iOS App Auditing

Nils Schmidt
Philipps-University Marburg, Germany
Department of Mathematics & Computer Science, Distributed Systems Group
E-Mail: schmidt89 at informatik.uni-marburg.de
2/ 2015

***Abstract*—The mobile device sector has grown over the past years. Most of us have a smartphone or tablet carrying with us every day. But the advanced mobility also requires special security enforcement as the devices store a lot of sensitive data and can be easily stolen or lost.**

Developers for the *iOS* platform have to pay attention to avoid common pitfalls when developing applications. Buffer overflows, format string vulnerabilities, SQL injections, Cross-Site Scripting, data leakage or theft are security issues which can happen if user input has not been sanitized correctly.

This paper is meant to point out common mistakes in the iOS app development by showing Objective-C and Swift source code. Furthermore, it introduces the reader to the principles of *reverse engineering* so that third party applications can be audited for those bug classes.

I. INTRODUCTION

Mobile devices have made the way into our daily lifes. Surfing, chatting, reading emails, playing games, listening to music. We can do all of this with the little devices we carry with us. In the year 2015, there is an app for everything. If we need an application for making video calls, online banking or creating documents, it is already there.

The question which arises is whether we can trust all these apps. Do they protect our data? Or do they even leak them?

There is a long list of security threats like data theft, impersonation, financial damage or even surveillance .

By now, Apples *App Store* offers more than one million apps [1] and even big companies introduce bugs into their applications. For example Skype had a XSS vulnerability which made the upload of the users address book to a malicious website possible [2]. Starbucks stored credentials in plaintext on the device so that an attacker with physical access could retrieve them and log into the Starbucks website [3].

With the help of *reverse engineering* one can analyse applications and detect software vulnerabilities or bugs. *Reverse engineering* works without inspecting the actual source code just by looking at the binary. With a *disassembler*[1], machine language can be translated into *assembler* code. This low-level representation of the application is complex to read but offers the possibilty to perform a full app audit.

The paper is outlined as follows: In chapter II the *iOS* system is briefly explained to the reader. Afterwards the basics of *reverse engineering*, arm assembly language, fairplay encryption and the Mach-O file format, enable the reader to dive into app auditing in chapter III where the focus is on common security pitfalls when developing *iOS* applications. Despite only enumerating them, attack vectors as well as countermeasures are given. Finally chapter IV concludes the paper and all shown bug classes together with the threats they introduce.

II. BASICS

As of this writing, the latest *iOS* version is 8.1. The operating system employs considerably more security features than compared to the desktop. The additional security is necessary because mobile devices carry on the one hand more sensitive data and on the other hand devices can be lost much easier or even get stolen.

For the protection of data, *iOS* encrypts each file which offers a *remote wipe* feature. Files can be further encrypted to safeguard them even if physical access to the device is available. The development and distribution of applications requires a membership in the *iOS developer program*. Each developer has a Apple-issued certificate which is used to sign the application so that it can be uploaded to the *App Store*. Therefore all applications are submitted by an identifiable person or organization [4].

The *App Store* review process checks for obvious bugs, the usage of private APIs and other malicious behaviour [4], [5], [6]. The *mandatory code signing* and review process ensures that only Apple-approved applications can be run on the device.

Furthermore, they are isolated from other apps as well as the Operating System (OS). This is achieved by the *sandboxing* mechanism which is also called *seatbelt* [6]. Each application is bound to a unique home directory[2] at the time it is installed. The *sandbox* prevents the application from accessing files of other applications or making changes to the system [4].

A. Reverse Engineering

1) ARM: *iOS* devices are ARM powered which is a Reduced Instruction Set Computer (RISC). The platform offers simple but powerful instructions with a large number of General Purpose Registers (GPRs). In contrast to *CISC*, operations cannot be performed directly on the memory. Instead, a *load/store architecture* has to be used to transfer data between

---

[1] All disassemblies shown, are done with the Hopper Disassembler. See http://www.hopperapp.com

[2] On a jailbroken device one can inspect the app contents at /private/var-/mobile/Containers/Data/Application/<UUID>/

memory and registers [7].

This paper focuses on the 32 Bit ARM architecture even though new *iOS* devices are capable of running 64 bit code. The 32 bit version has a 16 bit mode called *Thumb* which increases code density. Luckily with *ARMv7* there is a unified set of *mnemonics* so that the disassembly looks the same [8]. It features 16 GPRs from r0 until r15. The first four are used for passing arguments while calling functions. Additional arguments are pushed to the stack. Functions return values in r0. Other important registers are:

- r7: Pointer to the current stack frame.
- r13/sp: Pointer to the top of the stack.
- r14/lr: The link register holds the return address of the calling function.
- r15/pc: Stores the next instruction address.

Jumps are realized with the branch operation (*b*). An additional *l* sets the link register (lr) so that the function can return to its *callee* and *x* makes the exchange between *ARM* and *Thumb* mode possible [9].

2) Objective-C: *iOS* Applications are written in *Objective-C* or the new programming language *Swift*. The focus in this work will be on *Objective-C* instead of *Swift* as it is still very new and not much information about its runtime are public yet.

Objective-C is a strict superset of C, therefore its architecture is C-powered and application developers can use C too. *Objective-C* differs in that methods are not called. Instead a message is sent to the receiving object. This adds dynamic capabilities to the language. The function *objc_msgSend* serves as a dynamic dispatcher routine that walks the class hieararchy until it reaches the class implementing the method.

The first argument when sending a message, is the receiving object. The second is a *selector* which is a string representation of the method [10]. *setObject:forKey:* e.g. is a method which needs two arguments (one for each ":"). The first two arguments for the selector are passed as arguments to the *objc_msgSend* function. All other arguments are pushed to the stack [8].

Figure 1 shows an example of how message sending works in *Objective-C*. It shows a *ARMv7* disassembly of the code in figure 2.

But before we analyse the function in more detail, we have to look at the stack layout at the time before the function has been called which is visualized in figure 3. Calling a function means to put a new stack frame onto the stack. First of all, lr and frame pointer (r7) are pushed onto the stack. This is done in the so called *function prologue* to enable the *callee* to return to its *caller*. Moreover, r7 is set to the stack pointer (sp) and space on the new stack frame is allocated to hold local variables. The *assembler* code subtracts 4 from the sp due to the fact that the stack grows towards lower addresses and is 4 byte aligned [8].

At the beginning of each method, *r0* holds the self reference to the current class and *r1* the *selector*. *r2* and *r3* contain the first two function arguments. All others are mostly referenced through *r7* (the base pointer) to access the stack of the caller. Therefore values 1 and 2 are passed via the registers *r2* and *r3* and the value 3 is passed via the stack to the function *[self objCFunction:arg2:arg3]*. *blx* is the *mnemonic* responsible for performing the branch.

```
-[AppDelegate callWrapper]:

// function prologue
push {r7, lr} // save old base pointer, return addr
mov  r7, sp   // frame pointer = stack pointer
sub  sp, #0x4 // allocate additional space on stack

// body
movw r1, #0x3402 // higher r1 = 0x3402
movs r2, #0x3 // r2 = 3
movt r1, #0x0 // lower r1 = 0x0
str  r2, [sp, #0x4] // store 3rd argument on stack
add  r1, pc  // r1 = addr of selector(objCFunction:
    arg2:arg3:)
movs r2, #0x1 // r2 = 1
movs r3, #0x2 // r3 = 2
ldr  r1, [r1] // r1 = selector(objCFunction:arg2:
    arg3:)
// [self(r0) objCFunction:1(r2) arg2:2(r3) arg3:3(sp
    +4)]
blx  imp___symbolstub1__objc_msgSend

// function epilogue
add  sp, #0x4 // deallocate stack space
pop  {r7, pc} // restore old frame pointer & retaddr
```

Figure 1. Objective-C Message Sending (Hopper disassembly)

```
- (void)callWrapper{
  [self objCFunction:1 arg2:2 arg3:3];
}
```

Figure 2. Objective-C Message Sending Example

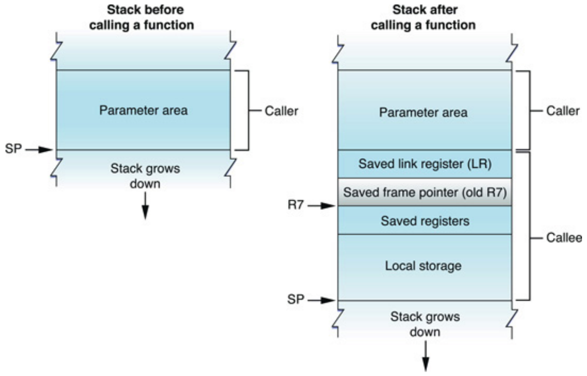

Figure 3. iOS stack layout [8]

In the same way the stack space has been set up, it gets destroyed by adding 4 bytes to the sp. Restoring the old frame pointer and popping the saved lr to the program counter (pc) moves the control flow back to the *callee*.

3) ABI: All applications are stored as *Mach-O* which is the default file format for *iOS* and OS X. It can contain multiple architectures so that a universal application can ship 32 and 64 bit code in one binary (*FAT binary*). *Mach-O* consists of a header that identifies the file format, load commands that set up the internal layout and segments.

Each segment contains code or data for a particular type and is further separated into sections. The most important ones are: *__TEXT*, *__DATA* and *__OBJC*. The first segment holds the executable code (*__text* section) and read only data. The second segment contains writable data. *__nl_symbol_ptr* has to be mentioned here as it contains non-lazy symbol pointers which are indirect referenes from an import statement [8] and needed for the app audit.

The last segment contains metadata about the *Objective-C* program like the list of implemented classes in the binary (*__objc_classlist*), references to classes from imports (*__objc_classref*) as well as categories, selectors and protocols. The complete layout of the *Objective-C* part of the app can be reverse engineered[3] through this segment.

4) FairPlay: Applications are encrypted by default (decryption key stored in keychain [10]) thus preventing *reverse engineering*. However as they have to be decrypted during runtime, this can be exploited to strip the entire encryption. On a jailbroken device, this can be easily achieved with *dumpdecrypted* [10]. Alternative one can use *gdb* to dump the decrypted memory contents.

III. AUDITING iOS APPLICATIONS

In this section we focus on auditing *iOS* apps in terms of security and common pitfalls during the development that can lead to leakage of sensitive data or implement severe security breaches.

A. Memory corruption

A major source of vulnerabilities in the C programming language are *buffer overflows* and *format string* vulnerabilities. *Objective-C* performs bounds checking but overflows are still possible because it is a superset of C, hence it inherents all security issues coming from C.

1) Format Strings: A format function like *printf* is used to format datatypes according to a *format string* and outputs a string.

The *format string* defines the behavior of the format function. For each format specifier prefixed with "%", an argument is taken from either a register or the stack (depending on the calling conventions) and formatted.

Figure 4 depicts the *printf* format function. Its vulnerable because the format string has not been supplied. Attacks are possible if the *format string* gets processed by an untrusted source. For example via user input, the network or the file system [11].

```
char input[200];
// populate input
printf(input);
```

Figure 4. Vulnerable format string

If the attacker can control the *format string*, he can control the behavior of the format function as well.

By supplying a specially crafted *format string*, an attacker can crash the program, inspect the memory or even dump it completely or write data to an arbitrary address in memory.

The format function holds a stack pointer internally. After the first four values have been retrieved from registers, the remaining ones are taken from the stack. Therefore, the attacker can increase the sp and navigate through the stack.

He is able to read and write to arbitrary memory. To achieve this, he needs to encode the memory address in the *format string*. Since the *format string* mostly is located on the stack, he can navigate to it by supplying enough format parameters such as "%.f" or "%x" until the sp points to the *format string*. This enables the attacker to supply his own arguments to the format function. *Format string* vulnerabilities can be detected very easily e.g. by *Xcode* and prevented by always supplying a *format string*. Moreover, code should be written in *Objective-C* where possible due to the fact that it doesn't support the "%n" format specifier. [12] But even *Objective-C* is not entirely safe, as with "%@" a pointer may be called under certain circumstances [13]. Despite printf there are a lot of format functions available which have to be checked for missing *format strings*:

- Classes: NS(Mutable)String, NSAlert, NSPredicate, NSException, NSRunAlertPanel
- Selectors: *format:*, *stringWithFormat:*, *initWithFormat:*, *appendFormat:*
- C functions: (f|sp|sn|as|d|v|vf|vs|vsn|va|vd)printf, syslog.

2) Buffer Overflows: A *buffer overflow* occurs if a program writes beyond a buffer. This can happen if no proper length checking of user input has been done. Like *format string* vulnerabilities, they can be used to overwrite memory and hijack the control flow. Compared to *format string vulnerabilities* one cannot write to arbitrary memory, but its easier to overwrite it. Vulnerable functions and their safe equivalents [11] are:

- strcat, strncat → strlcat
- strcpy, strncpy → strlcpy
- sprintf → snprintf, asprintf
- vsprintf → vsnprintf, vasprintf
- gets → fgets

Apple also marks the "n" functions like *strncpy* as unsafe, as only e.g. *strlcpy* truncates the string at the second last character and adds a null terminated character for the case that the string which shall be copied is larger than the destination buffer. Another countermeasure is to use *Objective-C* as it performs boundary checking although libraries may still contain C code [11]. Unsecure functions can be easily detected by looking at the symbol table with the *nm* command.

3) Dangling Pointers: Until yet, we only dealed with memory corruption on the stack. But for dynamically allocated memory the heap plays an important role. Bugs arise if allocated memory (with free(C) or alloc(Objective-C)) is used or deallocated again after it already has been deallocated.

If an attacker can overwrie the previously deallocated memory

---

[3] See https://code.google.com/p/networkpx/wiki/class_dump_z

and the pointer is deallocated again (*double-free*) or used again (*use-after-free*), he can contol the process execution [13], [14]. The problem only arises, if the application uses C code or Objective-C without *ARC*. Otherwise the compiler takes care of the memory management [14].

4) Bypassing Exploit Mitigation Techniques: iOS introduced a lot of security features to prevent exploitation.
Traditional *code injection attacks* are mitigated by using the No-eXecute (NX)-Bit of the ARM processor to mark the stack and heap as non-executable. Nethertheless Data Execution Prevention (DEP) can be circumvented through a technique called Return Oriented Programming (ROP) [15]. ROP uses e.g. a *buffer overflow* to hijack the control flow and uses so called *gadgets* which are sequences of instruction found in libraries or the main executable. These gadgets all perform a certain functionality and are executed sequentially. They provide a *turing complete* way of writing the attack payload without the need to inject code [16].
Moreover, *iOS* prevents *stack overflows* with the help of *stack canaries*. The Stack Smashing Protection (SSP) compiler technique places a random value number between the local variables of the stack frame and the functions return address so that in the case of an overflow it can be detected[4]. This protects the return address, saved frame pointer and function arguments. The *canary* is placed at the stack during the function prologue and checked upon the functions epilogue. The protection can be activated with the "-fstack-protector-all" compiler flag [13].
Due to SSP one cannot overwrite the return address but the app may use function pointers which can be overwritten. Davi et al. show a ROP-attack without resorting to returns by using indirect subroutine calls like *BLX r3* [9]. If the attacker can find and load a *gadget* that loads the return address from attacker controlled memory, he can bypass SSP and DEP.
Even if DEP and SSP can be bypassed, there is still a randomization of the process's memory layout. Address Space Layout Randomization (ASLR), which has been introduced in *iOS* 4.3, loads the app executable, dynamic libraries, stack and heap at a different location each time the app is loaded. Third party applications are automatically compiled with *ASLR* if *Xcode* has been used [4]. Although ASLR makes the addresses unpredictable, it is vulnerable to information leakage. Because it enforces only module level randomization, the memory layout of a module can be inferred by getting one absolute address [15].
Most of the time several vulnerabilities are combined to bypass all security mechanisms.

B. Transport Security

iOS provides several APIs for network communication: Depending on the level of control the developer needs, he can choose between the *URL Loading System* and *CFNetwork*.
The first offers a high level of abstraction for retrieving data specified through a *URL*. The latter offers fainer-grained control and can be adopted to custom needs.

1) URL Loading System: The *URL Loading System* is an *Objective-C* API that consists of several classes and *protocols*. They support standard protocols like *HTTP(S)*, *FTP*, local file access or uploading data to a server.
The heart of the system is the *NSURL* class which can be used by *NSURLConnection* to retrieve data via the network. The class enables a delegate to respond to authentication challenges by implementing the appropriate protocol methods. Support authentication challenges are for example: HTTP basic or form authentication, *NTLM* as well as *SSL/TLS* authentication [6]. In general, all pieces of code where authentication is implemented, are of special interest due to the fact that credentials may be hardcoded, stored in a file or database.
In the following, we are having a deeper look at how to circumvent *SSL/TLS* certificate validation so that e.g. debugging code can use self- signed certificates. If the code hasn't been removed, it can break the whole transport security.
Objective-C properly verifies the *SSL/TLS* certificate by default, but by implementing the protocol for *NSURLConnection*, one can disable it.

Figure 5 shows two methods. The first method signals that the delegate can handle the authentication method *NSURLAuthenticationMethodServerTrust* which is a symbolic string for *SSL/TLS*. Therefore it's responsible for performing the actual authentication and evaluating the trust, before creating the *NSURLCredential* object, which is not done in the second method.

```
- (BOOL)connection:(NSURLConnection *)connection
    canAuthenticateAgainstProtectionSpace:(
    NSURLProtectionSpace *)protectionSpace {
  return [protectionSpace.authenticationMethod
      isEqualToString:
      NSURLAuthenticationMethodServerTrust];
}

- (void)connection:(NSURLConnection *)connection
    didReceiveAuthenticationChallenge:(
    NSURLAuthenticationChallenge *)challenge {
    [challenge.sender useCredential:
     [NSURLCredential credentialForTrust:challenge.
         protectionSpace.serverTrust]
         forAuthenticationChallenge:challenge];
}
```

Figure 5. URL Loading System Security prior iOS8 [17]

The disassembly of the first method is depicted in figure 6. Custom certificate handling can be found by searching for *NSURLAuthenticationMethodServerTrust* in the *__nl_symbol_ptr* section. For *iOS 8*, the above methods are deprecated. The replacement is the method *connection:willSendRequestForAuthenticationChallenge:* which can be implemented exactly as *connection:didReceiveAuthenticationChallenge:* shown in Figure 5. Another way to circumvent certificate validation is to use

---

[4] If the binary is compiled with SSP, its symbol table contains the symbols "__stack_chk_fail" and "__stack_chk_guard" [10]. See "otool -I -v" [13]

```
-[AppDelegate connection:
    canAuthenticateAgainstProtectionSpace:]:
push {r4, r5, r7, lr} // save reigsters
movw r0, #0x3bd8  // higher r0 = 0x3bd8
add  r7, sp, #0x8 // set r7 above saved registers
movt r0, #0x0     // lower r0= 0x0
add  r0, pc       // addr of selector(
    authenticationMethod)
ldr  r1, [r0]     // load selector from addr
mov  r0, r3      // r0 = arg2 = NSURLProtectionSpace
// [arg2(r0) authenticationMethod(r1)]
blx  imp___symbolstub1__objc_msgSend
mov  r7, r7
// [[arg2 authenticationMethod]
    retainAutoreleasedReturnValue]
blx  imp___symbolstub1__objc_retainAutoreleased
    ReturnValue
mov  r4, r0      // r4 = return value of branch
movw r0, #0x30e2 // higher r0 = 0x30e2
movt r0, #0x0    // lower  r0 = 0x0
movw r1, #0x3bb8 // higher r1 = 0x3bb8
add  r0, pc      //
    imp___nl_symbol_ptr__NSURLAuthenticationMethod
    ServerTrust
movt r1, #0x0    // r1 = 0x0
add  r1, pc // addr of selector(isEqualToString:)
ldr  r0, [r0] //
    imp___nl_symbol_ptr__NSURLAuthenticationMethod
    ServerTrust
ldr  r1, [r1] // r1 = selector(isEqualToString:)
ldr  r2, [r0] // r2 =
    _NSURLAuthenticationMethodServerTrust
mov  r0, r4 // r0 = arg2
// [arg2(r0) isEqualToString:
    _NSURLAuthenticationMethodServerTrust(r2)]
blx  imp___symbolstub1__objc_msgSend
mov  r5, r0 // r5 = [arg2 isEqualToString:
    _NSURLAuthenticationMethodServerTrust]
mov  r0, r4 // ro = r4
// [[[arg2 authenticationMethod]
    retainAutoreleasedReturnValue] release]
blx  imp___symbolstub1__objc_release
mov  r0, r5          // return r5 => delegate can/
    cannot handle challenge
pop  {r4, r5, r7, pc} // restore registers
```

Figure 6. [NSURLConnectionDelegate connection:canAuthenticateAgainstProtectionSpace:] disassembly

Apples private Application Programming Interfaces (APIs). For instance the class *NSURLRequest* offers the private method *setAllowsAnyHTTPSCertificate:forHost*: which can be called to allow exceptions for certain hosts. Another way is to define a category on *NSURLRequest* and provide a runtime patch for the method *allowsAnyHTTPSCertificateForHost:* by returning YES for all trusted hosts [18].

As both possibilities use Apples private APIs, it might be detected by the *App Store* review process and could get rejected. Nethertheless it's worth having a closer look at it.

2) CFNetwork: A low-level framework for network communication is *CFNetwork*. Based on *BSD sockets*, it offers a way of reading and writing synchronously or asynchronously bytes to or from a stream with *CFReadStream* and *CFWriteStream*. In contrast to the *URL Loading System* where the main purpose is data access, *CFNetwork*'s focus is more on network protocols [19]. Due to the fact that the API is low-level, the fainer-grained control allows the developer to influence the *SSL/TLS* certificate validation in such a way that not just only the validation of the *trust chain* can be disabled, but also a custom name can be used for name verification or disabling it at all. Moreover, the security level can be set to a specific *SSL/TLS* version. Figure 7 shows the methods to achieve this on a *CFReadStream* or *CFWriteStream*. The first argument of the methods is the stream. The second parameters specifies the property on which a value shall be set and the third argument is a dictionary containing the appropriate values. E.g. *kCFStreamSSLValidatesCertificateChain* may be used as key with value *NO* in the dictionary for the key *kCFStreamPropertySSLSettings* to disable certificate validation.

```
// populate settings dictionary
CFReadStreamSetProperty(inputStream, <kCFStream* key
    >, settings);
CFWriteStreamSetProperty(outputStream, <kCFStream*
    key>, settings);
```

Figure 7. CFNetwork Transport Security [20]

The following listing shows the available keys and values which are relevant in terms of security:

1) kCFStreamPropertySSLSettings
   - kCFStreamSSLValidatesCertificateChain: Disable the *chain of trust* validation.
   - kCFStreamSSLPeerName: Sets a custom name used for *common name* verification. Can be disabled at all with kCFNull.
2) kCFStreamSSLLevel
   - kCFStreamSocketSecurityLevelNone
   - kCFStreamSocketSecurityLevelSSLv2
   - kCFStreamSocketSecurityLevelSSLv3
   - kCFStreamSocketSecurityLevelTLSv1
   - kCFStreamSocketSecurityLevelNegotiatedSSL

There is also an equivalent *Objective-C* API for *CFNetwork*. *CFStreams* are toll-free-bridge to the class *NSStream*. Therefore one also has to check these *SSL* versions. The constants are prefixed with *NSStreamSocketSecurityLevel* and set with the *selector setProperty:forKey:* [6].

3) SSL/TLS security flaws: If any of these values are set, one should take a closer look at it. Disabling the certificate verification enables man-in-the-middle (MitM) attacks, where an attacker can on the side intercept the whole traffic and on the other side may be able to inject code or data.

Disabling the name verification is nearly as worse as don't checking the *trust chain*. It enables the attacker to present any valid certificate. Using a lower *SSL/TLS* version offers a bigger attack surface because there are certain attacks on older versions like *POODLE* or *BEAST*. The first affects SSL 3.0, the latter also TLS 1.0 and enables an attacker to leak information such as decrypted *HTTPS cookies* [21], [22].

C. IPC

Apple provides a simple form of *IPC* mechanism to allow apps to exchange data and launch other apps through *URL* *schemes*. Every app can register a scheme, which is hardcoded in the applications *Info.plist* under the key *CFBundleURLTypes*. The value of *CFBundleURLSchemes* contains the actual scheme [6].

An application can handle a custom *URL scheme* by implementing the method *application:handleOpenURL:* or *application:openURL:sourceApplication:annotation* of the *UIApplicationDelegate* protocol. The first is deprecated since *iOS* 4.2. The latter is the replacement which improves security due to the fact that the *BundleID* of the calling application is supplied.

For example the Facebook app can be launched by opening the *URL scheme* depicted in figure 8. If the user hasn't logged out, the user will be redirect to his profile in the Facebook app. Dhanjani shows that this can be used to decloak[5] the identity of a person through the injection of the malicious iframe shown in the figure. To support several actions, a developer can distinguish between transactions encoded in the *URL*. By default, the *IPC* mechanism doesn't require user authorization, hence the application either has to implement it itself or ensure that transactions aren't allowed to modify or delete user data. *URL schemes* can be exposed by examing the strings[6] [17] as they are likely to be hardcoded or *reverse engineering* the scheme registration methods. To prevent misuse or a possible attack, the developer should do thorough input validation as well as asking for authorization before performing any action [17]. An actual attack depends on the functionality implemented by the developer and cannot be generalized.

```
<iframe src="fb://profile/"></iframe>
```

Figure 8. MitM De-cloaking [17]

D. Data Storage

Every *iOS* device has a dedicated AES crypto engine for efficient data encryption. One has to distinguish between the encryption of the whole file system and the protection of data offering additional encryption.

The first is implement to allow a fast *remote wipe* so that in the case of a device theft or loss, the data cannot be accessed. In order to do that, a random *file system key* is generated while the operating system is installed or a *remote wipe* has been executed. The key is stored in *Effaceable Storage* and renders the data useless by deleting they key.

1) Data Protection API: Despite the file system encryption, one can further protect data by using the *Data Protection API* which adds an additional layer of encryption. Depending on the need and functionality there are several protection levels available:

- Complete protection: Renders a file inaccessible after the device has been locked.
- Protected unless open: A device unlock offers the appliaction to obtain a file handle so that the file is not encrypted unless it has been closed (uses elliptic curves [4]).
- Until first user authentication: This protection is equivalent to full-disk encryption on the desktop.
- No protection: Files are only encrypted with the file system key.

The API uses a hierarchy of keys: Each device has a Unique device ID (UID) 256 Bit AES key so that files can be cryptographically tied to the device. Furthermore, for every file, a random 256 Bit AES *per-file key* is generated. To implement the various protections, each level has it's distinct *class key* which is used to wrap the *per-file key*. The wrapped key is stored in the file's metadata. The *class key* is protected with the UID and passcode/*touchID* for some classes.

The decryption works by decrypting the file's metadata, unwrapping the wrapped per-*file key* with the *class key* and finally passing the *per-file key* to the AES engine [4].

The API can be used either through the class *NS(Mutable)Data* or *NSFileManager*. The protection level constant names differ between these classes in that they are prefixed with either *NSDataWriting* or *NS*. The constants to look out for are the followings:

- FileProtectionNone
- FileProtectionComplete
- FileProtectionUnlessOpen
- FileProtectionCompleteUntilFirstUserAuthentication [13]

Moreover, the protections levels differ in their representation. While for *NSFileManager* the levels are declared as strings, the levels for *NS(Mutable)Data* are integers.

To detect the protection with *NSFileManager* in the disassembly, one has to look out for the selector *createFileAtPath:contents:attributes:* on an instance of *NSFileManager* where the parameter for *attributes* is a dictionary with the protection level for the key *NSFileProtectionKey* which is also a string [13]. The protection levels can also be detected with Hopper by searching for the protection level in the *__nl_symbol_ptr* section.

For *NS(Mutable)Data* one has to check which integer value has been passed to the *options* parameter of the selector *writeToFile:options:error:*. The protection levels reach from 1 to 4 in the order of the enumeration given above.

2) Keychain: Sensitive data like credentials or access tokens need stronger protection. *Keychain*, an encrypted database, is made exactly for this purpose. Apps are restricted to their own *keychain* through the *keychain access group* they belong to. Items are protected with a class similiar like the *Data Protection API* used it. There are two important ways of modifying the protections: *SecItemAdd* can be used to add an item to the *keychain* and *SecItemUpdate* can update it [13].

The protection levels are:

- kSecAttrAccessibleAlways(ThisDeviceOnly)
- kSecAttrAccessibleWhenUnlocked(ThisDeviceOnly)
- kSecAttrAccessibleAfterFirstUnlock(ThisDeviceOnly)

---

[5] His attack uses a MitM attack (transparent proxy on fake WiFi AP) to intercept the traffic and inject custom data with *BurpSuite*

[6] Use the command "strings <App.app>" to view all string constants

Every protection constant begins with *kSecAttr*. Moreover, the *ThisDeviceOnly* prefixed constants are further protected with the passcode/*touchID* and are not part of any backup. These non-migratory levels are wrapped with the *UID* to tie it to the device. The default protection is *kSecAttrAccessibleAlways* so that the item is accessable at any time and can be migrated to another device and is also included in backups [4].

3) NSUserDefaults: Developers have a convenient way to store preferences through the *NSUserDefaults* class. Values can be stored by calling the selector *setObject:forKey:*[7] and retrieved via *objectForKey:* [6]. To store sensitive data or credentials, the developer should use the *keychain* instead because the preference file can be downloaded via the Apple File Communication Protocol (AFC) protocol.

4) Pasteboard: The copy and paste functionality of the operating system is implemented within the class *UIPasteboard*. It enables apps the easy exchange of data in the app itself as well as between apps. The general pasteboard can be accessed by every application by obtaining its instance with *[UIPasteboard generalPasteboard]*. It's persistent[8] so that it endures even after a reboot. Standard application pasteboards are not persisent by default, but can be set to be. By knowing the name of the applications pasteboard, another app can access the data stored inside it [6].

A developer should use the pasteboard functionality with care due to the fact that the general pasteboard makes the stored data public. App specific pasteboards are public too, if they are persistent [6]. The pasteboard is often used for a migration from a free to a paid version. In general the pasteboard shouldn't be used for sensitive data. Furthermore, the copy/paste functionality can be deactivated for fields with sensitive data [18].

5) SQL: Application data is very often managed with the help of *SQL*. Unsanitized user data can lead to injection attacks where data is interpreted as code. The risk can be circumvented by using parameterized statements so that user input gets sanitized correctly and no code can be injected [23].

Figure 9 shows a *SQL* query which uses input from an external source. Possible attacks could leak information, gain database access or possibly delete all data.

```
sqlite3 *db;
char sqlbuf[256], *err;
sqlite3_open("sample.db", &db);
snprintf(sqlbuf, sizeof(sqlbuf), "SELECT * FROM
    table WHERE user = %s", attackerControlled);
sqlite3_exec(db, sqlbuf, NULL, NULL, &err);
```

Figure 9. SQL Injection [20]

E. Non Persistent Data

Despite the data at rest, the developer should also secure non persistent information.

1) Keyboard Cache: The autocorrection feature, if turned on, uses the input of text fields inside the applications. Except for strings with digits only or very small text, the value is not cached. Due to this caching, senstive data or even credentials can be leaked. The feature can be disabled at all or only on particular text fields: Either set *autocorrectionType* to *UITextAutocorrectionNo* or mark it as secure by setting *secureTextEntry* to YES [24].

2) Logging: Developers should be aware of the fact that logging with *NSLog* redirects the output to the Apple system log facility so that they can be viewed in *Xcode*. Therefore one has to pay attention to what actual gets logged. Especially debugging code can reveal sensitive information or even credentials. A countermeasure against unintentional logging is a pre-processor macro that performs conditional compilation [13].

3) App transitions: When an application changes from foreground to background, *iOS* takes a screenshot of the running application to offer a zoom out and in animation [17]. This can leak[9] private information if the application does not protect sensitive data. The protection can be achieved by setting fields hidden if the application enters background and make them visible if the application becomes active again.

The screenshots are secured with *Data Protection* (*NSFileProtectionComplete*) if the passcode or touchID is used.

F. UIWebView

Developers want to keep their users as long as possible in their apps. They can leverage the *UIWebView* to achieve this. It can render web pages, PDFs, images, office documents etc. so that the user can stay inside the app. Like every browser, the rendering engine can execute *JavaScript* so that it can be affected by Cross-Site Scripting (XSS) vulnerabilities if user input has not been sanitized correctly. Therefore code injection is possible which can lead to execution of arbitrary *HTML* and *JavaScript* code. A XSS attack against an older Skype version made it possible to upload the address book to a external server [2].

Often developers use the *UIWebView* as GUI and imlement a *JavaScript* to *Objective-C* bridge. Thus Cross-Site Scripting in the *UIWebView* can be much more severe compared to classical XSS attacks.

Figure 10 depicts the rendering of a local HTML page from the application bundle. The index.html file contains the line ”<script>document.write(myvar);</script>” that adds the code from *myvar* to the Document Object Model (DOM) of the web view [13]. If an attacker has control over the variable *myvar*, he could inject arbitrary code (*DOM based XSS*). This can lead to data theft or even Cross-Site Request Forgery as

---

[7] The preference file can be found at /private/var/mobile/Containers/Bundle/Application/<UUID>/Library/Preferences/<BundleIdentifier>.plist

[8] Persistent pasteboards entries are stored unencrypted in the plist /private/var/mobile/Library/Caches/com.apple.UIKit.pboard/pasteboardDB and data is base64 encoded [6]

[9] The taken screenshot can be found at /private/var/mobile/Containers/Data/Application/<UUID>/Library/Caches/Snapshots/<BundleIdentifier>/<BundleIdentifier> [6]

```
let myvar = "<script>alert('XSS!');</script>"
let js = "var myvar=\"" + myvar + "\";"
webView.stringByEvaluatingJavaScriptFromString(js)

webView.loadRequest(
  NSURLRequest(
    URL: NSURL(
      fileURLWithPath: NSBundle.mainBundle().
          pathForResource("index", ofType: "html")!
    )!
  )
)
```

Figure 10. XSS UIWebView (Swift version, ported from [13])

the attacker has control over the DOM.

Another important aspect to consider when using *UIWebView* is that it doesn't show the *URL* and can be used to spoof the user interface. Therefore the user can be tricked into thinking the displayed site is the trusted party he wants to visit [17].

G. *The Big Picture*

The previous sections showed common mistakes and security threats that can happen to a *iOS* developer. This chapter is meant to give a short summary: Developers should leverage the *Data Protection API* to protect files and store sensitive data like credentials in the more secure *Keychain*. *NSUserDefaults* or the *UIPasteBoard* are a comfortable way of storing and exchanging data but lack any kind of encryption. Sensitive data in the user interface needs special treatment so that it cannot be copied to the system pasteboard, cached for autocorrection or get exposed into application screenshots.

Apps always have to deal with data management. *SQL* is a common way of storing data but imposes the threat of *SQL injection* when not using *prepared statements*. Exchanging information or sending data over the network should implicate encryption and the proper use of certificate handling. Otherwise MitM attacks on the one hand can leak data and on the other hand can even manipulate it.

Even simple logging statements like *NSLog* or *printf* can lead to *format string* vulnerabilities and leak data to the system log.

IV. CONCLUSION

iOS introduces a lot of security mechanisms. But this doesn't mean that applications are magically safe. Instead developers need to take care of common security pitfalls and always keep security in mind. Nearly all kind of threats arise from unsanitized user input which leads to injection of code. Therefore validation on user controlled data is always necessary.

Focussing on the higher languages *Objective-C* or *Swift* prevents common memory threats introduced through the C programming language.

Often debug code is the main source of bugs which disables e.g. the correct *SSL/TLS* validation to enable self-signed certificates in test environments.

Finally data is not hidden in source code. Login tokens, encryption keys etc. should never be stored plaintext because *reverse engineering* as well as debugging or runtime patching through the dynamic capabilities of *Objective-C* reveals them. Even anti-debugging techniques and obfuscation may not protect data but instead only harden the analysis.

REFERENCES

[1] Apple announces 1 million apps in the app store, more than 1 billion songs played on itunes radio. Last visited January 15, 2015. [Online]. Available: http://www.theverge.com/2013/10/22/4866302/apple-announces-1-million-apps-in-the-app-store

[2] Superevr. Skype xss explained. Last visited January 15, 2015. [Online]. Available: https://www.superevr.com/blog/2011/skype-xss-explained/

[3] The starbucks bug: not as awful as reported. http://www.zdnet.com/article/the-starbucks-bug-not-as-awful-as-reported/. Last visited January, 15, 2015.

[4] ios security. Apple Inc. [Online]. Available: https://www.apple.com/privacy/docs/iOS_Security_Guide_Oct_2014.pdf

[5] App store review guidelines. Apple Inc. Last visited January 14, 2015. [Online]. Available: https://developer.apple.com/app-store/review/guidelines

[6] M. Binna, "ios application security," Master's thesis, Ruhr University Bochum, 2011. [Online]. Available: https://www.syssec.rub.de/media/emma/arbeiten/2012/11/16/2011-06-06-Dipl-Binna.pdf

[7] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann, 2004.

[8] ios abi function call guide. Apple Inc. [Online]. Available: https://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," *System Security Lab-Ruhr University Bochum, Tech. Rep*, 2010.

[10] M. RENARD, "Practical ios apps hacking," *G 2 reHack 012*, p. 14.

[11] Secure coding guide. Apple Inc. Last visited January 2, 2015. [Online]. Available: https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html

[12] scut / team teso. Exploiting format string vulnerabilities. 17th Chaos Communication Congress. [Online]. Available: http://julianor.tripod.com/bc/formatstring-1.2.pdf

[13] D. Chell, "Whitepaper ios application (in)security," MDSec Consulting Ltd. [Online]. Available: https://www.mdsec.co.uk/research/iOS_Application_Insecurity_wp_v1.0_final.pdf

[14] H. Dwivedi, *Mobile application security*. Tata McGraw-Hill Education, 2010.

[15] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee, "Jekyll on ios: When benign apps become evil." in *Usenix Security*, vol. 13, 2013.

[16] T. Kornau, "Return oriented programming for the arm architecture," *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[17] N. Dhanjani, "New age application attacks against apple's ios (and countermeasures)," *Blackhat Barcelona*, 2011.

[18] D. Thiel. Secure development on ios. iSEC Partners. [Online]. Available: https://www.isecpartners.com/media/11221/secure_development_on_ios.pdf

[19] Cfnetwork programming guide: Cfnetwork concepts. Apple Inc. Last visited January 6, 2015. [Online]. Available: https://developer.apple.com/library/ios/documentation/Networking/Conceptual/CFNetwork/Introduction/Introduction.html

[20] I. van Sprundel. Writing secure ios applications. Chaos Communication Camp 2011. IOActive. [Online]. Available: http://events.ccc.de/camp/2011/Fahrplan/attachments/1846_SECURE_iOS_APPS_LOGIN.pdf

[21] B. Möller, T. Duong, and K. Kotowicz, "This poodle bites: Exploiting the ssl 3.0 fallback," 2014.

[22] Z. Jeelani, "An insight of ssl security attacks."

[23] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, 2011, p. 2.

[24] J. Zdziarski, *Hacking and securing iOS applications: stealing data, hijacking software, and how to prevent it.* " O'Reilly Media, Inc.", 2012.