Information Security

State of the art iOS penetration testing

Eine empirische Analyse vorhandener Techniken und Tools

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

eingereicht von

Florian Gassner
is171806

im Rahmen des
Studiengangs Information Security an der Fachhochschule St. Pölten

Betreuung
Betreuer/Betreuerin: Dipl.-Ing. Dipl.-Ing. Christoph Lang-Muhr, BSc

St. Pölten, TT.MM.JJJJ

(Unterschrift Autor/Autorin)

(Unterschrift Betreuer/Betreuerin)

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

St. Pölten, TT.MM.JJJJ

_____

(Unterschrift Autor/Autorin)

Zusammenfassung

Nachrichten von Hackerangriffen oder Datendiebstählen sind ständige Begleiter in einer Zeit der zunehmenden Digitalisierung. Es vergeht kaum ein Tag, an dem kein Bericht über die Cyberkriminalität in den Medien zu lesen beziehungsweise zu hören ist. Ein ununterbrochenes Katz-und-Maus-Spiel zwischen Datenschützerinnen und Datenschützern beziehungsweise Entwicklerinnen und Entwickler und Hackerinnen und Hackern. Da mobile Anwendungen im Unternehmensbereich immer mehr an Beliebtheit gewinnen, gelten auch die darin verarbeiteten unternehmenskritischen Daten als besonders schützenswert. Die Aktualisierung beziehungsweise Neuimplementierung von Sicherheitsmaßnahmen durch Apple ist ein laufender Prozess. Jedoch werden immer wieder Wege gefunden um diese Sicherheitsfunktionen zu umgehen. Das Ziel dieser Arbeit ist es, Kenntnisse über die Sicherheitsmaßnahmen von iOS zu erlangen, durch Entwicklerinnen und Entwickler implementierte Schwachstellen aufzuzeigen. Es wird ein Überblick über die bekanntesten Frameworks und deren Einsatz bei praktischen Penetration Tests gegeben. Das Ergebnis dieser Diplomarbeit ist die Darlegung häufiger Implementierungsfehler und eine Gegenüberstellung der am weitest verbreiteten Frameworks (Frida, objection, Cycript und Needle), wobei im Detail aufgezeigt wird, welchen Mehrwert diese für eine Sicherheitsüberprüfung bieten.

Abstract

Messages of hacker attacks or data theft are constant companions in a time of increasing digitalization. Hardly a day goes by without a report on cybercrime being read or heard in the media. Keeping data secure is an uninterrupted cat-and-mouse game between data protectors or developers and hackers. Since mobile applications are becoming more and more popular in the corporate sector, the company-critical data process is specifically worth to be protected. The updating or re-implementation of security measures by Apple is an ongoing process. However, ways are always found to bypass these security functions. The aim of this work is to gain knowledge about the security measures and to identify weak points implemented by developers. An overview of the most popular frameworks and their use in practical penetration tests is given. The result of this diploma thesis is the presentation of common implementation errors. A comparison of the most common frameworks (Frida, objection, Cycript and Needle) is shown in detail and the added value they offer for a security check is illustrated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ACL	Access Control List
ASLR	Address Space Layout Randomization
API	Application Programming Interface
App(s)	Applikation(en)
APT	Advanced Package Tool
AR	Augmented Reality
ARC	Automatic Reference Counting
ATS	App Transport Security
Burp	Burp Suite Scanner
CA	Certificate Authority
DEP	Data Execution Prevention
DVIA	Damn Vulnerable iOS Application
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IMAP	Internet Message Access Protocol
IPA	iOS App Store Package
IPC	Inter Process Communication
MITM	Man-in-the-Middle
MMS	Multimedia Messaging Service
pid	Prozess ID
PIE	Position Independent Executable
plist	Property List
POP3	Post Office Protocol
SMS	Short Message Service
SSH	Secure Shell
TLS	Transport Layer Security
UID	Unique Identifier
URL	Uniform Ressource Locator
UUID	Universal Unique Identifier
XSS	Cross Site Scripting
ZAP	OWASP Zed Attack Proxy

1. Einleitung

Mobile Anwendungen finden auch im Unternehmensbereich immer mehr Verbreitung [1]. Sowohl E-Mails als auch firmeninterne Dokumente sollen für eine Angestellte oder einen Angestellten von überall aus abrufbar sein, sei es vor Ort bei einer Kundin beziehungsweise einem Kunden oder auch auf dem Nachhauseweg. Dies hat zur Folge, dass immer mehr Unternehmen mobile Applikationen (Apps) in den Arbeitsalltag integrieren. Wie ist es jedoch um die Sicherheit der Daten bestimmt? Durch die Mobilität von unternehmenskritischen Informationen ist es besonders wichtig, deren Schutz sicherzustellen und die Zugriffe durch mobile Anwendungen zu überprüfen. Die vorliegende Diplomarbeit trägt diesem Thema Rechnung und setzt sich mit dem mobilen Betriebssystem von Apple auseinander beziehungsweise wie sich darauf befindliche Anwendungen auf deren Sicherheit überprüfen lassen. Darüber hinaus erfolgt eine empirische Analyse von Frameworks, die bei Sicherheitsüberprüfungen unterstützend eingesetzt werden können. Die Analyse von Implementierungsfehlern wird dabei mit einer speziellen Anwendung, der „Damn Vulnerable iOS Application“ (DVIA) [2], durchgeführt. Die Motivation für die Entwicklung dieser Applikation war es eine Anwendung für das Testen und Erweitern von Penetration Testing Skills bereitzustellen. Ziel dieser Diplomarbeit ist das mobile Betriebssystem iOS sowie die von Apple implementierten Sicherheitsfunktionen näher zu betrachten.

Basierend auf einer gründlichen Literaturrecherche erfolgte die Auswahl und Auswertung relevanter Dokumente. Eine darauf aufsetzende empirische Analyse versucht folgende Forschungsfragen zu beantworten:

- Welche Sicherheitsfunktionen sind im mobilen Betriebssystem von Apple vorhanden?
- Welche Phasen durchläuft ein Penetration Test einer iOS Applikation?
- Wie können einzelne Implementierungsfehler überprüft werden?
- Welche Frameworks wirken bei einer Sicherheitsüberprüfung unterstützend?

2. Entstehung von iOS

Bei der Vorstellung des iPhones [3], [4] durch Apple im Jahre 2007 trug das mobile Betriebssystem noch den Namen „iPhone OS“. 2010 [5] erfolgte die Namensänderung des Operating Systems von „iPhone OS“ auf „iOS“ inklusive einer Versionsnummer. Die derzeit aktuellste Fassung hat die Versionsnummer 12, trägt also den Namen „iOS 12“. Apple veröffentlicht in regelmäßigen Abständen Updates des mobilen Betriebssystems (siehe Kapitel 2.2.6). Dabei implementiert der iPhone Hersteller nicht nur neue Funktionen, sondern es erfolgt auch eine stetige Verbesserung der Sicherheit und Stabilität von „iOS“.

2.1. iOS Versionen

Für eine Sicherheitsüberprüfung sind meistens jene Versionen interessant, für die ein Jailbreak (siehe Kapitel 3) vorhanden ist. Jedoch kommt dies stark darauf an, welche Version von iOS von der zu testenden Anwendung unterstützt wird. Am 29. Oktober 2018 wurde von Apple die Verteilung der einzelnen iOS Versionen gemessen.

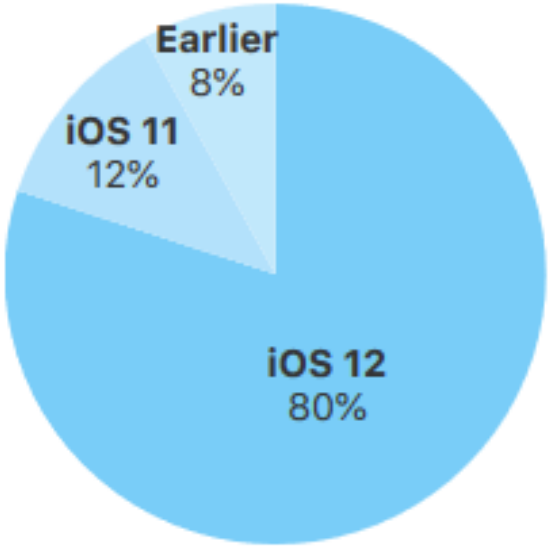

Abbildung 1: Verteilung iOS Versionen [6]

Abbildung 1 verdeutlicht, dass 80 Prozent aller iOS Geräte die aktuelle Hauptversion des Betriebssystems verwenden. Auf insgesamt 20% der iPhones und iPads ist iOS 11 oder eine ältere Version installiert. In den nächsten Unterkapiteln wird auf einige Neuerungen der letzten drei großen Veröffentlichungen näher eingegangen.

2.1.1. iOS 10

In der Version 10 [7] von iOS, welche im September 2016 erschien, wurde das haptische Feedback verbessert, „SiriKit“ implementiert, sowie Änderungen an der Nachrichtenapplikation vorgenommen. Für Penetration Testerinnen und Penetration Tester wichtig sind jedoch Änderungen im Bereich der Sicherheit des Betriebssystems. Zum Beispiel die „SecKey“-Application Programming Interface (API) Verbesserungen für die asymmetrische Schlüsselerzeugung. Die symmetrische RC4-Chiffriersuite ist nun standardmäßig für alle SSL/TLS-Verbindungen deaktiviert, und SSLv3 wird in der Secure Transports-API nicht mehr unterstützt. Jedoch können sowohl die kryptografischen Algorithmen SHA-1, als auch 3DES noch verwendet werden.

2.1.2. iOS 11

Im Herbst 2017 veröffentlichte Apple Version 11 des mobilen Betriebssystems. Diese Aktualisierung stand ganz im Zeichen von „Augmented Reality“ (AR). Das neu implementierte „ARKit Framework“ erlaubt Benutzerinnen oder Benutzer AR Erlebnisse zu erstellen. Des Weiteren wurden einige Änderungen in den App Frameworks vorgenommen. Beispielsweise besteht die Möglichkeit per Drag & Drop Elemente von einem Ort zum anderen auf dem Bildschirm zu ziehen. Dies kann entweder innerhalb einer einzelnen Anwendung als auch applikationsübergreifend verwendet werden. Diese Neuerung [8] könnte, wie in Kapitel 4.4.6.2 beschrieben, zu Side Channel Data Leakage führen.

2.1.3. iOS 12

Die derzeit aktuellste mobile Betriebssystemversion von Apple ist iOS 12 [9] und wurde im September 2018 veröffentlicht. In diesem Update versuchte der iPhone Hersteller die Performance sowie die Stabilität und Sicherheit zu optimieren. Aus diesem Grund erfolgten kaum Implementierungen von neuen Features. Laut Apple verbessert iOS 12 die Leistung bis hin zum iPhone 5s und iPad Air. Des Weiteren starten Applikationen mit dieser neuen Aktualisierung um bis zu 40% schneller als noch unter iOS 11. Dieses Betriebssystem ist noch vergleichsweise jung. Ob in iOS 12 Schwachstellen enthalten sind, welche für einen Penetration Test Relevanz haben, wird sich in naher Zukunft zeigen.

2.2. iOS Operating System Security

Apple [10] hat die iOS Plattform so konzipiert, dass die Sicherheit im Mittelpunkt steht. Dabei erfolgte die Entwicklung und Integration innovativer Funktionen, welche die mobile Sicherheit erhöhen und standardmäßig das gesamte System schützen. Viele dieser Funktionen sind im Regelfall aktiviert, so dass zum Beispiel IT-Abteilungen von Firmen, die Apple Geräte und mobile Anwendungen nutzen, keine umfangreichen Konfigurationen vornehmen müssen. Wichtige Sicherheitsfunktionen, wie zum Beispiel die Geräteverschlüsselung, sind nicht konfigurierbar, so dass Benutzerinnen oder Benutzer diese nicht aus Versehen deaktivieren können. Einige dieser Sicherheitsmaßnahmen werden anschließend näher betrachtet und der Mehrwert, den diese für die Sicherheit eines mobilen iOS Gerätes mit sich bringt, erläutert. Die wichtigsten Implementierungen für den Schutz des Betriebssystems sind folgende:

- Data Execution Prevention (DEP) (W^X)
- ASLR
- Code Signing
- Encryption, Data Protection
- Privilege Model
- Updates

2.2.1. Data Execution Prevention (W^X)

Fehler in der Software (zum Beispiel Buffer Overflow oder Heap Overflow) erlauben es schadhaften Code in den Speicherbereich einer mobilen Anwendung einzuschleusen. Data Execution Prevention [11], [12] wird von vielen Plattformen implementiert, um zu verhindern, dass beliebiger Code im Speicher zur Ausführung kommt. Dieser Mechanismus wird unter iOS auch als „W^X“, also „W XOR X“ genannt. Diese Abkürzung bedeutet, dass der Speicher, welcher zu einer Anwendung gehört, entweder beschreibbar oder ausführbar ist, jedoch unter keinen Umständen beschreib- und ausführbar zur selben Zeit. Der in iOS Geräten verwendete Prozessor verfügt über ein Flag, welches den Zustand für Speicherbereiche steuert. Der Prozessor führt nur Instruktionen in Speicherbereichen aus, die als ausführbar gekennzeichnet sind. Außerdem werden Änderungen nur in jenen Bereichen erlaubt, welche als beschreibbar markiert sind. Ganz verhindert werden Exploits dadurch

nicht, jedoch werden durch diese Sicherheitsmaßnahmen derartige Angriffe schwieriger zu entwickeln und auszuführen.

2.2.2. Address Space Layout Randomization

Alle mit iOS verteilten Anwendungen und Bibliotheken werden so kompiliert, dass diese beim Start den Adressbereich zufällig auswählen. Diese Technik ist unter dem Namen „Address Space Layout Randomization“ (ASLR) [10]–[12] bekannt. Die Zufallsgenerierung des Adressraumlayouts schützt vor der Ausnutzung von Speicherfehlern. Eingebaute Anwendungen verwenden ASLR um sicherzustellen, dass alle Speicherbereiche beim Start zufällig ausgewählt werden. Die zufällige Anordnung der Speicheradressen von ausführbarem Code, Systembibliotheken und verwandten Programmkonstruktionen reduziert die Wahrscheinlichkeit vieler komplexer Exploits. Xcode, die iOS-Entwicklungsumgebung, kompiliert automatisch Programme von Drittanbietenden mit aktivierter ASLR-Unterstützung.

2.2.3. Code Signing

Um sicherzustellen, dass alle Apps aus einer bekannten und genehmigten Quelle stammen und nicht manipuliert wurden, verlangt iOS, dass der gesamte ausführbare Code mit einem von Apple ausgestellten Zertifikat signiert wird. Dieser Vorgang wird von Apple als „Code Signing“ [10], [13] bezeichnet. Nach dem Start des iOS-Kernels wird gesteuert, welche Benutzerprozesse und Anwendungen ausgeführt werden können. Die mit dem Gerät ausgelieferten Apps, wie zum Beispiel die Telefon-Applikation oder der Kalender, sind von Apple signiert. Anwendungen von Drittanbietenden müssen ebenfalls mit einem von Apple ausgestellten Zertifikat validiert und signiert werden. Die obligatorische Code Signatur erweitert das Konzept der Vertrauenskette vom Betriebssystem auf Anwendungen und verhindert, dass Anwendungen von Drittanbietenden unsignierte Coderessourcen laden oder selbst modifizierenden Code verwenden. Jede Applikation muss, bevor diese im Apple App Store freigegeben wird, durch Apple auf die Einhaltung der Richtlinienanforderungen geprüft werden. Dabei wird die Anwendung auf den Inhalt, sowie die zulässige Verwendung von Apple Entwickler APIs kontrolliert. Eine von Apple freigegebene Applikation wird mit dem privaten Schlüssel des Unternehmens signiert und kann somit auf einem iOS Gerät installiert werden.

2.2.4. iOS Encryption, Data Protection

Apple verwendet für die Verschlüsselung [12] von Dateisystem und Daten zwei unterschiedliche Verschlüsselungsfunktionen. iOS Geräte verschlüsseln immer den Flash-Speicher zum Schutz vor Hardwareangriffe, wie beispielsweise „NAND-Dumping“. Als „NAND-Dumping“ wird jener Vorgang bezeichnet, bei dem eine Angreiferin oder ein Angreifer versucht Sicherheitsmechanismen zu umgehen, indem dieser Daten direkt vom ausgebauten Flash-Speicher ausliest. Jeder iOS Prozessor enthält eine eindeutige Identitätsnummer, die sogenannte „Unique Identifier“ (UID). Der UID-Schlüssel ist über keine Softwarefunktionen oder durch bekannte Hardwareangriffe auslesbar. Die UID dient als Schlüssel für die Verschlüsselung des Flash-Speichers.

Die zweite Verschlüsselungsmethode dient zum Schutz der applikationsspezifischen Daten. Apple bezeichnet diesen Mechanismus als „Data Protection“. Dieses Verfahren kommt nur dann zur Anwendung, wenn eine Benutzerin oder ein Benutzer ein Passwort oder einen PIN-Code zum Sperren des iOS Geräts verwendet. Für diese Verschlüsselung kommt wiederum ein eindeutiger Schlüssel zum Einsatz. Nur Applikationen mit entsprechenden Berechtigungen können darauf zugreifen und die Entschlüsselung vornehmen.

2.2.5. Privilege Model

Das iOS Betriebssystem stellt seinen Benutzerinnen beziehungsweise Benutzern minimale Benachrichtigungen über Berechtigungen zur Verfügung. iOS generiert ein Popup-Fenster, sollte eine

Applikation den Zugriff auf eine bestimmte Funktion verlangen. Fordert eine Applikation zum Beispiel Zugriff auf den aktuellen lokalen Standort des Mobilgeräts, kann eine Anwenderin oder ein Anwender den Zugang zu den Standortdaten zulassen oder auch ablehnen. Jedoch stehen einer App zahlreiche alternative Berechtigungen zur Verfügung. Entwicklerinnen und Entwickler können über öffentliche oder private Anwendungsprogrammierschnittstellen den Zugriff auf sensible Daten durch eine Anwendung ermöglichen. Apple legt beim Genehmigungsprozess vor der Veröffentlichung einer Anwendung fest, ob die Funktionalitäten verwendet werden dürfen oder nicht. Der iPhone Hersteller verbietet die Interaktion zweier Anwendungen durch deren Sandbox. Diese Sicherheitsmaßnahme verhindert eine Manipulation von Daten durch eine andere Applikation.

2.2.6. Updates

Apple stellt regelmäßig iOS-Updates für alle Kundinnen und Kunden zur Verfügung. Die Benutzerin oder der Benutzer erhält dabei eine Benachrichtigung, sobald eine neue Version verfügbar ist. Folgende Kategorien von Updates stellt Apple zur Verfügung:

- Update der Hauptversion: ein grundlegendes Update, zum Beispiel von iOS 11 auf iOS 12 (einmal jährlich).
- Major Update: ein großes Update, zum Beispiel von iOS 12.0 auf iOS 12.1 (viermal jährlich).
- Bug-Fix Update: ein kleines Update, zum Beispiel von iOS 12.1 auf iOS 12.1.1 (in unregelmäßigen Abständen).

Die Installation des Updates kann über die native Applikation „Einstellungen“ erfolgen. Der iPhone Hersteller kontrolliert indessen die Ende-zu-Ende Verteilung von Hard- und Software, im Gegensatz zu anderen Mitbewerberinnen und Mitbewerber, welche zum Beispiel auf das Betriebssystem Android von Google setzen. Wird eine neue Version von Android veröffentlicht, müssen die Herstellerinnen und Hersteller Anpassungen am Betriebssystem vornehmen, um die Kompatibilität mit der eigenen Hardware zu gewährleisten. Durch die Ende-zu-Ende Verteilung kann keine Installation von Software durch Firmen, welche für den Mobilfunk verantwortlich sind, oder anderen Quellen, erfolgen. Aus diesem Grund kann Apple Updates schneller und zielgerichteter an die Benutzerinnen oder die Benutzer verteilen, als jede andere Anbieterin oder jeder andere Anbieter von mobilen Geräten. In der Regel liefern neue Versionen verbesserte Sicherheitsmaßnahmen sowie Neuerungen im Bereich der Funktionalität. Bisher gab es kaum Updates, welche nur die Behebung von Schwachstellen adressiert haben. Benutzerinnen oder Benutzer haben die Wahl eine Aktualisierung sofort zu installieren oder die Installation auf später zu verschieben. Verschiebt eine Anwenderin oder ein Anwender das Update auf später, erfolgt die Installation, sobald das iOS Gerät an eine dauerhafte Stromquelle angeschlossen ist. Apple verhindert die Installation älterer iOS Versionen, indem diese nicht mehr signiert werden. Wird eine iOS Version auf einem Gerät installiert, dann wird zu Beginn des Installationsvorganges der Status der zu installierenden Version von den Apple Servern abgefragt. Ist die entsprechende Version nicht mehr signiert, kann diese auch nicht mehr auf dem iOS Gerät aufgebracht werden.

3. Jailbreak

Als „jailbreaking“ [11], [14] werden Prozesse bezeichnet, mit denen versucht wird, einen unberechtigten Zugriff oder erhöhte Privilegien auf einem iOS System zu erhalten. Die Begriffe sind zwischen den einzelnen mobilen Betriebssystemen unterschiedlich. Diese Abweichungen in der Terminologie spiegeln die Unterschiede in den Sicherheitsmodellen wieder, welche von den Herstellerinnen oder Herstellern der Betriebssysteme verwendet werden. Die für die Installation eines Jailbreaks auf einem iOS Mobilgerät verwendete Software entfernt die vom Hersteller entwickelten Einschränkungen und Sicherheitsfunktionen. Dabei kommt es zu unbefugten Änderungen am Betriebssystem durch benutzerdefinierte Kernel. Resultierend daraus ermöglichen Jailbreaks einen Code auszuführen, welcher keine Autorisierung und Signierung von Apple erhalten hat. Auf diese Weise können Benutzerinnen oder Benutzer zusätzliche Anwendungen, Erweiterungen und Patches installieren, welche nicht von Apple freigegeben sind.

3.1. Geschichte der Jailbreaks

Bereits kurz nach der Markteinführung [11] der iPhones wurden die ersten Jailbreaks veröffentlicht, die sich anfangs auf das Installieren und Löschen von Klingeltönen beschränkten. In weiterer Folge wurden kompliziertere Jailbreaks entwickelt, die es ermöglichten, Applikationen von nicht autorisierten Drittanbietenden installieren zu können. Die bekannteste Bezugsquelle von derartigen Anwendungen trägt die Bezeichnung „Cydia“. Auf diesem von Apple nicht bewilligten App-Store wird in Kapitel 3.3 näher eingegangen. Es besteht ein inhärentes Risiko bei der Installation solcher Anwendungen, da diese weder qualitätsgeprüft sind noch den von Apple verpflichteten Genehmigungs- und Anwendungs-genehmigungsprozess durchlaufen haben. Die Gefährdung der Sicherheit des Mobilgeräts kann aufgrund von schadhaften Applikationen erfolgen.

3.2. Arten von Jailbreaks

Die Einteilung von Jailbreaks [11], [14] erfolgt in zwei Hauptgruppen, die sogenannten „tethered“ und „untethered“ Jailbreaks. Diese beiden Kategorien unterscheiden sich hauptsächlich in der Benutzerfreundlichkeit. Die Unterschiede zwischen den zwei Arten von Jailbreaks werden in den folgenden Kapiteln näher erläutert.

3.2.1. Tethered

Ist ein Jailbreak „tethered“ [14], bedeutet dies, dass nach jedem Neustart des Geräts der Vorgang für den Jailbreak wiederholt werden muss. Wird das Smartphone von einer Benutzerin oder einem Benutzer ohne Jailbreak-Tools neu gestartet, dann ist der Kernel auf diesem Gerät nicht mehr gepatcht und damit auch nicht mehr gejailbreakt. Dies kann zur Folge haben, dass das Smartphone in einem teilweise gestarteten Zustand, wie beispielsweise im Wiederherstellungsmodus, stecken bleibt. Damit das Gerät vollständig starten kann und der Jailbreak wieder aktiv ist, muss dieser bei jedem Einschalten mit einem Computer neu ausgeführt werden. Alle zuvor gemachten Änderungen, Installationen von Paketdateien und bearbeitete Systemdateien bleiben auch nach einem Neustart erhalten.

3.2.2. Semi-tethered

Ein Jailbreak ist „semi-tethered“ [14], wenn das Gerät in der Lage ist zu starten, ohne zuvor an einen Computer angeschlossen zu werden. Nach einem Neustart hat das Smartphone keinen gepatchten Kernel mehr und kann deshalb keinen modifizierten Programmcode ausführen. Jedoch kann es normale Funktionen, die das Betriebssystem beinhaltet, weiterhin durchführen. Um den Jailbreak wieder zu aktivieren, muss dieser mit Hilfe eines Computers neu aufgespielt werden.

3.2.3. Semi-untethered

Ein „semi-untethered“ [14] Jailbreak bedeutet, dass zur Wiederaktivierung des Jailbreaks eine Applikation auf dem Gerät verwendet werden kann. Diese Anwendungen werden meistens mittels „Cydia Impactor“ [15] am Gerät installiert. Mit Hilfe von „Cydia Impactor“ können Anwendungen von Drittherstellenden auf einem Mobilgerät aufgespielt werden. Dieser Vorgang wird als „sideloading“ bezeichnet. Wird ein Smartphone mit einem „semi-untethered“ Jailbreak neu gestartet, dann startet es in einen Zustand, in dem der Jailbreak nicht aktiv ist. Um den Jailbreak wiederherzustellen, muss die Applikation nach jedem Neustart ausgeführt werden.

3.2.4. Untethered

Ist ein Jailbreak „untethered“ [14], heißt das, dass die Benutzerin oder der Benutzer nach Belieben das Gerät aus- und einschalten kann, ohne den Jailbreak zu verlieren. Diese Jailbreaks nutzen Schwachstellen aus, die es erlauben, den Kernel ohne Computer zu patchen. Diese Jailbreaks sind für den Endnutzer am angenehmsten, jedoch auch am seltensten.

3.3. Cydia

Eines der begehrtesten Effekte des Jailbreaks eines iOS Geräts ist der Zugriff auf den alternativen App Store „Cydia“. „Cydia“ basiert auf der beliebten Debian Linux „Advanced Package Tool“ (APT) Paketverwaltungsmethode und ist standardmäßig in fast allen Jailbreaks enthalten. Der Vertrieb von kostenlosen aber auch kommerziellen Anwendungen erfolgt über diesen App Store. Selbst von Apple aus dem App Store verbannte Applikationen sind dadurch beziehbar. Des Weiteren bietet „Cydia“ Anwendungen mit zweifelhafter Legalität an. Für Penetration Testerinnen und Tester stellt diese alternative Bezugsquelle qualitativ hochwertige Werkzeuge zur Verfügung, welche eine Sicherheitsüberprüfung maßgeblich vereinfachen. Saurik [16], der Entwickler von Cydia, hat Ende 2018 bekanntgegeben, dass der Cydia Store nicht mehr weiterentwickelt wird. Der Cydia Store ist das Backend-Zahlungssystem, mit dem Jailbreak-Tweaks aus Repositories erworben werden konnten. Die für Penetration Tests wichtige Paketverwaltung kann weiterhin verwendet werden und wird nach wie vor mit dem aktuellsten iOS 12 Jailbreak namens „Unc0ver“ [17] ausgeliefert.

4. iOS Penetration Testing

Ein Penetration Test [18] dient dazu, die Sicherheit einer oder mehreren Systemkomponenten zu testen und zu bewerten. Dazu sollten Erfahrungen im Bereich Web, Wireless und mobile Applikationen vorhanden sein, um alle zu einem Penetration Test dazugehörende Kategorien abzudecken. Eine Sicherheitsüberprüfung einer mobilen Anwendung teilt sich in folgende Phasen:

- Pre-Engagement
- Reconnaissance
- Scanning
- Exploitation
- Post-Exploitation

Die Betrachtung der einzelnen Phasen sowie die damit einhergehenden Aufgaben werden in den folgenden Unterkapiteln näher beschrieben.

4.1. Phasen eines iOS Penetration Tests

Um einen Penetration Test [18] eines iPhones, beziehungsweise einer Applikation, welche auf einem iPhone installiert wurde, durchführen zu können, müssen gewisse Vorbereitungen getroffen werden. Diese Vorbereitungen sind notwendig um die Sicherheitsüberprüfung effizient und schnell durchführen zu können.

Es besteht die Notwendigkeit mindestens zwei mobile Geräte mit einer Betriebssystemversion, welche von der Applikation unterstützt wird, zur Verfügung zu haben. Des Weiteren müssen Nutzungsbeschränkungen des iPhone Herstellers mit Hilfe eines Jailbreaks auf einem der Geräte entfernt werden. Als Folge dessen können Softwarepakete installiert werden, die für einen Penetration Test benötigt werden. Die Notwendigkeit zweier Testgeräte besteht darin, dass sich eine Applikation auf einem iPhone mit Jailbreak anders verhält und Funktionalitäten verändern beziehungsweise nicht bereitstellen könnte.

Darüber hinaus braucht es auch ein macOS-fähiges Gerät, da viele der Werkzeuge nur für dieses Betriebssystem konzipiert wurden. Ebenso sind die Xcode [19] Entwicklungsumgebung und das iOS SDK nur für macOS verfügbar. Das hat zur Folge, dass Quellcode-Analysen sowie Debugging nur auf einem Gerät funktioniert, auf dem das Apple Desktopbetriebssystem ausgeführt wird.

Im nachfolgenden Kapitel wird grafisch dargestellt, welche Schritte eine Sicherheitsüberprüfung durchlaufen kann.

4.1.1. Workflow

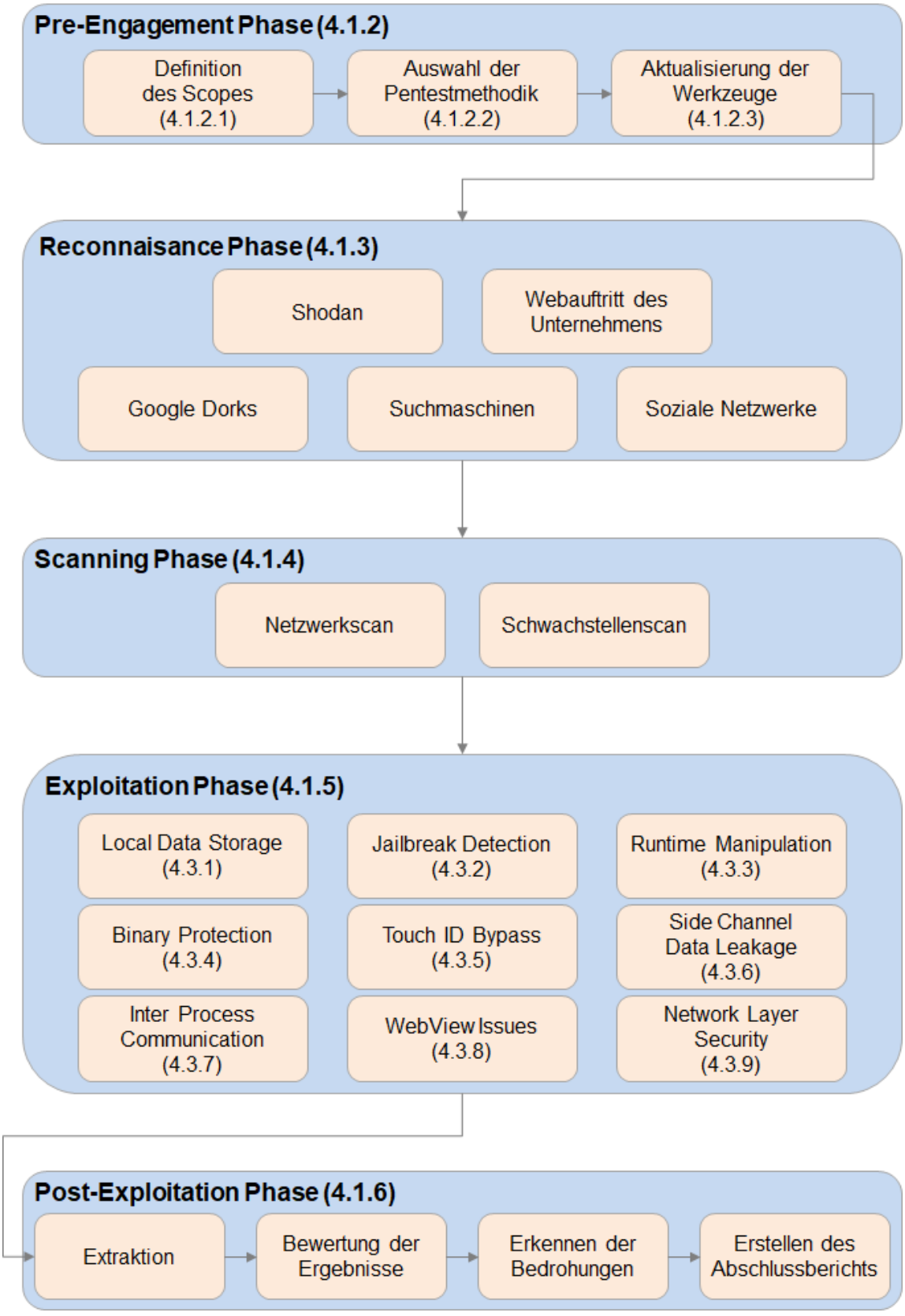

Abbildung 2: Workflow eines Penetration Tests

4.1.2. Pre-Engagement

Die Pre-Engagement Phase [18] eines Penetration Tests wird in folgende Abschnitte unterteilt:

- Definition des Scopes
- Auswahl des Pentestmethodik
- Aktualisierung der Werkzeuge

4.1.2.1. Definition des Scopes

In der Pre-Engagement Phase eines Penetration Tests müssen alle Parameter, zum Beispiel die zu verwendenden mobilen Geräte, iOS Version, Systeme und Netzwerke, definiert werden. Dazu muss allen involvierten Parteien der Umfang des Tests bekannt sein. Um den Umfang der Sicherheitsüberprüfung abzustecken, kann eine Kick-Off Besprechung durchgeführt werden.

Bei einem Penetration Test einer mobilen Anwendung, müssen verschiedene Systemkomponenten beachtet werden. Diese sind:

- Anwendung für ein mobiles iOS-Gerät
- Smartphones und Tablets
- klassische Computersysteme wie Laptop und Desktop-PCs, auf denen eine Sicherungskopie gespeichert wird
- Back-End Systeme, welche für die Verwendung der Applikation gebraucht werden

4.1.2.2. Auswahl des Pentestmethodik

Des Weiteren ist es essentiell mit der Auftraggeberin oder dem Auftraggeber eines Penetration Tests abzuklären, welche Art von Tests durchgeführt werden sollen. Folgende unterschiedliche Testvarianten können dabei betrachtet werden:

- Whitebox [20]: Bei einem sogenannten „Whitebox“ Test stellt die Auftraggeberin oder der Auftraggeber alle Informationen über die Anwendung zur Verfügung. Dazu gehören zum Beispiel der Quellcode, etwaige Dokumentationen, Netzwerkdiagramme. Vorteile eines „Whitebox“-Tests sind die erhöhte Anzahl an erkannten Schwachstellen, der geringere Einarbeitungsaufwand sowie die Automatisierung der statischen Codeanalyse, da der Code vorliegt.
- Blackbox [20]: Der „Blackbox“ Test ist ein Test, welcher die Situation einer echten Angreiferin beziehungsweise eines echten Angreifers am besten widerspiegelt. Es werden keinerlei Informationen über die Applikation oder der dahinterliegenden Infrastruktur zur Verfügung gestellt. Nachteil dieser Art einer Sicherheitsüberprüfung ist, dass, sofern die Testerinnen oder die Tester die Sicherheitsmaßnahmen nicht durchbrechen können, alle Schwachstellen der dahinterliegenden Dienste unentdeckt bleiben und nicht behoben werden.
- Greybox [20]: Der „Greybox“-Test ist eine Kombination aus „Whitebox“- und „Blackbox“-Test. Bei dieser Art des Tests steht der Source Code nicht zur Verfügung, jedoch eine spezielle Version der zu überprüfenden Applikation. Um den Nachteil des „Blackbox“-Tests zu kompensieren sind etwaige Sicherheitsmaßnehmen, wie zum Beispiel Certificate Pinning oder Jailbreak Erkennung, deaktiviert.

4.1.2.3. Aktualisierung der Werkzeuge

Ein weiterer wichtiger Punkt in der ersten Phase des Tests ist es, alle Tools, die in den nachfolgenden Phasen ihre Verwendung finden, auf deren Aktualität zu überprüfen und gegebenenfalls zu aktualisieren. Dies ist notwendig, da viele Programme Open Source sind und von deren Urheberinnen oder Urheber in ihrer Freizeit entwickelt beziehungsweise weiterentwickelt werden. Die Weiterentwicklungen beinahlten oftmals neue Funktionen aber auch erhoffte Stabilitätsverbesserungen. Aus diesem Grund kommt es bei manchen Werkzeugen immer wieder zu Abstürzen, weshalb die Erwartungen an deren Zuverlässigkeit nicht zu hoch gesetzt werden kann.

4.1.3. Reconnaissance

Die Reconnaissance Phase [18] dient dazu, Informationen über die zu testende Applikation sowie die dahinterliegende Umgebung über öffentlich zugängliche Ressourcen zu sammeln. Hilfreich dafür sind Suchmaschinen, soziale Netzwerke, aber auch der Internetauftritt des Unternehmens, welches einen Penetration Test in Beauftragung gegeben hat.

Über Suchmaschinen kann zum Beispiel nach dem Namen der Applikation gesucht werden, um eventuell verwendete Frameworks oder Technologien ausfindig zu machen. Shodan [21] ist eine gute Option verwundbare Systeme in der zu testenden Umgebung zu finden. Eine weitere Möglichkeit Informationen über das Unternehmen und die zu testende Applikation, die eingesetzten Technologien, aber auch mehr über die dahinterliegende Infrastruktur zu sammeln, bietet Google Dorks [22]. Google Dorks kann dazu verwendet werden, um potenziell verwundbare Ports aufzulisten, sowie Dateien aber auch sensible Informationen offenzulegen, welche auf den für das Hosting verantwortlichen Servern liegen. Durch diese gesammelten Daten kann beispielsweise im Vorfeld schon auf eingesetzte Frameworks und Technologien geschlossen und etwaige ausnutzbare Schwachstellen identifiziert werden.

4.1.4. Scanning

Die Scanning-Phase [18] stellt einen wesentlichen Teil der Zeit dar, der für eine Sicherheitsüberprüfung aufgewendet wird. Zu dieser Phase gehören unter anderem das Enumerieren der Netzwerkumgebung und der mobilen Endgeräte, um Systeme und Dienste, welche mit der Applikation in Verbindung stehen, zu identifizieren. Des Weiteren kann auch ein Schwachstellenscan der involvierten Umgebung erfolgen. Die in der Scanning Phase gesammelten Informationen können in weiterer Folge dazu genutzt werden um Schwachstellen zu erkennen. In dieser Phase kommen Schwachstellenscans, sowie Netzwerkscans zum Einsatz.

4.1.5. Exploitation

In der Exploitation Phase [18] werden identifizierte Fehler und Schwachstellen mit Hilfe von Open Source, kommerziellen, aber auch benutzerdefinierten Tools, ausgenutzt. In diesem Abschnitt des Penetration Tests kann die Stabilität und Verfügbarkeit von den betroffenen Systemen stark beeinträchtigt werden. Da es in dieser Phase zu Ausfällen der im Test verwendeten Applikationen und Backend-Systeme kommen kann, sollte unbedingt davor mit der auftraggebenden Organisation abgeklärt werden, ob die Sicherheitsüberprüfung am Produktivsystem oder an einem separaten Testsystem durchgeführt werden soll. Bei der Durchführung der Schwachstellenüberprüfung (siehe Kapitel 4.4) gibt es keine festgelegte Reihenfolge, abhängig vom zuvor definierten Scope des Penetration Tests können einzelne Tests auch entfallen.

4.1.6. Post-Exploitation

Die Post-Exploitation Phase [18] dient dazu, repräsentative Informationen aus den erfolgreich eingedrungenen Systemen zu extrahieren und zu bewerten. Dieser Teil eines Penetration Tests ist unerlässlich, um das Risiko

einer identifizierten Schwachstelle für die auftraggebende Organisation zu erkennen. Die Sicherheitsüberprüfung wird mit einem Bericht, in dem alle relevanten Schwachstellen zusammengefasst sind, sowie mit einer Endbesprechung abgeschlossen.

4.2. Tools

Im Zuge der Diplomarbeit kommen folgende Tools zum Einsatz und werden in den nächsten Unterpunkten betrachtet:

- Intercepting Proxy (Burp Suite Scanner)
- keychain_dumper
- Realm Browser
- BinaryCookieReader.py
- ios-deploy
- SSH Client (Putty / Terminal)
- otool
- class-dump
- Xcode
- security
- Frameworks

4.2.1. Intercepting Proxy

Ein Intercepting Proxy ermöglicht es Testerinnen und Testern alle Anfragen und Antworten zwischen Browser und Zielanwendung abzufangen, auch wenn das Hypertext Transfer Protocol Secure Protokoll verwendet wird. Dabei können einzelne Nachrichten angezeigt, bearbeitet oder gelöscht werden, um die server- oder clientseitige Komponente einer Anwendung zu manipulieren. In dieser Diplomarbeit wurde auf den Intercepting Proxy „Burp Suite Scanner“ [23] zurückgegriffen. Der Proxy findet in Kapitel 4.4.9 seine Verwendung.

4.2.2. keychain_dumper

Das Programm “keychain_dumper” [11], [24] wird verwendet um Keychain-Einträge (siehe Kapitel 4.4.1.4) auf iOS-Geräten auszugeben. Voraussetzung dafür ist ein aktiver Jailbreak (siehe Kapitel 3).

4.2.3. Realm Browser

Der “Realm Browser” [25] ist ein Programm zum Anzeigen und Editieren von „.realm“ Datenspeicherdateien (siehe Kapitel 4.4.1.6). Das Programm kann direkt aus dem Mac App Store bezogen werden.

4.2.4. BinaryCookieReader.py

Das Skript “BinaryCookieReader.py” [26] kann dazu verwendet werden, um persistente Cookies aus der binären Cookies.binarycookies-Datei zu extrahieren. In Kapitel 4.4.6.3 wird gezeigt, dass das Skript die binäre Datei in lesbarer Form darstellen kann.

4.2.5. ios-deploy

“ios-deploy” [27] wurde entwickelt, um iOS-Anwendungen über die Befehlszeile zu installieren und zu debuggen. Dies funktioniert auch auf iPhones oder iPads, welche keinen aktiven Jailbreak aufweisen. In

Kapitel 4.4.9.2 wird veranschaulicht, wie mit „ios-deploy“ eine Applikation auf einem iOS-Gerät ohne Jailbreak installiert werden kann.

4.2.6. SSH Client

Um eine Verbindung über das Secure Shell (SSH) Protokoll mit einem iOS-Gerät mit aktiven Jailbreak aufzubauen, wird ein SSH Client benötigt. Dabei kann auf Putty [28] oder das Programm „Terminal“, welches standardmäßig in macOS enthalten ist, [29] von Apple zurückgegriffen werden. Das von Apple vergebene Standardpasswort lautet "alpine" [30].

4.2.7. otool

Mit dem Programm “otool” können bestimmte Teile von Objektdateien oder Bibliotheken angezeigt werden. In dieser Diplomarbeit wird „otool“ dazu verwendet, um binäre Schutzmechanismen einer Applikation (siehe Kapitel 4.4.4) zu identifizieren.

4.2.8. class-dump

„class-dump“ [31] ist ein Programm zur Untersuchung des Objective-C-Segments von Mach-O-Dateien. Dies ermöglicht eine Analyse, welche Methoden (siehe Kapitel 4.4.3) in der ausführbaren Datei existieren.

4.2.9. Xcode

“Xcode” [19] ist die Entwicklungsumgebung von Apple für macOS und iOS. Mit Hilfe von „Xcode“ können Programme für alle Apple Plattformen entwickelt werden. Sicherheitsforscherinnen und Sicherheitsforscher können unter Zuhilfenahme der Entwicklungsumgebung Provisioning Profiles und Code-Signing Zertifikate erstellen, um in weiterer Folge manipulierte iOS App Store Package (IPA)-Dateien signieren (siehe Kapitel 4.4.9.2) und installieren zu können.

4.2.10. security

Das “security” [32] Kommandozeilenprogramm kann für das Verwalten von Keychains, Schlüssel, Zertifikaten und das Security-Framework verwendet werden. In Kapitel 4.4.9.2 wird „security“ eingesetzt, um das Vorhandensein des Code-Signing Zertifikats zu bestätigen, da der Hash des Zertifikats in weiterer Folge für das Manipulieren einer IPA-Datei verwendet wird.

4.3. Frameworks

Um die Arbeit einer Penetration Testerin, beziehungsweise eines Penetration Testers immens zu vereinfachen, wurden Frameworks entwickelt. Diese Frameworks assistieren innerhalb einer Sicherheitsüberprüfung dahingehend, dass diese die statische ebenso wie die dynamische Begutachtung unterstützen. Darüber hinaus können einige der Frameworks dafür genutzt werden, um Methoden und deren Rückgabewerte zur Laufzeit zu verändern. Das bedeutet, dass Sicherheitsmechanismen, wie beispielsweise Apples iOS Touch ID oder auch Jailbreak Erkennungen, wirkungslos gemacht werden können. Um Anwendungen zur Laufzeit zu untersuchen, werden Skripte mit einer Mischung aus Objective-C und JavaScript in einen Prozess injiziert. Folgende Frameworks werden in diesem Kapitel näher behandelt:

- Frida
- Cycript
- Needle

In diesem Zusammenhang werden die Aktualität der Frameworks und welche Leistungsmerkmale ebendiese bieten näher betrachtet.

4.3.1.1. Frida

Frida [33] ist ein Toolkit für die dynamische Codeinstrumentierung. Es ermöglich Entwicklerinnen und Entwickler, Reverse Engineerer sowie Sicherheitsforscherinnen und Sicherheitsforscher Ausschnitte von JavaScript oder der eigenen Bibliothek in native Anwendungen unter iOS einzubinden. Außerdem bietet Frida auch einige einfache Tools, die auf der Frida-API basieren. Diese können unverändert verwendet werden oder als Beispiele für die Verwendung der API dienen. So kann beispielsweise in einem iOS Anwendungsprozess dieser eingebunden werden, um die Ausgabe des Prozesses selbst zu extrahieren. Dabei spricht man von „Hooking“. Der Begriff „Hooking“ umfasst eine Reihe von Techniken, welche verwendet werden, um das Verhalten eines Betriebssystems, einer Anwendung oder anderer Softwarekomponenten zu verändern oder zu erweitern. Dies geschieht, indem Funktionsaufrufe, Rückgabewerte oder Ereignisse abgefangen werden, die zwischen Softwarekomponenten übertragen werden. Folgende Besonderheiten bietet Frida:

- Skriptbar: Einfügen von benutzerdefinierten Skripts in Prozesse, um eine Logik auszuführen. Es kann dabei jede Funktion gehookt, jeder API Aufruf überwacht sowie Anwendungscode zur Laufzeit verfolgt werden.
- Multi Plattform: Frida funktioniert unter Windows, macOS, Linux, Android und, speziell für diese Thesis wichtig, unter iOS.
- Kostenlos und Open Source: Den Entwicklerinnen und Entwicklern zufolge soll Frida immer frei verfügbar bleiben.

Frida bietet einige Werkzeuge, welche bei einem Penetration Test unterstützen:

- Frida CLI: Eine Schnittstelle, welche Rapid Prototyping sowie einfaches Debugging ermöglicht.
- frida-ps: Kommandozeilen-Tool, das zum Auflisten von Prozessen verwendet werden kann. Dies ist besonders hilfreich, für den Fall, dass mit einem Remotesystem gearbeitet werden muss.
- frida-trace: Dient zur dynamischen Ablaufverfolgung von Funktionsaufrufen.
- frida-discover: Werkzeug zum Auffinden interner Funktionen in einer Anwendung, welche dann mit „frida-trace“ nachvollzogen werden können.
- frida-ls-devices: Kann für die Auflistung von angeschlossenen Geräten verwendet werden und sich als nützlich erweisen, sollte mit mehreren Smartphones gearbeitet werden.
- frida-kill: Eignet sich, um Prozesse auf verbundenen Geräten zu beenden.

Das Framework unterstützt drei Betriebsarten:

- Injected
- Embedded
- Preloaded

Die gängigste Art Frida zu benutzen, ist der „Injected“ Modus. Hierdurch kann eine Penetration Testerin oder ein Penetration Tester eine bestehende Applikation spawnen oder sich an eine laufende Applikation anhängen, um danach das Skript in diesem Programm auszuführen. Voraussetzung für dieses Verfahren ist ein Jailbreak auf dem Smartphone. Ist ein solcher vorhanden, kann über den Paket Manager „Cydia“ das komplette Frida Paket auf dem Smartphone installiert werden. Als Folge dessen wird auf dem Endgerät der

Frida Server eingerichtet. Dieser dient dazu, den Code in den laufenden Prozess einer Applikation zu injizieren.

Sollte das Smartphone jedoch keinen aktiven Jailbreak besitzen, muss, damit eine Applikation mit Frida instrumentiert werden kann, die Frida Bibliothek in die Anwendung geladen werden. Diesen Vorgang wird als „Embedded“ Modus bezeichnet. Für derartige Fälle existiert das sogenannte „frida-gadget“, eine Bibliothek, welche in eine Applikation eingebettet werden kann. Der Prozess dafür wurde bereits in Kapitel 4.4.9.2 beschrieben.

Der dritte Betriebsmodus, „Preloaded“, verändert die ausführbare Datei nicht. Stattdessen wird ein JavaScript Skript geladen, bevor das Programm zur Ausführung kommt. Dies ist für iOS Penetration Tests kaum relevant und wird aus diesem Grund auch nicht weiter betrachtet.

Die Interaktion mit dem Frida Server ist sowohl bei einem nicht vorhandenen Jailbreak, als auch bei einem Gerät mit Jailbreak gleich. Steht die Verbindung zwischen dem Server und der Kommandozeile, kann mit der dynamischen Begutachtung der Applikation begonnen werden. Folgende Funktionalitäten stellt Frida innerhalb des iOS Betriebssystems zur Verfügung:

- auflisten aller Prozesse
- auflisten aller angeschlossenen Geräte
- verfolgen nativer APIs
- verfolgen von Objective-C APIs
- zurückverfolgen eines Objective-C-Methodenaufrufs
- schreiben von Daten in eine Datei
- aufrufen einer nativen Funktion.

Die aktuellste Version von Frida unterstützt iOS Versionen bis zur elften sowie Betaversionen von iOS 12 und wird von den Entwicklerinnen und Entwicklern aktiv betreut.

4.3.1.2. objection

„objection“ ist ein Werkzeug, welches auf Frida aufbaut und Applikationen zur Laufzeit manipulieren kann. Es wurde mit dem Ziel entwickelt, die Überprüfung mobile Anwendungen und ihre Sicherheitslage zu vereinfachen. Weshalb viele Funktionen, wie zum Beispiel Touch ID sowie Certificate Pinning Bypass, schon implementiert sind. Des Weiteren wurde „objection“ dahingehend entwickelt, dass ein aktiver Jailbreak nicht zwingend notwendig ist. Folgende Funktionen sind in „objection“ enthalten und müssen nicht erst durch die Penetration Testerin oder den Penetration Tester implementiert werden:

- Ausgabe des Inhalts der iOS Keychain,
- Ausgabe von Daten aus NSUserDefaults sowie NSHTTTPCookieStorage,
- Umgehung bestimmter Implementationen von Touch ID und Certificate Pinning,
- Überwachung vom Aufruf einer Methode,
- Überwachung des iOS Pasteboards sowie
- Ausgabe von .plist Dateien in lesbarer Form.

4.3.1.3. Cycript

Cycript [34] ist ein JavaScript Interpreter, welcher auch die Objective-C Syntax versteht. Das bedeutet, dass sowohl JavaScript als auch Objective-C in einem Befehl vermischt werden können. Auch Cycript kann dazu verwendet werden, um es in einen laufenden Prozess einzubinden und die Anwendung während der Laufzeit zu verändern, beziehungsweise zu manipulieren. Folgende Anwendungen bietet Cycript in Bezug auf iOS-Applikationen:

- Hooken eines laufenden Prozesses und Extrahierung der Namen aller verwendeten Klassen. Dies bedeutet, dass sowohl die View-Controller als auch die verwendeten internen und externen Bibliotheken ausgegeben werden können.
- Generieren einer Liste an verwendeten Methoden für eine bestimmte Klasse.
- Entnahme der Namen aller Instanzvariablen und deren Werte zu einem bestimmten Zeitpunkt während der Laufzeit einer Anwendung.
- Verändern von Werten der Instanzvariablen zur Laufzeit.
- Ersetzen von Code einer bestimmten Methode durch eine andere benutzerdefinierte Implementierung.
- Aufruf jeglicher Methode in einer Anwendung, auch wenn diese nicht im eigentlichen Code enthalten ist.

Erfolgt eine Manipulation einer Applikation mit Cycript, passiert dies auf eine nicht persistente Weise. Daraus ergibt sich, dass vorgenommene Änderungen nur im Speicher erfolgen. Kommt es zu einem Neustart der Anwendung, gehen diese Modifikationen verloren. Es stehen mehrere Lösungen zur Verfügung, um die Anwendung von Cycript Manipulationen zu automatisieren. Eine Möglichkeit ist beispielsweise, bei jedem Start einer Anwendung das entsprechende Cycript Skript auszuführen. Um mit Cycript entwickeln zu können, sollten Kenntnisse im Bereich Objective-C, iOS APIs sowie JavaScript vorhanden sein.

Ein Nachteil von Cycript ist, dass die offizielle Version nicht mehr weiterverfolgt wird. Jedoch nahm sich das Unternehmen „NowSecure“ [35] dieses Problems an und entwickelte ein neues Cycript Backend, welches auf Frida basiert. Das ermöglicht die Ausführung von Cycript auf allen Plattformen, die auch von Frida unterstützt werden. Dies hat zur Folge, dass Cycript sowohl auf iOS Smartphones mit als auch ohne Jailbreak verwendet werden kann. Es kann davon ausgegangen werden, dass in Zukunft alle Funktionalitäten von Cycript direkt in Frida aufrufbar sind.

4.3.1.4. Needle

Needle [36] ist ein modulares Framework, welches zur Optimierung des Prozesses zur Durchführung von Sicherheitsüberprüfungen von iOS Anwendungen verwendet werden kann. Ein Penetration Test einer mobilen iOS Applikation erfordert in der Regel eine Vielzahl an Werkzeugen. Jedes dieser Werkzeuge wurde dabei für einen bestimmten Bedarf entwickelt. Durch Needle wird nunmehr versucht, viele Tools in einem Framework zu verpacken und zu vereinheitlichen. Das von „MWR InfoSecurity“ entwickelte Framework zielt darauf ab, den gesamten Prozess einer Sicherheitsüberprüfung an einem zentralen Punkt zu rationalisieren. Needle soll nicht nur für Sicherheitsexperten nützlich sein, sondern auch für Entwicklerinnen und Entwickler, welche ihre Applikationen auf Schwachstellen überprüfen wollen. Das Framework kann für folgende Aufgaben herangezogen werden:

- Überprüfung der sicheren Datenspeicherung,
- Validierung von korrekter Interprozesskommunikation,
- Prüfen der Netzwerkkommunikation,

- statische Codeanalyse,
- Hooking von Klassen und Methoden, sowie
- Verifikation der Verwendung von binären Sicherheitsfunktionen.

Ist die Installation und Einrichtung von Needle erfolgt, steht dem Beginn der Sicherheitsüberprüfung nichts mehr im Wege. Folgende Befehle stellt das Framework zur Verfügung:

- show modules: Liste der Module, welche für den Penetration Test verfügbar sind.
- use: Laden eines bestimmten Moduls.
- info: Anzeigen von Hilfs- und Nutzungsinformationen für ein Modul.
- set: Dient zur Steuerung globaler oder modulspezifischer Einstellungen.
- run: Ausführen des geladenen Moduls.
- back: Verlassen des Moduls und Rückkehr zum globalen Konfigurationsmodus.
- shell: Öffnen einer SSH-Shell auf dem iOS Gerät.

Folgende Module sind im Needle Framework enthalten:

- Binary Modules: Diese Module stellen Funktionalitäten zur Verfügung, um Informationen über Anwendungen und deren Applikationsdateien zu sammeln. Des Weiteren helfen diese Module bei Analyse- sowie Systemkonfigurationsaufgaben.
- metadata Module: Das metadata Modul sammelt grundlegende Informationen über die Zielapplikation.
- Dynamic Modules: Um dynamische Informationen von laufenden iOS Anwendungen zu erhalten, kann die Verwendung der dynamischen Module nützlich sein.
- heap_dump Module: Dieses Modul durchsucht den zugewiesenen Speicher einer Applikation nach einer gewünschten Zeichenkette.
- Storage Modules: Die Speichermodule von Needle automatisieren den Prozess der Datenerfassung von einem iOS Gerät.
- files_binarycookies Module: Um die Cookies.binarycookies Datei einer Applikation zu lesen, kann der Einsatz des files_binarycookies Moduls nützlich sein.

Voraussetzung um mit Needle arbeiten zu können, ist ein aktiver Jailbreak auf einem iOS Gerät. Des Weiteren unterstützt das Framework nur die iOS Versionen 8, 9 sowie 10.

4.4. Überprüfbare Sicherheitsrisiken

In den folgenden Unterpunkten wird auf Sicherheitsrisiken eingegangen, welche bei einem Penetration Test einer mobilen iOS Applikation untersucht werden können. Dabei wird aufgezeigt, welche Werkzeuge und Methoden bei Sicherheitsüberprüfungen zum Einsatz kommen. Des Weiteren wird versucht, Auswirkungen etwaiger Schwachstellen aufzuzeigen. Folgende Sicherheitsrisiken [2] werden in den kommenden Unterpunkten näher betrachtet:

- Local Data Storage
- Jailbreak Detection
- Runtime Manipulation
- Binary Protection

- Touch ID Bypass
- Side Channel Data Leakage
- Inter Process Communication (IPC) Issues
- WebView Issues
- Network Layer Security

4.4.1. Local Data Storage

Das mobile Gerät einer Benutzerin oder eines Benutzers speichert viele sensible Daten, darunter Passwörter, E-Mails, Notizen und Browser-Lesezeichen. Darüber hinaus speichern viele Geräte auch andere Inhalte im Rahmen der Betriebssystemfunktionalität, wie zum Beispiel Short Message Service- (SMS) oder Multimedia Messaging Service (MMS)-Nachrichten, Verlauf von eingehenden und ausgehenden Anrufen sowie Tastenanschläge. Des Weiteren werden andere Ressourcen wie Binärdateien von Anwendungen, digitale Zertifikate sowie das Wörterbuch der Tastatur häufig auf Geräten gespeichert, was für eine Angreiferin oder einen Angreifer sehr nützlich sein kann. Die wichtigsten Speicherarten und -orte sind folgende:

- App Folder Structure
- plist
- NSUserDefaults
- Keychain
- CoreData
- Realm

Werden sensible Daten unverschlüsselt am iOS Gerät abgespeichert, ist dies auf einen Implementierungsfehler der Entwicklerin oder des Entwicklers zurückzuführen. Apple bietet zwar mit der Keychain die Möglichkeit, Daten verschlüsselt am Gerät abzulegen, sollte jedoch ein Jailbreak auf einem Gerät aktiv sein, können auch diese Informationen im Klartext ausgelesen werden (siehe Kapitel 4.4.1.4). Aus diesem Grund sollte die Applikation sensible Daten verschlüsseln bevor diese am iOS Gerät abgespeichert werden.

4.4.1.1. App Folder Structure

Bei der Installation einer iOS-Applikation auf einem mobilen Gerät gibt es Strukturen [37], die immer ident sind. Seit der iOS Version 10 werden Anwendungen unter „/private/var/containers/Bundle/Application“ entpackt. Jede Applikation wird durch eine Universal Unique Identifier (UUID), eine 128-Bit Nummer, eindeutig identifiziert. Folgende Ordner sind für Sicherheitsüberprüfungen interessant, da diese wichtige oder sensible Informationen enthalten können:

- /private/var/mobile/Containers/Data/Application/[UUID]/Application.app – Dieser Ordner umfasst statischen Inhalt sowie die kompilierte Binärdatei der Anwendung. Des Weiteren wird der Inhalt dieses Ordners zur Überprüfung der Code-Signatur verwendet.
- /private/var/mobile/Containers/Data/Application/[UUID]/Documents – Beinhaltet benutzergenerierte Daten. Diese Daten werden durch das Benutzen der Applikation durch die Anwenderin oder den Anwender erstellt.
- /private/var/mobile/Containers/Data/Application/[UUID]/Library – Enthält Dateien, welche nicht benutzerspezifisch sind, wie zum Beispiel: Caches, Einstellungen, Cookies und Konfigurationsdateien der Property List (plist).

- /private/var/mobile/Containers/Data/Application/[UUID]/tmp – In diesem Order werden temporäre Dateien gespeichert, welche zwischen zwei Starts einer Anwendung nicht vonnöten sind.

Abbildung 3 veranschaulicht beispielhaft den Applikationsordner des „h3lix“ Jailbreaks.

```
iPhone:/private/var/containers/Bundle/Application/4E78543F-F5D1-4E1C-9903-
956578DD2248 root# ls -la
total 4
drwxr-xr-x  3   _installd _installd  136   Dec 18 11:26 .
drwxr-xr-x  31 _installd _installd  1054 Dec 18 11:26 ..
-rw-r--r--     1    root         wheel    371  Dec 18 11:26
.com.apple.mobile_container_manager.metadata.plist
drwxr-xr-x  5 _installd    _installd 1292 Dec 18 11:26 h3lix.app
```

Abbildung 3: Ausgabe eines Applikationsordners

4.4.1.2. plist

Property List Dateien [14] werden unter iOS dazu verwendet, um Einstellungen, Präferenzen und Systemdaten für System- und App Store Anwendungen zu speichern. Bei Apple Produkten sind die plist-Dateien üblich, analog zur Microsoft Windows Registry, mit der Ausnahme, dass plist-Dateien im gesamten Dateisystem verteilt sind. Diese Dateien können beliebige Daten speichern und werden oft dazu verwendet, um eingebettete plist-Daten innerhalb einer Property List Datei zu speichern.

Um Zugriff auf plist-Dateien zu erhalten, muss ein Gerät samt Jailbreak vorhanden sein. Interessante Dateien, welche sensible Informationen enthalten können, sind in Kapitel 4.4.1.1 angeführt sowie folgende:

- /private/var/mobile/Containers/Data/Application
  - GUID/Library/com.apple.Maps-com.apple.MapsSupport.history.plist: Speichert den Verlauf der gesuchten Orte auf Apple Maps. Zusätzlich Dateien in diesem Verzeichnis enthalten auch Lesezeichen sowie Standortinformationen.
  - GUID/Library/SuspendState.plist: Beinhaltet eine Liste an im Hintergrund befindlichen Registerkarten im Safari Browser. Diese Informationen bleiben auch nach dem Leeren des Browser-Cache erhalten.
  - GUID/Library/Safari/SearchDescriptions: Inkludiert Daten von Websites, die Suchfunktionen bieten, welche in die Suchfunktion von Mobile Safari integriert werden können. Dies deutet oft auf andere Websites hin, die vom Nutzer außerhalb der gängigen Suchmaschinen von Google, Bing, etc. besucht werden.
  - GUID/Library/Preferences/com.apple.mobilemail.plist: Konfigurationsinformationen für Apple Mail einschließlich Protokolleinstellungen, beispielsweise Internet Message Access Protocol (IMAP) und Post Office Protocol (POP3), sowie Servernamen.
- /private/var/mobile/Library
  - Mail/<E-Mail Adresse>/.mboxCache.plist: Eine gecachte Liste aller serverseitigen Ordner, welche für den Mailaccount verwendet werden.
  - com.apple.accountsettings.plist: Eine Liste der in Apple Apps verwendeten Konten mit verschlüsselten Passwörtern.

Anwendungen von Drittanbietenden aus dem Apple App Store speichern plist-Informationen im „/private/var/mobile/Containers/Data/Application“ Verzeichnis. Obwohl Apple Apps keine Passwörter oder andere Anmeldeinformationen im Klartext speichern, ist es für Entwicklerinnen und Entwickler von Drittanbieterapplikationen üblich, sensible Anmeldeinformationen in einem unverschlüsselten Format zu speichern.

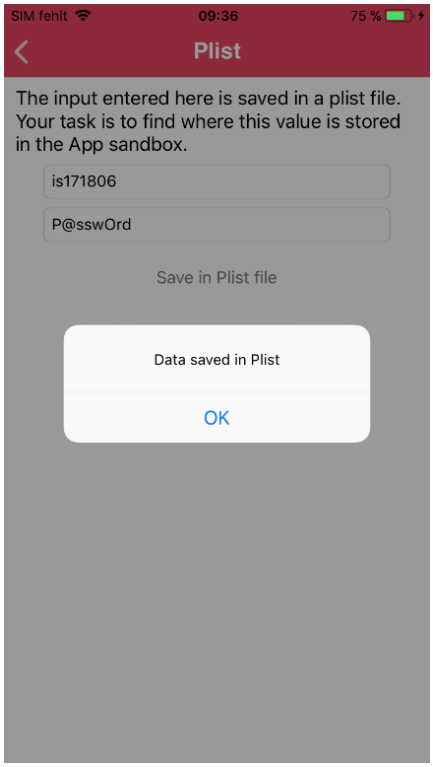

Abbildung 4: plist

In Abbildung 4 ist zu sehen, dass der Benutzername „is171806“, sowie das Passwort „P@ssw0rd“ in einer plist-Datei abgespeichert wurden. Diese Property List Datei wird unter „/private/var/mobile/Containers/Data/Application/UUID/Documents“ abgelegt. Wie in Abbildung 5 ersichtlich ist, wurden die zuvor eingetragenen Nutzerdaten im Klartext am Smartphone abgelegt. Wurde das Gerät mit einem Jailbreak modifiziert, kann diese Datei mit einfachen Mitteln ausgelesen werden.

```
iPhone:/private/var/mobile/Containers/Data/Application/ACEAEFE2-6568-454F-832E-
E550A8D66965/Documents root# cat userInfo.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
      <key>password</key>
      <string>P@sswOrd</string>
      <key>username</key>
      <string>is171806</string>
</dict>
</plist>
```

Abbildung 5: Ausgabe der Datei userInfo.plist

4.4.1.3. NSUserDefaults

Eine der häufigsten Möglichkeiten Benutzereinstellungen und -eigenschaften in einer Anwendung zu speichern, ist die Verwendung von NSUserDefaults [11]. Diese werden im Verzeichnis „/var/mobile/Containers/Data/Application/[UUID]/Library/Preferences/“ des App-Ordners abgelegt. Die Objective-C Klasse NSUserDefaults bietet eine programmgesteuerte Schnittstelle für die Interaktion mit dem Standardsystem, die es einer Anwendung ermöglicht, ihr Verhalten an die Präferenz der Benutzerin oder des Benutzers anzupassen. Die gespeicherten Informationen bleiben auch dann erhalten, wenn die Anwendung geschlossen und von einer Benutzerin oder einem Benutzer erneut gestartet wird. Eines der Beispiele für das Speichern von Informationen in NSUserDefaults ist der Anmeldestatus einer Anwenderin oder eines Anwenders. Schließt und startet eine Benutzerin oder ein Benutzer die Anwendung neu, kann mit Hilfe der Daten von NSUserDefaults entschieden werden, ob er angemeldet ist oder nicht. Je nach Anmeldestatus können dann unterschiedliche Benutzeroberflächen angezeigt werden. Viele Entwicklerinnen und Entwickler verwenden NSUserDefaults jedoch, um private Daten zu speichern. Die stellt keine empfehlenswerte Möglichkeit der Datenspeicherung dar, da diese Daten leicht aus dem Dateisystem ausgelesen werden können.

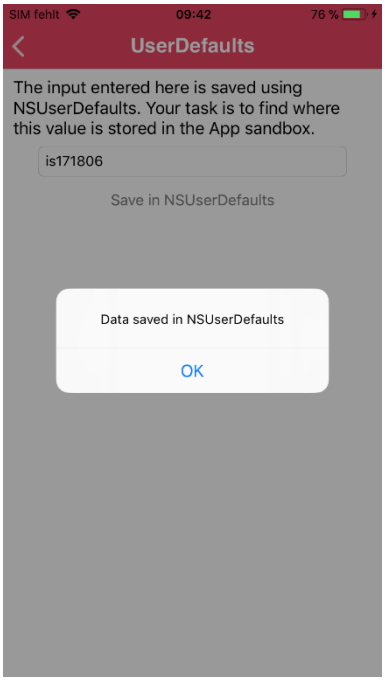

Abbildung 6: NSUserDefaults

Wie in Abbildung 6 zu sehen ist, wurde der Benutzername „is171806“ mit Hilfe der NSUserDefaults am Gerät abgespeichert. Auch der Inhalt dieser Datei ist unverschlüsselt am Gerät zu finden und kann mit einfachsten Mitteln ausgelesen werden. Die nachfolgende Abbildung 7 zeigt, dass der Benutzername im Klartext abgefragt werden kann.

```
iPhone:/Preferences root#  cat com.highaltitudehacks.DVIAswiftv2.plist
bplist00?YDemoValueXis171806
```

Abbildung 7: Ausgabe der Datei com.highaltitudehacks.DVIAswiftv2.plist

4.4.1.4. Keychain

Keychain [37],[38] ist eine verschlüsselte Datenbank, um sensible Benutzerdaten wie Benutzernamen, Passwörter, Netzwerkkennwörter sowie Authentifizierungstoken für verschiedene Anwendungen zu speichern. Diese Keychain ist als SQLite-Datenbank implementiert, auf die nur über Keychain-APIs zugegriffen werden kann. Entwicklerinnen und Entwickler verwenden die Keychain oft, um Anmeldeinformationen zu speichern, anstatt diese in NSUserDefaults oder plist-Dateien zu speichern. Grund dafür ist, dass die Applikation benutzerfreundlich sein soll und daher nicht nach jedem Start nach Benutzername und Passwort fragen muss. Obwohl die Keychain eine der sichersten Möglichkeiten ist, Informationen auf dem Gerät zu speichern, können diese Daten auf einem Gerät mit Jailbreak ausgelesen werden.

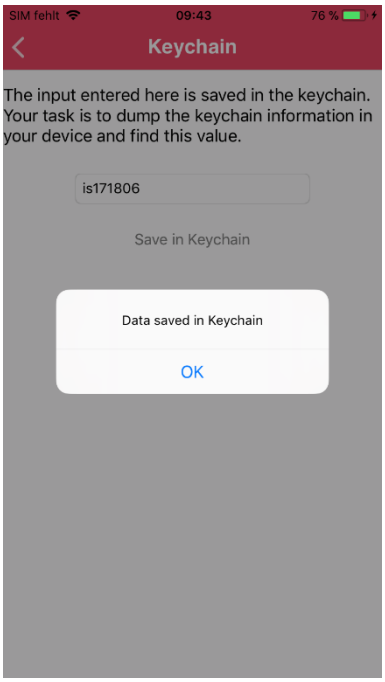

Abbildung 8: Keychain

Abbildung 8 veranschaulicht, dass wiederum der Benutzername „is171806“ am Smartphone gespeichert wurde. Diesmal kam dafür die Keychain zum Einsatz. Sowohl mit „objection“ [39] als auch mit dem Werkzeug „keychain-dumper“ [24] kann der Inhalt der Keychain angezeigt werden. In Abbildung 9 wird verdeutlicht, dass auch mit Hilfe der Keychain sensible Daten nicht genügend gesichert am iOS-Gerät abgelegt werden.

```
iPhone:/tmp root# /usr/bin/keychain_dumper
Generic Password
----------------
Service: HighAltitudeHacks.com.DamnVulnerableIOSApp
Account: keychainValue
Entitlement Group: 9UBS947V73.HighAltitudeHacks.com.DamnVulerabnleIOSApp
Label: (null)
Generic Field: (null)
Keychain Data: is171806
```

Abbildung 9: Ausgabe keychain_dumper

4.4.1.5. CoreData

CoreData [40] ist ein Framework, mit dem die Objekte der Modellschicht in einer Anwendung verwaltet werden können. Es bietet allgemeine und automatisierte Lösungen für häufige Aufgaben im Zusammenhang mit dem Objektlebenszyklus und dem Objektgrafik-Management, einschließlich der Persistenz. Für Entwicklerinnen und Entwickler ist es wichtig zu beachten, dass die, über CoreData [41, S. 5] gespeicherten Daten im Klartext ausgelesen werden können, sobald das iOS Gerät entsperrt wurde.

4.4.1.6. Realm

Eine Realm Datenbank [42] kann als Alternative zu SQLite und CoreData verwendet werden. Die Realm Plattform besteht dabei aus zwei Hauptkomponenten, der Realm Datenbank und dem Realm Object Server. Diese beiden Komponenten arbeiten zusammen um Daten automatisch zu synchronisieren, was eine Vielzahl von Anwendungsfällen ermöglicht, die von Offline-Applikationen bis hin zu komplexen Backend-Integrationen reichen. Kommt eine Realm Datenbank in einer iOS-Anwendung zur Verwendung, wird diese unter „/private/var/mobile/Containers/Data/Application/UUID/Documents“ abgelegt.

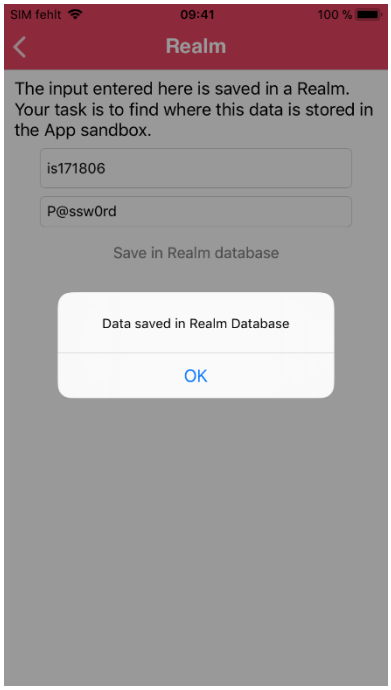

Abbildung 10: Realm

Abbildung 10 verdeutlicht, dass ein Benutzername sowie ein Passwort in einer Realm Datenbank am Smartphone gespeichert wurden. Mit Hilfe des Realm Browsers [25] kann die zuvor vom Gerät heruntergeladene Datenbank inspiziert werden. Abbildung 11 zeigt die Ausgabe der Realm Datenbank im Realm Browser. Dabei ist zu sehen, dass auch hier sowohl Name als auch Passwort im Klartext abgelegt werden.

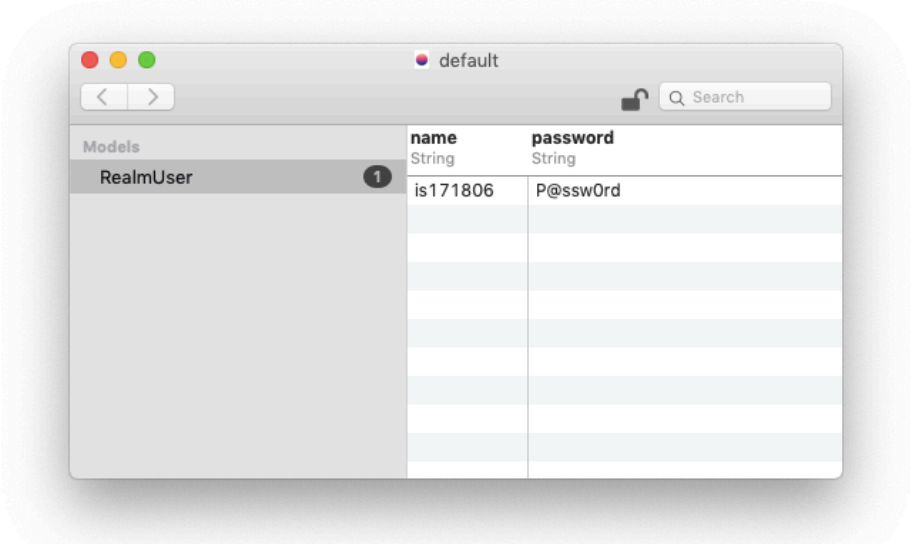

Abbildung 11: Realm Browser

4.4.2. Jailbreak Detection

Viele Entwicklerinnen und Entwickler statten ihre Applikation mit einer oder mehreren Jailbreak-Erkennungsfunktionen aus. Diese Funktionen werden verwendet, um zu überprüfen, ob die Anwendung auf einem iOS-Gerät mit aktivem Jailbreak ausgeführt wird. Angreifende können verschiedene Werkzeuge, wie zum Beispiel GDB, Frida aber auch Cycript verwenden, um Analysen zur Laufzeit durchzuführen und sensible Daten aus einer Applikation zu entwenden. In Kapitel 4.4.9.2 wird gezeigt, wie eine Applikation auf einem Gerät ohne aktiven Jailbreak mit Frida manipuliert werden kann. Eine zusätzliche Sicherheitsebene ist die sogenannte Jailbreak-Detection. Sollte ein Jailbreak erkannt werden, kann die Ausführung der Anwendung unterbunden oder einige Funktionen blockiert werden.

Folgende Techniken [11] zur Erkennung eines Jailbreak gibt es:

- Verzeichnisberechtigungen: Im Gegensatz zu unveränderten iOS-Geräten lassen Geräte mit aktiven Jailbreak vollständige Dateisystemberechtigungen zu. Aus diesem Grund kann eine Methode zur Jailbreak-Erkennung sein, dass die UNIX Dateiberechtigungen bestimmter Dateien und Verzeichnisse mit Hilfe von NSFileManager-APIs oder Low-Level C-Funktionen wie statfs() überprüft werden. Werden exzessive Berechtigungen festgestellt, kann davon ausgegangen werden, dass das Smartphone einen Jailbreak besitzt.
- Vorhandensein von Verzeichnissen und Dateien: Eine der einfachsten und beliebtesten Methoden zur Erkennung, ob ein aktiver Jailbreak vorhanden ist, ist der Zugriff auf gewisse Verzeichnisse, beziehungsweise Dateien. Die Applikation versucht dabei auf bestimmte private Verzeichnisse oder Dateien, wie zum Beispiel „/usr/bin/syslogd“ zuzugreifen. Ein erfolgreicher Zugriff auf diese Verzeichnisse bestätigt, dass dieses Gerät einen Jailbreak besitzt.
- Process forking: Applikationen haben normalerweise nicht die Berechtigungen fork(), popen() oder andere ähnliche Low-Level Systemaufrufe zur Erstellung von Kindprozessen zu verwenden. Auf einem Gerät mit Jailbreak werden diese Aufrufe jedoch erfolgreich ausgeführt. Durch die Überprüfung der zurückgegebenen Prozess ID (pid) eines fork()-Aufrufes kann auf einen aktiven Jailbreak geschlossen werden.
- system() [43]: Der Aufruf der system() Funktion mit einem NULL Argument gibt auf einem Gerät ohne Jailbreak den Wert 0 zurück, da diese Funktion die Existenz von "/bin/sh" überprüft. Wenn der

Kommandozeileninterpreter auf einem Gerät mit aktiven Jailbreak vorhanden ist, liefert der idente Aufruf den Rückgabewert 1.

Eine Möglichkeit, um die Jailbreak Erkennung einer Applikation auszuhebeln, ist Frida. Frida ist ein dynamisches Framework. Es ermöglicht einer Penetration Testerin oder einem Penetration Tester, Anwendungen, sei es unter Windows, macOS, GNU/Linux, iOS oder Android, zur Laufzeit zu verändern. Frida wird in Kapitel 4.3.1.1 näher behandelt. In Abbildung 12 ist zu sehen, dass ein Jailbreak erkannt wurde.

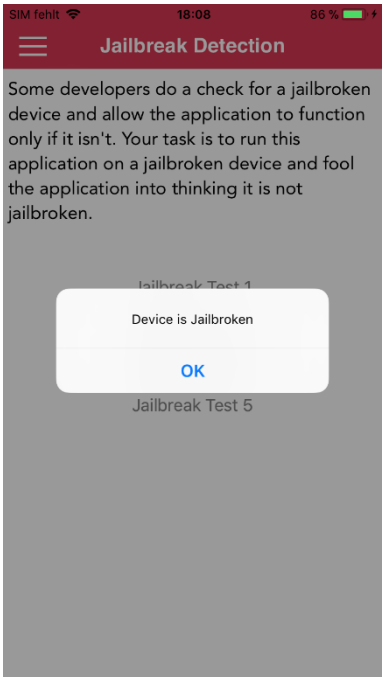

Abbildung 12: Jailbreak Erkennung

Um die Erkennung des Jailbreaks zu unterbinden, können alle vorhandenen Klassennamen der Applikation mit folgenden JavaScript Skript (Abbildung 13) ausgegeben werden.

```
/**
  * @desc   Ist eine Objective-C-Laufzeit geladen, dann
  *         werden alle Klassennamen des Applikationsprozesses ausgegeben.
  * @output Ausgabe aller Klassennamen
*/
if (ObjC.available) {
      for (var className in ObjC.classes) {
            if (ObjC.classes.hasOwnProperty(className)) {
                  console.log(className);
            }
      }
}
```

Abbildung 13: Quellcode von dump_classes.js

Im folgenden Textabschnitt (Abbildung 14) wird deutlich, dass eine Klasse namens „JailbreakDetection“ in der Applikation vorhanden ist. Daraus kann der Schluss gezogen werden, dass diese Klasse für die Erkennung eines aktiven Jailbreaks verantwortlich ist. Wird Frida mit dem Übergabeparameter „-U“ aufgerufen, bedeutet dies, dass das Framework über die USB-Schnittstelle nach verfügbaren Geräten sucht. Mit dem Parameter „-l“ wird ein Skript (siehe Abbildung 13) geladen, und zur Laufzeit in den „DVIA-v2“ Prozess injiziert.

```
Gassnerf:Diplomarbeit gassnerf$ frida -U -l dump_classes.js DVIA-v2 | grep -i
Jailbreak
JailbreakDetection
DVIA_v2.JailbreakDetectionViewController
```

Abbildung 14: Ausgabe von dump_classes.js

Um die Namen der Methoden der vorhin erwähnten Klasse zu erhalten, kann folgendes Skript (Abbildung 15) verwendet werden.

```
/**
  * @desc   Ist eine Objective-C-Laufzeit geladen, dann werden
  *               alle Methodennamen der Klasse JailbreakDetection ausgegeben.
  *
  * @output Ausgabe aller Methodennamen einer Klasse (JailbreakDetection).
*/
if (ObjC.available) {
    try {
        var className = "JailbreakDetection";
        var methods = eval('ObjC.classes.' + className + '.$methods');
        for (var i = 0; i < methods.length; i++) {
            try {
                  console.log("[-] "+methods[i]);
            }
            catch(err) {
                  console.log("[!] Error: " + err.message);
            }
        }
    } catch(err) {
      console.log("[!] Error: " + err.message);
   }
}
```

Abbildung 15: Quellcode von dump_methods.js

Abbildung 16 zeigt, dass in der Klasse „JailbreakDetection“ eine Methode namens „isJailbroken“ existiert.

```
gassnerf:Diplomarbeit gassnerf$ frida -U -l dump_methods.js DVIA-v2 | grep -i
"jailbreak\|jailbroken"
[-] + isJailbroken
```

Abbildung 16: Ausgabe von dump_methods.js

Die Methode „isJailbroken“ gibt mittels Rückgabewert den Status des iOS Gerätes zurück. Abhängig davon, ob ein Jailbreak auf diesem Smartphone aktiv ist oder nicht, wird „0x1“ für Wahr oder „0x0“ für Falsch zurückgegeben. Um den Rückgabewert dieser Methode sichtbar zu machen, kann folgender Code verwendet werden (Abbildung 17).

```
/**
  * @desc   Ist eine Objective-C-Laufzeit geladen, dann wird in der Klasse
  *         JailbreakDetection nach der Methode isJailbroken gesucht.
  *         Wurde die Methode gefunden, dann wird ein Interceptor
  *         an die Methode angehängt. Erfolgt nun ein Aufruf der Methode,
  *         so wird beim Verlassen der Methode der Rückgabewert ausgegeben.
  *
  * @output Ausgabe des Rückgabewertes der Methode isJailbroken.
*/
if (ObjC.available) {
      try {
            var _class = "JailbreakDetection";
            var _func = "isJailbroken";
            var _hook = eval('ObjC.classes.' + _class + '["' + _func + '"]');

            Interceptor.attach(_hook.implementation, {
                  onLeave: function(value) {
                        console.log("[*] Class Name: " + _class);
                        console.log("[*] Method Name: " + _func);
                        console.log("[-] Type of return value: " + typeof value);
                        console.log("[-] Return value: " + value);
                  }
            });
      }
      catch(err) {
            console.log("[!] Error: " + err.message);
      }
}
```

Abbildung 17: Quellcode von get_return_value.js

Da das verwendete Gerät einen Jailbreak besitzt, gibt die Methode den Wert „0x1“ zurück, was wiederum der booleschen Variable „Wahr“ entspricht (siehe Abbildung 18).

```
gassnerf:Diplomarbeit gassnerf$ frida -U -l get_return_value.js DVIA-v2

     ____
    /  _  |   Frida 12.2.22 - A world-class dynamic instrumentation toolkit
   |  (_|  |
    >  _  |   Commands:
   /_ /  |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/

[iPhone::DVIA-v2]-> [*] Class Name: JailbreakDetection
[*] Method Name: isJailbroken
[-] Type of return value: object
[-] Return Value: 0x1
```

Abbildung 18: Ausgabe von get_return_value.js

Mit dem folgenden Beispiel (Abbildung 19) wird der Rückgabewert während der Laufzeit verändert und die Erkennung des Jailbreaks umgangen.

```
/**
  * @desc   Ist eine Objective-C-Laufzeit geladen, dann wird in der Klasse
  *         JailbreakDetection nach der Methode isJailbroken gesucht.
  *         Wurde die Methode gefunden, dann wird ein Interceptor an die
  *         Methode angehängt. Erfolgt nun ein Aufruf der Methode,
  *         so wird beim Verlassen der Methode der Rückgabewert
  *         der Methode auf "0x0" gesetzt. "0x0" entspricht dem booleschen
  *         Wert "Falsch".
  *
  * @output Manipulation des Rückgabewertes der Methode isJailbroken
  *         von "0x1" auf "0x0".
*/
if (ObjC.available) {
      try {
            var _class = "JailbreakDetection";
            var _func = "isJailbroken";
            var _hook = eval('ObjC.classes.' + _class + '["' + _func + '"]');

            Interceptor.attach(_hook.implementation, {
                  onLeave: function(old_value) {
                  console.log("[*] Class Name: " + _class);
                  console.log("[*] Method Name: " + _func);
                  console.log("[+] Type of return value: " + typeof old_value);
                  console.log("[+] Original return value: " + old_value);
```

```
                new_value = ptr("0x0")
                old_value.replace(new_value)
                console.log("[+] New return value: " + new_value)
                }
            });
        }
        catch(err) {
                console.log("[!] Error: " + err.message);
        }
}
```

Abbildung 19: Quellcode von jailbreak_detection.js

Nach Ausführung des Skripts wurde der Rückgabewert der Methode zur Laufzeit auf „Falsch“, also auf „0x0“, geändert (Abbildung 20).

```
gassnerf:Diplomarbeit gassnerf$ frida -U -l scripts/jailbreak_detection.js DVIA-
v2

     ____
    /  _   |   Frida 12.2.22 - A world-class dynamic instrumentation toolkit
   |  (_|  |
    >  _   |   Commands:
   /_ /  |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/

[iPhone::DVIA-v2]-> [*] Class Name: JailbreakDetection
[*] Method Name: isJailbroken
[+] Type of return value: object
[+] Original return value: 0x1
[+] New return value: 0x0
```

Abbildung 20: Ausgabe von jailbreak_detection.js

Abbildung 21 zeigt, dass mit Hilfe von Frida die Erkennung eines Jailbreaks durch eine Applikation unterbunden werden kann.

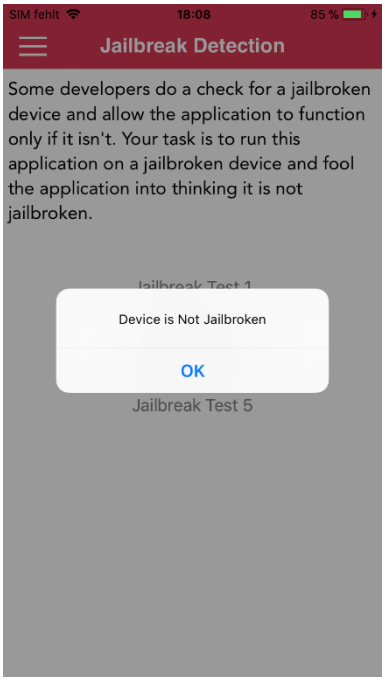

Abbildung 21: Umgangene Jailbreak Erkennung

Jede Applikation sollte Methoden zur Erkennung von Jailbreaks implementiert haben, wenn diese mit sensiblen Informationen arbeitet. Jedoch kann eine versierte Angreifferin oder ein versierter Angreifer jegliche Art der Erkennung von Jailbreaks mit Hilfe von Manipulationen zur Laufzeit umgehen (siehe Kapitel 4.4.3). Das Hauptproblem liegt aber hierbei bei Apple, da in den iOS Versionen immer wieder Schwachstellen gefunden werden, die einen Jailbreak ermöglichen.

4.4.3. Runtime Manipulation

Runtime Manipulation gestattet es einer Penetration Testerin oder einem Penetration Tester, Auskunft über eine Klasse, wie zum Beispiel Methoden oder Instanzvariablen, zu erhalten. Diese Informationen können in weiterer Folge dazu verwendet werden, um die gefundenen Methoden oder Variablen zur Laufzeit zu verändern. Dies kann dazu führen, dass etwaige Implementationen, welche zur Sicherheit der Applikation beitragen sollen, umgangen werden können. Wie in Kapitel 4.4.2 exemplarisch gezeigt, kann die Erkennung des Jailbreaks zur Laufzeit umgangen werden.

In diesem Kapitel wird gezeigt, wie die Login-Funktion einer Applikation mit Hilfe von „objection“ [39] manipuliert werden kann. Zuerst wird versucht sich mit fehlerhaften Login-Daten anzumelden, was richtigerweise nicht gelingt. Nach Manipulation der Login Methode zur Laufzeit kann mit denselben falschen Login-Daten ein erfolgreicher Login durchgeführt werden. Abbildung 22 zeigt den fehlerhaften Login-Versuch.

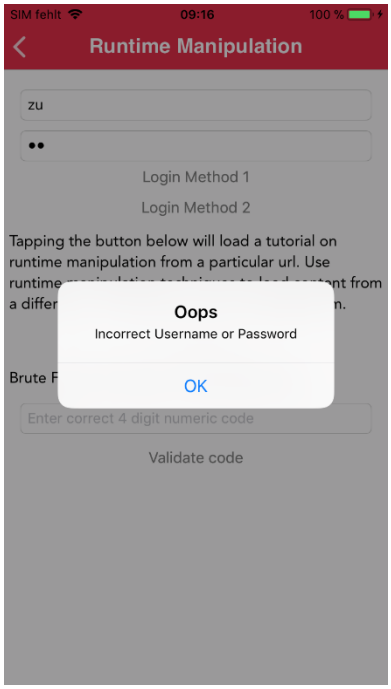

Abbildung 22: Falsche Login Daten

Mit dem Werkzeug „class-dump“ [31] können Objective-C Header aus Mach-O Dateien generiert werden. Mach-O ist das native ausführbare Format von Binärdateien in OS X, sowie iOS. Wie Abbildung 23 zeigt, gibt es eine Methode namens „LoginValidate“ mit einer booleschen Variable „isLoginValidated“.

```
./class-dump ../DVIA-v2-swift/DVIA-v2
//
//     Generated by class-dump 3.5 (64 bit) (Debug version compiled Jun  9 2015
22:53:21).
//
//     class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2014 by Steve
Nygard.
//
@interface LoginValidate : NSObject
{
}

+ (void)validateCode:(long long)arg1 viewController:(id)arg2;
+ (_Bool)isLoginValidated;

@end
```

Abbildung 23: Ausgabe von class_dump

In weiterer Folge wird der Rückgabewert dieser Methode ausgegeben, dazu wird folgendes Skript verwendet (Abbildung 24).

```
gassnerf:scripts gassnerf$ frida -U -l get_return_value.js DVIA-v2

     ____
    /  _  |   Frida 12.2.22 - A world-class dynamic instrumentation toolkit
   |  (_|  |
    >  _  |   Commands:
   /_ /  |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/

[iPhone::DVIA-v2]-> [*] Class Name: LoginValidate
[*] Method Name: isLoginValidated
[-] Type of return value: object
[-] Return value: 0x0
```

Abbildung 24: Ausgabe von get_return_value.js

Der Rückgabewert der Methode ist „0x0“, das heißt der Versuch des Logins war nicht erfolgreich. Im nachfolgenden Skript (Abbildung 25) wird dieser Wert zur Laufzeit auf „0x1“ gesetzt, was „Wahr“ entspricht.

```
/**
  * @desc   Ist eine Objective-C-Laufzeit geladen, dann wird in der Klasse
  *         LoginValidate nach der Methode isLoginValidated gesucht.
  *         Wurde die Methode gefunden, dann wird ein Interceptor an die
  *         Methode angehängt. Erfolgt nun ein Aufruf der Methode,
  *         so wird beim Verlassen der Methode der Rückgabewert
  *         der Methode auf "0x1" gesetzt. "0x1" entspricht dem booleschen
  *         Wert "Wahr".
  *
  * @output Manipulation des Rückgabewertes der Methode isJailbroken
  *         von "0x0" auf "0x1".
*/
if (ObjC.available) {

      try {
            var _class = "LoginValidate";
            var _func = "isLoginValidated";
            var _hook = eval('ObjC.classes.' + _class + '["' + _func + '"]');

            Interceptor.attach(_hook.implementation, {
                  onLeave: function(old_value) {
                    console.log("[*] Class Name: " + _class);
                    console.log("[*] Method Name: " + _func);
                    console.log("[+] Type of return value: " + typeof old_value);
```

```
                    console.log("[+] Original return value: " + old_value);
                    new_value = ptr("0x1")
                    old_value.replace(new_value)
                    console.log("[+] New return value: " + new_value)
                  }
              });
            }
            catch(err) {
                  console.log("[!] Error: " + err.message);
            }
}
```

Abbildung 25: Quellcode von bypassLogin.js

Mit der Manipulation (Abbildung 26) wird die Logik der Login-Methode umgangen und der Wert der Instanzvariable „isLoginValidated“ auf „Wahr“ gesetzt, obwohl falsche Login-Daten verwendet wurden.

```
gassnerf:scripts gassnerf$ frida -U -l bypassLogin.js DVIA-v2

     ____
    /  _  |   Frida 12.2.22 - A world-class dynamic instrumentation toolkit
   |  (_| |
    >  _  |   Commands:
   /_ /  |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/

[iPhone::DVIA-v2]-> [*] Class Name: LoginValidate
[*] Method Name: isLoginValidated
[+] Type of return value: object
[+] Original return value: 0x0
[+] New return value: 0x1
```

Abbildung 26: Ausgabe von bypassLogin.js

Abbildung 27 zeigt, dass mit Hilfe von Frida und der Manipulation zur Laufzeit der Login mit denselben falschen Daten wie in Abbildung 22 diesmal erfolgreich ist.

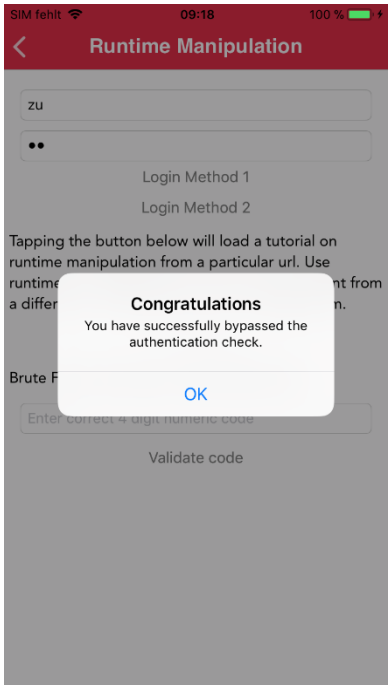

Abbildung 27: Umgehung Login Check

Die Hauptursache, dass eine Manipulation zur Lauzeit möglich ist, sind Jailbreaks, die für die jeweilige iOS Version zur Verfügung stehen. Gäbe es keine Jailbreaks, dann kann auch eine „Runtime Manipulation“ ausgeschlossen werden.

4.4.4. Binary Protection

Apple hat das Sicherheitsmodell [11], [12] von iOS seit der Version 1.0 drastisch verbessert. In den früheren Versionen wurden zum Beispiel alle Anwendungen als Root-Benutzer ausgeführt. Deshalb wurden mehrere Schritte unternommen, um den Missbrauch von Software auf iOS-Plattformen zu verhindern. Sowohl zur Verhinderung der Nutzung ihrer Software auf anderen nicht zugelassenen Hardware-Plattformen, als auch zum Schutz der Endbenutzerin oder des Endbenutzers vor Angriffen. Obwohl Xcode standardmäßig alle binären Sicherheitsfunktionen aktiviert, kann es von Vorteil sein, diese vor allem bei alten Anwendungen zu überprüfen oder die Fehlkonfigurationen von Kompilierungsoptionen durch eine Entwicklerin oder einen Entwickler zu überprüfen. Einige dieser Schutzmaßnahmen werden in weiterer Folge dargestellt.

4.4.4.1. Automatic Reference Counting

Automatic Reference Counting (ARC) [44] wird dazu verwendet, um den Speicherverbrauch einer App zu verfolgen und zu verwalten. Eine Entwicklerin beziehungsweise ein Entwickler muss sich daher nicht selbst um die Speicherverwaltung kümmern. ARC gibt automatisch den von Klasseninstanzen verwendeten Speicher frei, wenn diese Instanzen nicht mehr benötigt werden. Mit dem Werkzeug „otool“ (Abbildung 28) kann nachgesehen werden, ob das Automatic Reference Counting bei einer Anwendung zum Einsatz kommt.

```
gassnerf:DVIA-v2.app gassnerf$ otool -Iv DVIA-v2 | grep release
[TRUNCATED]
0x00000001003a59f0 70736 _objc_autorelease
0x00000001003a59f8 70737 _objc_autoreleasePoolPop
```

```
0x00000001003a5a00 70738 _objc_autoreleasePoolPush
0x00000001003a5a08 70739 _objc_autoreleaseReturnValue
0x00000001003a5a98 70757 _objc_release
0x00000001003a5aa8 70759 _objc_retainAutorelease
0x00000001003a5ab0 70760 _objc_retainAutoreleaseReturnValue
0x00000001003a5ab8 70761 _objc_retainAutoreleasedReturnValue
0x00000001003a5b10 70772 _objc_unsafeClaimAutoreleasedReturnValue
```

Abbildung 28: Ausgabe von otool -Iv DVIA-v2 | grep release

4.4.4.2. Stack Canary

Das Stack Canary wird dazu verwendet, um Buffer Overflow Angriffe zu verhindern. Dazu wird eine kleine Zufallszahl direkt vor dem Return Pointer eingefügt. Ein Buffer Overflow überschreibt oft einen Speicherbereich, um den Return Pointer zu überschreiben und die Kontrolle über einen Prozess zu erlangen. Ist das Stack Canary aktiv und gesetzt, dann wird auch bei diesem Angriff das Canary überschrieben. Bevor der Return Pointer vom Stack verwendet wird, wird der Wert des Canary überprüft, ob dieser verändert wurde. Wird eine Veränderung erkannt, dann bedeutet das, dass der Stack unerlaubt überschrieben wurde. In Abbildung 29 ist ersichtlich, dass die Applikation diese Sicherheitsmaßnahmen verwendet.

```
gassnerf:DVIA-v2.app gassnerf$ otool -Iv DVIA-v2 | grep stack
0x0000000100329b80 70498 ___stack_chk_fail
0x00000001003a4458 70499 ___stack_chk_guard
0x00000001003a55b8 70498 ___stack_chk_fail
```

Abbildung 29: Ausgabe von otool -Iv DVIA-v2 | grep stack

4.4.4.3. Position Independent Executable

Position Independent Executable (PIE) bedeutet, dass Anwendungen beim Ausführen an einer zufälligen Speicheradresse geladen werden. Sollte es einer Angreiferin oder einem Angreifer gelingen Schadcode über eine Applikation in den Speicher einzufügen, kann dieser nicht ausgeführt werden, da nicht bekannt ist, an welcher Adresse sich der Schadcode befindet. Auch hier kann wieder das Werkzeug „otool“ verwendet werden, um zu verifizieren, ob die Applikation als Position Independent Executable kompiliert wurde. Die folgende Ausgabe (Abbildung 30) zeigt, dass unter „flags“ das Flag „PIE“ zu finden ist.

```
gassnerf:DVIA-v2.app gassnerf$ otool -hv DVIA-v2
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
MH_MAGIC_64   ARM64        ALL  0x00     EXECUTE    66       7184   NOUNDEFS
DYLDLINK TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE
```

Abbildung 30: Ausgabe von otool -hv DVIA-v2

4.4.4.4. Third Party Libraries

iOS-Anwendungen greifen häufig auf Bibliotheken von Drittanbietenden zurück. Auch Entwicklerinnen und Entwickler greifen gerne auf diese Bibliotheken zu, da der Entwicklungsprozess dadurch beschleunigt und vereinfacht wird. Jedoch können diese Bibliotheken unerwünschte Nebenwirkungen haben:

- eine Bibliothek kann eine Schwachstelle enthalten, welche die Anwendung verwundbar macht,
- eine Bibliothek kann eine Lizenz wie die LGPL2.1 verwenden, die vom Anwendungsautor verlangt, dass er denjenigen, die die Anwendung verwenden, Zugang zum Quellcode und Einblick in ihre Quellen gewährt.

Mit Hilfe von „otool“ können alle, von der Applikation verwendeten, Bibliotheken angezeigt werden. Mittels einer Suchmaschine könnten dann Schwachstellen aufgedeckt werden, die in späterer Folge von einer Angreiferin oder einem Angreifer ausgenutzt werden. In Abbildung 31 wird deutlich, welche Bibliotheken von der „Damn Vulnerable iOS Application“ verwendet werden.

```
gassnerf:DVIA-v2-swift gassnerf$ otool -L DVIA-v2
DVIA-v2:
	/usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version 400.9.1)
	/usr/lib/libsqlite3.dylib (compatibility version 9.0.0, current version 274.6.0)
	/usr/lib/libz.1.dylib (compatibility version 1.0.0, current version 1.2.11)
	@rpath/Bolts.framework/Bolts (compatibility version 1.0.0, current version 1.0.0)
	@rpath/Parse.framework/Parse (compatibility version 1.0.0, current version 1.0.0)
	@rpath/Realm.framework/Realm (compatibility version 1.0.0, current version 1.0.0)
	@rpath/RealmSwift.framework/RealmSwift (compatibility version 1.0.0, current version
1.0.0)
	[TRUNCATED]
```

Abbildung 31: Ausgabe von otool -L DVIA-v2

4.4.5. Touch ID Bypass

Mit dem iPhone 5s führte Apple Touch ID ein. Touch ID ist ein in die Home Taste integrierter Fingerabdruck-scanner. Mit dem Scanner können Benutzerinnen oder Benutzer ihr iOS-Gerät sowie die iOS-Keychain entsperren. Des Weiteren können mit Hilfe von Touch ID Einkäufe über den App Store, iTunes oder auch In-App getätigt werden. Wird ein Fingerabdruck einer Anwenderin oder eines Anwenders für die spätere Verwendung auf dem Smartphone registriert, erstellt der Fingerabdrucksensor einen Abgleich des Fingers. Dieser Abgleich wird in weiterer Folge mit einem unbekannten Algorithmus gehasht und dann in einem schreibgeschützten Speicherbereich auf dem Prozessor abgelegt. Dieser Speicherbereich ist für jede Software auf dem iOS-Gerät unzugänglich und kann nur dann für die anschließende Hash-Validierung verwendet werden, wenn neue Fingerabdrücke übermittelt werden. Mit Hilfe der Klasse „LAContext“ kann eine Anwendung die Authentifizierungsdialoge sichtbar machen und die Authentifizierung durchführen. Abhängig vom Erfolg oder Misserfolg der Authentifizierung selbst wird ein Rückgabewert zurückgegeben, der einen booleschen Wert enthält, der angibt, ob der Authentifizierungsvorgang erfolgreich war oder nicht. Unter Zuhilfenahme von „objection“ [39] kann der Rückgabewert bei dem Authentifizierungsvorgang manipuliert werden. Dazu wird ein Hook ausgeführt, welcher auf die Aufrufe des Selektors „-[LAContext evaluatePolicy:localizedReason:reply:]“ wartet. Wird die Methode „evaluatePolicy“ aufgerufen, ersetzt dieser

Hook den booleschen Wert durch ein „Wahr“. Abbildung 32 zeigt, dass Vorgang zur Authentifizierung mittels Touch ID nicht erfolgreich war.

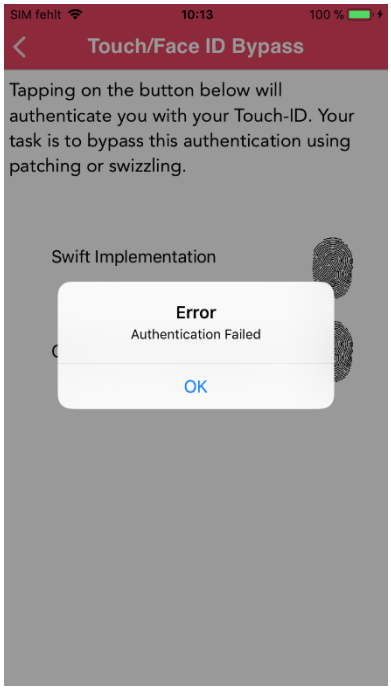

Abbildung 32: Touch ID Anmeldung nicht erfolgreich

Das Werkzeug „frida-ps“ kann dazu verwendet werden, um alle Prozesse, welche auf dem iOS-Gerät gerade ausgeführt werden, aufzulisten. Die Ausgabe des Befehls (Abbildung 33) zeigt sowohl eine Prozess Identifikationsnummer, als auch den Namen des Prozesses.

```
gassnerf:scripts gassnerf$ frida-ps -U | grep DVIA
2153  DVIA-v2
```

Abbildung 33: Ausgabe von frida-ps -U | grep DVIA

Ist der Name des Prozesses, welcher mit „objection“ zur Laufzeit manipuliert werden soll, bekannt, kann „objection“ damit gestartet werden. Mit dem Befehl „ios ui touchid_bypass“ kann, wie in Kapitel 4.4.5 zu Beginn erwähnt wurde, die nötigen Methoden gehookt werden. Wie Abbildung 34 zeigt, wurde der Rückgabewert des Authentifizierungsvorganges durch Touch ID von „Falsch“ auf „Wahr“ geändert.

```
gassnerf:Tools gassnerf$ objection --gadget "DVIA-v2" explore


     _     _       _   _
 ___| |_  |_|___ ___| |_|_|___ ___
| . | . | | |  _|  _|  _| | . |   |
|___|___| |_|___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.4.3
[tab] for command suggestions
....highaltitudehacks.DVIAswiftv2 on (iPhone: 11.2.5) [usb] # ios ui
touchid_bypass
```

```
Job: ac19c518-1e86-4f7d-9655-7159760ec727 - Starting
[7159760ec727] [touchid-bypass] Hooked -[LAContext
evaluatePolicy:localizedReason:reply:]
Job: ac19c518-1e86-4f7d-9655-7159760ec727 - Started
....highaltitudehacks.DVIAswiftv2 on (iPhone: 11.2.5) [usb] # [7159760ec727]
[touchid-bypass] Localized Reason for auth requirement: Please authenticate
yourself
[7159760ec727] [touchid-bypass] OS authentication success response: false
[7159760ec727] [touchid-bypass] Marking OS response as True instead
[7159760ec727] [touchid-bypass] TouchID bypass run complete
```

Abbildung 34: Ausgabe von objection --gadget "DVIA-v2" explore

In Abbildung 35 ist nunmehr zu sehen, dass der Touch ID Authentifizierungsprozess erfolgreich umgangen wurde.

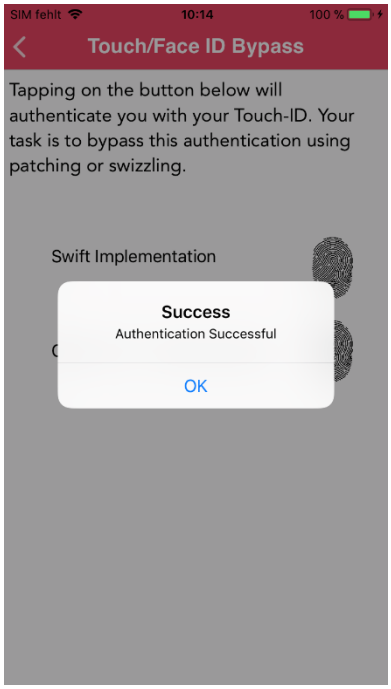

Abbildung 35: Umgehung Touch ID Anmeldung

Implementierung von Touch ID sollte mit Hilfe der Verwendung von Keychain und Access Control Lists (ACL) stattfinden. Keychains können Zugriffskontrolllisten verwenden, um Richtlinien für Zugriffs- und Authentifizierungsanforderungen festzulegen. ACLs werden innerhalb der Secure Enclave ausgewertet und nur dann an den Kernel freigegeben, wenn die spezifischen Einschränkungen erfüllt sind. Bei dieser Implementierung muss eine Validierung des Fingerabdruckes stattfinden, da sonst kein Zugriff auf die Keychain möglich ist.

4.4.6. Side Channel Data Leakage

Verarbeitet eine Anwendung sensible Daten, welche von einer Benutzerin oder einem Benutzer eingegeben werden oder aus einer anderen Quelle kommen, kann dies dazu führen, dass diese Daten an einem Ort auf unsichere Weise abgelegt werden. Diese Art der unsicheren Speicherung von sensiblen Daten kann durch andere böswillige Anwendungen auf demselben Gerät ausgenutzt werden. Können Informationen durch „Side Channel Data Leakage“ ausgelesen werden, dann handelt es sich um Implementierungsfehler seitens der Entwicklerung oder des Entwicklers. Mit welchen Methoden beziehungsweise böswillige Applikationen Angreiferinnen oder Angreifer an sensible Daten von Nutzerinnen und Nutzern oder anderen Anwendungen kommen können, wird in den nachfolgenden Kapiteln näher behandelt.

4.4.6.1. App Screenshot

Eines der Merkmale von iOS ist, dass das Betriebssystem einen Screenshot jener Anwendung macht, die durch die Benutzerin oder den Benutzer in den Hintergrund geschoben wurde. Eine Angreiferin beziehungsweise ein Angreifer mit physischem Zugriff auf das System oder eine Applikation mit bösen Absichten kann leicht auf diesen Screenshot zugreifen und die darin enthaltenen vertraulichen Informationen einsehen. Dieses Bildschirmfoto wird im anwendungsspezifischen Ordner unter „Snapshots“ abgelegt, genauer gesagt unter „/private/var/mobile/Containers/Data/Application/UUID/Library/Caches/Snapshots/“. Abbildung 36 zeigt, dass die Applikation „Damn Vulnerable iOS Application“ in den Hintergrund gestellt wurde und einen Screenshot abgelegt hat.

```
iPhone:/private/var/mobile/Containers/Data/Application/ACEAEFE2-6568-454F-832E-
E550A8D66965/Library/Caches/Snapshots/com.highaltitudehacks.DVIAswiftv2 root# ls
-la
total 56
drwxr-xr-x 4 mobile mobile   128 Nov 21 11:09 .
drwxr-xr-x 3 mobile mobile    96 Nov 15 09:35 ..
-rw-r--r-- 1 mobile mobile 54345 Nov 21 11:09 117C679E-EE71-426F-9251-
5479D739E92D@2x.ktx
drwxr-xr-x 3 mobile mobile    96 Nov 21 11:09 downscaled
```

Abbildung 36: Ausgabe eines Snapshot Ordners

Wie in Abbildung 37 zu erkennen ist, wurde ein Screenshot aufgenommen, als die Applikation in den Hintergrund gestellt wurde. Dabei ist zu sehen, dass sensible Daten auf dieser Bildschirmkopie zu erkennen sind.

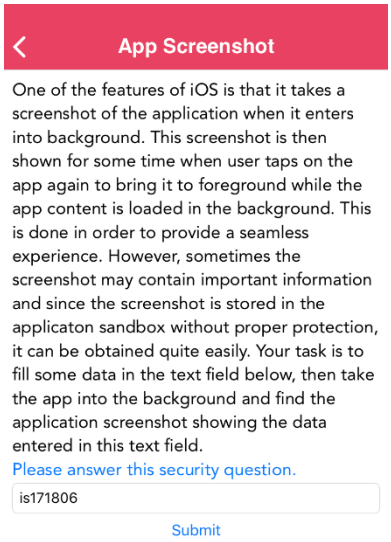

Abbildung 37: App Screenshot

Bevor eine Applikation in den Hintergrund gestellt wird, sollten sensible Informationen entfernt werden. Dabei kann die Entwicklerin oder der Entwickler auf die Methode „applicationDidEnterBackground:“ zurückgreifen um beispielsweise persönliche Daten oder Passwörter aus den Ansichten zu entfernen oder zu überdecken.

4.4.6.2. Pasteboard

Wird ein Text in iOS kopiert beziehungsweise ausgeschnitten, wird dieser im sogenannten „Pasteboard“ zwischengespeichert. Unter iOS ist das „Pasteboard“ ein alltäglicher Bestandteil aller Anwendungen. Sind Daten in diesem Zwischenspeicher vorhanden, können auch andere Anwendungen darauf zugreifen. Wird zum Beispiel von einer Benutzerin oder einem Benutzer ein Passwort kopiert um es in eine Login-Maske einzufügen, können auch andere Applikationen, aber auch Angreiferinnen oder Angreifer dieses auslesen. Abbildung 38 zeigt, dass in der Testapplikation sowohl ein Name, eine Kreditkartennummer als auch der Card Validation Value eingegeben und in weiterer Folge in den Zwischenspeicher, also das „Pasteboard“, kopiert wurden.

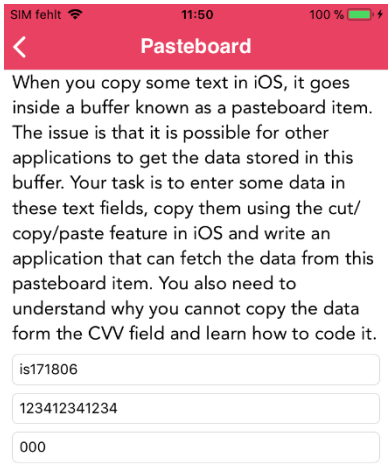

Abbildung 38: Pasteboard

Der nachfolgende Ausschnitt (Abbildung 39) zeigt, dass das „Pasteboard“ mit Hilfe von „objection“ überwacht wurde. Der Reihe nach wurden die zuvor in die Applikation eingegebenen Werte in den Zwischenspeicher geladen und erfolgreich aus dem „Pasteboard“ ausgelesen.

```
gassnerf:~ gassnerf$ objection --gadget "DVIA-v2" explore


     _     _       _   _
 ___|_|_ _|_|___ ___| |_|_|___ __
| . | . | | | -_|  _|  _| | . |  |
|___|___|_|_|___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.4.3

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
....highaltitudehacks.DVIAswiftv2 on (iPhone: 11.2.5) [usb] # ios pasteboard
monitor
Job: 0f687e9d-ad4f-4bc2-99cc-dbb3924f823c - Starting
Job: 0f687e9d-ad4f-4bc2-99cc-dbb3924f823c - Started
....highaltitudehacks.DVIAswiftv2 on (iPhone: 11.2.5) [usb] #
[dbb3924f823c] [pasteboard-monitor-string] is171806
[dbb3924f823c] [pasteboard-monitor-string] 123412341234
[dbb3924f823c] [pasteboard-monitor-string] 000
```

Abbildung 39: Ausgabe von objection --gadget "DVIA-v2" explore

Das Hauptproblem, warum Daten aus dem Zwischenspeicher ausgelesen werden können, liegt am iOS Betriebssystem. Eine Entwicklerin oder ein Entwickler kann jedoch Maßnahmen implementieren, die das „Pasteboard“ nach gewissen Funktionen leert, indem ein leerer Text in den Zwischenspeicher geladen wird.

4.4.6.3. Cookies

Safari-Browser und iOS-Anwendungen speichern die persistenten Cookies in der Datei Cookies.binarycookies. Persistente Cookies, also Cookies, welche ein zukünftiges Verfallsdatum haben, werden für jede Applikation unter „/private/var/mobile/Containers/Data/Application/UUID/Library/Cookies“ abgespeichert. Die meisten iOS-Anwendungen erstellen Session Cookies mit einem Ablaufdatum in der Zukunft, da sie Nutzerinnen oder Nutzer nicht jedes Mal zur Anmeldung auffordern wollen. Normalerweise werden diese Cookies nie invalid, es sei denn, die Benutzerin oder der Benutzer meldet sich von der Anwendung ab. Der nachfolgende Ausschnitt zeigt den Inhalt einer Cookies.binarycookies Datei. Wird diese Datei mit einem normalen Texteditor geöffnet, sind die gesetzten Werte nicht genau zu erkennen (Abbildung 40).

```
iPhone:/private/var/mobile/Containers/Data/Application/ACEAEFE2-6568-454F-832E-
E550A8D66965/Library/Cookies root# cat Cookies.binarycookies
?
Kbplist00? NSHTTPCookieAcceptPolicycks.comusername/admin123d8NWY@w|??A?.???Ahigha
ltitudehacks.compassword/dvpassword&
```

Abbildung 40: Ausgabe der Datei Cookies.binarycookies

Um den Inhalt der Cookies.binarycookies Dateien lesbar zu machen, wurde ein Python Skript, namens BinaryCookieReader.py [26], entwickelt. Der anschließende Auszug (Abbildung 41) zeigt, wie das Skript den Inhalt des Cookies lesbar darstellt.

```
gassnerf:scripts gassnerf$ python BinaryCookieReader.py ../Cookies.binarycookies
#*********************************************************************#
# BinaryCookieReader: developed by Satishb3: http://www.securitylearn.net #
#*********************************************************************#
Cookie : username=admin123; domain=highaltitudehacks.com; path=/; expires=Wed, 22
Mar 2051;
Cookie : password=dvpassword; domain=highaltitudehacks.com; path=/; expires=Wed,
22 Mar 2051;
```

Abbildung 41: Ausgabe von python BinaryCookieReader.py ../Cookies.binarycookies

Der darin gefundene Username sowie das Passwort gehören zu einer Login-Funktion in der Anwendung „Damn Vulnerable iOS Application“. Wie Abbildung 42 verdeutlicht, speichert das Gerät die Anmeldedaten in einem nicht sicheren Format.

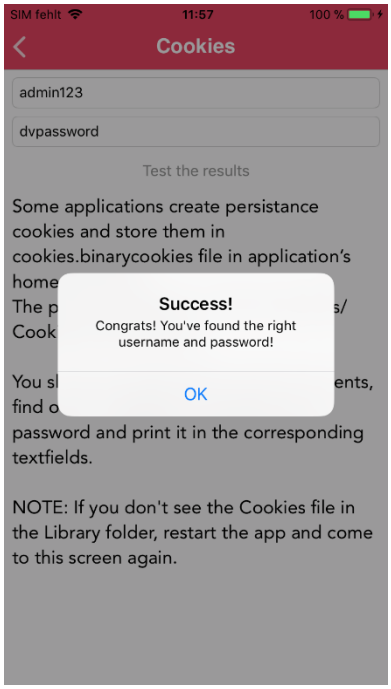

Abbildung 42: Korrekte Anmeldedaten aus dem Cookie

Eine Entwicklerin oder ein Entwickler sollten keine sensiblen Daten in der Cookies.binarycookies Datei abspeichern. Persönliche Informationen und Passwörter sollten verschlüsselt in der Keychain abgelegt werden (siehe Kapitel 4.4.1)

4.4.7. Inter Process Communication Issues

Inter Process Communication [44] bedeutet, dass eine Applikation mit Hilfe von Uniform Ressource Locator (URL) Schemata Funktionen anderer Anwendungen aufrufen können. iOS-Anwendungen und Webanwendungen, die in Safari auf jeder Plattform laufen, können diese Schemata verwenden, um sich in Systemanwendungen zu integrieren und den Nutzerinnen und Nutzern ein nahtloses Erlebnis zu bieten. Zeigt zum Beispiel eine Applikation eine Telefonnummer an, kann dazu eine entsprechende URL verwendet werden. Um die Telefon Anwendung zu starten, sollte dieser Link angetippt werden. Einige der nativen URL Schemata sind folgende:

- mailto: öffnet die Mail Applikation und kann automatisch den Empfänger, Betreff sowie den Inhalt der Mail ausfüllen
- tel: kann dazu verwendet werden, um ein Telefongespräch einzuleiten
- facetime: selbe Funktion wie „tel://“, jedoch über FaceTime
- sms: erstellt eine neue Textnachricht und trägt einen Empfänger ein

Nimmt eine Applikation Eingaben von einer Benutzerin oder einem Benutzer entgegen, ohne dass diese, bevor sie verarbeitet und angezeigt werden, einer Validation unterzogen werden, kann dies zu einem sogenannten „Client Side Injection“ Angriff führen. Die nachfolgende Ausgabe (Abbildung 43) zeigt den Source Code einer manipulierten Webseite. Diese Webseite wurde so gestaltet, dass beim Aufruf dieser die Telefon-Applikation mit einer vorausgefüllten Telefonnummer gestartet wird.

```
<!DOCTYPE html>
<html>
      <head>
            <title>iOS Penetration Test - is171806</title>
      </head>
      <body>
            <h1>iOS Penetration Test - is 171806</h1>
            <script>document.location="tel://06641234567"</script>
      </body>
</html>
```

Abbildung 43: Quelltext von index.html

Abbildung 44 zeigt den Aufruf der manipulierten Webseite sowie den Dialog zum Tätigen des Anrufes.

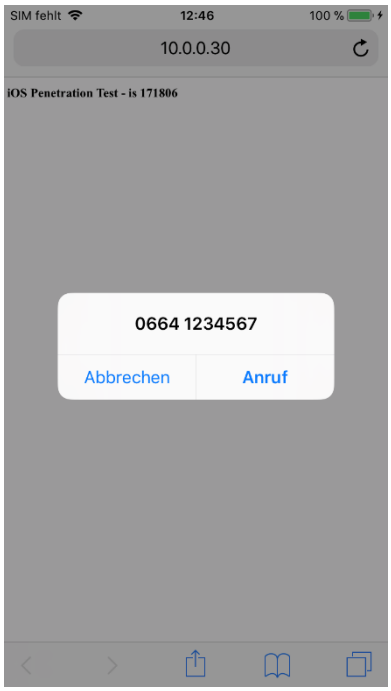

Abbildung 44: Erfolgreiche Ausnutzung von IPC

Entwicklerinnen und Entwickler können für ihre eigenen Applikationen maßgefertigte URL Schemata erstellen, die beim Aufruf eine definierte Funktion ausführen. Werden in den Funktionen sensible Daten verarbeitet, kann das mit Hilfe der in diesem Kapitel gezeigten Methode zur Offenlegung dieser Daten führen.

4.4.8. WebView Issues

UIWebViews [45] werden in Applikationen dazu verwendet, um Webinhalte einzubinden. Diese Views bauen auf dem WebKit [46] Framework auf, um Browserfunktionen, wie zum Beispiel das Folgen von Links, für Applikationen bereitzustellen. UIWebViews können Remoteinhalte darstellen und unterstützen auch die Ausführung von JavaScript. Aus diesem Grund sind iOS Anwendungen, welche dieses Framework verwenden, genauso anfällig für Cross Site Scripting (XSS) wie jede andere Webanwendung, wenn diese Benutzereingaben vor deren Verarbeitung nicht validieren. Abbildung 45 verdeutlicht, dass Cross Site Scripting nicht nur bei Webanwendungen durchgeführt werden kann.

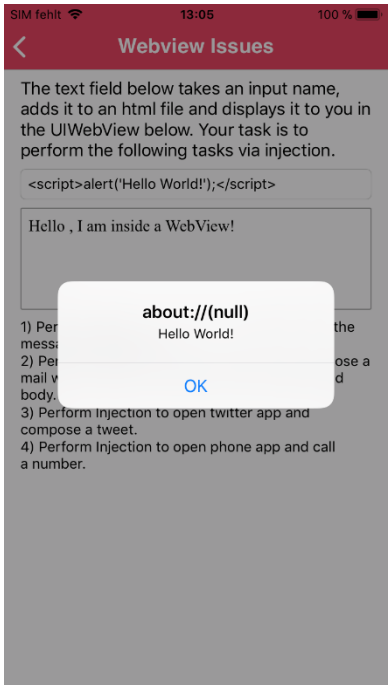

Abbildung 45: XSS in der Applikation

Wie in Kapitel 4.4.7 erläutert, können Funktionen von Applikationen mittels ULS Schemata aufgerufen werden. Ist eine iOS Anwendung anfällig für XSS, können diese Funktionen damit aufgerufen werden. Abbildung 46 zeigt, dass die Telefonanwendung mit Hilfe von JavaScript aufgerufen werden kann.

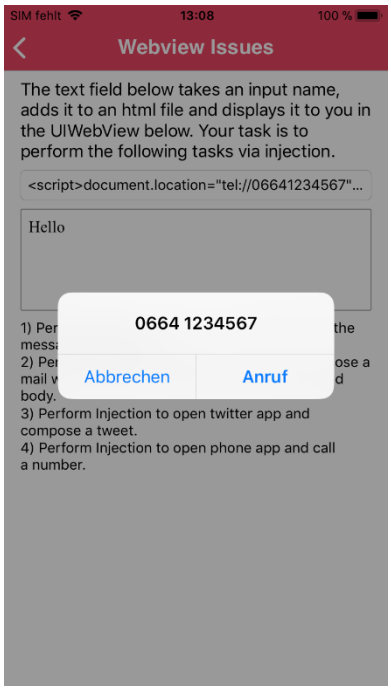

Abbildung 46: Aufruf der Telefon Applikation

Auch diese Schwachstellen in Applikationen können unter Einsatz von URL Schemata zu schwerwiegenden Folgen für eine unachtsame Benutzerin oder einen unachtsamen Benutzer führen.

4.4.9. Network Layer Security

Seit der iOS Version 9 ist die Sicherheitsfunktion App Transport Security (ATS) bei Applikationen standardmäßig aktiviert. Dies hat zur Folge, dass Anwendungen für Hypertext Transfer Protocol (HTTP) Verbindungen das sichere HTTP Secure Protokoll verwenden müssen. Dabei kommt das Protokoll Transport Layer Security (TLS) Version 1.2 zum Einsatz. Viele Anwendungen wollen sicherstellen, dass der Austausch von Daten nur mit dem angegebenen Server vonstattengeht, weshalb eine Sicherheitsmaßnahme namens „Certificate Pinning“ implementiert wird. Dabei wird das SSL Zertifikat des Zielservers innerhalb der Applikation gespeichert und bei jeder Verbindung überprüft. Gerade die Kommunikation zwischen Applikation und Backend-Server ist für Penetration Testerinnen, beziehungsweise Penetration Tester von besonderer Bedeutung. Wie das „Certificate Pinning“ einer Applikation umgangen werden kann, wird in den nachfolgenden Unterkapiteln näher betrachtet.

4.4.9.1. Web Proxy Server

Ein Web Proxy Server oder auch Intercepting Proxy kann als Man-in-the-Middle (MITM) zwischen Browser, in diesem Fall der mobilen Anwendung, und einem Webserver positioniert werden. Dadurch besteht die Möglichkeit, eingehenden und ausgehenden HTTP Datenverkehr abzufangen, zu inspizieren, zu modifizieren und gegebenenfalls auch zu verwerfen. Die zwei bekanntesten Proxies sind Burp Suite Scanner (Burp) [23] sowie OWASP Zed Attack Proxy (ZAP). In dieser Arbeit wird ausschließlich Burp Suite Scanner verwendet. Um keine Warnungen beim Aufruf von gesicherten HTTP-Verbindungen zu bekommen, muss das entsprechende Certificate Authority (CA) Zertifikat [47] von Burp am iOS-Gerät eingespielt werden. Dazu muss in der Netzwerkkommunikation [48] manuell der Proxy eingetragen werden. Danach kann mittels Safari-Browser am mobilen Gerät die URL „http://burp“ aufgerufen werden. Unter dem Link „CA Certificate“ kann das Zertifikat runtergeladen und anschließend installiert werden.

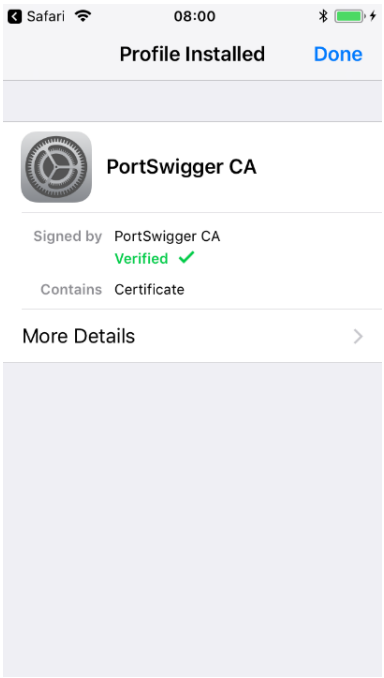

Abbildung 47: PortSwigger CA installiert

Wie in Abbildung 47 ersichtlich ist, wurde das Zertifikat der PortSwigger CA erfolgreich am iOS-Gerät installiert. Resultierend aus der Konfiguration des Intercepting Proxies und des mobilen Gerätes ist es nun

möglich, die Kommunikation zwischen der Applikation und dem Backend-Server einzusehen, zu manipulieren oder gar zu verwerfen (siehe Abbildung 56).

Durch das Inspizieren der Kommunikation können viele Informationen über die Arbeitsweise der Applikation und des dahinterliegenden Servers gewonnen werden. Dabei kann es sich um Benutzernamen, Passwörter, sensible Informationen, aber auch Application Programming Interface (API) Aufrufe handeln. Diese Informationen können in weiterer Folge dazu verwendet werden, um in der Exploitation-Phase unbefugten Zugriff auf Ressourcen zu erhalten.

4.4.9.2. Certificate Pinning

Certificate Pinning [11] ist eine Technik, um die Kommunikation zwischen Client und Server weiter abzusichern. Dabei vertraut die Applikation nur bekannten Kommunikationspartnern und die Kommunikation mit nicht vertrauenswürdigen Partnern wird ablehnt. Bei dieser Methode wird der Public-Key-Fingerprint des Backend-Servers fest in die Applikation implementiert und die App lehnt Verhandlungen mit Servern ab, wenn es zu Unstimmigkeiten kommt. Certificate Pinning macht es fast unmöglich, Man-in-the-Middle Angriffe ohne Jailbreak durchzuführen.

Um das Certificate Pinning einer Applikation auszuhebeln, gibt es mehrere Techniken:

- Installation von Software, welche das Certificate Pinning aushebelt
- Verwendung von „objection“ [39] und Frida [33]
- Verwendung von Disassembler, um die iOS App Store Package Datei zu modifizieren

Die Installation des CA Zertifikates des Intercepting Proxies ist der erste Schritt, um Transport Layer Security Fehler zu beseitigen. Die Installation ist innerhalb von iOS relativ einfach und wurde in Kapitel 4.4.9.1 näher erklärt.

Sollte es nach dem Einspielen des CA Zertifikates immer noch zu Fehlern kommen, besteht die Möglichkeit, dass der Backend-Server eine Art TLS-Certificate-Chain Validierung oder Zertifikatspinning verwendet. Um das Pinning zu umgehen, wurden Werkzeuge entwickelt, die das Leben einer Penetration Testerin oder eines Penetration Testers um ein Vielfaches vereinfachen. Diese Tools patchen bestimmte Low-Level-SSL-Funktionen innerhalb der Secure Transport API, um die standardmäßige Zertifikatsvalidierung des Systems sowie jede Art von benutzerdefinierter Zertifikatsvalidierung zu überschreiben und zu deaktivieren.

Folgende Programme können dafür verwendet werden:

- SSLKillSwitch [49]
- Burp Mobile Assistant [50]

Eine weitere bewährte Methode ist die Verwendung von Frida-Hooks [33] und „objection“ [39]. Auf einer sehr hohen Ebene ist Frida ein Framework, mit dem zur Laufzeit in den Code einer Anwendung eingegriffen werden kann. In diesem Fall soll mit dem Framework die Logik der Zertifikatsvalidierung umgangen werden. Dabei wird gezeigt, wie „objection“ beziehungsweise Frida ohne Jailbreak verwendet werden können. Um in späterer Folge die gepatchte IPA-Datei signieren zu können, bedarf es eines validen Provisioning Profiles und eines Code-Signing Zertifikats eines Apple Developer Accounts. Ein gültiges Provisioning Profile kann erstellt werden, indem eine Testanwendung in Xcode angelegt wird. Der Entwickler Account kann kostenlos bei Apple registriert werden. Mit Hilfe von Xcode kann das Herunterladen des Zertifikats auf den Computer erfolgen. Dazu wird in Xcode unter Einstellungen der Reiter „Accounts“ geöffnet. Um das Apple ID Konto hinzuzufügen,

muss auf das Pluszeichen in der linken unteren Ecke gedrückt und sowohl Apple ID als auch Passwort eingegeben werden.

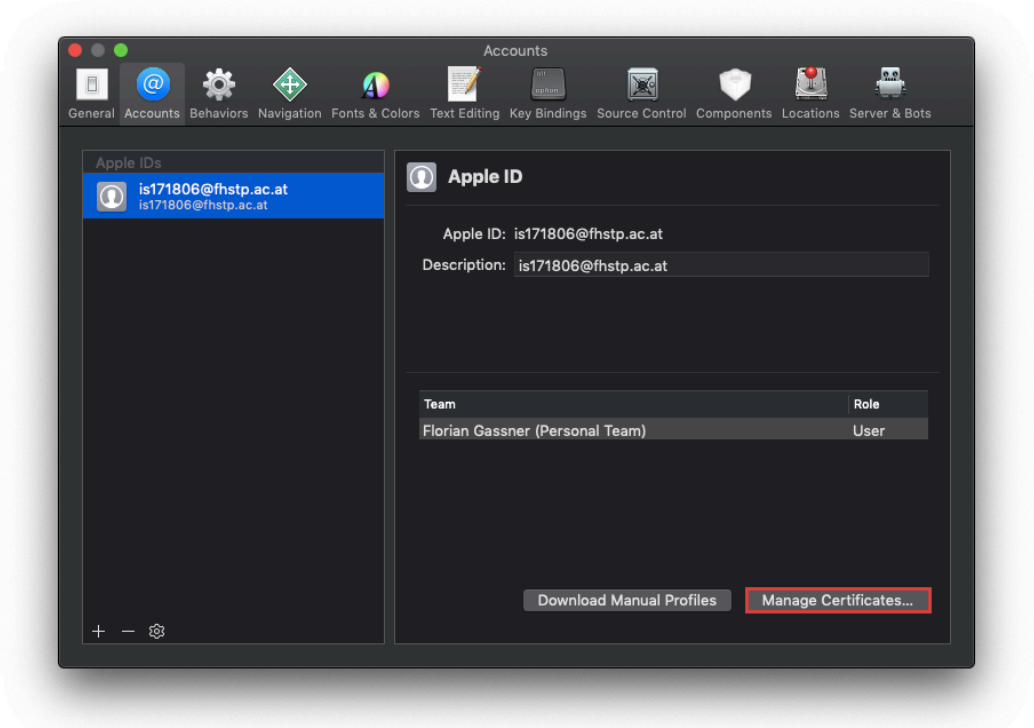

Abbildung 48: Xcode Account Einstellungen

Abbildung 48 zeigt, dass der Entwickler Account in die Entwicklungsumgebung Xcode hinzugefügt wurde. Anschließend kann mittels Klick auf „Manage Certificates…“ die Übersicht über alle Zertifikate, welche in Verbindung mit dem Entwickler Account stehen, eingesehen werden.

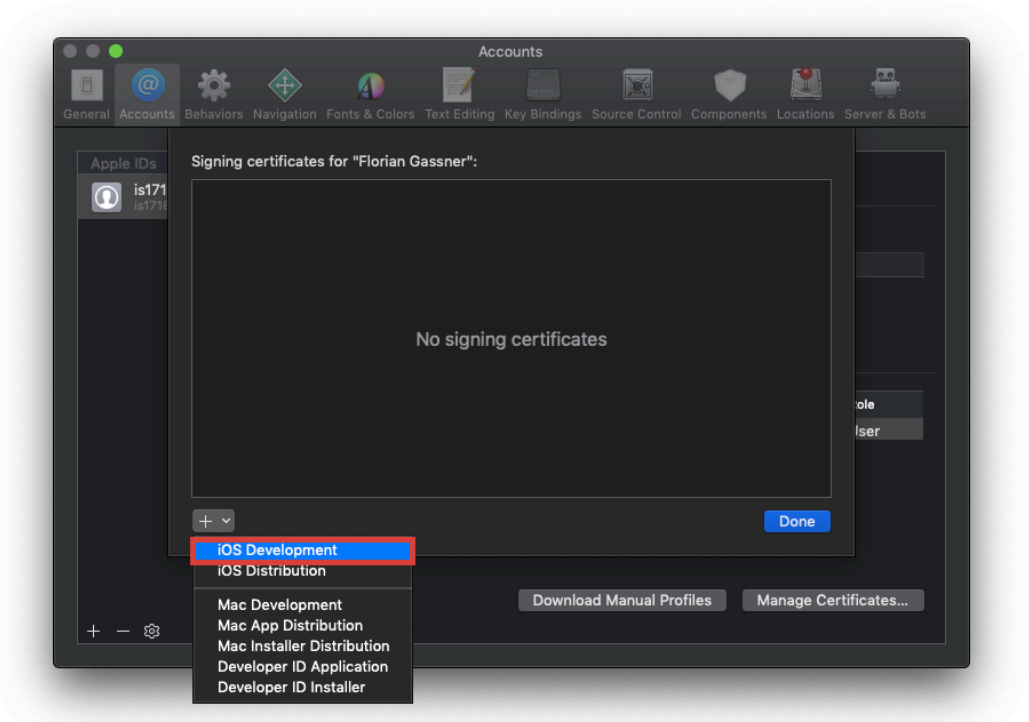

Abbildung 49: Xcode Download des Entwicklerzertifikats

Abbildung 49 zeigt die Oberfläche, mit welcher ein neues Entwicklerzertifikat von Apple angefordert werden kann. Die Erstellung eines neuen Zertifikats erfolgt, indem wiederum das kleine Pluszeichen in der linken unteren Ecke des Fensters ausgewählt wird. Im nächsten Schritt muss „iOS Development“ ausgewählt werden.

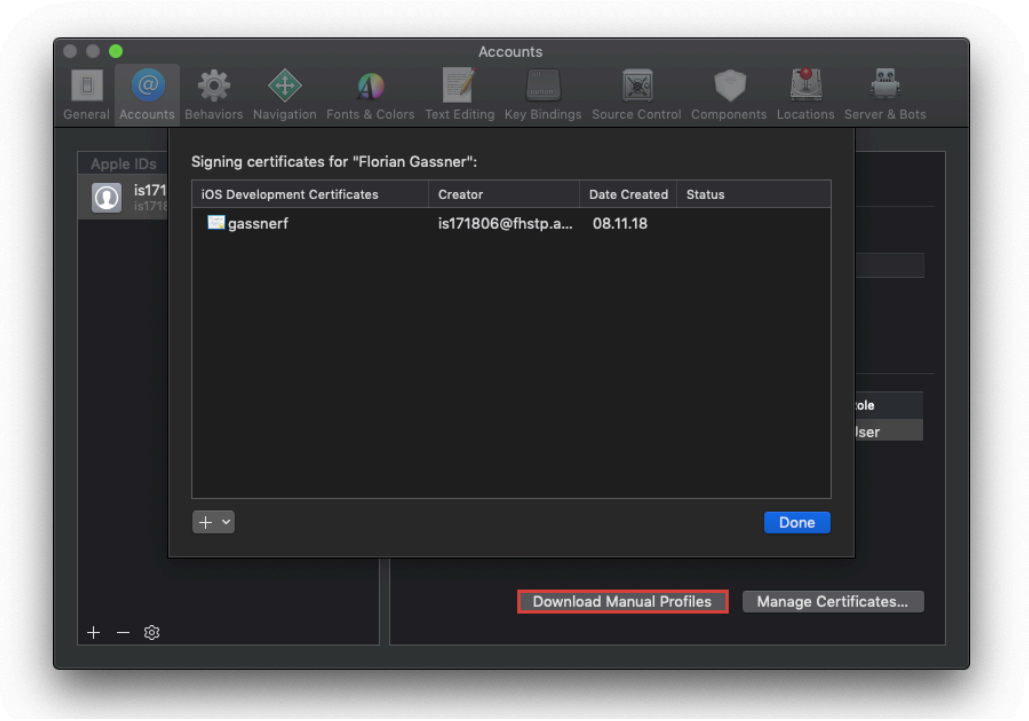

Abbildung 50: Xcode Entwicklerzertifikat

In Abbildung 50 ist ersichtlich, dass die Generierung eines neues „Signing Certificate“ für iOS Applikationen erfolgte. Nachdem das Zertifikat erfolgreich erstellt wurde, kann dieses Fenster mit „Done“ geschlossen werden. Zum Abschluss kann mit Hilfe von „Download Manual Profile“ das iOS Development Zertifikat auf den Computer geladen werden.

Mittels des Befehles „security find-identity -p codesigning -v“ (Abbildung 51) kann verifiziert werden, dass das Zertifikat am Computer zur Verfügung steht.

```
gassnerf:DVIA-v2-swift gassnerf$ security find-identity -p codesigning -v
  1) 5187E8FCBCBB82350896957C81B5A154CF201B28 "iPhone Developer:
is171806@fhstp.ac.at (K7GZRHPH5Q)"
     1 valid identities found
```

Abbildung 51: Ausgabe von security find-identity -p codesigning -v

Als nächster Schritt soll die dynamische Frida Gadget Bibliothek geladen werden, um die Anwendung zur Laufzeit modifizieren zu können. Unter Zuhilfenahme der Testapplikation DVIA wird diese mobile Anwendung extrahiert und die Binärdatei geändert, um die Bibliothek „FridaGadget.dylib“ zu laden. Danach werden sowohl die Bibliothek als auch die Binärdatei codiert und die aktualisierte IPA-Datei neu verpackt. „objection“ kann diese Schritte automatisiert in einem Arbeitsschritt durchführen. Dazu wird folgender Befehl in einem Terminal eingegeben: „objection patchipa -s IPA-Datei -c Zertifikatshash“ (Abbildung 52).

```
gassnerf:DVIA-v2-swift gassnerf$ objection patchipa -s DVIA-v2-swift.ipa -c
5187E8FCBCBB82350896957C81B5A154CF201B28
Using latest Github gadget version: 12.2.23
Patcher will be using Gadget version: 12.2.23
No provision file specified, searching for one...
Found provision file
/Users/gassnerf/Library/Developer/Xcode/DerivedData/TestProject-
ejjhaoahjnlgnzhetnfvtklxlmyc/Build/Products/Debug-
iphoneos/TestProject.app/embedded.mobileprovision expiring in 4 days,
10:09:57.165038
Found a valid provisioning profile
Working with app: DVIA-v2.app
Bundle identifier is: com.highaltitudehacks.DVIAswiftv2
Codesigning 21 .dylib's with signature 5187E8FCBCBB82350896957C81B5A154CF201B28
Code signing: libswiftRemoteMirror.dylib
Code signing: libswiftCoreImage.dylib
Code signing: libswiftObjectiveC.dylib
Code signing: libswiftCore.dylib
Code signing: libswiftCoreGraphics.dylib
Code signing: libswiftUIKit.dylib
Code signing: libswiftMetal.dylib
Code signing: libswiftCoreData.dylib
Code signing: libswiftDispatch.dylib
Code signing: libswiftos.dylib
Code signing: libswiftCoreFoundation.dylib
```

```
Code signing: FridaGadget.dylib
Code signing: libswiftDarwin.dylib
Code signing: libswiftQuartzCore.dylib
Code signing: libswiftCoreAudio.dylib
Code signing: libswiftAVFoundation.dylib
Code signing: libswiftFoundation.dylib
Code signing: libswiftCoreMedia.dylib
Code signing: libswiftCoreLocation.dylib
Code signing: libswiftSwiftOnoneSupport.dylib
Code signing: libswiftsimd.dylib
Creating new archive with patched contents...
Codesigning patched IPA...
Cannot find entitlements in binary. Using defaults
Not signing /var/folders/jk/swnmrj456z5d8mqj7kj4 k440000gn/T/DVIA-v2-swift-
frida.ipa.ac496030-2c62-4dea-9f31-cae4a185b71e/Payload/DVIA-
v2.app/libswiftRemoteMirror.dylib

Copying final ipa from /var/folders/jk/swnmrj456z5d8mqj7kj4 k440000gn/T/DVIA-v2-
swift-frida-codesigned.ipa to current directory...
Cleaning up temp files...
```

Abbildung 52: Ausgabe von objection patchipa -s DVIA-v2-swift.ipa -c <Zertifikatshash>

Eine erfolgreiche Ausführung des Befehls hat zur Folge, dass eine neue IPA-Datei erstellt wurde. Diese gepatchte IPA-Datei kann somit auf einem iOS-Gerät installiert und verwendet werden. Für die Installation kann ios-deploy [27] als auch Cydia Impactor [15] eingesetzt werden. Um ios-deploy verwenden zu können, muss die Applikation zuerst entpackt werden. Danach kann mit folgendem Befehl die Anwendung auf das iOS-Gerät geladen werden: „ios-deploy --bundle PFAD_ZUR_ENTPACKTEN_APPLIKATION -d“ (Abbildung 53). Die Option „-d“ bedeutet, dass nach erfolgreicher Installation, die Applikation in lldb [51], einem Softwaredebugger, aufgerufen werden soll.

```
gassnerf:DVIA-v2-swift gassnerf$ ios-deploy --bundle Payload/DVIA-v2.app -d
[....] Waiting for iOS device to be connected
[....] Using 4272806cdd484420b8d4f07d3233e674ada32261 (N69uAP, iPhone SE,
iphoneos, arm64) a.k.a. 'Florian’s iPhone'.
------ Install phase ------
[  0%] Found 4272806cdd484420b8d4f07d3233e674ada32261 (N69uAP, iPhone SE,
iphoneos, arm64) a.k.a. 'Florian’s iPhone' connected through USB, beginning
install
[TRUNCATED]
[ 95%] GeneratingApplicationMap
[100%] Installed package Payload/DVIA-v2.app
------ Debug phase ------
Starting debug of 4272806cdd484420b8d4f07d3233e674ada32261 (N69uAP, iPhone SE,
iphoneos, arm64) a.k.a. 'Florian’s iPhone' connected through USB...
[  0%] Looking up developer disk image
```

```
 [ 95%] Developer disk image mounted successfully
[100%] Connecting to remote debug server
--------------------------
(lldb) command source -s 0 '/tmp/F6275467-7F76-4D2B-BB66-8D12C49E0357/fruitstrap-
lldb-prep-cmds-4272806cdd484420b8d4f07d3233e674ada32261'
Executing commands in '/tmp/F6275467-7F76-4D2B-BB66-8D12C49E0357/fruitstrap-lldb-
prep-cmds-4272806cdd484420b8d4f07d3233e674ada32261'.
[TRUNCATED]
(lldb)     command script add -s asynchronous -f
fruitstrap_4272806cdd484420b8d4f07d3233e674ada32261.safequit_command safequit
(lldb)     connect
(lldb)     run
success
2018-11-11 09:18:47.011317+0100 DVIA-v2[660:137872] Frida: Listening on 127.0.0.1
TCP port 27042
```

Abbildung 53: Ausgabe von ios-deploy --bundle Payload/DVIA-v2.app -d

Nachdem die Applikation installiert wurde, kann diese geöffnet und mit „objection“ verbunden werden. Dies kann mit dem Befehl „objection explore“ (Abbildung 54) erfolgen.

```
gassnerf:DVIA-v2-swift gassnerf$ objection explore


     _     _       _   _
 ___| |_ |_|___ ___| |_|_|___ ___
| . | . | | |  -_|  _|  _| | . |   |
|___|___| _| |___|___|_| |_|___|_|_|
      |___|(object)inject(ion) v1.4.3

     Runtime Mobile Exploration
        by: @leonjza from @sensepost

[tab] for command suggestions
....highaltitudehacks.DVIAswiftv2 on (iPhone: 11.4.1) [usb] # ios sslpinning
disable
Job: 81e7faaf-b492-467b-90f1-9ae63f3c6cd0 - Starting
[9ae63f3c6cd0] [ios-ssl-pinning-bypass] [NSURLSession] Found 14 matches for
URLSession:didReceiveChallenge:completionHandler:
[9ae63f3c6cd0] [ios-ssl-pinning-bypass] [NSURLConnection] Found 5 matches for
connection:willSendRequestForAuthenticationChallenge:
[9ae63f3c6cd0] [ios-ssl-pinning-bypass] Hooking lower level method:
SSLSetSessionOption
[9ae63f3c6cd0] [ios-ssl-pinning-bypass] Hooking lower level method:
SSLCreateContext
[9ae63f3c6cd0] [ios-ssl-pinning-bypass] Hooking lower level method: SSLHandshake
```

```
[9ae63f3c6cd0] [ios-ssl-pinning-bypass] Hooking lower level method:
tls_helper_create_peer_trust
Job: 81e7faaf-b492-467b-90f1-9ae63f3c6cd0 - Started
....highaltitudehacks.DVIAswiftv2 on (iPhone: 11.4.1) [usb] #
```

Abbildung 54: Ausgabe von objection explore

Als finaler Schritt erfolgt die Deaktivierung des Certificate Pinnings mit Hilfe des Befehles „ios sslpinning disable“. Als Folge dessen ist die Kommunikation zwischen der Applikation und dem Server einsehbar. „objection“ sucht zur Laufzeit bekannte Implementationen von Certificate Pinning und ändert definierte Werte, um den gewünschten Effekt zu erzielen.

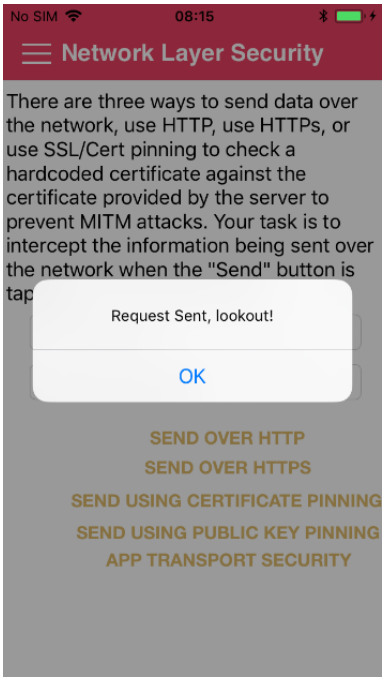

Abbildung 55: Senden eines HTTPS Requests

Abbildung 55 zeigt das Versenden einer Anfrage. Im Unterpunkt „Network Layer Security“ der „Damn Vulnerable iOS Application“ kann eine Anfrage an den Backend-Server über HTTP Secure Protokoll gesendet werden. Wurde das Certificate Pinning durch die vorangegangenen Schritte erfolgreich umgangen, besteht nun die Möglichkeit den Inhalt dieser Anfrage mittels eines Intercepting Proxies zu inspizieren. Abbildung 56 zeigt die Kommunikation zwischen einer iOS-Applikation und einem Server, welche mit dem Intercepting Proxy Burp Suite Scanner abgefangen wurde.

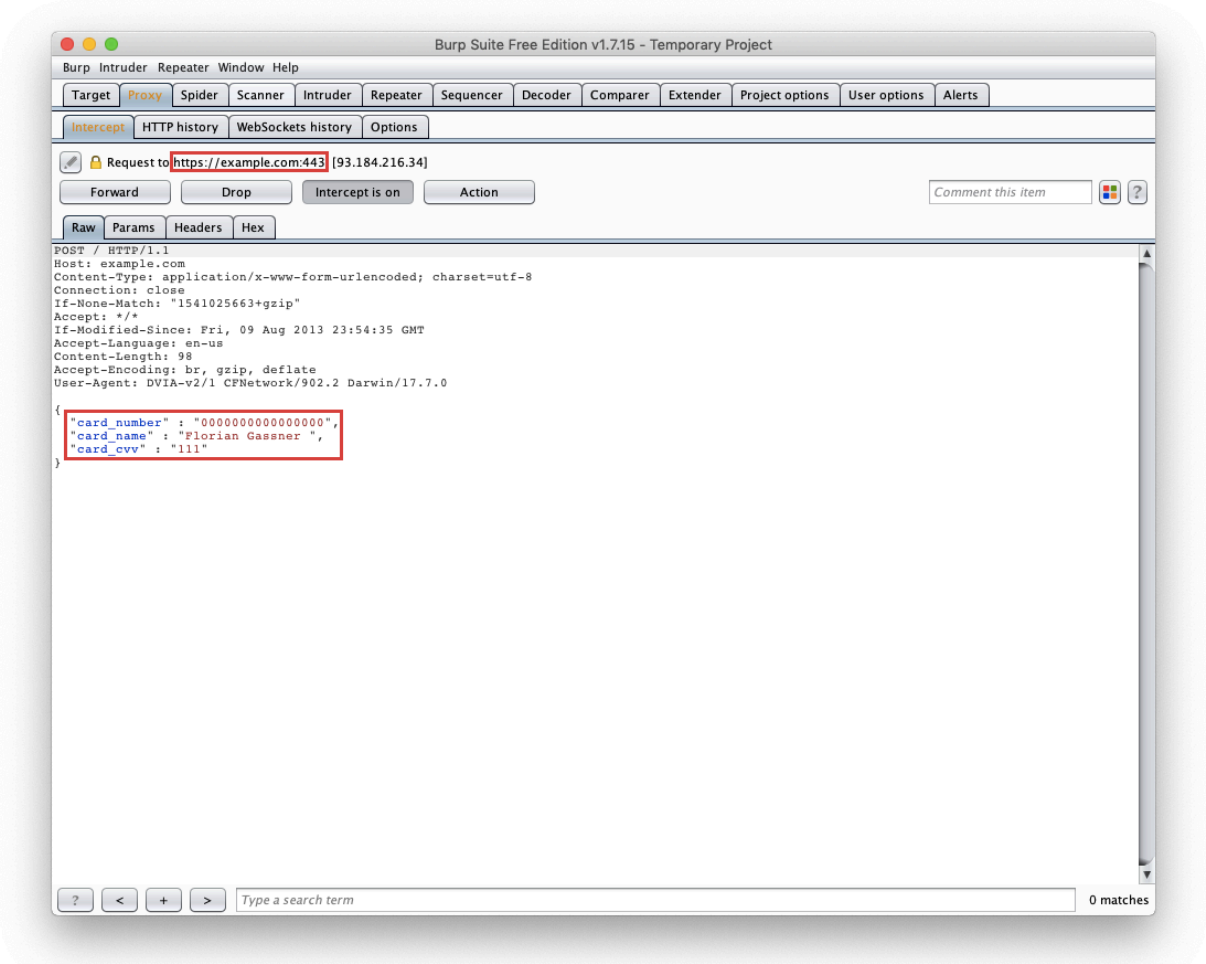

Abbildung 56: Einsicht in die Client-Server-Kommunikation

Wenn die oben genannten Techniken nicht erfolgreich sind, dann gibt es immer noch die Möglichkeit, Disassembler zu verwenden, um eine Zertifikatsvalidierung zu umgehen. Dabei wird die Applikation in einem Disassembler geladen und die Logik der Validierung gesucht. Je nachdem, welche Art des Certificate Pinnings bei der Erstellung der App verwendet wurde, muss diese dementsprechend angepasst werden. Zwei der bekanntesten und gebräuchlichsten Disassembler sind IDA [52] und Hopper [53]. Das Patchen der binären Datei wird in dieser Arbeit nicht näher behandelt.

4.4.10. Fazit

Sicherheitsüberprüfungen von iOS Applikationen fordern ein breites Spektrum an Wissen über alle Arten von Penetration Tests. Da Sicherheitsüberprüfungen von Anwenungen auf iOS Geräten aus mehreren Komponenten (Netzwerk, Web, Infrastruktur usw.) besteht, wird fundiertes Know-How in diesen Bereichen benötigt. Zum einen verbessert Apple permanent bestehende Mechanismen zum Schutz der iOS Geräte, und implementiert fortlaufend neue Hindernisse, welche für einen positiven Ausgang eines Penetration Tests ausgehebelt werden müssen. Zum anderen hängt der Erfolg einer Sicherheitsüberprüfung stark von der Aktualität sowie Funktionalität der verwendeten Werkzeuge ab. Da sich das mobile Betriebssystem iOS ständig weiterentwickelt und die Sicherheitsmechanismen laufend verbessert werden, müssen die Frameworks und Tools ebenfalls fortwährend angepasst werden. Zumal viele der Werkzeuge Open Source sind und in der Freizeit der Programmiererinnen beziehungsweise der Programmierer weiterentwickelt werden, kann dies zur Folge haben, dass die von Seiten Apple implementierten Sicherheitsfunktionen nicht

wirkungslos gemacht werden können. Ein ununterbrochenes Katz-und-Maus-Spiel zwischen Apple und Sicherheitsexpertinnen und Sicherheitsexperten. Die nachfolgende Tabelle veranschaulicht, ob Implementierungsfehler von der Entwicklerin oder dem Entwickler geschuldet sind oder auf das iOS Betriebssystem zurückzuführen sind.

	iOS Betriebssystem	Entwicklerin / Entwickler
Local Data Storage		x
Jailbreak Detection	x	
Runtime Manipulation	x	
Binary Protection		x
Touch ID Bypass		x
App Screenshot	x	
Pasteboard	x	
Cookies		x
Inter Process Communication Issues		x
WebView Issues		x
Network Layer Security		x

Abbildung 57: Tabelle Implementierungsprobleme

Die vorgehend beschriebenen Frameworks unterscheiden sich hauptsächlich in deren Aktualität. Frida ist mit Abstand am aktuellsten, Cycript und Needle hinken bereits zwei iOS Versionen nach. Allerdings existieren Bemühungen Cycript mit Hilfe von Frida am Leben zu erhalten. Es kann jedoch davon ausgegangen werden, dass Funktionen, welche ausschließlich durch Cycript offeriert werden, in naher Zukunft in das Frida Framework integriert werden. Vieles deutet darauf hin, dass Frida sich von allen Frameworks herauskristallisieren wird, sofern dies nicht bereits geschehen ist. In absehbarer Zeit wird es voraussichtlich eine Vielzahl an Werkzeugen geben, welche auf Frida aufbauen und Prozesse von Penetration Tests automatisieren, respektive erheblich erleichtern.

Frida zeichnet sich durch die Aktualität der unterstützten iOS Versionen, ebenso der Funktionalität aus. Zusätzlich existieren einige Werkzeuge, die auf dem Frida Framework aufbauen und die Arbeit, welche mit einer Sicherheitsüberprüfung einhergeht, ungemein erleichtert. Positiv hervorzuheben ist ebenfalls die Unterstützung bei Geräten, welche keinen aktiven Jailbreak besitzen. Wurde in die Applikation eine Jailbreak Erkennung implementiert, kann mit Hilfe des „Embedded“ Betriebsmodus von Frida diese für den Test als gegenstandslos betrachtet werden.

Mit der Entwicklung eines neuen Cycript Backends durch das Unternehmen „NowSecure“ besteht wieder Hoffnung, dass auch Cycript in der Zukunft für Penetration Tests von iOS Applikationen zur Verfügung stehen wird. Resultierend aus der Entscheidung, dass Frida für das Backend gewählt wurde, können hiermit auch alle Betriebsmodi verwendet werden. Ebenso wie beim Frida Framework, unterstützt damit auch Cycript iPhones, welche keinen Jailbreak aufweisen.

Das modulare Framework Needle unterstützt weder die aktuellste iOS Version 12, noch die vorherige elfte Version des mobilen Betriebssystems. Da auch die letzte Version bereits Mitte des Jahres 2017 publiziert wurde, deutet dies darauf hin, dass bei zukünftigen Sicherheitsüberprüfungen nicht mehr auf Needle zurückgegriffen sollte.

Zusammenfassend kann gesagt werden, dass mithilfe von Frameworks die Arbeit bei Penetration Tests mobiler iOS Anwendungen enorm vereinfacht wird. Ungeachtet davon bedarf es eines enormen Aufwands die notwendige Software kompatibel und funktionell zu halten, weshalb der Erfolg einer Sicherheitsüberprüfung immer mit der Aktualität sowie Funktionalität der verwendeten Werkzeuge einhergeht.

5. Schlussfolgerung und Ausblick

Mobile Applikationen finden auch in Firmen immer mehr Akzeptanz. Um die Sicherheit der unternehmenskritischen Daten zu gewährleisten, sollte auch die Sicherheit der eingesetzten Anwendungen einer Überprüfung unterzogen werden. Wie auch in allen anderen Bereichen der IT Welt, handelt es sich auch hier um ein ununterbrochenes Katz-und-Maus Spiel zwischen Angreiferinnen beziehungsweise Angreifern und Entwicklerinnen und Entwickler beziehungsweise Sicherheitsexpertinnen und Sicherheitsexperten. Das Gebiet der Penetration Tests mobiler Anwendungen ist im Gegensatz zu herkömmlichen Sicherheitsüberprüfungen, beispielsweise der Tests von Netzwerken oder auch Webanwendungen, noch jung. In den nächsten Jahren wird sich in diesem Bereich sicherlich einiges bewegen.

Um Penetration Tests mobiler Applikationen durchführen zu können, ist ein breites Spektrum an Wissen Voraussetzung. Eine Anwendung besteht nicht nur aus der Applikation selbst, sondern hat darüber hinaus mehrere Bereiche, welche auch in den Geltungsbereich einer Sicherheitsüberprüfung fallen. Ein Beispiel dafür ist ein Backend-Server. Es wurde gezeigt, dass bei schlechter Implementierung von Funktionalitäten, mit einfachen Mitteln sensible Daten einer Benutzerin, eines Benutzers oder eines Unternehmens öffentlich gemacht werden können.

Apple implementiert zwar laufend neue und aktualisierte Sicherheitsmaßnahmen. Jedoch erfolgt immer wieder die Entdeckung neuer Methoden um die Funktionen, welche die Applikation und deren Daten schützen sollten, zu umgehen. Am Beispiel von Jailbreaks ist erkennbar, dass die Exploits, die zu einer Übernahme des ganzen Systems führen, seltener werden. Aus diesem Grund ist die Wartezeit auf einen neuen Jailbreak nicht mehr so kurzweilig wie es bei früheren iOS Versionen der Fall war. Ein weiterer Anlass, welcher zu deutlich wenigeren Veröffentlichungen von Jailbreaks führt, ist die Tatsache, dass der monetäre Wert einer unveröffentlichten iOS Schwachstelle jenseits von einer Million Euro liegt [54]. Deswegen erfolgt die Suche nach alternativen Möglichkeiten, um eine Sicherheitsüberprüfung auf einem nicht veränderten Mobilgerät durchführen zu können. Wie am Beispiel Frida zu sehen ist, existieren bereits Methoden um Applikationen auch ohne Jailbreak zur Laufzeit analysieren beziehungsweise manipulieren zu können. Frida ist mit Abstand das aktuellste Framework, welches für Sicherheitsüberprüfungen herangezogen werden kann. Es gibt jedoch auch Bestrebungen, Cycript, weiter am Leben zu erhalten. Dafür wurde ein neues Backend entwickelt, welches auf dem Frida Framework aufbaut. Es kann davon ausgegangen werden, dass Funktionalitäten von Cycript eventuell den Weg in das Frida Framework finden. Es scheint, dass Needle kaum noch aktualisiert wird. Dies ist daran erkennbar, dass Needle die beiden neuesten Versionen des mobilen Betriebssystems von Apple, iOS 11 & iOS 12, derzeit nicht mehr unterstützt.

Zusammenfassend lässt sich sagen, dass das Ergebnis von Penetration Tests mobiler Applikationen stark von der Aktualität und Funktionalität der unterstützend eingesetzten Werkzeuge abhängig ist. Die Entwicklung der meisten Tools erfolgt durch freiwillige Entwicklerinnen und Entwickler. Es passiert nicht selten, dass die Entwicklung dieser Werkzeuge vernachlässigt und in weiterer Folge ganz aufgegeben wird. Aus den oben genannten Gründen, kann gesagt werden, dass Penetration Tests mobiler Anwendungen ein sich ständig weiterentwickelndes Gebiet ist und immer wieder einer Auffrischung des Wissens bedürfen jedoch für Unternehmen unabdingbar sind.

Literaturverzeichnis

[1] V. Kolici, V. Seguí, L. Barolli, F. Xhafa, E. E. D. Pinedo, und J. L. Nunez, „Analysis of Mobile and Web Applications in Small and Medium Size Enterprises“. .

[2] „DVIA (Damn Vulnerable iOS App) - A vulnerable iOS app for pentesting“, *DVIA (Damn Vulnerable iOS App)*. [Online]. Verfügbar unter: http://damnvulnerableiosapp.com/. [Zugegriffen: 08-Nov-2018].

[3] „Apple Reinvents the Phone with iPhone“, *Apple Newsroom*. [Online]. Verfügbar unter: https://www.apple.com/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/. [Zugegriffen: 29-Nov-2018].

[4] „Apple Previews Developer Beta of iPhone OS 3.0“, *Apple Newsroom*. [Online]. Verfügbar unter: https://www.apple.com/newsroom/2009/03/17Apple-Previews-Developer-Beta-of-iPhone-OS-3-0/. [Zugegriffen: 29-Nov-2018].

[5] „Local and Remote Notification Programming Guide: Document Revision History“. [Online]. Verfügbar unter: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/RevisionHistory.html. [Zugegriffen: 29-Nov-2018].

[6] „App Store - Support - Apple Developer“. [Online]. Verfügbar unter: https://developer.apple.com/support/app-store/. [Zugegriffen: 29-Nov-2018].

[7] „iOS 10.0“. [Online]. Verfügbar unter: https://developer.apple.com/library/archive/releasenotes/General/WhatsNewIniOS/Articles/iOS10.html#//apple_ref/doc/uid/TP40017084-SW1. [Zugegriffen: 29-Nov-2018].

[8] „UIPasteConfiguration - UIKit | Apple Developer Documentation“. [Online]. Verfügbar unter: https://developer.apple.com/documentation/uikit/uipasteconfiguration. [Zugegriffen: 29-Nov-2018].

[9] „iOS 12 - Features“, *Apple*. [Online]. Verfügbar unter: https://www.apple.com/ios/ios-12/features/. [Zugegriffen: 05-Dez-2018].

[10] Apple Inc., „iOS 12 Security Guide“. Apple Inc., 2018.

[11] K. Relan, *iOS Penetration Testing*, 1. Aufl. Apress.

[12] J. Wright, *Device Architecture and Common Mobile Threats*. sans.org, 2017.

[13] „Code Signing - Support - Apple Developer“. [Online]. Verfügbar unter: https://developer.apple.com/support/code-signing/. [Zugegriffen: 29-Nov-2018].

[14] J. Wright, *Mobile Platform Access and Application Analysis*. sans.org, 2017.

[15] „Cydia Impactor“. [Online]. Verfügbar unter: http://www.cydiaimpactor.com/. [Zugegriffen: 08-Nov-2018].

[16] „[News] Andrew Wiik recommend that everyone removes their saved paypal info from their cydia account until further notice.“, *reddit*. [Online]. Verfügbar unter: https://www.reddit.com/r/jailbreak/comments/a5wfq9/news_andrew_wiik_recommend_that_everyone_removes/. [Zugegriffen: 12-März-2019].

[17] „GitHub - pwn20wndstuff/Undecimus: unc0ver jailbreak for iOS 11.0 - 12.1.2“. [Online]. Verfügbar unter: https://github.com/pwn20wndstuff/Undecimus. [Zugegriffen: 12-März-2019].

[18] J. Wright, *Penetration Testing Mobile Devices Part 1*. sans.org, 2017.

[19] „Xcode - Apple Developer“. [Online]. Verfügbar unter: https://developer.apple.com/xcode/. [Zugegriffen: 07-Nov-2018].

[20] „Testing Guide Introduction - OWASP“. [Online]. Verfügbar unter: https://www.owasp.org/index.php/Testing_Guide_Introduction. [Zugegriffen: 16-März-2019].

[21] „Shodan“. [Online]. Verfügbar unter: https://www.shodan.io/. [Zugegriffen: 07-Nov-2018].

[22] „Google Hacking Database (GHDB)“. [Online]. Verfügbar unter: https://www.exploit-db.com/google-hacking-database/. [Zugegriffen: 07-Nov-2018].

[23] „Burp Suite Scanner | PortSwigger“. [Online]. Verfügbar unter: https://portswigger.net/burp. [Zugegriffen: 07-Nov-2018].

[24] P. Toomey, *A tool to check which keychain items are available to an attacker once an iOS device has been jailbroken: ptoomey3/Keychain-Dumper*. 2018.

[25] „Realm Browser“, *Mac App Store*. [Online]. Verfügbar unter: https://itunes.apple.com/us/app/realm-browser/id1007457278?mt=12. [Zugegriffen: 21-Nov-2018].

[26] Murphy, *A tool to read the binarycookie format of Cookies on iOS applications: as0ler/BinaryCookieReader*. 2018.

[27] „GitHub - ios-control/ios-deploy: Install and debug iPhone apps from the command line, without using Xcode“. [Online]. Verfügbar unter: https://github.com/ios-control/ios-deploy. [Zugegriffen: 08-Nov-2018].
[28] „Download PuTTY - a free SSH and telnet client for Windows“. [Online]. Verfügbar unter: https://www.putty.org/. [Zugegriffen: 18-März-2019].
[29] „Willkommen bei der App ‚Terminal‘ auf dem Mac“, *Apple Support*. [Online]. Verfügbar unter: https://support.apple.com/de-at/guide/terminal/welcome/mac. [Zugegriffen: 18-März-2019].
[30] „Password · Cydia“. [Online]. Verfügbar unter: https://cydia.saurik.com/password.html. [Zugegriffen: 19-März-2019].
[31] S. Nygard, *Generate Objective-C headers from Mach-O files. Contribute to nygard/class-dump development by creating an account on GitHub*. 2018.
[32] „security command - macOS - SS64.com“. [Online]. Verfügbar unter: https://ss64.com/osx/security.html. [Zugegriffen: 18-März-2019].
[33] „iOS“, *Frida • A world-class dynamic instrumentation framework*, 05-Nov-2018. [Online]. Verfügbar unter: https://www.frida.re/docs/ios/. [Zugegriffen: 08-Nov-2018].
[34] „Cycript“. [Online]. Verfügbar unter: http://www.cycript.org/. [Zugegriffen: 18-März-2019].
[35] *Cycript fork powered by Frida. Contribute to nowsecure/frida-cycript development by creating an account on GitHub*. NowSecure, 2018.
[36] *The iOS Security Testing Framework. Contribute to mwrlabs/needle development by creating an account on GitHub*. MWR Labs, 2018.
[37] S. Schleier und J. Willemsen, „Mobile Security Testing Guide“. OWASP.
[38] „Keychain Services | Apple Developer Documentation“. [Online]. Verfügbar unter: https://developer.apple.com/documentation/security/keychain_services. [Zugegriffen: 13-Nov-2018].
[39] *objection: runtime mobile exploration.* SensePost, 2018.
[40] „Core Data Programming Guide: What Is Core Data?“ [Online]. Verfügbar unter: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/index.html. [Zugegriffen: 13-Nov-2018].
[41] „iOS 5.0“. [Online]. Verfügbar unter: https://developer.apple.com/library/archive/releasenotes/General/WhatsNewIniOS/Articles/iOS5.html#//apple_ref/doc/uid/TP30915195-SW1. [Zugegriffen: 18-März-2019].
[42] „Realm Platform Documentation“. [Online]. Verfügbar unter: https://realm.io/docs/. [Zugegriffen: 22-Nov-2018].
[43] „Mac OS X Manual Page For system(3)“. [Online]. Verfügbar unter: https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/system.3.html. [Zugegriffen: 18-März-2019].
[44] „About Apple URL Schemes“. [Online]. Verfügbar unter: https://developer.apple.com/library/archive/featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html. [Zugegriffen: 26-Nov-2018].
[45] „UIWebView - UIKit | Apple Developer Documentation“. [Online]. Verfügbar unter: https://developer.apple.com/documentation/uikit/uiwebview. [Zugegriffen: 26-Nov-2018].
[46] „WebKit | Apple Developer Documentation“. [Online]. Verfügbar unter: https://developer.apple.com/documentation/webkit. [Zugegriffen: 26-Nov-2018].
[47] „Installing Burp’s CA Certificate in...“, *PortSwigger Web Security*. [Online]. Verfügbar unter: https://support.portswigger.net/customer/portal/articles/1841109-Mobile%20Set-up_iOS%20Device%20-%20Installing%20CA%20Certificate.html. [Zugegriffen: 07-Nov-2018].
[48] „Configuring an iOS Device to Work With Burp | Burp Suite Support Center“. [Online]. Verfügbar unter: https://support.portswigger.net/customer/portal/articles/1841108-configuring-an-ios-device-to-work-with-burp. [Zugegriffen: 07-Nov-2018].
[49] A. Diquet, *Blackbox tool to disable SSL certificate validation: including certificate pinning - within iOS and OS X Apps - nabla-c0d3/ssl-kill-switch2*. 2018.
[50] „Mobile testing“. [Online]. Verfügbar unter: https://portswigger.net/burp/documentation/desktop/mobile-testing. [Zugegriffen: 08-Nov-2018].
[51] „LLDB Homepage“. [Online]. Verfügbar unter: https://lldb.llvm.org/. [Zugegriffen: 09-Nov-2018].
[52] „IDA: About“. [Online]. Verfügbar unter: https://www.hex-rays.com/products/ida/. [Zugegriffen: 09-Nov-2018].
[53] „Hopper“. [Online]. Verfügbar unter: https://www.hopperapp.com/. [Zugegriffen: 09-Nov-2018].

[54] „ZERODIUM - How to Sell Your 0day Exploit to ZERODIUM“. [Online]. Verfügbar unter: https://www.zerodium.com/program.html. [Zugegriffen: 18-Dez-2018].