

Whitepaper

iOS Application (In)Security

May 2012



Prepared by: Dominic Chell

E-Mail: research@mdsec.co.uk

MDSec Consulting Ltd

Contents

1.	Introduction	4
2.	Background	5
2.1.	iOS Security Features Primer	5
	Code Signing	5
	Exploit Mitigation Features	5
	Sandboxing	6
	Encryption	7
2.2.	iOS Application Overview	7
2.3.	Previous Work	7
3.	Black Box Assessment	8
	Introduction	8
3.1.	Decrypting AppStore Binaries	8
3.2.	Locating the Position Independent Executable	10
3.3.	Identifying the use of Stack Smashing Protection	12
3.4.	Identifying the use of Automatic Reference Counting	12
3.5.	Inspecting the Binary	13
3.6.	Manipulating the Runtime	14
	Example: Bypassing Jailbreak Detection	14
3.7.	Defending the Binary	17
3.8.	Conclusion	19
4.	Auditing Insecure API Usage	20
	Introduction	20
4.1.	Evaluating Transport Security	20
4.2.	Abusing Protocol Handlers	25
4.3.	Locating Insecure Data Storage	29
4.4.	Attacking the iOS Keychain	33
4.5.	Conducting Cross-Site Scripting (XSS) through UIWebViews	36

4.6.	Attacking XML Processors	37
4.7.	SQL Injection	39
4.8.	Filesystem Interaction	41
4.9.	Geo-Location	43
4.10.	Logging	44
4.11.	Backgrounding	44
5.	Memory Corruption Issues	45
	Introduction	45
5.1.	Format Strings	45
5.2.	Object Use-after-Free	47
6.	Conclusions	49
7.	iOS App Compliance Checklist	50
8.	About MDSec	52
9.	Acknowledgements	52
10.	References.....	53

1. Introduction

In the last year, MDSec's consultants have performed an increasing number of security assessments of iOS applications and their supporting architecture where data security is paramount, specifically the retail/business banking sector.

Smartphones have become commonplace not only in the consumer markets but now also in the enterprise. Smartphones combine the traditional mobile features with computer like functionality. The increased processing power and memory of the modern smartphone has led to a surge in mobile application development as developers look to take advantage of the feature rich offerings of the platform. Application development is indeed now so popular that Apple's trademark slogan "There's an app for that" is bordering on reality.

A growing trend that we have witnessed over 2011 has been an increase in demand for security assessments of mobile applications, with iOS and Android apps being the front-runners. Market research conducted by NetApplications [1] shows that iOS devices control approximately 52% of the global mobile market.

Drawn from MDSec's hands-on training course on iOS Application Security, the focus of this whitepaper is to document the categories of issues that typically affect iOS applications and provide a single reference point for not only security assessors but also developers wishing to adhere to security best practice.

2. Background

2.1. iOS Security Features Primer

Before discussing the security issues that affect iOS applications, it is important to have a fundamental understanding of the security features of the platform, not only to provide context to application vulnerabilities but also to highlight opt-in features that an application can take advantage of.

The core security features of the iOS platform can be summarised as:

- Code Signing
- Generic native language exploit mitigations
 - Address Space Layout Randomisation
 - Non executable memory
 - Stack Smashing Protection
- Process level sandboxing
 - Also known as Seat Belt
- Data at rest encryption

A comprehensive review of these features can be found within the whitepaper “Apple iOS 4 Security Evaluation” by Dino Dai Zovi [2] which provided much of the foundation for the details discussed in this section.

Code Signing

Code signing is a runtime security feature of the platform that attempts to prevent unauthorised applications running on the device by validating the application signature each time it is executed. Additionally, applications may also only execute code signed by a valid, trusted signature.

For an application to be run on the device, it must first be signed by a trusted certificate. Developers can install trusted certificates on a device through a provisioning profile signed by Apple. The provisioning profile contains the embedded developer certificate and set of entitlements that the developer may grant to applications. In production applications, all code must be signed by Apple, this is performed during the AppStore submission process. This process allows Apple some degree of control over apps and to govern the APIs and functionality used by developers. For example, Apple looks to prevent apps using private APIs or downloading to installing executable code [3].

Exploit Mitigation Features

Address Space Layout Randomisation (ASLR) [4] is a security feature that attempts to increase the complexity of vulnerability exploitation by randomising where data and code is mapped in a processes address space. ASLR was first introduced to iOS in version beta 4.3 and since inception has gradually improved with each release. The primary weakness in the ASLR

implementation was the lack of relocation of the dyld, this was addressed with the release of iOS 5.0. Applications can have ASLR applied in two different flavours, either partial ASLR or full ASLR depending on whether they have been compiled with support for Position Independent Execution (PIE). In a full ASLR scenario, all the application memory regions are randomised and iOS will load a PIE enabled binary at a random address each time it is executed. An application with partial ASLR will load the base binary at a fixed address and use a static location for the dynamic linker (dyld). An in-depth assessment of ASLR in iOS has been conducted by Stefan Esser and is recommended reading for those looking to gain a greater understanding [5].

ASLR is designed to frustrate exploitation due to the lack of knowledge an attacker will have of the process layout in memory and thus the addresses they need to target. However, there are a number of techniques that can weaken its effectiveness. The most common of these techniques is memory revelation. This is where a separate vulnerability is used to leak or confirm memory layout to an attacker prior exploitation of a vulnerability that will yield arbitrary code execution.

In an attempt to further mitigate exploitation of native language vulnerabilities, iOS combines ASLR with the implementation of a "W^X" non-executable memory policy, meaning that memory pages cannot be marked as writeable and executable at the same time. As part of this policy, executable memory pages that are marked as writeable cannot also be later marked back to executable. In many ways this is similar to the Data Execution Protection (DEP) features implemented by Microsoft Windows, Linux and Mac OS X desktop OS'. While non-executable memory alone can be trivially bypassed using Return Orientated Programming (ROP) based payloads, the complexity of exploitation is significantly increased when compounded with ASLR and Mandatory Code Signing.

iOS applications can look to add additional exploit mitigation at compile time through stack smashing protection. Stack canaries in particular introduce some protection against buffer overflows by placing a random, known value before the local variables. The stack canary is checked upon return of the function. If an overflow occurs and the canary is corrupted, the application is able to detect and protect against the overflow.

Sandboxing

All third party applications on iOS run within a sandbox; this is a self-contained environment that isolates applications not only from other applications but also the operating system. While applications all run as the "mobile" operating system user, they are contained within a unique directory on the filesystem and separation is maintained by the XNU Sandbox kernel extension. The operations that can be performed in the sandbox are governed

by the seatbelt profile. Third party applications are assigned the “container” profile which will generally limit file access to the application home directory, allow read access to media, read and write to the address book as well as unrestricted access to outbound network connections, with the exception of `launchd`’s network sockets. See “The Apple Sandbox” [6] for recommended further reading.

Encryption

By default, all data on the iOS filesystem is encrypted using block-based encryption (AES) with the File System Key, which is stored on the flash. The filesystem is encrypted only at rest; when the device is turned on the hardware based crypto accelerator unlocks the filesystem.

In addition to the hardware encryption, individual files and keychain items can be encrypted using the Data Protection (DP) API that uses a key derived from the device passcode. Consequently, when the device is locked, content encrypted using the DP API will be inaccessible unless cached in memory. Third party applications wishing to encrypt sensitive data should employ the Data Protection API to do so. However consideration should be given for background processes how they will behave if the at-rest becomes unavailable due to the device becoming locked.

2.2. iOS Application Overview

Third party iOS applications use the Cocoa Touch API to interact with the device. This framework provides a means of abstraction from the OS and is written in Objective-C, a superset of C.

Development of iOS applications can be performed using the freely available XCode IDE for OS X. XCode provides a simulator for compiling and running applications, however it should be noted that this is simulation rather than emulation. In order to run the application on a non-jailbroken device, you must be a member of the subscription-based iOS Developer Program and have a development certificate.

2.3. Previous Work

To our knowledge, there are two noteworthy presentations on evaluating iOS application security. Both of these presentations are recommended reading for those performing iOS application assessments:

- “Auditing iPhone and iPad Applications” by Ilja van Sprundel [7]
- “Secure Development on iOS” by David Thiel [8]

3. Black Box Assessment

Introduction

A common assumption made by organisations is that an application's inner workings are in some way protected from an attacker who does not have access to the application source code. In practice, it is a relatively straightforward process for an attacker to access the decrypted application, locate the key methods it contains, hook into them at runtime, and alter variables and execution flow. This generally requires the following steps:

3.1. Decrypting AppStore Binaries

Apps originating from the AppStore are protected by Apple's binary encryption scheme. These apps will be decrypted at runtime by the kernel's mach loader; as such recovering the decrypted files is a relatively straightforward process. Removing this encryption allows the attacker to get a greater understanding of how the binary works, the internal class structure and to get the binary in a suitable state for reverse engineering.

Removing the AppStore encryption can be achieved by letting the loader decrypt the app then using the debugger to dump out the decrypted image. This process has been automated by two applications available via Cydia, namely Crackulous [9] and AppCrack. However, the process can also be performed manually using GDB.

Encrypted binaries can be identified by the value in the "cryptid" field of the LC_ENCRYPTION_INFO [10] load command, for example:

```
mdsec-iPhone:/var/mobile/Applications/E938B6D0-9ADE-4CD6-83B8-712D0549426D/99Bottles.app root# otool -l 99Bottles | grep -A 4 LC_ENCRYPTION_INFO
cmd LC_ENCRYPTION_INFO
cmdsize 20
cryptoff 4096
cryptsize 12288
cryptid 1
```

In some instances, apps may be compiled for multiple architectures; these are known as fat binaries. The architectures an app is compiled for can again be identified using `otool`:

```
mdsec-iPhone:/var/mobile/Applications/68E3B644-9203-4B8F-A707-A52E23B793B6/Kik.app root# otool -f Kik
Fat headers
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
```



```

cputype 12
cpusubtype 6
capabilities 0x0
offset 4096
size 865152
align 2^12 (4096)
architecture 1
cputype 12
cpusubtype 9
capabilities 0x0
offset 872448
size 867488
align 2^12 (4096)

```

In the above example cputype 12 with cpusubtype 6 corresponds to ARM v6 and cputype 12 with cpusubtype 9 is ARM v7; if required a binary can be “thinned” to the desired architecture using `lipo`.

To retrieve the decrypted segment of the app, we must first let the loader run; this can be achieved by setting a breakpoint on “doModInitFunctions” which is called after all objects have been loaded:

```

mdsec-iPhone:/var/mobile/Applications/E938B6D0-9ADE-4CD6-83B8-712D0549426D/99Bottles.app root# gdb --quiet -e ./99Bottles
Reading symbols for shared libraries . done
(gdb) set sharedlibrary load-rules ".*" ".*" none
(gdb) set inferior-auto-start-dyld off
(gdb) set sharedlibrary preload-libraries off
(gdb) rb doModInitFunctions
Breakpoint 1 at 0x2fe0ce36
<function, no debug info>
__dyld__ZN16ImageLoaderMachO18doModInitFunctionsERKN11ImageLoader11LinkContextE;
(gdb) r
Starting program: /private/var/mobile/Applications/E938B6D0-9ADE-4CD6-83B8-712D0549426D/99Bottles.app/99Bottles
Breakpoint 1, 0x2fe0ce36 in
__dyld__ZN16ImageLoaderMachO18doModInitFunctionsERKN11ImageLoader11LinkContextE
()
(gdb)

```

At this stage, the loader has decrypted the app and we can dump the clear text segments directly from memory. The location of the encrypted segment is specified by the `cryptoff` value in the `LC_ENCRYPTION_INFO` load command, which gives the offset relative to the header. Consequently, the encrypted segment begins at offset 0x2000 (cryptoff of 0x1000 (4096) plus the start address of 0x1000). The address range to dump memory is simply the address

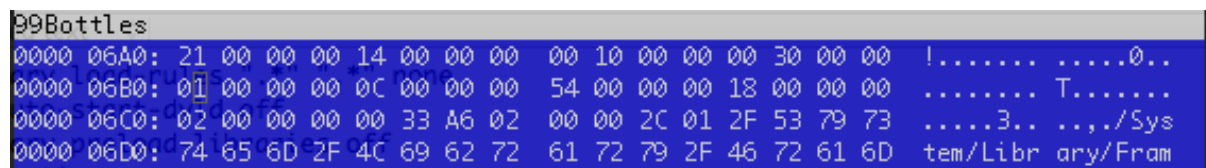
of the start of the encrypted segment, plus the size of the encrypted segment that is specified by the cryptsize (12288, 0x3000), resulting in an end address of 0x5000 (0x2000 + 0x3000). The decrypted segment can be retrieved using the “dump memory” GDB command:

```
(gdb) dump memory 99bottles.dec 0x2000 (0x2000 + 0x3000)
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) q
mdsec-iPhone:/var/mobile/Applications/E938B6D0-9ADE-4CD6-83B8-
712D0549426D/99Bottles.app root# ls -al 99bottles.dec
-rw-r--r-- 1 root mobile 12288 Mar  4 16:31 99bottles.dec
```

The resultant file should be exactly the same size as our cryptsize value. The decrypted section can then be written to the original binary, replacing the original encrypted segment:

```
mdsec-iPhone:/var/mobile/Applications/E938B6D0-9ADE-4CD6-83B8-
712D0549426D/99Bottles.app root# dd seek=4096 bs=1 conv=notrunc
if=./99bottles.dec of=99Bottles
12288+0 records in
12288+0 records out
12288 bytes (12 kB) copied, 0.471737 s, 26.0 kB/s
```

Finally, the cryptid value must be set to 0 to denote that the file is no longer encrypted and the loader should not attempt to decrypt it. Using vbindiff, search for the location of the LC_ENCRYPTION_INFO command; this can be found by searching for the hex bytes 2100000014000000. From this location flip the cryptid value to 0, which is located 16 bytes in advance of the cmdsize (0x21000000):



The image shows a hex dump of the LC_ENCRYPTION_INFO command. The first two lines are highlighted in blue. The first line shows the command type 0x000006A0 and the command length 0x00000014. The second line shows the cryptid value 0x00000000 and the cmdsize value 0x00000014. The third line shows the cryptid value 0x00000000 and the cmdsize value 0x00000014. The fourth line shows the cryptid value 0x00000000 and the cmdsize value 0x00000014.

Figure 1 - Hex dump of LC_ENCRYPTION_INFO

At this stage, the app should be decrypted and will run as normal once code signed again.

3.2. Locating the Position Independent Executable

Position Independent Executable (PIE) is an exploit mitigation security feature that allows an application to take full advantage of ASLR. In order for this to happen, the app must be compiled using the “-fPIE -pie” flag; using XCode this can be enabled/disabled using the “Generate Position-Dependent Code” option from the compiler code generation build setting. As previously mentioned, an app compiled without PIE will load the executable at a fixed address; consider the following simple example that will print the address of

the main function:

```
int main(int argc, const char* argv[])
{
    NSLog(@"Main: %p\n", main);
    return 0;
}
```

Compiling the above application without PIE and running on the iPhone, we can see that despite system wide ASLR the main executable is loaded at a fixed address:

```
mdsec-iPhone:~ root# for i in `seq 1 5`; do ./nopie-main;done
2012-03-01 16:56:17.772 nopie-main[8943:707] Main: 0x2f3d
2012-03-01 16:56:17.805 nopie-main[8944:707] Main: 0x2f3d
2012-03-01 16:56:17.837 nopie-main[8945:707] Main: 0x2f3d
2012-03-01 16:56:17.870 nopie-main[8946:707] Main: 0x2f3d
2012-03-01 16:56:17.905 nopie-main[8947:707] Main: 0x2f3d
```

Recompiling the same application with PIE, we can see the app now loads the main executable at a dynamic address:

```
mdsec-iPhone:~ root# for i in `seq 1 5`; do ./pie-main;done
2012-03-01 16:57:32.175 pie-main[8949:707] Main: 0x2af39
2012-03-01 16:57:32.208 pie-main[8950:707] Main: 0x3bf39
2012-03-01 16:57:32.241 pie-main[8951:707] Main: 0x3f39
2012-03-01 16:57:32.277 pie-main[8952:707] Main: 0x8cf39
2012-03-01 16:57:32.310 pie-main[8953:707] Main: 0x30f39
```

From a blackbox perspective, the presence of PIE can be verified using the otool application, which provides functionality to inspect the Mach-O header. For example, comparing the two binaries above we can easily detect the PIE executable:

```
mdsec-iPhone:~ root# otool -hv pie-main nopie-main
pie-main:
Mach header
      magic cputype cpusubtype  caps   filetype ncmds sizeofcmds      flags
      MH_MAGIC   ARM             9   0x00   EXECUTE    18      1948   NOUNDEFS
DYLDLINK TWOLEVEL PIE

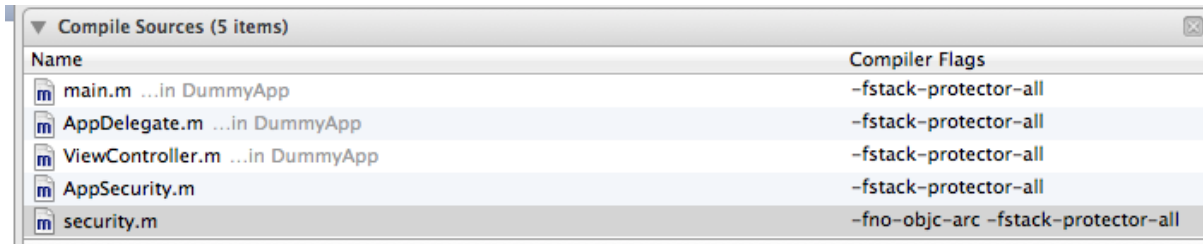
nopie-main:
Mach header
      magic cputype cpusubtype  caps   filetype ncmds sizeofcmds      flags
      MH_MAGIC   ARM             9   0x00   EXECUTE    18      1948   NOUNDEFS
DYLDLINK TWOLEVEL
```

In iOS 5, all of the built-in applications are compiled with PIE by default, however in practice third-party applications do not commonly take advantage

of this protection feature [5].

3.3. Identifying the use of Stack Smashing Protection

As previously noted, iOS applications can apply stack smashing protection at compile time. This can be achieved by specifying the `-fstack-protector-all` compiler flag, as shown below:



Name	Compiler Flags
main.m ...in DummyApp	-fstack-protector-all
AppDelegate.m ...in DummyApp	-fstack-protector-all
ViewController.m ...in DummyApp	-fstack-protector-all
AppSecurity.m	-fstack-protector-all
security.m	-fno-objc-arc -fstack-protector-all

Figure 2 – Xcode compile sources

When an app is compiled with stack smashing protection, a known value or “canary” is placed on the stack directly 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. The compiler uses a heuristic to intelligently apply stack protection to a function, typically functions using character arrays.

From a black box perspective, the presence of stack canaries can be identified by 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`” and “`__stack_chk_guard`”. The symbol table from an app can be dumped using the `otool` application:

```
$ otool -I -v DummyApp | grep stack
0x00003fc4    14  __stack_chk_fail
0x0000400c    14  __stack_chk_fail
0x0000406c    15  __stack_chk_guard
```

3.4. Identifying the use of 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 apps (See section 5.2).

ARC can be enabled in an application within XCode by setting the compiler option “Objective-C Automatic Reference Counting” to “yes”. To identify the presence of ARC in a black box review of a compiled app, an evaluator can look for the presence of ARC related symbols in the symbol table, as shown below:

```
$ otool -I -v DummyApp-ARC | grep "_objc_release"
0x00003fe8    181  _objc_release
```

```
0x00004030    181  _objc_release
$
```

The symbols that highlight the presence of ARC are:

- `_objc_retainAutoreleaseReturnValue`
- `_objc_autoreleaseReturnValue`
- `_objc_storeStrong`
- `_objc_retain`
- `_objc_release`
- `_objc_retainAutoreleasedReturnValue`

At compile time, ARC can be explicitly disabled on specific source files by using the `-fno-objc-arc` compiler flag and this should be highlighted as part any white box iOS application assessment.

3.5. Inspecting the Binary

With a decrypted binary, there is a wealth of information in the `__OBJC` segment that can be useful to a reverse engineer. The `__OBJC` segment provides details on the internal classes, methods and variables used in the app; this information is particularly useful when looking to understand how the app functions, patching the app or hooking the app at runtime.

Parsing the `__OBJC` segment can be performed using the `class-dump-z` [11] application; for example running the previously decrypted 99Bottles app through `class-dump-z` yields the following:

```
@interface BottleLayer : CALayer {
@private
    BOOL flown;
}
@property(assign, nonatomic) BOOL flown;
-(void)drawInContext:(CGContextRef)context;
-(void)jiggle;
-(void)flyAway;
-(void)animationDidStop:(id)animation finished:(BOOL)finished;
-(void)dealloc;
@end

__attribute__((visibility("hidden")))
@interface RootViewController : UIViewController <UIActionSheetDelegate> {
@private
    UILabel* numberDisplay;
    NSMutableArray* marr;
    Player* player;
    UIView* wall;
```

```
BottleLayer* currentBottle;
NSArray* names;
NSArray* names10;
int count;
BOOL paused;
```

In the above example snippet, `class-dump-z` has identified a number of methods including “jiggle”, “flyAway” and “drawInContext”; these can all be hooked and modified at runtime.

3.6. Manipulating the Runtime

Hooking the Objective-C runtime is a powerful method of observing and modifying the internal behaviour of an application. The most common method for hooking the runtime is using MobileSubstrate [12], a hooking framework for jailbroken devices, similar to that of Application Enhancer on OS X. MobileSubstrate typically comes as default with many of the iOS jailbreaks and facilitates hooking of not only Objective-C but also C and C++.

Cycript [13] provides a programming language to interface with a JavaScript to Objective-C bridge from the command line. As well as blending JavaScript and Objective-C, Cycript allows runtime hooking using MobileSubstrate. Perhaps one of the most useful features of Cycript is the ability to attach to a running process and manipulate the runtime. For example, cycript can be used to inject into the running SpringBoard process on a jailbroken broken device, disable the passcode requirement and unlock the device, bypassing the passcode:

```
mdsec-iPhone:~/Documents/Cracked root# cycript -p SpringBoard
cy# SBAwayController.messages['isPasswordProtected'] = function() {return NO;}
{}
cy# [SBAwayController.sharedAwayController unlockWithSound:1]
cy#
```

For those looking to write MobileSubstrate extensions, iOSSOpenDev provides a fantastic means of integrating MobileSubstrate into XCode using XCode templates. iOSSOpenDev [14] uses the CaptainHook framework to simplify writing MobileSubstrate tweaks.

Example: Bypassing Jailbreak Detection

For example, consider an app that attempts to detect and prevent the app being run on a jailbroken device. The most common way for this to be accomplished is to check the filesystem for a list of files known to be associated with jailbreaks, for example:

```
@implementation AppSecurity
-(BOOL) isJailBroken
```

```
{
    NSString *filePath = @"/Applications/Cydia.app";
    if ([[NSFileManager defaultManager] fileExistsAtPath:filePath])
    {
        return TRUE;
    }
    return FALSE;
}
```

The above method can be hooked and modified at runtime using a MobileSubstrate tweak such as the following:

```
#import <Foundation/Foundation.h>
#import <CaptainHook/CaptainHook.h>
#include <notify.h>
@interface hookDummy : NSObject
@end
@implementation hookDummy

-(id)init
{
    if ((self = [super init])){}
    return self;
}
@end

@class AppSecurity;
CHDeclareClass(AppSecurity);
CHOptimizedMethod(0, self, BOOL, AppSecurity, isJailBroken)
{
    NSLog(@"##### isJailBroken hooked");
    return false;
}

CHConstructor
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    CHLoadLateClass(AppSecurity);
    CHHook(0, AppSecurity, isJailBroken); // register hook
    [pool drain];
}
```

Once compiled, placing the library in the DynamicLibraries folder causes it to be loaded every time an application is launched on the device:

```
-rwxr-xr-x 1 root wheel 10912 Mar  8 10:15
/Library/MobileSubstrate/DynamicLibraries/hookDummy.dylib*
```

```
Mar  8 21:03:56 unknown DummyApp[1722] <Notice>: MS:Notice: Installing:
MDSec.DummyApp [DummyApp] (675.00)

Mar  8 21:03:56 unknown DummyApp[1722] <Notice>: MS:Notice: Loading:
/Library/MobileSubstrate/DynamicLibraries/Activator.dylib

Mar  8 21:03:56 unknown DummyApp[1722] <Notice>: MS:Notice: Loading:
/Library/MobileSubstrate/DynamicLibraries/hookDummy.dylib

Mar  8 21:03:56 unknown kernel[0] <Debug>: launchd[1722] Builtin profile:
container (sandbox)

Mar  8 21:03:56 unknown kernel[0] <Debug>: launchd[1722] Container:
/private/var/mobile/Applications/1F6A9800-DBD0-4831-A7C9-C4826C6F7EAD [69]
(sandbox)

Mar  8 21:03:57 unknown DummyApp[1722] <Warning>: ##### isJailBroken hooked
```

The library can be configured to only load into specific applications by creating a plist for the library containing the application bundle identifier, similar to:

```
Filter = {
    Bundles = (MDSec.DummyApp);
};
```

Using a real world example, the CommBank Kaching application implements a similar method to detect jailbroken devices; we can identify the relevant methods using class-dump:

```
@interface RootViewController : /private/tmp/KIA_IPHONE_SOURCE/
<UIWebViewDelegate, DILDisplayView, UIAlertViewDelegate>
{
<snip>
- (BOOL)isJailbrokenDevice;
```



When run on a jailbroken device, the app will display an UIAlertView with an exception and does not proceed past the alert.

Figure 3 - Jailbroken error in Kaching

The following MobileSubstrate tweak can be used to bypass this protection and use the app as normal:

```
@class RootViewController;
CHDeclareClass(RootViewController);
CHOptimizedMethod(0, self, BOOL, RootViewController, isJailbrokenDevice)
{
    NSLog(@"##### isJailbrokenDevice hooked");
    return false;
}
CHConstructor
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    CHLoadLateClass(RootViewController);
    CHHook(0, RootViewController, isJailbrokenDevice);
    [pool drain];
}
```

Rerunning the app will cause the app to function as normal due to the isJailBrokenDevice method being hooked and modified:

```
Mar  8 21:15:46 unknown KIA[1786] <Notice>: MS:Notice: Installing:
au.com.commbank.kaching [KIA] (675.00)
Mar  8 21:15:46 unknown KIA[1786] <Notice>: MS:Notice: Loading:
/Library/MobileSubstrate/DynamicLibraries/Activator.dylib
Mar  8 21:15:46 unknown kernel[0] <Debug>: launchd[1786] Builtin profile:
container (sandbox)
Mar  8 21:15:46 unknown kernel[0] <Debug>: launchd[1786] Container:
/private/var/mobile/Applications/63DC8037-5A2F-4C5C-ADDB-30AF3BF49449 [69]
(sandbox)
Mar  8 21:15:47 unknown KIA[1786] <Notice>: MS:Notice: Loading:
/Library/MobileSubstrate/DynamicLibraries/hookDummy.dylib
Mar  8 21:15:47 unknown securityd[1787] <Notice>: MS:Notice: Installing: (null)
[securityd] (675.00)
Mar  8 21:15:47 unknown securityd[1787] <Notice>: MS:Notice: Loading:
/Library/MobileSubstrate/DynamicLibraries/hookDummy.dylib
Mar  8 21:15:47 unknown KIA[1786] <Warning>: **** READ 60 LOG ENTRIES FROM DISK
****
Mar  8 21:15:47 unknown KIA[1786] <Warning>: ##### isJailbrokenDevice hooked
```

3.7. Defending the Binary

Developers looking to mitigate against runtime attacks or increase the complexity of reverse engineering can employ some defensive strategies to thwart attackers. However, it should generally be accepted that there is no full proof method for protecting the app when running on a compromised OS such as in a jailbroken environment.

One of the most common approaches for defending the runtime is to integrity

check classes for expected addresses or checksums, allowing an app to determine if the Objective-C runtime has been hooked or modified. This approach is typically achieved by retrieving and validating the address of the class. The runtime provides the `class_getMethodImplementation` method that returns a function pointer to a class method, if that class method was invoked.

Consider the following simple implementation:

```
#import "security.h"
@implementation security
void * perform_sec_check()
{
    void * addr = verify_address("AppSecurity", "isJailBroken");
    fprintf(stderr, "\ncaddr = %p\n", addr);
    if(addr != 0x25a9) take_evasive_action();
}
void * verify_address(const char * cname, const char * method)
{
    id class = objc_lookUpClass(cname);
    SEL selector = sel_registerName(method);
    IMP imp = class_getMethodImplementation(class, selector);
    return imp;
}
void * take_evasive_action() {
    fprintf(stderr, "%s", "Tamper detected\n");
    exit(-1);
}
@end
```

The above class provides a simple implementation of runtime tamper detection. The `verify_address` function retrieves the address of the function pointer for the `AppSecurity: isJailBroken` method, taken from the earlier example. This address is then compared to the known safe address, hardcoded by the developer. If the address differs, tampering may have occurred and appropriate action is taken.

Running the application without any runtime hooking, the app jail break detection executes as normal:

```
caddr = 0x25a9
2012-04-18 20:51:51.580 DummyApp[595:707] ##### Sorry, you are running on a
jailbroken device
```

The address printed is the expected address for the `AppSecurity: isJailBroken` method. If the application is run again with the `MobileSubstrate` library from the above example present, the address of the `AppSecurity:`

isJailBroken method has changed:

```
caddr = 0x76ec5  
Tamper detected
```

While the above anti-tamper detection can be effective, it can also be trivially be bypassed by hooking the detection or patching the binary. To further improve the detection, the functions can be in-lined which causes the compiler to fully insert the function body whenever the function is called. Consequently, the attacker would need to patch every occurrence of the function each time it is called. This can be achieved simply by using the keyword inline:

```
inline void * perform_sec_check()  
{  
    void * addr = verify_address("AppSecurity", "isJailBroken");  
    fprintf(stderr, "\ncaddr = %p\n", addr);  
    if(addr != 0x25a9) take_evasive_action();  
}
```

3.8. Conclusion

In conclusion, we have reviewed some of the techniques that can be employed during a black box assessment of an iOS application. Indeed, it is possible to gain an in depth understanding of the inner workings of an app, even those protected by the AppStore encryption. Runtime hooking provides a powerful means to interact, asses and modify an application, in particular it allows an evaluator to get inside an app and utilise inner functionality such as APIs that would otherwise need to be reverse engineered to verify functionality.

From a defensive perspective, developers looking to protect their apps from tampering can do so by using checksums or validating the runtime address of classes and methods. The effectiveness of these protections can be further improved by using inline functions. Where possible, developers should look to refactor code to increase the complexity of reverse engineering and reduce the amount of information disclosed on class structure.

4. Auditing Insecure API Usage

Introduction

iOS applications typically leverage a standard set of APIs to interoperate with servers, local resources and other applications. Whilst many of these implement secure defaults, MDSec have audited many applications where the default options are not used, or where an API is simply trusted to operate securely. The following key touch points in an application should be reviewed when performing source code reviews of reviewing iOS applications.

4.1. Evaluating Transport Security

Most iOS applications will perform some network communication and due to the nature of mobile devices this communication may often occur over an untrusted or insecure network such as hotel or café WiFi, mobile hotspot or cellular. Consequently, it is imperative that this communication is performed in a secure manner.

iOS apps will commonly interact with online web applications or web technology based RPC mechanisms; these interactions are often performed using the `NSURLConnection` class. This class takes an `NSURLRequest` object and performs an HTTP(S) request with it. The API uses a default set of SSL ciphers to perform secure connections; unfortunately the API is not granular enough to allow the developer to select which ciphers from the suite to negotiate with. There are some differences between the transports that are negotiated for different versions of the SDK, these are summarised in the table below:

SDK Version	Protocol	"Weak" Cipher Suites	Total Cipher Suites
4.3	TLS 1.0	5	29
5.0	TLS 1.2	0	37
5.1	TLS 1.2	0	37

The table highlights an improvement in the cipher suites negotiated over time with the release of the newer versions of the SDK.

Consider the following example, which will perform a simple HTTPS connection to the localhost:

```
@implementation insecuressl
int main(int argc, const char* argv[])
{
    NSString *myURL=@"https://localhost/test";

    NSURLRequest *theRequest = [NSURLRequest requestWithURL:[NSURL
    URLWithString:myURL]];

    NSURLResponse *resp = nil;
    NSError *err = nil;

    NSData *response = [NSURLConnection sendSynchronousRequest:
    theRequest returningResponse: &resp error: &err];

    NSString * theString = [[NSString alloc] initWithData:response
    encoding:NSUTF8StringEncoding];

    [resp release];
    [err release];
    return 0;
}
```

Compiling the application with both the 5.0 or 5.1 and 4.3 SDKs and then running it while monitoring the communication produces different results.

For version 4.3 of the SDK, the application negotiates a TLS1.0 session with one of 29 cipher suites, as shown in figures 3 and 4:

```
Handshake Type: Client Hello (1)
Length: 135
Version: TLS 1.0 (0x0301)
▷ Random
Session ID Length: 0
Cipher Suites Length: 58
▷ Cipher Suites (29 suites)
Compression Methods Length: 1
Compression Methods (1 method)
Extensions Length: 36
▷ Extension: server_name
▷ Extension: elliptic_curves
▷ Extension: ec point formats
```

Figure 4 - 4.3 SSL Negotiation

```

    Cipher Suites (29 suites)
      Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
      Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
      Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
      Cipher Suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
      Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
      Cipher Suite: TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012)
      Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
      Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
      Cipher Suite: TLS_ECDH_ECDSA_WITH_RC4_128_SHA (0xc002)
      Cipher Suite: TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc003)
      Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
      Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
      Cipher Suite: TLS_ECDH_RSA_WITH_RC4_128_SHA (0xc00c)
      Cipher Suite: TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA (0xc00d)
      Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
      Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
      Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
      Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
      Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
      Cipher Suite: TLS_RSA_WITH_DES_CBC_SHA (0x0009)
      Cipher Suite: TLS_RSA_EXPORT_WITH_RC4_40_MD5 (0x0003)
      Cipher Suite: TLS_RSA_EXPORT_WITH_DES40_CBC_SHA (0x0008)
      Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
      Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
      Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
      Cipher Suite: TLS_DHE_RSA_WITH_DES_CBC_SHA (0x0015)
      Cipher Suite: TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA (0x0014)
    Compression Methods Length: 1
  
```

Figure 5 - 4.3 SSL Ciphers

Using version 5.0 or 5.1 of the SDK, the application negotiates a TLS1.2 session with one of 37 cipher suites, as shown in figures 5 and 6:

```

    TLSv1 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 181
    Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 177
      Version: TLS 1.2 (0x0303)
      Random
      Session ID Length: 0
      Cipher Suites Length: 74
      Cipher Suites (37 suites)
      Compression Methods Length: 1
      Compression Methods (1 method)
      Extensions Length: 62
      Extension: server_name
      Extension: elliptic_curves
      Extension: ec_point_formats
      Extension: signature_algorithms
  
```

Figure 6 – iOS 5.0/5.1 SSL Client Hello

```

▼ Cipher Suites (37 suites)
  Cipher Suite: Unknown (0x00ff)
  Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
  Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
  Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
  Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
  Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
  Cipher Suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)
  Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
  Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
  Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
  Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
  Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
  Cipher Suite: TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012)
  Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384 (0xc026)
  Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 (0xc025)
  Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384 (0xc02a)
  Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256 (0xc029)
  Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
  Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
  Cipher Suite: TLS_ECDH_ECDSA_WITH_RC4_128_SHA (0xc002)
  Cipher Suite: TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc003)
  Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
  Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
  Cipher Suite: TLS_ECDH_RSA_WITH_RC4_128_SHA (0xc00c)
  Cipher Suite: TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA (0xc00d)
  Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)
  Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
  Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
  Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
  Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
  Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
  Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
  Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0x0067)
  Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x006b)
  Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
  Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
  Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
  Compression Methods Length: 1

```

Figure 7 - iOS 5.0/5.1 SDK Cipher Suites

In the above 4.3 SDK negotiation, the following cipher suites can be seen as weak:

- TLS_RSA_WITH_DES_CBC_SHA
- TLS_RSA_EXPORT_WITH_RC4_MD5
- TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
- TLS_DHE_RSA_WITH_DES_CBC_SHA
- TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA

In order to prevent Man-in-the-Middle attacks, it is essential for iOS applications to prohibit the use of self-signed certificates. The default behaviour for the `NSURLRequest` class is to reject self-signed certificates and raise an `NSErrorDomain` exception. However, it is not uncommon to see developers override this behaviour to accept any certificate, frequently to allow the use of self-signed certificates deployed in pre-production environments. The certificate validation can be disabled for the requested domain using the `allowsAnyHTTPTSCertificateForHost` method, similar to that in the following example:

```
#import "loadURL.h"

@interface NSURLRequest (DummyInterface)

+ (BOOL)allowsAnyHTTPTSCertificateForHost:(NSString*)host;
+ (void)setAllowsAnyHTTPTSCertificate:(BOOL)allow forHost:(NSString*)host;

@end

@implementation loadURL

-(void) run
{
    NSURL *myURL = [NSURL URLWithString:@"https://localhost/test"];
    NSMutableURLRequest *theRequest = [NSMutableURLRequest requestWithURL:myURL
    cachePolicy:NSURLRequestReloadIgnoringCacheData timeoutInterval:60.0];

    [NSURLRequest setAllowsAnyHTTPTSCertificate:YES forHost:[myURL host]];
    [[NSURLConnection alloc] initWithRequest:theRequest delegate:self];
}

@end
```

The `allowsAnyHTTPTSCertificateForHost` method is a private method and using it in production code may result in the application being rejected from the App Store. An alternate approach for bypassing SSL verification that is not uncommon is using the `continueWithoutCredentialForAuthenticationChallenge` selector, implemented within the `NSURLConnection` delegate method `didReceiveAuthenticationChallenge`, as shown below:

```
-(void)connection:(NSURLConnection *)connection
didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
{
    if ([challenge.protectionSpace.authenticationMethod
    isEqualToString:NSURLAuthenticationMethodServerTrust])
    {
        [challenge.sender useCredential:[NSURLCredential
        credentialForTrust:challenge.protectionSpace.serverTrust]forAuthenticationChallen
        ge:challenge];

        [challenge.sender
        continueWithoutCredentialForAuthenticationChallenge:challenge];

        return;
    }
}
```

The `CFNetwork` framework provides an alternate API for implementing SSL, indeed the framework allows greater control and customisation of the SSL session for the developer. Similarly to `NSURLRequest`, it is not uncommon to see developers weaken the SSL configuration. `CFNetwork` however provides more granular controls, allowing the application to accept expired certificates or roots, allow any root or even perform no validation on the certificate chain.

Consider the following `onSocket` delegate method, taken from a real-world application:


```
- (void)onSocket:(AsyncSocket *)sock didConnectToHost:(NSString *)host
port:(UInt16)port {
    NSMutableDictionary *settings = [[NSMutableDictionary alloc]
initWithCapacity: 3];
    [settings setObject:[NSNumber numberWithBool:YES]
forKey:(NSString *)kCFStreamSSLAllowsExpiredCertificates];
    [settings setObject:[NSNumber numberWithBool:YES]
forKey:(NSString *)kCFStreamSSLAllowsAnyRoot];
    [settings setObject:[NSNumber numberWithBool:NO]
forKey:(NSString *)kCFStreamSSLValidatesCertificateChain];
    [sock startTLS:settings];
}
```

Unfortunately, when using the CFNetwork framework, there is no clear method of modifying the cipher suite and again, the SDK default set of ciphers is used.

In conclusion, it is imperative for mobile applications to implement transport methods in a secure manner and in the default mode and using the latest SDK, this is likely to be the case when developing an iOS application. However, the APIs do allow the transport security to be weakened and it is not uncommon to see this implemented by developers. Developers looking to temporarily weaken transport security for development or staging environments should be cautious to ensure that this code does not persist in to production. This simplest way to achieve this is to use a pre-processor macro to include the code for development builds only.

4.2. Abusing Protocol Handlers

Due to the restrictions imposed by the iOS sandbox, Inter-Process Communication (IPC) is generally prohibited. However, a simple form of IPC is supported by the API if the application registers a custom protocol handler.

There are many reasons why a developer might want to support IPC ; some examples that we've seen in practice include determining the presence of other apps, allowing the app to be launched from Safari or passing data between apps.

There are two API methods commonly used to implement protocol handlers on iOS, "application:openURL" and "application:handleOpenURL", the latter now deprecated. The advantage of using the "openURL" method is that it supports validation of the source application that instantiated the URL request.

A custom URL scheme can be registered in an iOS application by adding a URL type to the application plist file, as shown in the VulnerableiPhoneApp project below which registers the "vuln" protocol handler:

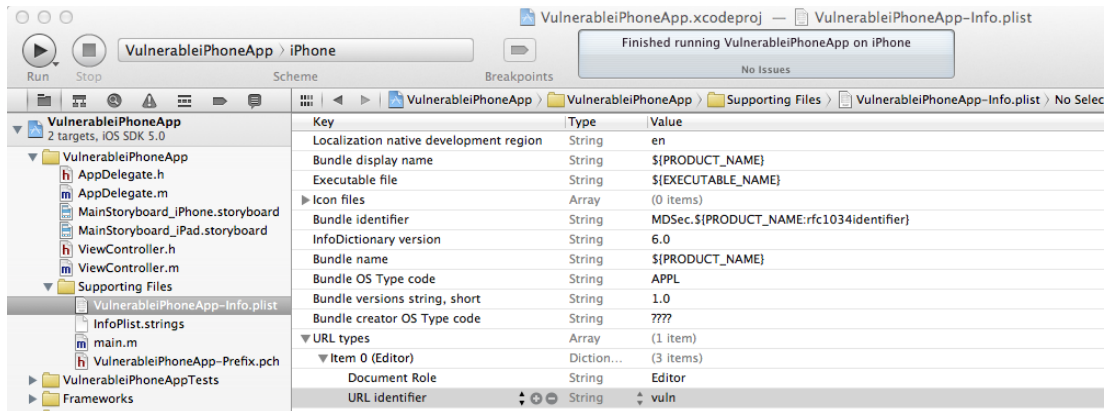


Figure 8 - Registering an IPC in XCode

The protocol handling code can then be implemented using the application delegate methods `handleOpenURL` or `openURL` similar to the following which will simply display an `alertView` with the requested URL text:

```
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url {
    UIAlertView *alertView;
    NSString *text = [[url host]
        stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    alertView = [[UIAlertView alloc] initWithTitle:@"Text" message:text
        delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
    [alertView show];
    return YES;
}
```

An assessment of a real world application found a custom URL handler used to implement configuration changes: a feature initially built in to the application for developer convenience but which had persisted through to production release. Consider the following implementation of the `handleOpenURL` method:

```
- (BOOL)application:(UIApplication *)application handleOpenURL:(NSURL *)url {
    if (!url) { return NO; }
    NSString *method = [[url host]
        stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    if ([method isEqualToString:@"setHomeURL"])
    {
        Settings *s = [[Settings alloc] init];
        NSString *querystr = [[url query]
            stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
        NSArray *param = [querystr componentsSeparatedByString:@"="];
        NSString *value = [param objectAtIndex:1];
        [s setHomeURL:value];
    }
    return YES;
}
```

In this example, the custom URL handler is used to update the default URL that the application opens when it is started. The method accepts an NSURL object which is then parsed; if the host that is passed is "setHomeURL" the method will call the "setHomeURL" method of the Settings object with an argument of the first URL parameter's value.

The setHomeURL method of the Settings object configures the application preferences and is implemented as follows:

```
@implementation Settings
- (void) setHomeURL:(NSString*)url
{
    NSUserDefaults *prefs = [NSUserDefaults standardUserDefaults];
    [prefs setObject:url forKey:@"homeURL"];
    [prefs synchronize];
}
```

An attacker could exploit this issue to reconfigure the default landing page for the application using a malicious iframe, similar to:

```
<iframe src="vuln://setHomeURL?url=http://mdattacker.net"></iframe>
```

A possible solution to this issue is to use the updated API call "openURL" that also provides information on the application from which the URL request originated. The following example will verify that the URL was invoked from within the application itself:

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
sourceApplication:(NSString *)sourceApplication annotation:(id)annotation {
    NSString* myBid = [[NSBundle mainBundle] bundleIdentifier];
    if ([sourceApplication isEqualToString:myBid])
    {
        return NO;
    }
    else if (!url) { return NO; }
    NSString *method = [[url host]
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
    if([method isEqualToString:@"setHomeURL"])
    {
        Settings *s = [[Settings alloc] init];
        NSString *querystr = [[url query]
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
        NSArray *param = [querystr componentsSeparatedByString:@"="];
        NSString *value = [param objectAtIndex:1];
        [s setHomeURL:value];
    }
    return YES;
}
```

Alternatively, if the developer wishes to ensure that the URL can only be invoked from another app, for example Safari, this could be implemented as follows:

```
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
sourceApplication:(NSString *)sourceApplication annotation:(id)annotation {
    NSString *SafariPath = @"/Applications/MobileSafari.app";
    NSBundle *bundle = [NSBundle bundleWithPath:SafariPath];
    if ([sourceApplication isEqualToString:[bundle bundleIdentifier]])
    {
        return No;
    }
}
```

A public real-world vulnerability example could be found in the Skype iOS application that registers the "skype" protocol handler which could be used to instantiate calls and chats. An attack to perform a call without authorization using a malicious iframe was first documented by Nitesh Dhanjani [15]. The attack payload could be triggered from MobileSafari to launch the Skype app, which would perform the call as shown below:

```
<iframe src="skype://123456789?call"></iframe>
```

Skype resolved this issue by displaying a `UIView` that allows the user to accept or decline the call.

A simple method of identifying valid URLs in AppStore apps is to take the decrypted app and check for protocol strings, an example using the Facebook application (truncated from 558 URLs):

```
bash-3.2# strings Facebook.app/Facebook | grep "://" | grep -v "http"
fb://upload/actions/newalbum
fb://root
fb://birthdays
fb://messaging
fb://notifications
fb://requests
fb://publish
fb://publish/profile/(gatePublishWithUID:)
fb://oldpublish
fb://oldpublish/profile/(initWithUID:)
fb://publish/post/(initWithPostId:)
fb://publish/photo/(initWithUID:)/(aid:)/(pid:)
fb://publish/mailbox/(initWithFolder:)/(tid:)
fb://publish/privacy
fb://place/create
fb://compose
fb://compose/profile/(initWithUID:)
```

In conclusion, protocol handlers can provide a convenient method for developers to perform inter-process communication. However, developers should be careful to perform validation on the source and content of any data entering the application and avoid using protocol handlers to access sensitive or dangerous functionality.

4.3. Locating Insecure Data Storage

The protection of data stored on a mobile device is perhaps one of the most important issues that an application developer has to deal with. It is imperative that developers protect sensitive data that is stored client-side in a secure manner. As previously noted, developers wishing to encrypt sensitive content on the device should employ the Data Protection API. Unfortunately, it is common practice to find even apps from large multinationals storing their sensitive data in clear text. A good example of this was highlighted in 2010 where vulnerabilities in the Citigroup online banking application caused it to be pulled from the AppStore, as reported by The Register:

"In a letter, the US banking giant said the Citi Mobile app saved user information in a hidden file that could be used by attackers to gain unauthorized access to online accounts. Personal information stored in the file could include account numbers, bill payments and security access codes..."

[16].

While this paper will only focus on app data storage and how applications can use the Data Protection API, an in depth presentation on iPhone encryption has been performed by Jean-Baptiste Bedrune and Jean Sigwal of ESEC [17].

Client-side data can be stored in a number of forms, including but not limited to:

- Custom created files,
- Databases,
- System logs,
- Cookie stores,
- Plists,
- Data caches.

All of these may contain sensitive data that should be protected if the handset were lost or stolen. This data will generally be stored within the application's sandboxed container.

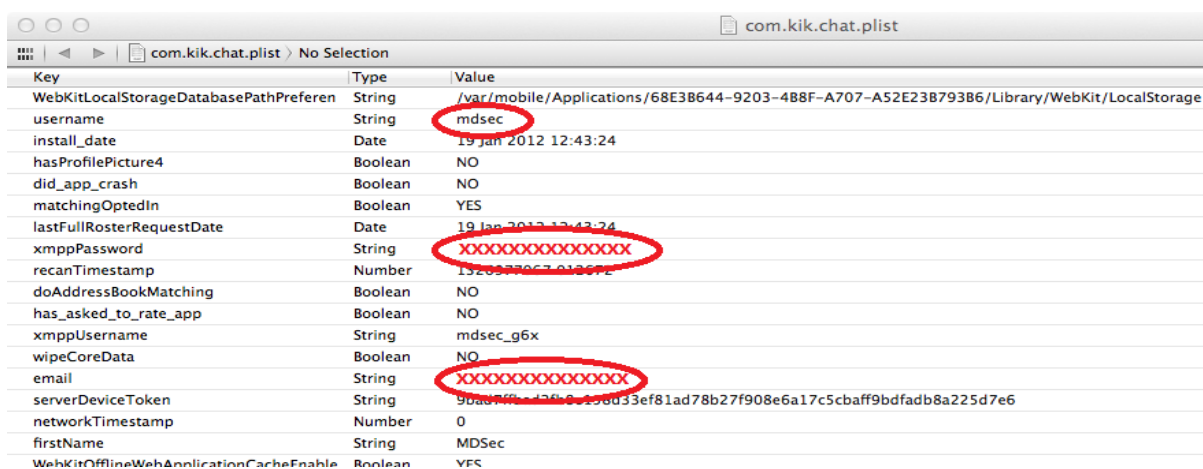
Applications are stored on the file-system as the "mobile" user under the "/var/mobile/Applications" directory where a unique GUID is used as a sub directory container to store the app data. The application directory structure is as follows:

Directory	Description
Application.app	Stores the static content of the application and compiled app. This content is signed and checked at runtime.
Documents	A persistent store for application data; this data will be synched and backed up to iTunes.
Library	This folder contains support data used by the app such as configurations, preferences, cache data and cookies.
tmp	This folder is used to store temporary files.

An attacker looking to extract application data is likely to find it within this directory structure in one form or another. However, such exploration of the filesystem will first require a jailbroken device.

Let's take a look at a real world app, taken from the AppStore. Kik Messenger is a social networking application with a 4+ star rating from 6405 ratings on the AppStore and well over 1 million users. The application allows users to send free instant messages via the devices data connection. In order to do this, the user must sign-up for a free Kik account.

Within the Kik application directory is the preferences plist, "Library/Preferences/com.kik.chat.plist" that is used by the app to store configuration information, including the user's username, password and e-mail address, as shown below (obscured for reader):

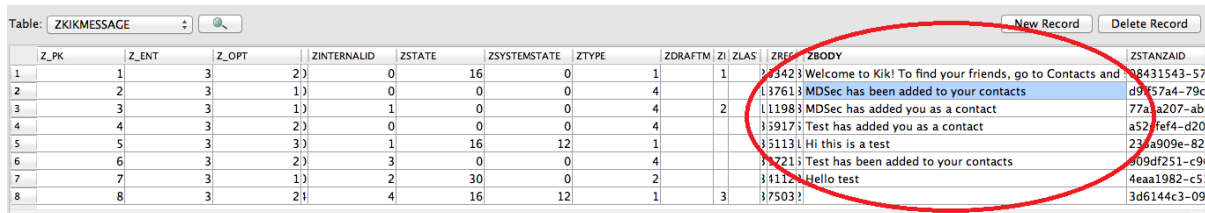


Key	Type	Value
WebKitLocalStorageDatabasePathPreferen	String	/var/mobile/Applications/68E3B644-9203-488F-A707-A52E23B793B6/Library/WebKit/LocalStorage
username	String	mdsec
install_date	Date	19 Jan 2012 12:43:24
hasProfilePicture4	Boolean	NO
did_app_crash	Boolean	NO
matchingOptedIn	Boolean	YES
lastFullRosterRequestDate	Date	19 Jan 2012 12:43:24
xmppPassword	String	XXXXXXXXXXXXXXXX
recanTimestamp	Number	132637863.018672
doAddressBookMatching	Boolean	NO
has_asked_to_rate_app	Boolean	NO
xmppUsername	String	mdsec_g6x
wipeCoreData	Boolean	NO
email	String	XXXXXXXXXXXXXXXX
serverDeviceToken	String	9ba37ff-4268-415dd33ef81ad78b27f908e6a17c5cbaff9bdfadb8a225d7e6
networkTimestamp	Number	0
firstName	String	MDSec
WebKitOfflineWebApplicationCacheEnable	Boolean	YES

Figure 9 - Kik Messenger plist

The plist detailed above is not protected by the Data Protection API and therefore resides unencrypted on the file-system while the device is enabled, regardless of lock state. This is a classic example of misuse of data storage as sensitive information such as credentials should be stored in the keychain rather than on the filesystem as a plist.

In addition to the above, Kik stores other information on the device, including the SMS chat history and contact information. This data is stored in a sqlite database in "Documents/kik.sqlite" which again is not encrypted:



Z_PK	Z_ENT	Z_OPT	ZINTERNALID	ZSTATE	ZSYSTEMSTATE	ZTYPE	ZDRAFTM	ZI	ZLAS	ZREF	ZBODY	ZSTANZAID
1	1	3	2	0	16	0	1	1	1	1342	Welcome to Kik! To find your friends, go to Contacts and	8431543-57
2	2	3	1	0	0	0	4			13761	MDSec has been added to your contacts	d9f57a4-79c
3	3	3	1	1	0	0	4	2		11198	MDSec has added you as a contact	77a207-ab
4	4	3	2	0	0	0	4			33917	Test has added you as a contact	a52fef4-d20
5	5	3	3	1	16	12	1			3113	Hi this is a test	23a909e-82
6	6	3	2	3	0	0	4			3721	Test has been added to your contacts	309df251-c9
7	7	3	1	2	30	0	2			34112	Hello test	4eaa1982-c5
8	8	3	2	4	16	12	1	3		37503		3d6144c3-09

Figure 10 - Kik SQLite Database

A common dilemma and one faced by the Kik application is that it is a real-time app that receives messages while backgrounded and regardless of lock state. If the app was to apply `NSFileProtectionComplete`, it would not be able to access the SQLite store when the phone is locked. Partial mitigation might be achieved by encrypting the data until the first phone unlock by setting the `NSFileProtectionCompleteUntilFirstUserAuthentication` constant. Subsequent reboots would cause the data to be encrypted, however this feature is only available from iOS 5.

The Kik app also allows users to send attachments such as photos within IMs, these are stored unencrypted in the "Documents/fileAttachments" directory. For example, the following shows a photo sent via an IM attachment:

```
mbp:Documents $ file fileAttachments/057a8fc9-0daf-4750-b356-5b28755f4ec4
fileAttachments/057a8fc9-0daf-4750-b356-5b28755f4ec4: JPEG image data, JFIF
standard 1.01 mbp:Documents $
```

It is worth considering that iOS itself does not apply data protection to photos stored on the device; however it is a risk that the application could potentially avoid.

The Data Protection API allows four levels of file-system protection that are configurable by passing an extended attribute to the `NSData` or `NSFileManager` classes. The possible levels of protection are:

Level	Description
No Protection	The file is not encrypted on the file-system.
Complete Protection	The file is encrypted on the file-system and inaccessible when the device is locked.

Complete Unless Open	The file is encrypted on the file-system and inaccessible while closed. When a device is unlocked an app can maintain an open handle to the file even after it is subsequently locked, however during this time the file will not be encrypted.
Complete Until First User Authentication	The file is encrypted on the file-system and inaccessible until the device is unlocked for the first time. This helps offer some protection against attacks that require a device reboot.

In order to apply one of the above levels of protection, one of the following extended attributes must be passed to the relevant class:

NSData	NSFileManager
NSDataWritingFileProtectionNone	NSFileProtectionNone
NSDataWritingFileProtectionComplete	NSFileProtectionComplete
NSDataWritingFileProtectionCompleteUnlessOpen	NSFileProtectionCompleteUnlessOpen
NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication	NSFileProtectionCompleteUntilFirstUserAuthentication

For example, consider an application that needs to save some data to the file-system, but it does not require access to the file while the device is locked, such as an app that allows you to download documents and then later view them. As the app does not require access to the files when the device is locked, it can take advantage of the complete protection by setting the `NSDataWritingFileProtectionComplete` or `NSFileProtectionComplete` attributes:

```

-(BOOL) getFile
{
    NSString *fileURL = @"http://www.mdsec.co.uk/training/wahh-live.pdf";
    NSURL *url = [NSURL URLWithString:fileURL];
    NSData *urlData = [NSData dataWithContentsOfURL:url];
    if ( urlData )
    {
        NSArray *paths =
        [searchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES)];
        NSString *documentsDirectory = [paths objectAtIndex:0];
        NSString *filePath = [NSString stringWithFormat:@"%@/%@",
        documentsDirectory,@"wahh-live.pdf"];
        NSError *error = nil;

        [urlData writeToURL:filePath options:NSDataWritingFileProtectionComplete
        error:&error];
    }
}

```



```

    return YES;
}

return NO;
}

```

In this scenario, the document will only be accessible while the device is unlocked. The OS provides a 10 second window between locking the device and this file being unavailable. The following shows an attempt to access the file while the device is locked:

```

mdsec-iPhone:/var/mobile/Applications/7F5ED565-781E-47FD-8787-4C76CD7A4DD5 root#
ls -al Documents/ total 372
drwxr-xr-x  2 mobile mobile   102 Jan 20 15:24 ./
drwxr-xr-x  6 mobile mobile   204 Jan 20 15:23 ../
-rw-r--r--  1 mobile mobile 379851 Jan 20 15:24 wahn-live.pdf
mdsec-iPhone:/var/mobile/Applications/7F5ED565-781E-47FD-8787-4C76CD7A4DD5 root#
strings Documents/wahn-live.pdf
strings: can't open file: Documents/wahn-live.pdf (Operation not permitted)
mdsec-iPhone:/var/mobile/Applications/7F5ED565-781E-47FD-8787-4C76CD7A4DD5 root#

```

Developers wishing to apply the relevant protection levels to data stored on the device can achieve this in a similar manner to the above by passing the relevant attribute that best fits the developer's requirement for file access.

In conclusion, iOS leaves data protection very much in the developer's hands, providing granular controls to configure the level of protection that can be applied to data written to the filesystem. Unfortunately, it is common to find that developers do not take advantage of this protection and leave sensitive data at risk of compromise.

4.4. Attacking the iOS Keychain

The iOS keychain is an encrypted container used for storing sensitive data such as credentials whilst restricting apps to accessing only their own keychain items unless they are a member of a keychain access group. Similar to files on the filesystem, a protection level can be applied using the Data Protection API. The following table describes the available protection levels for keychain items:

Attribute	Description
kSecAttrAccessibleAlways	The keychain item is always accessible.
kSecAttrAccessibleWhenUnlocked	The keychain item is only accessible when the device is unlocked.

kSecAttrAccessibleAfterFirstUnlock	They keychain item is only accessible after the first unlock from boot. This helps offer some protection against attacks that require a device reboot.
kSecAttrAccessibleAlwaysThisDeviceOnly	The keychain item is always accessible but cannot be migrated to other devices.
kSecAttrAccessibleWhenUnlockedThisDeviceOnly	The keychain item is only accessible when the device is unlocked and may not be migrated to other devices.
kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly	The keychain item is accessible after the first unlock from boot and may not be migrated to other devices.

Keychain items can be added using the `SecItemAdd` or updated using the `SecItemUpdate` methods, which accept one of the above attributes to define the protection level to apply. By default all keychain items are created with a protection level of `kSecAttrAccessibleAlways` which will allow access at any time and allows migration to other devices.

Applications' access to keychain items is limited by the entitlements they are granted. The keychain uses application identifiers stored in the "keychain-access-group" entitlement of the provisioning profile for the app; a sample provisioning profile that allows keychain access only to the app's keychain is shown overleaf:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>application-identifier</key>
  <string>my.company.VulnerableiPhoneApp</string>
  <key>get-task-allow</key>
  <true/>
  <key>keychain-access-group</key>
  <array>
    <string>my.company.VulnerableiPhoneApp</string>
  </array>
</dict>
</plist>
```

As previously noted, an app can add an item to the keychain using the `SecItemAdd` method; consider the following example app that wishes to store

a license key in the keychain and only requires access to the item when the device is unlocked:

```
- (NSMutableDictionary *)getkeychainDict:(NSString *)service {
    return [NSMutableDictionary dictionaryWithObjectsAndKeys:
        (id)kSecClassGenericPassword, (id)kSecClass,
        service, (id)kSecAttrService, service, (id)kSecAttrAccount,
        (id)kSecAttrAccessibleWhenUnlocked, (id)kSecAttrAccessible, nil];
}

- (BOOL) saveLicense:(NSString*)licenseKey {
    static NSString *serviceName = @"my.company.VulnerableiPhoneApp";
    NSMutableDictionary *myDict = [self getkeychainDict:serviceName];
    SecItemDelete((CFDictionaryRef)myDict);
    NSData *licenseData = [licenseKey dataUsingEncoding:NSUTF8StringEncoding];
    [myDict setObject:[NSKeyedArchiver archivedDataWithRootObject:licenseData]
    forKey:(id)kSecValueData];
    OSStatus status = SecItemAdd((CFDictionaryRef)myDict, NULL);
    if (status == errSecSuccess) return YES;
    return NO;
}
```

Firstly, the app creates a dictionary of key-value pairs which are the configuration attributes for the keychain. In this instance the app sets the `kSecAttrAccessibleWhenUnlocked` attribute to allow access to the keychain item whenever the device is unlocked. The app then sets the `kSecValueData` attribute to the value of the data that it wishes to store in the keychain, in this instance the license key data, and adds the item to the keychain using the `SecItemAdd` method.

Under the hood, the keychain is simply a SQLite database and can be queried like any other database. For example, to find out the list of the keychain groups the following query can be executed:

```
mdsec-iPhone:/var/Keychains root# sqlite3 keychain-2.db "select agrp from genp"
apple
apple
apple
ichat
com.apple.apsd
apple
apple
T84QZS65DQ.platformFamily
T84QZS65DQ.platformFamily
apple
apple
```

```
my.company.VulnerableiPhoneApp
mdsec-iPhone:/var/Keychains root#
```

On a jailbroken phone, it is possible to dump all the keychain items for any application under the same caveats previously detailed with the Data Protection API. This is achieved by creating an app that is assigned to all the relevant keychain-access-groups and querying the keychain service to retrieve the protected items [18].

4.5. Conducting Cross-Site Scripting (XSS) through UIWebViews

UIWebView is the iOS rendering engine for displaying text; it supports a number of different file formats, including:

- HTML
- PDF
- RTF
- Office Documents (doc, xls, ppt)
- iWork Documents (Pages, Numbers and Keynote)

The web view is built upon WebKit and uses the same core frameworks as Safari and MobileSafari. Consequently, a web view is also a web browser and can be used to fetch and display remote content. As would be expected of a web browser, web views also support JavaScript, allowing apps to perform dynamic, client-side scripting; however there is no configurable option to disable this feature within the API. Consequently, just like a traditional web application, iOS apps can be affected by Cross-Site Scripting (XSS).

Cross-Site Scripting in iOS apps can often be much more severe than traditional XSS attacks such as session theft, as developers commonly expose native iOS functionality by implementing a JavaScript to Objective-C bridge; some examples MDSec have witnessed in practice include taking a photo, accessing geolocation and sending SMS/E-Mails from JavaScript. Cross-Site Scripting can occur in an iOS in any scenario where user supplied input is blindly populated in to a UIWebView without sanitisation. Often this can happen when a developer needs to use a user controlled objective C variable in a web view. The Skype iOS application was affected by such a vulnerability when displaying a user's "Full Name" for an incoming call. The Skype app used a local HTML file as a template for a UIWebView without sanitising the user's "Full Name" from the incoming call. In this instance the attacker could access the local file system due to the file being loaded in a local context; a proof of concept exploit was developed to retrieve and upload the device's address book [19].

Consider the following simple example where a username from an objective C variable is added to the DOM of the UIWebView:

```
NSString *javascript = [[NSString alloc] initWithFormat:@"var myvar=\"%@\";",
username];
```

```
[mywebView stringByEvaluatingJavaScriptFromString:javascript];

[mywebView loadRequest:[NSURLRequest requestWithURL:[NSURL
fileURLWithPath:[NSBundle mainBundle] pathForResource:@"index"
ofType:@"html"]isDirectory:NO]]];
```

Firstly, the username is added to an NSString object that represents the JavaScript, this is then added to the DOM of the web view using the "stringByEvaluatingJavaScriptFromString" method. Whilst there is also Cross-Site Scripting at this point as the JavaScript is directly evaluated by the UIWebView, the variable is also populated into the local HTML file stored in the bundle directory:

```
<html>
  <p>
    Cross-Site Scripting in UIWebView:
  </p>
  <p>
    This is an example of XSS:
    <script>document.write(myvar);</script>
  </p>
</html>
```

Much like the traditional Cross-Site Scripting attacks, the key to prevention is strictly sanitising all data on arrival into the iOS application and ensuring that data is suitably encoded when presenting it in the UIWebView.

4.6. Attacking XML Processors

XML is widely used in mobile application deployments to represent data and the iPhone SDK provides two options for parsing XML, the NSXMLParser and libxml2. However, there are also a number of popular third party XML parser implementations.

A common attack often associated with XML parsers is the "billion laughs" [20] attack in which the parser is supplied with a number of nested entities which if expanded can cause a Denial of Service. The default parsers included with the iOS SDK are not vulnerable to this attack; when a nested entity is detected the NSXMLParser will raise an NSXMLParserEntityRefLoopError exception, while the libxml2 parser will throw an error stating "Detected an entity reference loop".

Another common attack scenario with XML parsers is the parsing of external XML entities. While parsing of external XML entities is not enabled by default on the NSXMLParser, it is enabled by default if the developer uses the alternate LibXML2 parser. To enable the parsing of external entities with NSXMLParser, the developer must set the setShouldResolveExternalEntities option which causes the delegate method foundExternalEntityDeclarationWithName to be invoked when an

entity is found.

A simple, vulnerable NSXMLParser implementation might look something like the following:

```
#import "XMLParser.h"

@implementation XMLParser

- (void)parseXMLStr:(NSString *)xmlStr {
    BOOL success;

    NSData *xmlData = [xmlStr dataUsingEncoding:NSUTF8StringEncoding];
    NSXMLParser *addressParser = [[NSXMLParser alloc] initWithData:xmlData];
    [addressParser setDelegate:self];
    [addressParser setShouldResolveExternalEntities:YES];
    success = [addressParser parse];
}

- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
attributes:(NSDictionary *)attributeDict {}

- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string {}
- (void)parser:foundExternalEntityDeclarationWithName:publicID:systemID {}

- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError *)parseError{
    NSLog(@"Error %i, Description: %@", [parseError code],
        [[parser parseError] localizedDescription]); }

@end
```

The developer has enabled the parsing of external entities by setting the parser option "setShouldResolveExternalEntities". When the implementation is called, the parser will attempt to resolve the entity:

```
NSString *xmlStr = @"<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n
    <!DOCTYPE foo [ \n
        <!ELEMENT foo ANY > \n
        <!ENTITY xxe SYSTEM \"http://192.168.0.7/hello\"> \n
    ]> \n
    <foo>&xxe;</foo>";

XMLParser *xp = [[XMLParser alloc] init];
[xp parseXMLStr:xmlStr];
```

When the parser attempts to resolve the entity, it will force a HTTP request from the device to the web server:

```
bash-3.2# nc -lvp 80
listening on [any] 80 ...
```

```
192.168.0.2: inverse host lookup failed: Unknown host connect to [192.168.0.7]
from (UNKNOWN) [192.168.0.2] 49287 GET /hello HTTP/1.0

Host: 192.168.0.7

Accept-Encoding: gzip
```

An implementation of the same attack vector using the libxml2 parser might look like the following:

```
#import <libxml/xmlmemory.h>

@implementation LibXml2

-(BOOL) parser:(NSString *)xml {

    xmlDocPtr doc = xmlParseMemory([xml UTF8String], [xml
lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);

    xmlNodePtr root = xmlDocGetRootElement(doc);

}

@end
```

In this example, the external entity is parsed when the “xmlParseMemory” method is called on the XML string and will result in an outbound HTTP connection from the device. In other circumstances, developers should be aware that it may also be possible to open local files using the “file:///” protocol handler, under the constraints of the sandbox.

4.7. SQL Injection

iOS apps will typically need to store some application data client-side; one of the simplest ways to achieve this is to use a SQLite data store. Much like when SQL is used within web applications, if the statement is not formed correctly it can lead to SQL injection. While in most circumstances this has little impact as the data store is client-side, it is an exploitable condition if untrusted data supplied by a malicious user is retrieved from the server. To perform data access on client-side SQLite databases, iOS provides the built-in SQLite data library. If using SQLite, the application will be linked to the “libsqlite3.dylib” library.

Similarly to traditional web applications, iOS app SQL injection occurs when un-sanitised user input is used to construct a dynamic SQL statement. In order to compile a SQL statement, the statement must first be defined as a constant character array and passed to one of the SQLite prepare methods.

Consider the following example of a social networking application that reads multiple users’ status messages and stores the results for offline viewing in a SQLite database. The application reads from multiple user feeds and renders a link to the user’s profile and their display name in the app. The following code example shows a dynamically created SQLite statement that is executed when the user’s message feed is read:

```
sqlite3 *database;
sqlite3_stmt *statement;
if(sqlite3_open([databasePath UTF8String], &database) == SQLITE_OK)
{
    NSString *sql = [NSString stringWithFormat:@"INSERT INTO messages VALUES ('1',
    '%@','%@','%@')", msg, user, displayname];
    const char *insert_stmt = [sql UTF8String];
    sqlite3_prepare_v2(database, insert_stmt, -1, &statement, NULL);
    if (sqlite3_step(statement) == SQLITE_DONE)
```

In the above code excerpt, the developer first opens the SQLite database, stored in the “databasePath” variable. If the database was successfully opened, an NSString object is initialised to create a dynamic SQL statement using the unsanitised, attacker-controlled “msg”, “user” and “displayname” variables. The SQL query is then converted to a constant character array and compiled as a SQL statement using the “sqlite3_prepare_v2” method. Finally, the SQL statement is executed using the “sqlite3_step” method.

As the parameters that are used to construct the statement originate from the user, the resultant statement can be user controlled. For example, consider a malicious user setting a status message as follows:

```
Check out my cool site http://mdattacker.net', 'Goodguy', 'Good guy');/*
```

This would result in the following SQL query being executed:

```
INSERT INTO messages VALUES('1', 'Check out my cool site http://mdattacker.net',
'Goodguy', 'Good guy');/*','originaluser','Original User');
```

Consequently, the attacker is able to control the subsequent fields in the query and make the message appear as though it originated from another user.

The resolution is similar to SQL injection prevention in traditional applications; the query structure should be defined using bind variables and parameterised queries. SQLite provides the sqlite3_bind_text function for binding text values to prepared statements. The previous example can be resolved as follows:

```
const char *insert_stmt = "INSERT INTO messages VALUES('1', ?, ?, ?)";
sqlite3_prepare_v2(database, insert_stmt, -1, &statement, NULL);
sqlite3_bind_text(&insert_stmt, 1, [msg UTF8String], -1, SQLITE_TRANSIENT);
sqlite3_bind_text(&insert_stmt, 2, [user UTF8String], -1, SQLITE_TRANSIENT);
sqlite3_bind_text(&insert_stmt, 3, [displayname UTF8String], -1,
SQLITE_TRANSIENT);
if (sqlite3_step(statement) == SQLITE_DONE)
```

Using the parameterised query, the msg variable will be bound in to the bind variable in the compiled statement and cannot be escaped.

4.8. Filesystem Interaction

Filesystem interaction in iOS can be achieved using the `NSFileManager` or `NSFileHandle` classes. While the `NSFileManager` class is explicitly used for the filesystem, `NSFileHandle` also allows access to sockets, pipes and devices.

The `NSFileManager` class offers robust file system interaction with a number of instance methods to perform file operations, including:

Instance Methods	Description
<code>fileExistsAtPath</code>	Determines if a file exists.
<code>contentsEqualAtPath</code>	Compares the contents of two files.
<code>isReadableFileAtPath</code> , <code>isWritableFileAtPath</code> , <code>isExecutableFileAtPath</code> , <code>isDeletableFileAtPath</code>	Determines if a file is readable, writeable, executable or deletable.
<code>moveItemAtPath</code>	Renames the specified file.
<code>copyItemAtPath</code>	Copies a file to the specified destination.
<code>removeItemAtPath</code>	Deletes the specified file.
<code>createSymbolicLinkAtPath</code>	Creates a symbolic link to the specified file.

The `NSFileHandle` class provides a more advanced means of interacting with a file descriptor. This class is closer to the traditional C file operations and provides a means to seek to offsets within the file and leaves the responsibility of closing the handle to the developer.

Both the `NSFileManager` and `NSFileHandle` classes can be affected by directory traversal issues in a scenario where by the attacker can control part of the filename.

Consider the following implementation to read a file's contents using both classes:

```
- (NSData*) readContents:(NSString*) location
{
    NSFileManager *filemgr;
    NSData *buffer;

    filemgr = [NSFileManager defaultManager];

    buffer = [filemgr contentsAtPath:location];

    return buffer;
}
```

```
- (NSData*) readContentsFH:(NSString*)location
{
    NSFileHandle *file;
    NSData *buffer;

    file = [NSFileHandle fileHandleForReadingAtPath:location];
    buffer = [file readDataToEndOfFile];
    [file closeFile];
    return buffer;
}
```

In the above methods, the developer has made no attempt to sanitise the location string prior to opening the file, leading to a directory traversal vulnerability using traditional traversal strings such as `"../../"`:

```
NSString *fname = @"../Documents/secret.txt";

NSString *sourcePath = [[NSString alloc] initWithFormat:@"%s/%s", [[NSBundle mainBundle] resourcePath], fname];

NSLog(@"##### PATH = %s", sourcePath);

NSString *contents = [[NSString alloc] initWithData:[fm readContentsFH:sourcePath] encoding:NSUTF8StringEncoding];

NSLog(@"##### File contents: %s", contents);
```

In the above example, the `fname` variable originates from a user controlled string allowing the attack to traverse outside of the resource bundle directory and in to the Documents directory:

```
2012-02-11 15:58:18.029 VulnerableiPhoneApp[3291:707] ##### PATH =
/var/mobile/Applications/E84D97BB-79E7-4603-93D3-
09A88CB4FA71/VulnerableiPhoneApp.app/ ../Documents/secret.txt

2012-02-11 15:58:18.040 VulnerableiPhoneApp[3291:707] ##### File contents:
Password=abc123
```

Developers should also be aware of the risks of mixing Objective-C and C, in particular when performing file operations. Objective C does not use null bytes to terminate a string in an `NSString` object. If a developer uses an `NSString` object with a user controlled file path and later file operations are performed in C, the attacker may be able to terminate the string early. For example, consider the following:

```
NSString *fname = @"../Documents/secret.txt\0";

NSString *sourcePath = [[NSString alloc] initWithFormat:@"%s/%s.jpg", [[NSBundle mainBundle] resourcePath], fname];

char line[1024];

FILE *fp = fopen([sourcePath UTF8String], "r");

fread(line, sizeof(line), 1024, fp);

NSString *contents = [[NSString alloc] initWithCString:line];

fclose(fp);
```

In the above example the developer expects the string to provide a location to

a JPG file, and attempts to restrict the extension by manually defining it within the initialisation of the `sourcePath` variable. However, the null byte in the `fname` variable causes the string to be terminated early when converted to a C string, allowing the attacker to open any file type.

4.9. Geo-Location

Apple provides a means of accessing the device's geo-location features using the Core Location framework. Device coordinates can be determined using GPS, cell tower triangulation or WiFi network proximity. When using geo-location data, there are two main privacy concerns that developers should consider: how and where data is logged and the requested accuracy of coordinates.

Core Location is event driven and an app looking to receive location information must register to receive event updates. Event updates can provide longitude and latitude coordinates for use in the app. As previously mentioned, an important consideration when reviewing an app is to evaluate how this coordinate data is stored. If the app must store coordinate information client-side, the developer should protect this data using one of the previously detailed methods. However, to avoid the app being used to track a user's movements, it is generally recommended that location information is not stored on the device. In addition to client-side logging, if the app passes coordinate information to a server, developers should ensure that if this information is logged, it is done so anonymously.

Another consideration for developers when requesting event updates is the accuracy of the information they require. For example, if the app is used for satellite navigation, then it is likely to require very accurate location information. Whereas, an app that provides information about the closest restaurant does not need to be as accurate. Similarly to location logging, the accuracy of the coordinates raises privacy concerns and should be considered by developers when writing iOS applications.

When using `CLLocationManager`, an app can request accuracy using the `CLLocationAccuracy` class that offers the following constants:

- `kCLLocationAccuracyBestForNavigation`
- `kCLLocationAccuracyBest`
- `kCLLocationAccuracyNearestTenMeters`
- `kCLLocationAccuracyHundredMeters`
- `kCLLocationAccuracyKilometer`
- `kCLLocationAccuracyThreeKilometers`

4.10. Logging

Logging can prove a valuable resource for debugging during development, however in some cases it can leak sensitive or proprietary information, which is then cached on the device until the next reboot. Logging in objective C is typically performed using the `NSLog` method that causes a message to be sent to the Apple System Log. These console logs are not only accessible using the Xcode organiser application but by any app installed on the device, using the ASL library.

In some cases jailbreaking a device will cause `NSLog` output to be redirected to `syslog`. In this scenario, it is possible that sensitive information may be stored on the file system in `syslog`. As such, best practice recommends that developers avoid using `NSLog` to log sensitive or proprietary information.

The simplest way for developers to avoid compiling `NSLog` into production releases is to redefine it with a dummy pre-processor macro such as `"#define NSLog(...)"`.

4.11. Backgrounding

If an application is open, it is possible that it can be sent in to the background by a change in state, such as the user pressing the Home button or from an incoming call. When an application is suspended in the background, iOS will take a "snapshot" of the app and store it in the application caches directory. When the application is reopened, the device will use the screenshot to create the appearance that the application loads instantly rather than the small amount of time it actually takes to reload the application and for it to become useable again.

If any sensitive information is open in the application when it enters the background, the snapshot is written to the filesystem in clear text. However, the `UIApplication` delegate method `applicationDidEnterBackground` can be used to detect when an application is entering the background and modify the display accordingly. For example, if there are specific fields that contain sensitive information, the application can hide these using the "hidden" attribute:

```
- (void)applicationDidEnterBackground:(UIApplication *)application {  
    viewController.creditcardNumber.hidden = YES;  
}
```

Conversely, when the application restarts, it can unhide these by doing the reverse in the `applicationDidBecomeActive` delegate:

```
- (void)applicationDidBecomeActive:(UIApplication *)application {  
    viewController.creditcardNumber.hidden = NO;  
}
```

5. Memory Corruption Issues

Introduction

iOS apps are typically resilient to classic memory corruption issues such as buffer overflows if the developers rely on Objective-C to perform memory allocations as the developer cannot specify fixed sizes for buffers. However, as previously mentioned, C can be intermingled in to iOS apps; it is not uncommon to see the use of external libraries or performance dependent code such as cryptography developed in C. These scenarios give rise to the traditional memory corruption vulnerabilities. There is however a small number of memory corruption issues that have transcended into Objective-C and are detailed below.

5.1. Format Strings

Format String vulnerabilities are a class of memory corruption bug that arise through improper use of Objective-C methods that accept a format specifier. Vulnerable Objective-C methods include the following:

- NSLog
- [NSString stringWithFormat]
- [NSString stringByAppendingFormat]
- [NSString initWithFormat]
- [NSMutableString appendFormat]
- [UIAlertView alertWithMessageText]
- [UIAlertView informativeTextWithFormat]
- [NSException format]
- [NSMutableString appendFormat]
- [NSPredicate predicateWithFormat]

Format string vulnerabilities arise when an attacker is able to provide the format specifier in part or as a whole to the relevant method. For example, consider the following:

```
NSString *myURL=@"http://10.0.2.1/test";

NSURLRequest *theRequest = [NSURLRequest requestWithURL:[NSURL
URLWithString:myURL]];

NSURLResponse *resp = nil;
NSError *err = nil;

NSData *response = [NSURLConnection sendSynchronousRequest: theRequest
returningResponse:&resp error: &err];

NSString * theString = [[NSString alloc] initWithData:response
encoding:NSUTF8StringEncoding];

NSLog(theString);
```

In this example a request is made to a web server running on 10.0.2.1, the response is then stored in a NSData object, converted to a NSString and logged using NSLog. The documented usage of the NSLog function where

NSLog is a wrapper for NSLogv and args is a variable number of arguments is:

```
void NSLogv (
    NSString *format,
    va_list args
);
```

However, in this instance the developer has supplied only a single argument, allowing the attacker to specify the type of parameter that would be logged.

Running the above example in a debugger, we can see how the format string vulnerability can be triggered using a simple HTTP web server response:

```
bash-3.2# nc -lvp 80
listening on [any] 80 ...
10.0.2.2: inverse host lookup failed: Unknown host
connect to [10.0.2.1] from (UNKNOWN) [10.0.2.2] 52141
GET /test HTTP/1.1
Host: 10.0.2.1
User-Agent: fmtstrtest (unknown version) CFNetwork/548.0.4 Darwin/11.0.0
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive

HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 16

aaaa%x%x%x%x%x%x
```

The HTTP response body is logged to NSLog and triggers the format string vulnerability, causing stack memory to be dumped to the console log, as shown below:

```
(gdb) r
Starting program: /private/var/root/fmtstrtest
objc[8008]: Object 0x11f0b0 of class NSURL autoreleased with no pool in place -
just leaking - break on objc_autoreleaseNoPool() to debug
objc[8008]: Object 0x11e310 of class NSURLRequest autoreleased with no pool in
place - just leaking - break on objc_autoreleaseNoPool() to debug
objc[8008]: Object 0x11f540 of class NSThread autoreleased with no pool in place
- just leaking - break on objc_autoreleaseNoPool() to debug
2012-02-29 17:02:36.304 fmtstrtest[8008:303] aaaa124a600782fe5b84411f0b00
Program exited normally.
(gdb)
```

Exploitation of traditional format string vulnerabilities can be accomplished using the “%n” format specifier, allowing an attacker to write to an arbitrary

Consider the following example (taken from [7]) that simply passes the value from `argv[1]` to `NSLog`:

Popping enough data to reach the user controlled part of stack memory, we can see how the `%@` specifier causes a crash when dereferencing our pointer:

However in most situations Objective-C will use the heap for storing objects and therefore in practice, exploitation is unlikely. Further information on exploiting format string vulnerabilities can be found in [7].

Memory Corruption Issues

In the example above, an instance of the MDSec class is first created then

freed using release. However after the object has been released, the echo method is called on the previously freed pointer. In this instance a crash is unlikely, as the memory will not have been corrupted through reallocation or deconstruction.

However, consider an example whereby the heap has been sprayed with user-controlled data:

```
MDSec *mdsec = [[MDSec alloc] init];
[mdsec release];
for(int i=0; i<=50000; i++) {
    char *buf = strdup(argv[1]);
}
[mdsec echo: @"MDSec!"];
```

Running the above example will cause an access violation when the echo method is called due to the reuse of heap memory used by the previously freed object instance:

```
(gdb) r AAAA
Starting program: /private/var/root/objuse AAAA

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414149
0x320f8fbc in ?? ()
(gdb)
```

The release of iOS 5 saw the introduction of Automatic Reference Counting (ARC) (See section 3.4) which passes the responsibility of memory management from the developer to the compiler. Consequently, for apps using ARC there is likely to be a significant reduction in the number of use-after-free issues as the developer no longer bears the responsibility for releasing or retaining objects.

6. Conclusions

In conclusion, thanks to the increase in adoption of mobile computing, mobile security has never been so important. With the growing use of mobile apps both in the consumer and enterprise markets, mobile app security will continue to come under increasing scrutiny.

Perhaps the two most important issues in mobile app security are how data transport and storage are implemented as these pose the greatest risk to individuals and businesses. While there are many API specific injection style attacks and memory corruption flaws the attack surface often relies on a compromised server or insecure transport mechanisms that allow an attacker to manipulate communications to the device.

The security features offered by the platform provide a growing number of hurdles that an attacker must overcome to successfully exploit memory corruption flaws on the device. Indeed memory corruption flaws in third party apps pose little risk to the platform unless these can be combined with vulnerabilities in iOS itself.

Going forwards, while it is unlikely mobile app vulnerabilities will disappear in the near future; the advent of mobile security projects such as that of OWASP's [22] will raise the awareness of mobile security issues and hopefully help raise the bar in mobile development. Organisations looking to implement secure mobile applications should integrate security assessments throughout the development lifecycle and ensure developers and QA teams receive sufficient security education.

7. iOS App Compliance Checklist

During iOS application assessments, it is often useful to have a base line of compliance that an app should conform to, providing guidelines not only for security evaluators but also developers. Below, MDSec provides a possible platform to guide iOS assessments:

Issue	Compliance
Compiler Protection	
Application is compiled with PIE	PASS/FAIL
Application is compiled with stack cookies	PASS/FAIL
Application uses Automatic Reference Counting	PASS/FAIL
Transport Security	
Application rejects self-signed certificates: allowsAnyHTTPSCertificateForHost / continueWithoutCredentialForAuthenticationChallenge	PASS/FAIL
Application rejects expired certificates: kCFStreamSSLAllowsExpiredCertificates	PASS/FAIL
Application validates root certificates: kCFStreamSSLAllowsAnyRoot	PASS/FAIL
Application validates certificate chain: kCFStreamSSLValidatesCertificateChain	PASS/FAIL
Inter Process Communication	
Application validates the source bundle: handleOpenURL	PASS/FAIL
Application validates content of IPC parameters	PASS/FAIL
Data Storage	
Application encrypts data written with NSData	PASS/FAIL
Application encrypts data written with NSFileManager	PASS/FAIL
Keychain	
Keychain items are protected using the Data Protection API: SecItemAdd / SecItemUpdate	PASS/FAIL
UIWebViews	
Application does not load UIWebView from a local resource	PASS/FAIL

Application validates user controlled content populated in to a UIWebView: stringByEvaluatingJavaScriptFromString	PASS/FAIL
XML Processing	
Application disables external entities with NSXMLParser: setShouldResolveExternalEntities	PASS/FAIL
Application disables external entities with LibXML2	PASS/FAIL
Application builds XML with user controllable strings	PASS/FAIL
SQL	
Application uses parameterized queries for data access: sqlite3_prepare_v2	PASS/FAIL
File System	
Application sanitises path for traversal characters	PASS/FAIL
Application validates NSString paths for null bytes	PASS/FAIL
GeoLocation	
Application uses suitable level of accuracy: CLLocationAccuracy	PASS/FAIL
Application does not log location data client-side	PASS/FAIL
Logging	
NSLog is disabled in production builds	PASS/FAIL
Custom logs contain no sensitive data	PASS/FAIL
Backgrounding	
Application removes sensitive data from view when backgrounded: applicationDidEnterBackground	PASS/FAIL
Memory Corruption	
Application uses correct format specifiers for vulnerable functions	PASS/FAIL
Application does not reference freed objects	PASS/FAIL

8. About MDsec

MDsec, the company behind the Web Application Hacker's Handbook, is a global authority with a passion for information security. This has helped establish our role in defining, formalising and expanding information security through publications, tools and worldwide training. As a vendor-neutral organisation with no external investment, we can draw on our team's years of blended experience to provide security advice on technical and non-technical subjects.

The company was founded in 2011 and has seen an explosive growth in its client base which includes prestigious companies across all sectors and throughout the world. If you would like to find out how MDsec can help improve the security of your organisation, please feel free to contact sales@mdsec.co.uk for a friendly and open discussion.

9. Acknowledgements

The author would like to thank Marcus Pinto of MDsec, Ollie Whitehouse of Recx, and Hubert Seiwert for advice and recommendations during the development and review of this whitepaper.

10. References

- [1] Mobile/Tablet Top Operating System Share Trend - NetMarketShare
<http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=1>
- [2] Apple iOS 4 Security Evaluation – Dino Dai Zovi
https://media.blackhat.com/bh-us-11/DaiZovi/BH_US_11_DaiZovi_iOS_Security_WP.pdf
- [3] iOS Standard Agreement
https://developer.apple.com/programs/terms/ios/standard/ios_standard_agreement_20100909.pdf
- [4] Address Space Layout Randomization
http://en.wikipedia.org/wiki/Address_space_layout_randomization
- [5] Steffan Esser's ASLR Research
http://antid0te.com/CSW2012_StefanEsser_iOS5_An_Exploitation_Nightmare_FINAL.pdf
<http://www.suspekt.org/>
- [6] Apple Sandbox - Dionysus Blazakis
<http://securityevaluators.com/files/papers/apple-sandbox.pdf>
- [7] Auditing iPhone and iPad Applications – Ilja van Sprundel
<http://cansecwest.com/csw11/iPhone%20and%20iPad%20Hacking%20-%20van%20Sprundel.ppt>
- [8] Secure Development on iOS – David Thiel
http://www.isecpartners.com/storage/docs/presentations/iOS_Secure_Development_SOURCE_Boston_2011.pdf
- [9] Crackulous
<http://hackulo.us/wiki/Crackulous>
- [10] Mach-O File Format Reference
<https://developer.apple.com/library/mac/#documentation/developertools/conceptual/MachORuntime/Reference/reference.html>
- [11] Class-Dump-Z
http://code.google.com/p/networkpx/wiki/class_dump_z
- [12] MobileSubstrate
<http://iphonedevwiki.net/index.php/MobileSubstrate>

- [13] Cycript: Objective-Javascript
<http://www.cycript.org/>
- [14] iSOPenDev
<http://www.iosopendev.com/>
- [15] Insecure Handling of URL Schemes in Apple's iOS – Nitesh Dhanjani
<http://software-security.sans.org/blog/2010/11/08/insecure-handling-url-schemes-apples-ios/>
- [16] Citigroup iPhone Data Storage Issues
http://www.theregister.co.uk/2010/07/27/citi_iphone_app_weakness/
- [17] iPhone data protection in depth
<http://esec-lab.sogeti.com/dotclear/public/publications/11-hitbamsterdam-iphonedataprotection.pdf>
- [18] Keychain Dumper
<https://github.com/ptoomey3/Keychain-Dumper>
- [19] Skype iOS XSS
<https://superevr.com/blog/2011/skype-xss-explained/>
- [20] Billion Laughs
http://en.wikipedia.org/wiki/Billion_laughs
- [21] Abusing the Objective-C Runtime
<http://www.phrack.org/issues.html?issue=66&id=4>
- [22] OWASP Mobile Security Project
https://www.owasp.org/index.php/OWASP_Mobile_Security_Project