TU WIEN Informatics

Entwicklung von Sicherheitsrichtlinien für iOS-App-Entwickler

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Philip Karl Niedertscheider
Matrikelnummer 11776814

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. DI Dr. Thomas Grechenig

Wien, 8. Oktober 2022

Unterschrift Verfasser

Unterschrift Betreuung

TU WIEN Informatics

Developing security guidelines for iOS app developers

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Philip Karl Niedertscheider
Registration Number 11776814

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. DI Dr. Thomas Grechenig

Vienna, 8th October, 2022

Signature Author	Signature Advisor

Entwicklung von Sicherheitsrichtlinien für iOS-App-Entwickler

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Philip Karl Niedertscheider
Matrikelnummer 11776814

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. DI Dr. Thomas Grechenig

Wien, 8. Oktober 2022

Erklärung zur Verfassung der Arbeit

Philip Karl Niedertscheider

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Oktober 2022

Philip Karl Niedertscheider

Kurzfassung

Die Zahl der veröffentlichten mobilen Apps steigt jährlich, und mit ihr wächst auch die Vielfalt der relevanten Geschäftsfelder, zu denen auch softwarekritische Branchen gehören. Im Idealfall werden bei Apps, die von einer großen Zahl von Nutzern verwendet werden, die höchsten Sicherheitsstandards angewandt, um das Risiko von Sicherheitslücken zu verringern und ihre Nutzer vor Cyberangriffen zu schützen. Allerdings können die App-Nutzer die Sicherheit der Apps selbst nicht überprüfen und sind daher darauf angewiesen, dass die Entwickler über ein umfassendes Verständnis zur mobilen Sicherheit verfügen. Leider müssen nun aufgrund der weltweit steigenden Nachfrage nach Softwareentwicklern viele Anfänger zuerst geschult werden, um diese erforderlichen Fähigkeiten zu erwerben.

Diese Arbeit befasst sich mit dem Bedarf an Sicherheitsrichtlinien für Entwickler mobiler Anwendungen, indem sie verfügbare Richtlinien untersucht und zu dem Schluss kommt, dass ein Prozess zur Entwicklung neuer Richtlinien erforderlich ist. Dieser vorgestellte Prozess bietet eine Möglichkeit, Sicherheitslücken im App-Ökosystem des Betriebssystems iOS zu finden und die Risiken durch die Erstellung einer Reihe von validierten Richtlinien zu reduzieren. Die vorgestellten Maßnahmen können von iOS-App-Entwicklern während der Programmierung angewendet werden und verringern so das Auftreten gängiger Sicherheitslücken. Die behandelten Themen sind die Absicherung der Netzwerkkommunikation, die Validierung der vom Benutzer eingegebenen Daten, die Verwendung von sicheren Datenspeichern und die Härtung gegen Binary-Reverse-Engineering und Binary-Manipulation. Die Analyse der Bedrohungen und der damit verbundenen Angriffe, sowie die Bereitstellung möglicher Lösungen, tragen dazu bei, das Sicherheitsbewusstsein zu erhöhen. Darüber hinaus wird ein Entwickler nach Anwendung der in dieser Arbeit gezeigten Prinzipien ein tieferes Verständnis für die Sicherheit mobiler Anwendungen haben und kann dies als Grundlage für weitere Forschungen nutzen, wodurch das allgemeine Sicherheitsniveau erhöht wird. Der vorgestellte Prozess wird mithilfe praktischer und reproduzierbarer Validierungstests in verschiedenen Szenarien bewiesen. Schließlich können der entwickelte Prozess und die Richtlinien, die für das Betriebssystem iOS vorgestellt wurden, auch als Grundlage für die Entwicklung ähnlicher Richtlinien für andere Plattforme (wie Android) verwendet werden.

Keywords: *Sicherheit, Richtlinien, iOS, Entwickler, Mobile, Apps*

Abstract

The number of mobile applications released every year is increasingly growing and with it, the variety of relevant business fields begin to include software-critical industries too. Ideally, apps used by large quantities of users have applied the highest security standards to reduce the chances of carrying security vulnerabilities, and to protect their users from cyber-attacks. However, the app users cannot verify the security of apps themselves, thus relying on developers to have an extensive understanding of mobile security principles. Unfortunately, due to the globally growing demand for software developers, many beginners need to be trained first to gain that necessary skill set.

This thesis reflects on the need for security guidelines for mobile app developers by researching the state of the art and concluding that a process for developing new ones is required. The presented process provides a way of finding security crunchpoints in the application eco-system offered by the operating system iOS and reducing the risks by creating a set of validated guidelines. The results are a set of security guidelines, applicable by iOS app developers during the development process, thus reducing the risk of common security flaws. The discussed topics are securing network communication, validating user-injected data, the use of secure data storage, and hardening against binary-reverse-engineering and binary-tampering. Analyzing threats and their associated attacks and providing potential solutions, helps raising security awareness. Furthermore, after applying the principles shown in this thesis, a developer will have a deeper understanding of mobile app security and can use it as a foundation for further research, thus improving the overall security level. The proposed process is proven by performing practical and reproducible validation tests in a variety of scenarios. Finally, the process and guidelines presented for the operating system iOS can also be used as a foundation to develop similar guidelines for other platforms such as Android.

Keywords: *Security, Guidelines, iOS, Developer, Mobile, Apps*

Contents

CHAPTER 1

Introduction

Cybersecurity is a prominent topic in the field of computer information technologies (IT) and applies to a vast variety of different systems and technologies. This thesis explores the state of cybersecurity for mobile applications, existing problems, and opportunities for improvements in the industry.

1.1 Problem Description

The global market for mobile app development is growing every year [105] and with it the variety of apps widely available to smartphone users [104]. With the increasing availability of apps, the number of different business fields using mobile applications for their use cases also expands, including software-critical industries. One example are health apps, which nearly doubled in count between 2015 and 2019 [8]. New regulations for the critical financial sector, such as the payment service directive 2 (PSD 2) by the European Commission, even force banks to utilize mobile applications to collect payment confirmations from their customers [74].
As mobile applications become part of the daily lives of users, they need to trust them without the option of validating the behavior themselves, as they might not have the knowledge to understand the technology, nor have the insight into proprietary software source code. Consequently, they fully rely on the expertise of software developers to implement secure apps. This security responsibility needs to be fulfilled by the engineers, which furthermore requires an extensive understanding of what security is and how it applies to the mobile software landscape.
An increasingly wider range of apps built and therefore, also maintained, leads to a higher demand for mobile app developers. Unfortunately, a report by the Association for Computing Machinery (ACM) [6] states, that the American Council on Education (ACE) noticed a 43% decline in international students enrolling in computer and information science in the U.S. due to the COVID-19 crisis [6]. They estimate this decline would

"[...] reduce BLS (U.S. Bureau of Labor Statistics) estimated job growth by 38% in 2021 by leaving thousands of jobs unfilled [...]". The already high demand will therefore eventually result in an extensive effort to hire and train more entry-level workers. But as many beginners might not know the mobile app eco-system well enough (or do not prioritize mobile security in the beginning of their career), they need to be trained to know industrial IT security standards and policies to raise security awareness. Additionally, Gasiba et. al [19] claim that current approaches focus on security-aware groups, such as pen-testers, or on more general IT fields, such as password handling, leaving out the field of software developers which acts in between.
Along with this growing market, the mobile security report by Check Point Research [67] states that the interest of malicious parties exploiting mobile apps is intensifying, and their surveyed organizations had at least one employee who faced mobile malware in the past year.
This combination of inexperienced developers introduced to mobile platforms, in combination with increasing security risks and requirements, sparks the interest in creating a set of security guidelines for iOS developers, which should be directly applicable during development.

1.2 Expected Results

This thesis aims to define a collection of security guidelines, which can be applied during the development process of applications for iOS. Analyzing threats and their associated attacks and providing potential solutions will help raising security awareness of developers and reduce common security breaches. This increase in awareness should eventually lead to more research focusing on more in-depth security problems and reduce the amount of time invested in well-known issues.
This thesis intends to provide developers with an established set of guidelines regarding security practices for iOS, which can be used as a baseline for decision reflection. These guidelines serve as a quantitative overview of relevant security topics and summarize their most important aspects.
This paper will not provide a full definite standard for app developer security measures, but instead, highlight issues that can be circumvented in a reasonable time. Proposed solutions should be considered as techniques to harden the software against malicious intent, rather than fully mitigating all attacks.

1.3 Methodological Approach

This paper is built on a foundation established through scientific literature research. Upon performing extensive research, the core concepts of security in computer systems and the CIA triad are defined. Furthermore, multiple existing scientific definitions of the terms "threat" and "attack" are used as the basis to define the context of this thesis. To narrow down the fully observable domain, the focus is set on the mobile operating system iOS, discussing its main capabilities and potential security issues. As the operating

system serves as a platform for mobile app developers to run their apps, the runtime environment as well as relevant architectures and structures are explored.
After the discussion of core principles and the technical context, this thesis will evaluate existing guidelines, such as the OWASP Top 10 Mobile Risk [95] and the Apple Platform Security Guide [42] from a qualitative viewpoint. Furthermore, scientific approaches are explored to reflect on previous works in this field.
Combining the existing guidelines with further research of the topics covered, additional ones based on the collected information are defined. To create a new guideline, it is necessary to first identify the security crunchpoints and the threats it aims to decrease. To visualize the relevance of specific security issues from a practical side, and validate the effectiveness of the guidelines, validation examples are shown as well. These also present potential mitigation measures used to harden the software projects.

1.4 Structure Of The Thesis

This paper is organized into seven chapters. After the introduction of the objective of this paper in *Chapter 1*, the fundamentals of software cyber-security, and their core terminology are described in *Chapter 2*. Transitioning from the security aspect to operating systems, *Chapter 3* reduces the observed domain of operating systems to the mobile operating system iOS. This chapter covers more relevant details of the operating system, highlights its security attack surface, and justifies the requirement for development guidelines. *Chapter 4* covers existing security guidelines for mobile app development and describes the need for further ones. The following *Chapter 5* explains the process of developing guidelines, and shows the ones developed in the scope of this paper. To validate their effectiveness, *Chapter 6* tests the proposed guidelines, concluding the paper in the final *Chapter 7*.

1.5 Test Lab Setup

The technical examples and practical validations shown in this thesis can be reproduced with the following setup.

MacBook Pro with macOS 12 macOS 12 is the operating system shipped with Apple MacBook Pro devices. For this work, version 12.3 (21E230) has been used on a MacBook Pro (16-inch, 2021) with the ARM-based processor Apple M1 Max.

Xcode Xcode is the integrated development environment (IDE) distributed by Apple. It can be downloaded from the macOS App Store or their developer website [64]. Xcode consists of many tools used to build apps for Apple platforms. Next to a text editor with code highlighting and syntax auto-completion, it also includes an iOS simulator, performance measurement tools, and memory-debugging utilities. Furthermore, the IDE includes the "Xcode Command Line Tools", which includes the Apple LLVM compiler and

linker used to create binaries for Apple platforms. Xcode is still under active development, and for this work, version Xcode 13.3 is used.

Docker Docker is a container virtualization management tool allowing to locally build, share, and run containerized applications and operating systems [72]. Using different containers allows to run additional toolsets and operating systems. It can be downloaded and installed from their website bundled as the Docker Desktop application. For this work, the version Docker Desktop 4.12.0 (85629) [73] is used.

Visual Studio Code Visual Studio Code is an open-source text editor by Microsoft [82]. It is optimized for building and debugging modern web and cloud applications but can be extended by a vast variety of plugins. For this thesis, the version 1.72.0, with the Hex Editor plugin version 1.9.9, is used.

Ghidra Ghidra is a reverse engineering tool suite by the Nation Security Agency (NSA) of the United States of America [108]. It can be downloaded for free and used to reverse engineer, namely decompile and patch, executable binaries. For this work, version 10.1.5 is used.

Proxyman Proxyman by Proxyman LLC [97] is a network relay proxy to intercept and manipulate network communication. It enables the installation of self-signed root certificates to perform Man-in-the-Middle (MITM) attacks. For this thesis, the version 3.8.0 is used.

radare2 Radare2 [98] is a UNIX-like reverse engineering framework and command-line toolset. In this thesis the version 5.7.6 is used.

This setup suffices to perform the practical part described in this thesis. Additional optional but helpful tools can be found in the Mobile Security GitBook by the OWASP Mobile Security Testing Guide [94].

CHAPTER 2

Fundamentals

To explore the landscape of cybersecurity the fundamental concepts need to be analyzed and defined first. The full problem space is vast but can be reduced to the principles of the *CIA Triad*, the definitions of the terms *Threat* and *Attack*, and the observed domain of *Mobile Operating Systems*.

2.1 CIA - Triad

According to the "NIST Handbook - Introduction to Computer Security" [20] the *CIA triad* is an acronym representing *Confidentiality*, *Integrity*, and *Availability*, the three main principles defining the foundation of computer security. These principles can be considered the main constraints that need to be considered at any point in time to provide a secure system. Security measures are therefore actions taken to reduce the probability of breaking one of these constraints, while adversaries actively try to weaken them.

2.1.1 Confidentiality

Confidentiality requires engineers to design systems that protect information and restrict access [20]. This not only includes secure data storage and secure data communication but also data privacy and the hiding of proprietary information of any kind.

An example security issue for broken *Confidentiality* are leaked account passwords because it leads to an unauthorized entity gaining access to sensitive data, which the entity is not supposed to have.

2.1.2 Integrity

Integrity is the principle of protecting data and systems against improper modification or deletion [20]. As data and systems can change daily, it is required that solutions check

for authorization during modification. Furthermore, it requires traceability of intentional and unintentional changes made to data and systems.

An example security issue for broken *Integrity* is publishing malicious content on an external website because it modifies the website without the consent of the website owner, therefore presenting malicious information in the owner's name.

2.1.3 Availability

Availability is the principle of providing reliable access with appropriate access times [20].

An example security issue for reduced *Availability* is overloading a system with a distributed denial of service (DDoS) attack, with the intent to deny the processing of valid requests, thus preventing users from accessing the system features.

2.2 Defining The Term "Threat"

The term "threat" is widely used in security for computer information technologies. The scientific literature research shows that various definitions with similar approaches, but also with slight differences. It is also defined in the RFC standard [77] which is a technical document provided by the Internet Engineering Task Force [76]. The research results are used to define the term for the context of this thesis.

The *National Systems Security Engineering Project* (SSE) by the *National Institute of Standards and Technology* (NIST) [26] describes a threat as an "[...] event or condition that has the potential of causing asset loss and undesirable consequences or impact from such loss". Therefore, it is a vulnerability of a system that makes it possible to exploit undesired behavior, breaking at least one principle defined by the CIA triad.

As of RFC4949 [28] a threat is defined as a "[...] potential of violation of security [...]". The standard also differentiates between threats and vulnerabilities, with threats being *caused* by security vulnerabilities of a system. Sion et. al [29] define the process of "threat analysis" as the identification of potential security issues and the evaluation of violation risks.

Threats can be categorized into *intentional threats* and *accidental threats* [28]:

Intentional threats These threats are caused by a malicious party, e.g., an attacker with access to a system installing a backdoor for privileged remote access [28].

Accidental threats These threats are caused by human error or low quality of work, e.g., not handling all known edge cases or missing defense against previously unknown edge cases [28].

Based on the research of [26], [28], and [29], the term "threat" can be defined as:

Definition 1: Threat

The risk of malicious functioning of a system with consequences violating the security principles of the CIA triad.

2.3 Defining The Term "Attack"

Another important term in computer security is "attack", which is also widely used in scientific works and defined in various ways. Similar to the process of defining the term "threat", the scientific and standard definitions are used to define the term in the context of this thesis.

The RFC4949 [28] defines the term *attack* as an "[...] intentional act by which an entity attempts to evade security services [...]" and as a form of assault on a computer system.

In the context of computer security, attacks are also actions taken to exploit a vulnerability represented by a threat, to break at least one of the core principles of the CIA triad.

The standard [28] declares the perceived danger posed by an attack to be considered the *attack potential*, which represents the probability of successfully using an attack against a system.

The cybersecurity glossary of the *National Initiative for Cybersecurity Careers and Studies* (NICCS) [83] defines terms based on multiple standards, including the glossary [69] published by the governmental entity *Committee on National Security Systems* (CNSS) [68]. This glossary defines the term *attack* as the intentional attempt to bypass security services to gain unauthorized access to a system. They extend the previous definition of an *active attack* with "[...] attempts to alter a system, its resources, its data, or its operations" [83] and *passive attack* with the intent not to modify the system.

Different kinds of attacks are closely related and can be characterized by their intent (*active* versus *passive* attacks), their point of initiation (*inside* versus *outside* attacks) and the method of delivery (*direct* versus *indirect* attacks) [28].

Active attacks An active attacker can actively interfere with the process to change the outcome [28].

An example is a Man-in-the-Middle attack that is used to manipulate remote communication.

Passive attacks A passive attacker observes a process to gain more information about it [28].,

An example is listening to network traffic to find unencrypted data.

Inside attacks An inside attack is caused by an adversary with authorized access to the target system [28].

An example is a dissatisfied employee with authorized access to private information leaking said private information to a public audience, violating *Confidentiality*.

Outside attacks An outside attacker does not have authorized access to the victim system at the time of attack [28] but might gain access as a result of the attack.

Examples include cybersecurity attacks trying to gain information the attacker was never intended to obtain, e.g., intruding into a private network without using legal methods.

Direct attacks A direct attack is targeted directly at a specific victim [28].

An example is gaining physical access to a specific computer system, to e.g., damage the hardware.

Indirect attacks In an indirect attack the attacker redirects attacks via a third party [28].

An example for indirect attacks is supply-chain attacks, with the attacker targeting an intermediate victim, e.g., a company providing a third-party service, in result affecting the actual victim by e.g., breaking the *Availability* of mentioned service.

Based on the research of existing terminology from [28] and [83], the term "attack" can be defined as:

Definition 2: Attack

The act of exploiting a security risk given by a threat, with the objective of violating the security principles of the CIA triad and causing consequences admirable to the attacker.

2.4 Defining Mobile Operating Systems

The term "Mobile Operating Systems" consists of two parts, the "Operating System" and the reduced observed domain "Mobile". Before defining mobile operating systems as a whole, it is helpful to define operating systems on their own.

Operating systems (OS) evolved over the previous decades from direct hardware programming, over computing serial job management and controlling, up to multi-program multi-task processing and scheduling [31]. Historically only used by programmers with technical backgrounds, modern OS evolved even further due to the higher accessibility to non-technical users.

Stallings [31] defines an OS as a software layer by providing access to the hardware and its processor units to "[...] provide a set of services to system users.". Additionally, the hardware of a computer consists of input/output (I/O) devices and various memory storages. An OS provides an abstraction layer built on top of the computer hardware, providing low-level services such as program management and execution, I/O communication with peripheral devices, and memory management.

Furthermore, modern OS can be extended by third-party developers, by providing application programming interfaces (API) to programmers to access hardware components, such as high-security chip storage.

The core of an OS is considered a *kernel*, with UNIX being the most influential system in computing history even half a century after its original inception in the early 1970s [30]. This specific OS provides additional functionality such as multi-tasking and multi-user support, and separates the core OS from user applications. By additionally splitting up applications into multiple processes, it is possible to build a multi-processor environment.

Simultaneously to the evolution of OS, the size of computers changed as well. Andrews et al. stated in an article [4] in the mid-20th century, that with the transition from analog computers to digital computers and the introduction of high-speed memory, the higher computing requirements can be met with more execution time, rather than larger systems.

Starting with room-sized general purpose workstations in the mid-20th century, over personal desktop computers in the 1980s, up to mobile computing devices, such as "smartphones" in the beginning of the 21st century, the sizes of systems keep getting smaller.

Originally only accessible by authorized personnel in isolated environments, modern mobile devices are non-stationary and carried in public by large numbers of people.

Statistics provided by Statista [102] shown in Figure 2.1 display the wide spread of mobile devices, with at least 14.2 billion devices shipped since 2007, with 1.354 billion in 2021 alone.

With higher connectivity options, such as cellular data networks or Bluetooth communication, the devices include more variety in hardware and operating systems are constantly adapted to meet the latest requirements. Connectivity solutions also need to be accessible by application programmers and the, therefore, larger attack surface requires more internal security measures.

As an operating system is a collection of software, it also needs to provide general security measures. A few examples are different execution modes for user- and kernel-execution [31], or hardware-level encryption of file storage.

With the reduction of hardware size, the mobility of these devices changed drastically as well. Ai et al. [3] evaluates the current functionality and challenges of smartphones, with a prospect of what the "smartphones of tomorrow" will be able to perform. The

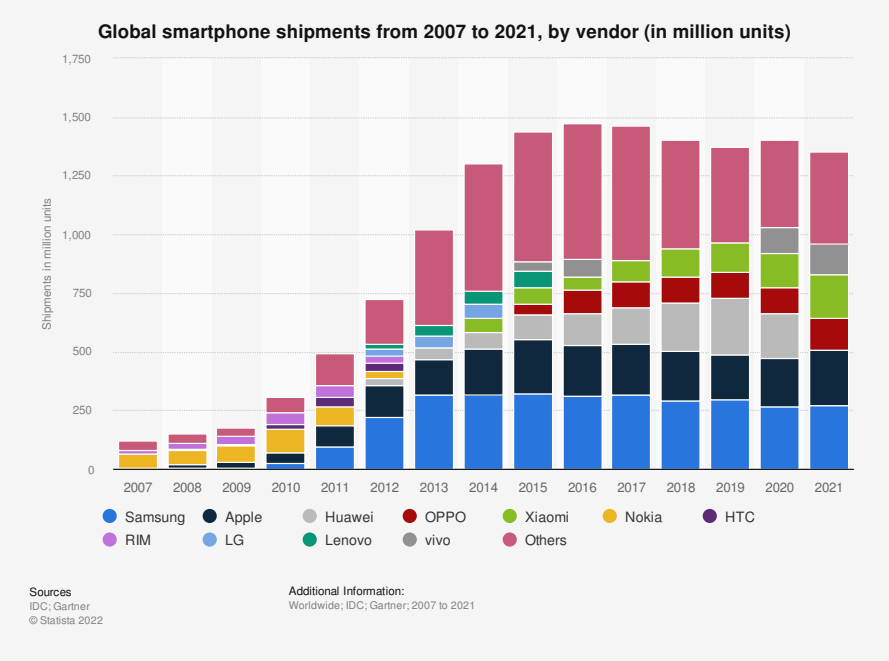

Figure 2.1: Global smartphone shipments from 2007 to 2021 [102]

devices at the time of writing this paper already provide a large set of hardware, such as Wi-Fi and Bluetooth access, multiple cameras, wireless charging options, and even LiDAR depth sensors, which allow for interaction with the environment [25].

Based on the research results from [3], [25], [30], and [31] the following definition can be established:

Definition 3: Mobile Operating System

A mobile operating system is the core software installed on a non-stationary handheld device, with the intention of combining cellular connectivity and computing, providing system users with services to interact with the hardware, and additionally offering developers interfaces for extending the software capabilities available on the device.

According to Statista [103] in January of 2022, the market share for mobile OS is divided into 69.74% claimed by Android (developed by Google), 25.49% taken by iOS (developed by Apple) and 0.77% claimed by others. As seen in Figure 2.2, the market share of iOS has been consistent at about 20-30% since 2012, while most other operating systems have been overtaken by Android.

For the context of this thesis, the whole area of mobile operating systems is reduced to the specific mobile operating system iOS.

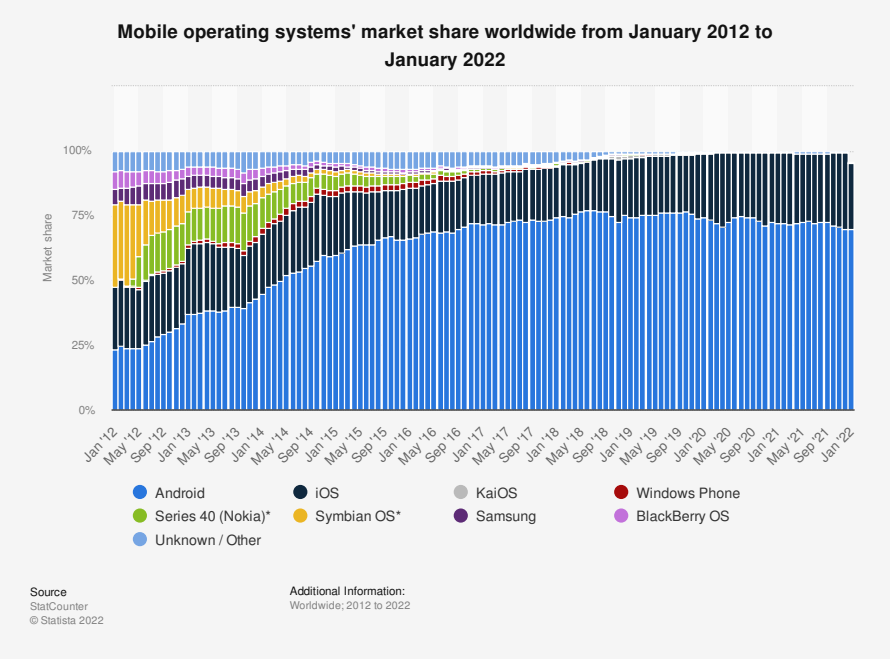

Figure 2.2: Market Share of mobile operating systems [103]

CHAPTER 3

Apple iOS Operating System Fundamentals

iOS (formerly called *iPhone OS* until 2010) is a mobile operating system developed by Apple [49] for their flagship smartphone product "iPhone". Until 2019 the operating system was also used for the tablet device "iPad", which was then derived into its own "iPadOS". iOS (and therefore, also iPadOS) finds its heritage back in "OS X" [21], which was the original name of Apple's desktop operating system "macOS" [53], until the release of macOS 10.12 in 2016.

Due to the shared history of these operating systems, they utilize similar system architectures. The main difference can be found in the user interface technologies, which are "Cocoa Touch" for iOS and iPadOS instead of the "Cocoa" layer for macOS [55]. Furthermore, some technologies do not exist on all systems, due to limitations in device capabilities (e.g., virtualization using *Hypervisor* [47] is only available on macOS), while others have a counterpart for each platform (e.g., *UIKit* [62] versus *AppKit* [41]).

3.1 Operating System Architecture Of iOS

Due to the shared history and similar architectures, it is possible to analyze the operating system iOS by taking a closer look at the architecture of macOS instead, as more architecture documentation resources are provided by Apple for macOS, rather than iOS.

According to the *Apple Documentation Archive* [38], the desktop operating system macOS consists of 5 layers, with the top layers using services and technologies provided by lower layers (see Figure 3.1).

Cocoa Touch (Application) Layer The Cocoa Touch layer includes technologies for building user interfaces and reacting to user interaction and communication [43]. The

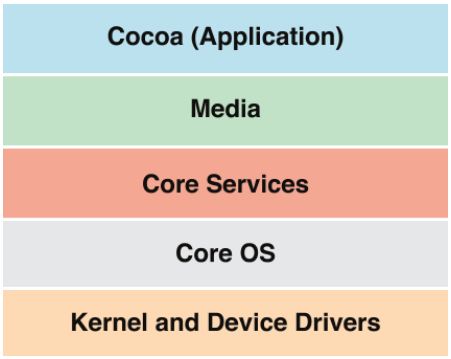

Figure 3.1: Layer model used by the operating system *macOS* [38]

two most relevant frameworks provided by this layer are "UIKit" for iOS and "AppKit" for macOS (and "SwiftUI" as an abstraction of both). Furthermore, it integrates with user services providing accessibility support and systemwide search capabilities [55].

Media Layer The Media layer provides apps with specialized technologies for working with graphics, audio, and video [54]. These include a wide range from handling different media file types and playback, over animation frameworks and complex font typography management, down to services like "Metal" or "OpenGL" for low-level access to the graphics processing unit (GPU) chipset.

Core Service Layer The Core Services layer contains a vast number of fundamental services and frameworks provided to applications running on the system [44]. The frameworks can be utilized by application developers to implement common use cases, such as internationalization, multi-thread dispatching, network discovery, and secure data persistence (see Section 3.2.5).

Core OS Layer The Core OS layer contains many high-level features and low-level services [44]. A few highlights of the high-level features are "Gatekeeper", app sandboxing (as discussed in Section 3.2.1), and code signing (as shown in Section 5.3.5). Low-level services are provided to the system as frameworks, such as the high-performance computing framework "Accelerate" [39] or access to the device network configuration settings with the framework "System Configuration" [59]

Kernel and Device Drivers The Kernel and Device Drivers layer is the lowest layer including the UNIX-like kernel XNU (abbrev. for "XNU is Not Unix") and hardware drivers [50]. The XNU kernel is a hybrid, open-source kernel developed by Apple and is based on the Mach-kernel for device hardware management and on a portion of the BSD-kernel for providing standard POSIX-compliant system calls [65]. The layer also includes other low-level components for different requirements, such as supporting file systems, networking, security, inter-process communication, and dynamic runtime environments.

3.2 Application Eco System

The operating system iOS has strict guidelines for running apps, to provide limited working space with clearly defined boundaries and interfaces. To further understand this environment, it is informative to investigate a few specific aspects, which have an impact on the development process.

3.2.1 Isolated Execution/Sandboxing

Sandboxing is the security mechanism for restricting applications from interacting with resources, they do not require during runtime [5]. Its main purpose is the isolation of processes, so that the potential damage of malicious ones is contained by default.

iOS apps need to support sandboxing to be installable on an iPhone or iPad, and to be accepted for further distribution via the Apple App Store [42]. As the App Store is the only existing distribution channel at the time of writing, it is not possible to distribute iOS apps to end-users without sandboxing being enabled. On a closer look, iOS differentiates between private and public frameworks, with the latter one offering shared system libraries and APIs available to all third-party apps, while the former ones should only be accessed by system applications [7]. This separation is enforced by Apple in their App Review process, which needs to be conducted for every release deployed to the iOS App Store.

Bucicoiu et al. [7] describe the application sandboxing security architecture as a three-layer software system, as shown in Figure 3.2. The three layers start with the kernel layer at the bottom, offering basic system services (e.g., a file system) and a kernel module. On top of the kernel layer is the Objective-C framework layer and privacy setting service, and finally the least-privileged application layer for third-party and built-in apps. The access control is managed by a combination of kernel components and sandboxing profiles.

Furthermore, the "sandboxed" app has limited access to the file system, by using a unique home directory that is created at the time of installing the app. This unique directory is located at `~/Library/Containers/<random app bundle id>` on a compatible device running iOS, or at the path in Figure 3.3 for an iOS simulator running on macOS.

Additionally, the file system access from user apps is restricted by the operating system permission system, resulting in access denial errors when accessing system files, as the apps are run in a non-privileged user-mode and the OS is mounted as a read-only file system [42]. A read-only file system will disable any writing capabilities starting at the time of booting, therefore reducing the attack surface even if the attacker is able to perform privilege escalation.

3.2.2 Cross-Platform-Compatibility With macOS

Many different computing platforms are used with a variety of technologies in different use case scenarios. Examples include server software running headless (without a graphical

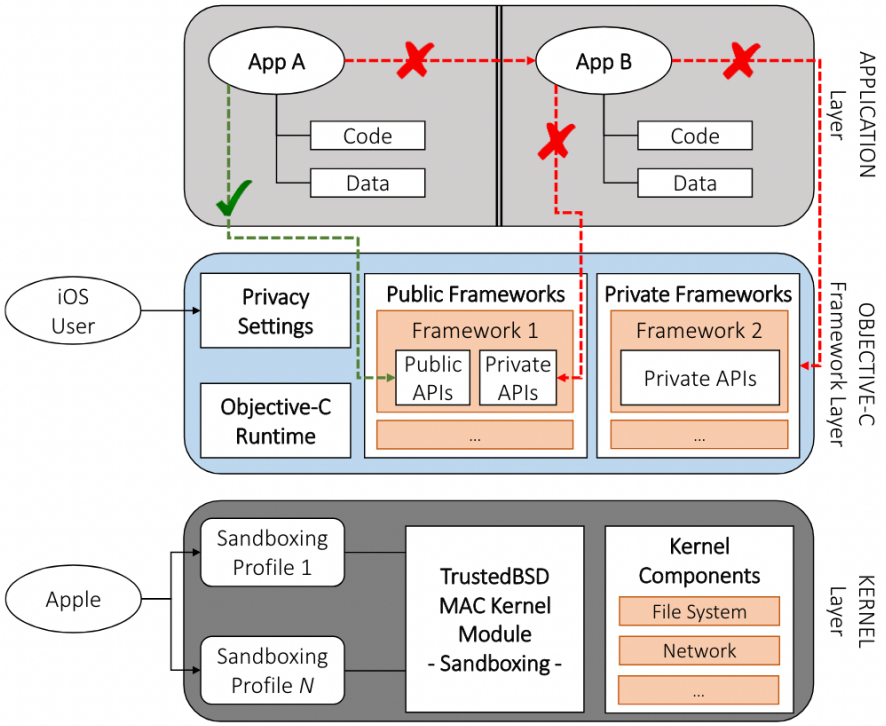

Figure 3.2: Sandboxing Security Architecture Model [7]

```
/Users/<macOS_username>/Library/Developer/CoreSimulator/
    ↪ Devices/<simulator_id>/data/Containers/Data/
    ↪ Application/<random_app_bundle_id>
```

Figure 3.3: Path of an app container file system of an app running in the iOS Simulator on macOS.

user interface), information of a website served to users via a web browser, to desktop applications running on the personal computers of end-users. Over time the variety of end-user devices and operating systems grows, forcing developers to provide their products on multiple systems in parallel.

From an economic standpoint it is in the highest interest of a product developer to create and maintain a product only once, but still be able to distribute it on as many platforms as possible. One strategy is developing a product in such way, so that it directly runs on multiple platforms, and this feature is called *Cross-Platform-Compatibility*.

While not only designing and distributing software products is challenging for multiple platforms, the technical aspect also introduces downsides as well. Due to the release of the latest processors based on the ARM64 architecture by Apple, it is now possible to run iOS apps natively on macOS [5], enabling cross-platform compatibility without

re-compilation. But the introduction of a new processor architecture for a major operating system brings security issues with itself, and as mentioned by Hetterich et. al [9], security experts immediately started with extensive research for new and existing attack variants, as previous research on desktop systems mostly focused on Intel x86 architectures.

As illustrated in Apple's documentation on "Running your iOS apps in macOS" [58], the system automatically provides appropriate alternatives for iOS-specific features available on the macOS platform. Furthermore, they provide guides [40] to adapt the iOS apps to be compatible.

An example of a significant difference between iOS and macOS is, that the iOS App Store requires apps to use the sandbox model, but for macOS, these requirements do not fully apply, as sandboxing is optional [5]. This leads to iOS apps being exposed to vulnerabilities found on macOS, e.g., the vulnerable sandbox on macOS mentioned by Xing et. al [35].

In conclusion, an iOS app developer is creating apps for the ecosystem of the mobile operating system iOS and therefore, should focus on securing the application for that environment. That being said, developers planning to run their iOS apps on macOS as well should perform further research on the differences between the ecosystems.

3.2.3 Access To System Services

iOS provides app developers with a variety of system services, e.g., sending notifications to the user notification center or interacting with the voice assistant Siri. It follows the *Principle of Least Privilege* [27] and gives apps the smallest set of privileges necessary to function, to reduce the probability of unwanted privilege misuse. If app developers require more privileges to interact with the system services, they need to enable them individually (and, in specific cases, request them from Apple as described in this section). These enabled privileges are called *Entitlements*.

Entitlements are stored in a file with the extension `.entitlements` containing key-value pairs in the format of a *Property List*, which is a commonly used file format for serialized data developed by Apple. According to Blochberger et. al [5] a total of 50 entitlements are documented by Apple. Furthermore, the entitlements file is part of the compilation process and embedded in the app's executable, thus its integrity is also protected by the code signing process. As this configuration is processed during at compile-time, rather than at runtime, an attacker cannot change the entitlements later on without also altering the integrity of the binary and therefore, invalidating the code signature.

Blochberger et. al [5] also mentioned a few cases where Apple allowed app developers to use *Private Entitlements*, for example UBER, which had the entitlement `com.apple.private.allow-explicit-graphics-priority` [5]. This claim is outdated, as the entitlement cannot be found in the application at the time of writing this thesis anymore, as shown in Figure 3.4.

```
Executable=/Users/Philip/Desktop/Uber
    ↪ 3.519.10005/Payload/Helix.app/Helix
[Key] application-identifier
[Key] aps-environment
[Key] com.apple.developer.applesignin
[Key] com.apple.developer.associated-domains
[Key] com.apple.developer.icloud-container-environment
[Key] com.apple.developer.icloud-container-identifiers
[Key] com.apple.developer.in-app-payments
[Key] com.apple.developer.siri
[Key] com.apple.developer.team-identifier
[Key] com.apple.developer.usernotifications.time-sensitive
[Key] com.apple.security.application-groups
[Key] keychain-access-groups
```

Figure 3.4: Excerpt of the entitlement keys of the iOS app *UBER* version 3.519.10005 extracted using *codesign*.

But there are still other apps using private entitlements. As public and private entitlements can also be used for macOS applications, an example of a macOS app with at least one private entitlement is the app *Transmit 5* by Panic, Inc. [60].

To display the entitlements granted to an iOS (or macOS) application the command line utility `codesign`, which is part of the *Command Line Utilities* included with Xcode and requires the path to a `.app` package as a parameter, can be used. In the example of *Transmit 5* the application can be downloaded from the macOS App Store [60] onto a macOS-capable device, and the entitlements are shown by executing `codesign -d` as in Figure 3.5.

```
codesign -d --entitlements :- /Applications/Transmit.app
```

Figure 3.5: Command used to display entitlements of the macOS app *Transmit 5*

The command output in Figure 3.6 displays the key-value pairs for each entitlement. By cross-checking them with the official documentation by Apple [46] and statements by an Apple engineer in the Apple Developer Forum [52], it can be concluded that the entitlement `com.apple.developer.security.privileged-file-operations` is an example of a private entitlement used in a production app.

These entitlements allow third-party apps to interact with system services such as the *AutoFill Credential Provider* (providing usernames and passwords for login processes), access to the local contacts or systemwide user health information, as well as integrating with the underlying networking layer i.e., providing VPN network access.

Other services provided by the system include higher-level applications, such as the option of sending SMS text messages or calling a number using the cellular service.

```
Executable=/Applications/Transmit.app/Contents/MacOS/Transmit
[Dict]
    [Key] com.apple.application-identifier
    [Value]
        [String] VE8FC488U5.com.panic.transmit.mas
    [Key] com.apple.developer.security.privileged-file-operations
    [Value]
        [Bool] true
    [Key] com.apple.developer.team-identifier
    [Value]
        [String] VE8FC488U5
    [Key] com.apple.security.app-sandbox
    [Value]
        [Bool] true
    [Key] com.apple.security.application-groups
    [Value]
        [Array]
            [String] VE8FC488U5.com.panic.Transmit
    [Key] com.apple.security.automation.apple-events
    [Value]
        [Bool] true
    [Key] com.apple.security.files.downloads.read-write
    [Value]
        [Bool] true
    [Key] com.apple.security.files.user-selected.read-write
    [Value]
        [Bool] true
    ...
```

Figure 3.6: Excerpt of the entitlements of the macOS app *Transmit 5* version 5.8.8 extracted using *codesign*.

It is also possible for iOS developers to extend existing systemwide services, such as providing additional commands for the voice assistant *Siri* or providing a file system to the *Files* app (e.g., an app connecting to a network attached storage not supported by iOS natively).

3.2.4 Inter-App Communication

Apps often do not only have to communicate with the underlying operating system iOS but also with each other. This can be grouped into two categories:

1. Inter-app communication between two apps by the same developer
2. Inter-app communication between two unassociated apps

These two categories need to be differentiated, because of their associated level of trust. By relying on the limited communication levels, regulated by the operating system, it can be assumed that the origin of the communication channel is credible. This takes away the need for a developer to enforce a custom access control for these kinds of *private communication*. Furthermore, it can be assumed that the other party in the communication knows the relevant protocols and tries its best to conform. An example of communication between two apps by the same developer is sharing stored login credentials using *App Groups* or *Shared Keychain Access*, as described in more detail in Section 3.2.5.

In case of communication between two unassociated apps, the recipient offers a public interface which is consumed by the sending app. On iOS, these communication channels are regulated via the operating system, such as the *UIActivityViewController*, *Custom URL Schemes* also called *Deep Links*, *Share Extensions*, and *Custom Pasteboards*. Furthermore, it is possible to communicate with external network services using *NSConnection*.

Deep Links, also called *Schemes* [35], are custom URL schemes only available in the domain of the device, e.g., `yelp://search?terms=Coffee` launching the app Yelp, which then processes the full URL and i.e., performs a search for the given term "Coffee". Xing et al. [35] state that only one single app can be associated with a given scheme but does not need to be registered with Apple or configured by the end-user in the OS, and instead can be defined by the developer in the app configuration. This can lead to multiple apps using the same URL scheme and therefore, the operating system needs to resolve the conflict itself. Even though the Apple Documentation [45] claims the "[...] app the system targets is undefined [...]" and "[...] there's no mechanism to change the app [...]", Xing et. al [35] claim that macOS picks the first app that has registered the scheme, while iOS picks the last one.

3.2.5 Secure Data Storage

To reuse data between executions, computer programs need to write data to a storage solution which can then be read again. These systems are called *Data Storage* and are implemented as a file system on iOS devices. When using the file system, one needs to consider who should be able to access the stored data, and especially when storing sensitive data (e.g., passwords or encryption keys) it is necessary to consider the principles of the CIA triad.

As file systems might be accessible with the device being powered off, the operating system takes measures to always keep the data securely stored.

On a hardware level, Apple uses an AES-256 hardware crypto engine, which encrypts all data on the file system [7]. Hayran et al. [22] state that the necessary keys are the UID and GID, which are integrated into the silicon during the manufacturing of the device to help prevent an adversary from extracting the data storage hardware, as it would require extracting additional memory chips as well. This strategy involves the specific device identifiers during the encryption process, basically binding the data to the particular device. To further increase data security, data is also bound to a person by using the person's personal passcode as entropy for the encryption keys [22].

The personalization strategy is extended by the biometric hardware provided by iPhones and iPads. To further limit access to sensitive data, iOS provides the system apps and third-party apps the system service *Keychain*. As stated in the Apple Platform Security Guide [42] the items in the Keychain can also be protected by Touch ID or Face ID, and are only released from the secure enclave if the biometrics match.

To manage the access protection level of files, iOS offers app developers a variety of *Data Protection classes* [42]. The class *Complete Protection* [42] derives a class key from the

user passcode or password and is discarded 10 seconds after locking the device, blocking any access until the user logs in again using the passcode, Touch ID, or Face ID [24]. For background processes that need to continue after the device has been locked, the class *Protected Unless Open* [42] can be used, which uses a combination of asymmetric cryptography and short-lived encryption keys stored on-device. *Protected Until First User Authentication* [42] works similarly to *Complete Protection*, with the difference of keeping the decrypted class key even when locking the device, which protects against attacks requiring a system reboot. The last class is called *No Protection* [42] which is the default form of encryption, with encryption keys protected by the UID and stored in the file system. The naming of *No Protection* is misleading, as all data is encrypted on iOS and iPadOS.

On the application level, passwords and other sensitive data can be stored in the system service Keychain, which is implemented as an SQLite database, encrypted using AES-256-GCM [42]. As mentioned in Section 3.2.1, apps are isolated from each other, but it is possible to create access groups, enforced by the Apple Developer program [42], to allow multiple apps by the same developer to access the same Keychain items.

3.3 Structure Of An App

Applications for iOS are distributed as packages compressed as archives with the extension `.ipa` with the file structure shown in Figure 3.7, in the example of the messaging app *WhatsApp* [63].

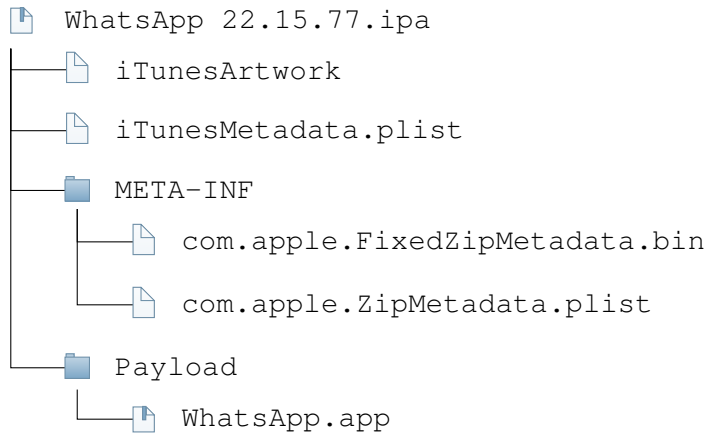

Figure 3.7: Structure of the IPA archive of WhatsApp version 22.15.77 [63].

The files *iTunesArtwork* and *iTunesMetadata.plist* contain metadata information about the app bundle and App Store purchase. The files *com.apple.FixedZipMetadata.bin* and *com.apple.ZipMetadata.plist* contain install instructions for the operating system, such as the command to extract the archive and checksums.

These files are more relevant to the App Store than to app developers, as app developers foremostly distribute the `.app` archive contained in the folder *Payload*, which is called the *application bundle* or *app bundle*.

An iOS app bundle consists of a variety of files, as shown in the example of the financing app *together* in Figure 3.8.

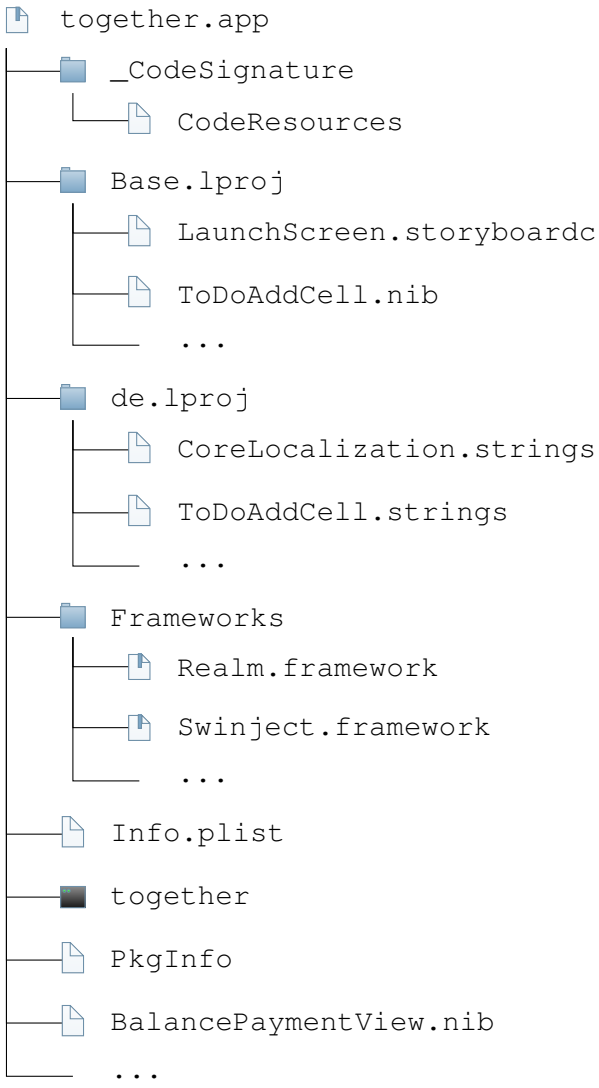

Figure 3.8: Excerpt of the structure of the app bundle of *together* version 1.0.0 [107].

The most important file is the executable binary file with the same name as the app bundle, i.e., *together*. It is the main entry point and contains the instructions performed by the processor. iOS binaries are built using the compiler infrastructure optimizer LLVM [80] for the ARMv7 architecture [16]. Starting with the iPhone 6, it also supports binaries compiled for ARM64 [57]. All configuration information and metadata required

by iOS is stored in the "information property list file" *Info.plist*. The settings available in this file are set by developers during the development process.

Furthermore, an app bundle can contain various resource files, such as user interface builder files (`.nib`), localization files (`.strings`), resource bundles (`.bundle`) and configuration files. In addition, it is possible to embed high-level resource files such as `.frameworks` containing binary libraries and further assets, providing a hierarchical dependency structure.

3.4 Attacks On The Operating System iOS

An operating system used by many devices on a global scale provides an interesting attack landscape, due to the high impact of a successful attack. Garcia et. al [18] performed an extensive iOS malware sample classification and categorized the common attack goals into spamming, data theft, fraud, and spying apps. Apple performs a vetting process of third-party apps created by external developers, to provide iOS users a monitored user experience [10]. According to Su et. al [32] this vetting process is mostly a black box but accepted in general as a mechanism against malware. Still, they were able to provide proof-of-concepts (PoC) to perform attacks such as device PIN cracking (breaking *Confidentiality*), blocking phone calls (breaking *Availability*), and posting to Twitter without the user's consent (breaking *Integrity*) using private APIs. These APIs can be used by third-party apps too, even though they are intended for internal applications only (as described in Section 3.2.1), thus Apple actively disallowing their usage in their developer license agreement [10] and denying the distribution of apps using them, they were still able to pass the review and submit their PoC app to the Apple App Store.

Xing et. al [35] analyzed 1,612 apps for macOS and 200 apps for iOS on high-impact security weaknesses regarding inter-app interaction services, especially hijacking elements in the Keychain, missing endpoint authentication for *NSConnection*, hijacking of URL schemes (as mentioned in Section 3.2.4) and spoofing the bundle identifier (on macOS), with at least 88.6% of the scanned apps being vulnerable.

Apple is actively introducing new security measures in their yearly major releases of iOS, while also improving the hardware with every iteration of their product line. Examples include the introduction of address space layout randomization (ASLR) in iOS 4.3 to ensure unpredictability of the memory layout, biometric user finger-print identification with TouchID in iOS 6, and kernel memory integrity protection mechanisms in iOS 10 [36].

Yixiang et. al [36] mention that iOS is hard to be analyzed by security researchers due to the complicated rights and encryption management, which lead to safety personnel breaking the limitations using loopholes in the hardware and software, to gain full control over the operating system. This technique is called *Jailbreak*, and even though it is intended to disable code signing and enable root access, it also allows attackers to run arbitrary code on the device, opening up vulnerability to worms and other malware.

Yixiang et. al [36] list various different malicious applications with "YiSpecter" and "Pegasus/Trident" directly attacking the underlying operating system. They describe YiSpecter as malware which installs malicious applications using enterprise certificates (which is a valid method of code signing) to collect user privacy information, while Pegasus uses multiple zero-day loopholes to control the kernel and execute code to disable iOS protection mechanisms.

Even though Apple is constantly improving the security of iOS and no anti-virus program is needed for it [2], exploiting the low-level kernel is still a vital approach to attack the operating system.

CHAPTER 4

Existing Guidelines For Mobile Security

Before diving into the process of developing new security guidelines for iOS mobile app developers, the results from the literature research are discussed by reviewing existing guidelines. These serve as a foundation for the following sections covering general-purpose guidelines for mobile platforms, as well as specific ones provided by the device manufacturer, and some with scientific background.

4.1 OWASP Top 10 Mobile Risks

The *European Union Agency For Network And Information Security* (ENISA) published its *Smartphone Secure Development Guidelines* [14] which are used by the *Open Web Application Security Project* (OWASP) to generate its guideline collection *OWASP Mobile Top 10* [95]. These lists aid developers in applying basic security to their mobile software. It focuses on presenting a list of risks with their associated threat, attack vector, impact, and example attack scenarios.

At the time of writing, the final list from 2016 is the latest released version. It includes the following ten risks, with a brief summary for each one:

M1: Improper Platform Usage [84] Misuse of a security platform feature, such as the secure storage Keychain, leading to insecure configurations or exposed confidential data. It can be prevented by using secure coding and configuration practices.

M2: Insecure Data Storage [86] Malware or adversaries with access to the device can exploit weak storage solutions to steal personally identifiable information or other

sensitive information. It is crucial to investigate how the app is handling intermediate and cached data.

M3: Insecure Communication [87] Data is commonly exchanged between clients and servers via the public internet or low-proximity communication technologies, such as Bluetooth. It is required to assume the network as insecure, apply cryptographic encryption and use a full trust chain without exceptions (such as self-signed certificates).

M4: Insecure Authentication [88] Vulnerabilities in the authentication functionality can lead to attacks. Therefore, multiple security patterns need to be applied, e.g., persistent authentication (commonly referred to as a "Remember Me" functionality) should never directly store the user's password.

M5: Insufficient Cryptography [89] Encrypted data should use strong, well-known encryption mechanisms, rather than custom-made solutions. Always assume an adversary can bypass built-in code encryption, and properly use the key management system.

M6: Insecure Authorization [90] Insecure authorization, e.g., hidden endpoints without further authorization checks, can be exploited by adversaries to gain more rights than originally intended for them. It can be circumvented by not relying on the information sent by the client itself, but instead verifying on the backend.

M7: Client Code Quality [91] Poor code quality can lead to security vulnerabilities such as buffer overflows. Even though it cannot be prevented, regular code maintenance and prioritizing system-critical issues, can improve the code quality.

M8: Code Tampering [92] Apps from third-party app stores and distribution channels, could have been modified during communication.

M9: Reverse Engineering [93] Attackers can get access to the app package, including its binary, and analyze it in a non-standard iOS environment. With decompilers, it is possible to reveal further information about the secure workings in the app. An obfuscator makes it harder to decompile back into readable code.

M10: Extraneous Functionality [85] Hidden functionality can be exploited to gain unauthorized access to other functionality. It can be prevented by examining app configuration settings, logs, and included code for sensitive data.

Advantages and Disadvantages

Even though the OWASP provides scenario examples and potential counter measures, they are not viable for the specific projects developers are working on, as they do not give a developer technical insight into the reasoning behind each guideline.

4.2 Apple Platform Security Guide

The *Apple Platform Security Guide* by Apple [42] is a document providing details on how security measures are implemented in their hardware and software products. It covers different topic areas, such as hardware security and biometrics, system security of the integrated hardware and software functions, as well as app and service security, providing a safe runtime ecosystem for apps and available communication technologies.

The document explains the different processes starting at the boot of an Apple device, such as the iPhone. It also covers the different hardware components required by modern computers, such as a *True Random Number Generator* [37]. Many services require to communicate with each other, such as the biometric authentication services Touch ID or Face ID, which allow access to the secure storage Keychain.

According to page 31 in Apple's documentation [42], the operating system iOS uses a modified C compiler toolchain to build the system bootloader, to prevent memory- and type-safety issues, such as buffer overflows or heap exploitation during the boot process. Furthermore, different hardware components, such as the T2 Chip, enforce valid firmware signatures at boot-time. This security feature also affects the signing process of kernel extensions, provided by third-party app developers.

On the topic of app security, the Apple Platform Security Guide [42] explains the full process of *Code Signing* (verifying the identity of the developer), *Sandboxing* (see Section 3.2.1), as well as *Entitlements* (see Section 3.2.3). It also includes details explaining how developers are authenticating themselves when integrating with system services, such as the systemwide password management.

Advantages and Disadvantages

The Apple Platform Security Guide [42] can be considered a public statement by Apple, declaring their commitment to providing security on their hardware and software, rather than a set of guidelines that can be applied by app developers. Based on different simpler topics, like which password entropy is identified as weak, up to specific cryptographical details, the document tries to target readers from a wide range of technical backgrounds.

The guide can be considered as a foundation for developers building software for Apple software platforms, but it hardly contains any guidelines which are applicable during the development process.

4.3 Proposed Scientific Guidelines

The journal article "A standard for developing secure mobile applications" [13] provides a list of measures software developers can undertake to improve security.

In a publication by Gasiba and Lechner [19], it is mentioned, that "[...] secure coding awareness must be based on [...] guidelines (and) [...] security awareness [...]". Therefore,

when training software developers to implement secure coding practices, one needs to rely on common security guidelines based on industry standards and policies.

A more creative approach of boosting security awareness relies on gamification, as proposed by Heid et.al [23]. Their approach consists of ongoing security training by asking trainees security or privacy-related questions about the apps installed on their device, in combination with the incentive of collecting virtual score points. They expect higher engagement of trainees, but also mention that the questions need to be adapted for each field, and automatically generating them from installed apps leads to technical and privacy-concerning issues.

The journal article "A standard for developing secure mobile applications" [13] provides a wide collection of guidelines, such as "[...] never include hardcoded references [...]" and highlights vulnerabilities of apps, operating systems and even cryptography concerns. While many guidelines can be applied by developers, the paper does not provide sufficient background into the reasoning, therefore not raising the security awareness a developer can use for further research. Additionally, it was published in 2014, which can be considered outdated due to the nature of fast changes in the computer science industry.

Advantages and Disadvantages

The proposed scientific guidelines are highlighting the importance of security guidelines for mobile app development and provide examples on how to avoid security vulnerabilities. Still, the background information on how exactly an attack could work is not mentioned in the paper and require readers to understand attack schemes (such as a Buffer Overflow Attack).

4.4 Conclusion Of Existing Guidelines

The literature research concludes that existing works either target corporations and/or institutions as a whole, or give a list of guidelines without the relevant context or proof of relevancy. This thesis will therefore combine these different approaches, by highlighting a list of applicable guidelines for the iOS mobile platform, and further explaining their fundamentals.

The proposed guidelines will aggregate information based on existing facts and applications examples to provide the necessary skills to detect more security issues by the reader themselves in the future.

CHAPTER 5

Developing Security Guidelines

Based on the principles of cybersecurity and research of the existing security guidelines, the following sections focus on developing a new set of security guidelines.

5.1 Purpose And Usage

At the time of writing application code, iOS app developers need to think of potential changes ahead of them. Applying best practices and guidelines from the beginning of a project and continuously revisiting them during development, reduces the need to adapt or even rewrite an application in the future.

The security guidelines proposed in the following sections support iOS app developers actively during their app development process. Furthermore, by providing examples the theoretical approaches can be extended with hands-on experience. This allows developers reading this thesis to understand the problem space and test it on a generic use case, as a preparation for applying the same principles to their specific projects.

The recommended approach to using these guidelines is reading through their descriptions, optionally acquiring further information and the required background knowledge to fully understand the workspace, and then working through the examples and solutions. Afterwards, the guidelines can be applied in regular intervals during the development process, such as part of a quality assurance (QA) process.

As an extended approach, it might be possible to further automate parts of the verification steps, e.g., by using static binary analysis to identify sensitive information.

5.2 Identifying Security Crunchpoints

Security crunchpoints are points in an ecosystem where security vulnerabilities have a higher chance of occurring and breaking the principles of the CIA triad presented in

Section 2.1. As mentioned before these principles must hold at all times, thus exploring features described in Section 2.4 leads to the discovery of security risks and potential mobile risks shown in Chapter 4.

The crunchpoint of network communication is the given transparency of the communication channel used to transfer data and the identity verification of the participant. As data can be intercepted, and therefore, leak information, counter measures can be used to protect its *Confidentiality* (see Section 2.1.1) and *Integrity* (see Section 2.1.2), such as cryptographical encryption mentioned in Section 4.1.

Sources of input data, with user-injected data being a prominent one due to the heavy use of user interaction, need to be validated for the data *Integrity* and the negative effects of bad input data affecting the system *Availability* (e.g., too much data, more than the system can handle, see Section 2.1.3). Furthermore, to store data between executions, data storages allow apps to store information and need to protect the *Confidentiality* and *Integrity* of sensitive data.

When storing data between execution times, the data needs to keep its *Confidentiality* and therefore, use a secure data storage. Using improper storage solutions instead of the secure solutions shown in Section 3.2.5 can break this principle.

Many attacks rely on insider know-how of a system, therefore the executable binary itself can be subject to modification (*Integrity*) to leak information or directly contain sensitive data which should not be easily available (*Confidentiality*). Techniques such as the sandbox execution model offered by the operating system, and described in Section 3.2.1, can be used to verify the *Integrity* of the application at runtime, while techniques such as anti-reversing techniques are used to complicate the process of reverse engineering.

5.3 Guidelines

To correct the issues occurring at the security crunchpoints mentioned in the previous section, a set of guidelines is developed. Each guideline explores its problem space, showcases an attack vector, and provides possible hardening or mitigation techniques.

5.3.1 Securing Network Communication

A common communication standard with external services is socket-based network connections using network protocols such as HTTP, WebSockets, and others. Due to the nature of the data being processed on remote endpoints outside of smaller networks, and the ability of mobile devices to be in a public network, the communication channel is not isolated. As the transmitted data often contain sensitive information, i.e., login credentials or authentication tokens, only the sender and the receiver should be able to read it. Therefore, attackers may try to actively circumvent measures taken to hinder this.

SSL Hijacking The plain-text network protocol HTTP can be upgraded to a secure connection using the HTTPS protocol, establishing a two-way handshake, and using strong cryptography for hiding the information inside network packages. An example is shown in Figure 5.1 which contains the same information as Figure 5.2 but without encryption. This requires the server software to offer HTTP connections and the app developer to request them. Due to various reasons, this might not be possible (e.g., server has no public interface, therefore it is not possible to establish a certificate chain), or due to convenience reasons (e.g., developer uses HTTP connection because the server is only available in a local or private network).

```
POST /post HTTP/1.1
Host: httpbin.org
User-Agent: curl/7.79.1
Accept: */*
Content-Length: 31
Content-Type: application/x-www-form-urlencoded

username=user%26password%3Dpass
```

Figure 5.1: An unencrypted HTTP request sent via *curl*.

```
0000   f4 d4 88 5b f2 01 c6 c3 6b 00 60 64 08 00 45 00   ...[....k.`d..E.
0010   00 34 68 61 00 00 e4 06 c0 15 22 e3 d5 52 ac 14   .4ha......"..R..
0020   0a 03 01 bb fa 6f 41 8f d3 c6 e3 ec 84 77 80 10   .....oA......w..
0030   00 6e 34 ce 00 00 01 01 08 0a 56 18 ca fe cb e9   .n4.......V.....
0040   2c 4e                                             ,N
```

Figure 5.2: TCP packet used for an encrypted HTTPS request, unreadable from the plain data.

In a study Fahl et. al [15] performed an analysis and captured network traffic from 884 apps from the App Store, with a 20.5% leaking information.

To improve the network security Apple introduced the *App Transport Security* (ATS), which forces apps to use HTTPS instead of HTTP URLs by default. This prevents unencrypted data from being sent over the public network by the operating system iOS, therefore an attacker will not be able to read the content of the network packages. But it is possible for developers to downgrade the ATS settings to disable certificate validation for testing or development purposes [24], which leads to the first guideline:

Guideline G.1: Upgrade Security

Downgrading the transport security settings of the application, even for development or testing purposes, is considered bad practice. Instead, the remote endpoint should upgrade its security and use encrypted communication channels.

To establish a secure connection, the server provides the client with an encryption certificate during the handshake phase, which is then used to encrypt the messages sent afterwards. These certificates can be created by anyone using OpenSSL, and if they are created by the developer themselves, they are called *Self-Signed Certificates*. Self-signed certificates provide security against passive attacks, such as intercepting network packages, but they do not protect against active Man-in-the-Middle (MITM) attacks. In its simplest form, the attacker can intercept the message (e.g., by using DNS Spoofing to resolve a hostname with the attacker server IP instead of the real server IP) with the real server certificate and instead forward its own attacker certificate to the client, as shown in Figure 5.3. The app would then use the attacker certificate to encrypt the messages, which would allow the attacker to further decrypt all the messages sent.

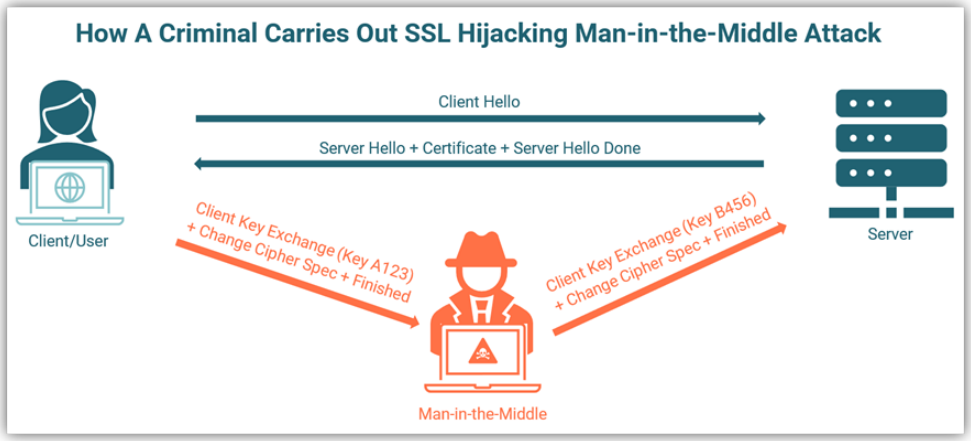

Figure 5.3: Graphic representing the communication during an MITM attack [100].

As a mitigation technique the application (and the operating system) needs to verify the validity of the received certificate and make sure it is associated with the expected remote endpoint, rather than the anyone else (i.e., the attacker). This is achieved by providing an additional certificate (the "intermediate certificate") which proves that the received server certificate is legit. But the intermediate certificate could also be manipulated by the attacker and therefore, they can also be linked to even more certificates, forming a certificate chain. The last certificate in the chain should be one of the global root certificates, provided by a *Certificate Authority* (CA), which are pre-installed on the device and therefore, can't directly be modified by an attacker.

The remaining option for an attacker to hijack the certificate chain, and force the client to accept their certificate, is to install a self-signed root certificate on the target's device. This can be done by using other techniques, such as phishing attacks, but it is also possible that the attacker and the target are the same entity, and the actual attack is a reverse engineering attempt to exploit the communication channel (e.g., to find an API key in the requests). Even though Apple is asking a user multiple times to confirm that they want to install another root certificate, an attacker cannot be stopped from doing

so on their own device, because it is not considered malicious behavior by itself. In this case the developers cannot fully trust all of the installed root certificates anymore, and therefore, cannot rely on the idea of "any of the installed root certificates can validate the certificate chain".

Guideline G.2: Attacker vs Target

The attacked target and the attacker can be the same entity, trying to exploit the software, making them the attack victim. Therefore, do not fully trust the environment and do not assume it has not been tampered with.

To handle an environment with untrustful entities, one proposed solution is using *Certificate Pinning*, defined by Mayer et al. [24] as preventing the usage of certificates issued by a malicious or compromised CA.

Certificate Pinning The practice of certificate pinning is performed by the client, which knows exactly which certificate should be sent by the server. This way the attacker cannot fool the client into using another certificate and has no option to read intercepted messages nor can they trick the client into accepting another root certificate.

As SSL certificates should still be short-lived, and only intermediate certificates should be long-lived, it is good practice to use a certificate chain and pin the provisioning intermediate CA certificate instead.

Starting with iOS 14.0 and iPadOS 14.0 (and macOS 11.0) it is possible to set a list of one or more allowed certificate authority certificates in the App Transport Security settings inside the `Info.plist` [56] such as the example shown in Figure 5.4.

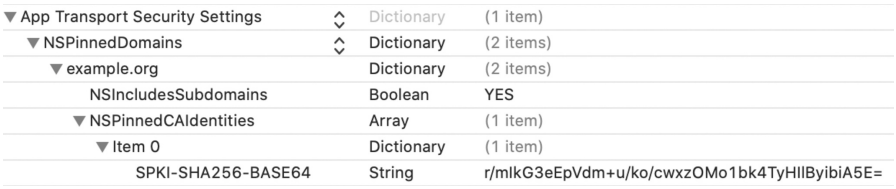

Figure 5.4: App Transport Security settings with pinned CA certificate [48].

Guideline G.3: Certificate Pinning

Use certificate pinning for certificates created by a Certificate Authority to protect secure communication channels from being tampered with.

5.3.2 Validating User-Injected Data

Inter-Process Communication (IPC) is the concept of communication between two related or unrelated processes, to exchange information and produce commands. According to Mayer et. al [24] iOS "[...] does not provide a proper IPC mechanism [...]" but instead relys on the concepts introduced in Section 3.2.4, such as custom URL schemes. As with any IPC mechanism, the two-way communication consists of **input data** available when receiving information, and **output data** sent to an external endpoint to provide it to that particular party, with different security aspects being applied to each one.

The communication between two processes requires rules known by both ends, also called a *protocol*. An example of a well-known protocol is the *Hypertext Transfer Protocol* (HTTP) [17] widely used for distributed information systems, such as the world wide web. When two parties agree to use a specific protocol, it is considered a communication *contract*.

In a perfect scenario, every party would be fully conformant to their contracts and that they could blindly assume that the sent and received data is exactly in the formats conforming to the rules defined in the protocol. Unfortunately, this scenario hardly ever exists in a real-world setting, because e.g., programmers make mistakes while programming the sender or receiver logic, the communication channel is modified as explained in Section 5.3.1, etc. Therefore, in scenarios such as inter-app communication, where the other party is often unknown, it can easily happen, that the communication contract is not well established. This requires the processes to take further measures so that the communication channel can still be trusted.

One option as a receiving app *rec-app* to overcome the missing trust in the sending app (send-app) is validating the input data. In this particular case, the rec-app does not know if the send-app has implemented the protocol correctly and holds the contract. Therefore, the rec-app performs an analysis on the input data transmitted and handles the data based on validity.

For custom URL schemes the following example scenario can be examined.

A social media app provides a custom URL scheme `social://` to allow other apps to post content into the user's profile. It does that by using the URL path as the command, e.g., `/post/image`, with the post data associated in the query, e.g., the data of an image encoded as a Base64 string in `?data=dGhpcyBpcyBub3QgcmVhbGx5IGFuIGltYWdlCg`. These combined result in the deep link URL shown in Figure 5.5. Due to improper programing the app does not validate the data at all, but instead directly starts to upload the file to a remote location so that other users can see it on the social media platform. Even though this might not result in a greate user experience (requiring less user feedback to perform a task), from a technical perspective it is plausible.

An attacker app may now try to inject malicious data into the social media app for any reason, e.g., to upload a huge file to overload the network bandwidth (breaking the

```
// Note: By using three forward slashes, the hostname is considered empty.
let url = "social:///post/image?data=dGhpcyBpcyBub3QgcmVhbGx5IGFuIGltYWdlCg"
```

Figure 5.5: Example deep link URL for a social media app

Availability described in Section 2.1) or maybe to upload illegal material on the remote endpoint, causing legal issues for the hoster.

These potential vulnerabilities and their potential effects lead to the following guideline:

Guideline G.4: Validate Input Data

Before processing input data, strictly validate the input data and (if applicable) check that the source can be trusted.

A potential validation strategy for the input data could be reading the image data, then examining the file size, the metadata such as image resolution and others, and afterward enforcing sensible limitations, e.g., max. 2MB file size or less than 2000px x 2000px resolution. These limitations depend on the expected data format and the use case of the data, therefore, need to be implemented by the developer on a use case basis.

In this example the protocol is the expected format of the URL, which is shared and known by others (because the social network wants others to use it). In cases where the URL scheme is undocumented, the attack URL components need to be constructed by analyzing the victim app.

The initial step is unarchiving the app bundle and opening the file `Info.plist`, which contains relevant metadata (as explained in Section 3.3). Apps that support custom URL schemes contain a field named `CFBundleURLTypes` holding the configuration used by the operating system. For the example social media app, a configuration might look like Figure 5.6.

The array with the key `CFBundleURLSchemes` contains the URL schemes which are handled by the app. Any iOS app installed on the same device as the target app can open an URL with the given scheme, and it will be handled in the target app.

To find the exact combination of host, path and query parameters required by the app, one needs to analyze the app by using reverse engineering. The techniques which can be applied to perform this are described in Section 5.3.4.

iOS apps handle URLs in different ways, depending on their deployment target OS version, and the underlying UI framework in use. Apps not supporting *Multi-Tasking* (multiple instances or windows of the same app) need to implement the delegation method shown in Figure 5.7, with more details available in the Apple Documentation [61]. This method is called by the application and needs to decide if the app handles the URL.

```
<dict>
    <key>CFBundleURLTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeRole</key>
            <string>Editor</string>
            <key>CFBundleURLSchemes</key>
            <array>
                <string>social</string>
            </array>
        </dict>
    </array>
</dict>
```

Figure 5.6: Example configuration for custom URL schemes in *Info.plist*.

```
class AppDelegate: UIApplicationDelegate {
    func application(
        _ app: UIApplication,
        open url: URL,
        options: [UIApplication.OpenURLOptionsKey : Any] = [:]
    ) -> Bool {
        return true /* if the url is handled */
    }
}
```

Figure 5.7: Application delegate handling an URL.

Alternatively, if the app supports Multi-Tasking, it implements a `SceneDelegate` as shown in Figure 5.8. Another approach when using the framework SwiftUI is calling the modifier `onOpenURL(action: (URL) -> Void)` providing a handler lambda function. For the scope of this thesis, and due to SwiftUI using UIKit at its core, the following example will focus on UIKit.

After receiving the `UIOpenURLContext` with the URL and optional metadata, a developer should validate the URL components and then examine the data from an application-level point of view. An example implementation for the social app is provided in Figure 5.8.

5.3.3 Using Secure Data Storage

The OWASP mention in *M2: Insecure Data Storage* [86] that development teams might assume attackers cannot get physical access to a victim's device. This assumption is not guaranteed and therefore, it is possible that the attacker can access the device directly, and further try to extract sensitive data with extended tools.

Even more critical are attacks on backups of iOS systems, as iOS supports the creation

```swift
class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {
        // Iterate each URL Context
        for context in URLContexts {
            let url = context.url
            guard url.scheme == "social" else {
                return // invalid scheme
            }
            guard url.path == "/post/image" else {
                return // invalid path
            }
            let queryItems = URLComponents(string: url.absoluteString)?.queryItems ?? []
            // Find the 'value' of the query item named 'data'
            guard let base64EncodedData = queryItems.first(where: { $0.name == "data" })?.value
    ↪ else {
                return // missing query item data
            }
            // (Optional) Validate the source application bundle if necessary
            guard context.options.sourceApplication == "com.not-the-attacker.app" else {
                return // The url was called from an invalid source
            }
            // Read and convert the Base64 encoded data into an image
            guard let data = Data(base64Encoded: base64EncodedData) else {
                return // invalid Base64 String
            }
            guard let image = UIImage(data: data) else {
                return // invalid image data
            }
            // -- Apply the validation --
            // Validate the data size is less than 2MB
            guard data.count < 2 * 1024 * 1024 else {
                return // Data too large
            }
            guard image.size.width > 2000 && image.size.height > 2000 else {
                return // Image resolution too large
            }
            // Validation successful, continue using the given data
            upload(imageData: data)
        }
    }
}
```

Figure 5.8: Example validation for handling a Base64 encoded image.

of backups and the restoring of devices in case of unintended system behavior or when swapping the device. A backup includes the data of apps stored on the device and by backing up the data onto an external device, e.g., by connecting it to a device running macOS, the app data can leave the iPhone or iPad, and become accessible in a completely different environment.

Suarez et. al [33] state that, the iOS hardware encryption blocks an attacker using offline attacks, because the data is decrypted at system runtime. This means that, if an attacker can extract data from a running device, it might still be accessible. Even though iOS backups can be encrypted, they can also be stored unencrypted, and reconstructed into a readable file structure. This opens up again the opportunity of exploiting the device data without having a running device nor even having physical access.

Due to these risks, developers need to consider how sensitive the processed data is, how it should be stored, and by whom it should be accessible. In a more concise example, a developer needs to consider if sensitive data should be stored directly on the file system

or instead in the secure data storage Keychain. Every solution comes with trade-offs, e.g., the Keychain uses strong encryption as explained in Section 3.2.5, but might not be as accessible as desired, i.e., accessible without user interaction when using protection classes.

One software to analyze a backup from an iOS device on macOS is *iMazing* [71]. A backup is organized in a hierarchy of folders and files using hexadecimal names without extensions. Utilities like iMazing can reconstruct the actual file hierarchy and associate them to the installed apps, thus providing access to the files of each application sandbox.

In example of the document-scanning app *SwiftScan* [81] the scanned files are stored in the app's document directory and are also included in the backup. In Figure 5.9 the backup includes a scanned PDF document `Bank Account.pdf`, which can now be extracted from the backup and opened with a PDF document viewer. The bank statement might contain sensitive data and is therefore leaking information and breaking the principle of *Confidentiality* (see Section 2.1.1).

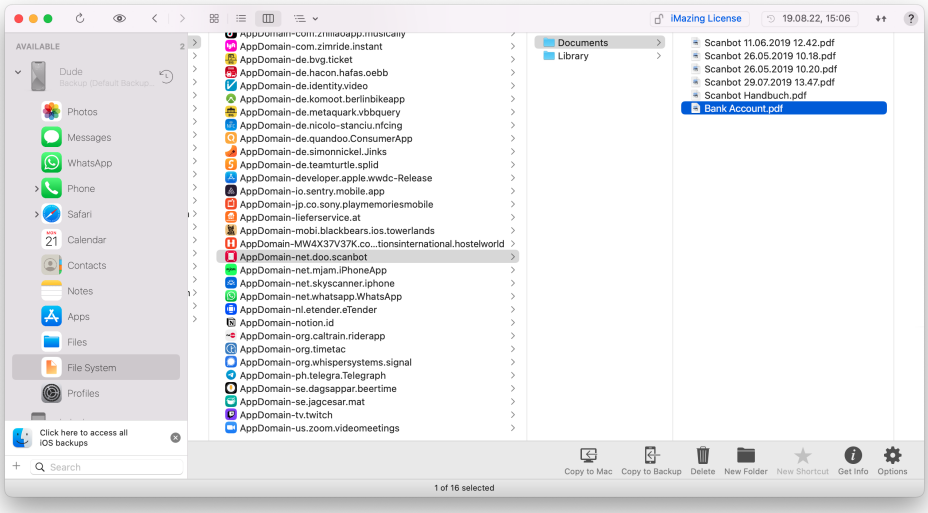

Figure 5.9: Apps and files stored by the macOS app *SwiftScan* version 9.6.20 [81] giving direct access to the bank statement PDF file containing confidential information.

Developers might consider encrypting the local file, but cryptography in apps will always require the usage of an encryption key that needs to be available on device. As soon as a security key is part of the application itself, it can be subject to reverse engineering and requires further hardening as it will be presented in Section 5.3.4.

All of these insights lead to the following guideline for using secure storage:

Guideline G.5: Use Secure Storage

When storing data on a device, consider how sensitive that data is, and if it needs to be confidential then use the secure data storage *Keychain* instead of the local file system.

One use case that needs to be highlighted is using `UserDefaults`, which is a built-in class to interact with the defaults system, storing settings in a property list file format. This system allows developers to easily save basic data formats such as numbers, text, and binary data. Introduced in iOS 2.0 it is a mature framework used by Apple and many others as the first choice for storing data. But these files are also saved as unencrypted files into the documents folder of the app sandbox, therefore the same issues apply.

Guideline G.6: Security Of UserDefaults

The property list file format is not encrypted, and therefore, do not store any sensitive data using the `UserDefaults` and use the Keychain instead.

5.3.4 Protecting App Binary Against Reverse Engineering

Reverse Engineering is the process of converting a compiled program back into a human-understandable format to extract additional information on its intended functionality [1]. When performing reverse engineering the adversary wants to gain insights on e.g., login procedures, encryption techniques, used libraries, and others.

As application binaries are shipped to the devices of end-users, they live in an (from the developer's perspective) external environment, which needs to be considered insecure. Apple applies a digital rights management technology called *FairPlay DRM* which encrypts the app binary as a protection layer to prevent static analysis [11]. This protection mechanism can be circumvented by using utilities such as *Clutch* [79] or *frida* [99], which can be used to export an unencrypted iOS binary from a jailbroken iOS device [35]. An extensive guide on setting up a reverse engineering setup can be found in the *OWASP Mobile Security Testing Guide* [94].

Due to the availability of these techniques, there is always a chance of an adversary trying to reverse engineer an app, which leads to the following guideline:

Guideline G.7: Expect Decompilation

Expect apps to be unarchived and decompiled, therefore do not assume the app is only available in a closed environment.

To further understand the effects of reverse engineering it is helpful to take a closer look at the structure of an iOS app binary. The following example application is a minimal app using the framework UIKit, which can be created by using the default *iOS > App > Storyboard* template provided by the Xcode IDE.

After creating the project, adapt the default `ViewController.swift` to match the code shown in Figure 5.10. After running the example project, the compiled binary is usually located in the default derived data folder located at `~/Library/Developer/Xcode/Derived Data` on macOS.

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        print("Hello World")
    }
}
```

Figure 5.10: Minimal UIKit view controller printing 'Hello World' on load.

Binaries for iOS are compiled into the *Mach object file format* (Mach-O) [51] which is an executable file format used by iOS and macOS. A Mach-O file contains a header, which is used as the main entry point, and multiple memory sections. Using the command line tool `objdump` [75] it is possible to print a summary of the section header names, as shown in Figure 5.11.

Two sections which are significantly relevant for this reverse engineering process are the `__text` and the `__cstring` sections. Apple mentions in their documentation [51] that the `__text` section contains the executable code, while the `__cstring` section contains literal strings, which are strings encapsulated in quotes in the actual source code.

To list all literal character strings in a binary, the command line utility `strings` can be used, which finds the printable strings in the binary. By running the utility, the output log shown in Figure 5.12 clearly prints the `"Hello World"` used in the example source code. This can be considered harmless for this specific example, but poses an immediate security issue when using, e.g., hard-coded license keys.

Instead of hard-coding a sensitive value into the binary a developer should therefore consider other approaches instead, e.g., generating relevant encryption keys at the initial run after the app installation, and then saving it into the secure data storage for further usage. In cases where generating the data is not possible, e.g., an API key which authorizes the app client with a remote endpoint, it should be considered to apply code obfuscation to harden the extraction.

```
$ objdump -h MinimaliOSApp

MinimaliOSApp:  file format mach-o arm64

Sections:
Idx Name              Size     VMA              Type
  0 __text            00001c90 0000000100003c08 TEXT
  1 __stubs           000001e0 0000000100005898 TEXT
  2 __objc_methlist   000000e8 0000000100005a78 DATA
  3 __objc_methname   00000d20 0000000100005b60 DATA
  4 __cstring         00000e5a 0000000100006880 DATA
  5 __const           00000320 00000001000076e0 DATA
  6 __swift5_typeref  0000009b 0000000100007a00 DATA
  7 __swift5_fieldmd  00000058 0000000100007a9c DATA
  8 __swift5_types    00000010 0000000100007af4 DATA
  9 __swift5_entry    00000004 0000000100007b04 DATA
 10 __swift5_builtin  00000014 0000000100007b08 DATA
 11 __swift5_reflstr  0000002a 0000000100007b1c DATA
 12 __swift5_assocty  00000030 0000000100007b48 DATA
 13 __swift5_proto    00000018 0000000100007b78 DATA
 14 __entitlements    0000011a 0000000100007b90 DATA
 15 __unwind_info     00000170 0000000100007cac DATA
 16 __eh_frame        000001dc 0000000100007e20 DATA
 17 __got             00000178 0000000100008000 DATA
 18 __const           00000140 0000000100008178 DATA
 19 __objc_classlist  00000018 00000001000082b8 DATA
 20 __objc_protolist  00000020 00000001000082d0 DATA
 21 __objc_imageinfo  00000008 00000001000082f0 DATA
 22 __objc_const      00000728 000000010000c000 DATA
 23 __objc_selrefs    00000098 000000010000c728 DATA
 24 __objc_protorefs  00000020 000000010000c7c0 DATA
 25 __objc_classrefs  00000018 000000010000c7e0 DATA
 26 __objc_data       00000d80 000000010000c7f8 DATA
 27 __data            000000d0 000000010000d578 DATA
 28 __bss             00000300 000000010000d650 BSS
```

Figure 5.11: Mach-O sections printed using the command line utility *objdump*.

```
$ strings MinimaliOSApp
...
Hello World
...
```

Figure 5.12: Printable strings found using the command line utility *strings*.

This leads to the following guideline:

Guideline G.8: Obfuscate Strings

Obfuscate plain strings to make them harder to extract.

Code Obfuscation

Code Obfuscation is the process of transforming source code to make it hard to understand by humans, but still executable by computers, to increase the cost of reverse engineering

until it is ineffective or uneconomical [34]. Wang et. al [34] released a full paper on their experiences with obfuscating iOS apps.

An example for code obfuscation is replacing the leaking literal string with a byte sequence instead, as presented in Figure 5.13.

```
/// Example password check using literal string
/// --> can be found in '__cstrings' section
func isPasswordValid(_ password: String) -> Bool {
    return password == "super-secret"
}

/// Example password check using byte sequence
/// --> cannot be found in '__cstrings' section
func isPasswordValid(_ password: String) -> Bool {
    password == String(bytes: [
        0x73, 0x75, 0x70, 0x65, 0x72, 0x2d,
        0x73, 0x65, 0x63, 0x72, 0x65, 0x74
    ], encoding: .utf8)
}
```

Figure 5.13: Example of obfuscating a literal string with a byte sequence.

As a conclusion for anti-reverse engineering techniques, developers need to remember that obfuscation is not mitigating a vulnerability, but instead make it harder to exploit.

Anti-reversing techniques

In general, anti-reversing checks are performed at runtime, especially at the start of the app, to verify the runtime environment has not been modified to gain more capabilities than expected. Reguła et al. [109] has created an open source collection of anti-reversing (and anti-tampering) detection checks called *IOSSecuritySuite*. At the time of writing this thesis, the toolset includes the following anti-reversing modules:

Jailbreak detector Detects if the device iOS is jailbroken by testing suspicious custom URL schemes and files, unusually open file system permissions (e.g., `/root/` is writable), and dynamic libraries used to tweak the system [111]. As this check only looks for suspicious files known during development, the check can also fail to detect a Jailbreak.

Debugger detector and deny Detects if the process is being debugged by using `sysctl` to see if the process tracing flag `P_TRACED` is set [110].

Reverse engineering tools detector Checks the environment for suspicious files and dynamic libraries, e.g., frida [99], and if ports known to be used for debugging servers are open. [112]

All of these checks can be applied to block users from running the application in a modified environment, e.g., a jailbroken iOS device.

Guideline G.9: Use Anti-Reversing

Anti-reversing techniques can be used to block users from running the app in an invalid environment.

Unfortunately, these checks are part of the executable code of the app itself and can be disarmed by modifying the binary. Therefore, when using anti-reversing checks, it is also necessary to apply protections against binary tampering.

5.3.5 Protection Against Binary Tampering

The process of binary tampering is adapting the executable binary to behave differently as originally intended. Possible use cases include changing conditional statements (e.g., to circumvent a license key check), unlocking further functionality (e.g., enabling a debug server), artificially generating in-app content (e.g., in-game currencies), and malicious purposes, such as stealing sensitive data from users of the tampered app.

A counter measure to this threat is code signing, which is the process of cryptographically signing the apps submitted to the App Store. As code signing is required for the App Store distribution of iOS apps, developers need to enroll in the iOS Developer Program by Apple, which also demands an identity verification. Afterwards a developer is provided with a signing certificate uniquely identifying the creator of the app signatures.

D'Orazio et al. [12] mentions that code signing can be circumvented by using jailbroken devices and connecting via SSH onto an iOS device to remotely execute a tampered application binary. Furthermore, it is also possible to modify a binary and execute it on a non-jailbroken device, as shown by the OWASP Mobile Security Testing Guide [96]. After modifying an app binary, with already removed FairPlay DRM encryption, an attacker can also create their own Apple Developer account and re-sign any app with their own certificate. The Mobile Security Testing Guide even showcases an example of adapting an compiled app, without debugging enabled (which is default for apps distributed on the Apple App Store), to allow remote instrumentation by loading the dynamic library `FridaGadget.dylib` [96], adding the entitlement `get-task-allow` not available in App Store distributed apps, and signing it again using their attacker certificate.

One option of checking if the application has been re-signed is validating the checksum or parsing the file `embedded.mobileprovision` in the app bundle. This has been publicly discussed by developers on the question and answer forum StackOverflow [101] with the conclusion that these validation steps produce a larger development overhead and can block modified binaries from running, but can also be easily circumvented.

Anti-tampering checks are similar to anti-reversing checks but focus on the integrity of the executable binary and its resources. The software collection IOSSecuritySuite

introduced in Section 5.3.4 also provides a *File Integrity Verifier*, used to calculate, and validate the checksum of files in the app bundle.

As the anti-tampering checks are also part of the released app binary, these checks are useless against active binary modification, as an attacker can also exchange the expected checksums with modified ones. Furthermore, if the executable checksum is part of the executable binary, changing the checksum will change the binary, and vice-versa, causing a circular dependency and making the development process complicated.

Guideline G.10: Anti-Tampering Techniques Can Be Ineffective

When applying anti-tampering techniques do not assume that they cannot be circumvented, therefore consider exactly against whom the app should be protected.

As a conclusion of the anti-reversing and anti-tampering measures, even the authors of the OWASP Mobile Security Testing Guide states that "[...] ultimately, the reverse engineer always wins" [96], leading to the conclusion in this thesis, that these mechanisms only increase the effort for an attacker, and the effectiveness should be considered with care.

CHAPTER 6

Testing Proposed Guidelines

To verify the developed guidelines proposed in Chapter 5, it is necessary to validate them.

In the context of this thesis, a security guideline is declared as validated, if its test yields that adversaries need more steps or more time to perform an exploit. Testing cannot be used to completely mitigate a security vulnerability, as everything can be exploited with enough effort, time, and resources. Instead, security guidelines should increase these amounts into unreasonable scales.

6.1 Association Guidelines And Tests

Table 6.1 presents the association matrix between guidelines developed in Section 5.3 and the test cases shown in the upcoming sections. For each guideline at least one validation test is provided, and a single validation test can prove multiple guidelines.

Section	Guidelines	Validation Test
5.3.1 Securing Network Communication	G.1 Upgrade Security G.2 Attacker vs Target G.3 Certificate Pinning	6.3.1 Testing Secure Network Communication Techniques
5.3.2 Validating User-Injected Data	G.4 Validate Input Data	6.3.2 Hardening Input Data Formats
5.3.3 Using Secure Data Storage	G.5 Use Secure Storage G.6 Security Of UserDefaults	6.3.3 Hardening Against Backup Exploits
5.3.4 Protecting App Binary Against Reverse Engineering	G.7 Expect Decompilation G.8 Obfuscate Strings	6.3.4 Applying Code Obfuscation
	G.9 Use Anti-Reversing	6.3.5 Applying Anti-Reversing And Anti-Tampering Techniques
5.3.5 Protection Against Binary Tampering	G.10 Anti-Tampering Techniques Can Be Ineffective	6.3.5 Applying Anti-Reversing And Anti-Tampering Techniques

Table 6.1: Association matrix between developed security guidelines and validation tests.

6.2 Validation Process

The validation process of a security guideline is used to prove that the proposed measures are effectively contributing to the hardening against the security issue of the associated product.

The reasoning behind this approach can be reduced to the comparison of the risk imposed by the correctness of the software code and the risk to human operators. For example, the requirement of using password protection on a system can easily be circumvented if the human operator writes down the password at a publicly accessible place.

Therefore, the following sections will validate each guideline by setting up a testing environment and then displaying how the vulnerability can be exploited. For each exploit, the proposed guideline introduced in Section 5.3 is applied by providing hardening or mitigation techniques, therefore proving the guidelines effectiveness by causing an unreasonable higher effort to successfully apply the given exploit again.

6.3 Validation Tests

Each validation tests is based on the validation process described in the previous Section 6.2, with the results summarized in Section 6.4.

6.3.1 Testing Secure Network Communication Techniques

Exploiting the network communication of an app means tapping into the communication channel and reading/modifying the sent data. Counter measures improving the security of this issue are presented in Section 5.3.1. One example of exploiting insecure communication channels is using the plain text protocol HTTP without any encryption, thus making it possible to extract sensible information by inspecting the TCP packets transmitted via the network.

To reproduce this scenario, it is necessary to set up a validation environment with a communication channel between an HTTP server accepting a request with the HTTP method `POST`, allowing a client to send a payload containing sensitive data such as user credentials, and an iOS app using an HTTP client to communicate with the server.

One available option for the server is the simple HTTP request and response service *httpbin.org* [78]. This service provides a RESTful HTTP API with a couple of convenience endpoints useful for testing and inspecting HTTP requests, e.g.,
`GET https://httpbin.org/headers` returning the request's header fields, or
`POST https://httpbin.org/post` accepting the HTTP method `POST` and returning the full request as a JSON (JavaScript Object Notation) object.

For convenience reasons, this setup uses the publicly hosted server [78], but alternatively it is also possible to run the service locally by using Docker [72] and exposing the port `80` of the container. On a sidenote it is considered best practice not using the port `80` on the host system, as this port might already be used by a different process, so instead map it to a different one, like `8000`, as shown in Figure 6.1.

```
$ docker run -it --rm -p 8000:80 kennethreitz/httpbin
```

Figure 6.1: Shell command used to run an interactive instance of *httpbin* using Docker, available on the host port 8000, which is deleted after the process terminates.

For the app client it is possible to use the functionality offered by iOS. First create a new project (or use an existing one) inside Xcode and adapt the source code in the file `AppDelegate.swift` to include the client code shown in Figure 6.2.

With this setup it is possible to run the app in the iOS Simulator using Xcode. It will then send requests continuously every couple of seconds, which can then be examined locally on macOS. Logging and inspecting the communication channel, and thus TCP packets sent through the network interface, can be done using the network sniffer *Wireshark* [106]. Analyzing these TCP packets enables an attacker to dissect and extract sensitive data.

After installation of Wireshark, the app should run in the iOS Simulator to continuously send the HTTP requests containing the credentials in the HTTP body. By running Wireshark on the network interface used to communicate with `httpbin.org` (or selecting the loopback interface, if running the Docker container locally) the packets begin to show

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions
    ↪ launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {
    // Run the loop from a background queue,
    // so that the UI does not get blocked.
    DispatchQueue.global(qos: .background).async {
        while true {
            let url = URL(string: "http://httpbin.org/headers")!
            // Create the URL Request with the HTTP method 'POST'
            // and credentials in the HTTP body
            var request = URLRequest(url: url)
            request.httpMethod = "POST"
            request.httpBody = "username=hero&password=secret"
                .data(using: .utf8)!
            // Send the request asynchronously
            URLSession.shared.dataTask(with: request) { _, _, error in
                if let error = error {
                    print("Failed to send request", error)
                } else {
                    print("sent data")
                }
            }
            .resume()
            sleep(2)
        }
    }
    return true
}
```

Figure 6.2: Example client code sending an HTTP POST request to httpbin.org every few seconds.

up in the log. When inspecting the HTTP request sent to the endpoint, the username and password are clearly visible and readable as shown in Figure 6.3.

One option to intercept and manipulate a network communication channel is the macOS application *Proxyman* [97]. After installing the app, it setups itself as a proxy for the system, relaying all communication locally and logging them as shown in Figure 6.4.

After setting up the environment the security guideline can be validated.

Hardening against passive attacks by encrypting plain requests

By changing the URL in line 6 in Figure 6.2 from the protocol `http` to `https` the end-to-end encryption is enabled, causing Wireshark not being able to read the content of the requests anymore (see Figure 6.5).

This proves guideline G.1 stating that the usage of an encrypted channel hardens against passive network sniffing.

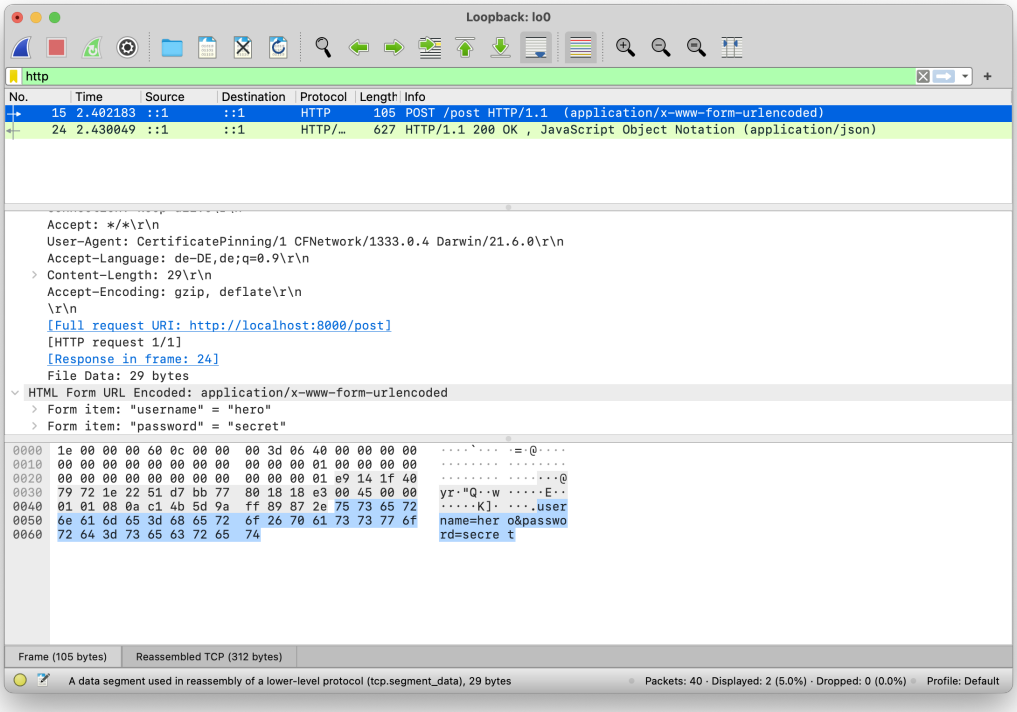

Figure 6.3: Unencrypted HTTP request found network log using Wireshark.

Decrypting an encrypted communication channel

To be able to analyze encrypted network packets, the attacker needs to have access to the encryption keys and decrypt the messages. In most cases these keys are kept confidential by the service provider and therefore, it is assumed that the attacker does not have access to the original key used.

Instead, an attacker may perform a Man-In-The-Middle (MITM) attack and tamper with the establishment of the secure channel. During this establishment process, the server returns its encryption certificate to the client, which is used by the client to create messages only decryptable by that specific server. In an MITM attack the attacker intercepts the certificate from the server and returns its own encryption certificate to the app client instead, allowing the attacker to decrypt any further messages sent by the client.

One option of interception is running a proxy server, which relays all the communication between the client and the server. After enabling the Proxyman proxy server, it will intercept server certificates and instead return its own self-signed certificates to the app client, which will cause the `Internal Error` shown in Figure 6.6. This is caused by the app client failing to verify the certificate as described in Section 5.3.1, with the error

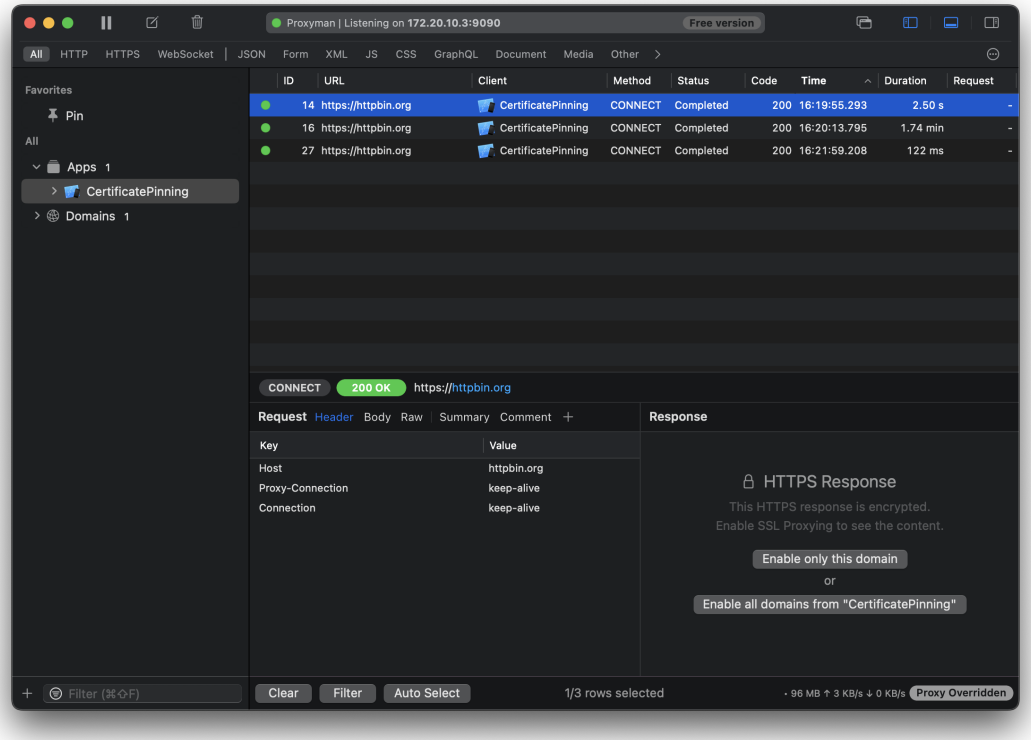

Figure 6.4: Encrypted HTTP requests relayed by Proxyman.

logged to the console in Xcode (see Figure 6.7).

To re-establish a trusted certificate chain, the attacker needs to install the proxy root certificate on the victim's iOS device, so that the validation succeeds without changing the app security settings (which is usually not possible, as it requires the attacker to create a modified app and install it on the victim's device).

To install a root certificate, it needs to be downloaded to the device as an `.p12` file, which is then detected as a configuration profile, as shown in Figure 6.8. Afterwards the device user needs to open *Settings* > *About* > *Certificate Trust Settings* to install and activate the certificate, by accepting an additional confirmation dialog (see Figure 6.8). When using Proxyman in the iOS simulator, this installation process is automated by the application.

After the root certificate is installed, Proxyman is now able to intercept requests and decrypt them, showing once again the sensible data in Figure 6.9

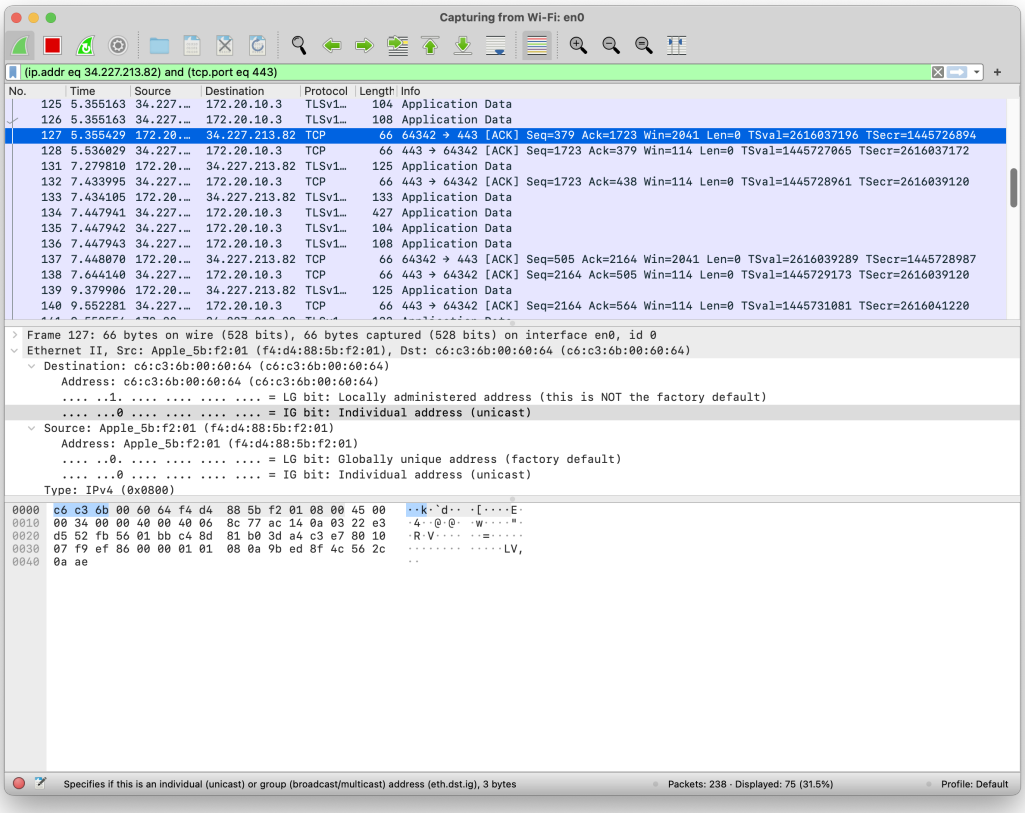

Figure 6.5: Encrypted HTTP request found network log using Wireshark

Hardening encrypted communication

To harden the communication channel and blocking these kinds of SSL hijacking attacks, a developer can apply Certificate Pinning.

As explained in Section 5.3.1 this is done by computing the identity of the intermediate certificates issued by a Certificate Authority (CA) and adding it to the *App Transport Security Settings*.

A guide from Apple [48] describes the pinned public key identity as a "[...] Base64-encoded SHA-256 digest of an X.509 certificate's DER-encoded ASN.1 Subject Public Key Info structure". The process of generating a public key identity can be summarized into the following steps:

1. Fetch the CA root or intermediate certificate using `openssl` for the domain `httpbin.org` (see Figure 6.10). This command connects to the remote endpoint, establishes an TLS connection, and prints the certificate chain with a `depth`

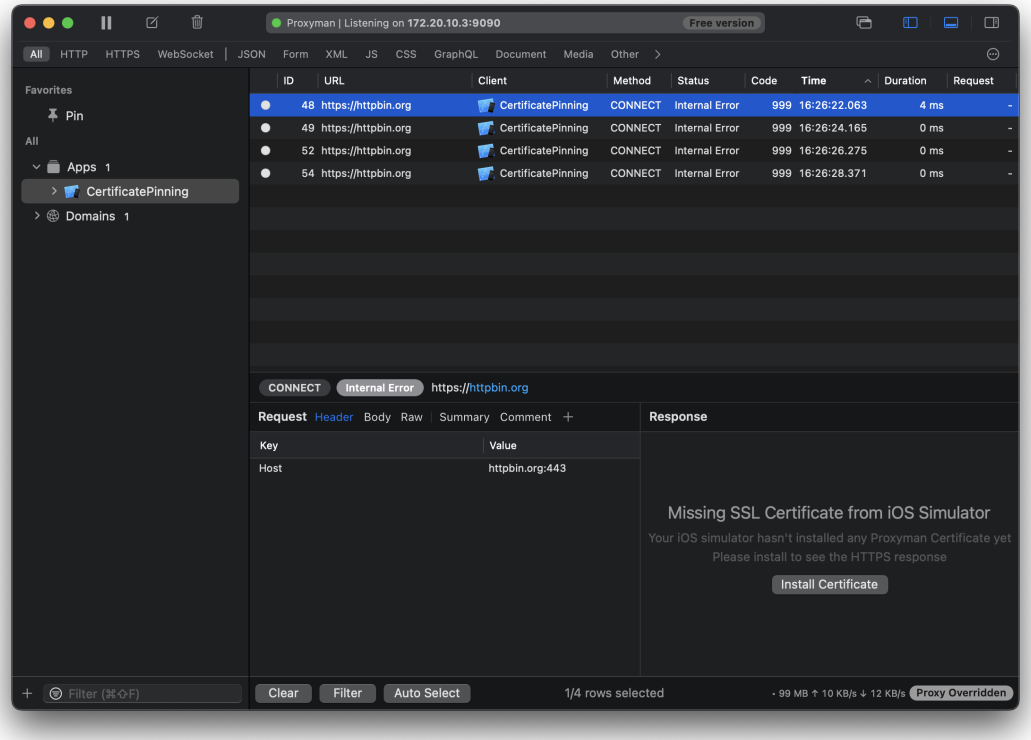

Figure 6.6: Proxyman logging internal errors

counter. In the given example the certificate at `depth=4` is the root certificate, with the ones at `depth=3` to `depth=1` acting as intermediate certificates and the leaf certificate at `depth=0`.

2. Due to the parameter `-showcerts` the output also displays the certificates in the *Privacy Enhanced Mail* (`pem`) format, encoded as Base64 encoded strings between the tags `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----`. Select one of the root or intermediate certificates and save it (including the tags) to a local file called `ca.pem`.

3. Extract the public key from the certificate using the command `openssl x509` and write it to a local file called `ca.pubkey.pem` in the format `pem`, as presented in Figure 6.11.

4. Use the extracted public key to compute its identity as shown in Figure 6.12 by converting it from the `pem` format to the `der` format, computing a SHA256 digest and encoding the binary output to Base64.

The Base64 encoded certificate, which should be pinned, needs to be saved to a file

```
Error Domain=NSURLErrorDomain
Code=-1202
"The certificate for this server is invalid. You might be connecting
   ↪ to a server that is pretending to be “httpbin.”org which could
   ↪ put your confidential information at risk."
UserInfo={
    ...
    NSErrorPeerCertificateChainKey=(
        "<cert(0x11f821800) s: httpbin.org i: Proxyman CA (7 Apr
   ↪ 2022, turbo.local)>"
    ),
    NSErrorClientCertificateStateKey=0,
    NSErrorFailingURLKey=https://httpbin.org/post,
    NSErrorFailingURLStringKey=https://httpbin.org/post,
    ...
}
```

Figure 6.7: Error message logged by the app failing to validate the certificate chain of the proxy certificate.

`ca.pem` and can then be used as an input in the command shown in Figure 6.12 to generate the public identity key.

After adding the generated public key identity to the `Info.plist` settings, as shown in Figure 5.4, the requests will fail once again with the error "An SSL error has occurred and a secure connection to the server cannot be made", because even though the root certificate of the proxy is trusted by the system, the app itself only trusts certificates issued by the expected certificate authority.

After disabling the proxy, a secure connection can be established once again and the requests finish successfully. This proves that certificate pinning can be used to circumvent SSL hijacking, as defined in guideline G.3.

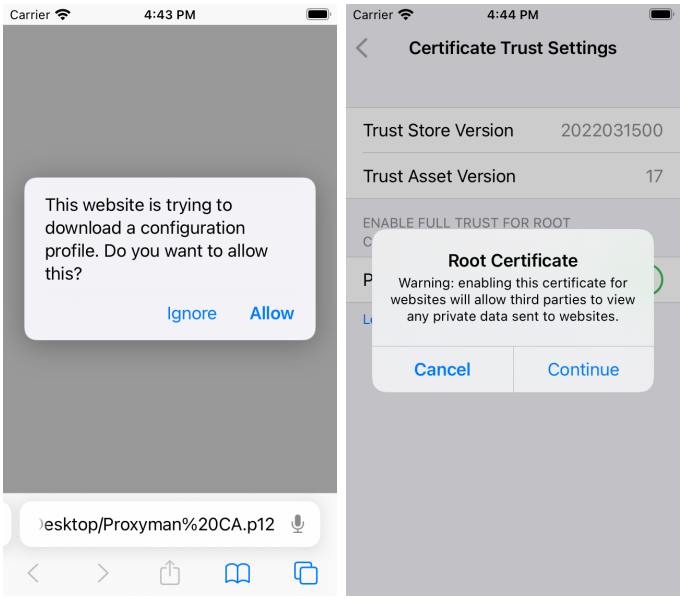

Figure 6.8: Steps required to install a new root certificate on an iOS device.

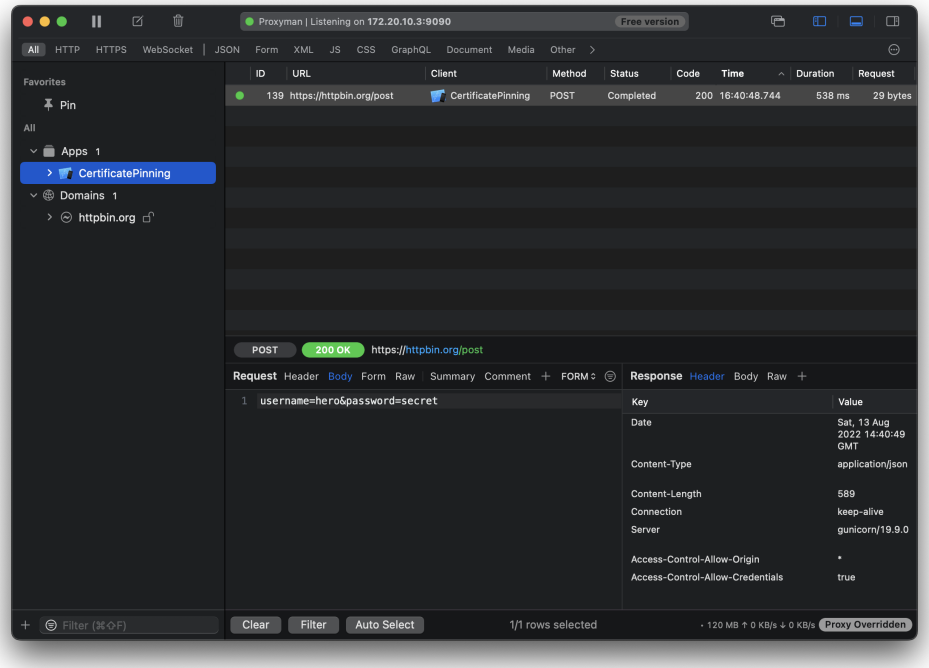

Figure 6.9: Decrypted request shown in Proxyman

```
$ openssl s_client -connect httpbin.org:443 -showcerts < /dev/null

depth=4 C = US, O = "Starfield Technologies, Inc.", OU = Starfield Class 2 Certification Authority
verify return:1
depth=3 C = US, ST = Arizona, L = Scottsdale, O = "Starfield Technologies, Inc.", CN = Starfield
    ↪ Services Root Certificate Authority - G2
verify return:1
depth=2 C = US, O = Amazon, CN = Amazon Root CA 1
verify return:1
depth=1 C = US, O = Amazon, OU = Server CA 1B, CN = Amazon
verify return:1
depth=0 CN = httpbin.org
verify return:1
---
...
2 s:/C=US/O=Amazon/CN=Amazon Root CA 1
i:/C=US/ST=Arizona/L=Scottsdale/O=Starfield Technologies, Inc./CN=Starfield Services Root
    ↪ Certificate Authority - G2
-----BEGIN CERTIFICATE-----
MIIEkjCCA3qgAwIBAgITBn+USionzfP6wq4rAfkI7rnExjANBgkqhkiG9w0BAQsF
ADCBmDELMAkGA1UEBhMCVVMxEDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNj
b3R0c2RhbGUxJTAjBgNVBAoTHFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4x
OzA5BgNVBAMTMlN0YXJmaWVsZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1
dGhvcml0eSAtIEcyMB4XDTE1MDUyNTEyMDAwMFoXDTM3MTIzMTAxMDAwMFowOTEL
MAkGA1UEBhMCVVMxDzANBgNVBAoTBkFtYXpvbjEZMBcGA1UEAxMQQW1hem9uIFJv
b3QgQ0EgMTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALJ4gHHKeNXj
ca9HgFB0fW7Y14h29Jlo91ghYPl0hAEvrAIthtOgQ3pOsqTQNroBvo3bSMgHFzZM
9O6II8c+6zf1tRn4SWiw3te5djgdYZ6k/oI2peVKVuRF4fn9tBb6dNqcmzU5L/qw
IFAGbHrQgLKm+a/sRxmPUDgH3KKHOVj4utWp+UhnMJbulHheb4mjUcAwhmahRWa6
VOujw5H5SNz/0egwLX0tdHA114gk957EWW67c4cX8jJGKLhD+rcdqsq08p8kDi1L
93FcXmn/6pUCyziKrlA4b9v7LWIbxcceVOF34GfID5yHI9Y/QCB/IIDEgEw+OyQm
jgSubJrIqg0CAwEAAaOCATEwggEtMA8GA1UdEwEB/wQFMAMBAf8wDgYDVR0PAQH/
BAQDAgGGMB0GA1UdDgQWBBSEGMyFNOy8DJSULghZnMeyEE4KCDAfBgNVHSMEGDAW
gBScXwDfqgHXMCs4iKK4bUqc8hGRgzB4BggrBgEFBQcBAQRsMGowLgYIKwYBBQUH
MAGGImh0dHA6Ly9vY3NwLnJvb3RnMi5hbWF6b250cnVzdC5jb20wOAYIKwYBBQUH
MAKGLGh0dHA6Ly9jcnQucm9vdGcyLmFtYXpvbnRydXN0LmNvbS9yb290ZzIuY2Vy
MD0GA1UdHwQ2MDQwMqAwoC6GLGh0dHA6Ly9jcmwucm9vdGcyLmFtYXpvbnRydXN0
LmNvbS9yb290ZzIuY3JsMBEGA1UdIAQKMAgwBgYEVR0gADANBgkqhkiG9w0BAQsF
AAOCAQEAYjdCXLwQtT6LLOkMm2xF4gcAevnFWAu5CIw+7bMlPLVvUOTNNWqnkzSW
MiGpSESrnO09tKpzbeR/FoCJbM8oAxiDR3mjEH4wW6w7sGDgd9QIpuEdfF7Au/ma
eyKdpwAJfqxGF4PcnCZXmTA5YpaP7dreqsXMGz7KQ2hsVxa81Q4gLv7/wmpdLqBK
bRRYh5TmOTFffHPLkIhqhBGWJ6bt2YFGpn6jcgAKUj6DiAdjd4lpFw85hdKrCEVN
0FE6/V1dN2RMfjCyVSRCnTawXZwXgWHxyvkQAiSr6w10kY17RSlQOYiypok1JR4U
akcjMS9cmvqtmg5iUaQqqcT5NJ0hGA==
-----END CERTIFICATE-----
...
```

Figure 6.10: Command and output of an OpenSSL connection to fetch the certificate chain.

```
$ openssl x509 -pubkey -inform pem -in ca.pem -noout > ca.pubkey.pem
```

Figure 6.11: Command used to extract the public key from an PEM encoded certificate and save it as PEM encoded data to a local file called ca.pubkey.pem.

```
$ openssl pkey -in ca.pubkey.pem -pubin -inform pem -outform der \
    | openssl dgst -sha256 -binary \
    | openssl enc -base64
> ++MBgDH5WGvL9Bcn5Be30cRcL0f5O+NyoXuWtQdX1aI=
```

Figure 6.12: Example command used to create a Base64-encoded SHA-256 digest of the X.509 certificate's DER-encoded ASN.1 Subject Public Key Info [48].

6.3.2 Hardening Input Data Formats

This validation test setup requires two iOS apps with one app being the victim app offering a custom URL scheme to receive data, and the other one being an attacking app trying to exploit the provided functionality. Both apps can be built in Xcode and run within the iOS simulator.

The input data format is different for every functionality and app, therefore the following test case is unique on its own, but implements the generic principles illustrated in Section 5.3.2.

The victim app represents a file sharing app, which allows the app user to upload files to a remotely hosted data drive. To allow other apps to share documents via the file sharing service, the victim app offers a custom URL scheme `upload`. To provide the user a faster experience, the victim app does not ask the app user to confirm the file upload, but instead directly uploads the file and notifies the user afterwards.

The functionality is provided by a custom path with the associated data in the query parameters `upload:///add?data=<Base64 encoded String>`.

A minimal example implementation for this app is provided in Figure 6.13. Furthermore, the app needs to register itself with the operating system by providing the `CFBundleURLTypes` properties in its `Info.plist` configuration file. The example shown in Figure 5.6 can be adapted by replacing `social` with `upload`.

```
1  class SceneDelegate: UIResponder, UIWindowSceneDelegate {
2
3      func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {
4          for context in URLContexts {
5              let url = context.url
6              let queryItems = URLComponents(string: url.absoluteString)?.queryItems ?? []
7              guard url.scheme == "upload" else {
8                  return // ignore invalid scheme
9              }
10             if url.path == "/add" {
11                 guard let base64EncodedData = queryItems.first(where: {
12                             $0.name == "data"
13                         })?.value,
14                         let data = Data(base64Encoded: base64EncodedData) else {
15                     return // invalid Base64 String
16                 }
17                 upload(data: data)
18             }
19         }
20     }
21
22     func upload(data: Data) {
23         // transfer the data to the remote location, e.g., using HTTP PUT
24         //  not relevant for this validation test
25         print("Finished uploading \(data.count) bytes of data")
26     }
27 }
```

Figure 6.13: Implementation of the victim app providing a custom URL scheme.

The attacker app is the app which sends the injection payload via the custom URL scheme into the victim app. As the custom URL scheme is registered at iOS, the attacker

app can call the application method `UIApplication.open(url:)` with the target URL. The file sharing app might expect other apps to send reasonable payloads like the one presented in Figure 6.14 that is then received in the victim app indicated by the log line in Figure 6.15, but an attacker can also abuse the functionality and spam the victim app with large payloads, filling up the remote storage size with invalid data causing a malfunction and therefore, breaking its *Availability* (see Section 2.1.3).

```
// Create a simple message
let payload = "This is a good note"
let payloadData = payload.data(using: .utf8)!
// Open the URL
let base64DataString = payloadData.base64EncodedString()
UIApplication.shared.open(URL(string: "upload:///add?data=" + base64DataString)!)
```

Figure 6.14: Example of intended usage of the custom URL scheme.

```
Finished uploading 19 bytes of data
```

Figure 6.15: Log of the victim app indicating that the good URL passed all existing checks.

One implementation of an attacker app is presented in Figure 6.16, where a 100 megabyte random payload is generated and attached to the URL string. When running the example code, the generated data is passed to the victim app, indicated by the log message shown in Figure 6.17.

```
// Create a simple message
let payload = "This is some evil text, with a long random string here: "
let payloadData = payload.data(using: .utf8)!
// Create a large piece of random data
let randomData = Data((0..<(100 * 1000 * 1000)).map { _ in
    UInt8.random(in: UInt8.min...UInt8.max)
})
let data = payloadData + randomData
// Open the URL
let base64DataString = data.base64EncodedString()
UIApplication.shared.open(URL(string: "upload:///add?data=" + base64DataString)!)
```

Figure 6.16: Example of abusing the custom URL scheme to send huge data files.

```
Finished uploading 104857656 bytes of data
```

Figure 6.17: Log of the victim app indicating that the bad URL passed all existing checks.

Adding input validation to the URL parser

For this specific example the simplest input validation, to protect the victim app against huge payloads, is adding a size limit by adapting the method `upload(data: Data)`

starting at line 22 of figure Figure 6.13 to the replacement shown in Figure 6.18, which is dropping every payload that is too large.

```
func upload(data: Data) {
    // Apply 100 kilobyte size limit validation
    guard data.count < 100 * 1000 else {
        print("Payload with \(data.count) bytes too large to upload")
        return
    }
    // transfer the data to the remote location, e.g. using HTTP PUT, not relevant for this
    ↪ validation test
    print("Finished uploading \(data.count) bytes of data")
}
```

Figure 6.18: Example validation applying a 100 kilobyte data size limit.

After applying the changes, the log of the victim app indicates that the received data is too large to be uploaded, by printing the log shown in Figure 6.19.

```
Payload with 104857619 bytes too large to upload
```

Figure 6.19: Log of the victim app indicating thatthe bad URL failed to pass the validation checks.

This proves the effectiveness of guideline G.4 applying validation of the input data.

6.3.3 Hardening Against Backup Exploits

As described in Section 5.3.3, storing sensitive information in the file system does not provide enough protection against extracting the information from a backup. To validate the guidelines in that section, the example app in Figure 6.20 stores user credentials and loads it from the storage. To showcase the insecure approach, it is stored in the local documents directory, which is a specialized directory in the iOS sandbox, designated for long-term data storage.

```
let credentials = """
username = my-username
password = no-one-else-should-read-this
"""
// Request the URL for the designated documents directory
let documentsDirectoryURL = FileManager.default
    .urls(for: .documentDirectory, in: .userDomainMask)[0]
// Create a credentials file URL
let credentialsFileURL = documentsDirectoryURL
    .appendingPathComponent("credentials")
    .appendingPathExtension("txt")
// Write the credentials to the file system
try! credentials.write(to: credentialsFileURL, atomically: true, encoding: .utf8)
// Restore the credentials
let restoredCredentials = try! String(contentsOf: credentialsFileURL)
print(restoredCredentials == credentials) // --> prints 'true'
```

Figure 6.20: Example code for insecure saving and loading of user credentials.

App documents are not accessible on iOS by default, and therefore, the data needs to be transferred to a macOS system. The backup extraction method shown in Section 5.3.3 is applied to extract the credentials file and read its data.

When looking into the file system of the app sandbox, the documents folder can be found, including the `credentials` written in Figure 6.20. By opening it with any text editor application, the credentials are clearly visible and accessible in Figure 6.21.

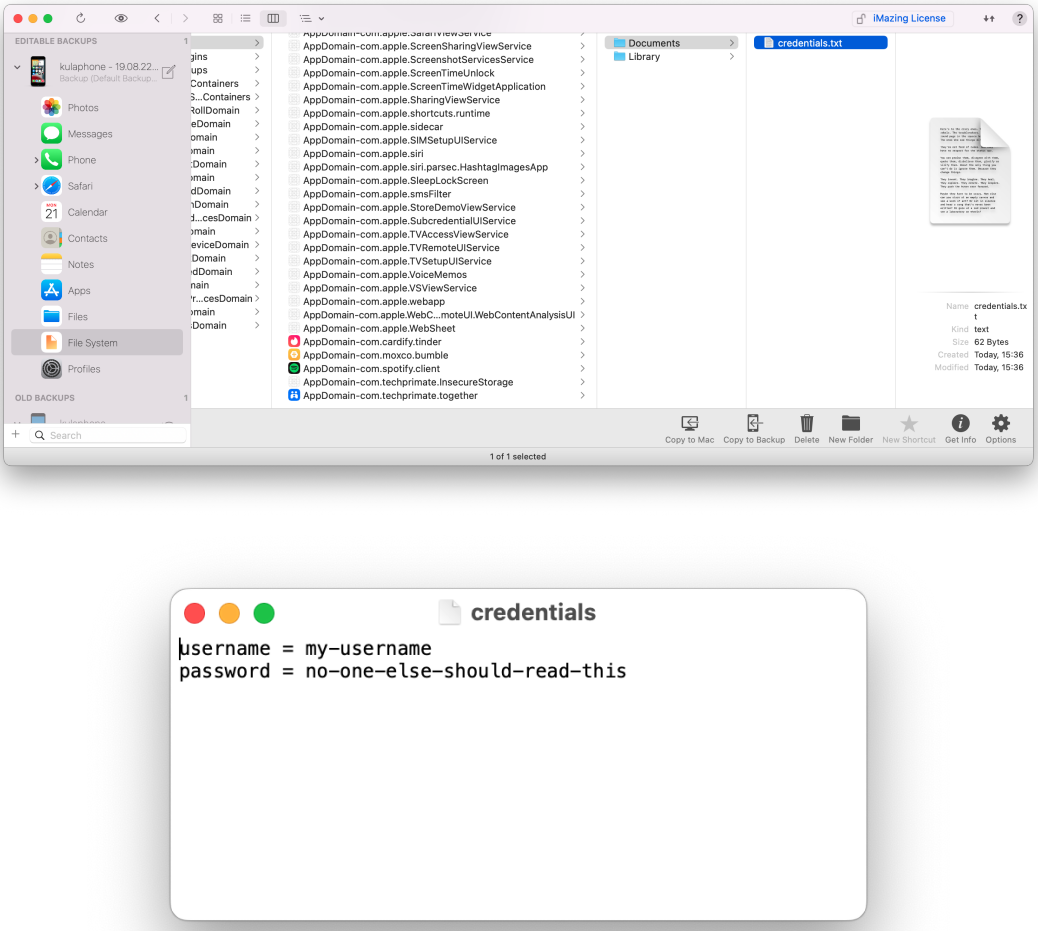

Figure 6.21: Clear credentials shown in iMazing file explorer.

Using Keychain for secure data storage

To apply the security guideline, the insecure storage is exchanged with a different approach using Keychain in Figure 6.22. The Keychain uses a query based system, where the

developer needs to define read ("copy"), add, update, and delete queries. The fields defined in the queries are then saved with the credentials as attributes and can be used to identify them again later, i.e., `kSecLabel` can be used to label the item.

```
// Label used to identify the Keychain item
let label = "my-item"

// Credentials to save
let username = "my-username"
let password = "no-one-else-should-read-this"
let passwordData = password.data(using: .utf8)!

// Create the insertion query with the Keychain item configuration
let insertQuery: [String: Any] = [
    kSecClass as String: kSecClassGenericPassword,
    kSecAttrLabel as String: label,
    kSecAttrAccount as String: username,
    kSecValueData as String: passwordData
]
var status = SecItemAdd(insertQuery as CFDictionary, nil)
guard status == errSecSuccess else {
    fatalError("Failed to insert item, error: \(status)")
}
// Create the read query to find one item for the given username
let readQuery: [String: Any] = [
    kSecClass as String: kSecClassGenericPassword,
    kSecAttrLabel as String: label,
    kSecMatchLimit as String: kSecMatchLimitOne,
    kSecReturnData as String: true,
    kSecReturnAttributes as String: true,
]
var item: CFTypeRef?
status = SecItemCopyMatching(readQuery as CFDictionary, &item)
guard status != errSecItemNotFound else {
    fatalError("Item not found")
}
guard status == errSecSuccess else {
    fatalError("Failed to copy item, error: \(status)")
}
// Cast the returned data into usable objects
guard let attributes = item as? [String: Any],
        let storedUsername = attributes[kSecAttrAccount as String] as? String,
        let storedPasswordData = attributes[kSecValueData as String] as? Data,
        let storedPassword = String(data: storedPasswordData, encoding: .utf8) else {
    fatalError("Failed to decode the Keychain item")
}
// Validate
print(storedUsername == username) // --> prints 'true'
print(storedPassword == password) // --> prints 'true'
```

Figure 6.22: Example code for secure saving and loading of user credentials.

After applying the changes, the credentials can still be saved and loaded, but they are not stored in the app documents directory anymore and therefore, not added as plain text files in the backup. As stated by the Apple Platform Security Guide [42] the "[...] user's keychain database is backed up (too) [...] but remains protected by a UID-tangled key. This allows the keychain to be restored only to the same device from which it originated, and [...] no one else [...] can read the user's keychain items [...]", which protects the credentials from being extractable from the backup.

This proves using the Keychain service as a data storage contributes to hardening against backup exploits, as stated in guideline G.5 and G.6.

6.3.4 Applying Code Obfuscation

To validate the claims and techniques previously introduced in Section 5.3.4 and Section 5.3.5, the first step is setting up a suitable environment. The following validation test requires an example application, in this case, it is a simple app where a user can get secret information if they enter the correct password. The app project can be created using the default project in Xcode under *iOS* > *App* > *Storyboard*, and by adapting the `ViewController.swift` to the code shown in Figure 6.23. Its function is rather simple because a user only needs to enter the words `this is secret` and the alert will display the secret message `SUPERSECRET`.

```
class ViewController: UIViewController {

    private let textField = UITextField()

    override func viewDidLoad() {
        super.viewDidLoad()

        textField.placeholder = "Enter the secret"
        textField.autocapitalizationType = .none

        let button = UIButton(configuration: .tinted())
        button.setTitle("Check", for: .normal)
        button.addTarget(self, action: #selector(checkPassword), for: .touchUpInside)

        let stackView = UIStackView(arrangedSubviews: [
            textField,
            button
        ])
        stackView.axis = .vertical
        stackView.spacing = 8
        view.addSubview(stackView)

        stackView.translatesAutoresizingMaskIntoConstraints = false
        NSLayoutConstraint.activate([
            stackView.leadingAnchor.constraint(equalTo: view.layoutMarginsGuide.leadingAnchor),
            stackView.trailingAnchor.constraint(equalTo: view.layoutMarginsGuide.trailingAnchor),
            stackView.centerYAnchor.constraint(equalTo: view.layoutMarginsGuide.centerYAnchor),
        ])
    }

    @objc func checkPassword() {
        let alert: UIAlertController
        if isPasswordValid(textField.text) {
            alert = UIAlertController(title: "You did it", message: "This is the secret:
    ↪ SUPERSECRET", preferredStyle: .alert)
        } else {
            alert = UIAlertController(title: "Wrong Password!", message: nil, preferredStyle:
    ↪ .alert)
        }
        alert.addAction(UIAlertAction(title: "OK", style: .default))
        present(alert, animated: true)
    }

    func isPasswordValid(_ password: String?) -> Bool {
        password == "this is secret"
    }
}
```

Figure 6.23: App showing a secret message alert if the user enters the correct password 'this is secret'.

To decompile the binary after compilation, a decompiler such as *Hopper* [70] or *Ghidra*

[108] can be used, with the latter being used in the following steps. Furthermore, to analyze the binary strings the command utility `rabin2` of the *radare2* toolkit [98] is used. To simplify the validation process, and due to lacking access to a jailbroken iOS device, this validation test uses the app binary compiled for the iOS simulator, without removing the FairPlay DRM applied by the App Store distribution process.

Finding the password

In an initial step, the attacker analyzes the state of the software. By running the app and entering any invalid password, the user is presented with an alert titled *"Wrong Password!"*. This is the first indicator which can be looked up in the app binary.

Using `rabin2 -zz <path/to/app/binary> | grep -i password` returns the strings contained in binary with their binary memory address, including the alert title as shown in Figure 6.24.

```
...
77  0x00005882 0x100005882 13  14   3.__TEXT.__objc_methname  ascii   checkPassword
193 0x000067c1 0x1000067c1 15  16   4.__TEXT.__cstring        ascii   Wrong Password!
429 0x000112a0 0x1000112a0 17  18                            ascii   3checkPasswordyyF
430 0x000112b4 0x1000112b4 24  25                            ascii   5isPasswordValidySbSSSgF
...
```

Figure 6.24: Output of the rabin2 command displaying the addresses of literal strings and dynamic method names containing the word 'password'.

After opening the binary in Ghidra and performing the binary analysis according to their documentation [108], the function *Navigation > Go To...* is used to jump to the address `0x000067c1` found in the rabin2 command output. In Ghidra, it is possible to navigate to other references to the address by double-clicking on the `XREF` in the code viewer, as shown in Figure 6.25, leading straight to the compiled version of the method `checkPassword`.

```
                    s_Wrong_Password!_1000067c1                    XREF[1]:     _$s16Obfuscation...
1000067c1 57 72 6f       ds         "Wrong Password!"
          6e 67 20
          50 61 73 ...
```

Figure 6.25: Detected string in Ghidra after analysis a binary, showing the relevant references to the address.

After some manual cleanup the decompiled code looks like Figure 6.26. Without further steps it is possible to see that the password, is most likely validated by calling a function in line 2, which would require more work to decompile and understand. Luckily, string obfuscation has not been applied and the secret message can directly be found in line 15.

This proves guideline G.7 stating that apps can be decompiled.

```
...
isPasswordNotEqual = (**(code **)((*unaff_x20 & *(ulong *)__got::_swift_isaMask) +
    ↪ 0x60))(local_60,local_58);
__stubs::_swift_bridgeObjectRelease(local_58);
if ((isPasswordNotEqual & 1) == 0) {
    _$sSo17UIAlertControllerCMa();
    optionalText = __stubs::_$sSS21_builtinStringLiteral17utf8CodeUnitCount7isASCIISSBp_BwBi1_tcfC
        ("Wrong Password!",0xf,1);
    alertController =
     ↪ _$sSo17UIAlertControllerC5title7message14preferredStyleABSSSg_AFSo0abF0VtcfCTO
        (SUB168(optionalText,0),SUB168(optionalText >> 0x40,0),0,0,1);
} else {
    _$sSo17UIAlertControllerCMa();
    optionalText = __stubs::_$sSS21_builtinStringLiteral17utf8CodeUnitCount7isASCIISSBp_BwBi1_tcfC
        ("You did it",10,1);
    message = __stubs::_$sSS21_builtinStringLiteral17utf8CodeUnitCount7isASCIISSBp_BwBi1_tcfC
        ("This is the secret: SUPERSECRET",0x1f,1);
    alertController =
     ↪ _$sSo17UIAlertControllerC5title7message14preferredStyleABSSSg_AFSo0abF0VtcfCTO
        (SUB168(optionalText,0), SUB168(optionalText >> 0x40,0), SUB168(message,0), SUB168(message
     ↪ >> 0x40,0), 1);
}
...
```

Figure 6.26: Decompiled code of the method 'checkPassword'.

Hardening against reverse engineering using string obfuscation

Obfuscation can be achieved in many ways. For this example, the literal strings are replaced with UTF-8 byte sequences, which are then decoded into strings at runtime by calling a new method `deobfuscate`. For the setup of the validation test, the example code in Figure 6.23 is adapted by replacing all relevant literal strings as shown in Figure 6.27.

After re-compilation, the analysis of the binary using `rabin2` does not include the strings used in the UI anymore (see Figure 6.28). Due to UIKit using the Objective-C runtime, which uses dynamic method selectors and calls methods using their literal name strings, it is still possible to find the relevant code sections in Ghidra, because of the address of `checkPassword` in the rabin2 output.

By also renaming the method names `checkPassword` and `isPasswordValid` into something unreadable, their names are also not included in the string analysis anymore (see Figure 6.29).

Due to the absence of literal strings in the output of the command, the reverse engineering process was made more difficult. This proves that code obfuscation can harden against reverse engineering, as stated in guideline G.8.

```
@objc func checkPassword() {
    let alert: UIAlertController
    if isPasswordValid(textField.text) {
        alert = UIAlertController(title: deobfuscate([0x59, 0x6f, 0x75, 0x20, 0x64,
                                                      0x69, 0x64, 0x20, 0x69, 0x74]),
                                  message: deobfuscate([0x54, 0x68, 0x69, 0x73, 0x20,
                                                        0x69, 0x73, 0x20, 0x74, 0x68,
                                                        0x65, 0x20, 0x73, 0x65, 0x63,
                                                        0x72, 0x65, 0x74, 0x3a, 0x20,
                                                        0x53, 0x55, 0x50, 0x45, 0x52,
                                                        0x53, 0x45, 0x43, 0x52, 0x45,
                                                        0x54]),
                                  preferredStyle: .alert)
    } else {
        alert = UIAlertController(title: deobfuscate([0x57, 0x72, 0x6f, 0x6e, 0x67,
                                                      0x20, 0x50, 0x61, 0x73, 0x73,
                                                      0x77, 0x6f, 0x72, 0x64, 0x21]),
                                  message: nil, preferredStyle: .alert)
    }
    alert.addAction(UIAlertAction(title: deobfuscate([0x4f, 0x4b]), style: .default))
    present(alert, animated: true)
}

func isPasswordValid(_ password: String?) -> Bool {
    password == deobfuscate([0x74, 0x68, 0x69, 0x73, 0x20, 0x69, 0x73,
                             0x20, 0x73, 0x65, 0x63, 0x72, 0x65, 0x74])
}

func deobfuscate(_ bytes: [UInt8]) -> String {
    String(bytes: bytes, encoding: .utf8) ?? ""
}
```

Figure 6.27: App showing a secret message alert if the user enters the correct password 'this is secret'.

```
...
77   0x00005842 0x100005842 13  14   3.__TEXT.__objc_methname  ascii   checkPassword
432  0x000113d2 0x1000113d2 17  18                            ascii   3checkPasswordyyF
433  0x000113e6 0x1000113e6 24  25                            ascii   5isPasswordValidySbSSSgF
...
```

Figure 6.28: Output of the rabin2 command after applying String obfuscation.

```
--> no output
```

Figure 6.29: Output of the rabin2 command after applying method obfuscation.

6.3.5 Applying Anti-Reversing And Anti-Tampering Techniques

For the validation of the effectiveness of guideline G.9 an example application is required. Depending on the implemented checks the setup can be extended, e.g., by setting up a jailbroken iOS device. In this test the app should check if debugging is enabled on the app start and terminate if it is detected. The implementation can be found in the open source package IOSSecuritySuite [109], which is also used in this test.

The app project can be created using the default project in Xcode under *iOS* > *App* > *Storyboard* and adapting the `ViewController.swift` to the code shown in Figure 6.30.

```
1  import UIKit
2  import IOSSecuritySuite
3
4  class ViewController: UIViewController {
5
6      override func viewDidAppear(_ animated: Bool) {
7          super.viewDidAppear(animated)
8
9          let alert = UIAlertController(
10             title: "Hello World",
11             message: "Debugging is " + (IOSSecuritySuite.amIDebugged() ? "enabled" : "disabled"),
12             preferredStyle: .alert)
13         alert.addAction(UIAlertAction(title: "OK", style: .default, handler: { _ in
14             if IOSSecuritySuite.amIDebugged() {
15                 // Exit the application if debugger is enabled
16                 exit(1)
17             }
18         }))
19         present(alert, animated: true)
20     }
21 }
```

Figure 6.30: App displaying an alert and terminating if a debugger is attached.

When running from Xcode with the debugger enabled, the app displays the message "Debugging is enabled" and exits the process after tapping on "OK". If the app is run directly by tapping on the app icon on the iOS home screen, the app starts and displays the message "Debugging is disabled", without exiting the process.

This proves guideline G.9 that anti-reversing techniques can be used to block users from an app with additional functionality.

Circumventing startup checks using binary tampering

To circumvent the check introduced in the previous section, an attacker can decompile the binary using the same techniques described in Section 6.3.4. This test extends the previous one, by decompiling the compiled binary, finding the call of the debugging check and flipping the conditional statement in line 14 of Figure 6.30.

After opening and analyzing the binary in Ghidra, the callback handler function of the `UIAlertController` can be found in the binary's assembly code (see Figure 6.31) with its decompiled code in Figure 6.32 matching the original source code in lin 14 to 17Figure 6.30 quite closely.

```
1  100001e3c ff c3 00 d1    sub    sp,sp,#0x30
2  100001e40 f4 4f 01 a9    stp    x20,x19,[sp, #local_20]
3  100001e44 fd 7b 02 a9    stp    x29,x30,[sp, #local_10]
4  100001e48 fd 83 00 91    add    x29,sp,#0x20
5  100001e4c ff 07 00 f9    str    xzr,[sp, #local_28]
6  100001e50 e0 07 00 f9    str    x0,[sp, #local_28]
7  100001e54 00 00 80 d2    mov    x0,#0x0
8  100001e58 4a 53 00 94    bl     _$s16IOSSecuritySuiteAACMa
9  100001e5c f4 03 00 aa    mov    x20,x0
10 100001e60 f7 51 00 94    bl     _$s16IOSSecuritySuiteAAC11amIDebuggedSbyFZ
11 100001e64 80 00 00 36    tbz    w0,#0x0,LAB_100001e74
12 100001e68 01 00 00 14    b      LAB_100001e6c
13
14    LAB_100001e6c
15 100001e6c 20 00 80 52    mov     w0,#0x1
16 100001e70 ca c7 00 94    bl      __stubs::_exit                          void _exit(int param_1)
17
18    LAB_100001e74
19 100001e74 fd 7b 42 a9    ldp    x29=>local_10,x30,[sp, #0x20]
20 100001e78 f4 4f 41 a9    ldp    x20,x19,[sp, #local_20]
21 100001e7c ff c3 00 91    add    sp,sp,#0x30
22 100001e80 c0 03 5f d6    ret
```

Figure 6.31: Assembly instruction code of the debug alert handler

```
1  void _$s19AntiReversingChecks14ViewController
       ↪ C13viewDidAppearyySbFySo13UIAlertActionCcfU_(void)
2  {
3      ulong uVar1;
4
5      _$s16IOSSecuritySuiteAACMa(0);
6      uVar1 = _$s16IOSSecuritySuiteAAC11amIDebuggedSbyFZ();
7      if ((uVar1 & 1) != 0) {
8          __stubs::_exit(1);
9      }
10     return;
11 }
```

Figure 6.32: Decompiled code of the debug alert handler

The assembly help annotations of Ghidra already shows the conditional check using the `tbz` and `b` instructions in line 11 and 12 of Figure 6.31. Basically, the instruction `tbz` checks if the bit is zero and jumps to the given label `LAB_100001e74`, inverting the condition in line 7 of Figure 6.32. Otherwise it continues with instruction `b` jumping to `LAB_100001e6c` and calling the method `__stubs::__exit` which terminates the process. More details on the exact functionality of these two instructions can be found in the ARM reference documentation [66].

To circumvent the termination an attacker can modify the instruction in line 11of Figure 6.31 from a `tbz` to a `tbnz` to invert the check and jump if the bit is non-zero. The patch can be applied by opening up the binary, located in the derived data folder (see Figure 6.34), in an hex editor, i.e., Visual Studio Code, and edit the eight byte in the line `00001E60` from `36` to `37`, as shown in Figure 6.33.

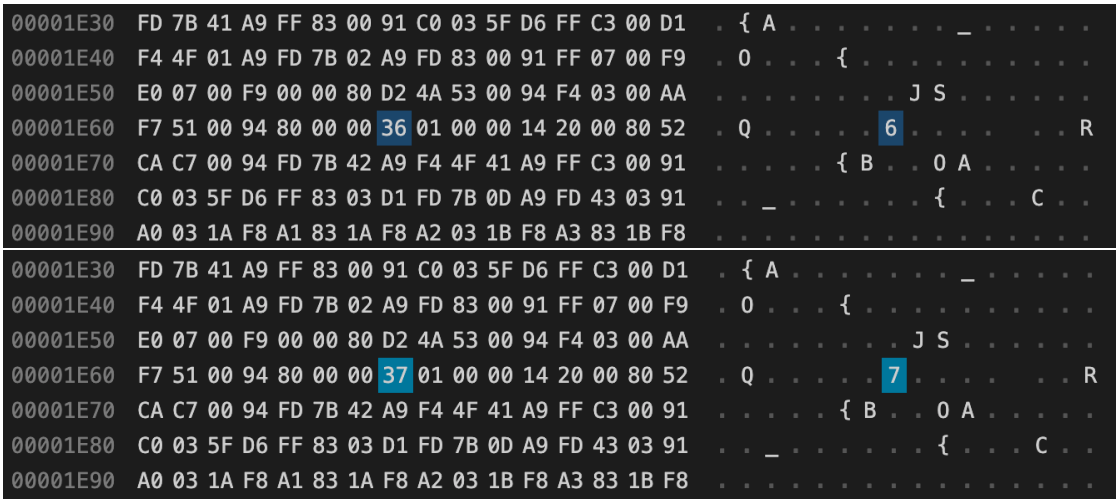

Figure 6.33: Patching the 'tbz' instruction to 'tbnz'.

```
1 /Users/<macOS_username>/Library/Developer/Xcode/DerivedData/
    ↪ AntiReversingChecks-egeosiudsrjoxhcnlkhbytxfyhaz/
    ↪ Build/Products/Debug-iphonesimulator/AntiReversingChecks.app/
    ↪ AntiReversingChecks
```

Figure 6.34: Path to binary in the derived data of Xcode of the example app 'AntiReversingChecks'.

After patching the file and re-running the binary in Xcode without building (the shortcut is `CMD+CTRL+R`), to run the patched binary, the debugger is attaching to the app. The alert appears once again with the message "Debugging is enabled" but the application does not exit anymore, disarming the startup check.

When dealing with additional protection mechanism, such as FairPlay DRM or code signing, further decompilation and patching steps are required. Still, this validation test proves that app binaries can be modified and, therefore, confirms guideline G.10.

6.4 Validation Results

The validation results are summarized in the following overview table. For each validation test, the original state has been identified and by applying the mitigation or hardening techniques discussed by the guidelines in their respective sections, a result state has been reached.

Validation Test	Guidelines	Original State	Applied Mitigation/Hardening/Exploit Technique	Result State
6.3.1 Testing Secure Network Communication Techniques	G.1 Upgrade Security G.2 Attacker vs Target G.3 Certificate Pinning	Network connection between the iOS app and a remote endpoint is vulnerable to passive network sniffing and active Man-in-the-Middle (MITM) attacks	Applying encryption and certificate pinning	Traffic is encrypted and secure network connection cannot be hijacked in an MITM attack anymore
6.3.2 Hardening Input Data Formats	G.4 Validate Input Data	Any size of data was injectable into the victim app	Applying validation of the payload data	Injectable data is limited
6.3.3 Hardening Against Backup Exploits	G.5 Use Secure Storage G.6 Security Of UserDefaults	Sensitive data stored in insecure data storage can be extracted from iOS backups	Use Keychain for sensitive data	Sensitive data is not stored in the app file system, and therefore, not extractable from backups anymore
6.3.4 Applying Code Obfuscation	G.7 Expect Decompilation G.8 Obfuscate Strings	App binary contained sensitive information as literal strings	Strings and method names are obfuscated	App binary does not contains sensitive information in strings output, and is therefore harder to find the relevant checks
6.3.5 Applying Anti-Reversing And Anti-Tampering Techniques	G.9 Use Anti-Reversing G.10 Anti-Tampering Techniques Can Be Ineffective	App applies anti-debugging technique and exits as soon as a debugger is detected	Modified assembly instructions by tampering with the executable binary	Anti-debugging check is disabled and app does not exit anymore when a debugger is detected

CHAPTER 7

Conclusion

After establishing a foundation on security for computer and information systems, this thesis narrowed down the observed domain to the mobile operating systems iOS, by researching the system architecture and application ecosystem, in combination with its functionalities and limitations, and discussing the technical environment of app execution and communication with system services. By analyzing existing guidelines, such as the OWASP Mobile Top 10 [95] and the Apple Platform Security Guide [42], the process of identifying security crunchpoints of the ecosystem and developing security guidelines to harden against or mitigate vulnerabilities has been established.

The developed guidelines include techniques to secure network communication, such as certificate pinning (see Section 5.3.1), validating user-injected data (see Section 5.3.2), using the secure data storage Keychain (see Section 5.3.3), as well as techniques to circumvent anti-reverse-engineering and anti-binary-tampering techniques (see Section 5.3.4 and Section 5.3.5). To prove their effectiveness, a practical reproducible validation test has been conducted for each one, by applying techniques such as Man-in-the-Middle attacks or binary-reverse-engineering.

The results of this thesis can now also be transferred to a wider scope. The other major mobile operating system next to iOS is Android [103], and by performing further research the security guidelines created in this thesis might also be applicable to mobile apps developed for Android.

As discussed in Section 3.2.2, iOS apps can also run natively on macOS with ARM-based chipsets. This opens up a completely different area of further research possibilities, as the system environment established on macOS is not identical to iOS, and therefore, techniques such as sandboxing (see Section 3.2.1) can be looser, thus opening up the possibility of security vulnerabilities.

Additional work can also be done by researching the individual guidelines presented in this thesis more in-depth and using them as a foundation for other security guidelines

in the specific topics. Furthermore, the process of identifying security crunchpoints in iOS proposed in this thesis, can be used to identify other threats in the operating system. By applying the same process of identifying threats, proposing guidelines to harden or mitigate, and validating them using tests, the overall security awareness can be intensified.

As Apple releases new major updates of their operating system iOS in a yearly interval, defining new security guidelines based on updated and newly introduced features might become a regular task. Therefore, further research can also be conducted on automating the process of identifying vulnerabilities and proposing new guidelines.

List of Figures

List of Tables

Bibliography

[1] Adascalitei, I. Mobile Devices Risks and Recommendation. *Informatica Economica*, 24(3):54–63, 2020. doi:10.24818/issn14531305/24.3.2020.05.

[2] Ahmad, M. S. et al. Comparison between Android and iOS Operating System in Terms of Security. In *2013 8th International Conference on Information Technology in Asia (CITA)*, pages 1–4, 2013. doi:10.1109/CITA.2013.6637558.

[3] Ai, N. and Lu, Y. and Deogun, J. The Smart Phones of Tomorrow. *SIGBED Rev.*, 5(1), 2008. doi:10.1145/1366283.1366299.

[4] Andrews, J. Mathematical Applications of the Dynamic Storage Analog Computer. In *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '60 (Western), pages 119–131, New York, NY, USA, 1960. Association for Computing Machinery. doi:10.1145/1460361.1460378.

[5] Blochberger, M. et al. State of the Sandbox: Investigating MacOS Application Security. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, WPES'19, pages 150–161, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3338498.3358654.

[6] Breaux, T. and Moritz, J. The 2021 software developer shortage is coming. *Communications of the ACM*, 64(7):39–41, 2021. doi:10.1145/3440753.

[7] Bucicoiu, M. et al. XiOS: Extended Application Sandboxing on iOS. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 43–54, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2714576.2714629.

[8] Byambasuren, O. Current Knowledge and Adoption of Mobile Health Apps Among Australian General Practitioners: Survey Study. *JMIR mHealth and uHealth.*, 7, 2019. doi:10.2196/13199.

[9] Cavallaro, L. et al. *Detection of Intrusions and Malware, and Vulnerability Assessment: 19th International Conference, DIMVA 2022, Cagliari, Italy, June 29 –July 1, 2022, Proceedings*, volume 13358 of *Lecture Notes in Computer Science*. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-031-09484-2.

[10] Deng, Z. et al. iRiS: Vetting Private API Abuse in iOS Applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56, Denver Colorado USA, 2015. ACM. doi:10.1145/2810103.2813675.

[11] D'Orazio, C. J. and Choo, K. A Technique to Circumvent SSL/TLS Validations on iOS Devices. *Future Generation Computer Systems*, 74:366–374, 2017. doi:10.1016/j.future.2016.08.019.

[12] D'Orazio, C. J. et al. A Markov Adversary Model to Detect Vulnerable iOS Devices and Vulnerabilities in iOS apps. *Applied Mathematics and Computation*, 293:523–544, 2017. doi:10.1016/j.amc.2016.08.051.

[13] Dye, S. and Scarfone, K. A standard for developing secure mobile applications. *Computer Standards & Interfaces*, 36(3):524–530, 2014. doi:10.1016/j.csi.2013.09.005.

[14] ENISA. *Smartphone secure development guidelines*. European Union Agency for Network and Information Security., 2016. doi:10.2824/071102.

[15] Fahl, S. et al. Rethinking SSL Development in an Appified World. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 49–60. Association for Computing Machinery, 2013. doi:10.1145/2508859.2516655.

[16] Feichtner, J. and Missmann, D. and Spreitzer, R. Automated Binary Analysis on IOS: A Case Study on Cryptographic Misuse in IOS Applications. In *Proceedings of the 11th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '18, pages 236–247, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3212480.3212487.

[17] Fielding, R. T. and Nottingham, M. and Reschke, J. HTTP Semantics. Request for Comments RFC 9110, Internet Engineering Task Force, 2022. doi:10.17487/RFC9110.

[18] Garcia, L. and Rodriguez, R. J. A Peek under the Hood of iOS Malware. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 590–598, 2016. doi:10.1109/ARES.2016.15.

[19] Gasiba, T. E. and Lechner, U. Raising Secure Coding Awareness for Software Developers in the Industry. 2021. doi:10.48550/arXiv.2102.10431.

[20] Guttman, B. and Roback, E. An Introduction to Computer Security: the NIST Handbook, 1995. doi:10.6028/NIST.SP.800-12r1.

[21] Halvorsen, O. and Clarke, D. Mac OS X and iOS. In *OS X and iOS Kernel Programming*, pages 15–38. Apress, Berkeley, CA, 2011. doi:10.1007/978-1-4302-3537-8_2.

[22] Hayran, A. et al. Security Evaluation of IOS and Android. *International Journal of Applied Mathematics, Electronics and Computers*, pages 258–258, 2016. doi:10.18100/ijamec.270378.

[23] Heid K., Heider, J. and Qasempour, K. Raising Security Awareness on Mobile Systems through Gamification. In *Proceedings of the European Interdisciplinary Cybersecurity Conference*, EICC 2020. Association for Computing Machinery, 2020. doi:10.1145/3424954.3424958.

[24] Mayer, D A. Idb: A Tool for Blackbox iOS Security Assessments. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 281–282. Association for Computing Machinery, 2016. doi:10.1145/2897073.2897710.

[25] Pradhan, S. et al. Smartphone-Based Acoustic Indoor Space Mapping. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(2), 2018. doi:10.1145/3214278.

[26] Ross, R. and McEvilley, M. and Oren, J. Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems. Technical report, National Institute of Standards and Technology, 2018. doi:10.6028/NIST.SP.800-160v1.

[27] Saltzer, J.H. and Schroeder, M.D. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. doi:10.1109/PROC.1975.9939.

[28] Shirey, R. Internet Security Glossary, Version 2. Request for Comments RFC 4949, Internet Engineering Task Force, 2007. doi:10.17487/RFC4949.

[29] Sion, L. et al. Security Threat Modeling: Are Data Flow Diagrams Enough? In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 254–257. Association for Computing Machinery, 2020. doi:10.1145/3387940.3392221.

[30] Spinellis, D. Half-Century of Unix: History, Preservation, and Lessons Learned. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 1. IEEE Press, 2017. doi:10.1109/MSR.2017.1.

[31] Stallings, W. *Operating Systems : Internals and Design Principles*. Pearson Education Limited, ninth edition, 2018. ISBN: 0134670957.

[32] Su, M. K. Techniques for Identifying Mobile Platform Vulnerabilities and Detecting Policy-Violating Applications. page 120, 2016. Last accessed on October 8th 2022. URL: https://ink.library.smu.edu.sg/etd_coll_all/3.

[33] Suarez, D. and Mayer, D. Faux disk encryption: Realities of secure storage on mobile devices. In *Proceedings of the International Conference on Mobile Software*

Engineering and Systems, MOBILESoft '16, pages 283–284, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2897073.2897711.

[34] Wang, P. et al. Field Experience with Obfuscating Million-User iOS Apps in Large Enterprise Mobile Development. *Software: Practice and Experience*, 49(2):252–273, 2019. doi:10.1002/spe.2648.

[35] Xing, L. et al. Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS~X and iOS. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 31–43, Denver Colorado USA, 2015. ACM. doi:10.1145/2810103.2813609.

[36] Yixiang, Z. and Kang, Z. Review of iOS Malware Analysis. In *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, pages 511–515, 2017. doi:10.1109/DSC.2017.104.

[37] Zheng, G. and Lyu, Y. and Wang, D. True Random Number Generator Based on Ring Oscillator PUFs. In *Proceedings of the 2017 2nd International Conference on Multimedia Systems and Signal Processing*, ICMSSP 2017, pages 1–5, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3145511.3145518.

Other References

[38] Apple, Inc. About Developing for Mac | Apple Documentation Archive. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html#//apple_ref/doc/uid/TP40001067-CH204-TPXREF101, Last accessed on October 8th, 2022.

[39] Apple, Inc. Accelerate | Apple Developer Documentation. https://developer.apple.com/documentation/accelerate, Last accessed on October 8th, 2022.

[40] Apple, Inc. Adapting iOS code to run in the macOS environment | Apple Developer Documentation. https://developer.apple.com/documentation/apple-silicon/adapting-ios-code-to-run-in-the-macos-environment, Last accessed on October 8th, 2022.

[41] Apple, Inc. AppKit | Apple Developer Documentation. https://developer.apple.com/documentation/appkit, Last accessed on October 8th, 2022.

[42] Apple, Inc. Apple Platform Security. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf, Last accessed on October 8th, 2022.

[43] Apple, Inc. Cocoa Application Layer | Apple Developer Documentation. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CocoaApplicationLayer/CocoaApplicationLayer.html, Last accessed on October 8th, 2022.

[44] Apple, Inc. Core Services Layer | Apple Developer Documentation. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreServicesLayer/CoreServicesLayer.html#//apple_ref/doc/uid/TP40001067-CH270-BCICAIFJ, Last accessed on October 8th, 2022.

[45] Apple, Inc. Defining a custom URL scheme for your app | Apple Developer Documentation. https://developer.apple.com/documentation/xcode/

defining-a-custom-url-scheme-for-your-app, Last accessed on October 8th, 2022.

[46] Apple, Inc. Entitlements | Apple Developer Documentation. https://developer.apple.com/documentation/bundleresources/entitlements, Last accessed on October 8th, 2022.

[47] Apple, Inc. Hypervisor | Apple Developer Documentation. https://developer.apple.com/documentation/hypervisor, Last accessed on October 8th, 2022.

[48] Apple, Inc. Identity Pinning: How to configure server certificates for your app. https://developer.apple.com/news/?id=g9ejcf8y, Last accessed on October 8th, 2022.

[49] Apple, Inc. iOS 16. https://www.apple.com/ios/, Last accessed on October 8th, 2022.

[50] Apple, Inc. Kernel and Device Drivers Layer. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/SystemTechnology/SystemTechnology.html#//apple_ref/doc/uid/TP40001067-CH207-BCICAIFJ, Last accessed on October 8th, 2022.

[51] Apple, Inc. Mach-O Overview | Apple Documentation Archive. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html, Last accessed on October 8th, 2022.

[52] Apple, Inc. macOS entitlements | Apple Developer Forum. https://developer.apple.com/forums/thread/653890?answerId=620660022#620660022, Last accessed on October 8th, 2022.

[53] Apple, Inc. macOS Monterey. https://www.apple.com/macos/monterey/, Last accessed on October 8th, 2022.

[54] Apple, Inc. Media Layer | Apple Developer Documentation. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/MediaLayer/MediaLayer.html#//apple_ref/doc/uid/TP40001067-CH273-SW1, Last accessed on October 8th, 2022.

[55] Apple, Inc. Migrating from Cocoa Touch. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/MigratingFromCocoaTouch/MigratingFromCocoaTouch.html#//apple_ref/doc/uid/TP40001067-CH8-SW1, Last accessed on October 8th, 2022.

[56] Apple, Inc. NSPinnedCAIdentities | Apple Developer Documentation. https://developer.apple.com/documentation/bundleresources/information_property_list/nspinnedcaidentities, Last accessed on October 8th, 2022.

[57] Apple, Inc. Required device capabilities. https://developer.apple.com/support/required-device-capabilities/, Last accessed on October 8th, 2022.

[58] Apple, Inc. Running your iOS apps in macOS | Apple Developer Documentation. https://developer.apple.com/documentation/apple-silicon/running-your-ios-apps-in-macos, Last accessed on October 8th, 2022.

[59] Apple, Inc. System Configuration | Apple Developer Documentation. https://developer.apple.com/documentation/systemconfiguration, Last accessed on October 8th, 2022.

[60] Apple, Inc. Transmit 5 | App Store. https://apps.apple.com/at/app/transmit-5/id1436522307, Last accessed on October 8th, 2022.

[61] Apple, Inc. UIApplicationDelegate | Apple Developer Documentation. https://developer.apple.com/documentation/uikit/uiapplicationdelegate/1623112-application, Last accessed on October 8th, 2022.

[62] Apple, Inc. UIKit | Apple Developer Documentation. https://developer.apple.com/documentation/uikit, Last accessed on October 8th, 2022.

[63] Apple, Inc. WhatsApp Messenger | App Store. https://apps.apple.com/us/app/whatsapp-messenger/id310633997, Last accessed on October 8th, 2022.

[64] Apple, Inc. Xcode. https://developer.apple.com/xcode/, Last accessed on October 8th, 2022.

[65] Apple, Inc. XNU. https://opensource.apple.com/source/xnu/, Last accessed on October 8th, 2022.

[66] Arm. A64 general instructions in alphabetical order | ARM Compiler armasm Reference Guide Version 6.00. https://developer.arm.com/documentation/dui0802/a/A64-General-Instructions/A64-general-instructions-in-alphabetical-order?lang=en, Last accessed on October 8th, 2022.

[67] Check Point Ltd. Mobile Security Report 2021. https://pages.checkpoint.com/mobile-security-report-2021.html, Last accessed on October 8th, 2022.

[68] Committee on National Security Systems (CNSS). https://www.cnss.gov/cnss/, Last accessed on October 8th, 2022.

[69] Committee on National Security Systems (CNSS) Glossary. https://rmf.org/wp-content/uploads/2017/10/CNSSI-4009.pdf, Last accessed on October 8th, 2022.

[70] Cryptic Apps EURL. Hopper - The macOS and Linux Disassembler. https://www.hopperapp.com/, Last accessed on October 8th, 2022.

[71] DigiDNA SARL. iMazing. https://imazing.com/, Last accessed on October 8th, 2022.

[72] Docker, Inc. Docker. https://www.docker.com, Last accessed on October 8th, 2022.

[73] Docker, Inc. Install Docker Desktop on Mac. https://docs.docker.com/desktop/install/mac-install/, Last accessed on October 8th, 2022.

[74] European Parliament. Directive (EU) 2015/2366 of the European Parliament and of the Council of 25 November 2015 on payment services in the internal market, amending Directives 2002/65/EC, 2009/110/EC and 2013/36/EU and Regulation (EU) No 1093/2010, and repealing Directive 2007/64/EC. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32015L2366, Last accessed on October 8th, 2022.

[75] Free Software Foundation. GNU Binutils. https://www.gnu.org/software/binutils/binutils.html, Last access on October 8th, 2022.

[76] IETF. https://ietf.org/, Last accessed on October 8th, 2022.

[77] IETF. RFC Editor. https://www.rfc-editor.org/, Last accessed on October 8th, 2022.

[78] Kenneth Reitz. httpbin.org. https://httpbin.org/, Last accessed on October 8th, 2022.

[79] Kim Jong-Cracks. Clutch. https://github.com/KJCracks/Clutch, Last accessed on October 8th, 2022.

[80] LLVM.org. The LLVM Compiler Infrastructure Project. https://llvm.org, Last accessed on October 8th, 2022.

[81] Maple Media LLC. Swiftscan. https://swiftscan.app, Last accessed on October 8th, 2022.

[82] Microsoft, Inc. Visual Studio Code. https://code.visualstudio.com/, Last accessed on October 8th, 2022.

[83] National Initiative for Cybersecurity Careers and Studies. A Glossary of Common Cybersecurity Words and Phrases. https://niccs.cisa.gov/cybersecurity-career-resources/glossary, Last accessed on October 8th, 2022.

[84] OWASP Foundation. M1: Improper Platform Usage. https://owasp.org/www-project-mobile-top-10/2016-risks/m1-improper-platform-usage.html, Last accessed on October 8th, 2022.

[85] OWASP Foundation. M10: Extraneous Functionality. https://owasp.org/www-project-mobile-top-10/2016-risks/m10-extraneous-functionality.html, Last accessed on October 8th, 2022.

[86] OWASP Foundation. M2: Insecure Data Storage. https://owasp.org/www-project-mobile-top-10/2016-risks/m2-insecure-data-storage.html, Last accessed on October 8th, 2022.

[87] OWASP Foundation. M3: Insecure Communication. https://owasp.org/www-project-mobile-top-10/2016-risks/m3-insecure-communication.html, Last accessed on October 8th, 2022.

[88] OWASP Foundation. M4: Insecure Authentication. https://owasp.org/www-project-mobile-top-10/2016-risks/m4-insecure-authentication.html, Last accessed on October 8th, 2022.

[89] OWASP Foundation. M5: Insufficient Cryptography. https://owasp.org/www-project-mobile-top-10/2016-risks/m5-insufficient-cryptography.html, Last accessed on October 8th, 2022.

[90] OWASP Foundation. M6: Insecure Authorization. https://owasp.org/www-project-mobile-top-10/2016-risks/m6-insecure-authorization.html, Last accessed on October 8th, 2022.

[91] OWASP Foundation. M7: Poor Code Quality. https://owasp.org/www-project-mobile-top-10/2016-risks/m7-client-code-quality.html, Last accessed on October 8th, 2022.

[92] OWASP Foundation. M8: Code Tampering. https://owasp.org/www-project-mobile-top-10/2016-risks/m8-code-tampering.html, Last accessed on October 8th, 2022.

[93] OWASP Foundation. M9: Reverse Engineering. https://owasp.org/www-project-mobile-top-10/2016-risks/m9-reverse-engineering.html, Last accessed on October 8th, 2022.

[94] OWASP Foundation. Mobile Security. https://mobile-security.gitbook.io/mobile-security-testing-guide/appendix/0x08-testing-tools, Last accessed on October 8th, 2022.

[95] OWASP Foundation. OWASP Mobile Top 10. https://owasp.org/www-project-mobile-top-10/, Last accessed on October 8th, 2022.

[96] OWASP Foundation. Tampering and Runtime Instrumentation | Mobile Security. https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06c-reverse-engineering-and-tampering#tampering-and-runtime-instrumentation, Last accessed on October 8th, 2022.

[97] Proxyman LLC. Proxyman - Modern. Native. Web Debugging Proxy. https://proxyman.io/, Last accessed on October 8th, 2022.

[98] radare org. radare2. https://rada.re/n/radare2.html, Last accessed on October 8th, 2022.

[99] Ravnås, O. A. V. Frida. https://frida.re/, Last accessed on October 8th, 2022.

[100] Savvy Security. 8 Types of Man in the Middle Attacks You Need to Know About. https://cheapsslsecurity.com/blog/types-of-man-in-the-middle-attacks/, Last accessed on October 8th, 2022.

[101] StackOverflow. Detecting Ad Hoc and App Store builds. https://stackoverflow.com/a/3426899, Last accessed on October 8th, 2022.

[102] Statista. Global smartphone shipments from 2007 to 2021, by vendor. https://www.statista.com/statistics/271539/worldwide-shipments-of-leading-smartphone-vendors-since-2007/, Last accessed on October 8th, 2022.

[103] Statista. Mobile operating systems' market share worldwide from January 2012 to August 2022. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/, Last accessed on October 8th, 2022.

[104] Statista. Number of available apps in the Apple App Store from 1st quarter 2015 to 2nd quarter 2022. https://www.statista.com/statistics/779768/number-of-available-apps-in-the-apple-app-store-quarter/, Last accessed on October 8th, 2022.

[105] Statista. Revenue of mobile apps worldwide 2017-2025, by segment. https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast/, Last accessed on October 8th, 2022.

[106] Sysdig, Inc. Wireshark. https://www.wireshark.org/, Last accessed on October 8th, 2022.

[107] techprimate GmbH. together - Finances App. https://together.techprimate.com/, Last accessed on October 8th, 2022.

[108] U.S. National Security Agency. Ghidra - software reverse engineering suite of tools. https://ghidra-sre.org/, Last accessed on October 8th, 2022.

[109] Wojciech, R. IOSSecuritySuite. https://github.com/securing/IOSSecuritySuite, Last accesssed on October 8th, 2022.

[110] Wojciech, R. et al. DebuggerChecker.swift | IOSSecuritySuite. https://github.com/securing/IOSSecuritySuite/blob/master/IOSSecuritySuite/DebuggerChecker.swift, Last accesssed on October 8th, 2022.

[111] Wojciech, R. et al. JailbreakChecker.swift | IOSSecuritySuite. https://github.com/securing/IOSSecuritySuite/blob/master/IOSSecuritySuite/JailbreakChecker.swift, Last accesssed on October 8th, 2022.

[112] Wojciech, R. et al. ReverseEngineeringToolsChecker.swift | IOSSecuritySuite. https://github.com/securing/IOSSecuritySuite/blob/master/IOSSecuritySuite/ReverseEngineeringToolsChecker.swift, Last accesssed on October 8th, 2022.