted computer. In an office environment where iPhones or iPads are left on someone's desk beside their work computer / laptop or when co-workers share a computer using different login credentials, colleague(s) could easily connect the iOS devices to the trusted computer and execute this process.

To perform the evaluation of data in transit, we connected the "primary" device to a wireless network and establish an SSH connection with the PC via the *MobaXterm's* console using the 'ssh' command – "*ssh –p 22 root@10.230.59.21*" (note: 10.230.59.21 is the IP address assigned to the "primary" device). MEP was then run directly from the springboard of the "primary" device.

We examined whether secure connection with the app server was established when a request for the user medical history was submitted. For this purpose, all TCP packets were captured using the *tcpdump* command-line via the SSH connection. An analysis of the TCP packets using *Wireshark* revealed that all communications were encrypted using Hypertext Transfer Protocol Secure (HTTPS). We also

determined that the app uses iOS native APIs (rather than OpenSSL) and, therefore, not vulnerable to the recently discovered Heartbleed bug [19].

4.2. App decryption

The app was examined on the Macbook Pro to determine whether it was encrypted, and if so, search for relevant information to facilitate decryption. MEP was analysed using *MachOView* to search for the relevant values in the LC_ENCRYPTION_INFO section of the binary file. This allowed us to identify the following information:

```
Crypt Offset    4096   (0x1000)
Crypt Size      880640 (0xD7000)
Crypt ID        1      (1: encrypted;
                        0: unencrypted)
Crypt ID Offset 6776   (0x1A78)
```

These values indicated the following offset values, namely: where the encryption starts (0x1000), the size of the encrypted segment (0xD7000), whether the app is encrypted (1), and the offset where the Crypt ID is located (0x1A78). Offset values were calculated from the beginning of the file.

Using the same SSH connection established during the environment set-up, we then attached the app to the debugging tool (*gdb*) via the *MobaXterm*'s console:

```
root# gdb
(gdb) attach MEP
```

The command *gdb* starts the debugger while *attach* will attach the MEP process to the debugger. In order to obtain a decrypted dump of the app, it is necessary to calculate the range of memory addresses where the process is being executed. This occurs because of the *Address Space Layout Randomisation (ASLR)*, a technique implemented by Apple to ensure that all memory regions are randomised upon launch to prevent exploit attacks. The regions are easily calculated as follows:

```
(gdb) info mach-region 0x0
Region from 0x4000 to 0xDC000
```

The output values of the gdb's *info* command (0x4000 and 0xDC000) provide MEP's execution address space. These values and the Crypt Offset are necessary for dumping the decrypted segment of code from memory.

```
(gdb) dump binary memory
decrypted_code.bin 0x5000 0xDC000
```

The gdb's *dump* command created a file containing the decrypted code (decrypted_code.bin) on the "primary device". The initial dump start address (0x5000) was calculated as the app memory region start address (0x4000) plus the encryption offset (Crypt Offset = 0x1000). This file (decrypted_code.bin) was then copied to the PC where, using a hexadecimal editor, we modified the raw data of the original executable file to generate a decrypted version of MEP. This task includes overwriting the encrypted code (via copy and paste) and setting the Crypt ID (at offset 0x1A78) to '0' in order to indicate that the code was no longer encrypted.

We copied the decrypted version of MEP via the SSH connection to the app folder on the "primary device" replacing the original executable file (keeping a copy of this file is always recommended). Since executable permissions are not set by default when an executable file is copied from an external device, we changed the permissions of the decrypted app via the *MobaXterm's* console using the *chmod* command – "*chmod 755 ./MEP*".

Finally, the *ldid* command allows us to manipulate the signature block within the Mach-O binary (format for native executable files on iOS devices). The command "*ldid –s ./MEP*" was used to update the hashes of the decrypted app.

4.3. App disassembly (Static analysis)

As a result of the previous stage, a fully functional and decrypted version of the app was available for an in-depth inspection. On the PC, we used a trial version of *Hopper Disassembler* to conduct a static examination of the code (without executing MEP) and to identify the relevant classes, methods and API calls implemented within the app for the protection and encryption of the documents saved in the vault.

By using search patterns such as "PIN", "key" and "AES" we obtained the following list of methods.

```
1.  PinLockController_saveKeychainEntries
2.  KeychainItemWrapper_securedSHA256DigestHashforPIN
3.  Encryption_AESdecryptData_usingPassword
4.  Encryption_AESKeyForPassword_salt
```

Table 3. Method offsets and execution addresses

Method	File offset	Memory address
1	0x00010714	0x00014714
2	0x0006574c	0x0006974c
3	0x0007733c	0x0007b33c
4	0x00077174	0x0007b174

In addition, it was possible to identify the physical offsets of the entry points of these methods in the file. By adding the start address of the mach-region (0x4000) to the offsets, we were able to determine the exact memory addresses where the execution of these

methods started. The memory addresses (see Table 3) are set as breakpoints at the debugging stage.

We then analysed each of the four methods individually to obtain more clues on how documents are protected – see Sections 4.3.1 to 4.3.4.

4.3.1. Method 1: saveKeychainEntries

As shown in Figure 2, it appears that Method 1 was used to calculate the SHA256 digest of a hash value of the username in order to save the result in the keychain (a SQLite database stored on the iOS file system whose items are inaccessible until the user unlocks the device by entering their passcode [5]). Although at this stage it was still premature to know in details the functionality of this method, the static analysis provided useful leads.

```
movw  r1, #0xf13e
movt  r1, #0xe
add   r1, pc                          ; 0x1008cc
ldr   r1, [r1]                        ; @selector(userCRN)
blx   imp___symbolstub1__objc_msgSend
movw  r1, #0xf132
movt  r1, #0xe
add   r1, pc                          ; 0x1008d0
ldr   r1, [r1]                        ; @selector(hash)
blx   imp___symbolstub1__objc_msgSend
mov   r2, r0
movw  r0, #0xf11c
movt  r0, #0xe
movw  r3, #0x1c7e
movt  r3, #0xf
add   r0, pc                          ; 0x1008d4
add   r3, pc                          ; 0x103438
ldr   r1, [r0] ; @selector(securedSHA256DigestHashForPIN:)
```

Figure 2. Method 1: ARM code

4.3.2. Method 2: securedSHA256DigestHashForPIN

The most relevant part of this method, called from the previous method, is depicted in Figure 3.

```
movw        r2, #0x3552
ldr         r1, [r1]   ; @selector(stringWithFormat:)
movt        r2, #0xa
ldr         r0, [r0]   ; @bind__OBJC_CLASS_$_NSString
movw        r9, #0x355c
add         r2, pc     ; 0x109cd0
movt        r9, #0xa
add         r9, pc     ; 0x109ce0 (FvTivqTqZXsgLLx1v3P8....)
str.w       r9, [sp]
blx         imp___symbolstub1__objc_msgSend
mov         r2, r0
movw        r0, #0xb954
movt        r0, #0x9
add         r0, pc     ; 0x1020ec
ldr         r1, [r0]   ; @selector(computeSHA256DigestForString:)
mov         r0, r4
```

Figure 3. Method 2: ARM code

It can be seen from Figure 3 that a value (most probably, the hash of the username passed as a parameter from Method 1) and a string located at addresses 0x109ce0 are formatted as a whole NSString by performing a system call to *stringWithFormat*.

Using the *Hopper disassembler*, we were able to identify the full string as *'FvTivqTqZXsgLLx1v3P8TGRyVHaSOB1pvfm02wvGadj7RLHV8GrfxaZ84oGA8RsKdNRpxdAojXYg9iAj'*.

We were surprised (perhaps, shocked) that when we did a search for this string on the Internet, we discovered that the ARM code in Figure 3 shows significant similarities to the Objective C code on http://www.raywenderlich.com/6475/basic-security-in-ios-5-tutorial-part-1 (last accessed 13 May 2014) – see the screen capture in Figure 4.

```
                        First piece of code
[START]
...
// !!KEEP IT A SECRET!!
#define SALT_HASH @"FvTivqTqZXsgLLx1v3P8TGRyVHaSOB1pvfm02wvG
                   adj7RLHV8GrfxaZ84oGA8RsKdNRpxdAojXYg9iAj"
...
[END]

                        Second piece of code
[START]
...
+ (NSString *)securedSHA256DigestHashForPIN:(NSUInteger)pinHash
{
    // 1
    NSString *name = [[NSUserDefaults standardUserDefaults]
                       stringForKey:USERNAME];
    name = [name
            stringByAddingPercentEscapesUsingEncoding:
            NSUTF8StringEncoding];
    // 2
    NSString *computedHashString = [NSString
             stringWithFormat:@"%@%i%@", name, pinHash,
             SALT_HASH];
    // 3
    NSString *finalHash = [self
             computeSHA256DigestForString:computedHashString];
    NSLog(@"** Computed hash: %@ for SHA256 Digest: %@",
          computedHashString, finalHash);
    return finalHash;
}
...
[END]
```

Figure 4. Method 2: Objective C code

This suggested that the code from the Internet site may have been reused. Upon closer inspection, we determined that the random string from the Internet site (Figure 4) is also used as a salt value (random data used as an additional input to a one-way function that hashes a password) within MEP.

4.3.3. Method 3: AESdecryptData_usingPassword

In this method, we used the pseudo code visualisation (see Figure 5) feature provided by *Hopper Disassembler*.

```
var_8 = var_28;
var_12 = r6;
var_16 = r5;
var_20 = r0;
var_24 = &var_40;
r0 = CCCrypt(0x1, 0x0, 0x1, var_36);
```

Figure 5. Method 3: pseudo code

According to the iOS reference library [2] and CommonCryptor definition [1], the implementation of the *CCCrypt* API relates to the use of AES-128 for data decryption, and the four parameters are displayed in Figure 6.

```
0x1: the type of operation (CCOperation = kCCDecrypt);

0x0: the algorithm (CCAlgorithm = kCCalgorithmAES128);

0x1: block cipher options (CCOptions = kCCOptionPKCS7Padding);

var_36: a pointer to a data structure containing additional
information (*key, keyLength, *iv, *dataIn, dataInLength,
             *dataOut, dataOutAvailable, *dataOutMoved).
```

Figure 6. CCCrypt parameter specifications

The value returned from CCCrypt is stored in the 'r0' ARM register and indicates whether the operation is successful.

4.3.4. Method 4: AESKeyForPassword_salt

This method is called from Method 3 and its name suggests the generation of an AES key for the password salt (see reference to 0x109ce0 in Figure 7) which is passed as a parameter.

```
movt        r1, #0x8
movw        r6, #0x1982
add         r1, pc    ; 0x102610
movt        r6, #0x9
add         r6, pc    ; 0x109ce0 (FvTivqTqZXsgLLx1v3P8....)
mov         r2, r3
ldr         r1, [r1]  ; @selector(AESKeyForPassword:salt:)
```

Figure 7. Method 4: ARM code

```
movw        r1, #0x2710     <---
movs        r2, #0x1
str         r4, [sp]
mov         r3, r8
str         r2, [sp, #0x4]
mov         r2, r10
str         r1, [sp, #0x8]  <---
ldr         r1, [sp, #0x14]
str         r6, [sp, #0xc]
str         r0, [sp, #0x10]
movs        r0, #0x2
bl          sub_65df0       <---
mov         r0, r11
```

Figure 8. Method 4: Call to sub_65df0

The static analysis used in this method was not sufficient in determining how the AES key was calculated. However, as described in Figure 8, it was possible to discover that the 'sub_65df0' sub-function was called and a value equal to 0x2710 (10,000 in decimal notation) was stored in the stack (sp). This is a typical combination of instructions to save values in the stack for future use rather than passing them as parameters.

Our analysis of the sub_65df0 code suggested the use of the exclusive OR logical operator (usually denoted as XOR, EOR or $\oplus$) and CCHmac – see Figures 9 and 10. XOR is commonly used in cryptographic data encryption, and CCHmac is Apple's implementation of HMAC [3], an algorithm for the authentication of messages given a cryptographic hash function and a secret key. As the code in Figure 10 needs further examination to determine the hash function used and the secret key, Method 4 was selected for the dynamic analysis in the next step (see Section 4.4).

```
ldrb        r2, [r4], #0x1
subs        r1, #0x1
ldrb        r3, [r0]
eor.w       r2, r2, r3  <---
strb        r2, [r0], #0x1
bne         0x65f1e
```

Figure 9. Method 4: Use of exclusive OR (EOR)

```
movs        r0, #0x3
b           0x65fe4
movs        r0, #0x4
strd        r4, r5, [sp]
blx         imp___symbolstub1__CCHmac
```

Figure 10. Method 4: Call to CCHmac

4.4. App debugging (Dynamic analysis)

To perform a run-time evaluation of the app, we first need to establish an SSH connection with the 'primary' device using *MobaXterm* (as per described in Section 4.1) and to attach the MEP to the debugger (as per described in Section 4.2). We then proceeded with the security assessment of data at rest by debugging relevant methods identified during the static analysis. More specifically, the static analysis in Sections 4.3.2 and 4.3.3 provided in-depth explanations regarding Methods 2 and 3. Therefore, in the remainder of this subsection, we only demonstrate the dynamic analysis of Methods 1 and 4 (to avoid duplication).

For the dynamic analysis, it was necessary to set memory breakpoints as indicated in Figure 11.

```
(gdb) break *0x14714
Breakpoint 1 at 0x14714
(gdb) break *0x7b174
Breakpoint 2 at 0x7b174
(gdb) continue
Continuing.
Reading symbols for shared libraries .. done
```

Figure 11. Breakpoints

From the *MobaXterm*'s console, we set breakpoints at two of the memory addresses calculated in Section 4.3, respectively for Methods 1 and 4, followed by the *'continue'* command to allow the app continue the execution.

4.4.1. Method 1: saveKeychainEntries

The essential information required in Method 1 but still unknown at this stage was the hash algorithm applied to the username (see Figure 2). We reset the PIN number to save a new entry in the keychain so that the app halts on the first breakpoint when reaching the call to *hash* (referred as @selector(hash) in Figure 2). Once the app paused, we inspected the memory address pointed by the 'r0' ARM register (0x1f3b8f20) and obtained our stored username ('ZZ999999').

We also determined that the hash algorithm used was provided by Apple for the CFString class [4] (and not part of MEP). According to the CFString Class Reference, the hash algorithm will return an unsigned integer. Its source code is available on http://opensource.apple.com/source/CF/CF-476.17/CFString.c. For the user 'ZZ999999', the returned hash value was 2200911122.

The securedSHA256DigestHashForPIN method was used to calculate the SHA256 hash of 2200911122 concatenated with the salt value.

Finally, we inspected the keychain content using the *keychain_dumper* tool.

The results illustrated in Figure 12 confirmed that the value of 'Keychain Data' effectively corresponds to the SHA256 hash (see Figure 13).

```
Generic Password
----------------
Service: DHSHealthApp
Account: encryptionPassword
Entitlement Group: LZBL5M4HIS.DHSMyGovHealthApp
Label:
Generic Field: encryptionPassword
Keychain Data: 66e77d04e81c42e5a1fa69bc4de9e4bb
               362ba53e1e3dbb8f24b1a14d2757102c
```

Figure 12. Keychain content for MEP

```
SHA256(hash(username) + salt) = 66e77d04e81c42e5a1fa69bc4de9e4bb
                                362ba53e1e3dbb8f24b1a14d2757102c
```

Figure 13. SHA256 Digest Hash

We have now determined that the SHA256 hash is not a value for the PIN number, and securedSHA256DigestHashForPIN is simply a result of reusing code from the Internet. This hash value will not change for the same user regardless of the value of the PIN and the number of times the PIN is reset.

4.4.2. Method 4: AESKeyForPassword_salt

This method is used to save encrypted documents and display decrypted documents in the vault.

We accessed the vault and selected one of the stored documents that halt at the 0x7b174 breakpoint. When a step-by-step execution reached the call to CCHmac, we inspected the content of the 'r0', 'r1', 'r2' and 'r3' ARM registers which, passed as parameters, contained the following values:

```
r0 0x0
r1 0x1c5801e0
r2 0x40
r3 0x2fdd9c28
```

According to Apple's CCHmac specification [3] and the standard ARM calling convention [7], a value of '0' as the first parameter provides access to kCCHmacAlgSHA1 (SHA1 algorithm). The values in 'r1'and 'r3' were pointers to the key and the data to be hashed respectively, and the value in 'r2' was the key length. As shown in Figure 14, the key (pointed by 'r1') was actually the value calculated in Section 4.4.1 (referred as the *encryptionPassword* field in Figure 12) and the data (pointed by 'r3') was the salt.

```
(gdb) x/16 $r1
0x1c5801e0: 36366537 37643034 65383163 34326535 66e77d04e81c42e5
0x1c5801f0: 61316661 36396263 34646539 65346262 a1fa69bc4de9e4bb
0x1c580200: 33363262 61353365 31653364 62623866 362ba53e1e3dbb8f
0x1c580210: 32346231 61313464 32373537 31303263 24b1a14d2757102c
(gdb) x/20 $r3
0x2fdd9c28: 46765469 76715471 5A587367 4C4C7831 FvTivqTqZXsgLLx1
0x2fdd9c38: 76335038 54475279 56486153 4F423170 v3P8TGRyVHaSOB1p
0x2fdd9c48: 76666D30 32777647 61646A37 524C4856 vfm02wvGadj7RLHV
0x2fdd9c58: 38477266 78615A38 346F4741 3852734B 8GrfxaZ84oGA8RsK
0x2fdd9c68: 644E5270 7864416F 6A585967 3969416A dNRpxdAojXYg9iAj
```

Figure 14. Memory dump when calling CCHmac

Finally, we observed that the HMAC algorithm was computed 10,000 times (based on the value shown in Figure 8). The result from each iteration was XORed (see EOF reference in Figure 9) with the value obtained from the former iteration and then used as input data for the next iteration. The most significant 16 bytes resulting from the whole iterative process represent the AES key used to decrypt the PDF file.

At the end of this stage, we were able to determine the complete algorithm that generates the AES key for a particular username as illustrated in Figure 15. The resulting AES key for 'ZZ99999' was '77455F0A124F738FB77977C3EC0930D9', which is the value used for CCCrypt (see Figure 5).

```
;CALL (P1, P2,..., PN): API/Procedure invocation
;PX:Parameters
;⟵: Assign returned value
;SET: Set value
;[X0..XN]: Array of N+1 elements
;⊕: Exclusive OR

;Calculation of the encryption password (ep)
SET userName = 'ZZ999999'
SET salt = 'FvTivqTqZX...'
ah ⟵ CALL AppleHash(userName)
constant ⟵ CALL Concat(ah, salt)
ep ⟵ CALL SHA256(constant)
;Calculation of the AESkey
SET hKey = ep
SET hData = salt
hMac1 ⟵ CALL HMacSHA1(hKey, hData)
SET shaXOR = hMac1
SET i = 1
WHILE i < 10000
   hMac2 ⟵ CALL HMacSHA1(hKey, hMac1)
   SET shaXOR = shaXOR ⊕ hMac2
   SET hMac1 = hMac2
   SET i = i + 1
ENDWHILE
SET AESkey = shaXOR[0..15]
```

Figure 15. Decryption algorithm

4.5. Findings and recommendations

Since the encryption password (ep) is stored in both the keychain and unencrypted database of the app (referred as VaultDataModel.sqlite in Section 4.1), an attacker can skip the calculation of *ep* in Figure 15 and trivially compute the AES key without brute-forcing the username.

In the event that the attacker is unaware of the above possibility, an offline brute-force attack can be conducted due to the design weakness of MEP. Although the total number of key combinations for AES-128 is 3.40 x 10^{38}, deriving the PDF encryption key from the username (an 8-character field: two capital letters followed by six digits) reduces this combination to only 676,000,000 (all possible valid usernames from 'AA000000' to 'ZZ999999').

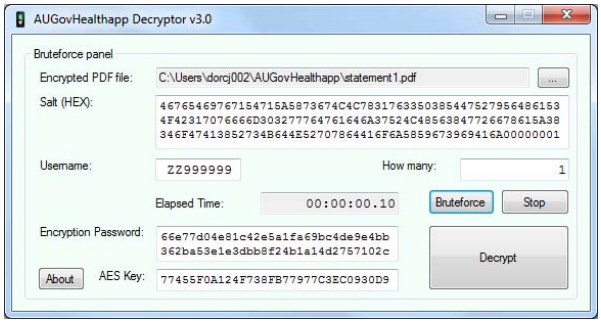

Figure 16. MEP document decryptor

As a proof of concept, we developed an application capable of generating 7 million combinations (in the format: two capital letters followed by six digits) in 24 hours using four instances of the application running simultaneously on an Intel Core i5-4570 CPU (3.20 GHz) PC – see Figure 16 for the screen capture.

The offline brute force attack can also be executed in parallel using cloud computing or more computers. For example, we could determine the username and AES key (total of 676,000,000 combinations) in less than ten days using 10 computers running Intel Core i5-4570 CPU (3.20 GHz).

Recommendation 1: As a result of our findings, we recommend that stored documents and the database (in all iOS apps) should belong to the *NSFileProtectionComplete* Data Protection Class [5] (also known as Class A), which offers a secure passcode-derived encryption. This would prevent the files to be directly acquired from a device that is locked. Note that Class A files could only be acquired from an iTunes backup that is not encrypted, since the device passcode is prompted by iTunes before performing the backup, leaving any Class A content decrypted in the backup folder.

Recommendation 2: To further complicate brute-force attacks, the AES key[2] in the MEP app should be derived from a username containing not only digits but also a combination of lower- and upper-case characters. Additionally, the 4-digit PIN number should be used to generate the AES initialisation vector (IV) (a starting variable that adds pseudo-randomisation) currently defined as null (an existing weakness).

Recommendation 3: There are obvious risks in reusing the salt value and source code available on the Internet (see Section 4.3.2) in any apps, as an attacker can use the known salt value to facilitate their access to the data.

5. Conclusion and future work

In this paper, we demonstrated how our proposed iOSAAP could be used to identify vulnerabilities and design weaknesses in iOS apps (e.g. inappropriate implementation of cryptographic algorithm and the

[2] As pointed out by one of the reviewers, cryptographic key should be derived from password rather than username. However, this practice is not applicable to the MEP app as the password is used only during the first login (i.e. we would not be able to compute the AES key without the password as it is not stored on the device).

storage of sensitive data and PII in an unencrypted database in our case study app).

Our findings highlighted that even though iOS devices are generally secure, the implementation of security mechanisms such as anti-debugging techniques can help to further enhance user data security and privacy.

One potential research focus is to examine healthcare apps and propose secure design principles that would enhance the protection of user data stored on mobile devices. In addition, we will extend iOSAAP to include the examination of encrypted SSL connections with the aim of determining whether an app is vulnerable to a man-in-the-middle attack.

References

[1] Apple. 2006. Apple OpenSource. http://www.opensource.apple.com/source/CommonCrypto/CommonCrypto-36064/CommonCrypto/CommonCryptor.h

[2] Apple. 2007. iOS Developer Library. https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man3/CCCrypt.3cc.html

[3] Apple. 2007. Mac Developer Library. https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/CCHmac.3cc.html

[4] Apple. 2012. Mac Developer Library. DOI= https://developer.apple.com/library/mac/documentation/CoreFoundation/Reference/CFStringRef/Reference/reference.html

[5] Apple. 2014. iOS security (Feb. 2014). http:// http://images.apple.com/ipad/business/docs/iOS_Security_Feb14.pdf

[6] Appthority. 2014. Medical Apps are here to stay: So how do we keep the value and lose the risk? (Mar. 2014). https://www.appthority.com/news/medical-mobile-apps

[7] ARM Infocenter. 2012. Procedure Call Standard for the ARM Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

[8] Berg Insight. 2013. The Mobile Application Market. http://www.berginsight.com/ReportPDF/ProductSheet/bi-app1-ps.pdf

[9] Boyd, C. & Choo, K-K R. 2005. Security of Two-Party Identity-Based Key Agreement. Lecture Notes in Computer Science, 3715/2005, 229–243.

[10] Choo, K-K R. 2009. Secure Key Establishment. Advances in Information Security 41, Springer.

[11] Craig, MLN. 2013. Technical Analysis of the Data Practices and Privacy Risks of 43 Popular Mobile Health and Fitness Applications (Aug. 2013). https://www.privacyrights.org/mobile-medical-apps-privacy-technologist-research-report.pdf

[12] D'Orazio, C., Ariffin, A. & Choo, K-K R. 2014. iOS Anti-forensics: How Can We Securely Conceal, Delete and Insert Data?, In HICSS 2014, pp.4838–4847, 6–9 October.

[13] Eng, DS. & Lee, JM. 2013. The promise and peril of mobile health applications for diabetes and endocrinology. Pediatric Diabetes, 14(4): 231-238.

[14] HP. 2013. Hewlett Packard Press Release. http://www8.hp.com/us/en/hp-news/press-release.html?id=1528865

[15] Kianas, A., Restivo, K. & Shirer, M. 2012. Android and iOS surge to new smartphone OS record in second quarter, according to IDC. Press release 8 August. http://www.idc.com/getdoc.jsp?containerId=prUS23638712

[16] Kim, Y.-H., Lim, I.-K., Lee, J.-P., Lee, J.-G. & Lee, J.-K. 2012. Development of Mobile Hybrid MedIntegraWeb App for Interoperation between u-RPMS and HIS. ICCSA 3, Volume 7335 of Lecture Notes in Computer Science, 248–258.

[17] Miller, C., Blazakis, D., DaiZovi, D., Esser, S., Iozzo, V. & Weinmann, RP. 2012. iOS Hacker's Handbook (1st ed.). Wiley Publishing.

[18] Mitchell, S., Ridley, S., Tharenos, C., Varshney, U., Vetter, R., & Yaylacicegi, U. Investigating Privacy and Security Challenges of mHealth Applications. 19th Americas Conference on Information Systems, AMCIS 2013 - Hyperconnected World: Anything, Anywhere, Anytime. 2013;3:2166-2174.

[19] OpenSSL Security Advisory. 2014. TLS heartbeat read overrun (CVE-2014-0160) (Apr. 2014). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[20] Paschou, M., Sakkopoulos, E. & Tsakalidis, A. 2013. easyHealthApps: e-Health Apps Dynamic Generation for Smartphones & Tablets. Journal of Medical Systems, 37(3), 1-12.

[21] Research2Guidance. 2013. Mobile Health Market. http://www.research2guidance.com/shop/index.php/downloadable/download/sample/sample_id/262/

[22] U.S. Food and Drug Administration. 2014. Examples of MMAs the FDA Regulates. http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/ConnectedHealth/MobileMedicalApplications/ucm368743.htm

[23] Wagner, R., Zumerle, D., Girard, J. & Feiman, J. (2013). Predicts 2014: Mobile Security Won't Just Be About the Device. Gartner, Inc.