

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
TUT Centre for Digital Forensics and Cyber Security

An Automated Framework for securing iOS Applications

Master's Thesis

ITC70LT

Student:	Olga Dalton
Student code:	132204IVCMM
Supervisors:	Rain Ottis Roger Kerse

Tallinn
2015

Declaration

I hereby declare that I am the sole author of this thesis. The work is original and has not been submitted for any degree or diploma at any other University. I further declare that the material obtained from other sources has been duly acknowledged in the thesis.

(date)

(signature)

Abstract

The main purpose of this thesis is to write a proof-of-concept framework for securing iOS applications. The framework must focus on solving the common vulnerabilities of iOS applications and its integration should be easy enough even for iOS developers who are less aware of security issues. Moreover, it must not require any changes to the architecture or class inheritance of the existing application.

The methodology behind the devised framework is based on the theoretical research on iOS applications security and an analysis of existing solutions. It includes multiple security controls, such as countermeasures for insecure data storage, unintended data leakage and insufficient transport layer protection vulnerabilities. Most of the security controls are injected automatically into the application and do not require much, if any, additional manual setup. Automatic injection is achieved by utilising advanced runtime manipulation techniques of Objective-C, the primary programming language for writing iOS applications.

The analysis of the framework in terms of performance and binary size overhead has indicated minimal impacts on the application. Moreover, a case study of the framework integration into a real world project has clearly proven the viability of the idea. Consequently, the goals of this thesis were successfully fulfilled.

The resulting framework offers multiple possibilities for improvements. There are at least two evident directions for future framework developments. First, more complex security controls must be implemented. In conjunction with these controls, additional performance and binary size optimisations might be required. Second, going beyond the ordinary framework and providing a plugin for the iOS development environment to randomly generate the framework variations is also an important route to consider.

The thesis has been written in English and includes 68 pages of text, 7 chapters, 12 figures and 4 tables.

Annotatsioon

Käesoleva lõputöö põhieesmärgiks on luua iOS rakenduste turvamiseks mõeldud tarkvararaamistiku kontseptsiooni tõestamise versioon. Raamistik peab lahendama rakendustes sageli esinevaid turvavigu, kusjuures raamistikuga sidumine peab olema piisavalt lihtne ka nendele arendajatele, kes teavad turvalisusest vähe. Lisaks ei tohi raamistiku kasutuselevõtt nõuda olemasoleva rakenduse arhitektuuri või klasside hierarhia muutmist.

Töö käigus luuakse raamistik, mis põhineb nii iOS rakenduste turvalisust käsitleval teoreetilisel uurimisel kui ka olemasolevate lahenduste analüüsil. Raamistik sisaldab mitmeid turvalisuse komponente, sealhulgas abinõusid ebatavalise andmete hoiustamise, soovimatu informatsiooni lekke ning ebapiisavate võrgusuhtluse kaitsemehhanismide vastu. Enamik turvakomponentidest lisatakse rakendusse kas täiesti automaatselt või minimaalse seadistusega. See baseerub Objective-C, põhilise iOS arenduses kasutatava programmeerimiskeele, dünaamilisusel ning programmi täitmise ajal funktsionaalsuse lisamisel.

Raamistiku mõjud nii jõudlusele kui ka aplikatsooni suurusele on vastavale analüüsile toetudes minimaalsed ja vastuvõetavad. Idee rakendatavuse analüüsiks viiakse läbi juhtumiuuring, mis räägib raamistiku integreerimisest olemasolevasse rakendusse ning selle mõjudest. Juhtumiuuring näitab positiivseid tulemusi. Sellest kõigest järeldub, et lõputöö eesmärgid on edukalt saavutatud.

Loodud raamistikul on mitmeid edasiarendamise võimalusi, sealhulgas vähemalt kaks selget edasimineku suunda. Esiteks on vaja lisada täiendavaid ja keerukamaid turvalisuse komponente. Uute komponentide lisamisega peaks kaasas käima ka jõudluse ja raamistiku suuruse optimeerimine. Teiseks tuleb kaaluda raamistiku komponentide varieerimist läbi dünaamilise koodi genereerimise. Seda oleks mõistlik teostada integreeritud programmeerimiskeskonna tarkvaramooduli näol.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 68 leheküljel, 7 peatükki, 12 joonist ning 4 tabelit

List of Acronyms and Terms

iOS	A mobile operating system developed by Apple Inc.
Jailbreaking	Jailbreaking is a technique to obtain system-level rights on iOS device by exploiting a flaw in the operating system.
Springboard	A standard iOS application that manages the home screen.
SSL	Secure Socket Layer, a transport layer protocol
XSS	Cross-site scripting, a security vulnerability that is typical for web applications.
API	Application programming routines
PLIST	Property list, an XML file that typically stores serialized objects.
SQL	Structured Query Language, a standard language for accessing databases.
OWASP	Open Web Application Security Project, a security-oriented online community.
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP over SSL
Dsym	A separate file that stores debug symbols of an iOS application.
URL	Uniform resource locator
OpenPGP	A communication standard for secure data exchange.
PKI	Public key infrastructure
MITRE	Massachusetts Institute Of Technology Research And Engineering
HFS	Hierarchical File Structure, a file system common for Apple devices.
Xcode	An integrated development environment for iOS and Mac OS apps development.
App Store	An electronic shop to buy and download iOS applications
OpenSSL	An open source toolkit for SSL protocols implementation

Table of Contents

1. Introduction	10
1.1 Motivation	11
1.2 Scope	11
1.3 Outline	12
2. Theoretical background	14
2.1 Overview of iOS development	14
2.1.1 Objective-C	14
2.1.2 Cocoa Touch	15
2.1.3 Common application architecture	15
2.2 Threat model	22
2.2.1 Attack goals	22
2.2.2 Attack scenarios	23
3. Related works	28
3.1 Cryptanium code protection	28
3.2 Security solutions from the EldoS Corporation	29
3.3 IMAS – iOS Mobile Application Security	30
3.4 Comparative analysis	32
4. A methodology for securing iOS applications	35
4.1 Fundamental security principles	35
4.2 Automatic or manual security	36
4.3 Patching and ease of integration	36
4.4 Proposed security controls	37
4.4.1 Binary patching	37
4.4.2 Runtime analysis	37
4.4.3 Insecure data storage	38
4.4.4 SSL attacks	39
4.4.5 Insecure data logging	40
4.4.6 Text entry caching	40
4.4.7 Application screenshot	40

4.4.8 Web content.....	41
4.4.9 URL schemes.....	42
5. Implementation of the security framework	43
5.1 Security framework architecture	43
5.1.1 Controls injection	46
5.1.2 Runtime protection	49
5.2 Framework configuration	50
6. Analysis of the framework	53
6.1 Impact on performance and binary size.....	53
6.2 Case study: using the framework in a real-world application	55
6.3 Limitations and possible issues	57
7. Conclusions and future works	59
References	62
Appendix 1: Runtime integrity protection.....	69
Appendix 2: Framework implementation guide.....	73
Appendix 3: Kosmos IMAX security review report	79

List of Figures

Figure 1: iOS Data Storage.....	17
Figure 2: Reading iOS crash reports.....	21
Figure 3: Insecure API details logging.....	21
Figure 4: Common attack scenarios [21, 32, 50].....	23
Figure 5: Eavesdropping SSL requests.....	25
Figure 6: Effect of inline functions.....	26
Figure 7: Adding an iOS framework in Xcode 6.3.....	43
Figure 8: Security framework architecture.....	44
Figure 9: Simple method swizzling.....	47
Figure 10: New method implementation.....	47
Figure 11: Objective-C method structure.....	48
Figure 12: Swizzling method with C functions.....	49

List of Tables

Table 1: Comparison of iOS security solutions..... 33

Table 2: Framework configuration options 51

Table 3: Encryption performance tests results 54

Table 4: Anti-piracy controls performance impact..... 55

1. Introduction

Smartphones are currently among the most popular technologies and many people consider them to be something of a necessity, not a privilege. Indeed, with more than 1 billion users worldwide [1], the mobile industry has become a serious competitor to desktop computers.

Smartphone users can be roughly divided into two categories. Ordinary users comprise the first category. Most of these users are not concerned about security when playing new games or ordering a taxi. Some of them are even jailbreaking their devices without the understanding that hacking in this way opens the device to more attacks and thus weakens the level of security [2]. The common opinion is that as long as jailbreaking offers more free apps or new springboard animations, security is not a significant concern [3].

The other types of users are in the minority. These users are hackers. Jailbreaking is essential for them, not because of the free apps or a lack of knowledge, but rather because of curiosity, money or revenge [4]. Hacking apps, stealing data or distributing malware is often their primary source of living, and if not, it is at least a hobby.

Both user categories pose a great challenge to mobile software developers. They must design apps in a way that ordinary users will be satisfied and so that hacking the program or stealing sensitive data will be as difficult as possible. However, in the rush to conquer a larger market share, developers often ignore security aspects. A study by Hewlett Packard Security found that 90% of more than 2000 of the examined mobile apps had various simple vulnerabilities, including improper handling of private data, inappropriate cryptography or a lack of protection mechanisms against reverse engineering. [5]

The situation is similar for both iOS and Android. Yet, Android developers are considered to be more security aware [6]. This is mainly due to the greater media coverage of Android vulnerabilities, while iOS is usually described as a very secure platform. Furthermore, according to Payscale statistics, iOS Developers generally have less work experience (approximately 4.5 years) when compared to developers in other fields. In contrast, Java developers have over 6 years of experience and the average for all software development is 6.4 years. [7, 8, 9] More experienced software developers typically pay more attention to security. Moreover, the available statistics only take into account professional developers, but

there are plenty of independent iOS developers who have no previous mobile or even programming experience.

1.1 Motivation

The iOS mobile market is a complex and desirable target. On one hand, application developers and companies need to start thinking more about security issues. On the contrary, staying competitive might mean cutting development costs, which in fact can reduce security. For example, 50% of all iOS Developers make less than \$500 per month from App Store sales and only 27% of iOS Developers make more than \$5000 per month [10]. This is considered the minimum salary level for a developer in a US company [7]. Requiring security competence on a limited budget such as this is almost impossible.

Therefore, there is a need for a security solution that:

- would be easy to integrate;
- would not require a great deal of security knowledge or awareness;
- would not have expensive commercial licenses;
- would help to avoid common vulnerabilities.

Such a solution will help to achieve better mobile app security without significant additional time resources. Initially this would be beneficial for independent developers, small development companies and startups. In addition, in the long run, it could also become a useful toolkit for enterprise applications.

1.2 Scope

The main purpose of this thesis is to write an iOS Framework that will help to eliminate the common vulnerabilities found in iOS applications. The scope of the framework implementation will be limited to providing a proof-of-concept and will thus focus on somewhat simple issues, such as insecure data storage, improper SSL certificate verification or runtime protection. Additionally, it will serve as a core platform for future developments to incorporate more complex protection mechanisms.

Prior to the actual implementation, a study that will examine the main concepts of iOS applications security and related works will be conducted. There have only been a handful of similar projects done, so these will be analysed in detail with regard to the simplicity and benefits of increased iOS security.

1.3 Outline

The thesis can be divided into five important sections:

- Researching theoretical background and technological aspects;
- Analysing similar works;
- Building up the technical foundation of the framework;
- Implementing the framework;
- Validating results with a performance and benefits study.

Thus, this thesis consists of five parts.

Section I offers insights into the theoretical aspects of iOS application security. It gives an overview of iOS programming languages, application architecture, frequent security mistakes and the common threat model. It shows that most security problems are caused by insufficient protection mechanisms provided by the operating system, so a parallel can be made between the iOS development framework and attack vectors. Mitigation options for those security issues are also briefly described.

Section II concentrates on researching and analysing related projects. The aim is to find works with similar objectives and explore possible ways to achieve these objectives.

Section III provides the conceptual foundation that is needed to implement a security framework. It describes the fundamental principles that form the basis of the framework and discusses what the reasonable amount of automation should be and what must be done manually. Additionally, it attempts to find a balance between ease of integration and making the act of patching a binary more difficult.

Section IV introduces the security framework, which is the main practical result of this thesis. It discusses the architecture and integration process of the framework.

Section V analyses the resulting framework in terms of performance and benefits. It also examines potential limitations of its implementation and provides solutions for these limitations. The issues discussed are both technical and conceptual in nature.

The conclusion summarises the significant results of the thesis, and also addresses unsolved problems and possible future studies.

2. Theoretical background

Development of iOS applications is heavily based on tools and methods that are provided by Apple, such as Xcode, which is the primary development environment for Apple devices. Until very recently, the only programming language used to develop native iOS applications was Objective-C. On October 22, 2014 Apple released another programming language called Swift, which can be used as an alternative or together with Objective-C. Even though it has officially been released, Swift is still under development and will most likely be adapted in the future. Because most iOS APIs are written in Objective-C, Swift was built from the very beginning to be compatible with Objective-C frameworks and classes. However, when it comes to advanced features of the language, Objective-C provides more flexibility, while Swift limits much of the dynamism to make the language simpler and safer [11].

Thus, this thesis will concentrate on and use Objective-C as a primary and more feature-rich iOS programming language that can also be utilised in Swift-based applications.

2.1 Overview of iOS development

2.1.1 Objective-C

Objective-C is the primary programming language choice for writing iOS applications. It is a superset of the C language, which adds object-oriented features and a dynamic runtime. Its syntax, primitive types and flow control are inherited from C, while the classes and methods have a specific syntax that is derived from Smalltalk. [12]

Objective-C implements all common object-oriented structures similarly to other popular programming languages. However, it provides a different feature called a category, which is unique. Categories allow extending functionality of classes without sub-classing them, even if there is no direct access to the original source code. For example, it is possible to add new methods to system classes or replace existing ones [12].

Another difference between Objective-C and other languages is that in Objective-C classes do not call their methods directly. Instead, each method is resolved dynamically by using an "objc_msgSend" runtime function. This function traverses the hierarchy of classes and determines the proper class and method according to the parameters given. This provides

great flexibility to override method calls at runtime and add dynamic behaviour to classes. [13]

2.1.2 Cocoa Touch

Cocoa Touch is a collection of frameworks that are used to build iOS mobile applications. It is generally written in Objective-C and is derived from a similar Mac OS framework called Cocoa. The framework helps to solve common iOS development tasks such as animation, UI elements, data storage or networking. Cocoa Touch is based on the model-view-controller architecture. Consequently, most iOS applications also follow this pattern. [14]

2.1.3 Common application architecture

Developing each iOS application requires solving a similar set of tasks. As an example, most iOS applications implement some type of backend communication, thus incorporating networking solutions. After querying the application or user specific data from the backend, it must be saved to the local database and presented graphically. Certain user actions trigger changes to this data, which need to be synchronised with the backend. Optionally, applications can use local data from the phone, such as GPS location, contents of the address book, Facebook friends etc. Each of these tasks has many possible solutions provided by the iOS operating system and the developer needs to choose an appropriate solution according to the needs of the application. This section will give an overview of commonly used elements and comment on the security risks of each element.

2.1.3.1 Data storage

One of the easiest ways to save data in iOS is to use preferences storage, called NSUserDefaults. Its usage is limited to small amounts of data, such as strings, booleans or keys, but implementation is very easy. For example, only a single line of code is needed to save a value. Hence, its wide usage comes as no surprise and sensitive data, such as user credentials or game bonuses, is often saved using NSUserDefaults. [15] However, NSUserDefaults are internally stored in the application's documents folder as a plain-text file and can be easily accessed or modified even on a non-jailbroken device.

Storing large amounts of data in NSUserDefaults is not wise from a performance standpoint. For larger files or data that is more complex, developers tend to use the iOS file management system. iOS frameworks provide simple methods to write almost all data types to file and retrieve it, including strings, dictionaries and arrays. Those files, if saved without additional

encryption, can be retrieved as easily as with `NSUserDefaults`. Furthermore, all such data becomes unencrypted, when the device is unlocked for the first time after a reboot. Therefore, it is generally accessible on the device even without knowing user's password. It is possible to encrypt it with a single key using iOS Data Protection API, but this method is rarely used. [16]

Managing non-trivial data models and classes using files is extremely troublesome. For that purpose, a relational database is one of the best choices. Similar to Android, iOS provides the ability to use SQLite databases. Nevertheless, SQLite is seldom used directly in iOS. Instead, a technology called Core Data is utilised, as it is easier to use than SQLite. Core Data adds a layer of abstraction on top of SQLite and allows the use of data classes directly without writing SQL queries. [15] The simplicity of integrating Core Data creates a feeling of a “magical” data storage solution, while in reality all data is maintained in a SQLite database. Similar to other data storage solutions, this database is unencrypted and located in the application's documents folder. [17]

Current iOS data storage solutions allow access for three possible attack scenarios:

1. If a device is stolen, sensitive data can be accessed.
2. On jailbroken devices, malicious applications can access sensitive data stored in a plain-text file and send it to the central server.
3. iOS games often store information about bonuses or purchased improvements in `NSUserDefaults`. Modifying contents of the `NSUserDefaults` file can provide a user with unlimited lives or virtual money and is fairly easy even for non-hackers to execute.

Insecure data storage is the second highest vulnerability in the OWASP Mobile TOP 10 2014 and the highest among mobile vulnerabilities that are related directly to iOS applications. [18]

Figure 1 shows an internal structure of a typical iOS application, which includes both `NSUserDefaults` and a Core Data database. Both of these are plaintext. The SEB Eesti iOS application is used as an example.

SEB Eesti		
Documents		535 kB
Atms-v1.sqlite	SQLITE	24 kB
Atms-v1.sqlite-shm	SQLITE-SHM	32 kB
Atms-v1.sqlite-wal	SQLITE-WAL	477 kB
Library		
Application Support		2 kB
Caches		
FlurryFiles		
Preferences		697 B
ee.seb.pank.plist	PLIST	697 B
StoreKit		4 kB
tmp		

Figure 1: iOS Data Storage

2.1.3.2 Transport layer

Cocoa Touch provides an easy networking encapsulator called `NSURLConnection`. It handles all common scenarios and allows modifying headers, cookies and cache settings to be used. `NSURLConnection` or its wrappers are enough for most applications. [19] One of the most common attack vectors for iOS applications is eavesdropping communication between the application and the server. This is possible due to the fact that many applications use plain HTTP connections. There is not much that can be done to protect against eavesdropping while using HTTP and inventing custom encryption mechanisms instead of using SSL is not reasonable.

However, even the incorporation of SSL does not always ensure security. For example, test backend environments usually use self-signed SSL certificates, which are not validated by any certificate authority. For testing purposes, certificate trust chain validation should be disabled on the client side as well, because by default, iOS does not accept self-signed certificates. It often happens that it remains disabled even in production versions of the application because of the negligence of developers. [17]

Another caveat is that any downloaded content that supports caching (by using `Cache-Control` headers) is automatically saved to the local plaintext SQLite file. Therefore, it might be necessary to disable caching for requests with highly sensitive data. [20]

2.1.3.3 Web content

`UIWebView` is used to embed web content in the application. [24] This might be as simple as a single help page or an entire application that is implemented using web technologies. Similar to all other web content, when it is embedded into iOS applications, it might become a

target for exploits. For example; a recent study, which focused on mobile banking apps security, revealed that 50% of the applications analysed were vulnerable to Javascript injections. [25] In addition to pure webpage Javascript injection, the application can be attacked through unvalidated user input, which is used to compile a native javascript call. The opposite scenario is possible in some cases as well, allowing an attacker to reach native functionality, such as sending an SMS. [25] This is possible because of the communication bridge between the native functionality and the web content.

2.1.3.4 Text entry

A common way to provide the text entry functionality is to use standard Cocoa Touch elements UITextField and UITextView. UITextField is used for shorter texts, e.g. usernames and passwords, while UITextView is designed for longer data. As a bonus, those keyboard classes offer autocorrect without any additional work. However, there is a problem with autocorrect because any data typed into a textfield with autocorrect turned on (which is a default value) becomes cached in clear text by the system. The only exception is a secure textfield meant for password entry, which visually hides entered data. Hence, it is not suitable for other text data, such as usernames or phone numbers. [21]

The iOS cut-and-paste buffer, called UIPasteBoard, raises an additional security concern, because all data copied to the pasteboard is also cached in plaintext. [21]

In order to stop caching entries for possibly sensitive data, autocorrect and pasteboard must be turned off. For example, there is no actual use for autocorrecting usernames, but it poses an unintended data leakage risk. The issue of unintended data leakage is also emphasised in the OWASP Mobile TOP 10. [18]

2.1.3.5 URL schemes

URL schemes are a frequently used iOS feature, which allows communication between different apps. By using URL schemes, applications and websites are allowed to launch other applications that are registered to recognize the requested scheme and ask to perform additional actions. For example, opening the URL "youtube://watch?token=VIDEO_ID" launches the Youtube app and starts playing the video "VIDEO_ID". [22]

URL schemes are often abused by malicious websites as could be seen with the Skype iOS application. More specifically, a vulnerability in Skype allowed to start a phone call without

the user's prompt by opening an URL Scheme. [23] URL scheme vulnerabilities can cause considerable financial damages, if discovered in popular applications.

One of the possible ways to prevent URL scheme vulnerability is to always ask for permission from the user before performing any action, as is done in the Apple Phone app before calling the requested number. [22]

Another method of vulnerability prevention is to check the caller application and block requests from suspicious origins. This would be suitable for applications that know in advance all possible users of URL schemes. However, the validation is often omitted even in such cases and the simpler delegate method without the source application information is generally preferred over the more secure method.

Again, the URL schemes exploit is listed in the OWASP Mobile TOP 10. [18]

2.1.3.6 iOS keychain

iOS keychain is a device-wide secure data container, which is used to secure applications' sensitive data. All data in the keychain is preserved across re-installations. It is usually advised not to store any passwords or session tokens on the device, but if they must be stored locally, iOS keychain is the most secure method for doing so. Nevertheless, communicating with keychain incorrectly can put a user's private data in danger.

In detail, keychain API allows different protection levels to be set for each value. The default protection level is specified as `kSecAttrAccessibleWhenUnlocked`, which prevents the data from being read without knowing the passcode. Many applications use another value `kSecAttrAccessibleAlways` or `kSecAttrAfterFirstUnlock` and that makes data readable even without knowing the passcode. The probable reason for setting an insecure protection class is the necessity to use keychain data when device is locked, or insufficient knowledge of security. [26]

2.1.3.7 Cryptography

iOS applications are protected with basic cryptography by default. However, the protection is limited only to the binary, while all associated files and data remain mostly unencrypted. Furthermore, a skilled attacker can easily wipe the default binary encryption using open source tools, which means that reverse engineering of the binary cannot be prevented. Reverse

engineering gives an attacker a good overview of an application's architecture and working principles. [17, 18, 27]

Consequently, all static sensitive data must be hidden in code as well as possible. One of the frequent mistakes in iOS applications is the use of a hardcoded encryption key, which is the same for all application instances. Finding a hardcoded encryption key endangers all owners of the application, because a key from one application instance can be used to decrypt data in all instances globally. [17, 18, 27]

2.1.3.8 Data logging

In general, iOS applications utilise multiple logging mechanisms. One of these mechanisms is automatic and concerns application crash logs; all application crashes are stored on the device and can be easily accessed. To symbolicate a crash log, an iOS developer needs the "dsym file". It is a file created together with a binary that contains debug symbols and is stored separately to protect crash information from attackers. However, dsym is only needed to symbolicate calls to application specific methods, but calls to underlying iOS APIs are still accessible. A stack trace of the crash in combination with deliberately malformed data can provide insight into the internal logic of the application. As an example, Figure 2 shows a typical crash log. It is not clear which methods are called by the application, but those method calls result in the tableview reload, which crashes the application due to the logic error in the tableview datasource. The tableview control is an iOS control, which presents data in rows and columns.

```

Incident Identifier: 6E887ADD-74BF-46B9-9F9B-C064695484A3
CrashReporter Key: 53d602632f6838b2be5fe95b6ef713e83c0d7459
Hardware Model: iPhone7,2
Process: [3867]
Path: /private/var/mobile/Containers/Bundle/Application/60D22269-DE36-4276-B3DC-22B1DE3B70C
Identifier:
Version: 2.1 (2.1)
Code Type: ARM (Native)
Parent Process: launchd [1]

Date/Time: 2015-02-27 19:16:04.067 +0200
Launch Time: 2015-02-27 19:16:02.943 +0200
OS Version: iOS 8.1.3 (12B466)
Report Version: 105

Exception Type: EXC_CRASH (SIGABRT)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Triggered by Thread: 0

Last Exception Backtrace:
0 CoreFoundation 0x29f1d49f -[ExceptionPreprocess + 127
1 libobjc.A.dylib 0x37734c80 objc_exception_throw + 38
2 CoreFoundation 0x29e319e -[NSArrayM objectAtIndex:] + 232
3 UIKit 0x2d4ada9 -[UITableView reloadData] + 400
4 0x000b7267 0xa4000 + 78439
5 0x000b4f03 0xa4000 + 69379
6 0x001db265 0xa4000 + 127405
7 0x000b53b9 0xa4000 + 70585
8 UIKit 0x2d3f3d3b -[UIViewController
  setViewAppearState:isAnimating:] + 438
9 UIKit 0x2d487a9f -[UINavigationController
  _startTransition:fromViewController:toViewController:] + 678
10 UIKit 0x2d487573 -[UINavigationController
  _startDeferredTransitionIfNeeded:] + 578
11 UIKit 0x2d4872dd -[UINavigationController
  viewWillLayoutSubviews] + 44
12 UIKit 0x2d487271 -[UILayoutContainerView layoutSubviews]
  + 184
13 UIKit 0x2d3dba6f -[UIView(CALayerDelegate)
  layoutSublayersOfLayer:] + 514
14 QuartzCore 0x2ce03a0d -[CALayer layoutSublayers] + 136
15 QuartzCore 0x2cdff3e5
  CA::Layer::layout_if_needed(CA::Transaction*) + 360
16 QuartzCore 0x2ce2de5d -[CALayer layoutIfNeeded] + 140
17 UIKit 0x2d454f99 -[UIViewController
  window:setupWithInterfaceOrientation:] + 344

```

Figure 2: Reading iOS crash reports

Another concerning area is manual data logging using an iOS standard output method, called NSLog. This method might be useful during application development, but logging authentication tokens, passwords or API details in the release version of the application is considered a high-risk vulnerability. All NSLog data is cached on the device until it is rebooted and can be retrieved anytime. Figure 3 shows a popular Estonian application that logs API endpoints using NSLog.

```

Mar 3 19:34:13 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/accounts/login/
Mar 3 19:34:13 Olgas-iPhone [4528] <Notice>; TestFlight: Started Session
Mar 3 19:34:14 Olgas-iPhone locationd[3189] <Notice>; Gesture EnabledForTopClient: 0, EnabledInDaemonSettings: 0
Mar 3 19:34:14 Olgas-iPhone [4528] <Notice>; TestFlight: Team Token Error - Your Team Token is not recognized
Mar 3 19:34:15 Olgas-iPhone syncdefaults[4638] <Notice>; (Note ) marked "com.me.keyvalueservice" topic as "opportunistic" on <APSCConnection:
0x126d183e0>
Mar 3 19:34:16 Olgas-iPhone webinspector[4642] <Warning>; No debugger connection after 2 seconds; exiting.
Mar 3 19:34:16 Olgas-iPhone com.apple.xpc.launchd[1] (com.apple.webinspector[4642]) <Warning>; Service exited with abnormal code: 1
Mar 3 19:34:20 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/push/register/
Mar 3 19:34:20 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/accounts/36559/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/reminders/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/movies/votes/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/accounts/36559/feed/friends/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/accounts/36559/feed/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/leaderboards/global/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/leaderboards/friends/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/orders/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/orders/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/movies/leaving/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/movies/upcoming/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; Request Path: /et/restapi/screenings/2015/3/3/
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; WE HAVE Screenings
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; WE HAVE Screenings
Mar 3 19:34:21 Olgas-iPhone [4528] <Warning>; NSScanner: nil string argument

```

Figure 3: Insecure API details logging

2.2 Threat model

As shown in the previous section, many of the default options provided by iOS APIs for app developers are not sufficiently secure. Data storage is a plentiful example, because there is no setting to encrypt user defaults or database contents without using additional third party libraries (see 2.1.3.1). Moreover, there are multiple recurring vulnerabilities due to an improper selection of options or hardcoded keys, which raises questions about trusting iOS applications.

Furthermore, Apple's development resources and popular tutorials lack information about potential security dangers. A good example would be the text entry caching that was discussed in section 2.1.3.4, which is not stated clearly anywhere in Apple documentation. Another similar vulnerability arises during the application minimising, when the iOS system takes a screenshot of the current view to present it during the application launch from the background mode. At first glance, this harmless feature makes the transition from the background mode smoother, however screenshots can capture sensitive data. [21]

Consequently, it appears that developers are presented with a broad range of challenges to mitigate possible attack vectors. In order to better understand those challenges, this section will first explain the motives behind attacks on iOS applications and then it will cover the various attack methods by presenting some example scenarios.

2.2.1 Attack goals

The goals of mobile applications hackers are similar to those that occur in the world of personal computers:

Private data A mobile device is an important storage tool for personal data, therefore private information plays a crucial role in hackers' behaviour. Attackers are mostly focused on the confidentiality aspect of information, because it is the most profitable. They are aware that people tend to reuse passwords on different sites and that it is highly possible that passwords for a mobile game and an email account will be the same. Applications that store credentials improperly or reuse cryptographic keys are the primary target here. [28]

Backend Another possible target is the backend of the application. Often, developers do not pay enough attention to APIs for mobile applications and their security. Therefore, exploiting mobile endpoints might be easier than compromising a web version of the application. The

main goal is to gain access to the server in order to use it in denial of service attacks or distribute malware to its visitors. Applications are used only to investigate API behaviour and obtain access tokens. [17]

Computing resources Hackers also look to utilise the computing resources of mobile devices to use them in botnets or bitcoin mining systems. While modern mobile devices are still not as powerful as personal computers, the large number of devices in combination with high speed Internet makes them an attractive target. [28]

Piracy The last aspect that is worth mentioning is piracy. Several popular applications on the iOS App Store take advantage of a “freemium” monetisation strategy, which means that the application is free, but it sells multiple premium additions. Piracy of such programs is rather popular among owners of jailbroken devices. Pirating an iOS application usually involves patching the binary. [29]

2.2.2 Attack scenarios

Due to the fact that most iOS applications are closed source, a common attack involves a blackbox analysis of the binary. Despite the widespread opinion that attacking iOS applications always requires a jailbroken device, some attacks can be performed even on a stock iPhone. Figure 4 provides a schematic overview of possible attacks. Attacks that require a jailbroken device are presented in blue and all others in red.

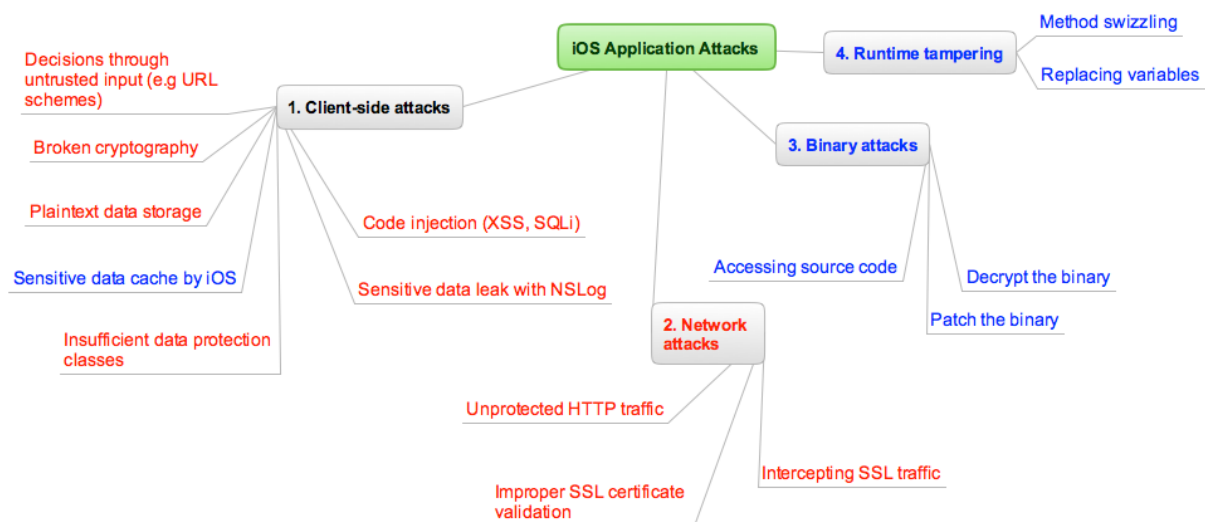


Figure 4: Common attack scenarios [21, 32, 50]

2.2.2.1 Client-side attacks

Client-side attacks focus directly on the detection of mistakes made by developers: misuse of iOS APIs, flaws in sensitive data storage and remotely exploitable vulnerabilities that can be potentially profitable. Testing and detecting most of them does not require a jailbroken device, because it essentially involves manual analysis using controls provided by the application and inspecting produced databases or files. Nevertheless, depending on the level of vulnerability that is found, actually making use of it might require a more sophisticated approach that involves patching the binary or creating a malicious application to steal valuable information.

2.2.2.2 Network attacks

Most network attacks target confidential communication between the client and the server. A passive man-in-the-middle attack intercepts data in order to collect it for further analysis, which includes retrieving passwords or credit card data. Active attacks alter data prior to sending it to the legitimate recipient in order to disclose more sensitive information or force the user to perform unwanted actions. [30] Both attacks are possible due to the plaintext HTTP connection between the iOS application and the server, which is very widespread. To intercept SSL traffic, it is possible to make use of an improper SSL certificate validation on the client side and replace the target certificate. The application will still accept the malicious certificate, because it is programmed to accept invalid certificates. The attacker will then be able to decrypt the communication channel and access plaintext data. [32]

Another reason that attackers execute man-in-the-middle attacks is to study API behind the application to discover possible weaknesses on both sides in order to gain a better overview of the application logic or to steal API keys. A great share of iOS applications is unprotected against such attacks, because even the properly implemented SSL does not prevent eavesdropping. The reason for this is that default certificate validation does not ensure that the provided certificate is legitimate; it only checks the technical validity by verifying the chain of trust and checking the hostname. An additional level of security can be added by allowing the connection to use only a specific certificate. This technique is called SSL pinning. [32] As an example, Figure 5 shows intercepted https communication from multiple popular applications. In contrast, there is also a request from the Cash application from Square Inc, which implements SSL pinning, so that all illegal requests are dropped and no sensitive data is revealed.

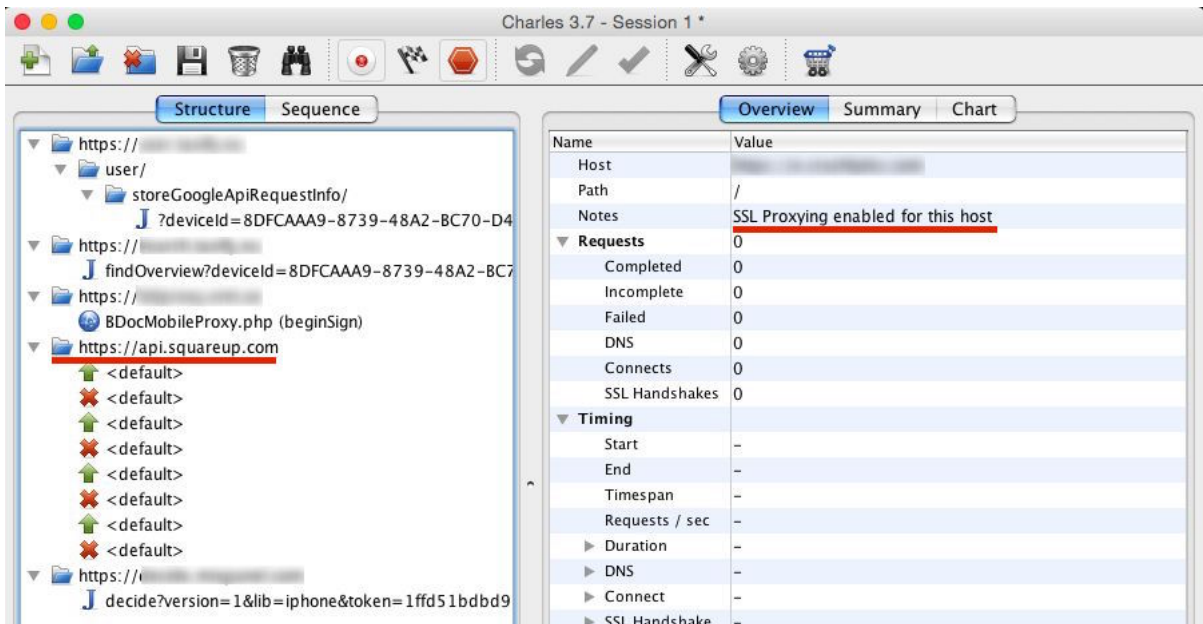


Figure 5: Eavesdropping SSL requests

2.2.2.3 Binary attacks

Once a skilled hacker gains access to the binary of the iOS application, which generally occurs by downloading it from the official App Store, reverse engineering is a matter of time. The problem with Objective-C is that it can be reversed back to the high-level source even without losing method and variable names. Thus, it is inevitable that an experienced attacker will gain access to the application's internal logic and can modify it by patching the binary. The purpose of security controls is to make it as difficult as possible, so most of ordinary hackers will give up. [31]

Possible security controls include:

Usage of C or C++ functions for security-critical operations C and C++ functions are less vulnerable to interface dumping, so it would be more difficult to find and change critical places. [31]

Inline functions Inline functions are C and C++ functions that instruct the compiler to copy the body of the function to the place where it was called instead of calling it directly. Its direct purpose is to decrease the overhead of a function call. From a security perspective, duplicating code increases the complexity of patching the binary, because instead of replacing the return value of the function in one place, it is needed to replace that value multiple times, wherever the function is called. When this occurs, an attacker will need more time to patch

the binary and it is more likely that a mistake will be made that will corrupt critical code sections. [31]

Figure 6 shows a graphic depiction of the result of using inline functions by providing decompiled code of the iOS binary. There is a method called `viewDidLoad` and a function called `_someInterestingFunction`. The `viewDidLoad` uses the `_someInterestingFunction`. Without inlining, `_someInterestingFunction` is called directly by the `viewDidLoad` method and the attacker needs to modify only the body of this function. After the inlining, `_someInterestingFunction` becomes a part of the `viewDidLoad` method. Hence, each place where `_someInterestingFunction` is used must be patched.



Figure 6: Effect of inline functions

Application integrity checks Inserting the defensive code to detect whether the binary was modified, or using injected payloads will make the patching more difficult because an attacker will also need to understand and tamper with integrity checks logic.

Jailbreak detection Many security-sensitive apps implement the jailbreak detection logic in order to prohibit application execution or limit its functionality in a less secure environment.

2.2.2.4 Runtime tampering

A runtime analysis of iOS applications is used to trace internal control flow and modify application execution logic. In Objective-C, implementations of any method can be replaced with a completely different method using a technique called “method swizzling”. For example, the method that is used to check whether the user is authorised to access protected assets, can easily be replaced with a method that always returns “true”, so protected areas will be accessible without authenticating.

One of the main tools used to perform a runtime analysis of applications is the debugger (GDB or LLDB). A debugger allows the attacker to set or call breakpoints for interesting methods. Values of instance variables can also be changed and additional methods can be written and injected into the application. To protect the application against runtime manipulation, debugging should be prohibited in the release mode. [21]

3. Related works

The general idea of providing frameworks to secure iOS applications is not new. This section will briefly describe 3 different projects that aim to secure iOS applications. Each selected project has its own unique approach to iOS security. Finally, a table of comparison will conclude the main principles of each framework.

3.1 Cryptanium code protection

Cryptanium Code Protection aims to mitigate both binary and runtime attacks by injecting integrity, debugger and jailbreak checks into the application and obfuscate the binary. It provides an easy-to-use GUI tool, which starts with the application analysis to find sensitive code sections, proceeds with protective checkers injection and finally obfuscates the code and the program's control flow. Cryptanium is not a traditional framework, but rather a toolkit that integrates security protection both on the code and assembly level. [33]

Because Cryptanium is a cross-platform tool that can protect applications for all major desktop and mobile operating systems, its protection mechanisms differ across target platforms.

The main protection mechanisms for iOS are:

- **Integrity, anti-debug and jailbreak protection** Security of the application is ensured by inserting specific functions called checkers into the application code. Integrity checkers calculate the checksums of different overlapping sections of the binary and the application's read-only data. In the case of a checksum mismatch, it is highly possible that the application has been tampered with, and it terminates deliberately. Anti-debug checkers periodically scan running processes to indicate the presence of a platform specific debugger (GDB or LLDB for iOS). Jailbreak checkers try to detect specialized applications that require root permissions or access system applications and settings. Such operations are only possible on a jailbroken device. [33]
- **Code obfuscation.** In addition to various security and anti-piracy checks, Cryptanium is also a sophisticated obfuscator. It incorporates multiple standard code obfuscation

techniques, such as methods renaming, functions inlining and strings hiding. Another interesting obfuscation technique used by Cryptanium is the control flow flattening algorithm, which makes analysis of the control flow more difficult by changing the execution of the program to run in a loop of parallel code blocks. [33]

Consequently, Cryptanium is a toolkit that attempts to mitigate binary tampering and runtime manipulation risks. It is introduced as a premium class product and is nearly unbreakable according to the vendor. [34] The project is closed-source and has an expensive commercial license. The price depends on the customer and project size, but in most cases, it is not affordable for small or medium applications. Therefore, Cryptanium is an excellent solution for banking and financial applications, which is also emphasized on the product's webpage. [35]

3.2 Security solutions from the EldoS Corporation

The EldoS Corporation has released a cross-platform commercial suite of software components that provide implementations for popular security standards, Internet protocols and data storage solutions. Frameworks are written in C++ to ensure cross-platform compatibility, however several platform specific editions are offered. [36]

The main security components for iOS are:

- **Secure data storage** EldoS strives to mitigate insecure data storage risks by providing a secure data container, called Solid File System. It is ensured that application files will be inaccessible by other applications or attackers due to the built-in encryption support. Solid File System uses a custom data storage format, which is similar to ZIP but has more advanced features, such as the support of SQL-like data search retrieving or file tagging. Both the encryption and custom format make attacking data storage extremely difficult. [37]
- **Secure bidirectional network communication** Another framework from the EldoS Corporation, SecureBlackbox offers the network communication component, which can work both as a server and a client. While acting as a server is not needed for most applications, it is important to secure network communication in both directions for all applications, including those acting as a server. [36]

- **Secure data transfer over unsecured channel** Despite the fact that SSL is almost always advised for network communications, there are situations when SSL is not available or stronger mechanisms are required. SecureBlackbox provides methods to encrypt the data before transferring it over to the unsecured channel by implementing two industry encryption mechanisms: OpenPGP and PKI. SecureBlackbox handles the entire communication process, including the key generation, signing, encryption and decryption. [36]

Thus, Solid File System and SecureBlackbox prevent network and insecure data storage attacks. Both components have commercial licenses that are quite affordable for medium-sized projects (Solid File System license fee starts at 250 € and SecureBlackBox will cost approximately 1000 € for a standard package). Additionally, all frameworks have free trials available.

However, the integration of frameworks might be problematic, especially for Solid File System, because it is not a drop-in replacement of the system framework, but rather a third-party library, which needs to be integrated separately. [37]

3.3 IMAS – iOS Mobile Application Security

IOS Mobile Application Security Project (IMAS) is not a cross-platform solution, such as Cryptanium or SecureBlackbox; its only target is iOS applications. It is an open source collection of security frameworks, controls and tutorials, which can be freely used without any fees. The project is sponsored by MITRE Corporation to produce open source iOS security controls and increase security awareness among iOS developers. Due to its unique characteristics and the fact that it can be freely used, it is not surprising that IMAS has acquired much attention from its inception; it has been widely referenced in the open source community and promoted by OWASP. [38, 39]

At the time this thesis was written, IMAS consisted of 10 subprojects with each project targeting a specific aspect of iOS applications security. Some of the projects are presented in a tutorial form, in which the accompanying text introduces base security concepts and the code is added only for illustrative purposes (e.g. *forced-inlining* repository [40]). There are also complete libraries that are meant to be integrated directly into the project (e.g. *encrypted-core-data* repository [41]).

The most popular IMAS security controls are:

- **Encrypted core data** As indicated in section 2.1.3.1, one of the most widely used iOS data storage solutions, Core Data, is internally managed in the plaintext SQLite database and thus creates sensitive data leakage vulnerability. IMAS proposes the solution to this problem, which is designed as a drop-in replacement framework for iOS Core Data. Its implementation is very similar to the native Core Data, but instead of accessing the SQLite database directly, all calls pass the SQLCipher interface, which encrypts and decrypts data on the fly. [41] SQLCipher is a third-party open-source extension to SQLite that provides a fully encrypted SQLite storage. [42] However, building up an advanced data storage solution similar to Core Data is not an easy task. The Encrypted Core Data project still has many issues and does not support all features of Core Data, such as ordered relationships or subqueries. [43]
- **Application-level file based keychain** iOS Keychain is the most secure data storage solution available for software developers, but it has one problem. Namely, application data stored in the iOS Keychain is not removed during the application uninstallation process. Given the fact that most iPhone owners only have four-digit passcodes or no passcode at all [47], data stored in the iOS keychain is potentially at risk. The IMAS Project mitigates that risk by providing an alternative implementation to the iOS keychain, which resides inside the application's documents folder. This file-based keychain is encrypted with a key and optional security questions that are provided by the application user. Accessing the keychain requires unlocking it by entering the passcode similarly to the device keychain. [44]
- **Debug and jailbreak checks** IMAS incorporates debug and jailbreak checks, similar to Cryptanium. However, it only provides methods to detect a jailbroken phone or an attached debugger, and does not call nor inject them automatically. The application author can decide, where and whether to call those methods. [45]
- **Secure memory** The IMAS Secure Memory framework offers a set of functions to clear up and encrypt sensitive instance variables or memory regions to decrease the possibility of revealing sensitive data through memory analysis. Additionally, it contains functions that will prevent malicious method tampering by tracking the

method's relative location in memory; if the location changes, it is probable that memory was manipulated. [46]

Ultimately, IMAS is a noteworthy open source project, which plays an important role in iOS application security. It has taken the approach of educating developers and giving them needed security controls or code samples. The developer must choose the necessary framework from the IMAS projects catalogue, analyse appropriateness, study its usage and finally integrate it into the project. Hence, IMAS projects provide useful accompanying materials that can ease the process of securing iOS applications and explain the importance of security controls, but that do not eliminate the necessity for a high degree of security awareness and motivation from its users.

3.4 Comparative analysis

All three of the projects mentioned above focus on securing iOS applications, but they all have different approaches, target security areas and licensing models. Table 1 concludes the main principles and security controls of those projects.

	IMAS	Cryptanium	EldoS projects: SecureBlackbox and Solid File System
Programming language	Mostly Objective-C, some C functions	C++	C++
License/cost	Apache License, Version 2.0, free to use	Commercial license, price depends on the project size	Commercial license SecureBlackbox standard license for 1 developer costs 934 € Solid File System lite package license for 1 developer costs 250 €
Open/closed source	Open source	Closed source	Closed source

Protection against client-side attacks	Yes (encrypted code data)	No	Yes (secure file system)
Protection against network attacks	No	No	Yes (secure network communication using SecureBlackbox)
Protection against binary attacks	Yes (jailbreak and integrity checkers)	Yes (jailbreak and integrity checkers)	No
Protection against runtime attacks	Yes (debugger checks and memory security)	Yes (debugger checks)	No
Easy to integrate (1 – 10)	Intermediate (6)	Quite easy (3)	Rather difficult (8)
Updates	Constant through github.com	No info about updates	Lifetime updates if purchased (+ 50% in price)
Conclusion	Suitable for start-ups and medium-sized projects that require specific security solutions (e.g. secure Core Data) and are willing to invest some time into software security.	An advanced code protection solution for large-scale banking/financial projects, which incur significant losses when an application is hacked. Not suitable for smaller projects due to an expensive license.	Advanced solutions for secure network communications, different Internet protocols and secure data storage. Suitable for medium-sized projects that have strong security requirements for data confidentiality.

Table 1: Comparison of iOS security solutions

The conclusions section from the Table 1 shows clearly that the security library choice depends solely on the project needs: networking solutions from the SecureBlackbox may be

enough for one project, but another might need to integrate both code protection from Cryptanium and encrypted core data from IMAS. However, what is currently missing is an all-in-one solution that would incorporate different types of security controls in one core framework, offer ways to extend its functionality by writing new security controls and that would be easy to integrate even for less security-aware novice developers. Given the relative novelty of the entire iOS development sphere, there is still insufficient research and related works available to address the different needs of application developers.

4. A methodology for securing iOS applications

By taking into account the theoretical background and related works in the field of iOS application security, the author of this thesis proposes a new iOS framework as a solution to raising security issues that arise from iOS applications. This section will provide insights into the conceptual working principles of the resulting framework.

4.1 Fundamental security principles

The default iOS security model has taken a huge step forward in the field of user protection by providing more advanced security mechanisms than those used in desktop computers. However, the fact that mobile devices are much easier to lose or steal [47] might mean that those precautions are not sufficient. Furthermore, there is almost no protection against debugging and reverse-engineering the application, which puts the results of a developers' hard work at risk for piracy and theft. The essence of Objective-C, especially its dynamic message dispatching, makes reverse-engineering and patching iOS applications very simple.

At the moment, whether or not to implement additional security controls to prevent common vulnerabilities is actually up to the application developer. iOS APIs implementation is focused primarily on the end user experience and although security is very important, it has become a secondary priority (e.g. security risks because of caching data for autocorrect, see section 2.1.3.4). The proposed solution follows a different approach; security should not be an opt-in feature, but the default option. The developer should not have to write extra code to make the application more secure, but code should be written to disable security controls. For example, adding the security framework would inject multiple debugger and jailbreak checks into the application's control flow automatically and it would be up to the developer to turn them off by changing the default configuration.

It is important to continue using default iOS APIs and frameworks as much as possible instead of offering custom drop in solutions in order to extend existing functionality. This is because writing custom solutions from the beginning is a very time consuming and error prone process. In addition, iOS development APIs are continually tested by millions of developers and have evolved over the years. New iOS versions and changes in system APIs can possibly break custom solutions. As an example, it is more reasonable to extend default

Core Data mechanisms by adding cryptography routines than to create a new solution in most cases. The security framework would thus take an existing iOS application model as a basis and extend it with security features.

4.2 Automatic or manual security

Without a doubt, not all security controls can be injected automatically, similar to debugger and jailbreak checks, due to multiple system and data restrictions. Indeed, no framework can anticipate with certainty, which data fields are important and must be protected in the context of a specific application. Protecting all fields would not be wise from a performance standpoint. An even worse scenario could occur with an iOS data storage solution Core Data, where automatic changes to the model data types might produce unwanted effects on the data model integrity because runtime changes would not be reflected in the GUI modelling tools.

Another example would be URL schemes whitelisting (see 2.1.3.5) because the application developer is the only person who knows which applications should be allowed to use URL schemes and which should not. Making whitelisting an automatic process based on some average parameters is not a solution due to differences between application architectures.

There are certainly more examples of situations in which automatically applying security controls would cause undesired effects. Essentially, this is a typical problem of putting too much trust into the software solution. The key to solving such problems is to use a more general-purpose configuration by default and to allow further configuration. As for security controls, which clearly require some manual setup, their integration must be done as transparently and easily as possible.

4.3 Patching and ease of integration

Finding a balance between ease of integration and preventing binary patching is another difficult task. To make patching more complex, security controls must be used in as many places as possible and they should overlap each other. The more these places must be modified to break the protection, the greater chance that a crucial mistake will be made and an attacker will give up early. On the other hand, requiring the developer to incorporate framework methods or use different framework classes throughout the application would make the integration process too troublesome.

The solution is to automatically inject needed security checkers into the program's control flow by using the dynamic features of Objective-C. However, the default methods for the checkers injection must be selected carefully. Obviously, performing the application integrity verification on every view state change would cause considerable performance overhead. Doing this on every new view appearance would be a more appropriate idea. The default framework configuration must provide a reasonable improvement to security without any noticeable or undesirable impact. Additionally, the framework user must have the opportunity to change the configuration at any moment in the application lifecycle.

4.4 Proposed security controls

Considering the previously described security principles, the following implementation for common security issues can be presented:

4.4.1 Binary patching

Protection mechanisms against binary patching would include:

- Validating that the binary is encrypted, as pirated binaries are usually distributed unencrypted;
- Checking the presence of the code signature;
- Validating that the Info.plist file is unchanged, because this file is usually modified in hacked applications.

All integrity checks will be injected automatically into various parts of the application code. Those can be disabled manually if necessary. All protections will be enabled only in the release mode so as not to disturb the application development process. Integrity checkers would provide a callback to implement custom application behaviour, if integrity anomalies are discovered. For example, the application might want to report the incident to developers or silently wipe user's data. The default behaviour would exit the program.

4.4.2 Runtime analysis

Similar to the defence solutions against binary patching, the framework will automatically inject debugger and jailbreak checks into the release version of the program. The framework will not rely on a single debugger or jailbreak detection method, but will attempt to use

multiple solutions to make deception of the checker more difficult for an attacker. Additionally, most important framework sections, such as checkers injection, will be protected against runtime tampering by validating the method's image origin. Similar to integrity protection, jailbreak checkers will integrate a custom callback system to report the application regarding a jailbroken device. Debugger checkers will simply exit the application if a debugger is attached, because there is almost no legitimate use of a debugger in a release build.

4.4.3 Insecure data storage

Insecure data storage issues will be solved using encryption algorithms. The selected encryption standard is AES256, which is also used by Apple for the device wide data encryption. [59] The framework will generate a random encryption key for each application installation and store it securely in the iOS Keychain. After the application removal, the encryption key will remain in the Keychain, but it will be useless without the application data. After the application reinstallation, the old encryption key will be removed and replaced with a newly generated key.

The encryption key will be used to automatically encrypt and decrypt any data that is saved to application preferences storage `NSUserDefaults`. The application developer can disable `NSUserDefaults` encryption at any time and for all data globally, or for a single data entry. When the `NSUserDefaults` encryption becomes globally disabled, data entries already stored in an encrypted form will remain encrypted until the first modification of their value. This would also apply to unencrypted values when the encryption becomes enabled somewhere in the middle of the application execution. Encrypting and decrypting all `NSUserDefaults` values during the global configuration change could cause performance problems. Moreover, automatically decrypting all data due to a single method call would make stealing the data easier, if the attacker manages to modify the global security framework configuration. Otherwise, the attacker would additionally need to find a way to decrypt data that has already been saved.

File encryption will use exactly the same principle, but there will be a threshold of the file size, which can be encrypted. The default threshold will be determined at runtime depending on the amount of the device's random-access memory, because encrypting files that are too large would negatively influence performance. Thus, if the file size exceeds the threshold, it

will be ignored and saved unencrypted. However, the threshold can be changed manually at any time. New settings will apply only to new files.

Core data encryption (see 2.1.3.1) will follow a different pattern. The application developer would need to select values that he wishes to encrypt in the graphical Core Data modelling tool and manually change their data types to the encrypted type that is provided by the security framework. The encrypted type will internally use Core Data Transformable attributes. Transformable is a special Core Data type, which allows transparent data encoding and decoding of database values. The main idea behind transformable attributes is that the developer can continue using ordinary types in code, but all values are internally transformed to binary data. [48] There is also no need for a separate Core Data encryption configuration mechanism, because it will be configured manually in the graphical modelling tool.

4.4.4 SSL attacks

SSL-related protection mechanisms would focus on solving improper SSL certificate validation and providing additional security by implementing SSL pinning techniques (described in section 2.2.2.2). SSL certificate validation problems are caused mainly by turning off the default system certificate validation for test environments. Therefore, prohibiting the ability to disable the certificate validation in the release mode could solve problems, especially when a developer forgets to comment the vulnerable code out before releasing the binary. However, a somewhat more sophisticated approach is required, because developers might need to make release builds against the test environment, so the aforementioned protection mechanism would break the application. Hence, the configuration should allow the ability to specify which endpoints require a correct certificate and which do not.

Proper implementation of the SSL pinning technique is a complex problem. The simplest way is to pin and validate the exact server certificate, but this will cause problems if the certificate expires. Another possibility is to pin to the root certificate, but this does not always ensure authenticity of the communication channel, especially in case of possible breaches with certificate authorities [51].

Pinning to the public key and to the related information (called SubjectPublicKeyInfo or SPKI) of a server certificate is considered to be the most flexible way to do SSL pinning, because the public key is usually fixed even after the certificate re-issuance. For example,

Google rotates server certificates every 30 days, but keeps public keys static. [54] Pinning only the public key without the related information can leave the client vulnerable to misinterpretation attacks. These types of attacks occur when an attacker uses the same public key, but manipulates the related information and makes the client trust it. [49]

4.4.5 Insecure data logging

Preventing insecure data logging is rather simple: the library will use a very popular method among iOS developers. The method includes defining a new function to use for data logging, which only works in the debug mode and ignores messages in the release mode. However, the framework will instead use the standard logging method NSLog, so changing logging functions everywhere would not be required. There will also be a way to enable certain message logging even in the release mode by using another logging function.

4.4.6 Text entry caching

Application developers generally use text fields in registration and login forms, so there is a significant chance that text fields will capture sensitive data such as usernames, contact information or personal codes. Preventing sensitive data caching by text entry methods requires turning off both autocorrect and pasteboard for those fields. The security framework will do this by default for all text fields in the application. However, it will be possible to disable this feature globally for all fields, for each individual field or for a view, which contains multiple fields. Disabling security restrictions for an individual field or a view will be implemented using runtime attributes, which can be changed both programmatically or in the graphical iOS interface builder to ease the integration.

4.4.7 Application screenshot

The iOS system takes a screenshot of the current view during the application transition to the background mode. This is done to create a feeling of an immediate application start, when the user returns to it. The latest screenshot is always saved to the application's documents folder and despite the fact that only one screenshot can be present at a time, deleted versions of screenshots can often be found in the HFS journal. [32] This means that it is highly probably that screenshots leak sensitive data, especially if registration or payment information is being entered.

The most popular solution for application screenshot vulnerability is to hide the main application window before suspending. As the main window is hidden, the only image that will be captured is a black screen. However, such a solution impacts the user experience, because the black screen, which is presented before the application launch, is not an expected behaviour of the iOS system.

The security framework will implement a different approach. Namely, all text fields, which may contain sensitive information, will be cleared during the application suspension. The data from the text fields will be encrypted and saved in memory, so it will be decrypted and inserted back into the appropriate text fields after the application relaunch. A security control such as this will prevent information from being captured on the screenshot and will also ensure information protection in memory. Sensitive fields will be determined based on the text entry caching prevention control; all fields on screen that are not marked as insecure, will be protected in the background mode.

4.4.8 Web content

Web content vulnerabilities, such as XSS and same origin policy issues, which are quite common in mobile websites, should be evaluated individually in most cases. Nevertheless, some typical mistakes can be eliminated. For example, many applications display local HTML content. The same origin policy works differently for local files and local HTML files are granted full access to all application data. This means that saving and presenting an untrusted HTML document locally creates a universal XSS vulnerability, allowing the attacker to access a corporate intranet, confidential application data or local cookies. Preventing that vulnerability is rather simple and requires setting the *baseURL* parameter to the request object. The security framework will check the parameter and set it to “about:blank”, if it is left unset.

For external connections, OWASP recommends prohibiting users from accessing arbitrary web content inside the application. [17] To achieve this, all requests must be whitelisted. The security framework will provide an easy whitelisting solution, which is based on the W3C Widget Access specification [52], and should be familiar to users of Apache Cordova, a popular technology for developing mobile web apps with native functionalities. [53] By using the wildcard identifier (*) it will be possible to combine request filters based on the protocol, host, resource or port. Withal, the web request's whitelisting functionality will require explicit

configuration of the whitelist due to the different needs of applications and that is why the default policy will allow all requests.

4.4.9 URL schemes

The security framework will check and whitelist the usage of URL schemes (see 2.1.3.5) and block requests from all origins not in the whitelist. The whitelist will be provided by the application developer as a filter for identifiers of source applications, which are allowed to use the URL scheme. Similar to the web content whitelisting, the default policy will allow all origins, because it is not possible to detect valid origins without additional configuration.

5. Implementation of the security framework

The technical output of this thesis is an iOS framework, which can be added to any other Xcode-based iOS project as a subproject or as a compiled framework. An iOS framework is a hierarchical directory, which may contain source code, headers, resources or shared libraries. Framework instances are shared among different applications if possible, because they are loaded into memory only when required by the application. The compiled framework binary is therefore provided separately from the application, but both framework loading and method invocations are incorporated into the application file. It is also possible to include multiple versions of a framework to ensure backward compatibility. [55]

The installation of an iOS framework is a somewhat straightforward process that can be done using Xcode GUI tools. Figure 7 shows a typical process of a framework addition to the project using Xcode 6.3.

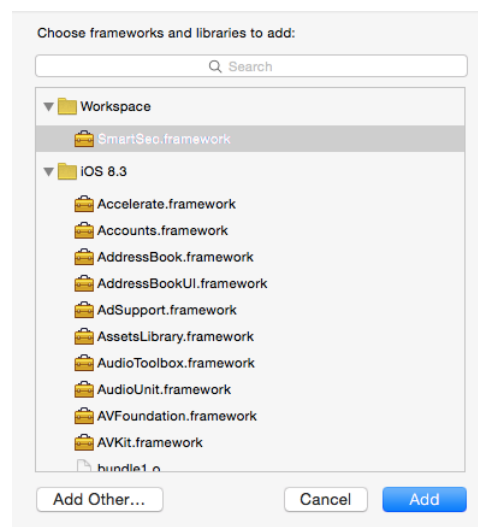


Figure 7: Adding an iOS framework in Xcode 6.3

5.1 Security framework architecture

The high-level model of the security framework architecture is depicted in Figure 8.

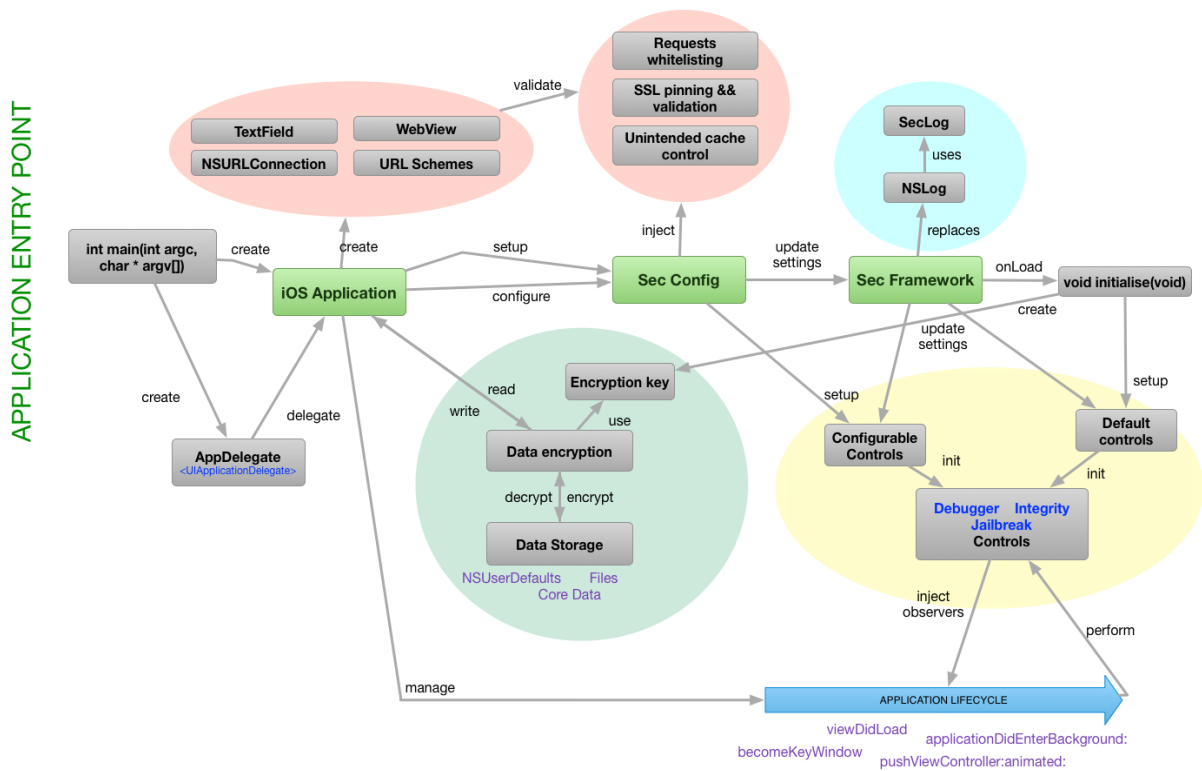


Figure 8: Security framework architecture

The lifecycle of an iOS application starts with the “main” function, which is the entry point for every Objective-C program. It is similar to main functions in C, Java and many other programming languages. The main function sets up an instance of the UIApplication class and assigns a developer-defined delegate to it. UIApplication is the system class that is responsible for managing application’s state changes. Both the UIApplication and its delegate are hereafter responsible for application lifecycle management.

The security framework methods are incorporated into various stages of the application lifecycle. All security controls can be broadly divided into three groups:

1. Application-independent controls, which can be set up without additional configuration or application context. As an example, detecting an attached debugger process or finding proofs of a device jailbreak relies entirely on system calls and does not need application specific input.
2. Application-independent controls, which require application context to function properly. These controls generally use system APIs, but address application specific

anomalies detection. This can be illustrated with an application encryption detection or by checking the ability of the application to write outside the sandbox.

3. Application-specific controls, which are activated only when the appropriate functionality exists. Those controls detect the initialisation of related classes and enable themselves by injecting the additional validation logic. Some controls are injected fully automatically and others require the proper configuration. Namely, if the application does not store any data locally, then the data storage encryption module will not be used. Similarly, a fully offline iOS application will not require networking and related SSL pinning mechanisms. However, the activation of the SSL pinning would require providing reference certificates or hashes.

Consequently, the organization of various security control types is done differently.

First, iOS frameworks are loaded into memory before the application is actually created, because they are needed from the very beginning of the application lifecycle. The dynamic loader, called dyld on iOS, processes the binary header to detect the needed frameworks to load. Additionally, the dynamic loader searches for specific framework initialization functions and calls them if found upon loading. That function must be named `initialise (void)` and should have an `__attribute__((constructor))` identifier. The `initialise` function is guaranteed to be called before any other framework class loads or method calls. [56]

Therefore, the `initialise` function is the proper place to setup multiple application-independent controls and perform any required preparations. The following actions are executed during the framework loading and initialisation:

1. The framework creates and injects multiple integrity, jailbreak and debugger controls.
2. The framework sets up application-wide encryption keys, which are later used for the data storage encryption. It communicates with iOS Keychain services throughout the encryption key setup process.
3. The framework loads the URL scheme and web content requests whitelists from the application settings file if configurations are present.
4. The framework modifies data writing and reading methods by adding encryption and decryption routines.

5. A new logging function called SecLog is defined, which internally uses an extended version of the system logging function NSLog. The standard version of the NSLog is replaced throughout the application with SecLog with a couple of pre-processing macros that configure logging based on the application running mode.

The next step in the Framework setup process follows the successful application launch and is triggered by the application developer, who calls the framework setup function. The setup function must be provided with an optional callback to the application session identifier. The optional session identifier is used to protect application wide encryption keys in memory, so they will not be stored unencrypted in a variable. The developer must ensure that the provided key is not static or hard-coded, but rather dynamically changing based on user authentication or session tokens. Implementing the callback is not crucial from the framework viewpoint, because otherwise encryption keys will be not saved in memory variables at all, but rather queried each time, when needed. However, memory storage greatly optimizes encryption performance. In addition to the direct setup, the invocation of framework functionality from the application context guarantees that the application launch is successfully finished and the application now has an active UIApplication instance, delegate and at least one main window. This allows not only the injection of remaining debugger, integrity and jailbreak checks, but it also allows them to run for the first time.

Finally, the setup of all other controls is performed on a needs basis. More specifically, the framework intercepts the creation of functionality related class instances and adds additional validation logic to the delegating class. Those controls include:

1. SSL certificate validation and SSL pinning for the NSURLConnection based networking (see 2.1.3.2);
2. Injection of web content and URL scheme whitelists;
3. Ensuring web content same origin policy;
4. Protection against text fields and application screenshots unintended data leakage;

5.1.1 Controls injection

Automatic controls and validation logic injection minimizes the amount of work needed to integrate security controls into an existing application, because it abolishes the necessity to

use subclasses for extending the objects functionality. Injection of controls is achieved by intercepting and modifying invocations of Cocoa Touch system methods or protocol implementations. In order to do this, the framework utilizes the Objective-C runtime and more specifically, a technique called method swizzling. This is the same method of runtime manipulation that is commonly used by iOS application attackers to modify or replace critical methods implementations in runtime. [57]

In the simplest case, the swizzling method can be achieved in three steps. Figure 9 illustrates the replacement of the `NSURLConnection` initialization method `initWithRequest:delegate:` with an another method `initWithRequestSwizzled:delegate:`. It first finds the original and the new method implementations. Thereafter, the methods' implementations are exchanged.

```
+ (void)load
{
    Method original, swizzled;

    //1. Find the original method
    original = class_getInstanceMethod(self,
    @selector(initWithRequest:delegate:));
    //2. Find the replacement method
    swizzled = class_getInstanceMethod(self,
    @selector(initWithRequestSwizzled:delegate:));
    //3. Exchange methods implementations
    method_exchangeImplementations(original, swizzled);
}
```

Figure 9: Simple method swizzling

The new method implementation is shown in the Figure 10. Essentially, the method performs SSL logic setup and then proceeds with the original method.

```
- (instancetype)initWithRequestSwizzled:(NSURLRequest *)request
delegate:(id)delegate
{
    // Add validation logic to the NSURLConnection delegate
    [self setupDelegate:delegate];
    // Call the original method
    return [self initWithRequestSwizzled:request
delegate:delegate];
}
```

Figure 10: New method implementation

At first glance, it might appear that the new method calls itself, leading to an infinite loop. However, as method implementations are exchanged, it becomes clear that the original method implementation now has a new name.

Method swizzling is internally achieved by manipulating the Objective-C class structure and its substructure `method_list`. Both Objective-C classes and methods are principally C structures. A method structure states its name, arguments and implementation, as shown in the Figure 11. During method swizzling, the methods' implementation pointers are exchanged. [58]

```
struct objc_method {  
    // Method's identifying name  
    SEL method_name;  
    // Method's arguments' types  
    char *method_types;  
    // Method's actual implementation == C function  
    IMP method_imp;  
};
```

Figure 11: Objective-C method structure

The framework architecture follows a similar 3-step approach to all controls injection:

1. Exchange the methods' implementations to direct method calls to the framework;
2. Add additional functionality (e.g. check application integrity) or modify the method's arguments (e.g. encrypt a string before writing);
3. Call the original method.

Consequently, the framework does not change any system API's behaviour, but only intercepts the methods' invocations, performs needed operations and proceeds with the original implementation.

Withal, the simplistic methods' exchange approach described above is not applicable everywhere, because it assumes that the target method exists. This assumption is always correct for system APIs, but is not valid for delegate classes, which might have optional methods. In that case, it is required to first check the existence of the target method and if it is present, exchange the implementations, and then add a new method with the original name. The framework implements such behaviour, when injecting whitelisting, SSL validation and text fields related controls.

Another problem might arise because of new Objective-C methods declaration. Objective-C methods that are declared for exchange are added to the Objective-C symbol table like any

other method. This is not an issue in most instances, but it simplifies the process of tracking down important framework parts. That is why a C-function based swizzling method is used for the important sections of the framework. The core principle is similar to the traditional method of swizzling, but instead of declaring a new Objective-C method, a C function is used. The implementation of the method in the structure is replaced manually with the new function, while the original implementation is cached in memory. Figure 12 illustrates that solution that is used for jailbreak and debugger checkers injection.

```
static IMP __original_DidLoad_IMP;

+ (void)load
{
    Method original;

    // Find the original method
    original = class_getInstanceMethod(self,
@selector(viewDidLoad));

    // Replace method's implementation.
    // method_setImplementation returns the previous
implementation. Save it for later usage
    __original_DidLoad_IMP = method_setImplementation(original,
(IMP)swizzledViewDidLoad);
}

void swizzledViewDidLoad(id self, SEL _cmd)
{
    // Call the original implementation
    ((void (*)(id,SEL))__original_DidLoad_IMP)(self, _cmd);

    // Notify event observing classes
    [[UIViewController class]
notifyObservers:NSStringFromSelector(@selector(viewDidLoad))
fromObservedObject:self];
}
```

Figure 12: Swizzling method with C functions

5.1.2 Runtime protection

The framework implementation relies heavily on the Objective-C runtime features, which are also often used by attackers. Without additional protection, a potential attacker will be able to exchange the methods' implementations back to original using similar techniques. Indeed, he would need to conduct a detailed study of the framework structure and find all of the intercepted methods, but it would simply be a matter of time before that is completed. In order to make those attacks more difficult, the security framework implements basic application runtime integrity protection.

Injecting malicious code into applications using Objective-C runtime techniques requires loading that code into the address space. The safeguard mechanism against malicious code loading should validate the address space of the critical application methods. In detail, the dynamic linker library function `dladdr` allows the image origin of any function to be obtained. In an iOS application, all legitimate code comes from Apple's frameworks, custom application frameworks or the application itself. If the function's image originates from any other location, the application's runtime is most likely being investigated or tampered with. Validation of the address space would force an attacker to inject malicious code into the existing address space, which is highly complicated in comparison with ordinary runtime attack methods. [21]

The framework incorporates automatic periodical checks of important framework sections and base Cocoa Touch classes as a part of application integrity protection. Moreover, it allows the application developer to check his own custom classes or methods to ensure that critical code sections were not altered. Appendix 1 demonstrates the implementation of the runtime protection solution.

5.2 Framework configuration

The central idea in the framework configuration is to provide default policies wherever possible, and simultaneously integrate extensive customization capabilities, because each iOS project has different security requirements. For instance, running on jailbroken devices might be unacceptable for mobile banking programs, while a file manager application would not only run on jailbroken devices, but also implement additional features for that target audience.

Therefore, each automatically activated control has a setting that will fully disable or enable it at any time. The only exception is logging functions, because those are configured during compile time using pre-processor macros. Additionally, some controls have custom settings, such as jailbreak detection callbacks or file encryption threshold settings, and it is possible to partially disable them. Table 2 summarises the available configurations for each framework control.

Control	Fully disable/enable the control	Partially disable the control	Additional configurations
Debugger checks	YES, using configuration functions	NO	-
Jailbreak checks	YES, using configuration functions	NO	Possible to set custom behaviour on jailbreak detection
Integrity checks	YES, using configuration functions	NO	Possible to set custom behaviour for missing encryption detection
NSUserDefaults encryption	YES, using configuration functions	YES, using custom saving methods to ignore encryption	-
File encryption	YES, using configuration functions	NO	Possible to set maximum file size that should support encryption
Core Data encryption	YES, using Xcode modelling tools	YES, through Xcode modelling tools	-
Text fields cache protection	YES, using configuration functions	Yes, by modifying runtime attributes in code or in Interface builder	-
iOS Screenshot protection	YES, using configuration functions	Yes, by modifying text fields cache settings	-
SSL pinning and SSL certificates validation	Yes, by providing no SSL settings	Yes, by changing SSL settings	-
Application logging	Yes, by importing or not importing framework logging headers	Yes, by importing framework logging headers only to selected files	-
Webview and URL scheme whitelisting	Yes, by providing no whitelist	Yes, by changing whitelist settings	-

Table 2: Framework configuration options

For most applications, default policies and the one line framework configuration will be enough. Applications that are more complex might need to configure the framework further during the application launch. It is rarely necessary to change framework configurations multiple times, but doing so might serve the purpose of confusing attackers. Indeed, if every developer configures the framework in his own way, understanding the security controls logic would immediately become more difficult.

Global framework settings are managed by the configuration module, which passes settings around other controls. All configuration methods are implemented as C functions, so they are not added to the Objective-C symbol table. A typical integration process of the framework can be concluded in 3 steps:

1. Include the framework into the project;
2. Add needed imports to the application classes;
3. Call the setup function with an optional session identifier callback.

In addition, the developer can optionally configure whitelists, setup database encryption and add runtime integrity checks. The complete integration guide is presented in Appendix 2.

6. Analysis of the framework

The golden rule of security states that a security solution must cost less than a potential loss of its breach. The most secure iOS application would be an empty application, which shows a white screen and does literally nothing. Of course, it would be worthless from an attacker's perspective as well. For that reason, it is important that any mobile security solution would not have a negative impact on application performance. A perfectly secure, and thus slow and unusable application would not be of interest to attackers anyway, because there would not be many daily users. This chapter will begin with an analysis of framework performance and size aspects and it will proceed with a case study of the framework benefits.

6.1 Impact on performance and binary size

The security solution of this thesis will add performance overhead in two ways; through data encryption and various anti-piracy controls, such as runtime integrity protection.

Overhead from encryption is usually considered to be inevitable, as long as the encryption algorithm is implemented using best practices. The security framework does not implement any custom encryption algorithms, but instead uses encryption routines provided by iOS in the encryption framework, called CommonCrypto. Apple's cryptography routines are implemented taking into account device hardware characteristics and thus ensure the best possible performance on iOS devices. Moreover, every iOS device has a dedicated AES256 crypto engine, which makes encryption even more efficient. [59]

In order to measure the performance overhead from data encryption, multiple tests were conducted. All tests were performed on two different devices: the latest iPhone 6 and a 3-year old iPhone 4S. Those devices were chosen because they have very different performance and hardware parameters. The iPhone 4S model is the oldest device, which is supported by the latest iOS 8 software and will most likely be dropped from support in the upcoming iOS 9 release. [60]

Table 3 summarises the results of the tests.

Test description	Test time without encryption (iPhone 6/iPhone 4S)	Test time with encryption (iPhone 6/iPhone 4S)	Average performance overhead
1000 NSUserDefaults operations (write and read)	0.314524 s / 0.975283 s	0.956544 s / 2.819648 s	2.9 – 3.1 times
1000 File writing operations (write and read)	4.161051 s / 7.137668 s	8.711572 s / 10.363644 s	1.4 - 2.1 times
1000 Core Data entities operations (write and read) with 6 attributes	5.642732 s / 7.461728 s	9.072054 s / 11.332890 s	1.5 – 1.6 times

Table 3: Encryption performance tests results

Table 3 shows that an average performance overhead per one thousand operations is 50-60% for core data operations, 40-110% for file writing operations and 190-210% for NSUserDefaults operations. The reason for the larger overhead for NSUserDefaults operations is the comparatively shorter base operation time, while data encryption takes equal time regardless of operation type. Considering the fact that is almost no application that writes thousands of NSUserDefaults entries, such an overhead is acceptable.

It is more difficult to measure overhead with regard to piracy checkers. The actual impact depends largely on the amount of user's interaction, as checkers are injected based on application state transitions, not time. With that said, in addition to an average control invocation time, it is also important to find an average control usage to gain a better overview of possible overhead. Table 4 shows that the overhead is rather unnoticeable in the context of a modern application.

Control type	Average time for control invocation (iPhone 6/iPhone 4S)	Average number of control invocations during the 10 minute usage*
Jailbreak controls	0.003149 s / 0.0068465 s	12
Debugger controls	0.000037 s / 0.000056 s	16
Binary integrity controls	0.0003055 s / 0.00253 s	11
Runtime integrity controls	0.0013635 s / 0.007963865 s	28

* Calculated by testing different applications. May vary largely for a different application usage style.

Table 4: Anti-piracy controls performance impact

Finally, the size of the framework is of great importance, because it directly influences application sales, especially in the case of a free app. Many people will ignore an interesting application if it is larger than the App Store over-the-air download limit and cannot be downloaded through mobile network. [61]

The main dilemma regarding the framework size is related to inline functions. The use of inline functions is a technique that is used throughout the framework to avoid having a single method to patch. Inline functions are good in terms of security, but aggressive copying of code pieces definitely makes the resulting binary bigger. However, as for the current state, the framework size of 475 KB should not be of great concern for an average iOS application size of 23 MB. [61] However, it may be an important point of optimization for future developments.

6.2 Case study: using the framework in a real-world application

In order to better analyse the security framework concept in a real-world context, a short case study of the framework integration will be conducted. The project that will be studied is an iPhone application in beta stage for the Kosmos IMAX Cinema in Tallinn. The main purpose of the study is to execute an application security review to ensure the appliance to best practice security requirements before the first official application release.

The Kosmos IMAX iOS application has a wide range of features, including login, registration, user ticket listing, movie programmes and tickets purchase. Most of the application functionality is based on the client-server model, where content is queried using multiple different APIs and displayed locally. However, the application also has a local layer of data storage to minimise the number of API queries.

The initial security audit for Kosmos IMAX focused mainly on the client application and is based on the OWASP mobile top 10 vulnerabilities list. [18] It was executed using 2 devices: a non-jailbroken iPhone 6 with iOS 8.2 and a jailbroken iPad Air with iOS 8.1.2. Jailbroken devices give more freedom in terms of security analysis, but many application vulnerabilities can be discovered and exploited on non-jailbroken devices as well.

The full security report is provided in Appendix 3. The security review revealed 14 issues in total, most of which were client side vulnerabilities, because that was the primary focus of the audit. However, the review also found an important API related flaw, which allowed bypassing authorisation mechanisms. All issues were reported to developers and subsequently fixed.

Among the issues mentioned above, there were many mistakes that were typical for iOS developers, such as saving sensitive data to plaintext data storage, hardcoding authentication tokens and logging too much information to the iOS console. Moreover, there were literally no binary protections, which allowed the manipulation of variables in runtime or debugging of the application. However, runtime manipulation is not considered the highest risk for this application, because of multiple strong API-side controls. For example, the security review report indicates the vulnerability about changing the contents of a tickets variable, which would potentially allow the generation of free tickets. Nevertheless, such an attack would probably fail the physical validation in the cinema using the QR reader with an order's database validation. A similar runtime manipulation attack could be used to change ticket' prices before proceeding to the payment screen. This would allow the user to buy cheaper tickets, but fortunately, such an attack fails the API-side validation upon the ticket purchase completion.

For the sake of analysing the benefits of the security framework, it was integrated into the Kosmos IMAX application. In conjunction with default configurations, some custom options were utilized, such as checking the integrity of order and a ticket's managing classes runtime,

disabling jailbreak checkers and encrypting user data related database fields. Furthermore, SSL pinning for the testing environment was added. The total integration process took less than two hours.

Integration of the framework resulted in a considerable improvement in the Kosmos IMAX application's security. Out of 14 issues, which were revealed during the security audit, integrating the framework fixed 10 of them. Issues that were not fixed were mostly related to API or application internal logic flaws (e.g. saving credentials locally) and could not conceptually be fixed with automated measures. Nevertheless, there was also a vulnerability that allowed the capture of sensitive bank account data on a screenshot, which could possibly be mitigated by a framework if an appropriate control existed.

It is clear that the integration of a single framework cannot solve all possible vulnerabilities, especially API related flaws. However, eliminating common client-side exploits already improves the overall system security notably. Moreover, each real-world framework integration example would provide valuable information about needed improvements and missing controls. By further analysing each integration sample and finding similarities between them, it would be possible to build a solid security framework, which would fix an even larger percentage of security flaws automatically.

It is worth noting that the importance of having such a framework increases when there are no financial, timing or knowledge opportunities for conducting security reviews, which is rather common for mobile projects. Integration of the security framework does not require a deep understanding of security principles and should therefore be possible even for novice developers. On the other hand, a developer who is more interested in security can use the framework as an educational reference for multiple security solutions.

6.3 Limitations and possible issues

The major purpose of this thesis was to provide a proof-of-concept security framework implementation and analyse its technical and conceptual viability. While this goal was successfully achieved, the project still has many limitations to overcome.

First, with the framework scaling and the addition of new controls, performance and optimisation aspects would become crucial. Possibly, the selection of control injection points

must be redesigned, because it would need to accommodate new controls, whereas existing logic would require further optimisation.

Another area of improvement is with controls, which currently require manual setup. Certainly, most of them cannot be automated due to conceptual limitations, such as requests whitelisting. However, there are also some controls, for which problems are mostly technical and can possibly be overcome by using novel solutions. As an example, Core Data currently requires manual encryption setup using Xcode modelling tools, but it might be possible to omit this step. Withal, it would require an immense additional research focused directly on the concrete technical problem.

One more potential improvement is related to randomizing the framework controls and injection points. Indeed, regardless of the solution merits, having similar framework invocation points for multiple projects would make overcoming security guards easier. There is a widespread problem of putting too much trust into a single solution; the more users there are, the greater the impact of its vulnerabilities. OpenSSL is a good example of a popular library, whose exploits have influenced millions of websites worldwide. [62] The proposed solution would be to randomise the implementation of controls and their invocation logic for each project. The general idea is to provide an Xcode plugin, which would take the current implementation as a basis, but generate each time a slightly different controls package and injections architecture. This can be easily achieved by having a large database of controls and by choosing a random combination whether automatically or manually.

Finally, no security framework can eliminate the necessity of a code obfuscation solution for Objective-C projects. Advanced features and the syntactic simplicity of Objective-C expose it to technically uncomplicated reverse-engineering attacks. While most of Objective-C obfuscators are premium class products, there are also advanced open-source alternatives that can be utilized by any security-concerned developer. [63]

7. Conclusions and future works

This thesis focuses on the issue of iOS applications security. More specifically, it researches the possibility of creating an all-in-one iOS security framework for solving common vulnerabilities in iOS applications. The selected methodology combines both a theoretical analysis of iOS applications security and a practical approach, which consists of creating a proof-of-concept iOS security framework and validating results from its usage.

The theoretical section describes common iOS applications security vulnerabilities and typical development patterns. It is shown that there are basically two primary reasons for weak iOS applications security: insufficient security mechanisms from the vendor (Apple) and poor security awareness of developers.

After completing the theoretical research, the author analyses different ways of mitigating security issues of iOS applications by investigating existing solutions in that field. Due to the fact that iOS development is a relatively new area for security concerns, there are still very few security-oriented projects that focus only on iOS. That is why two of the researched projects are actually cross-platform toolkits and only one project concentrates only on iOS applications.

The comparison of benefits and drawbacks of existing solutions provides a basis for defining core principles for the resulting security framework. The main idea concentrates on automating the integration of security controls into the existing project, so its implementation would require a low degree of security awareness or changes to the application architecture. Moreover, an initial combination of security controls that can be implemented has been defined. This includes countermeasures for insecure data storage, unintended data leakage and insufficient transport layer protection vulnerabilities. In addition, runtime and integrity protection mechanisms have been introduced.

The architecture of the resulting framework relies on advanced Objective-C runtime manipulation techniques, such as intercepting method invocation to modify its implementation. This allows the integration of most security controls to be automated and does not require class inheritance patterns to be changed. Similar runtime-based techniques are used for a different type of framework functionalities, which require additional manual

setup due to conceptual differences between applications. This makes manual setup of such controls easier. The framework provides extensive configuration capabilities to accommodate different requirements. As runtime manipulation is also quite widespread among attackers, the framework incorporates runtime integrity protection controls.

To indicate possible negative and positive effects of the framework integration, an analysis of performance, size and security impacts was conducted. Performance implications of the analysis were considered to be acceptable, but if new controls are added, further optimization would possibly be required. The current framework size of 475 KB is also small enough for modern iOS applications. For the sake of judging the direct benefits of the security framework, this thesis included a case study of the Kosmos IMAX iOS application. The case study conducted a black-box security review, which indicated 14 vulnerabilities, 13 of which were client-side issues. The framework integration mitigated 10 out of those 13 vulnerabilities, which clearly improved the overall security of the application, proving the viability of the framework and positive expectations for future development.

The results of this thesis are significant, because it demonstrates that current security solutions provided by the vendor are not sufficient, but it is possible to mitigate most of vulnerabilities using a standardized approach. The sample implementation of the security framework would be an important basis for future development and would provide a solid reference for security controls implementation.

As for future research, there are two directions that must be developed in parallel. First, new controls must be added and should be accompanied by performance and binary size optimisations. Moreover, there are currently multiple technical limitations, which increase amount of the necessary manual integration, such as enabling Core Data encryption. Solving those issues might require finding new techniques and slightly changing the overall framework architecture. The focus of the second direction is randomisation of the framework implementation. This would take the current implementation and architecture of the framework, and generate a different set of security controls and injection points for each project. Such a generator can be integrated into the iOS development environment as a third-party plugin to ensure seamless integration and take into account the specifics of each project.

Finally, it is important to understand that there is no protection mechanism that can ensure a bulletproof security. The goal of any security control is to slow the attacker down, so he will

eventually give up. Each additional safeguard would also stop less educated attackers who are not talented enough to break through the security. However, if there is enough financial motivation and educated human resources to attack the application, there is essentially no way to stop it. Nevertheless, making use of strong protection mechanisms and improving them over the time is the only way to resist attacks.

References

- [1] Ambika Choudhary Mahajan, Worldwide Active Smartphone Users Forecast 2014 – 2018: More Than 2 Billion By 2016 [REPORT]. DazeInfo, 18.12.2014 [WWW] <http://dazeinfo.com/2014/12/18/worldwide-smartphone-users-2014-2018-forecast-india-china-usa-report/> (14.02.2015)
- [2] Janus R. Nielsen, Which smartphone user are you? MYSecurityCenter Blog, 11.07.2013 [WWW] <http://blog.mysecuritycenter.com/2013/07/11/which-smartphone-user-are-you/> (14.02.2015)
- [3] Yuvraj Agarwal, Malcolm Hall, ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing, 2013 [Online] ACM Digital Library
- [4] Calum Macleod, Is that a hacker next to you?, IET Communications Engineer, 2007 [Online] IEEE Explore Digital Library
- [5] Christian D’Orazio, Kim-Kwang Raymond Choo, A generic process to identify vulnerabilities and design weaknesses in iOS healthcare apps, 48th Hawaii International Conference on System Sciences, 2015 [Online] IEEE Explore Digital Library
- [6] Jin Han, Qiang Yan, Debin Gao, Jianying Zhou, Robert Deng, Comparing Mobile Privacy Protection through Cross-Platform Applications, 20th Annual Network & Distributed System Security Symposium, 2012 [Online]
- [7] iOS Developer Salary (United States). Years of Experience. Payscale. [WWW] http://www.payscale.com/research/US/Job=iOS_Developer/Salary (17.02.2015)
- [8] Java Developer Salary (United States). Years of Experience. Payscale. [WWW] http://www.payscale.com/research/US/Job=Java_Developer/Salary (17.02.2015)
- [9] Software Developer Salary (United States). Years of Experience. Payscale. [WWW] http://www.payscale.com/research/US/Job=Software_Developer/Salary (17.02.2015)

- [10] Dan Rowinski, Among Mobile App Developers, The Middle Class Has Disappeared, 22.07.2014 [WWW] <http://readwrite.com/2014/07/22/app-developers-middle-class-opportunities> (17.02.2015)
- [11] Introducing Swift. Apple Inc [WWW] <https://developer.apple.com/swift/> (24.02.2015)
- [12] About Objective-C. Apple Inc [WWW] <https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html> (21.02.2015)
- [13] Manuel Egele, Christopher Kruegel†, Engin Kirda, Giovanni Vigna, PiOS: Detecting Privacy Leaks in iOS Applications, 2011 [Online] <https://iseclab.org/papers/egele-ndss11.pdf> (21.02.2015)
- [14] Cocoa Touch Frameworks. Apple Inc [WWW] <https://developer.apple.com/technologies/ios/cocoa-touch.html> (22.02.2015)
- [15] Hans-Eric Grönlund, Colin Francis, Shawn Grimes, Data Storage Recipes, 2012 [Online] SpringerLink Digital Library
- [16] NSFileManager Class Reference. Apple Inc [WWW] https://developer.apple.com/library/prerelease/ios/documentation/Cocoa/Reference/Foundation/Classes/NSFileManager_Class/index.html (22.02.2015)
- [17] iOS Developer Cheat Sheet. OWASP. [WWW] https://www.owasp.org/index.php/IOS_Developer_Cheat_Sheet (23.02.2015)
- [18] Projects/OWASP Mobile Security Project - Top Ten Mobile Risks. OWASP. [WWW] https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks (23.02.2015)
- [19] Jeff Kelley, Learn Cocoa Touch for iOS, 2012 [Online] SpringerLink Digital Library
- [20] NSURLRequest Class Reference. Apple Inc [WWW] https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSURLRequest_Class/index.html#/apple_ref/occ/instp/NSURLRequest/cachePolicy (23.02.2015)

- [21] Jonathan Zdziarski, Hacking and Securing iOS Applications, 2012 [Online] Safari Books Online
- [22] Rui Wang, Luyi Xing, XiaoFeng Wang, Shuo Chen, Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation, 2013 [Online] ACM Digital Library
- [23] SANS Software Security Institute, iOS Handling of URL Schemes May Lead to Identity Theft. Softpedia, 09.11.2010 [WWW] <http://archive.news.softpedia.com/news/iOS-Handling-of-URL-Schemes-May-Lead-to-Identity-Theft-165484.shtml> (24.02.2015)
- [24] UIWebView Class Reference. Apple Inc [WWW] https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWebView_Class/ (24.02.2015)
- [25] Ariel Sanchez, Personal banking apps leak info through phone. IOActive Labs Research, 08.01.2014 [WWW] <http://blog.ioactive.com/2014/01/personal-banking-apps-leak-info-through.html> (24.02.2015)
- [26] Jens Heider, Rachid El Khayari, iOS Keychain Weakness FAQ, Further Information on iOS Password Protection, Fraunhofer Institute for Secure Information Technology, 2012 [Online] https://www.sit.fraunhofer.de/fileadmin/dokumente/sonstiges/iPhone_keychain_faq.pdf (24.02.2015)
- [27] Encrypting and Hashing Data. Apple Inc [WWW] <https://developer.apple.com/library/ios/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html> (25.02.2015)
- [28] Goran Delac, Security Threats for Mobile Platforms, University of Zagreb, 2011 [Online] ACM Digital Library
- [29] Cracked iOS (iPhone, iPad) and Mac App Store (OS X) Apps and Books for Free - AppAddict. An example site offering cracked applications. [WWW] <https://www.appaddict.org> (25.02.2015)

- [30] Roi Saltzman, Adi Sharabani, Active Man in the Middle Attacks. A SECURITY ADVISORY. IBM Rational Application Security Group, 2009 [Online]
<http://blog.watchfire.com/amitm.pdf> (01.03.2015)
- [31] Mobile Application Integrity Protection Handbook. DEFEND AGAINST APP INTEGRITY RISKS & ATTACKS. Arxan Technologies, 2013 [Online]
http://www.arxan.com/assets/1/7/Mobile_App_Integrity_Protection_Handbook.pdf
(04.03.2015)
- [32] Dominic Chell, Tyrone Erasmus, Jon Lindsay, Shaun Colley, Ollie Whitehouse, The Mobile Application Hacker's Handbook, 2015 [Online] Google Books
- [33] Cryptanium Overview. White Paper. WhiteCryption, 2015 [Online]
<http://static1.squarespace.com/static/53ffba10e4b034368de43c27/t/54e535dae4b0b74ca98ff417/1424307674026/Overview.pdf> (12.03.2015)
- [34] CRYPTANIUM CODE PROTECTION Brochure. WhiteCryption, 2015 [WWW]
<http://static1.squarespace.com/static/53ffba10e4b034368de43c27/t/54234761e4b0b3ca2a23251c/1411598177686/Code-Protection-Datasheet.pdf> (11.03.2015)
- [35] Financial Institutions are Prime Targets for Cybercriminals. The Nine Application Vulnerabilities that Need Your Attention. WhiteCryption, 2015 [Online]
<http://static1.squarespace.com/static/53ffba10e4b034368de43c27/t/54932b69e4b05dca90b49b33/1418931049615/banking-f%5B1%5D.pdf> (12.03.2015)
- [36] Solutions for software developers. EldoS Corporation. [WWW]
<https://www.eldos.com/solutions/solutions-for-developers.php> (13.03.2015)
- [37] Solid File System. Virtual file system engine that can be embedded into your software. EldoS Corporation. [WWW] <https://www.eldos.com/solfs/std-benefits.php#product>
(13.03.2015)
- [38] OWASP iMAS iOS Mobile Application Security Project. Owasp.org [WWW]
https://www.owasp.org/index.php/OWASP_iMAS_iOS_Mobile_Application_Security_Project
(13.03.2015)

- [39] iOS Mobile Application Security. Defense for your iOS App. IMAS Project's official webpage [WWW] <http://project-imas.github.io/> (14.03.2015)
- [40] iMAS forced-inlining. IMAS Project [WWW] <https://github.com/project-imas/forced-inlining> (14.03.2015)
- [41] iOS Core Data encrypted SQLite store using SQLCipher. IMAS Project [WWW] <https://github.com/project-imas/encrypted-core-data> (14.03.2015)
- [42] SQLCipher. Full Database Encryption for SQLite [WWW] <https://www.zetetic.net/sqlcipher/> (14.03.2015)
- [43] Encrypted Core Data issues. IMAS Project [WWW] <https://github.com/project-imas/encrypted-core-data/issues> (14.03.2015)
- [44] Secure Foundation. IMAS Project [WWW] <https://github.com/project-imas/securefoundation/> (14.03.2015)
- [45] Application level, attached debug detect and jailbreak checking [WWW] <https://github.com/project-imas/security-check> (14.03.2015)
- [46] Tools for securely clearing and validating iOS application memory [WWW] <https://github.com/project-imas/memory-security> (14.03.2015)
- [47] Smart phone thefts rose to 3.1 million last year. ConsumerReports, 28.05.2014 [WWW] <http://www.consumerreports.org/cro/news/2014/04/smart-phone-thefts-rose-to-3-1-million-last-year/index.htm> (14.03.2015)
- [48] Core Data Programming Guide. Non-Standard Persistent Attributes. Apple Inc [WWW] <https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/Articles/cdNSAttributes.html> (16.03.2015)
- [49] Public Key Pinning, Adam Langley, Google, 2011 [WWW] <https://www.imperialviolet.org/2011/05/04/pinning.html> (17.03.2015)
- [50] IOS Application Security Testing Cheat Sheet. Owasp.org [WWW] https://www.owasp.org/index.php/IOS_Application_Security_Testing_Cheat_Sheet (04.04.2015)

- [51] Internet Security under Attack: The Undermining of Digital Certificates. Neal Leavitt, 2011 [Online] IEEE Explore Digital Library
- [52] Widget Access Request Policy. W3C Recommendation 7 February 2012. W3.org [WWW] <http://www.w3.org/TR/widgets-access/> (06.04.2015)
- [53] Apache Cordova Documentation. Whitelisting Guide. [WWW] https://cordova.apache.org/docs/en/4.0.0/guide_appdev_whitelist_index.md.html
- [54] Certification Practices Statement. Google Inc. [WWW] <https://static.googleusercontent.com/media/pki.google.com/et//GIAG2-CPS-1.0.pdf> (11.04.2015)
- [55] Framework Programming Guide. Apple Inc [WWW] https://developer.apple.com/library/ios/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html#//apple_ref/doc/uid/20002303-BBCEIJFI (14.04.2015)
- [56] Mach-O Programming Topics. Apple Inc [WWW] https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachOTopics/Mach-O_Programming.pdf (16.04.2015)
- [57] Hacking iOS on the Run: Using Cycrypt. Sebastián Guerrero, RSA Conference 2014 [WWW] http://www.rsaconference.com/writable/presentations/file_upload/hta-r04a-hacking-ios-on-the-run-using-cycrypt.pdf (18.04.2015)
- [58] Objective-C Runtime Programming Guide. Apple Inc [WWW] <https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html> (19.04.2015)
- [59] iOS Security Guide, iOS 8.1 or later. Apple Inc [WWW] https://www.apple.com/business/docs/iOS_Security_Guide.pdf (19.04.2015)
- [60] iOS 8 Overview. Apple Inc [WWW] <https://www.apple.com/ios/whats-new/> (20.04.2015)

[61] Dave Wooldridge, Taylor Pierce, The Business of iOS App Development: For iPhone, iPad and iPod touch, 2014 [Online] Springer Books

[62] Thomas Wadlow, Who Must You Trust?, 2014 [Online] ACM Digital Library

[63] Protecting iOS Applications. Polidea Sp, author of the iOS class guard, open-source Objective-C obfuscation tool [WWW]

http://www.polidea.com/#!heartbeat/blog/Protecting_iOS_Applications (23.04.2015)

Appendix 1: Runtime integrity protection

```
//
// MemoryCheck.h
// SmartSec
//
// Created by Olga Dalton on 12/04/15.
// Copyright (c) 2015 Olga Dalton. All rights reserved.
//

#import <Foundation/Foundation.h>

extern BOOL checkClassHooked(char * class_name);
extern BOOL checkClassHookedWithAllMethods(char * class_name);

extern BOOL checkClassMethodHooked(char * class_name, SEL methodSelector);

//
// MemoryCheck.m
// SmartSec
//
// Created by Olga Dalton on 12/04/15.
// Copyright (c) 2015 Olga Dalton. All rights reserved.
//
// Based on:
//
// The Mobile Application Hacker's Handbook
// By Dominic Chell, Tyrone Erasmus, Jon Lindsay, Shaun Colley, Ollie
// Whitehouse
//
https://books.google.ee/books?id=5gVhBgAAQBAJ&pg=PA149&hl=et&source=gbs\_toc\_r&cad=3#v=onepage&q&f=false

#import "MemoryCheck.h"
#import "Defines.h"
#import "LOCCryptString.h"

#import <dlfcn.h>
#import <objc/runtime.h>

// Declarations

FORCE_INLINE int endsWith(const char *str, const char *suffix);
FORCE_INLINE BOOL checkMethodImplementationHooked(IMP methodImp);
FORCE_INLINE BOOL checkClassHookedWithConfig(char * class_name, BOOL
checkAllMethods);

// Implementations

extern FORCE_INLINE BOOL checkClassMethodHooked(char * class_name, SEL
methodSelector)
{
    IMP methodImp =
class_getMethodImplementation(objc_getClass(class_name), methodSelector);
    return checkMethodImplementationHooked(methodImp);
}
```

```

extern FORCE_INLINE BOOL checkClassHooked(char * class_name)
{
    return checkClassHookedWithConfig(class_name, NO);
}

extern FORCE_INLINE BOOL checkClassHookedWithAllMethods(char * class_name)
{
    return checkClassHookedWithConfig(class_name, YES);
}

BOOL checkClassHookedWithConfig(char * class_name, BOOL checkAllMethods)
{
    Class aClass = objc_getClass(class_name);
    Method *methods;
    unsigned int nMethods;

    IMP methodimp;
    Method m;
    if (!aClass) return NO;

    methods = class_copyMethodList(aClass, &nMethods);

    int max = (int)(nMethods / 20);

    // Pass through all class methods
    // If checkAllMethods == NO, select methods to check randomly
    for (int i = 0; i < nMethods; i+= (checkAllMethods ? 1 :
(MAX((int)ceilf(arc4random()%(max ? max : 1)), 1))))
    {
        m = methods[i];

        methodimp = (void *) method_getImplementation(m);

        if (checkMethodImplementationHooked(methodimp))
        {
            free(methods);
            return YES;
        }
    }

    free(methods);
    return NO;
}

BOOL checkMethodImplementationHooked(IMP methodimp)
{
    if (!methodimp)
    {
        return NO;
    }

    Dl_info info;

    // Query DL_info from method implementation using dladdr
    int d = dladdr((const void *) methodimp, &info);

    if (!d)
    {
        // Something terribly wrong
        return YES;
    }
}

```

```

    }

    // Check image origin against legit origins
    if (strstr(info.dli_fname, [LOO_CRYPT_STR_N("/usr/lib/", 9)
UTF8String]))
    {
        return NO;
    }

    if (strstr(info.dli_fname,
[LOO_CRYPT_STR_N("/System/Library/Frameworks/", 27) UTF8String]))
    {
        return NO;
    }

    if (strstr(info.dli_fname,
[LOO_CRYPT_STR_N("/System/Library/PrivateFrameworks/", 34) UTF8String]))
    {
        return NO;
    }

    if (strstr(info.dli_fname,
[LOO_CRYPT_STR_N("/System/Library/Accessibility", 29) UTF8String]))
    {
        return NO;
    }

    if (strstr(info.dli_fname,
[LOO_CRYPT_STR_N("/System/Library/TextInput", 25) UTF8String]))
    {
        return NO;
    }

    // Compose application path
    char appPath[512];
    snprintf(appPath, sizeof(appPath), "%s/%s/",
        [[[NSBundle mainBundle] resourcePath] UTF8String],
        [[[NSBundle mainBundle]
objectForInfoDictionaryKey:@"CFBundleName"] UTF8String]);

    if (endsWith(info.dli_fname, appPath) == 1)
    {
        return NO;
    }

    char appPathShort[512];

    snprintf(appPathShort, sizeof(appPathShort), "%s/%s",
        [[[NSBundle mainBundle] resourcePath] UTF8String],
        [[[NSBundle mainBundle]
objectForInfoDictionaryKey:@"CFBundleName"] UTF8String]);

    if (endsWith(info.dli_fname, appPathShort) == 1)
    {
        return NO;
    }

    // Check that a swizzled method origins from the security framework
    if (endsWith(info.dli_fname,
[LOO_CRYPT_STR_N("/SmartSec.framework/SmartSec", 28) UTF8String])
        || endsWith(info.dli_fname,

```

```

[LOO_CRYPT_STR_N("/SmartSec.framework/SmartSec/", 29) UTF8String])
{
    return NO;
}

if (info.dli_fname)
{
    // At this point we should have mached at least something!
    // If nobody is swizzling methods of course
    return YES;
}

return NO;
}

int endsWith(const char *str, const char *suffix)
{
    if (!str || !suffix)
        return 0;
    size_t lenstr = strlen(str);
    size_t lensuffix = strlen(suffix);
    if (lensuffix > lenstr)
        return 0;
    return strncmp(str + lenstr - lensuffix, suffix, lensuffix) == 0;
}

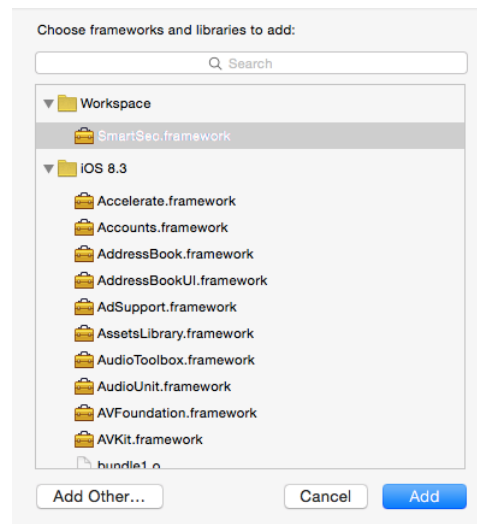
```


Appendix 2: Framework implementation guide

Basic configuration

1. Add the framework to your project

- Copy this repository
- Drag the SmartSec.xcodeproj somewhere into your project
- Navigate to Build phases -> Link binary With libraries and add the SmartSec.framework from the Workspace group



2. Add needed imports

- Open your prefix file (YourProjectName.pch) and add following lines

```
#import <SmartSec/SecImports.h>  
#import <SmartSec/Crypto.h>
```

- Open your AppDelegate file and add the framework import:

```
#import <SmartSec/SmartSec.h>
```

3. Setup the framework

- Add framework setup into the application:didFinishLaunchingWithOptions: method (or any other suitable place):

```

startSecurityFramework(^NSData *{
    return [User currentUser].sessionId;
});

```

That is it for the basic configuration.

Advanced configuration

1. Choose the needed controls

Each control has a way to fully or partially disable/enable it. Additionally, some controls have custom settings, such jailbreak detection callbacks or file encryption threshold settings, and the possibility to partially disable it.

All global settings are described in the SmartSecConfig header comments:

```

@interface SmartSecConfig : NSObject

/***** Callbacks *****/

// Set jailbreak callback
// It will be called upon discovering the device jailbreak
// If the jailbreak callback is not provided,
// the jailbreak detection will exit the application
extern void onJailbreakDetected(OnJailbreakDetected jailbreakDetected);

// Integrity encryption check callback
// It will be called, if the application binary is not encrypted
// It is usually the case for debug builds or cracked applications
// Encryption will not be checked in Debug mode though
extern void onMissingEncryption(OnEncryptionMissingDetected
missingEncryptionDetected);

/***** Settings *****/

// Debugger - enable/disable all possible debugger checks
extern void enableDebuggerChecks();
extern void disableDebuggerChecks();

// Jailbreak - enable/disable all possible jailbreak checks
extern void enableJailbreakChecks();
extern void disableJailbreakChecks();

// Integrity - enable/disable all possible integrity checks,
// including encryption detection check
extern void enableIntegrityChecks();
extern void disableIntegrityChecks();

// Disable controls partially for a specific subclass
extern void disableOnLoadControls(UIViewController *obj);

// UserDefaults encryption - enable/disable UserDefaults encryption

```

```

globally
// If disabled, already encrypted values will stay encrypted until value
rewriting
// Encrypted values will be retrieved normally, even if encryption is
disabled
extern void enableNSUserDefaultsEncryption();
extern void disableNSUserDefaultsEncryption();

// File encryption - enable/disable encryption for data/string/object
writing methods
// Already encrypted values will stay encrypted, if disabled
// Encrypts only data, which does not exceed threshold size
extern void enableFileEncryption();
extern void disableFileEncryption();

// File encryption settings
// Update file encryption threshold size
extern unsigned long long getThresholdFileSize();
extern void setThresholdFileSize(unsigned long long newFileSize);

// Textfields settings
// Enable/disable text fields securing globally for all fields
extern void disableSecureTextfields();
extern void enableSecureTextfields();

// Screenshots settings
// Enable/disable screenshots text fields protection globally
extern void disableAppScreenshotsProtection();
extern void enableAppScreenshotsProtection();

// SSL certificates validation config
// Set SSL certificates, which are allowed to fail validation
// It is useful for test environments,
// but it is highly recommended to setup SSL pinning for such certificates
even in test mode
extern void allowInvalidCertificatesInTestMode(NSArray *domains);
extern void allowInvalidCertificatesInReleaseMode(NSArray *domains);

// SSL pinning config
// Setup SSL certificates to pin
// The input dictionary should have target hosts as keys
// and embedded certificate path or certificate public key + related
information hash as values
// The hash way is recommended, but hide the hash string!
// You can set multiple certificates for one host

/*

Example configuration with embedded certificate path and hash combined:

NSMutableDictionary *sslPinDictionary = @{@"twitter.com" :
    @[[[NSBundle mainBundle] pathForResource:@"random-org"
ofType:@"der"],
@"cfb6fe515a13f0f84e058865c62087e890d8f0ea9d6723f8fc6a2193d29ced51"]];

pinSSLCertificatesWithDictionary(sslPinDictionary);

*/

extern void pinSSLCertificatesWithDictionary(NSMutableDictionary

```

```

*sslPinningDictionary);

/***** Setuping the framework *****/

// sessionPasswordCallback is an optional callback,
// which should return some dynamically changing password, associated with
a current user
// It is used for encryption keys memory protection

/*

Example configuration:

startSecurityFramework(^NSData *{
    return [User currentUser].sessionId;
});

*/

extern void startSecurityFramework(OnSessionPasswordRequired
sessionPasswordCallback);

@end

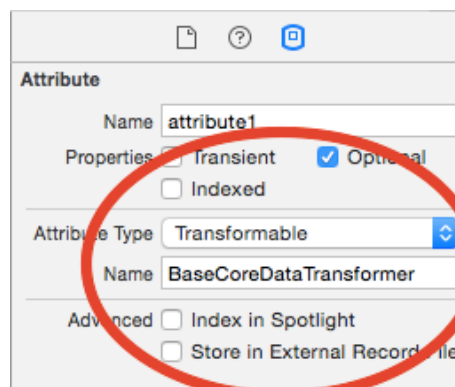
```

2. Setup Core Data encryption

- Open your data model and select attributes that you'd like to encrypt. Change their data types Transformable:



- Open the data model inspector and set transformer for the selected attribute to BaseCoreDataTransformer:



Important! Make sure that you're not using scalar datatypes in NSManagedObject subclasses. Scalar types cannot be used as Transformable attributes.

3. Setup whitelists

OWASP recommends to prohibit users to access arbitrary web content inside the application. To achieve this, all requests must be whitelisted. The security framework implements a whitelisting solution, which is based on the W3C Widget Access specification and should be familiar to users of Apache Cordova. By using wildcard identifier you can define complex request filters. Same whitelisting will also work for URL schemes source applications.

To setup whitelists add `WebAccessWhiteList` and `URLSchemesAccessWhiteList` arrays to the `Info.plist` file.

Example whitelist:

▼ WebAccessWhiteList	↕	Array	(2 items)
Item 0		String	http://*.com/*
Item 1		String	*://*.ee/*
▼ URLSchemesAccessWhiteList	↕	Array	(1 item)
Item 0		String	com.*.mobilesafari

4. Setup logging

The framework will automatically disable NSLog logging in the release mode. If you still need to log something in the release mode, use `ReleaseLog(...)` function instead. Both are defined using pre-processing macros - to skip this control, skip the `<SmartSec/SecImports.h>` import.

5. Setup textfields

Text fields, which are not used for sensitive information entry, should be marked as insecure. To do this, set the `insecureEntry` property of the text field, its superview or view controller to YES. Insecure text fields will not be masked on the background screenshot.

6. Configure SSL validation && pinning

SSL pinning works for `NSURLConnection` based requests. In order to configure it, you must provide whether embedded certificate path or the hash of the public certificate SPKI. The hash is the recommended way, but make sure you hide the hash string. You can provide multiple certificates for one host.

Example configuration:

```

NSString *certPath = [[NSBundle mainBundle] pathForResource:@"random-org"
ofType:@"der"];

NSString *certHash =
@"cfb6fe515a13f0f84e058865c62087e890d8f0ea9d6723f8fc6a2193d29ced51";

NSDictionary *sslPinDictionary = @{@"twitter.com" :
                                     @[certPath, certHash]};

```

To use self-signed certificates, use following functions:

```

extern void allowInvalidCertificatesInTestMode(NSArray *domains);
extern void allowInvalidCertificatesInReleaseMode(NSArray *domains);

```

The purpose of having different certificates for test and release mode is to avoid forgetting to remove self-signed certificate configuration code, when doing application release.

7. Add runtime integrity checks

You can optionally add runtime integrity checks for your custom classes to protect against method swizzling in runtime.

To do it, use following functions:

```

// Select randomly class methods and validate each method origin
// Returns YES, if method's image origin is unexpected
extern BOOL checkClassHooked(char * class_name);

// Same as previous, but validate each and every method
// Returns YES, if method's image origin is unexpected
extern BOOL checkClassHookedWithAllMethods(char * class_name);

// Validate a specific method for a specific class
// Returns YES, if method's image origin is unexpected
extern BOOL checkClassMethodHooked(char * class_name, SEL methodSelector);

```

For further configuration please refer to the example project.

Appendix 3: Kosmos IMAX security review report

This short security review focuses mainly on client side issues of the iOS application Kosmos IMAX, but also reveals some API flaws. It was carried out using 2 devices: a non-jailbroken iPhone 6 with iOS 8.2 and a jailbroken iPad Air with iOS 8.1.2. The security review uses the black-box analysis method of the beta version of the iOS application. It is an attempt to reproduce a scenario, where an attacker gets the application from the official App Store and does its black-box assessment. The review procedure follows the OWASP Mobile TOP 10 checklist. [18]

The analysis found 14 issues in total, which altogether expose the application to serious attacks.

M1: Weak Server Side Controls

Vulnerability 1: Facebook login API allows to login as an arbitrary person

Likelihood: High

Impact: High

A separate API component that allows logging in through Facebook requires the client to provide user's Facebook identifier and does not check, whether the client really has rights to access that account. API should switch to Facebook authentication token and validate user's identity using Facebook Graph API.

If an existing user hasn't yet connected his account with Facebook, API allows doing it by asking user's member identifier and Facebook identifier. The API does not validate, whether the authenticated Facebook user has rights to connect with a requested user account. The API should ask an active session identifier instead or together with member identifier to validate this.

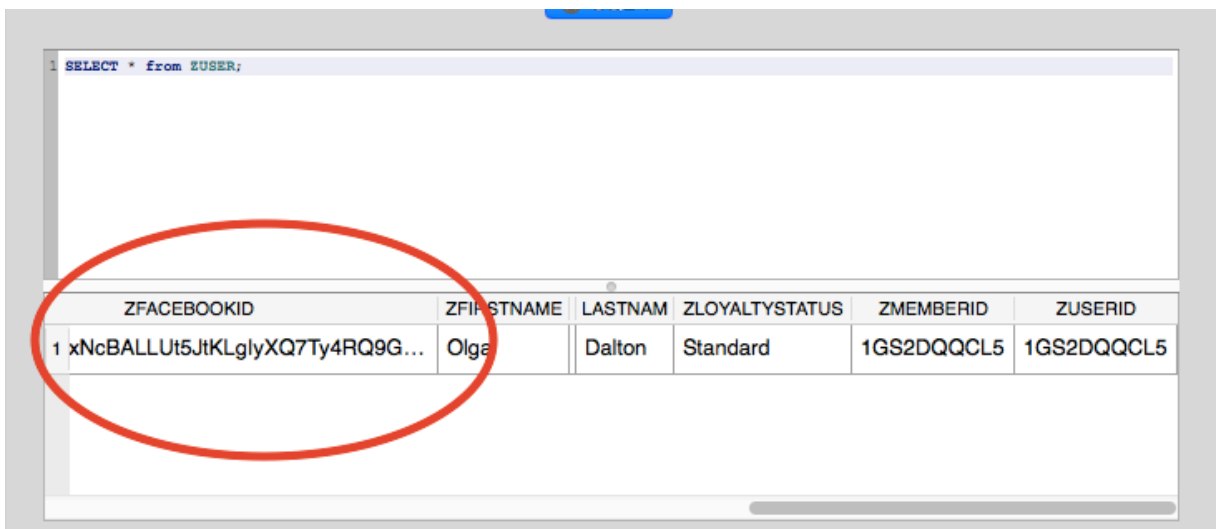
M2: Insecure Data Storage

Vulnerability 2: User's Facebook access token stored as plaintext in Core Data

Likelihood: Medium

Impact: High

If a user logs in through Facebook, his authentication token will be stored along user account details in the Core Data as plaintext entry. A stolen Facebook authentication token would give a temporary access to user's Facebook account, allowing malicious programs to authenticate with third party services as that user.



The image shows a screenshot of a database query interface. At the top, a SQL query is entered: `1 SELECT * from ZUSER;`. Below the query, a table of results is displayed. The table has six columns: ZFACEBOOKID, ZFIRSTNAME, LASTNAM, ZLOYALTYSTATUS, ZMEMBERID, and ZUSERID. The first row of data is circled in red and contains the following values: 1, xNcBALLU15JtKlglyXQ7Ty4RQ9G..., Olga, Dalton, Standard, 1GS2DQQCL5, and 1GS2DQQCL5.

ZFACEBOOKID	ZFIRSTNAME	LASTNAM	ZLOYALTYSTATUS	ZMEMBERID	ZUSERID
1 xNcBALLU15JtKlglyXQ7Ty4RQ9G...	Olga	Dalton	Standard	1GS2DQQCL5	1GS2DQQCL5

Vulnerability 3: User's account details, such as e-mail and member identifier, stored as plaintext in Core Data

Likelihood: Medium

Impact: Low

Account details for a logged in user are stored in Core Data as a plaintext entry, similarly to the Facebook authentication token. However, this is considered to be a low-importance issue, as user's member identifier does not give access to user's account or any highly sensitive data (given number 1 gets fixed). It can only be used to query user's movie ratings or purchases history.

SELECT * FROM ZUSER;

ZFACEBOOKID	ZFIRSTNAME	ZLASTNAME	ZLOYALTYSTATUS	ZMEMBERID	ZUSERID
10BALLU15JtKlglyXQ7Ty4RQ9G...	Olga	Dalton	Standard	1GS2DQQCL5	1GS2DQQCL5

Vulnerability 4: Username and password are stored in the iOS keychain and remain there even after the application uninstall

Likelihood: Very low

Impact: High

The application stores login credentials in the iOS keychain to automatically extend the expired session identifier. IOS keychain is the most secure way to store data, but it is still better to not store any credentials on device, if it is not an absolute must. Furthermore, retaining this data even after the application removal poses a security risk (e.g. in case of a device sale or theft).

```

Generic Password
-----
Service: Kino Kosmos
Account: UserName
Entitlement Group: K69T6NLJX8.com.mdigital.kosmos
Label: (null)
Generic Field: (null)
Keychain Data: olgadalton@gmail.com

Generic Password
-----
Service: Kino Kosmos
Account: Password
Entitlement Group: K69T6NLJX8.com.mdigital.kosmos
Label: (null)
Generic Field: (null)
Keychain Data: test1234

```

M3: Insufficient Transport Layer Protection

Vulnerability 5: Application accepts invalid and self-signed certificates

Likelihood: High

Impact: High

Application accepts a self-signed and expired certificate for every domain, even though the recognition of self-signed certificates is needed only for the test environment of the payment API.

M4: Unintended Data Leakage

Vulnerability 6: Sensitive data leakage through keyboard cache

Likelihood: Low

Impact: Medium

Text fields from the login and register screens have both autocorrect and copy-paste functionalities enabled. Both login and register screens may occasionally leak sensitive data, like usernames and personal codes.

Here are contents of the binary keyboard cache file (cleared before testing), which reveals the username entered into the login screen:

0	44796E61 6D696344 69637469 6F6E6172 792D3500 00000000 00000000 00000000	DynamicDictionary-5
32	01010047 6D61696C 00010100 68756765 00010100 48756E00 01006B69 72730001	Gmail huge Hun kirs
64	006B7269 73657300 01006E68 68686800 01006F6C 67617573 65726E61 6D650002	krisen nhhhh olgausername
96	006F6C67 6100	olga

And contents of the pasteboard, which has the copied username entry:

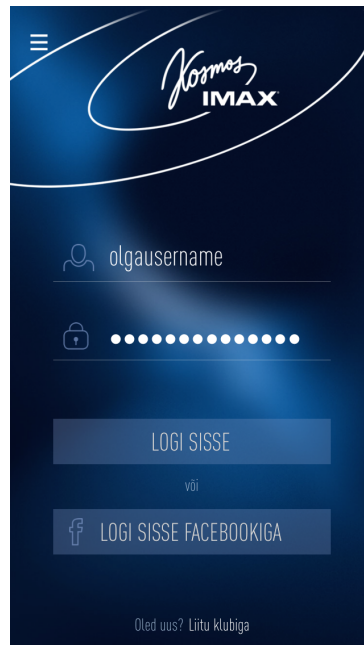
448	6E617475 72616C5C 70617274 69676874 656E6661 63746F72 300A0A5C 66305C66	natural\partightenfactor0 \f0\f
480	73323420 5C636630 206F6C67 61757365 726E616D 657D0100 00002300 00000100	s24 \cf0 olgausername} #
512	00000700 00005458 542E7274 66100000 0059FB37 55B60100 00000000 00000000	TXT.rtf Y*7U0
544	004C6F6C 67617573 65726E61 6D654F11 02293C21 444F4354 59504520 68746D6C	Lolgausername0)<!DOCTYPE html

Vulnerability 7: Login and registration data leakage through automatic iOS screenshot capture

Likelihood: Low

Impact: Medium

If the home button is pressed during the login or registration data entry, then iOS captures sensitive data on a screenshot. The screenshot can be later retrieved from the application's documents folder. Even deleted versions of screenshots can be sometimes found in the HFS journal.



Vulnerability 8: Payment details leakage through automatic iOS screenshot capture

Likelihood: Low

Impact: Low

If the home button is pressed during the payment procedure with bank links, then sensitive bank details can be captured on the iOS screenshot. More specifically, it can take a screenshot from the payment confirmation page, which may contain account number and balance.

Vulnerability 9: Insecure data logging

Likelihood: Medium

Impact: Medium

Application logs details of each API request to the iOS console (using NSLog). This reveals both API communication and user details, like Facebook authentication token. Every application user, who connects his device to the computer, can view all the log entries. Those

log entries can be used to explore API communication mechanisms or steal sensitive user data.

```
Apr 23 13:24:24 Olgas-iPhone Kosmos IMAX[8925] <Warning>; Current access token:
CAAwb6iwoxNcBALLUt5JtKLGiyXQ7Ty4RQ9GZCndfGwr7CUKTv6gq0zRm7ZBuCJA7YlGK1i4JmAjYqW6fDWDPRDpmmwCnv7jcQx44AW0j5dSC0srZazLMxuQ1cXuLTbcro6pCKQkVUyK8W2TZAx4hZBXP
KeB13A7tE07ZB1ZB0VaJ9r3ufkZATXK3ux80jLZCCZCtgcEB3yhneEEZAUcQlYsMuQpLD1ZBaXyHUYZD
Apr 23 13:24:24 Olgas-iPhone Kosmos IMAX[8925] <Warning>; Result: {
  email = "olgadaltan@hotmail.com";
  "first_name" = Olga;
  gender = female;
  id = 826795040734173;
  "last_name" = Dalton;
  link = "https://www.facebook.com/app_scoped_user_id/826795040734173/";
  locale = "en_GB";
  name = "Olga Dalton";
  timezone = 3;
  "updated_time" = "2015-03-23T19:10:11+0000";
  verified = 1;
}
Apr 23 13:24:24 Olgas-iPhone Kosmos IMAX[8925] <Warning>; Facebook access token:
CAAwb6iwoxNcBALLUt5JtKLGiyXQ7Ty4RQ9GZCndfGwr7CUKTv6gq0zRm7ZBuCJA7YlGK1i4JmAjYqW6fDWDPRDpmmwCnv7jcQx44AW0j5dSC0srZazLMxuQ1cXuLTbcro6pCKQkVUyK8W2TZAx4hZBXP
KeB13A7tE07ZB1ZB0VaJ9r3ufkZATXK3ux80jLZCCZCtgcEB3yhneEEZAUcQlYsMuQpLD1ZBaXyHUYZD
Apr 23 13:24:25 Olgas-iPhone Kosmos IMAX[8925] <Warning>; Global data update - LoginViewController
Apr 23 13:24:25 Olgas-iPhone Kosmos IMAX[8925] <Warning>; Global data update - MainViewController
```


```
Method: RESTLoyalty.svc/member
Request data: {"UserSessionId":"311eabf4bfe8efd82ec66591aad0510","CustomerLanguageTag":"et-EE"}
Response: {
  ErrorDescription = "<null>";
  LoyaltyMember = {
    Address1 = "";
    BalanceList = (
      {
        BalanceTypeID = 1;
        IsDefault = 1;
        LifetimePointsBalanceDisplay = 0;
        Message = "";
        Name = "Kosmos Points";
        NameAlt = "";
        NameTranslations = (
        );
        PointsRemaining = 0;
        RedemptionRate = "0.01";
      }
    );
    CardList = (
    );
    CardNumber = "";
    City = "";
    ClubID = 1;
    ClubName = "Galaktika valvurid";
    ContactByThirdParty = 0;
    DateOfBirth = "/Date(4099593600000)/";
    EducationLevel = 0;
    Email = "olgadaltan@hotmail.com";
    ExpiryDate = "/Date(4552238226000)/";
    ExpiryPointsList = "<null>";
    ExternalID = "";
    FirstName = Olga;
  }
}
```

M6: Broken Cryptography

Vulnerability 10: Application binary stores hardcoded API authorization tokens

During the application analysis with a “strings” tool, multiple token-like strings were found. Afterwards it turned out that those are used as API secret tokens. While secret tokens are not very efficient as API authorization scheme, if used, they must be better hidden in the binary.

```
http://vista.kinokosmos.ee/CDN/Image/Entity/FilmTitleGraphic/h-%@?width=225&height=450
LanguageUpdated
RESTData.svc/
OData.svc/
RESTLoyalty.svc/
RESTBooking.svc/
RESTTicketing.svc/
success
info
/order/cancel
/order/tickets
/order/seats
member/validate
member/signout
member
member/create
member/update
member/verifynewmembershipdetails
booking/search
Clubs
Cinemas
FilmGenres
Persons
Films
Sessions
FilmPersonLinks
/member/auth
/member/facebook
/member
/film/rating
?format=json
UserSessionID
ReturnMember
MemberPassword
```



M8: Security Decisions Via Untrusted Inputs

Vulnerability 11: Getting free tickets using runtime manipulation

Likelihood: Low

Impact: High

All information about existing user tickets is queried from API and stored locally in an array. It is possible to replace implementation of the method, which populates that array, and add an additional free ticket to the account. This will work, as long as a ticket controller does not recheck order identifier on entry. As a solution, both runtime controls and ensuring strong physical entry checks can be proposed.

M9: Improper Session Handling

Vulnerability 12: Logged in user session identifier stored as plaintext in NSUserDefaults

Likelihood: Low

Impact: High

A session identifier, which is used as an access token after the successful login, is stored as plaintext in NSUserDefaults. It gives a temporary access to all user related functionality, including list of tickets and account details change.

1RZSKcYgxn.plist > No Selection

Key	Type	Value
▼ Root	Dictionary	(8 items)
KosmosCinemaCode	String	9999
personsUpdateTime	Number	1 429 799 370,05536
com.facebook.sdk:serverConfiguratio...	Data	<62706c69 73743030 d4000100 02000300 04000500 06012301 24582476 65727369 6f
▶ com.facebook.sdk:lastInstallRespons...	Dictionary	(1 item)
UserSessionID	String	EYD4GTC7W6QAIFQVFFQA1SCAS8T1MC3H
genresUpdateTime	Number	1 429 799 369,9616
com.facebook.sdk:lastAttributionPing...	Date	23 Apr 2015 17:29:30
cinemasUpdateTime	Number	1 429 799 369,88553

Vulnerability 13: Application does not always destroy session identifiers for logged out users

Likelihood: Very low

Impact: Medium

If the application is offline, when sign out is requested, the user's session identifier remains active. When the user signs out, the client has to send a sign out request. However, if the request fails, the client does not try to repeat it and the session identifier remains active. Thus, the backend does not know about sign out.

M10: Lack of Binary Protections

Vulnerability 14: Application allows attaching a debugger to it

Likelihood: Medium

Impact: Low

The release version of the application can be debugged, which might give important clues about its internal logic. The application can be also used on jailbroken devices, but it is an expected behavior.