Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Maschood Ahmad

Lokale Sicherheitsanalyse mobiler Endgeräte am Beispiel von iOS

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Maschood Ahmad

Lokale Sicherheitsanalyse mobiler Endgeräte am Beispiel von iOS

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus-Peter Kossakowski
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Eingereicht am: 29. November 2016

Maschood Ahmad

Thema der Arbeit
Lokale Sicherheitsanalyse mobiler Endgeräte am Beispiel von iOS

Stichworte
Apple, iOS, iPhone, iOS Sicherheitsarchitektur, Sicherheitslücken, mobile Applikation

Kurzzusammenfassung
Die vorliegende Bachelorarbeit gibt einen Überblick zu Sicherheitsfunktionen und aufgetretenen Sicherheitslücken in mobilen Geräten am Beispiel von iOS. Es werden Sicherheitsfunktion von iOS vorgestellt, die zur Integrität des Systems beitragen. Zudem werden Sicherheitslücken die in iOS und im Vergleich zu Android und Windows Phone auftreten untersucht. Im Rahmen der Arbeit soll eine mobile Applikation für iOS entworfen und entwickelt werden. Diese soll den Endbenutzer Hinweise zu entsprechenden Sicherheitslücken in ihrem System geben.

Maschood Ahmad

Title of the paper
Local security analysis of mobile devices on the example of iOS

Keywords
Apple, iOS, iPhone, iOS security architecture, security vulnerabilities, mobile application

Abstract
The present thesis provides an overview of the security functions and observed vulnerabilities in mobile devices. The iOS operating system is investigated as a case study. A survey of security functions is discussed in iOS to present the integrity of the iOS based systems. In addition, the occurencerate of vulnerabilities in iOS, the Android and Windows Phone operating systems are compared and analyzed. A mobile application, which provides an overview of security vulnerabilities, was designed and developed in the iOS operating system.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Welt ist im Wandel: Die zunehmende internationale Verflechtung der Staaten, welche wir unter dem Begriff „Globalisierung“ kennen, hat eine neue Dimension erreicht. Wir sprechen heute von der „Digitalisierung“ der Welt. Angefangen bei Einkäufen, Wissen, Finanzen und Musik bis hin zu unserer sozialen Interaktion, alles verlagert sich Stück für Stück in die digitale Welt. Die Plattform für diese digitale Welt ist das Internet. Um dieses zu nutzen, war früher noch ein Personal Computer nötig, diese Zeiten sind mittlerweile vorbei. Heutzutage trägt fast jeder seinen kleinen portablen Mini-Computer im Alltag in Form eines Smartphones bei sich. Man muss sich nur die Masse an Informationen anschauen, welche auf dem Smartphone gespeichert werden können: Neben digitalen Medien wie Fotos, Videos, Musik und Sprachaufnahmen sind es vor allem die gespeicherten Passwörter für E-Mail-Konten, Online-Banking, soziale Netzwerke und vieles mehr, welche diese Geräte für Angreifer so attraktive machen. Zusätzlich verfügen sie über fast jegliche Art von Kommunikationsmöglichkeit wie WLAN, Bluetooth, Infrarot und Mobilfunk sowie GPS, Kameras und Mikrophone und können durch Sicherheitslücken quasi in mobile Überwachungsanlagen verwandelt werden. Galt früher das Sprichwort „Zeig mir, wer deine Freunde sind, und ich sage dir, wer du bist“, gilt heute eher Folgendes: „Zeig mir dein Smartphone und ich sage dir nicht nur, wer du bist, sondern auch, wo du dich gern aufhältst, welche (politischen) Einstellungen und Meinungen du hast, was du einkaufst – einfach alles!“ Für Cyberkriminelle sind Smartphones vor diesem Hintergrund äußerst lukrative Ziele. Dementsprechend sollte von Nutzern viel mehr Wert auf die Sicherheit dieser gelegt werden. Daraus resultieren das Interesse und die Relevanz, diesen Sachverhalt in der vorliegenden Bachelorarbeit näher zu beleuchten.

1.2 Ziel der Arbeit

Ein Ziel dieser Bachelorarbeit ist es, auf die Sicherheitsarchitektur von iOS einzugehen und diese mit anderen mobilen Betriebssystemen (Android, Windows Phone) zu vergleichen. Darauf basierend wird vorliegend eine Sicherheitsanalyse-Applikation (App) für iOS entwickelt. Diese soll dem Benutzer eines Apple-Smartphones ermöglichen, in seinem System eine Überprüfung auf Sicherheitslücken durchzuführen. Dieses ist sowohl lokal als auch über einen Server möglich. Die Sicherheitslücken werden durch diese App nicht geschlossen, es werden lediglich Informationen zu den im System bestehenden Sicherheitslücken angezeigt. Des Weiteren ist eine Überprüfung auf netzwerkbasierte Sicherheitslücken möglich, dies und die serverbasierte Systemüberprüfung werden durch einen externen Server durchgeführt, welcher nicht Bestandteil dieser Arbeit ist, sondern in einer weiteren Bachelorarbeit[1] umgesetzt wird.

[1] Davut Kuru, derzeitiger Titel der Bachelorarbeit „Erweiterung eines Servers zur Analyse der mobilen Endgeräte auf Sicherheitslücken“, Änderungen vorbehalten.

1.3 Zielgruppe der Arbeit

Die vorliegende Arbeit richtet sich an jeden, der sich für Sicherheit auf mobilen Geräten mit iOS interessiert. Leser, welche sich wenig oder nie mit Sicherheit in IT-Systemen beschäftigen, profitieren in besonderem Maßes durch diese Arbeit.

1.4 Struktur der Arbeit

In Kapitel zwei werden die Sicherheitsarchitektur von iOS und alle daran beteiligten Komponenten, die zur Systemsicherheit beitragen, vorgestellt. Darüber hinaus werden die Sicherheitsmechanismen für die Sicherheit von Applikationen sowie Daten und zur Verschlüsselung beschrieben. Anschließend wird erörtert, was Sicherheitslücken/Schwachstellen sind und welche verschiedenen Arten von diesen bei iOS vorhanden sind.

Das dritte Kapitel widmet sich der Analyse von Sicherheitsaspekten des Betriebssystems iOS und dem, wie die in Kapitel zwei vorgestellten Sicherheitsmechanismen eingesetzt werden. Dabei wird ein Vergleich zur Sicherheit von anderen mobilen Betriebssystemen angestellt, sowohl in Hinsicht auf die verwendeten Sicherheitsmechanismen als auch -lücken.

Im vierten Kapitel werden anhand der Anforderungsanalyse die nötigen Anforderungen zum Erstellen einer iOS-Applikation, welche ein iPhone auf dessen Schwachstellen hin überprüfen soll, ermittelt. Dabei werden die im dritten Kapitel gewonnenen Erkenntnisse einfließen. Die Anforderungen werden als Anwendungsfälle visualisiert.

Im fünften Kapitel wird zunächst die Systemarchitektur vorgestellt und, welche Vorgaben vom Server beachtet werden müssen. Anschließend wird ein Konzept vorgestellt, um die Anforderungen aus dem vierten Kapitel umzusetzen. Außerdem beschäftigt sich das Kapitel mit dem Entwurf der Grafischen Benutzeroberfläche der Applikation.

Das sechste Kapitel beschreibt sowohl die Anforderungen als auch, wie das Konzept aus den Kapiteln vier und fünf programmiertechnisch umgesetzt wurde. Außerdem wird ein Einblick in die Softwarearchitektur von iOS gegeben.

Im siebten Kapitel wird die fertig implementierte Applikation durch verschiedene Testfälle auf deren Funktion hin überprüft.

Das achte und letzte Kapitel gibt wieder, was im Rahmen dieser Arbeit erarbeitet wurde und, ob die zuvor definierten Ziele erreicht wurden. Schließlich gibt es einen Ausblick dahingehend, was in einer weiteren Arbeit verbessert oder der App hinzugefügt werden könnte.

2 Grundlagen

2.1 Sicherheitsarchitektur von iOS

Bei der Entwicklung von iOS seit 2007 hat Apple großen Wert auf die Sicherheit gelegt, diese soll zugleich funktional als auch benutzerfreundlich sein.

> *„Während der Entwicklung der besten mobilen Plattform aller Zeiten konnten wir auf jahrzehntelange Erfahrungen zurückgreifen, um eine völlig neue Architektur zu erstellen. Wir haben dabei auch an die Sicherheitsrisiken der Desktopumgebung gedacht und uns bei iOS für ein vollkommen neues Sicherheitskonzept entschieden.“* [1, S. 4]

Die Kernkomponenten der Systemsicherheit, bei der sowohl die Software als auch die Hardware sicher sind, bilden der sichere Startvorgang, die Systemsoftwareaktualisierung und die *„Secure Enclave“*-Architektur. Applikationen von Drittanbietern wie Google oder Adobe werden vor der Publizierung im App Store genauestens geprüft und analysiert, bevor diese ggf. eine Freigabe erhalten. Dieses Konzept führt dazu, dass es bei iOS, verglichen mit anderen Plattformen wie Android und Windows bis dato nur wenig Schadsoftware und kaum erfolgreiche Attacken seitens von Malware-Entwickler gibt. Nachfolgend werden die wichtigsten Komponenten, welche zur Systemsicherheit beitragen, detailliert vorgestellt und beschrieben: Die Abbildung 2.1 gibt hierzu einen Überblick. [1]

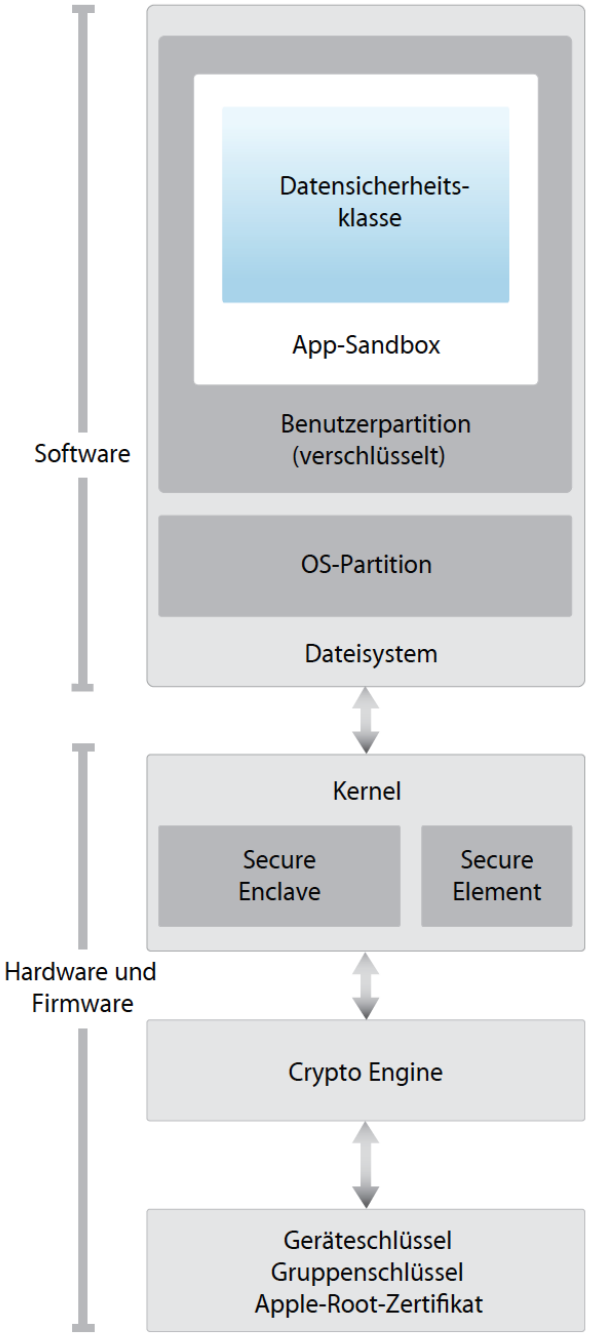

Abbildung 2.1: Diagramm zur Sicherheitsarchitektur von iOS [1, S. 4]

2.1.1 Startvorgang

Die einzelnen Bestandteile des Startvorgangs enthalten kryptografisch von Apple signierte Komponenten zur Wahrung der Systemintegrität. Der sichere Startvorgang wird als *„Chain of Trust“* bezeichnet und beginnt damit, dass beim Startvorgang des iOS-Geräts ein unveränderlicher Code, welcher implizit vertrauenswürdig ist, vom Boot-ROM geladen wird. Der Code wird bei der Herstellung des Chips festgelegt und enthält den öffentlichen Schlüssel der Apple-Root-Zertifizierungsstelle. Damit wird geprüft, ob der Low-Level-Bootloader (LBB) von Apple signiert wurde. Nach einigen weiteren Instanzen wird schlussendlich der iOS-Kernel (XNU) vom verifizierten Bootloader (iBoot) ausgeführt. Dieser Vorgang sorgt dafür, dass untere Softwareebenen von Angreifern nicht unbefugt manipuliert werden können. [1]

Bei iOS-Geräten, welche über eine Mobilfunknetzanbindung verfügen, verwendet das Basisband-Subsystem einen ähnlichen Prozess für den Startvorgang. Beim Secure-Enclave-Co-Prozessor (vgl. Kapitel 2.1.3) wird ebenfalls der sichere Startvorgang verwendet. Wird ein Schritt beim Bootvorgang nicht geladen oder kann der Prozess nicht geprüft werden, wird dieser abgebrochen und das Gerät in den Wartungsmodus versetzt. Es kann auch passieren, dass der Boot-ROM den LBB nicht laden oder prüfen kann, dann wird das Gerät in den DFU-Modus (Device Firmware Upgrade) versetzt. In beiden Fällen ist es unumgänglich, das Gerät auf die Option *„Werkeinstellung“* zurückzusetzen und die Systemfirmware neu zu installieren. [1]

2.1.2 Systemsoftwareautorisierung

Softwareaktualisierungen bzw. Security-Updates sollen Sicherheitslücken in der Systemsoftware schließen. Sind sie jedoch umkehrbar, geht ein Großteil ihres Nutzens verloren, dadurch wird das Gerät verwundbar (vulnerable). Sollte einem potenziellen Angreifer ein iPhone in die Hände fallen, könnte dieser die Software auf eine ältere Version zurücksetzen und dadurch bekannte Schwachstellen ausnutzen.

Um dem einen Riegel vorzuschieben, verwendet iOS einen Prozess der Systemsoftwareautorisierung. Diese soll verhindern, dass Geräte in eine ältere Softwareversion zurückversetzt werden. Die Aktualisierung kann über iTunes oder drahtlos (over the air, OTA) installiert werden. Bei Letzterer wird nicht die komplette Systemsoftware geladen, sondern nur die benötigten Komponenten, wodurch eine bessere Netzwerkeffizienz erreicht wird. Im Falle eines Updates, sei es via iTunes oder OTA, wird zunächst ein Apple-Server kontaktiert, welcher für die Autorisierung der Installation zuständig ist. Sobald die Verbindung hergestellt ist, sendet der Client eine Liste von verschlüsselten Kennzahlen für jede benötigte Komponente (z. B. LLB, iBoot, Kernel, iOS-Version Image), einen Anti-Zufallswert (Nonce) und die eindeutige Kennung des Gerätes (ECID). Die Kennzahlen werden nun vom Autorisierungsserver überprüft. Stimmen diese überein, wird die ECID den Kennzahlen hinzugefügt und das Ergebnis signiert. Beim Aktuali-

sierungsvorgang wird nun der komplett signierte Datensatz an das Gerät übertragen. Durch das Einfügen der ECID ist dieser nun spezifisch für das Gerät personalisiert. Beim Startvorgang kommt die Chain-of-Trust-Evaluierung zum Zuge (siehe 2.1.1) und prüft, ob die Signatur von Apple stammt. Außerdem wird geprüft, ob die Kennzahl in Kombination mit der ECID des Gerätes von der Signatur abgedeckt wird. Durch diese Maßnahme wird sichergestellt, dass die Autorisierung einzig und allein für das jeweilige Gerät gilt und nicht auf andere übertragen werden kann. Die Nonce soll verhindern, dass im Falle eines Abfangens der Serverantwort diese für die Manipulation eines Gerätes verwendet wird. [1]

Kategorien von Softwareaktualisierungen

iOS-Updates lassen sich wie folgt in drei Kategorien einteilen: Major, Minor und Patch-Update

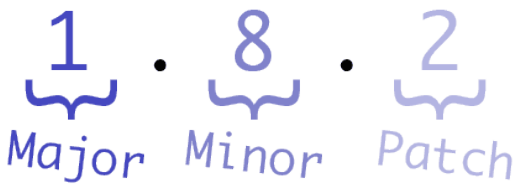

Abbildung 2.2: Major, Minor, Patch [2]

Major Update
Diese Art von Updates, beispielsweise iOS 9 auf iOS 10, erscheint normalerweise einmal pro Jahr. Hier können unter anderem gravierende Änderungen an dem System und der Prozessarchitektur vorgenommen werden. Dieses Update enthält meist viele kleine und große Erneuerungen bei den Funktionen.

Minor Update
Dieses ist ein Update innerhalb der Major Version, beispielsweise von iOS 9.0 auf iOS 9.1, in dieser Form von Updates, welche in der Regel zwei- bis dreimal im Jahr erscheinen, sind vorrangig neue Funktionen enthalten.

Patch Update
Ein Update innerhalb der Minor Version, beispielsweise von iOS 9.3.3 auf iOS 9.3.5, diese Form von Update dient der Behebung von Sicherheitslücken oder schwerwiegender Fehler.

2.1.3 Secure Enclave Co-Prozessor

Der Secure Enclave (SE) ist ein Co-Prozessor, der in den Prozessoren der A-Reihe seit dem Apple A7 (2013) integriert ist und als Erstes beim iPhone 5S zum Einsatz kam. Die Architektur basiert auf einem modifizierten L4-Microkernel [3]. Der SE verfügt über Zugriff auf die eindeutige Kennung (Unique - UID), die selbst Apple nicht kennt und worauf andere Komponenten des Systems keinen Zugriff haben. Die UID wird durch den SE selbst nach der Manufaktur unter Verwendung eines hardwarebasierten Zufallszahlengenerators erzeugt. [1]

Beim Startvorgang (siehe 2.1.1) wird ein temporärer Schlüssel erzeugt und mit der UID des Gerätes verknüpft, damit wird ein Teil des Speicherbereiches für den SE verschlüsselt. Zu den Aufgaben des SE gehören die Datenverschlüsselung und das Schlüsselmanagement. Er stellt die Integrität dieser sicher, selbst wenn der Kernel von Angreifern kompromittiert wurde. Zusätzlich werden die Daten, welche der SE ins Dateisystem schreibt, mit einem Schlüssel, der mit dem UID und einem Anti-Reply-Zähler verknüpft ist, verschlüsselt. Der vom SE verwendete Startvorgang und die Softwareaktualisierung sind vom Anwendungsprozessor unabhängig. Zudem verwenden der Anwendungsprozessor und der SE eine interruptgesteuerte Kommunikation (Mailboxes). [1]

2.1.4 Anwendungsprozessor

Ähnlich der UID vom SE verfügt der Anwendungsprozessor über eine Gerätegruppen-ID (GID), welche bei allen Prozessoren einer Geräteklasse gemeinsam verwendet wird, z. B. bei allen Apple-A8-Prozessoren. Diese wird bei sicherheitsunkritischen Aufgaben wie dem Bereitstellen von Systemsoftware verwendet. Durch die Einbettung der Schlüssel (UID und GID) in Silizium, wird sichergestellt, dass diese nicht manipuliert werden können und nur der Advanced Encryption Standard (AES)-Engine zugänglich sind. Es gibt keine Möglichkeit, über Debugging-Schnittstellen wie Joint Test Action Group (JTAG) darauf zuzugreifen. [1]

2.1.5 Touch ID

Touch ID (TID) ist ein Fingerabdrucksensorsystem, dieses ermöglicht dem Benutzer, das Gerät mit einem Fingerabdruck zu entsperren. Dadurch wird die Verwendung von längeren komplexeren Codes vereinfacht, da diese nun weniger oft eingegeben werden müssen. Für die Verwendung von Touch ID muss eine codebasierte Sperre für das Gerät eingerichtet werden. Es können bis zu fünf Fingerabdrücke mit dem Gerät registriert werden, sobald TID beim Scannen einen registrierten Fingerabdruck erkennt, wird das Gerät entsperrt. [1]

Die Fingerabdruck-Informationen werden verschlüsselt auf dem Gerät gespeichert und können nur vom Secure Enclave (siehe 2.1.3) entschlüsselt werden. Dieser stellt auch die nötigen Schlüssel bereit, damit TID das Gerät entsperren kann. Der SE verwirft die Schlüssel nach 48 Stunden beziehungsweise nach einem Neustart des Gerätes oder nach fünf fehlgeschlagenen Scanversuchen. [1]

Dass eine biometrische Sicherheitsfunktion für die Sperrung eines Gerätes nicht immer die beste Lösung ist, wurde 2013 von den Hackern vom Chaos Computer Club (CCC) unter Beweis gestellt. Ihnen ist es mit einfachsten Mitteln gelungen, einen Fingerabdruck zu fälschen und so das TID-System zu überlisten. Um dies zu realisieren genügte den Hackern einen Fingerabdruck, welchen sie von einer Glasoberfläche abfotographierten, um einen künstlichen Finger zu erzeugen. Damit waren es dann möglich, ein iPhone 5s zu entsperren, welches mit Touch ID geschützt war. [4]

> *„Wir hoffen, daß dies die restlichen Illusionen ausräumt, die Menschen bezüglich biometrischer Sicherheitssysteme haben. Es ist einfach eine dumme Idee, etwas als alltägliches Sicherheitstoken zu verwenden, was man täglich an schier unendlich vielen Orten hinterläßt. Die Öffentlichkeit sollte nicht länger von der Biometrie-Industrie mit falschen Aussagen an der Nase herumgeführt werden. Biometrie ist geeignet, um Menschen zu überwachen und zu kontrollieren, nicht um alltägliche Geräte vor dem Zugriff zu sichern.“*-
> Frank Rieger (Sprecher des CCC)[4]

2.2 Applikationssicherheit

Apps füllen das Betriebssystem sozusagen mit Leben und bieten dem Benutzer im Idealfall einen großen Produktivitätsgewinn, gehören aber auch zu den kritischen Elementen der Sicherheitsarchitektur. So können durch sie Benutzerdaten sowie die Systemsicherheit und Stabilität des Systems gefährdet werden. Daher ist es sehr wichtig, dass Apps signiert und auf Gültigkeit überprüft werden.

2.2.1 Applikations-Codesignierung

Nach dem Startvorgang (siehe 2.1.1) prüft der iOS-Kernel, welche Benutzerprozesse und Apps ausgeführt werden dürfen. Dafür wird der gesamte ausführbare Code durch das von Apple ausgegebene Zertifikat auf Validierung und Signierung geprüft. Dadurch wird sichergestellt, dass alle Apps aus genehmigten Quellen stammen und nicht manipuliert sind. Hierbei wird das vom Betriebssystem bekannte „Chain of Trust“-Konzept (siehe 2.1.1) auf Apps ausgeweitet. Dadurch wird ein Ausführen von nicht signierten Codes unterbunden. Die im Auslieferungszustand enthaltenen System-Apps (z. B. Safari) werden von Apple signiert. [1]

Apple prüft die Identität von jedem Entwickler (Privatperson oder Unternehmen) und stellt erst nach einer erfolgreichen Verifikation das Zertifikat, welches zum Validieren und Publizieren von Apps im App Store notwendig ist, aus. Dies schreckt viele Entwickler von Schadsoftware ab und steigert das Vertrauen der Nutzer in die Qualität der Apps. iOS erlaubt Entwicklern auch die Verwendung von Frameworks, diese müssen aber mit der Team-ID, welche aus dem von Apple ausgegebenen Zertifikat extrahiert wird, validiert werden. Durch Validierung der Codesignatur für alle dynamischen Bibliotheken sollen das System und andere Apps vor dem Ausführen von Codes in ihrem Adressbereich geschützt werden. Eine App darf nur einen Code laden, welcher von den im System vorinstallierten und mit derselben Team-ID versehenen Bibliotheken stammt. Der auf dem System vorinstallierte Code hat keine Team-ID. Aus diesem Grund kann dieser nur auf Bibliotheken, die ebenfalls in dem System vorinstalliert sind, zugreifen. [1]

Generell können Apps nur im App Store vertrieben werden, es gibt jedoch eine Ausnahme: das Apple Developer Enterprise Program (ADEP). Hierfür können sich Entwickler und Unternehmen mit einer Data Universal Numbering System (D-U-N-S)-Nummer bewerben. Im Gegensatz zu anderen mobilen Plattformen wie Android oder Symbian erlaubt iOS nicht das Herunterladen und Installieren von unsignierten Apps. [1]

2.2.2 Sandboxing

Sicherheitslücken in Apps können dazu führen, dass Angreifer das System kompromittieren und dadurch Zugriff auf Systemressourcen und Benutzerdaten bekommen. Um dies zu vermeiden, arbeiten Apps in einer Sandbox (2.3):

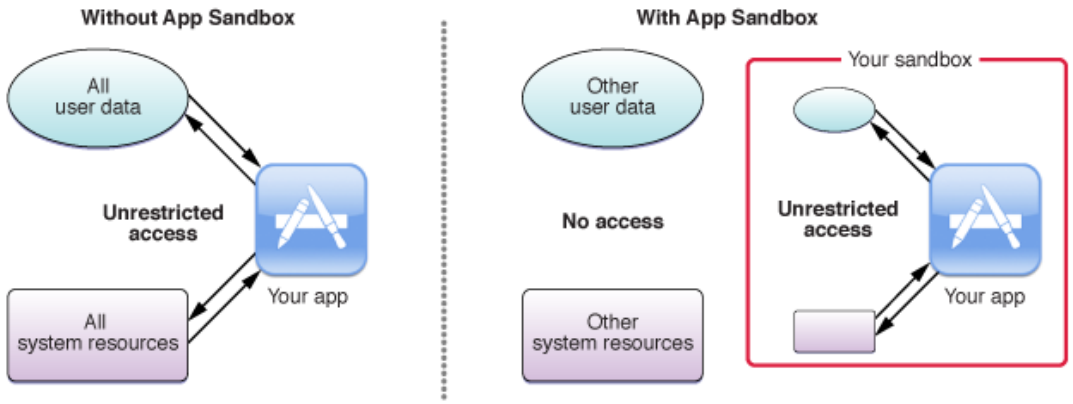

Abbildung 2.3: App-Sandbox in iOS [5]

Jede App hat ihr eigenes Heimatverzeichnis für ihre Daten, sie kann nicht auf Informationen von Drittanbieter-und System-Apps zugreifen oder diese verändern. Dadurch können Apps keine von anderen Apps gespeicherten Informationen abrufen oder verändern. Möchte eine App dennoch Informationen eines anderen Anbieters verwenden, welche ihr nicht zugeordnet sind, stellt iOS explizit Dienste hierfür bereit. Eine weitere Schutzmaßnahme ist, dass ein Großteil von iOS- und Apps anderer Anbieter über den nicht so privilegierten UNIX-Benutzer *mobil* ausgeführt werden. Wenn eine App andere Daten als die eigenen benötigt, wird dies über festgelegte Berechtigungen gesteuert. [1]

2.2.3 Berechtigungen/Entitlements

Berechtigungen (Entitlements) sind Schlüssel-Wert-Paare, welche eine Authentifizierung für den Zugriff auf Benutzerinformationen und die Erweiterbarkeit von Apps ermöglichen. Berechtigungen werden zusammen mit der App digital signiert und können daher nicht verändert werden. Sie werden auch von System-Apps und Hintergrundprozessen zur Durchführung bestimmter privilegierter Vorgänge (z. B. Hardwarezugriff auf Kamera, Bluetooth, Mikrophone etc.) verwendet, die sonst Root-Rechte erfordern würden. So wird das Risiko einer ungewollten Rechterweiterung durch einen manipulierten Hintergrundprozess, oder einer System-App verringert. Eine weitere Möglichkeit für den Austausch von Inhalten sind App-Gruppen. [1]

App-Gruppen

App-Gruppen ermöglichen Apps, Inhalte mit anderen Apps zu teilen, wenn diese zum selben Entwickler-Account gehören. Dafür muss eine App-Gruppe auf dem von Apple bereitgestellten Entwicklerportal erstellt und konfiguriert werden. [1]

2.2.4 ASLR

Die Speicherverwürfelung (Adress-Space-Layout-Randomization – ASLR) erschwert Angreifern die Ausnutzung von Sicherheitslücken wie Speicherüberläufen. Ab iOS 5 wird Apps und Bibliotheken ein zufälliger Speicherbereich zugewiesen. Durch die zufällige Platzierung ist ein Angriff deutlich schwieriger, da Speicheradressen und Segmentierungen nicht vorhergesagt und so im Exploit-Code hart codiert werden können. [1]

2.3 Datensicherheit und Verschlüsselung

Nachfolgend werden die Funktionen der Hard- und Softwareverschlüsselung vorgestellt.

2.3.1 Hardwaresicherheit

Für die Verschlüsselung auf jedem iOS-Gerät kommt eine 256-Bit-basierte AES- Crypto-Engine, welche im Direct Memory Access-Pfad (DMA-Pfad) zwischen dem Flash- und Hauptspeicher liegt, zum Einsatz (siehe Abb. 2.1). Diese Positionierung sorgt für eine höchst effiziente Dateiverschlüsselung, welche bei dem komplexen Verschlüsselungsvorgang auch nötig ist. Die Unique ID wird bei der Herstellung in den Anwendungsprozessor und Secure Enclave eingebrannt und die Group-ID wird kompiliert, somit ist ausgeschlossen, dass diese von einer Soft- oder Firmware direkt gelesen werden können (siehe 2.1.3, 2.1.4). Die Ergebnisse der Verschlüsselungs- oder Entschlüsselungsoperationen können lediglich von der AES-Engine gelesen werden. [1]

Die UID ermöglicht es, die Daten kryptografisch an ein Gerät zu binden, dadurch werden Speicherchips physisch an ein Gerät gebunden. Sollte nun der Speicherchip in ein anderes Gerät bewegt werden, so ist der Datenzugriff auf diesen nicht möglich. Alle weiteren kryptografischen Schlüssel für andere Operationen werden vom Zufallszahlengenerator (Random Number Generator) des Systems mit einem auf Counter mode Deterministic Random Byte Generator (CTR_DRGB)[6] basierenden Algorithmus generiert. [1]

Vor dem Hintergrund der sicheren Schlüsselerstellung stellt sich die Frage, wie Schlüssel ebenfalls sicher gelöscht werden können. Dies ist kein leichtes Unterfangen bei einem Flash-Speicher, da bei diesem aufgrund der Architektur möglicherweise mehrere Kopien gelöscht werden müssen. Um dieser Problematik entgegenzuwirken, bieten iOS-Geräte die Funktion Effaceable Storage (Auslöschbarer Speicher) zum sicheren Löschen der Schlüssel an. [1]

2.3.2 Datensicherheit

Zusätzlich zur Hardwareverschlüsselung werden Funktionen für die Datensicherheit verwendet (Apple nennt diese Funktion Data Protection), um die im Flash-Speicher abgelegten Daten zu schützen. System-Apps wie Nachrichten, E-Mail, Kalender oder

Fotos werden standardmäßig mit der Data Protection versehen. Apps anderer Anbieter erhalten diesen Schutz automatisch, wenn diese auf iOS 7 oder höher installiert wurden.

> *„Datensicherheit wird durch die Erzeugung und Verwaltung einer Hierarchie von Schlüsseln implementiert. Sie baut auf den Technologien zur Hardwareverschlüsselung auf, die in jedes iOS-Gerät integriert sind.“* [1, S. 11]

Jede Datei, die in der Datenpartition erstellt wird, erhält einen ihr explizit zugewiesenen 256-Bit-Dateischlüssel (Per-File-Key). Dieser wird von AES-Engine (siehe 2.3.1) zur Verschlüsselung mit dem AES-CBC-Modus (auf Geräten mit A8-Prozessor AES-XTS) verwendet, wenn die Datei in den Flash-Speicher geschrieben wird. Der Per-File-Key wird mit einem Klassenschlüssel sicher verpackt, welcher wiederum in den Metadaten der Datei gespeichert wird. Es besteht die Wahl zwischen vier verschiedenen Dateisicherheitsklassen: NSFileProtectionComplete, NSFileProtectionCompleteUnlessOpen, NSFileProtectionCompleteUntilFirstUserAuthentication und NSFileProtectionNone. Wann welche zum Einsatz kommt, richtet sich nach den Bedingungen, unter denen eine Datei zugänglich sein soll:

1. `NSFileProtectionComplete:`
   Der Klassenschlüssel wird mit einem Schlüssel, der aus dem Benutzer-Passcode und der UID des Gerätes abgeleitet wird, geschützt. Bei der Sperrung eines Gerätes wird der entschlüsselte Klassenschlüssel kurze Zeit nach dem Sperren (nach 10 Sekunden, wenn die Einstellung bei *„Code anfordern“* *„Sofort“* lautet) verworfen, somit sind alle Daten dieser Klasse unzugänglich, bis der Nutzer das Gerät wieder entsperrt.

2. `NSFileProtectionCompleteUnlessOpen:`
   Dateien, die während des Sperrens des Gerätes geschrieben werden müssen, beispielsweise E-Mail-Anhänge, können durch die Verwendung der Elliptische-Kurven-Kryptografie (ECDH mit Curve25519) im Hintergrund geladen werden.

3. `NSFileProtectionCompleteUntilFirstUserAuthentication:`
   Diese Klasse verinnerlicht dieselben Aspekte wie `NSFileProtectionComplete` bis auf den Unterschied, dass der entschlüsselte Klassenschlüssel bei der Sperrung des Gerätes nicht verworfen wird. Apps von anderen Anbietern wird diese Klasse standardmäßig zugewiesen, wenn diese nicht explizit eine andere Dateisicherheitsklasse gewählt haben.

4. `NSFileProtectionNone:`
   Der Klassenschlüssel wird nur mit der UID verschlüsselt und im *Effaceable Storage* abgelegt.

Der *Per-File-Key* obliegt während der kompletten Verarbeitung dem Secure Enclave und wird zu keinem Zeitpunkt dem Anwendungsprozessor offenbart. Bei der ersten Inbetriebnahme von iOS, nach einer Installation oder wenn die Daten vom Benutzer

vollständig gelöscht wurde, werden die Metadaten des Dateisystems mit einem zufälligen Schlüssel verschlüsselt und im *Effaceable Storage* abgelegt. Die Intention dahinter ist das schnelle Löschen auf Anforderung z. B. durch den Benutzer (Option *„Inhalte & Einstellung löschen“*) oder der Befehl zum Fernlöschen. Nach dem Löschen des Schlüssels sind alle Dateien auf dem Gerät kryptografisch unzugänglich. [1]

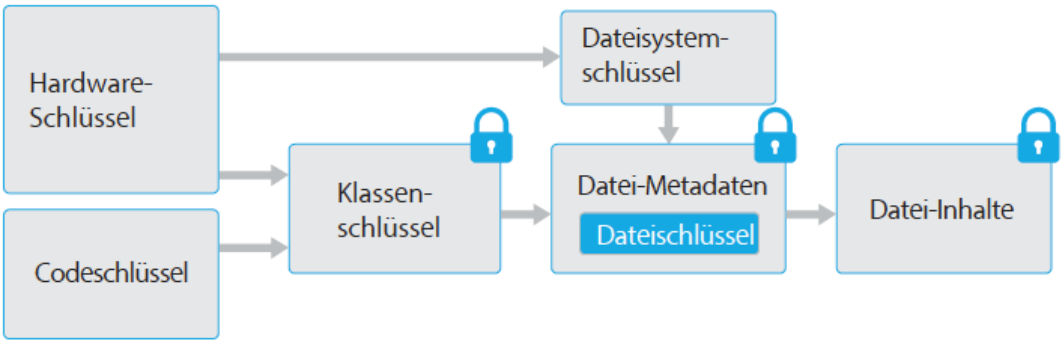

Abbildung 2.4: [1, S.12]

- Dateischlüssel (*Per-File-Key*) erstellen und damit Dateiinhalte verschlüsseln.
- Dateischlüssel mit Klassenschlüssel verpacken und in die Datei-Metadaten speichern.
- Klassenschlüssel mit einem von vier möglichen Dateisicherheitsklassen schützen.

Diese Anordnung (siehe Abb. 2.4)) bietet Flexibilität und Effizienz. Wird die Klasse einer Datei geändert, so muss nur der Dateischlüssel neu verpackt werden, bei Veränderungen am Code wird der Klassenschlüssel neu verpackt.

User Keybags

Die von den Dateisicherheitsklassen erzeugten Schlüssel, App-Passwörter oder auch kurze vertrauliche Datensätze (z. B. Schlüssel und Anmelde-Tokens) werden im iOS-Schlüsselbund (User-Keybags) gespeichert. Bei dem Schlüsselbund handelt es sich um eine SQLite-Datenbank, welche im Dateisystem gespeichert wird und nur einmal existiert. [1]

2.3.3 Gerätecodes

Durch die Festlegung eines Codes wird die Datensicherheit auf dem Gerät aktiviert. Der Code kann sowohl aus (6 oder 4) Ziffern als auch einer alphanumerischen Form beliebiger Länge bestehen. Abgesehen vom Entsperren des Gerätes stellt dieser die Entropie (siehe 2.3.3) für bestimmte Verschlüsselungscodes bereit. Sollte das Gerät einem Angreifer in die Hände fallen, so wird diesem der Zugriff auf die Daten bestimmter

Sicherheitsklassen ohne Gerätecode verwehrt. Je sicherer ein Gerätecode gewählt worden ist, desto sicherer ist auch der Verschlüsselungscode. Mit Touch-ID (siehe 2.1.5) wird dem Benutzer eine Möglichkeit geboten, um einen um ein Vielfaches stärkeren Gerätecode einzurichten, welcher gleichzeitig benutzerfreundlich ist. Dies schlägt sich in der Erhöhung der Effektivität der Entropie nieder, mit der die für den Datenschutz verwendeten Verschlüsselungsschlüssel geschützt werden. [1]

Entropie

Die Entropie beschreibt den Informationsgehalt und die Zufälligkeit einer Quelle. In einem kryptografischen System sollte der Angreifer möglichst wenige Informationen über die Quelle erhalten, eine Quelle mit großer Entropie. Die Entropie aus der Quelle eines vier- und sechsstelligen Zifferncodes, welcher aus Dezimalzahlen besteht, ist wesentlich geringer als die eines alphanumerischen Codes aus Klein- und Großbuchstaben des Alphabets sowie Dezimalzahlen und Sonderzeichen.

Der mittle Informationsgehalt einer Quelle wird anhand der Wahrscheinlichkeit ihres Auftretens gewichtet und aufsummiert.

$$H(p_1, ..., p_n) = -\sum_{i=1}^{n} p_i \log(p_i)$$

H wird als Entropie der Quelle bezeichnet und misst den Informationsgehalt, der im Durchschnitt erhalten wird, wenn die Quelle von Angreifern beobachtet wird. Die Entropie ist damit ein Maß für die Unbestimmtheit der Quelle. [7, S. 317]

2.4 Sicherheitslücken

Sicherheitslücken oder Schwachstellen beschreiben die Verwundbarkeit eines IT-Systems.

> *„Ein IT-System ist ein geschlossenes oder offenes, dynamisches technisches System mit der Fähigkeit zur Speicherung und Verarbeitung von Informationen. “* [7, S.3]

Mit der Verwundbarkeit wird das Risiko beschrieben, dass eine Schwachstelle durch einen Angreifer ausgenutzt wird, indem dieser die Sicherheitsdienste eines Systems umgeht, täuscht oder modifiziert. Dadurch kann es zur Beeinträchtigung der Datenintegrität, Informationsvertraulichkeit oder auch Verfügbarkeit des IT-System kommen. Dies erreichen Angreifer durch Schadsoftware wie Malware oder auch Exploits (siehe 2.4.1, 2.4.3). [7]

2.4.1 Malware

Malware bezeichnet ein Schadprogramm, welche von Angreifern unerwünschte oder schädliche Funktionen ausführt und damit einem Computersystem Schaden zufügt. Die Malware-Familie besteht unter anderem aus Viren, Trojanern, Keyloggern und Spyware. Belästigende Malware mit Erpressungsversuchen, sind Programme, die „Ransomware“ genannt werden. Jede Malware unterscheidet sich in Bezug auf den Schaden, den diese verursachen kann, und hat eine andere schädliche Zielsetzung. Malware verbreitet sich typischerweise durch die Ausnutzung von Sicherheitslücken in Betriebssystemen, einer Anwendungssoftware, einem Treiber oder anderem Programmcode. [8]

2.4.2 Spyware

Spyware sind Programme, die ohne Wissen des Benutzers auf einem Computersystem oder mobilen Gerät installiert und aktiviert werden. Harmlosere Spyware zeichnet das Surf-Verhalten eines Nutzers auf und übermittelt die gesammelten Daten über das Internet an interessierte Dritte, damit z. B. (vermeintlich) zum Nutzer passend Werbung angezeigt werden kann. Zähere Spyware-Programme protokollieren jegliche Benutzereingabe über Peripheriegeräte zum Missbrauch („Keylogger“). So können die Benutzereingaben bezüglich eines Onlineeinkaufes oder einer Anmeldung auf Internetkonten gespeichert werden. Andere Programme öffnen ungefragt Webfenster oder manipulieren Suchanfragen, damit sich bestimmte Webseiten öffnen. [9]

2.4.3 Exploit

Ein Exploit ist eine Art Schadprogramm, welches Sicherheitslücken von Anwendungsprogrammen ausnutzt, um zur Manipulation des Computersystems beizusteuern wie durch das erhalten der Administratorenrechte. Es gibt zwei Infizierungswege: zum einen den Besuch einer Webseite, welche einen Exploit-Code enthält, zum anderen das Öffnen von Dateien, die einen transparenten Exploit-Code beinhalten. Die häufigste Infizierung

eines Systems erfolgt per Spam oder Phishing-Mails. Phishing bezeichnet den Versuch, über gefälschte Webseiten, E-Mails oder Kurznachrichten die persönlichen Daten eines Internet-Benutzers auszuspähen und damit einen Identitätsdiebstahl zu begehen. [10]

2.4.4 Schwachstellentypen

Es gibt verschiedene Typen von Schwachstellen, die in Kategorien unterteilt werden.

TYP	ANZAHL
Bypass Something	107
Gain Information	130
Overflow	352
Memory Corruption	360
Execute Code	419
Denial of Service	458

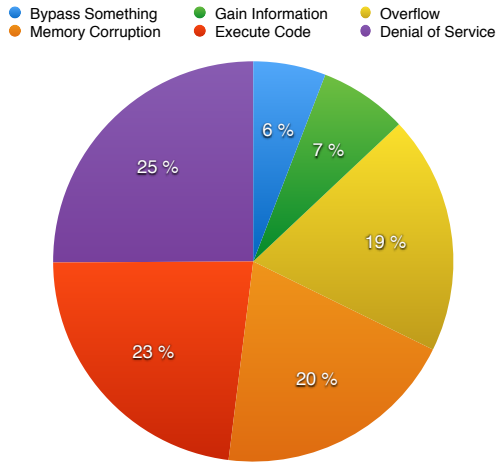

Abbildung 2.5: Sicherheitslückentypen in iOS (2010 - 2016) [11]

Die Abbildung 2.5 zeigt die verschiedenen Schwachstellentypen, welche bis dato im Betriebssystem iOS aufgetreten sind.

Denial of Service (DoS)

Das Ziel von DoS-Attacken liegt darin, einen Dienst unbrauchbar zu machen. Bei diesem Dienst kann es sich um einen Webservice wie einen E-Mail-Server oder eine Website handeln, die von Angreifern gezielt mit E-Mails bzw. Anfragen bombardiert wird. Die Folge ist, dass die Server unter der Last einbrechen und ihr Dienst dann nur noch teilweise oder gar nicht mehr verfügbar ist. [12]

Memory Corruption

Eine Memory-Corruption (Speicherkorruption) tritt auf, wenn durch einen Programmierfehler ein Speicherbereich unabsichtlich modifiziert wird. Greift das Programm auf diesen fehlerhaften Bereich zu, kann dies zu einem unvorhersehbaren Programmverhalten oder zum Absturz des Programms führen. Dies ist eine der häufigsten softwarebedingten Schwachstellen. Die Memory-Corruption wird in verschiedene Kategorien unterteilt, eine häufig auftretende ist der Speicherüberlauf (Overflow). [13]

> *„Viele Angriffe beginnen heute mit einer Memory-Corruption, die einen ersten Halt für weitere Infektion bietet.“* [14]

Overflow

Speicherüberläufe gehören zu den häufigsten Sicherheitslücken, mit ihnen versuchen Angreifer, Schadsoftware auf das jeweilige Gerät zu schleusen. Es gibt verschiedene Arten von Speicherüberläufen, eine sehr bekannte und häufig auftretende Art ist der Pufferüberlauf (Buffer-Overflow). Hier wird der Puffer mit unsinnigen Werten überschrieben, was dazu führt, dass Variablen oder Rücksprungadressen nicht mehr stimmen, was zum Absturz des Programms führt. Ein weitaus gefährlicheres Szenario ist, wenn die Rücksprungadresse auf einen vom Angreifer speziell präparierten Speicherbereich verweist. Hier kann der Angreifer dann seinen gewünschten Code hinterlegen und hat so einen Exploit entwickelt. [15]

Execute-Code

Bei diesem Schwachstellentyp haben Angreifer die Möglichkeit, einen beliebigen Befehl (Code) auf dem Zielgerät auszuführen, daher auch als *„Arbitrary Code-Execution“* (Ausführen von beliebigem Code) bekannt. Die meisten Schwachstellen dieser Art erlauben es dem Angreifer, einen Maschinen-Code auszuführen. Durch das Einschleusen (Injection) und Ausführen von Shellcodes haben die Angreifer eine einfache Möglichkeit, einen beliebigen (arbitrary) Code auszuführen. Die Fähigkeit, einen beliebigen Code aus der Ferne auszulösen, ist als *„Remote Code-Execution“* bekannt. [16]

2.4.5 CVE und CVSS

Um das Risko und den damit verbundenen möglichen und tatsächlichen Schweregrad einer Schwachstelle zu bewerten, kommt unter anderem der freie und offene Industrie-Standard Common Vulnerability Scoring System (CVSS) zum Einsatz. Dieser Standard wird anhand von verschiedenen Kriterien, den sogenannten Metrics, berechnet. Der Schweregrad wird auf einer Punkteskala von 0 bis 10 angegeben, wobei 10 die höchste Gefährdung darstellt, dadurch soll dem Benutzer die Einschätzung der Gefährlichkeit erleichtert werden. [17]

Common Vulnerabilities and Exposures (CVE) ist ein Standard für die einheitliche Kennzeichnung von Sicherheitslücken oder Schwachstellen in Computersystemen. Der Informationsaustausch zwischen den einzelnen Herstellern wird dadurch erleichtert. Die CVE-IDs werden aus dem Jahr, in welchem die jeweilige Schwachstelle entdeckt wurde, und einer fortlaufenden Nummer gebildet. [18]

3
Sicherheitsanalyse

3.1 Systemsicherheit und Risiken

Durch den rapiden Anstieg der Zahl der Smartphonenutzer in Deutschland rückt das Thema Sicherheit immer weiter in den Vordergrund. Bereits 76 % aller Bundesbürger (Personen ab 14 Jahren) verfügen im Juli 2016 über ein Smartphone, vor gut zwei Jahren (2014) waren es erst 55 %. An den Absatzzahlen wird deutlich, dass die Zahl der Benutzer weiter steigen wird (siehe Abb. 3.1). Parallel zum Absatz steigt die Gefahr, dass die Geräte zu potenziellen Zielen von Angreifern werden. Der physische Verlust eines Smartphones und der damit verbundene Datenverlust stellen eine große Gefahr dar. Auf dem heutzutage nicht unüblichen 16/32-GB-Speicher lassen sich Projektdaten jeglicher Art und E-Mails von vielen Jahren im vollem Umfang speichern. Darüber hinaus sind dort in der Regel Zugangsdaten verschiedenster Art, z. B. zu Firmennetzwerken, E-Mail-Konten und zum Online-Banking gespeichert. Dies macht das Smartphone für Phishing-Angriffe attraktiv. Ein unzureichend gesicherter Datenaustausch über Schnittstellen wie USB, WLAN und Bluetooth kann zur Entwendung entsprechender Daten führen. Der Datenverlust ist eine große Bedrohung, wovon nicht nur Privatpersonen betroffen sind. Den größeren (finanziellen) Schaden richtet Datendiebstahl in der Industrie an. In einer repräsentativen Umfrage des Digitalverbands Bitkom (2016), bei der 504 Unternehmen befragt wurden, ergab sich, dass die häufigste Ursache für Datenklau mit 32 % der Diebstahl von IT- oder Telekommunikationsgeräten ist (siehe Abb. 3.2). [19] [20]

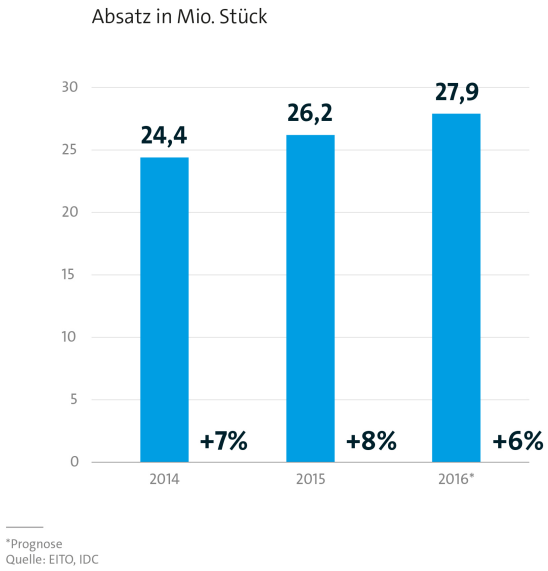

Abbildung 3.1: Smartphone Absatz [19]

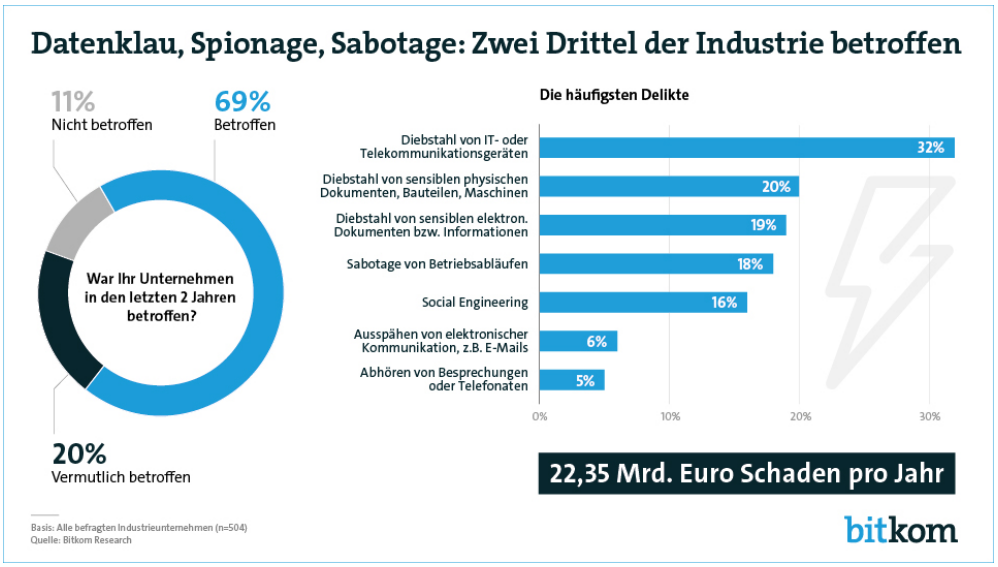

Abbildung 3.2: Industrie Datendiebstahl [20]

3.1.1 iOS Up-to-Date

Die schnellstmögliche Installation von neuen Sicherheits-Updates ist eine gute Möglichkeit, um potenzielle Schwachstellen im System zu verringern und dadurch einen Angriff zu erschweren. Daher stellt Apple in regelmäßigen Abständen Sicherheitsupdates zum Download bereit, denn die neueren Versionen eines Betriebssystems sind generell sicherer als ältere. Dann ist es jedoch die Aufgabe des Benutzers, diese auch zu installieren.

Am 13. September 2016 veröffentlichte Apple das Betriebssystem-Update iOS 10 und stellte dieses den Benutzern zum Download bereit. Das Bemerkenswerte daran war, dass bereits nach 16 Tagen mehr als die Hälfte aller iOS-Benutzer von älteren Versionen zum neuen System gewechselt ist (siehe Abb. 3.3).

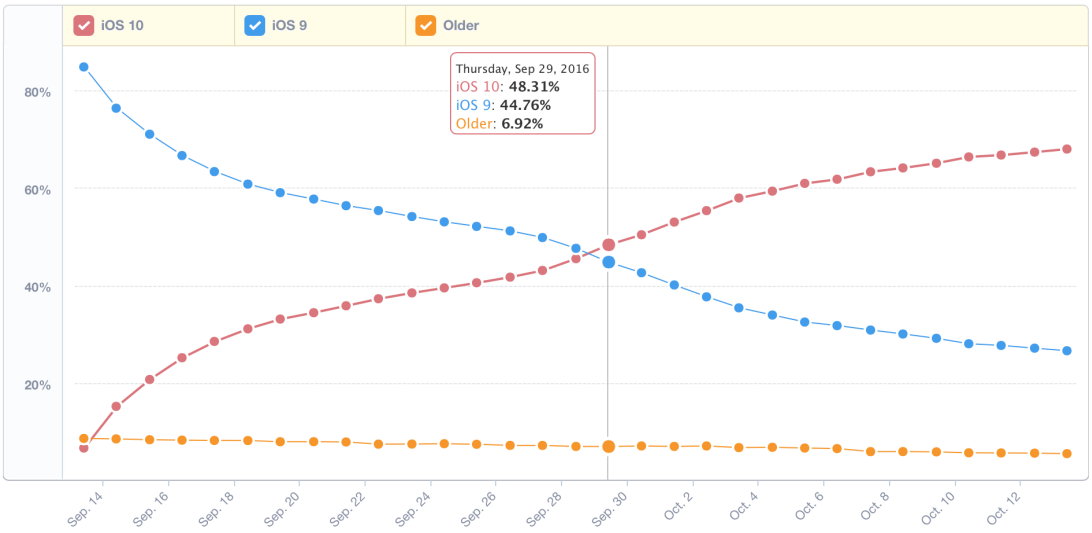

Abbildung 3.3: iOS 10 Updateverhalten 13. SEP - 13. OKT 2016 [21]

Die Gründe für das verglichen mit Android schnelle Wechseln der Benutzer liegt zum einem an der standardisierten Hardware und der breiten Kompatibilität mit neuen Versionen. Nur ältere Hardware wie die Modelle iPhone 4 und 4S, welche aber nur noch 3,8 % des gesamten Marktanteiles der Modelle ausmachen, werden nicht mit neueren Versionen der Betriebssysteme versorgt und haben dementsprechend wesentlich mehr Schwachstellen. Bei der Versorgung mit Updates hat iOS Vorteile gegenüber der Konkurrenz wie beispielsweise Android: Bei Android kommt es durch die große Zahl von Hardware-Variationen zu einer Verzögerung bei der Verfügbarkeit von Updates, da diese zunächst für die jeweilige Hardware-Plattform angepasst werden müssen. Jährliche Major Updates, welche das Betriebssystem nicht nur mit neuen Sicherheits-Updates versorgen, erweitern das iPhone um neue Funktionen. Diese ermutigen die Benutzer des iPhones dazu, die Updates schneller zu installieren. Der Prozess zur Systemsoftwareautorisierung, welcher im Kapitel 2.1.2 erläutert wurde, leistet ebenfalls einen Beitrag dazu, die

Zahl der alten Versionen gering zu halten. Durch diesen wird das Aufspielen älterer Softwareversionen verhindert bzw. erschwert. Dem Update-Lebenszyklus sind jedoch Grenzen gesetzt, ab einem gewissen Punkt werden ältere Geräte nicht mehr mit aktuellen Sicherheitsupdates versorgt. Spätestens dann sollte ein Benutzer einen Wechsel des Gerätes in Erwägung ziehen. [22] [23]

3.1.2 Walled Garden und Jailbreak

Walled Garden

Der Walled Garden ist eine Metapher für ein Technologiekonzept, bei dem der Hersteller – in diesem Fall Apple – die Kontrolle über vom iPhone ausgeführte Software, Medien und andere Inhalte behalten möchte. Dafür gestaltet der Hersteller mittels Digitaler Rechteminderung (DRM) entsprechende Funktionen (siehe 2.2), damit nur autorisierte bzw. signierte Inhalte auf dem Gerät nutzbar sind. Daraus resultieren gleich zwei große Vorteile für den Hersteller. Zum einen wird ein besserer Schutz vor Schadsoftware (Malware etc.) gewährleistet, da vor jeder App-Veröffentlichung diese im App Store geprüft (Review) wird. Doch der wahrscheinlich für den Hersteller weitaus wichtigere Punkt/Grund ist die Eindämmung von Softwarepiraterie [24]. So ist es bei iOS im Gegensatz zu Android nicht möglich, eine Applikation aus einer unbekannten Quelle (z. B. Webseiten, Filehoster) herunterzuladen und auf dem Gerät zu installieren. Die auf diversen Internetplattformen bereitgestellten illegalen Kopien werden oft von Angreifern zur Verbreitung von Schadsoftware verwendet. Für diese ist es ein Leichtes, die Kopien um einen schädlichen Code zu erweitern. Nichtsdestotrotz stößt das Geschäftsmodell von Apple nicht bei allen Nutzern auf breite Zustimmung. Nicht jeder ist gewillt, sich die Nutzung seines Eigentums vorschreiben zu lassen. Aus diesem Grund greifen einige Benutzer auf einen Jailbreak zurück, um sich aus der Bevormundung durch Apple zu lösen. [25]

Jailbreak

Ein Jailbreak (Gefängnisausbruch) bezeichnet die unautorisierte Entfernung von Nutzerbeschränkungen, mit welchen der Hersteller bestimmte Funktionen (z. B. Apps aus unbekannter Herkunft Installieren) serienmäßig entfernt hat. Sicherheitslücken im Gerät dienen dazu, die Sicherheitsmechanismen des Betriebssystems auszuhebeln, um einen Jailbreak zu installieren, durch den der Nutzer Angreifern die gleichen Möglichkeiten zur Manipulation des Systems ermöglicht.

Bei einem Jailbreak wird in den meisten Fällen die Softwareverwaltung Cydia installiert. Dieses ist ein Pendant zum App Store und wird als App auf dem Homescreen angezeigt. Cydia ermöglicht, Modifikationen und Programme zu installieren, und dabei werden sicherheitsrelevante Aspekte wie das Sandboxing (siehe 2.2.2) und die Applikations-Codesignierung (siehe 2.2.1) umgangen. Dem Nutzer ist es über die sogenannten Repos möglich, nicht signierte Applikationen oder Modifikationen am System vorzunehmen

oder zu installieren. Für manuell hinzugefügte Repositories (Repos), insbesondere solche mit illegalen Inhalten wie Raubkopien, besteht eine erhöhte Gefahr, Schadsoftware zum Opfer zu fallen. [26]

3.1.3 Antivirenprogramm

Bei Antivirenprogrammen wie Kaspersky Internet Security oder Avira Free Antivirus handelt es sich um Software, deren Hauptzweck darin besteht, Viren, Würmer und Trojaner ausfindig zu machen und diese gegebenenfalls zu blockieren oder zu entfernen. Entsprechende Antivirenprogramme sind auch im iOS-System vorhanden, nur bieten diese einen sehr geringen Mehrwert für den Nutzer, da die Antivirenprogramme stark durch das Sandboxing (siehe 2.2.2) des Systems behindert werden. Dieses schirmt alle Applikationen und Systemressourcen vom Zugriff durch andere Applikationen ab. Aus diesen Grund sind sie weder dazu in der Lage, das System nach Malware zu scannen noch andere Apps und die Daten dieser zu überprüfen. Dies war unter anderem ein Grund dafür, weshalb Apple im März 2015 alle Antiviren-Applikationen aus dem App Store verbannt hat. Ihre Existenz widersprach sozusagen dem Versprechen, dass iOS ein sicheres Betriebssystem sei. Vielmehr sorgt Apple selbst dafür, den App Store von Malware sauber zu halten. [27]

3.1.4 Datenschutz mit Secure Enclave Prozessor

Der Schutz von Benutzerdaten ist ein essentieller Bestandteil von iOS und dies wird durch die in den Grundlagen vorgestellte Data-Protection (siehe 2.3.2) und den Secure-Enclave-Prozessor (siehe 2.1.3) erreicht. Die Benutzerdaten werden durch einen kryptografisch starken *Master-Key*, welcher vom Gerätecode des Benutzers abgeleitet wird, geschützt. Die im User-Keybag enthaltenen Schlüssel werden ebenfalls mit dem *Master-Key* verpackt (wrapped).

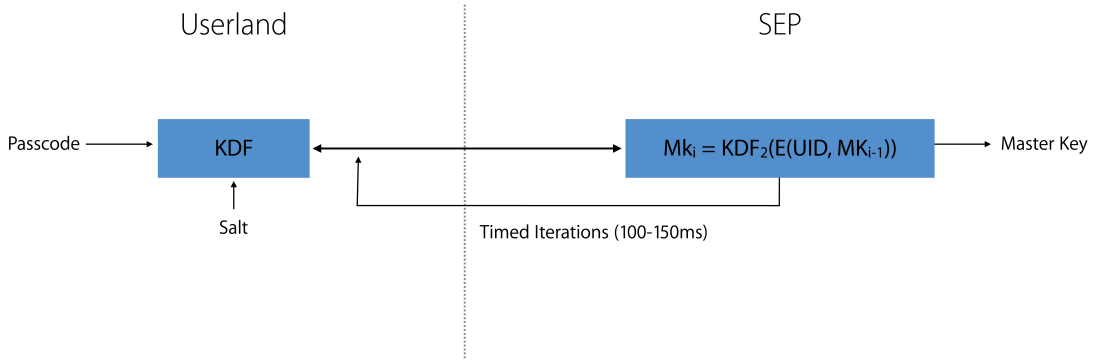

Abbildung 3.4: Ableitung des Master Key durch den Passcode

Wie in der Abbildung 3.4 zu sehen ist, wird beim Erzeugen des *Master-Key* zunächst der Passcode vom Benutzer mit einer zufällig gewählten Zeichenfolge (auch als Salt bezeichnet), in die Key-Derivation-Function (KDF) eingegeben. Der Salt wird verwendet, um die Entropie der Eingabe zu erhöhen. Das Ergebnis aus dem KDF wird dann dem SE übergeben, welcher kontinuierlich über diese iteriert, mit der UID des Gerätes. Die Zahl der Iterationen ist an eine zeitliche Vorgabe gebunden, welche mindestens 100 bis 150 ms beträgt. So wird sichergestellt, dass die Ableitung des Schlüssels eine gewisse Zeit benötigt. [28]

Die Schwachstelle in diesem System könnte nun der Gerätecode sein, da Benutzer häufig Passwörter benutzen, welche deutlich weniger Entropie aufweisen als für die Verwendung von kryptografischen Funktionen benötigt. Um hier Attacken wie einem Brute Force (rohe Gewalt) den Riegel vorzuschieben, wird eine Limitierung an Passworteingabeversuchen verwendet. Nach zehn falschen Gerätecodeeingaben erlaubt der SE keine weiteren Versuche mehr. [28]

Jede Datei in der Benutzerpartition wird mit einem eindeutigen und zufälligen Schlüssel (*Raw File-Key*), welcher vom SE erzeugt wird, verschlüsselt. Die *Raw File-Keys* werden dem Anwendungsprozessor niemals offenbart, sondern mit einem Schlüssel aus dem User-Keybag für die Langzeit-Aufbewahrung verpackt. Zudem werden die *Raw File-Keys* während der Verwendung mit einem Ephemeral Key (kurzlebiger Schlüssel), welcher

an eine Sitzung geknüpft ist, verpackt. Sobald das Gerät neu gestartet wird, endet die Sitzung und damit die Gültigkeit des Schlüssels, dann wird ein neuer Sitzungsschlüssel erzeugt. [28]

Die Abbildung 3.5 zeigt den Ablauf der Dateientschlüsselung. Zunächst wird beim

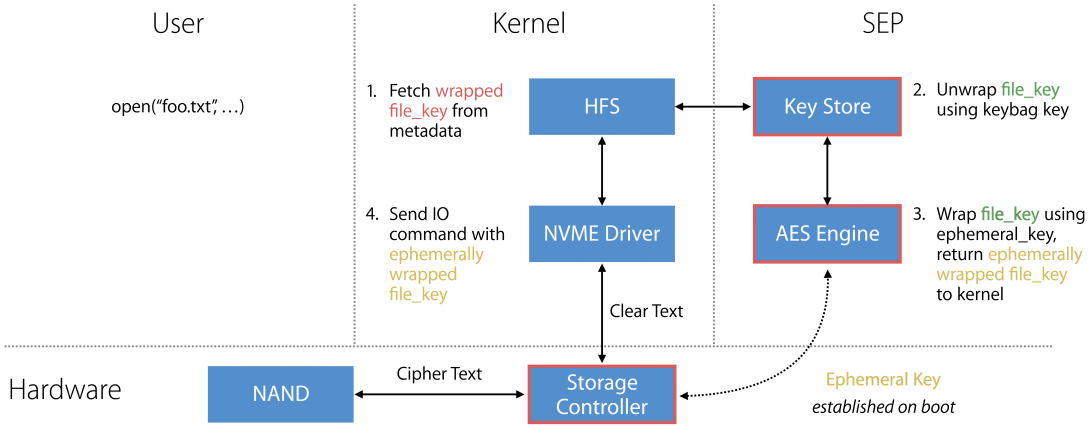

Abbildung 3.5: Dateiverschlüsseln mit der Data Protection

Startvorgang ein Ephemeral Key von der AES-Engine und dem SE erstellt und über einen sicheren Kanal an den Storage-Controller (Speicher-Controller) gesendet, welcher für den NAND (eine bestimmte Art von Flash-Speicher) zuständig ist. Nachdem das Gerät gestartet wurde und der Benutzer eine Datei zu öffnen versucht, wird zuerst der verpackte Dateischlüssel (*File-Key*) vom Kernel aus den Metadaten geholt und an den SE gesendet. Der SE entpackt den Dateischlüssel mit einem Schlüssel aus dem User-Keybag und verpackt diesen wieder mit dem *Ephemeral Key*. Der nun mit dem *Ephemeral Key* verpackte Dateischlüssel wird zurück an den Kernel gesendet, von dort wird der Schlüssel an den Storage-Controller weitergesendet. Dieser kann nun den verpackten Dateischlüssel entpacken, um so den Geheimtext im NAND zu entschlüsseln und danach den Klartext an den Kernel zu senden. Das Bedeutsame an dieser Art von Dateisystemverschlüsselung ist, dass kein Schlüssel, welcher über eine längere Gültigkeit verfügt, dem Kernel offenbart wird. [28]

3.2 Mobile Betriebssysteme im Vergleich

Nachdem zuvor ein Überblick über die wichtigsten Sicherheitskomponenten von iOS vermittelt worden ist, werden diese nun zu anderen mobilen Betriebssystem-Plattformen ins Verhältnis gesetzt. Im Jahr 2015 wurden weltweit mehr als 1,4 Milliarden Smartphones verkauft. Fünf von sechs verkauften Geräten verwenden das mobile Betriebssystem Android. Dies schlägt sich in der Marktaufteilung bezüglich der mobilen Betriebssysteme nieder. [29]

Quartal	Android	iOS	Windows Phone	Andere
2015Q3	84,3 %	13,4 %	1,8 %	0,5 %
2015Q4	79,6 %	18,6 %	1,2 %	0,5 %
2016Q1	83,4 %	15,4 %	0,8 %	0,4 %
2016Q2	87,6 %	11,7 %	0,4 %	0,3 %

Tabelle 3.1: Weltweite Smartphone BS Marktaufteilung [30]

Das derzeit mit Abstand am häufigsten (siehe Tabelle 3.1) verwendete Smartphone-Betriebssystem (87,6 %) ist *Android*, dies liegt an der großen Zahl der Smartphone-Hersteller (Samsung, Huawei, OPPO, vivo) die auf dieses frei verfügbare und quelloffene Software-System setzen. [31] Mit weitem Abstand folgt iOS (11,7 %) und dahinter – mit einem verschwindend geringen Anteil von 0,4 % – *Windows Phone*.

iOS und *Windows Phone* verfolgen zugleich einen proprietären Ansatz im Hinblick auf den Bezug von Soft- und Hardware. Dies bedeutet, dass bezogen auf die Software von diesen, im Gegensatz zu *Android*, keine Rechte und Möglichkeiten zur Wieder- und Weiterverwendung durch Nutzer und Dritte bestehen. Der hohe Verbreitungsgrad eines Betriebssystems erregt naturgemäß die Aufmerksamkeit von Hackern und Cyberkriminellen, denn je populärer eine Plattform ist, umso häufiger wird diese Opfer von Angriffen durch Computerviren, Trojaner und dergleichen. Ein Betriebssystem, welches eine geringe Verbreitung aufweist, ist für Angreifer wesentlich uninteressanter, bis zu dem Zeitpunkt, wenn das Betriebssystem an Popularität gewinnt und so ins Fadenkreuz der Angreifer gerät [32]. Daher sollte der Schutz von Nutzerdaten bei mobilen Geräten die höchste Priorität genießen. Dies wird bei *iOS*, *Android* und *Windows Phone* durch bereits im Gerät integrierte Sicherheitsmechanismen bewerkstelligt. Darüber hinaus verfolgen die Plattformbetreiber verschiedene Philosophien hinsichtlich der Transparenz ihrer Sicherheitsmerkmale. [7, S. 92]

Nachfolgend werden einige Sicherheitsmerkmale in mobilen Betriebssystemen aufgelistet:

- ***Access-Control***
  Diese Funktion dient der Zugriffskontrolle auf den Endgeräten, typischerweise ist dies die erste Verteidigungslinie gegenüber Angreifern bei einem Verlust des

Gerätes. Je nach Betriebssystem können hierzu PIN, Passwort, Sperrmuster oder biometrisches Passwort (Fingerabdruck, Augeniris) verwendet werden. Eine Kombination dieser Zugriffsarten ist ebenfalls möglich. Inwiefern die Zugriffskontrolle Einfluss auf andere Sicherheitsmerkmale hat, ist vom System abhängig (siehe 2.3.2).

- ***Sandboxing***
  Mobile Betriebssysteme verwenden Sandboxing, um Applikationen voneinander und vom Kernel zu isolieren (siehe 2.2.2). Dadurch wird z. B. für Malware der Zugriff auf einen gemeinsamen/entfernbaren Speicher erschwert. [33]

- ***Remote-Wipe***
  Diese Funktion ermöglicht dem Netzwerk-Administrator oder Besitzer des Smartphones, bestimmte Daten aus der Ferne zu löschen. Was Remote-Wipe letztendlich zu bewerkstelligen vermag, hängt vom spezifischen Betriebssystem (siehe 2.3.1) und der auf dem Gerät verwendeten MDM-Software ab [34]. Für den Benutzer stellen die Hersteller verschiedene Applikationen (iOS: Find my iPhone) zu Verfügung, um dieses Feature verwenden zu können. [35]

- ***Full-Device-Encryption***
  Die Geräteverschlüsselung dient der Sicherheit von gespeicherten Daten auf einem Gerät. Der Schutz von verschlüsselten Daten steht im direkten Zusammenhang mit dem Benutzer-Passwort. Konnte ein Angreifer das Passwort knacken, so werden diesem die Daten offengelegt. Entwickler haben dafür zu sorgen, alle Dateien, die in den Flash-Speicher geschrieben werden, zu verschlüsseln (siehe 2.3.2). [36, S. 14].

- ***Hardware-based Security (System-on Chip - SoC)***
  Dies beschreibt eine Technologie, bei welcher eine Hardware-basierte Sicherheit in den SoC integriert ist. Diese sorgt für einen vertrauenswürdigen Startvorgang des jeweiligen Betriebssystems. Zu ihren Aufgaben gehören unter anderem der Schutz von Authentifizierungsfunktionen, Kryptografie und Schlüsselverwaltung. Diese Technologie ist bei Apple unter dem Namen Secure Enclave (siehe 2.1.3) bekannt. Der Chipdesigner ARM bietet diese Funktionalität mit dem Modell ARM® TrustZone® an.

- ***Secure Application Provenance only***
  Ist ein Restriktiver Ansatz zur Installation von Applikationen, welche durch den Softwareanbieter geprüft wurden. Es können nur signierte Applikationen aus vertrauenswürdiger Quelle installiert werden (App Store, Windows Store). Ein anderer Ansatz basiert auf der freien Installation von unsignierten Applikationen aus jeglicher Quelle, dies fördert die Entwicklung von Malware. [33]

Sicherheitsmerkmal	Android	iOS	Windows Phone
Access-Control	✓	✓	✓
Sandboxing	✓	✓	✓
Remote-Wipe	✓	✓	✓
Full-Device-Encryption	✓	✓	✓
Hardware-based Security (SoC)	(✗)	✓	(✗)
Secure Application Provenance only	✗	✓	✓

Tabelle 3.2: Sicherheitsmerkmale in mobilen Betriebssystemen [33] [36] [37]

Die Tabelle 3.2 veranschaulicht, dass die Geräte viele Gemeinsamkeiten bei den Sicherheitsaspekten haben. *Zugriffskontrolle*, *Sandboxing*, *Remote-Wipe* und *Geräteverschlüsselung* sind bei allen drei Herstellern, auch wenn es bei diesen Funktionen Unterschiede gibt, standardmäßig vorhanden. iOS hat bei jedem Gerät ab dem iPhone 5S einen Hardware-basierten Schutz integriert, bei Android und Windows Phone ist dies nicht der Fall. Der Schutz kann aber unter der Voraussetzung, dass ein ARM-Prozessor mit der entsprechenden Architektur verwendet wird, genutzt werden.Doch letztendlich ist die Software von Windows Phone und Android nicht explizit auf dieses Feature hin konzipiert, so wie dies bei iOS der Fall ist. Die ausschließliche Installation von Software aus vertrauenswürdigen Quellen kennzeichnet die Sicherheitsphilosophie eines Unternehmens. Aber die geringe Zahl von Malware auf iOS-Geräten zeigt, dass diese Strategie nicht die schlechteste ist.

3.3 Sicherheitslücken in Mobilen Betriebssystemen

Wie im vorherigen Kapitel ersichtlich wurde, basiert das Interesse der Angreifer an der Entwicklung von Malware unter anderem auf der Popularität eines Betriebssystems. Die steigende Zahl der Nutzer von Smartphones (siehe 3.1) und anderen mobilen Geräten spiegelt sich in der steigenden Zahl von neu entdeckten Sicherheitslücken wider (siehe Tabelle 3.3).

2013	**2014**	**2015**
127	168	528
-	32 %	214 %

Tabelle 3.3: Neu entdeckte Sicherheitslücken in mobilen Geräten [38, S. 9]

Daraus resultiert, dass Cyberkriminelle mehr Zeit in anspruchsvolle Attacken, zur Entwendung von persönlichen Daten oder zur Erpressung von Geld, investieren. Obwohl die Benutzer von Android das Hauptziel von Malware-Entwickler bleiben , gibt es seit 2012 eine steigende Zahl von effektiven Attacken gegen iOS, auch ohne dass ein Jailbreak (siehe 3.1.2) benötigt wurde, um das jeweilige Gerät zu kompromittieren. [11]

3.3.1 iOS Sicherheitslücken

Die iOS-Plattform bleibt ebenso wenig von Schwachstellen verschont wie andere mobile Betriebssysteme: In den Jahren 2013 bis 2015 wurden die meisten Sicherheitslücken auf mobilen Betriebssystemen bei iOS entdeckt. [38, S. 11]

WebKit

Häufig treten Sicherheitslücken im WebKit der Web-Engine von iOS auf, allein im Jahr 2015 wurden über 100 Schwachstellen von Apple durch Updates geschlossen. Web-Engines enthalten viele Schwachstellen, wie Pufferüberläufe (siehe 2.4.4), Memory-Corruption (siehe 2.4.4) und andere. Diese Schwachstellen können von Angreifern für verschiedene Zwecke missbraucht werden, von Malware-Installationen bis hin zur vollständigen Kontrolle über das Handy. Die Sicherheit im WebKit ist dennoch unterschiedlich, denn Apple verwendet eine eigene Portierung von diesen für iOS, Mac und den App Store. Aktuell (Stand: Oktober 2016) sind sechs verschiedene Portierungen von WebKit verfügbar. Ein Großteil (70 %) des WebKit-Codes ist plattformübergreifend, dennoch ist ein nicht unerheblicher Teil (30 %) des Codes plattformspezifisch. Bei der Verwendung von Safari wird die iOS-Portierung vom WebKit verwendet, dieser wird von Apple regelmäßig mit Sicherheitsupdates versorgt, um Schwachstellen zu schließen. Dies ist bei Weitem nicht bei allen Portierungen der Fall. Des Weiteren müssen auch App-Entwickler das WebKit verwenden. So wird sichergestellt, dass jede iOS-Applikation, welche HTML-Seiten rendert (z. B. Google Chrome, Firefox), auch von den regelmäßigen Sicherheitsupdates profitiert. Die Tabelle 3.4 gewährt einen kleinen Einblick in die Sicherheitslücken im

Betriebssystem iOS. Anhand der Sicherheitslücken *Masque Attack*, *XARA* und *Pegasus* wird exemplarisch gezeigt, in welcher Form diese für einen Angriff verwendet werden können. [39]

Name / betroffene Komponente	**CVE-ID**	**Betroffene iOS-Versionen**
Masque Attack	CVE-2014-4494	7.1.1 bis einschließlich iOS 8.1.2
XARA	CVE-2015-5835	Versionen vor iOS 9
Pegasus	CVE-2016-4655 CVE-2016-4656 CVE-2016-4657	Versionen vor iOS 9.3.5
WebKit	CVE-2016-4759 CVE-2016-4763 CVE-2016-4764 CVE-2016-4765 CVE-2016-4766 CVE-2016-4767 CVE-2016-4768	Versionen vor iOS 10
Kernel	CVE-2016-4771 CVE-2016-4772 CVE-2016-4773 CVE-2016-4774 CVE-2016-4776 CVE-2016-4778	Versionen vor iOS 10
Sanbox	CVE-2016-4620	Versionen vor iOS 10
Safari Reader	CVE-2016-4618	Versionen vor iOS 10
CoreCrypto	CVE-2016-4712	Versionen vor iOS 10

Tabelle 3.4: Sicherheitslücken in iOS [40]

Masque Attack

Der Sicherheitsdienstleister FireEye hat 2014 eine Sicherheitslücke aufgedeckt, welche ermöglicht, aus dem App Store auf iPhone installierte Apps durch böswillige Kopien zu ersetzen. So können Angreifer das Aussehen der Original-Apps nachahmen und so die Zugangsdaten von ihren Opfern entwenden. Darüber hinaus verbleiben die Daten der Original-App im Verzeichnis, auch nachdem diese durch die böswillige Kopie ersetzt wurde. Diese hat dann Zugriff auf die empfindlichen Daten. Um die legitime App zu ersetzen, muss der Angreifer lediglich denselben *Bundle-Identifier* verwenden. Hierfür muss das Opfer jedoch dazu gebracht werden, eine bösartige App von außerhalb des App Stores zu installieren. Dieses funktioniert über ein Firmen-Zertifikat (siehe 2.2.1),

welches Firmen normalerweise zum internen Testen von Apps verwenden. Die *Masque Attack* kann aber nicht systemeinterne Apps wie den Safari Browser ersetzen. [41]

XARA

In iOS arbeiten Apps in einer Sandbox (siehe 2.2.2), das soll verhindern, dass kompromittierte Programme Zugriff auf Informationen anderer Apps oder des Systems haben. Bei der Sicherheitslücke XARA handelt es sich um eine sogenannten *„Cross-App-Resource-Attack“*, diese erlaubt dem Angreifer das Ausspähen von Passwörtern anderer Apps. Ein Team von Sicherheitsexperten der Indiana University Bloomington hat diese Sicherheitslücke 2015 entdeckt und eine Dokumentation darüber veröffentlicht [42]. Ihnen ist es gelungen, Schad-Apps in den App Store einzuschleusen, ohne dass dies bei der Sicherheitsprüfung von Apple bemerkt wurde. Ist eine solche App erst einmal installiert, hat diese die Möglichkeit, an die Passwörter im Keychain zu gelangen. Beim Keychain handelt es sich um einen virtuellen Schlüsselbund, in welchen gespeicherte Passwörter abgelegt werden (siehe 2.3.2). Der Angreifer gelangt aber nicht ohne Weiteres an die Schlüssel im Keychain. Hierzu löscht die Schad-App die im Keychain befindlichen Passwörter-Einträge, um eine Neueingabe durch die Benutzer zu erzwingen. Geben diese das Passwort erneut ein, wird der Schad-App der Zugriff auf dieses ermöglicht.

Pegasus

Pegasus ist eine Spyware, welche, einer Untersuchung von Citizen Lab im Jahr 2016 zufolge, von einer Organisation mit dem Namen NSO Group entwickelt wurde. Die NSO Group ist eine aus Israel stammende Organisation welche sich auf den Cyberkrieg spezialisiert hat. Sie wurde 2010 vom US-amerikanischen Unternehmen Francisco Partner Management übernommen. Die Pegasus-Spyware verwendet gleich eine Reihe von drei Zero-Day-Sicherheitslücken, um das Sicherheitssystem von iOS zu umgehen. Citizen Lab und Lookout bezeichnen dies als *The Trident Exploit Chain*. Erst durch die Verwendung aller drei Sicherheitslücken kann die Spyware ihr ganzes Potenzial entfalten. Dieses basiert aus der Ausnutzung von folgenden Sicherheitslücken: CVE-2016-4655, CVE-2016-4656 und CVE-2016-4657 [43]

1. **CVE-2016-4655**
   Hierbei wird eine Schwachstelle im Kernel ausgenutzt, welche den Angreifern ermöglicht, die Speicher-Adresse des Kernels zu berechnen. Dabei handelt es sich um eine Exploit in der Kernel-Address-Space-Layout-Randomization (KASLR) (siehe 2.2.4).

2. **CVE-2016-4656**
   Eine *Memory-Corruption* (siehe 2.4.4) führt zum Jailbreak des 32- und 64-Bit-iOS-Kernels, dieses ermöglicht dem Angreifer, unbemerkt einen Jailbreak des Gerätes vorzunehmen und so eine Überwachungssoftware zu installieren.

3. **CVE-2016-4657**
   Dies ist eine ebenfalls durch *Memory-Corruption* ausgelöste Schwachstelle im Browser Safari, die dem Angreifer das Kompromittieren des Gerätes ermöglicht, wenn der Benutzer auf einen Hyperlink klickt.

Die dann installierte Spyware verwandelt das Gerät in ein mobiles Spionage-Gerät und besitzt die Fähigkeit, die Kamera anzustellen, das Mikrophon abzuhören, die Aufnahme von WhatsApp-Anrufen, das Protokollieren von Nachrichten, welche durch mobile Chat-Apps versendet wurden, und die Bewegungsverfolgung des Opfers. Die hauptsächliche Intention zur Entwicklung dieser Spyware basiert auf der Überwachung von Personen in hohen Positionen. Unter anderem wurde der international anerkannte arabische Menschenrechtsaktivist Ahmed Mansoor Ziel dieser Attacke. Dieser erhielt am 10. August 2016 eine SMS-Nachricht, die ihm sehr verdächtig erschien (siehe Abb. 3.6). Der Nachrichtentext versprach, neue Geheimnisse über die Misshandlung von Häftlingen in Gefängnissen in den Vereinigten Arabischen Emiraten zu enthüllen, mit einem darin enthaltenen Hyperlink. Am nächsten Tag erhielt Mansoor erneut eine ähnliche SMS. Er leitete die Nachrichten zur Untersuchung an Citizen Lab weiter. Die anschließende Untersuchung ermittelte die obigen Sicherheitslücken. [44] [43]

Abbildung 3.6: SMS Nachricht iPhone 6 Ahmed Mansoor [43]

3.3.2 Plattformübergreifende Sicherheitslücken

Neben Sicherheitslücken, die nur eine bestimmte Plattform betreffen, gibt es plattformunabhängige Sicherheitslücken. Die Vorteile, welche sich daraus für den Angreifer ergeben, liegen auf der Hand: Durch die erhöhte Zahl von potenziellen Zielen erhöht sich die Wahrscheinlichkeit eines Erfolges und einer damit verbundenen Rentabilität.

Die Tabelle 3.5 zeigt eine Auflistung von Sicherheitslücken, welche bei mehreren mobilen Plattformen vorliegen.

Name	Android	iOS	Windows Phone	CVE-ID
Freak-Attacken	✓	✓	✓	CVE-2015-0204 CVE-2015-1637 CVE-2015-1067
Adobe Flash / Air	✓	✓	✓	CVE-2015-7663 CVE-2015-8042 CVE-2015-8043 CVE-2015-8044 CVE-2015-8046

Tabelle 3.5: Plattformübergreifende Sicherheitslücken in Mobilen Betriebssystemen

FREAK-Attacken

Das Kürzel FREAK steht für Factoring-RSA-Export-Keys und beschreibt eine 2015 von Karthikeyan Bhargavan entdeckte kryptografische Schwachstelle im SSL/TLS-Protokoll. Die Hauptaufgabe von SSL- und TLS-Protokollen besteht darin, den Kommunikationspartner unter Verwendung von asymmetrischen Schlüsselverfahren und Zertifikaten zu authentifizieren, um eine vertrauliche Ende-zu-Ende-Datenübertragung zu gewährleisten [7, S. 798]. Die Schwachstelle ermöglicht dem Angreifer, die Kommunikationspartner dazu zu bringen, sich beim Aufbau einer gesicherten Verbindung auf ein schwaches Verschlüsselungsverfahren zu einigen. Für den Angreifer stellt die schwache Verschlüsselung keine große Hürde da, um an die vertraulichen Daten der Kommunikation zu gelangen. Alle Internet-Browser (Internet Explorer, Google Chrome, Safari) der mobilen Betriebssysteme sind von dieser Schwachstelle betroffen. [45]

Adobe Flash / Air

In Adobe Flash/Air kommt es immer wieder zu Sicherheitslücken, die Gründe hierfür sind folgende: Adobe Flash/Air ist plattform- sowie browser unabhängig und kann in andere Dokumente und Formate integriert werden. Eine Sicherheitslücke in Flash ermöglicht es Angreifern, sich Zugriff auf gleich mehrere Plattformen zu sichern, und ist daher für diese äußerst wertvoll. In diesem Fall ermöglichten die Sicherheitslücken den Angreifern

das entfernte Ausführen von Arbitrary Codes (siehe 2.4.4) in iOS, Android und Windows Phone. Eine konkrete Sicherheitslücke (CVE-2015-8042) nutzte Fehler bei der Verarbeitung von Sound-Objekten. Beim Aufruf einer Sound-Methode war es möglich, einen Use-after-free auszulösen. Use-after-free bezeichnet einen Memory-Corruption-Fehler, dieser kann vom Angreifer eingesetzt werden, um einen Arbitrary Code auszuführen. [46]

Anhand dieser Beispiele wird deutlich, welche Art von Schwachstellen für plattformübergreifende Sicherheitslücken infrage kommen. Typischerweise sind es Plug-ins, die von mehreren Betriebssystemen verwendet werden, aber auch Internet-Protokolle wie TLS/SLL. Web-Attacken stellen die größte Gefahr für alle Plattformen dar. Wenn beispielsweise ein Webserver Sicherheitslücken aufweist, so sind die Webseiten, welche sich auf diesem befinden, ebenfalls betroffen und auch Nutzer, die diese besuchen. Nicht nur die Betriebssysteme, welche auf dem Webserver ausgeführt werden, machen diese verwundbar. Ein großes Problem ist die Sicherheit von Plug-ins. Die Zahl von Sicher-

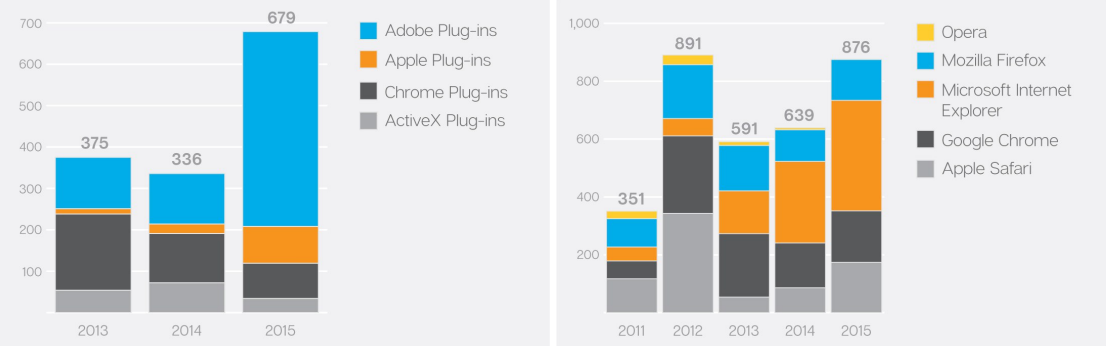

Abbildung 3.7: Sicherheitslücken in Plug-ins und Browsern [38]

heitslücken in Plug-ins wie Adobe ist im Jahr 2015 enorm gestiegen (siehe Abb. 3.7). Dies ist ein Indikator dafür, dass Angreifer nicht nur nach plattformübergreifenden Schwachstellen streben, sondern nach omnipräsenten Systemen. Die meisten Schwachstellen in Adobe stehen im Zusammenhang mit dem Adobe Flash Player. Dies war auch ein Grund dafür, weshalb Steve Jobs 2010 den Flash Player von iOS verbannt hat [47]. Auch alle Internet-Browser weisen viele Sicherheitslücken auf, hierbei führte der Internet Explorer von Windows das Ranking seit 2014 an. [38]

4 Analyse

4.1 Anforderungsanalyse

Die vorherigen Kapitel 2 und 3 gewährten Einblick in die Sicherheitsarchitektur und -aspekte von iOS. Die Erkenntnis, welche sich daraus ergibt, ist, dass iOS über eine grundsolide Sicherheitsarchitektur verfügt.

> *Apple hat bei der Entwicklung der iOS-Plattform die Sicherheit in den Mittelpunkt gestellt. Wir haben innovative Funktionen entwickelt und eingebaut, mit denen die mobile Sicherheit optimiert und das gesamte System standardmäßig geschützt ist. iOS ist deshalb ein großer Entwicklungssprung für die Sicherheit bei Mobilgeräten.* [1, S. 4]

Diese Aussage ist nicht ganz unberechtigt, denn in der Tat wurde durch den Einsatz unter anderem des Secure Enclaves (siehe 2.1.3) die Sicherheit bei mobilen Geräten auf ein neues Level gehoben. Nichtsdestotrotz leidet iOS genauso unter Sicherheitslücken wie die andere Mobilsysteme, wie in Kapitel 3 aufgezeigt wurde. Die Benutzer dürfen sich hierbei jedoch nicht aus der Verantwortung ziehen, denn die Sicherheitsarchitektur eines Systems ist immer nur so stark wie ihr schwächstes Glied. So sollte der Benutzer immer die neusten Sicherheits-Updates installieren und Passwörter mit höherer Entropie (siehe 2.3.3) verwenden. Nachfolgend wird eine Applikation für iOS entwickelt, die dem Endnutzer der Applikation Hinweise zu Sicherheitslücken in seinem System geben soll. Diese bietet im Wesentlichen zwei Funktionalitäten: Lokale und serverseitige Systemüberprüfung und Schwachstellen-Überprüfung im Netzwerk.

Lokale und serverseitige Systemüberprüfung:
Diese Funktion soll ermöglichen, bei einem Gerät und dessen Systemversion nach bekannten Sicherheitslücken (CVEs) zu suchen und diese dem Benutzer anzuzeigen. Dieses kann sowohl lokal als auch serverbasiert geschehen. Der Unterschied dabei ist, dass die App bei der lokalen Systemüberprüfung nur auf eine begrenzte Zahl von Metadaten (Sicherheitslücken) zurückgreift, welche in die App integriert sind,. Der Vorteil für den Nutzer ist, dass dieser das Gerät ohne jegliche Netzwerkanbindung auf Sicherheitslücken überprüfen kann. Dies hat jedoch auch eine Kehrseite, denn durch die Integration der Metadaten in die App können diese nicht in vollem Umfang gespeichert werden und auch nicht neuere bekannt gewordene Sicherheitslücken enthalten. Daher besteht eine weitere Möglichkeit der Systemüberprüfung auf Sicherheitslücken durch den Server: Dieser verfügt über eine Datenbank mit allen aktuellen Sicherheitslücken zu iOS und hält diese laufend auf dem neusten Stand. Für eine Überprüfung durch den Server muss sich das Endgerät aber im Netzwerk von ebendiesem befinden.

Schwachstellen-Überprüfung im Netzwerk:
Die zweite Funktion, welche die Applikation bieten soll, ist die Schwachstellen-Überprüfung für Netzwerkkomponenten durch den Browser. Hierfür kommt der Schwachstellen-Scanner OpenVAS, welcher auf dem Server betrieben wird, zum Einsatz. Für eine Überprüfung

durch Schwachstellen-Scanner muss eine Verbindung zum Server aufgebaut werden. Anhand dieser Anforderungen werden nun die funktionalen und nicht funktionalen Anforderungen zur Entwicklung der geplanten Applikation ermittelt.

4.1.1 Funktionale Anforderungen

Eine Auflistung der funktionalen Anforderungen zur erstellung des Kozeptes und anschließender Realisierung

FA-1 Die App muss die Systemversion des Gerätes ermitteln
FA-1.1 Unterteilung der Systemversion in Major, Minor und Patch-Version

FA-2 Die App muss mit der ermittelten Systemversion eine Lokale System -überprüfung auf dem Gerät durchführen.
FA-2.1 Verwendung von Sicherheitslücken aus Lokalen Metadaten
FA-2.2 Auswertung und Zuordnung der Sicherheitslücken zu einer Systemversion
FA-2.3 Ausgewertete Sicherheitslücken Anzeigen

FA-3 Die App muss mit der ermittelte Systemversion eine Serverseitige Systemüberprüfung auf dem Gerät durchführen.
FA-3.1 Verbindungsaufbau zum Server
FA-3.2 Geräte spezifische Anfrage an den Server senden
FA-3.2.1 Betriebssystem und Systemversion
FA-3.3 Auswertung der Serverantwort
FA-3.4 Ausgewertete Sicherheitslücken Anzeigen

FA-4 Detailansicht der Ausgewerteten Sicherheitslücken aus der Lokalen und Server-seitigen Systemüberprüfung
FA-4.1 Informationen zu CVE-ID, Sicherheitslückentyp, Beschreibung der Sicherheitslücke und Risiko Einstufung
FA-5 Die App muss das Webinterface vom Netzwerk-Schwachstellen Scanner welcher sich auf dem Server befindet aufrufen. Und hierdurch die Prüfung auf Netzwerk-Schwachstellen ermöglichen.
FA-5.1 Aufruf über den Safari Web Browser
FA-5.2 Logindaten für Webinterface bereitstellen
FA-5.3 IP des Gerätes auslesen und anzeigen
FA-5.4 Anleitung für den Netzwerk-Schwachstellen Scanner

4.1.2 Nicht funktionale Anforderungen

Die nicht funktionalen Anforderungen beschreiben die Qualitätseigenschaften der App, welche sicherlich aus der Sicht des Nutzers relevanter sind. In gewisser Hinsicht spiegeln

diese die Benutzerfreundlichkeit der Applikation wider und unterteilen sich in folgende Punkte: Benutzbarkeit, Effizienzanforderungen, Korrektheit, Installierbarkeit, Fehlertoleranz und Verfügbarkeit.

Benutzbarkeit:
Es ist eine leichte und für den Benutzer selbsterklärende Menüstruktur zu wählen, welche auf die wesentlichen Funktionen wie auswählen und starten reduziert ist. Dadurch soll vermieden werden, dass der Benutzer durch die sinnlose Überfrachtung mit Menüelementen verwirrt wird.

Effizienzanforderungen:
Die Applikation kommuniziert mit einem *Server*, auf das Antwortverhalten von diesem hat die App keinen Einfluss. Die App muss daher, nachdem eine Anfrage zur Analyse des Gerätes getriggert wurde, einen zeitlichen Rahmen setzen, um nicht endlos auf den *Server* zu warten.

Korrektheit:
Die von der Applikation zu einem *iPhone* ermittelten Schwachstellen bei einer lokalen Systemüberprüfung müssen diesem korrekt zugeordnet werden. Es wäre fatal, wenn falsche, d. h. nicht zur Systemversion des jeweiligen Gerätes passende Sicherheitslücken angezeigt würden.

Installierbarkeit:
Die Applikation soll auf allen *iPhones*, welche über die Systemversion iOS 9.0 oder höher verfügen, installierbar sein. So wird ein Großteil der Nutzerschaft abgedeckt (siehe 3.1.1).

Fehlertoleranz:
Die Applikation soll bei einem Fehlverhalten des Benutzers, wie dem Starten der Serverüberprüfung ohne vorherige Serververbindung angemessen reagieren (Benutzer bei Fehlverhalten über GUI informieren).

Verfügbarkeit:
Zur Verwendung der serverseitigen Systemüberprüfung und des Netzwerk-Schwachstellen-Scans wird eine Netzwerkverbindung zum Server benötigt. Die Netzwerkverbindung wird über einen drahtlosen Router realisiert, mit welchem sich der Benutzer der App verbinden muss.

4.2 Kontextabgrenzung

Die zuvor ermittelten funktionalen und nicht funktionalen Anforderungen decken nur den Funktionalitätsumfang der Applikation ab. Die vom Server bereitgestellten Funktionen sind nicht Bestandteil dieser Bachelorarbeit.

4.3 Anwendungsfälle

Anhand eines UML-Anwendungsfalldiagramms (siehe Abb. 4.1) wird die Funktionalität der Applikation festgehalten. Im Anwendungsfall werden alle möglichen Szenarien, welche durch den Benutzer des betrachteten Systems zur Erreichung des Ziels eintreten können, aufgezeigt, ohne auf eine konkrete technische Lösung einzugehen. Das System ist in diesem Fall die iOS-Applikation, auf welche die Akteure, d. h. Benutzer und Server, zurückgreifen. Die einzelnen Szenarien, welche sich aus dem Anwendungsfall ergeben, sind in den Tabellen 4.1, 4.2, 4.3, 4.4 und 4.5 aufgelistet.

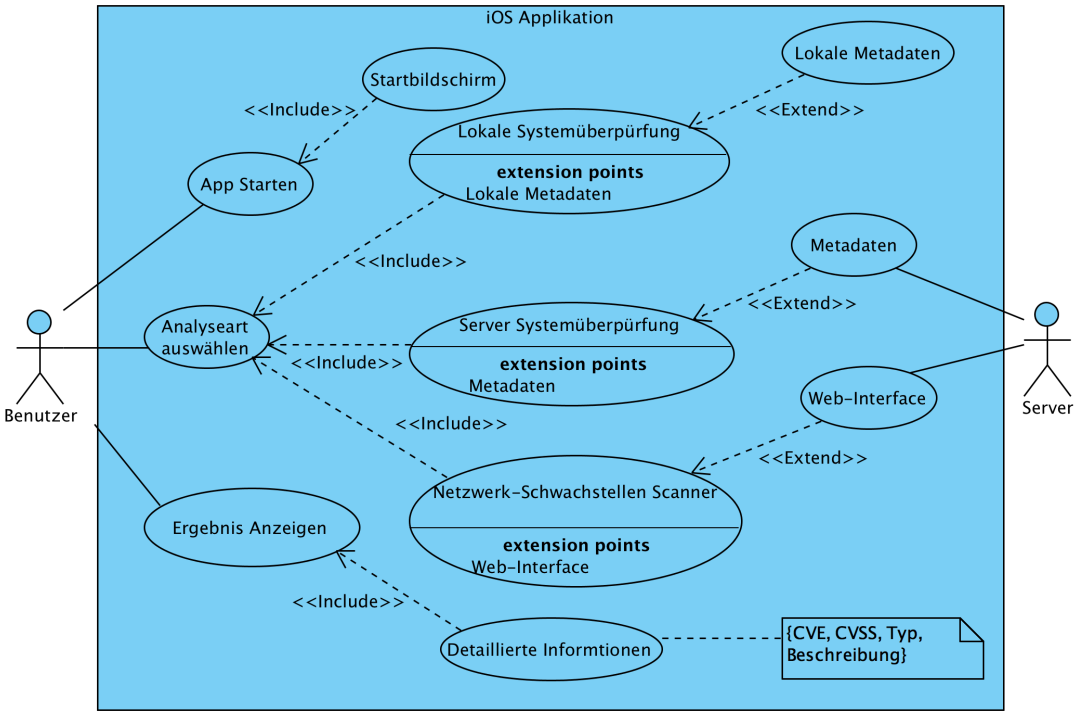

Abbildung 4.1: UML Diagramm zu den Anwendungsfälle

Name:	**App Starten**
Anwendungsfall-ID:	1
Akteure	Benutzer, App
Auslöser:	Der Benutzer drückt im Homescreen auf das App-Icon
Kurzbeschreibung	Der Benutzer startet die App
Vorbedingung	Der Benutzer hat die Applikation installiert
Nachbedingungen:	Der Benutzer kann eine Analyseart auswählen und diese starten
Ablaufschritte:	1. Der Benutzer drückt im Homescreen auf das App-Icon und startet damit die App 2. Die App hat den Startvorgang abgschlossen und wartet auf eine Eingabe durch den Benutzer

Tabelle 4.1: Anwendungsfall *„App Starten“*

Name:	**Lokale Systemüberprüfung**
Anwendungsfall-ID:	2
Actor	Benutzer, App
Kurzbeschreibung	Der Benutzer startet die Lokale Systemüberprüfung
Vorbedingung	Der Benutzer hat die App gestartet und die App befindet sich im Startbildschirm
Nachbedingungen:	Eine *Ergebnis-Anzeigen*-Taste erscheint im Startbildschirm der App
Auslöser:	Benutzer wählt als Analyseart *Lokal* aus und drückt auf die *Analyse-Starten*-Taste
Standard Ablaufschritte:	1. Der Benutzer wählt im *Segmented Control* den Reiter *Lokal* aus und betätigt die *Analyse-Starten*-Taste 2. Die App prüft das Gerät, anhand der Systemversion auf Schwachstellen und wertet diese aus 3. Die App zeigt die *Ergebnis-Anzeigen*-Taste an
Anforderungen:	FA-1.1, FA-2, FA-2.1, FA-2.2

Tabelle 4.2: Anwendungsfall „Lokale Systemüberprüfung“

Name:	**Server Systemüberprüfung**
Anwendungsfall-ID:	3
Actor	Benutzer, App, Server
Kurzbeschreibung	Der Benutzer startet die Serverbasierte Systemüberprüfung
Vorbedingung	Der Benutzer hat die Applikation gestartet und die App befindet sich im Startbildschirm
Nachbedingungen:	Eine *Ergebnis-Anzeigen*-Taste erscheint im Startbildschirm der App
Auslöser:	Benutzer wählt als Analyseart *Server* aus und drückt auf die *Analyse-Starten*-Taste
Standard Ablaufschritte:	1. Der Benutzer wählt im *Segmented Control* den Reiter *Server* aus und betätigt die *Analyse-Starten*-Taste 2. Die App sendet eine Anfrage auf Schwachstellen mit Systemversion und Betriebssystem zum Server 4. Der Server verarbeitet die Anfrage und wertet diese aus und sendet eine Antwort an die App 6. Die App verarbeitet die Antwort 7. Eine *Ergebnis-Anzeigen*-Taste erscheint im Startbildschirm der App
Fehlerfälle:	Die App konnte keine Verbindung zum Server aufbauen
	Keine Netzwerkverbindung vorhanden
Anforderungen:	FA-1, FA-3, FA-3.1, FA-3.2, FA-3.2.1, FA-3.3

Tabelle 4.3: Anwendungsfall *„Server Systemüberprüfung“*

Name:	**Ergebnis anzeigen**
Anwendungsfall-ID:	4
Actor	Benutzer, App
Kurzbeschreibung	Der Benutzer lässt sich das Ergebnis aus der Lokalen oder serverseitigen Systemüberprüfung anzeigen
Vorbedingung	Der Benutzer hat eine Systemüberprüfung (Lokal, Server) in der App abgeschlossen
Nachbedingungen:	Die App zeigt eine detaillierte Information zu einer Schwachstelle an

Tabelle 4.4 – *Fortsetzung auf der nächsten Seite*

Tabelle 4.4 – *Fortsetzung aus vorheriger Seite*

Auslöser:	Der Benutzer drückt auf die *Ergebnis-Anzeigen*-Taste
Standard Ablaufschritte:	1. Der Benutzer drückt auf die *Ergebnis-Anzeigen*-Taste 2. Die App zeigt die gefunden Schwachstellen in einer Liste an 3. Der Benutzer wählt eine Schwachstelle für genauere Informationen 4. Die App zeigt die detaillierte Information zu der Schwachstelle an
Alternative:	Es sind keine Sicherheitslücken vorhanden und somit sind auch keine detaillierten Informationen vorhanden
Anforderungen:	FA-4, FA-4.1, FA-4.2, FA-4.3, FA-4.4, FA-2.4, FA-3.4

Tabelle 4.4: Anwendungsfall „Ergebnis Anzeigen“

Name:	**Netzwerk-Schwachstellen Scanner**
Anwendungsfall-ID:	5
Actor	Benutzer, App, Server
Kurzbeschreibung	Der Benutzer führt eine Netzwerk-Schwachstellen Überprüfung durch den Server durch
Vorbedingung	Der Benutzer hat die Applikation gestartet und die App befindet sich im Startbildschirm
Nachbedingungen:	Der Benutzer zeigt sich das Ergebnis des Netzwerk-Schwachstellen Scanner an
Auslöser:	Der Benutzer wählt im *Segmented Control* den Reiter openVAS aus.

Tabelle 4.5 – *Fortsetzung auf der nächsten Seite*

Tabelle 4.5 – *Fortsetzung aus vorheriger Seite*

Standard Ablaufschritte:	1. Der Benutzer wählt im *Segmented Control* den Reiter *openVAS* aus 2. Der Benutzer durchläuft eine bebilderte Anleitung zur Verwendung des Netzwerk-Schwachstellen Scanner und betätigt am ende das Schließen Symbol 3. Die App zeigt die Logindaten und IP-Adresse des Gerätes an und eine *Analyse-Starten*-Taste 4. der Benutzer betätigt die *Analyse-Starten*-Taste 5. Die App ruft den Netzwerk-Schwachstellen Scanner mit der URL des Servers im Safari Browser auf und es wird eine Zertfikatswarnung angezeigt. 6. Der Benutzer bestätigt mit der Fortfahren-Taste 7. Der Server lädt das Web-Interface und zeigt es an 8. Der Benutzer gibt die Logindaten im Web-Interface ein 9. Der Server lädt die Startseite des Netzwerk-Schwachstellen Scanner 10. Der Benutzer gibt die IP-Adresse des gerätes ein und betätigt die *Scan-Starten*-Taste 11. Server führt den Schwachstellen Scan durch 12. Der Benutzer zeigt sich das Ergebnis des Netzwerk-Schwachstellen Scanner an
Fehlerfall:	Es konnte keine Verbindung zum Web-Interface des Servers aufgerufen werden.
Anforderungen:	FA-5, FA-5.1, FA-5.2, FA-5.3, FA-5.4

Tabelle 4.5: Anwendungsfall *„Netzwerk-Schwachstellen Scanner “*

5
Konzept und Design

5.1 System Architektur

Die Abbildung 5.1 gewährt einen Überblick über die Systemarchitektur und die wichtigsten Komponenten zur Umsetzung dieser. Die Architektur teilt sich in zwei wesentliche Bestandteile, Server und Client. Beim Client handelt es sich in diesen Fall um ein iPhone. Neben der lokalen Systemüberprüfung, welche sich auf die in der App bereitgestellten Informationen beschränkt, werden verschiedene Komponenten zur Umsetzung der serverseitigen Systemüberprüfung und des Netzwerk-Schwachstellen-Scans benötigt. Bevor der App eine Kommunikation mit dem Server ermöglicht wird, muss sich das iPhone im selben lokalen Netzwerk wie diese befinden. Dafür verbindet sich das Endgerät mit dem drahtlosen Netzwerk des Routers.

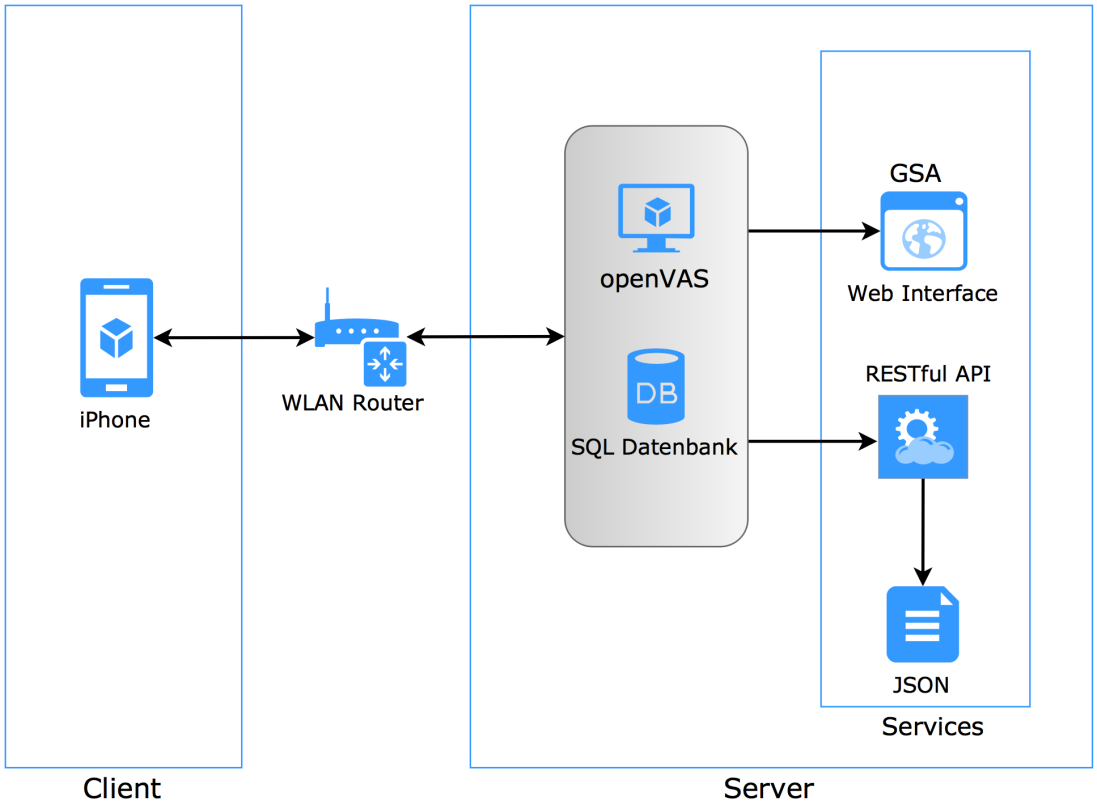

Abbildung 5.1: Überblick Systemarchitektur

Der Server stellt zwei Dienste zur Verwendung durch die App bereit: Zum einen verfügt dieser über eine REST-Schnittstelle, diese ermöglicht es, Anfragen bezüglich der Sicherheitslücken zu einer bestimmten Systemversion eines Betriebssystems an den Server zu senden. Die andere Funktionalität des Servers ermöglicht das Netzwerk-Schwachstellen-Scanning durch das openVAS-Software-Framework. Die Konzeptionierung und Realisierung der Serverfunktionalitäten sind nicht Bestandteil dieser Bachelorarbeit,

nichtsdestotrotz müssen diese beim Entwurf der Applikation berücksichtigt werden. Dies nimmt der Konzeptionierung aber auch gleichzeitig einige Entscheidungen diesbezüglich ab.

5.2 Serverseitige Auflagen

Bevor auf die Konzeptionierung der Applikation eingegangen werden kann, wird zunächst die Funktionsweise der Serverfunktionalitäten erläutert. Anhand dessen wird ein Konzept zur Umsetzung der Serverfunktionalitäten erstellt. Im Wesentlichen bietet der Server zwei Funktionalitäten: Serverseitige Systemüberprüfung und Netzwerk-Schwachstellen-Scan

5.2.1 Serverseitige Systemüberprüfung

Bei der serverseitigen Überprüfung wird, wie in der Abbildung 5.2 zu sehen, zunächst ein HTTP-GET-Request, welcher als Parameter die Betriebssystemvariante und Systemversion des iPhones enthält, an den Server gesendet. Nun wird die Anfrage durch den Server bearbeitet: Dieser sucht in der Datenbank für die in der Anfrage angegebenen Parameter nach bekannten Sicherheitslücken. Nachdem die Auswertung der Anfrage durch den Server abgeschlossen worden ist, verpackt dieser die Nachricht als Medientyp im JSON-Format und sendet diese an das iPhone respektive die App zurück.

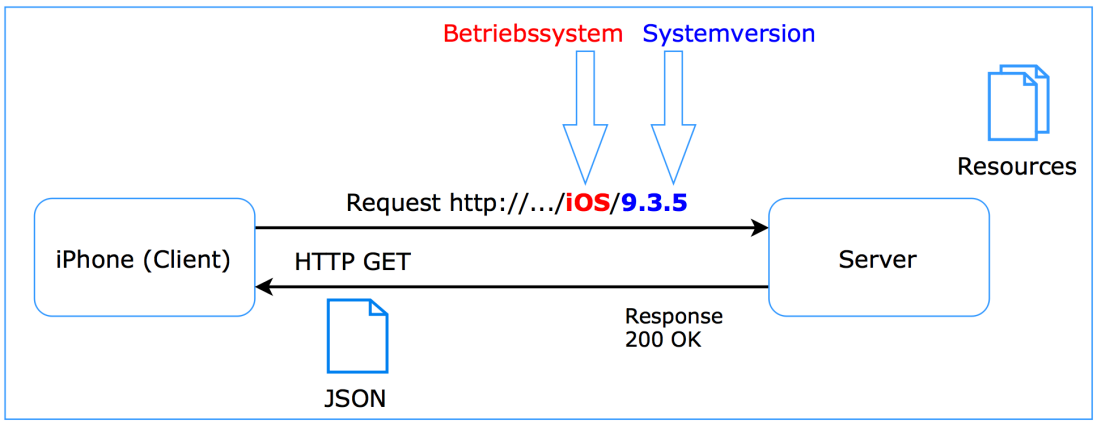

Abbildung 5.2: Dienst Serverseitige Systemüberprüfung

Zusammengefasst ergeben sich folgende Anforderungen an die App, um den Dienst der serverseitigen Analyse in Anspruch zu nehmen:

- Anfragen bezüglich Sicherheitslücken müssen über einen HTTP-GET-Request erfolgen.
- In der URL müssen Parameter für die Betriebs- und Systemversion enthalten sein.
- Die Verarbeitung von Antworten erfolgt im JSON-Format.

5.2.2 Netzwerk-Schwachstellen-Scan

Der weitere Dienst, welcher durch den Server bereitgestellt wird, ist der Schwachstellen-Scanner (openVAS). Dieser stellt ein Web-Interface, den Greenbone Security Assistant (GSA), welcher über einen Web-Browser aufgerufen werden kann, bereit.

```
1 https://server_domain_or_IP:9392
```

Listing 5.1: URL des Greenbone Security Assistent

Um einen Zugriff auf das Web-Interface zu erhalten, muss die Adresse des Servers im Browser aufgerufen werden (siehe Listing 2), der Greenbone Security Assistent ist über den Port 9392 erreichbar. Nachdem die Seite aufgerufen worden ist, muss sich der Nutzer zunächst authentifizieren. Dafür stellt der Server Zugangsdaten für den iOS-Benutzer bereit. Die Login-Daten lauten wie folgt: Benutzername = iOS und Passwort = iosscan. Nach dem erfolgreichen Login muss der User die IP seines Gerätes, welche in der App angezeigt wird, im Quick-Start-Wizard des GSA eingeben und kann über die Betätigung des *„Start Scan“*-Buttons den Startvorgang starten.

5.3 Applikation Entwurf

Nachdem zuvor ein Überblick über die Systemarchitektur vermittelt worden ist, wird anhand dessen ein Konzept ermittelt, um die zuvor aufgezeigte Architektur umzusetzen. Bei der Konzeptionierung der App müssen die serverseitigen Anforderungen berücksichtigt werden.

5.3.1 Lokale Funktionalitäten

Für die lokale Systemüberprüfung auf Sicherheitslücken werden Metadaten benötigt. Dazu, wie diese Daten gespeichert bzw. der App zugänglich gemacht werden können, bietet iOS verschiedene Lösungsansätze. Bevor ein bestimmter Lösungsansatz gewählt werden kann, wird zunächst ein Überblick über die verschiedenen Realisierungsmöglichkeiten aufgezeigt.

- SQLite
  Eine Möglichkeit ist die Verwendung einer Datenbank, iOS bietet hierfür die Verwendung der relationalen Datenbank SQLite an. SQLite ist eine Open-Source-basierte SQL-Programmbibliothek, welche sich direkt in die Anwendung integrieren lässt. Dadurch sind keine weitere Konfiguration und Server-Software nötig. Zudem ist sie nur wenige 100 Kilobyte groß und besteht nur aus einer einzigen Datei, die alle Tabellen, Views, Trigger usw. enthält und plattformunabhängig ist. [48] [49]

- Core Data
  Abgesehen von einer Datenbank bietet Apple das Core-Data-Framework für die persistente Speicherung von Objekten gemäß einem Datenmodell an. Der Fokus von

Core Data ist mehr auf Objekte als das klassische tabellenbasierte Datenbankmodell gerichtet, dies birgt Vor- und Nachteile. Zum einen ist das Datenaufkommen bei Core Data höher, da kein Datensatz, sondern das komplette Objekt gesichert wird, dafür können die Daten wesentlich schneller abgerufen werden. [50] [51]

- JSON
  Die JavaScript Object Notation (JSON) ist durch zwei Standards RF7159 von Douglas Crockford und ECMA-404 spezifiziert. Es handelt sich dabei um ein einfach lesbares Datenformat in Textform zum Datenaustausch. Es wird bei der Übertragung und Speicherung von strukturierten Daten verwendet und ist von der Programmiersprache unabhängig. Gerade bei mobilen Apps wird dieses daher häufig zum Transfer von Daten zwischen Client und Server genutzt. Im Gegensatz zu SQLite und Core Data ist mit JSON keine persistente Speicherung von Daten bzw. Objekten möglich. [52]

Bei großen Datenmengen wäre es allein aus Performanzgründen unumgänglich, auf Core Data oder SQLite zu setzen, in diesem Fall steht der Mehraufwand zur Umsetzung dieser aber nicht im Verhältnis zum Vorteil, welcher sich daraus für die Applikation ergeben würde. Auch wenn die in JSON gespeicherten Metadaten vor jeder Verwendung geparst werden müssen, ist bei solch geringen Datenmengen, wie es bei dieser App der Fall ist, kein Performanzunterschied zu bemerken. Daher fällt die Wahl vorliegend auf JSON und damit ist die Anforderung **FA-2.1** und **FA-4.1** (siehe 4.1.1) erfüllt. Das Listing 5.2 zeigt exemplarisch die Stuktur der JSON-Datei. In dieser sind alle benötigten Informationen sowohl zur Anzeige (cveId, vType, vScore, info) als auch zur Auswertung (Major, Minor, Patch) der Sicherheitslücke enthalten.

```
1 [{"cveId":"CVE-2015-5923",
2 "vType":"+Info",
3 "pdate":"2015-10-09",
4 "vScore":2.1,
5 "info":"Apple iOS before 9.0.2 does not properly restrict the
     options available on the lock screen, which allows physically
     proximate attackers to read contact data or view photos via
     unspecified vectors.",
6 "os":"ios",
7 "Major": 9,
8 "Minor":0,
9 "Patch":2}]
```

Listing 5.2: Struktur der JSON-Datei

5.3.2 Serverseitige Funktionalitäten

Um die serverseitigen Funktionalitäten umzusetzen, muss zuerst eine Verbindung zum Server aufgebaut werden. Dafür gibt es verschiedene Möglichkeiten, diese werden zunächst vorgestellt und anschließend wird eine Wahl aus diesen getroffen.

NSURLSession:
Die Klasse NSURLSession bietet eine API zum Herunterladen von Inhalten, welche eine breite Palette von Methoden für die Unterstützung der Authentifizierung bietet. Die Klasse ermöglicht auch die Durchführung von Hintergrund-Downloads, wenn die App nicht ausgeführt wird oder während die App in IOS gesperrt (ausgesetzt) ist. NSURLSession unterstützt standardmäßig (nativ) Daten, Dateien, FTP- sowie HTTP- und HTTPS-URL-Schemata mit einer transparenten Unterstützung für Proxy-Server und SOCKS-Gateways, welche bereits in den Systemeinstellungen des Benutzers konfiguriert werden.

Für die private Nutzung der Applikation kann man auch eigene benutzerdefinierte Netzwerkprotokolle und URL-Schemata hinzufügen. Die App erstellt mit der NSURLSession-API eine oder mehrere Sitzungen, welche eine Gruppe von zusammenhängenden (zugehörigen) Datenübertragungsaufgaben verbindet. Das heißt, innerhalb jeder Sitzung fügt die Applikation eine Reihe von Tasks hinzu, die jeweils eine Anforderung für eine bestimmte URL repräsentieren. Die Aufgaben innerhalb einer bestimmten URL-Sitzung teilen sich ein gemeinsames Sitzungskonfigurationsobjekt, welches das Verbindungsverhalten definiert. Dies beinhaltet z. B. die maximale Zahl gleichzeitiger Verbindungen, die an einem einzelnen Host vorgenommen werden sollen, oder Verbindungen, die über ein zellulares Netzwerk zulässig sind.

RestKit
RestKit ist ein modernes Objective-C-Framework zur Implementierung von clientseitigen RESTful-Webservices in iOS und Mac. Es handelt sich hierbei um eine Drittanbieter-Bibliothek, die über ein leistungsstarkes Objekt-Mapping verfügt, welches nahtlos mit Core Data (siehe 5.3.1) interagiert. Darüber hinaus sind vorgefertigte Funktionen zum Handling von einfachen Netzwerk-Funktionalitäten wie dem Mapping von HTTP-Anfragen (Requests) und Antworten (Responses), auf Basis der Netzwerk-Bibliothek AFNetworking, integriert. Die AFNetworking-Bibliothek baut auf dem im Foundation-Framework enthaltenen URL-Loading-System auf, in welchem auch die NSURLSession-Klasse enthalten ist. Das RestKit besteht im Wesentlichen folglich aus drei Komponenten:

- Netzwerk:
  RestKit verwendet AFNetworking in der Version 1.3.3 für die Netzwerk-Schicht.
- Objekt-Parser
  Er bietet eine einfache Schnittstelle, um Server-Antworten basierend auf JSON/XML in lokale Objekte zu konvertieren.

- Core Data
  Dies bietet zusätzlich zum Objekt-Mapping die Möglichkeit, Objekte persistent zu speichern.

Durch die Nutzung von RestKit kann das Schreiben von viel Code vermieden werden, nichtsdestotrotz lässt es sich nicht leicht in die Entwicklungsumgebung integrieren. Prinzipiell ist RestKit die optimale Wahl bei Applikationen, welche RESTful-Dienste in Anspruch nehmen. Doch in diesem Fall ist die Verwendung von NSURLSession völlig ausreichend, da zum einen die vom Server erhaltenen Metadaten nicht persistiert werden sollen und zum anderen der Code, zur Realisierung des Parsens der Metadaten, sehr gering ausfällt. Außerdem würde RestKit die Auswahl der Programmiersprachen beeinflussen, die verwendet werden müssten. Damit ist die Anforderung **FA-3.1** (siehe 4.1.1) erfüllt.

Um den Netzwerk-Schwachstellen-Scanner zu verwenden, muss der Greenbone Security Assistent im Webbrowser aufgerufen werden. Das Listing 5.3 zeigt die dafür benötigte openURL-Methode welche durch das UIKit-Framework bereitgestellt wird.

```
1 UIApplication.shared.openURL(URL(string:"https://
    server_domain_or_IP:9392")!)
```

Listing 5.3: GSA aufruf in der App

Zur Umsetzung einer bebilderten Anleitung für den Netzwerk-Schwachstellen Scanner wird die frei verfügbare BWWalkthrough-Bibliothek verwendet. [53] Damit ist die Anforderung **FA-5.4** (siehe 4.1.1) erfüllt.[53]

5.4 Fachliches Datenmodell

Entsprechend den Anforderungen (siehe 4.1.1) wurde ein fachliches Datenmodell entwickelt, dies wird anhand eines Klassendiagramms (siehe Abb. 5.3) dargestellt.

5.4.1 Models

SecurityVulnerability
Diese Klasse modelliert eine Schwachstelle und ihre Eigenschaften als Klassenattribute. Zu den Eigenschaften gehört das jede Schwachstelle über eine CVE-ID (cveId), Beschreibung (description), Risikoeinstufung(vScore) und einen Schwachstellen Typ(vType) verfügt. Bei der Instanziierung der Klasse müssen diese angegeben werden.

LocalSystemAnalysis
Diese Klasse ermöglicht zum einem das Parsen (jsonParsing-Methode) von Schwachstellen aus der Lokalen Metadaten (JSON-Datei) und die Auswertung (versionEvaluation-Methode) dieser. Zudem verfügt sie über eine Instanz zur Klasse DeviceInfo um die benötigten Referenzen für die Klassenattribute (Major, Minor, Patch) zu erhalten.

ServerSystemAnalysis
In dieser Klasse wird die connectToServer-Methode zum Aufbau einer Serververbindung bereitgestellt. Zudem wird die updateSearchResults-Methode zum Parsen von Serverantworten bereitgestellt und im Klassenattribut searchResults gespeichert.

DeviceInfo
In dieser Klasse wird die Systemversion und die einzelnen Versionsnummern über getter-Methoden zur Verfügung gestellt.

DeviceNetworkAddress
Diese Klasse kann die IP-Adresse des Gerätes ermitteln und zur Verfügung stellen.

ServerConnectionDelegate
Diese Klasse definiert einen Schnittstellenkontrakt mit einem Protokoll. Dieses Protokoll kommt im Form eines Delegate-Pattern zwischen der Klasse ServerSystemAnalysis und MainViewController zum Einsatz.

5.4.2 View und Controller

Bei den folgenden Klassen handelt es sich um Cocoa Touch Klassen (siehe 6.1), diese stellen Verbindungen zur Grafischen Benutzeroberfläche her.

LaunchScreenViewController
Dies ist die erste Klasse welche aufgerufen wird wenn die Applikation gestartet wird, die-

se zeigt den Ladebildschirm an und ruft anschließend die MainViewController-Klasse auf.

MainViewController
Diese Klasse stellt den Startbildschirm der Applikation dar und verfügt über verschiedene Bedienelemente wie UISegemented Control, UIButtons und UILabels. Eine vom Anwender gestartete Überprüfung (lokal oder server) wird über vorhandene Instanzen zu den Klassen von Lokaler oder serverseitiger Überprüfung aufgerufen. Des Weiteren verfügt die Klasse über Segues (siehe 6.3) zu den Klassen ResultTableViewController und NetworkScanViewController. Mit Segues(Übergang) ist es nicht nur möglich die ViewController der Klassen aufzurufen, sondern auch Daten (z. B. Ergebnis aus Lokaler Systemüberprüfung) mit der prepare-Methode zu übertragen. Zudem sind Methoden (progessUpdate(), resetAnalysis(), finishedAnalysis()) zur Aktualisierung der Grafischen Benutzeroberfläche enthalten, wie Beispielweise das Anzeigen der *Ergebnis-Anzeigen*-Taste wenn eine Überprüfungsvorgang abgeschlossen ist.

ResultTableViewController
In dieser Klasse wird das Ergebnis aus Lokaler und serverseitiger Überprüfung in Form einer Tabelle angezeigt. Die dafür benötigten Informationen werden über einen Segue zur MainViewController-Klasse erhalten. Wird eine Schwachstelle aus der Tabelle angeklickt wird über ein Segue die ResultDetailViewController Klasse aufgerufen und dieser die Information zur Sicherheitslücke übergeben (prepare-Methode).

CveTableViewCell
Diese Klasse definiert die Attribute und das Verhalten der Zellen welche im ResultTableViewController erscheinen.

ResultDetailViewController
In dieser Klasse werden die detaillierten Informationen zu einer Sicherheitslücke angezeigt. Die benötigten Informationen erhält die Klasse über ein Segue.

NetworkScanViewController
Diese Klasse stellt die benötigten Informationen für den Benutzer bereit, um den Netzwerk-Schwachstellen- Scanner starten zu können. Es wird zunächst eine bebilderte Anleitung gestartet und sobald diese abgeschlossen ist wird der NetworkScanViewController angezeigt.

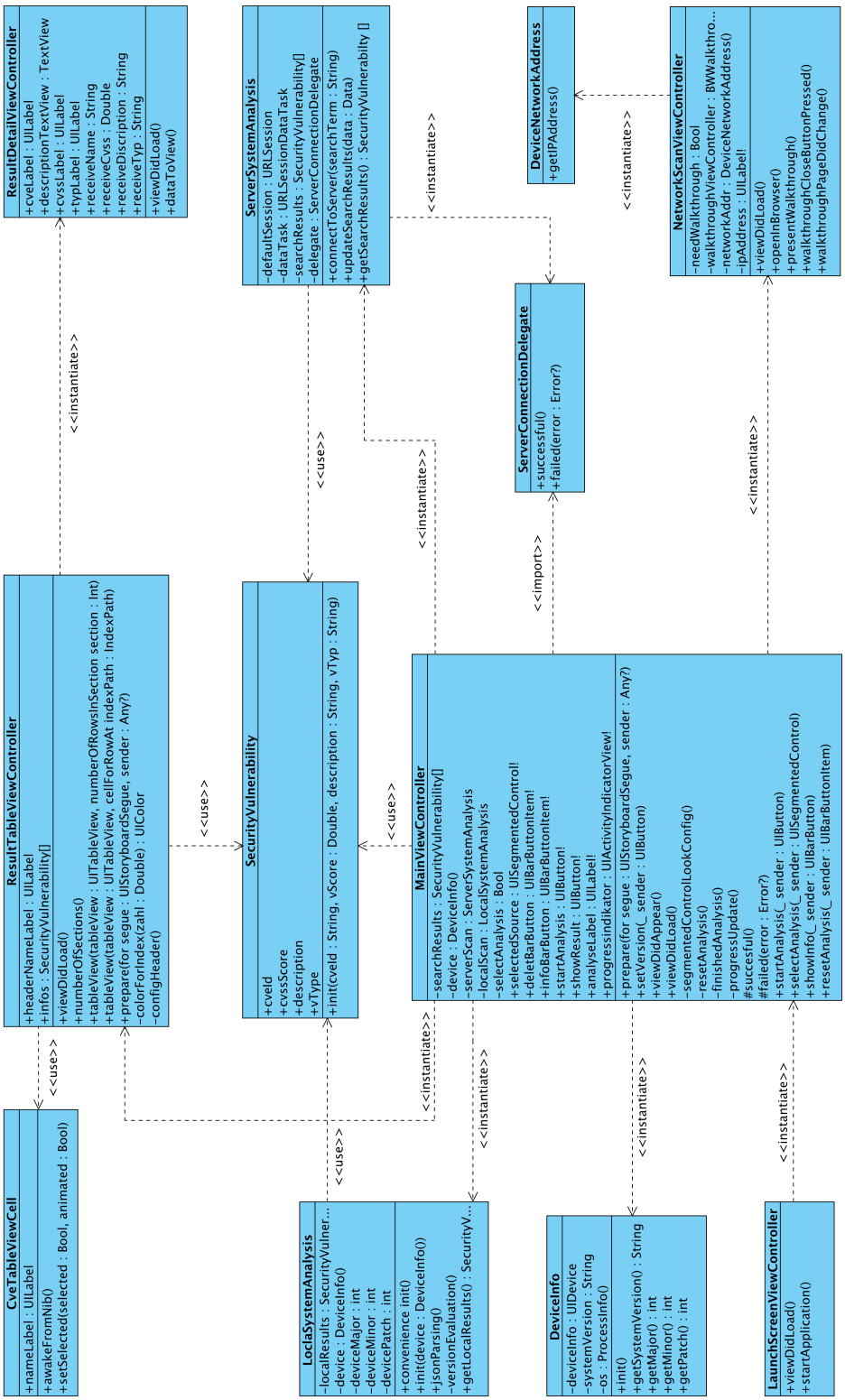

Abbildung 5.3: Klassendiagramm

5.5 Ablaufdiagramme

In diesem Abschnitt werden zwei Sequenzdiagramme vorgestellt, welche exemplarisch die Kommunikation der einzelnen Klassen bei den Szenarien „Lokale Systemüberprüfung“ und „Server Systemüberprüfung“ beschreiben. Mit diesen werden die Abhängigkeiten der Klassen in den Abläufen verdeutlicht.

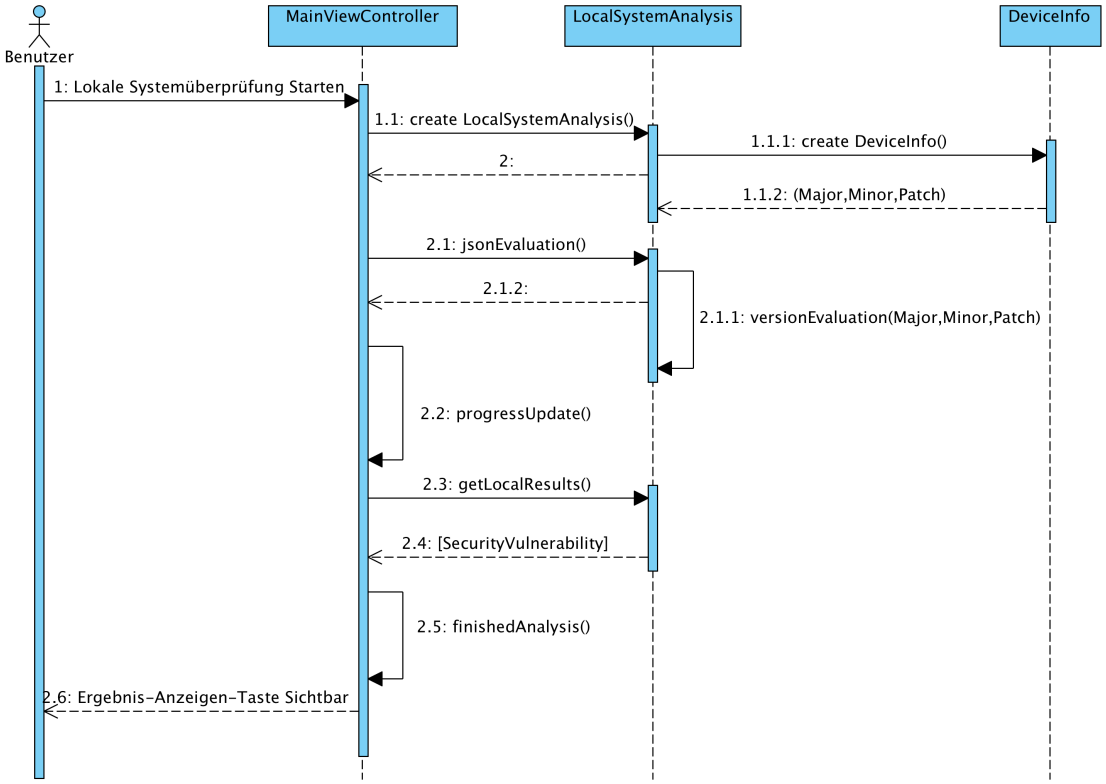

Abbildung 5.4: Sequenzdiagramm der Lokalen Systemüberprüfung

Das Sequenzdiagramm in der Abbildung 5.4 repräsentiert den Anwendungsfall „Lokale Systemüberprüfung“, dieser ist in der Tabelle 4.2 beschrieben.

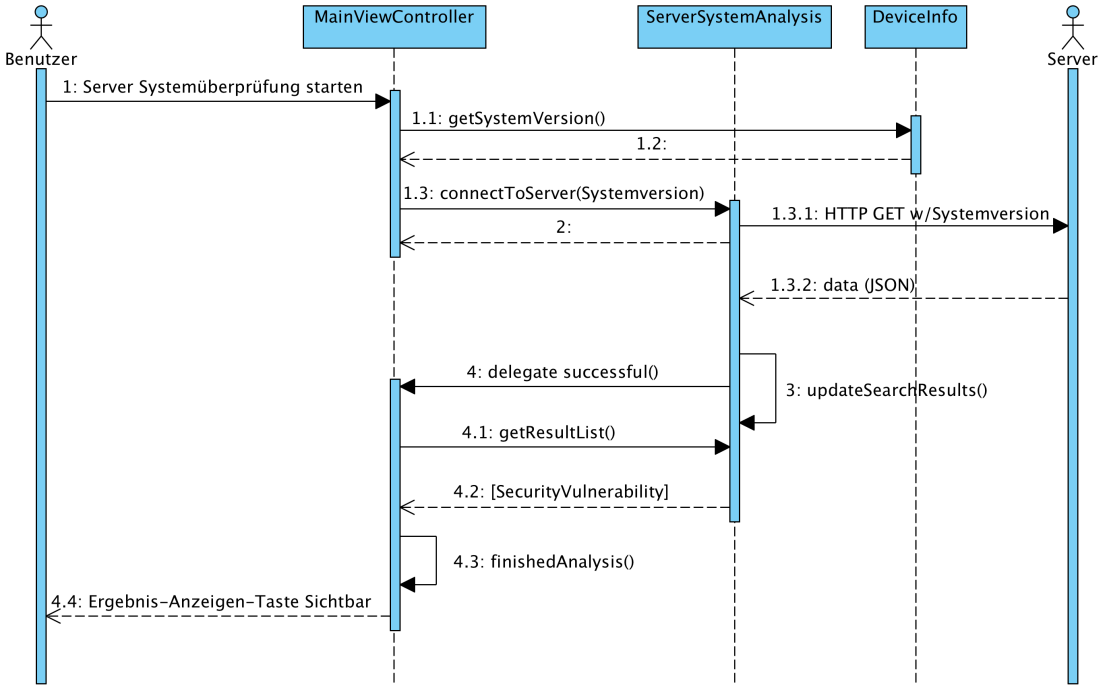

Abbildung 5.5: Sequenzdiagramm der Serverseitigen Systemüberprüfung

Das Sequenzdiagramm in der Abbildung 5.5 repräsentiert den Anwendungsfall „Server Systemüberprüfung“, dieser ist in der Tabelle 4.3 beschrieben.

5.6 GUI-Entwurf

Um dem Benutzer der App die Interaktion mit dem Programm zur ermöglichen, werden grafische Bedienelemente zur Verfügung gestellt, welche Bestandteil der grafischen Benutzeroberfläche (GUI) sind. Diese werden im Einzelnen vorgestellt und anhand eines Ablaufes erläutert.

5.6.1 Lokale und serverseitige Systemüberprüfung

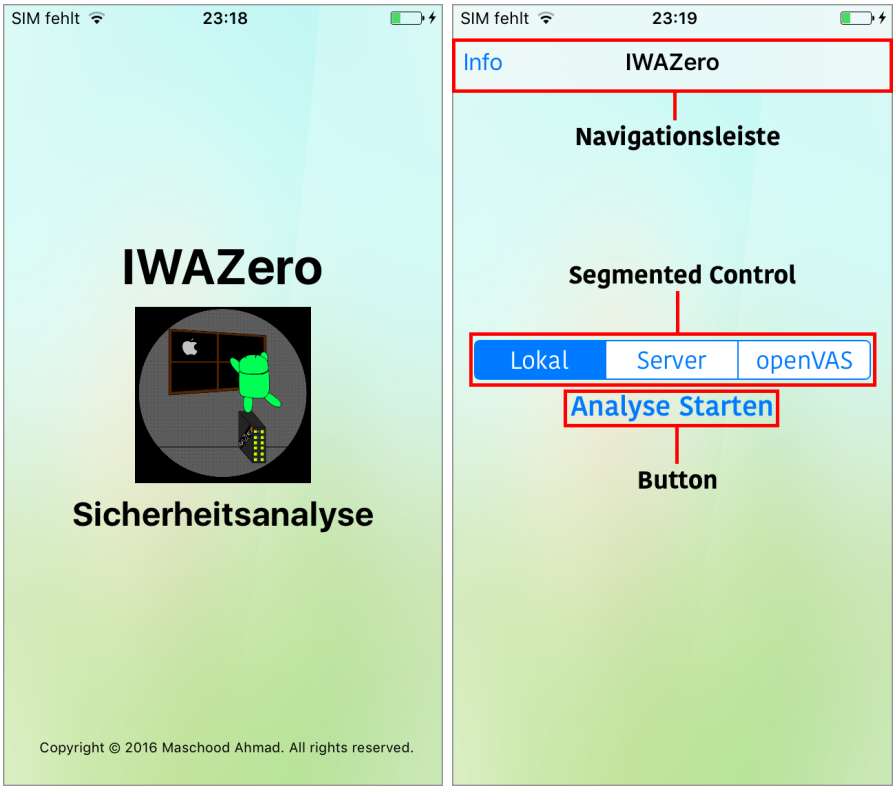

Abbildung 5.6: App Splash-Screen und Startbildschirm

Nachdem die Anwendung vom Benutzer gestartet worden ist, wird zunächst der Splash-Screen (grafischer Platzhalter, der während des Startens angezeigt wird) angezeigt (siehe Abb. 5.6, links). Dieser dient dazu, dem Benutzer den Eindruck zu vermitteln, dass die App schnell gestartet und einsatzbereit ist. Für das Publizieren einer App im App Store muss diese über einen Splash-Screen verfügen. Ist der Ladevorgang abgeschlossen, wird die Startseite angezeigt, diese ist einfach gehalten und besteht im Wesentlichen aus drei Bedienelementen: Zum einen aus einer Segmented Control (siehe Abb. 5.6, rechts), welche dem Benutzer die Auswahl zwischen den verschiedenen Überprüfungsmöglichkeiten ermöglicht (lokal, Server und openVAS), daneben gibt es eine *Info*-Taste in der Navi-

gationsleiste, welche Informationen zu den einzelnen Überprüfungsfunktionen anzeigt, und eine *Analyse-Starten*-Taste, um die lokale und serverseitige Systemüberprüfung zu starten. Damit werden die Anforderungen **FA-2** und **FA-3** (vgl. Kapitel 4.1.1 erfüllt.

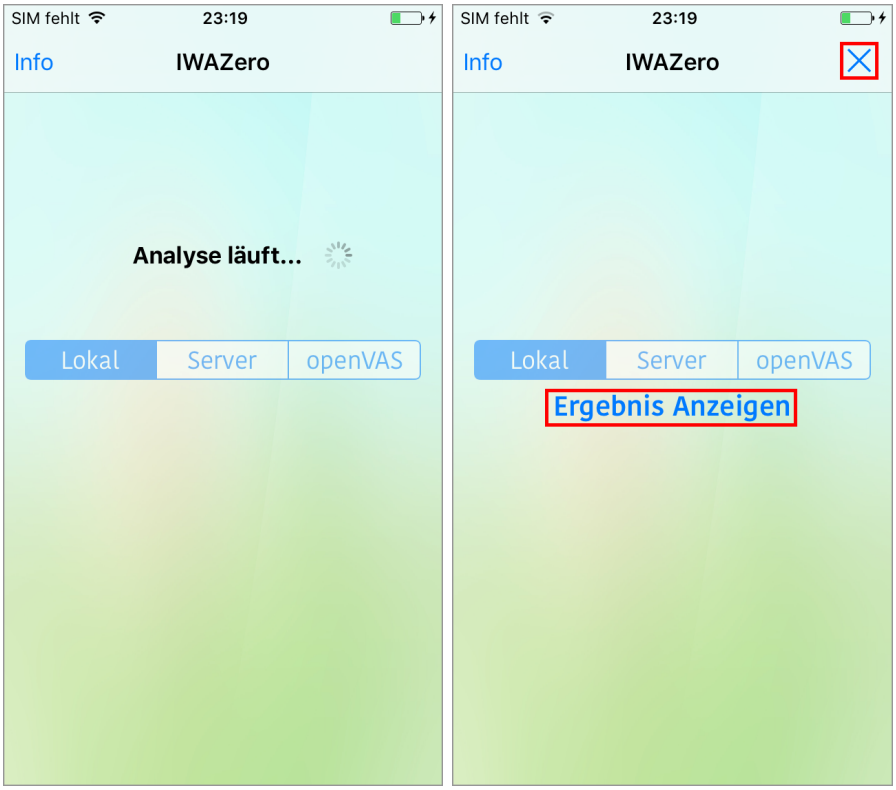

Abbildung 5.7: Eine Überprüfung gestartet und Abgeschlossen

Der Benutzer startet nun die lokale oder serverseitige Systemüberprüfung durch das Betätigen der *Analyse-Starten*-Taste. Nun wird dem Benutzer durch einen Indikator und einen Text (*Analyse läuft ...*) angezeigt, dass die App arbeitet (siehe Abb. 5.7, links). Der Text und der Indikator sind an einen zeitlichen Rahmen gebunden und werden immer vier Sekunden lang angezeigt. In der Zwischenzeit erledigt die Applikation die Aufgaben zur Auswertung der Schwachstellen für die jeweilige Systemversion. Nachdem die Zeit verstrichen ist, erscheinen zwei neue Tasten: eine *Ergebnis-Anzeigen*-Taste und eine weitere Taste in der Navigationsleiste, um das Ergebnis zu löschen (siehe Abb. 5.7, rechts).

Wird die *Ergebnis-Anzeigen*-Taste betätigt, wird ein neues Fenster geöffnet. Dieses Fenster listet die gefundenen Schwachstellen mit der CVE-ID auf und kennzeichnet diese farblich, anhand ihrer Risikoeinstufung (siehe Abb. 5.8, links). Damit werden die funktionalen Anforderungen **FA-2.4** und **FA-3.4** erfüllt (siehe 4.1.1). Das Fenster bietet nun zwei Optionen, zum einen kann auf eine Schwachstelle geklickt werden, um

detaillierte Informationen zu dieser zu erhalten, oder es kann über die Zurück-Taste in der Navigationsleiste zurück zum Startmenü gewechselt werden (siehe Abb. 5.8, links).

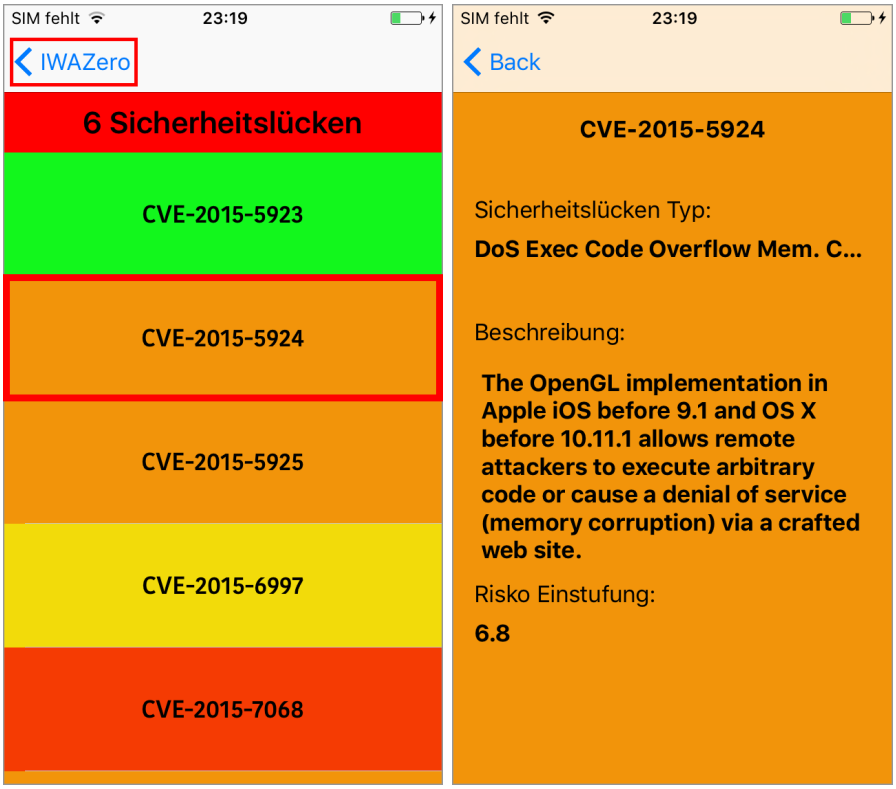

Abbildung 5.8: Ergebnis Anzeige (Tabelle) und Detaillierte Schwachstellen Informationen

Wird eine Schwachstelle angeklickt, werden die detaillierten Informationen zu dieser in einem neuen Fenster geladen (siehe Abb. 5.8, rechts). Dieses Fenster zeigt die CVE-ID, den Sicherheitslückentyp, die Beschreibung und die Risikoeinstufung an und enthält eine Taste in der Navigationsleiste, um zum vorherigen Fenster zurückzuwechseln. Dies ist der Ablauf sowohl für die lokale als auch die serverseitige Systemüberprüfung.

5.6.2 Netzwerk-Schwachstellen-Scanner

Für das Scannen von Netzwerk-Schwachstellen muss auf den Reiter openVAS im der Segmented Control geklickt werden (siehe Abb. 5.6, rechts), dann wird ein neues Fenster geladen. Bevor die App das eigentliche Fenster zum Starten des Netzwerk-Scanners anzeigt, wird zunächst die Anleitung zur Verwendung des openVAS-Scanners angezeigt, auch als Walkthrough bekannt (siehe Abb. 5.9, links).

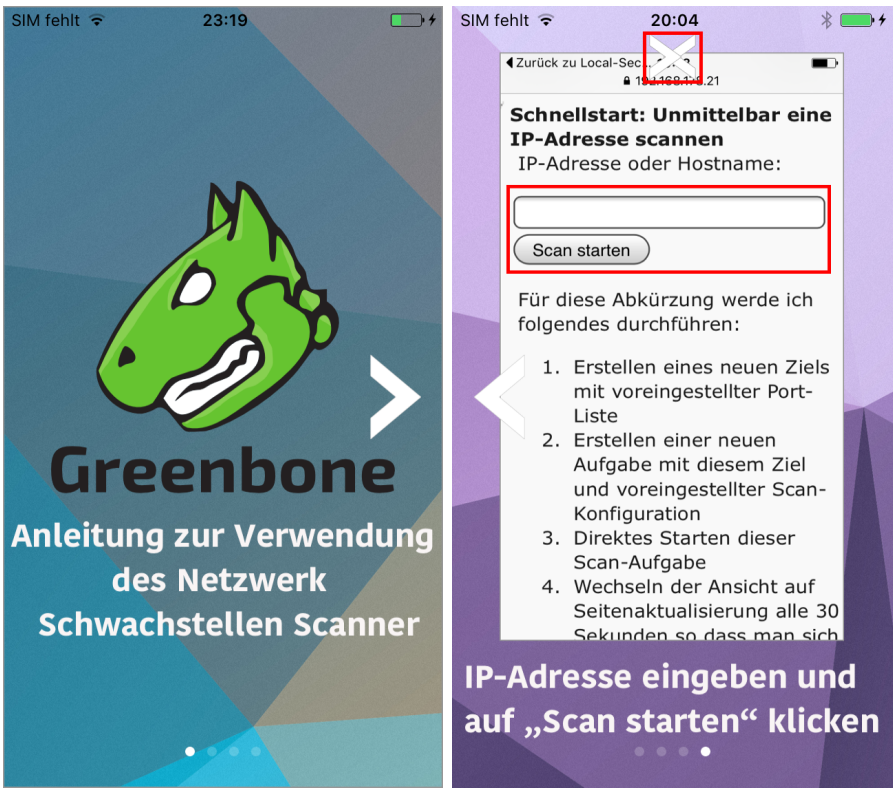

Abbildung 5.9: Walkthrough erste und letzte Seite

Dabei handelt es sich um eine bebilderte Anleitung, welche dem Nutzer anhand eines Beispiels den Ablauf zur Verwendung von openVAS anzeigt. Sobald die letzte Seite der Anleitung erreicht ist, wird eine Schließen-Taste angezeigt (siehe Abb. 5.9, rechts), um den Walkthrough zu verlassen. Der Vorteil gegenüber einer Anleitung in Textform ist, dass der Benutzer hierbei wesentlich schneller versteht, wie die Funktion anzuwenden ist.

Nachdem die Anleitung geschlossen worden ist, wird ein neues Fenster (siehe Abb. 5.10, links) mit jeweils drei Bedienelementen und Informationen angezeigt: zum einen die Zurück-Taste in der Navigationsleiste, eine Anleitungs-Taste, um sich die Anleitung erneut anzusehen, und eine *Analyse-Starten*-Taste, um den Netzwerk-Schwachstellen-

Scanner zu starten. Bei den drei Informationen, die angezeigt werden, handelt es sich um die beiden Logindaten (Benutzername und Passwort) und die IP-Adresse des Gerätes. Diese Informationen werden benötigt, um den Netzwerk-Scanner zu starten und zu verwenden. Wird der Netzwerk-Scanner gestartet, wird das Webinterface vom Netzwerk-Schwachstellen-Scanner im Browser Safari geöffnet .

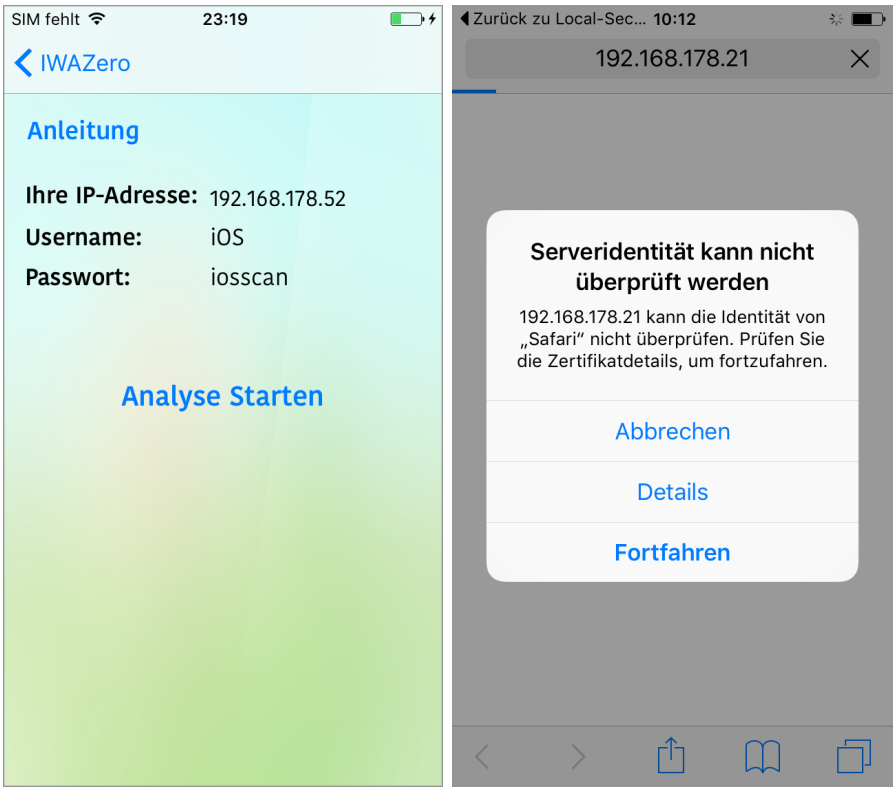

Abbildung 5.10: Netzwerk-Schwachstellen-Scanner Startbildschirm und Safari Zertifikatswarnung

Der Browser öffnet sich und zeigt zunächst die Warnung an, dass die Identität des Serverzertifikats nicht bestätigt werden kann (siehe Abb. 5.10, links). Dies ist zu erwarten und muss mit der *Fortfahren*-Taste bestätigt werden.

Nun erscheint der Login-Bildschirm, hier müssen die zuvor angezeigten Login-Daten eingegeben werden (siehe Abb. 5.11, links). Sind diese korrekt, wird die Startseite des GSA (Netzwerk-Schwachstellen-Scanner) geladen und angezeigt (siehe Abb. 5.11, rechts). Wie zu sehen ist das Web-Interface nicht für die Nutzung von mobile Endgeräten optimiert.

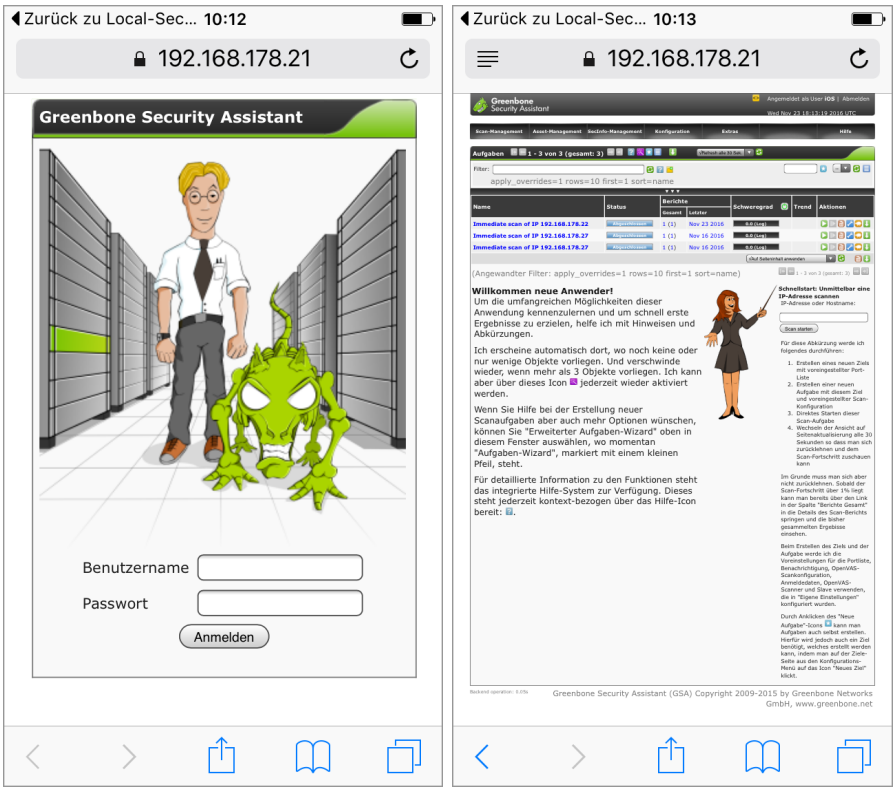

Abbildung 5.11: Login Bildschirm und Startseite vom GSA

Es besteht nun die Möglichkeit, über die Eingabe der IP-Adresse des Gerätes im Quick-Start-Wizard (Schnellstart) einen Scanvorgang zu starten. Wird die IP-Adresse eingegeben und mit dem Start der *Scan*-Taste betätigt, wird der Scanvorgang gestartet und in den Tasks angezeigt (siehe Abb. 5.12, links). Der Scanvorgang kann nun einige Minuten in Anspruch nehmen, ist dieser abgeschlossen, kann der Bericht (Report) dazu angezeigt werden (siehe Abb. 5.12, rechts).

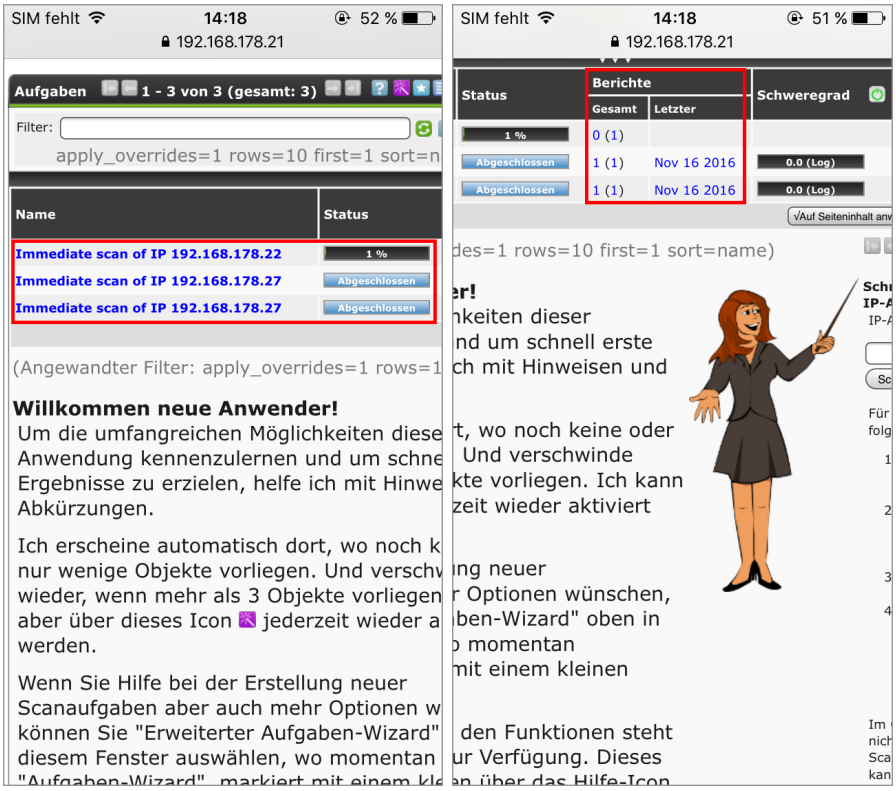

Abbildung 5.12: Taskliste und Report Anzeigen

5.6.3 Info- und Fehlermeldung

Konnte bei der serverseitigen Systemüberprüfung keine Verbindung zum Server hergestellt werden, wird ein Pop-up angezeigt, mit der Begründung, wieso dies nicht möglich war (siehe Abb. 5.13, rechts). Wird beim Startbildschirm die Info-Taste betätigt, wird ein Pop-up mit Informationen zu den einzelnen Überprüfungsfunktionen angezeigt (siehe Abb. 5.13, links).

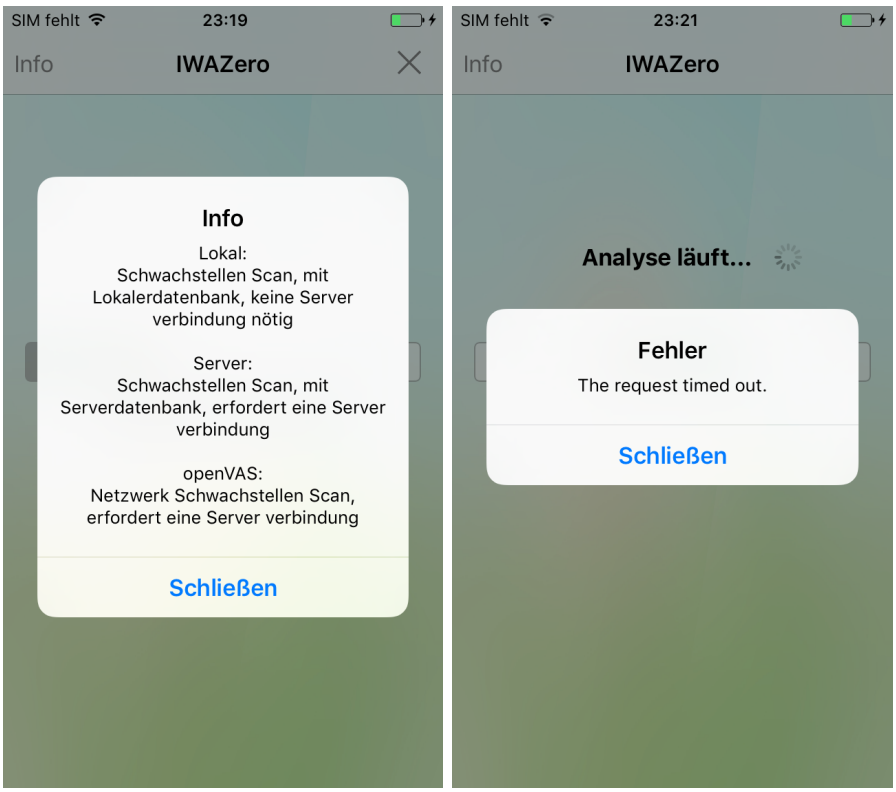

Abbildung 5.13: Pop-up für Info und bei einem Fehler

6 Implementierung

6.1 iOS Softwarearchitektur

Die iOS-Architektur basiert auf einem Schichtenmodell (siehe Abb. 6.1) und dient als Verbindungsglied zwischen Hardware und Applikation. Apps haben keine Möglichkeit der direkten Kommunikation zur Hardware, stattdessen kommunizieren diese über definierte Schnittstellen. [54]

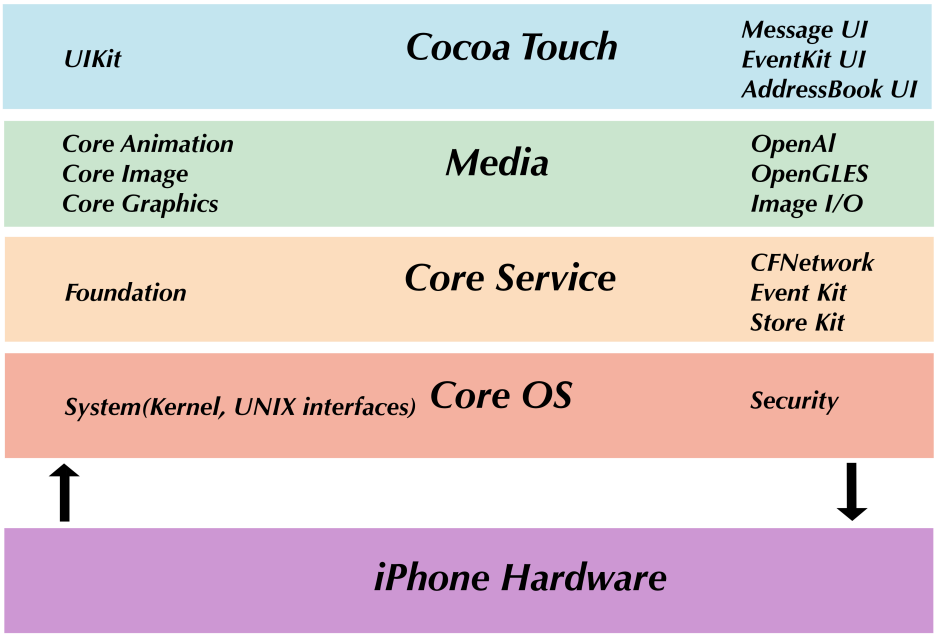

Abbildung 6.1: iOS Software Architektur mit einigen enthaltenen Frameworks

Jede Schicht stellt Schnittstellen und Frameworks zur Erfüllung unterschiedlicher Aufgaben bereit. Diese unterteilen sich in: Core-OS-Layer, Core-Services-Layer und Media-Layer, Cocoa-Touch-Layer.

- ***Core-OS-Layer***
  Das Core-OS bildet die unterste Schicht des Modells, enthält den Kernel, Treiber und Low-Level-UNIX-Schnittstellen des Betriebssystems und dient als Bindeglied zur Hardware. Der Kernel ist für Aspekte wie die virtuelle Speicherverwaltung, Threads, Dateisystem, Netzwerk und Interprozesskommunikation verantwortlich. Die Schicht beinhaltet unter anderem für die Sicherheit notwendige Frameworks:
  - Das *Local-Authentification-Framework* ermöglicht die Authentifizierung durch Touch-ID (siehe 2.1.5).
  - Das *Network-Extension-Framework* bietet eine Hilfestellung bei der Konfiguration und Kontrolle von VPN-Tunneln.

  - Das *Security-Framework* stellt Schnittstellen zur Verwaltung von Zertifikaten, Schlüsselverwaltung (öffentliche und private Schlüssel) und Sicherheitsrichtlinien bereit.

- ***Core-Services-Layer***
  Der Core-Services-Layer beinhaltet grundlegende Systemdienstleistungen für Apps, wichtige Dienste wie *Core-Foundation* und *Foundation-Framework*, welche die Basistypen (z. B. Zeichenkettenverwaltung, Datum und Uhrzeit, Threads) aller Apps definieren. Auch das bereits vorgestellte Sicherheitsmerkmal *Data-Protection* (siehe 2.3.2) ist in dieser Schicht angesiedelt. Darüber hinaus umfasst der Layer CFNetwork, Safari-Services und das WebKit-Framework, welches unter anderem für die Anzeige von HTML-Inhalten verantwortlich ist.

- ***Media Layer*** Der Media-Layer umfasst alle Multimediafunktionen und stellt sowohl die verschiedenen Funktionen für Audio-, Video- und Grafikfunktionalität zur Verfügung als auch Quartz für die 2D-Darstellung, OpenGL ES für 2D- und 3D-Darstellungen und Core-Animation für das Rendering.

- ***Cocoa Touch Layer***
  Der Cocoa-Touch-Layer umfasst alle grundlegenden Frameworks für die Erstellung der grafischen Benutzeroberfläche. Diese Schicht basiert auf den Cocoa-API von Mac OS X und wurde für die völlig andere Steuerung (Multitouch-Display, Gesten) vom iPhone angepasst. Das *UIKit-Framework* ist das wichtigste Framework bei der Entwicklung eigener Applikationen. Dieses beinhaltet grundlegende Klassen für die Erstellung von grafischen und eventgesteuerten Benutzeroberflächen. Textfelder (*UITextField*), Buttons (*UIButton*), Labels (*UILabel*) und Tabellen (UITableView), um nur einige zu nennen, sind Elemente des UIKit, die vom Entwickler für den Zusammenbau der Benutzeroberfläche verwendet werden können. Darüber hinaus ermöglicht das *UIKit* den gekapselten Zugriff auf die Hardware des Gerätes, somit auf Systeminformationen (iOS Version) sowie Kamera-Beschleunigungs- und Neigungssensoren. [55]

6.2 Entwicklungsumgebung

Xcode
Für die Programmierung dieser App wurde die integrierte Entwicklungsumgebung Xcode 8.2 von Apple eingesetzt. Für die Entwicklung mit Xcode 8.2 wird ein PC/Notebook mit macOS (Sierra 10.12 oder El Capitan 10.11.5) vorausgesetzt. Xcode unterstützt im Wesentlichen zwei Programmiersprachen, Swift und Objective-C, unter Verwendung der Cocoa-Frameworks (siehe 6.1).

Swift
Bei Swift handelt es sich um die neuere Programmiersprache, da sie zur Worldwide Developers Conference (WWDC) 2014 vorgestellt wurde. Hierbei handelt es sich um eine objektorientierte Programmiersprache, welche unter anderem von Objective-C, Ruby und Python beeinflusst wurde und bei iOS, OS X, tvOS, watchOS und Linux zum Einsatz kommt. Swift ist eine Alternative zu Objective-C, welche vorher die primäre Programmiersprache zur Entwicklung von Anwendungen darstellte. 2016 hat Apple bereits die dritte Version von Swift vorgestellt und auf lange Sicht wird diese wohl Objective-C völlig ablösen, daher kam in dieser Arbeit Swift als primäre Programmiersprache zum Einsatz.

Interface-Builder
Um die grafische Benutzeroberfläche zu designen, wurde der ebenfalls in Xcode integrierte Interface-Builder verwendet. Dieser ermöglicht es, die komplette Benutzeroberfläche zu erstellen, ohne eine Zeile Code schreiben zu müssen.

6.3 Realisierung GUI

Um die grafische Benutzeroberfläche zu realisieren, wird im Interface-Builder von Xcode ein Storyboard angelegt (siehe Abb. 6.2). Das Storyboard repräsentiert die Benutzeroberfläche einer iOS-Applikation, die einzelnen Bildschirme werden durch UIViewController und UIView-Objekte, welche es in verschiedenen Ausführungen (SplitViewController, PageViewController, TableViewController) gibt, dargestellt. Die Bildschirme werden auch als Szenen bezeichnet, daher ist ein Storyboard zusammengefasst eine Folge von Szenen und spiegelt den Ablauf der Applikation wider. Die Szenen können mit verschiedenen Elementen wie Buttons, Labels, Textfeldern und anderen Dingen versehen werden, dies geschieht alles per Drag & Drop und erfordert keinerlei Implementierungsarbeit. Die Verbindungen zwischen den Szenen werden als Segues bezeichnet und verkörpern den Übergang zwischen zwei ViewControllern. Durch diese ist es möglich, Daten zwischen Szenen zu übertragen.

```
1 override func prepare(for segue: UIStoryboardSegue, sender: Any?)
      {
2     if segue.identifier == "showTable"{
3       let newVC:ResultTableViewController = segue.destination as!
      ResultTableViewController
4       newVC.infos = searchResults;
5     }
6 }
```

Listing 6.1: *prepare*-Methode (MainViewController-Klasse)

Im Listing 6.1 wird die Prepare-Methode der Klasse MainViewController aufgezeigt. Diese Methode wird immer vor einem Übergang (Segue) zu einem anderen ViewController aufgerufen. Ein ViewController kann mit einem Identifier eindeutig gekennzeichnet werden, das ist bei ViewControllern hilfreich, die über mehrere Segues verfügen, wie auch in diesem Fall (siehe Abb. 6.2)). Da nur die Klasse ResultTableViewController das Ergebnis (Zeile 4) einer Auswertung (Server oder lokal) bei einem Übergang erhalten soll, wird der Identifier mit einem if-Statement abgefragt (Zeile 2).

Im Storyboard wird der Startpunkt der Applikation mit einem EntryPoint (siehe Abb. 6.2)) gekennzeichnet, dies ist die erste Szene, die beim Starten der App geladen wird. Die Navigation zwischen den einzelnen Szenen wird durch den NavigationController ermöglicht, dieser muss nur einmal eingefügt werden (siehe Abb. 6.2)).

Damit die grafische Oberfläche einen Zweck erfüllt und keine leere Hülle bleibt, musst das Storyboard mit Codes verknüpft werden. Dies wird durch ein sogenanntes Outlet oder eine Action ermöglicht (siehe Listing 6.2)). Mit einem Outlet (Zeile 1) wird die Verbindung einer Eigenschaft mit @IBOutlet zum Interface-Builder markiert. Mit der Action (Zeile 2) wird eine Methode zur Ereignisbehandlung mit @IBAction zum Interface-Builder markiert.

```
1 @IBOutlet var progressIndikator: UIActivityIndicatorView!
2 @IBAction func startAnalysis(_ sender: UIButton) {}
```

Listing 6.2: Codeabschnitt Outlet und Actions

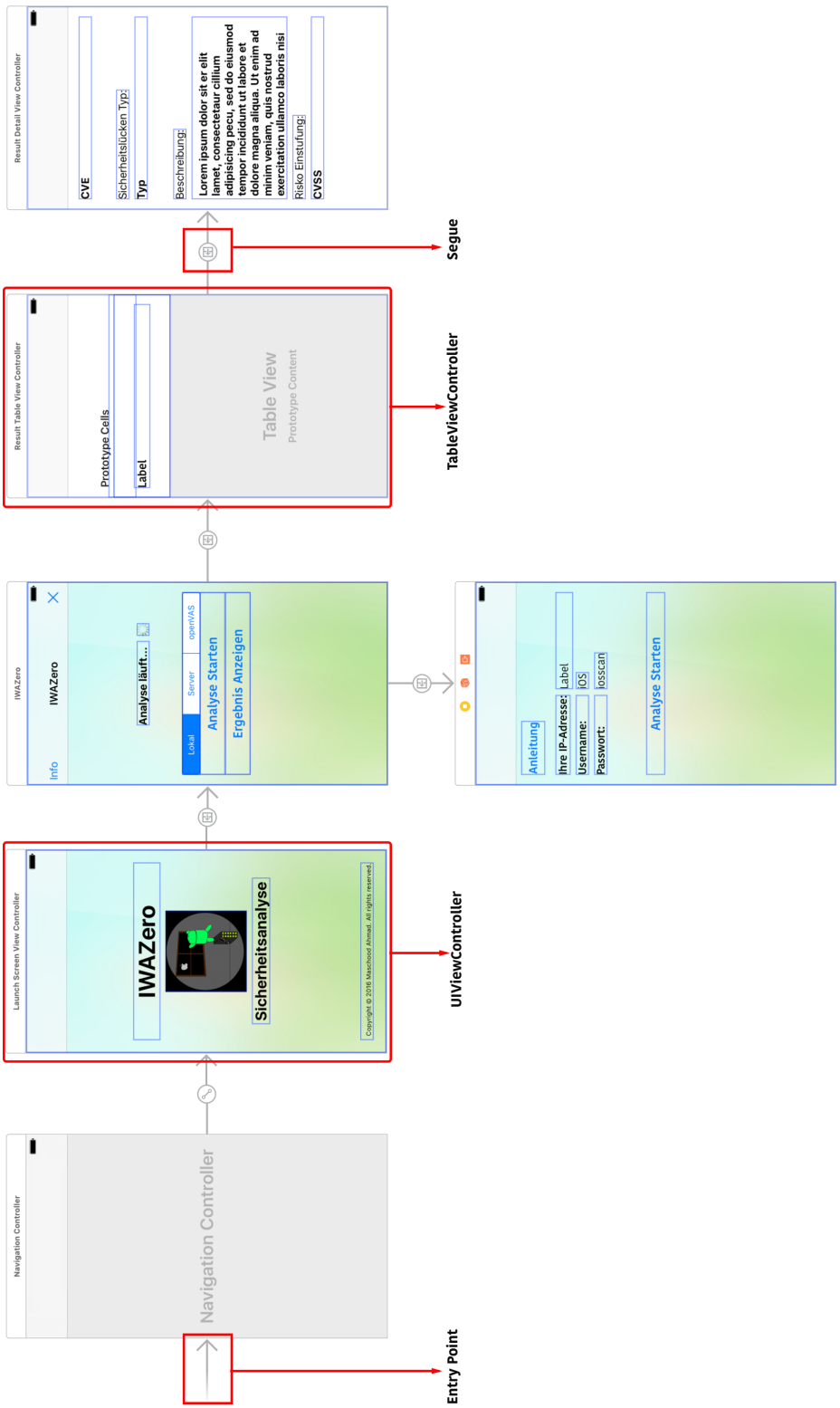

Abbildung 6.2: Applikation Storyboard

6.4 Realisierung Lokale- und Serverfunktionalitäten

In diesem Abschnitt wird auf die Realisierung des Konzeptes (siehe 5) eingegangen, wobei auf die wichtigsten Aspekte zur Implementierung der lokalen und serverseitigen Funktionen eingegangen wird.

Damit eine lokale oder serverseitige Systemüberprüfung durchgeführt werden kann, muss zunächst die Systemversion des Gerätes ermittelt werden. Anhand der Systemversion kann dann geprüft werden, ob Sicherheitslücken in der betreffenden Version vorhanden sind oder nicht.

Um die Systemversion eines Gerätes zu ermitteln, stellt das UIKit-Framework die Klasse UIDevice.current (siehe Listing 6.3, Zeile 1) zur Verfügung. Über den Methodenaufruf systemVersion wird die Systemversion als Zeichenkette (String) zurückgegeben. Der Nachteil einer Zeichenkette ist, dass diese nur auf Gleichheit mit einer anderen Zeichenkette hin verglichen werden kann. Da Schwachstellen jedoch verschiedene Systemversionen betreffen (siehe 3.3.1), wäre es folglich nicht möglich, die Systemversion mit der Version einer Schwachstelle auf kleiner oder größer zu vergleichen. Eine Konventierung der Systemversion zu einer Integer-Zahl, ist nur sehr mühsam möglich da die einzelnen Zahlen der Systemversion von Punkten (z. B. 9.3.5) getrennt werden. Daher kommt die Klasse ProcessInfo().operatingSystemVersion (siehe Listing 6.3, Zeile 4) aus dem Foundation-Framework zum Einsatz. Diese ermöglicht es, die Systemversion über den jeweiligen Methodenaufruf für die Major , Minor und Patch-Version in Form einer Integer-Zahl zurückzuliefern.

```
1 let deviceInfo = UIDevice.current
2 let systemVersion: String = deviceInfo.systemVersion
3
4 let os = ProcessInfo().operatingSystemVersion
5 let majorVersion = os.majorVersion
6 let minorVersion = os.minorVersion
7 let patchVersion = os.patchVersion
```

Listing 6.3: Codeabschnitt Systemversion ermitteln (DeviceInfo-Klasse)

Für den Aufruf der Systemüberprüfung (lokal oder serverseitig) wird die Ereignisbehandlungsmethode startAnalysis (siehe Listing 6.4, Zeile 1) implementiert, diese wird zur Verknüpfung mit dem Storyboard mit @IBAction markiert. Damit wird der Programmcode mit der grafischen Benutzeroberfläche verbunden, reagiert somit auf Benutzereingaben und führt den entsprechenden Code zu einem Event aus. Ein Event wäre beispielsweise das Drücken eines Buttons.

Bei der lokalen Systemüberprüfung wird über die Instanz zur Klasse LocalSystemAnalysis (siehe Listing 6.4, Zeile 6) zunächst die Methode jsonParsing aufgerufen. Dann wird ein Hintergrundthread erzeugt, welcher den Aufruf für das Abholen des Ergebnisses aus der Systemüberprüfung (siehe Listing 6.4, Zeile 8) und die Benutzeroberfläche aktuali-

siert (siehe Listing 6.4, Zeile 9). Der Programm-Block wird erst nach dem Ablauf einer festgelegten Zeit ausgeführt, in diesem Fall sind es 4 Sekunden. Dies wird so gehandhabt, um dem Benutzer der Applikation den Eindruck zu vermitteln, dass die Überprüfung eine gewisse Zeit in Anspruch nimmt. Währenddessen arbeitet der Hauptthread weiter und aktualisiert die Benutzeroberfläche (siehe Listing 6.4, Zeile 11).

```
1  @IBAction func startAnalysis(_ sender: UIButton) {
2
3  switch selectedSource.selectedSegmentIndex {
4
5  case 0:
6      localScan.jsonParsing()
7      DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() +
       dispatchTime , execute: {
8          self.searchResults = self.localScan.getLocalResults()
9          self.finishedAnalysis()
10     })
11     progressUpdate()
12
13 case 1:
14     let search = device.getSystemVersion()
15     serverscan.connectToServer(searchTerm: search)
16     progressUpdate()
17     default: break
18   }
19 }
```

Listing 6.4: *startAnalysis*-Methode (MainViewController-Klasse)

Der Ablauf bei der Server-Systemüberprüfung ist etwas anders, dort wird zunächst über die Instanz zur Klasse DeviceInfo die Systemversion ermittelt (siehe Listing 6.4, Zeile 14). Diese kann dann beim Aufruf der Server-Systemüberprüfung übergeben werden (siehe Listing 6.4, Zeile 15). Abschließend wird die Benutzeroberfläche aktualisiert. Um ein Ergebnis aus der angestoßenen Systemüberprüfung zu erhalten, wird das Protokoll (siehe Listing 6.5) von der Klasse MainViewController implementiert.

```
1 protocol ServerConnectionDelegate:class {
2 func successful()
3 func failed(error: Error?)}
```

Listing 6.5: ServerConnectionDelegate Protokoll (ServerSystemAnalysis-Klasse)

Das Listing 6.6 zeigt die Implementierung der successful-Methode aus dem ServerConnectionDelegate-Protokoll. Beim Aufruf dieser wird zunächst – wie auch bei der lokalen Überprüfung – der Code-Abschnitt (siehe Listing 6.6, Zeile 3, 4) erst nach 4 Sekunden von einem Hintergrundthread ausgeführt. In beiden Fällen wird abschließend das Ergebnis im lokalen Klassenattribut searchResults gesichert und die Benutzeroberfläche aktualisiert, sodass ein Zugriff der *MainViewController*-Klasse auf das Ergebnis möglich ist.

```
1 internal func successful() {
2   DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() +
     dispatchTime, execute: {
3     self.searchResults = self.serverscan.getSearchResults()
4     self.finishedAnalysis()
5   })}
```

Listing 6.6: *successful*-Methode (MainViewController-Klasse)

6.4.1 Lokale Systemüberprüfung

Für die Lokale Systemüberprüfung werden im wesentlichen zwei Methoden *jsonParsing* (siehe Listing 6.7) und *versionEvaluation* (siehe Listing 6.8) welche in der Klasse *LocalSystemAnalysis* implementiert sind verwendet.

Um eine Lokale Systemüberprüfung zu ermöglichen wurde eine JSON-Datei, auf Basis des im Konzept vorgestellten Modells (vgl. Listing 5.2), erstellt und mit Metadaten gefüllt. Anschließend wurden einige Sicherheitslücken zu den einzelnen Systemversionen ab iOS 9 (z. B. 9.0.2, 9.1.0, 9.3.5, 10.0) in die Metadaten gespeichert. Sicherheitslücken in Systemversionen vor iOS 9 wurde nicht berücksichtigt, aufgrund der Installations-Voraussetzung (vgl. Kapitel 4.1.2) der App.

Im Listing 6.7 ist die jsonParsing-Methode zusehen, welche die Metadaten aus der JSON-Datei einliest (siehe 6.7, Zeile 4 - 6). Jede Sicherheitslücke aus den Metadaten wird dann geparst und anhand des Aufrufes der versionEvaluation-Methode (siehe 6.7, Zeile 18), in einem if-Statement, auf Zugehörigkeit zur Systemversion geprüft und erst dann im Klassenattribut localResults (siehe 6.7, Zeile 19) gespeichert.

```
1 public func jsonParsing(){
2 localResults.removeAll()
3
4   if let file = Bundle.main.path(forResource: "metadaten", ofType
     : "json") {
5     let url = URL(fileURLWithPath: file)
6     let data = try! Data(contentsOf: url)
7     let jsonResult = try! JSONSerialization.jsonObject(with: data
     , options: JSONSerialization.ReadingOptions.mutableContainers)
8
9     for anItem in jsonResult as! [Dictionary<String, AnyObject>]
     {
```

```
10          let sName = anItem["cveId"] as? String
11          let sCvss = anItem["vScore"] as? Double
12          let sDesc = anItem["info"] as? String
13          let sType = anItem["vType"] as? String
14          let sMajor = anItem["Major"] as? Int
15          let sMinor = anItem["Minor"] as? Int
16          let sPatch = anItem["Patch"] as? Int
17
18          if versionEvaluation(sVMajor: sMajor!, sVMinor: sMinor!,
    sVPatch: sPatch!){
19            localResults.append(SecurityVulnerability(cveId: sName
    !, vScore: sCvss!, description: sDesc!, vTyp: sType!))
20          }//endif
21       }//endfor
22    }//endif
23 }
```

Listing 6.7: *jsonParsing*-Methode (LocalSystemAnalysis-Klasse)

In der versionEvaluation-Methode (siehe Listing 6.8) werden die, zuvor aus den Lokalen Metadaten geparsten, Attribute sMajor, sMinor und sPatch (siehe Listing 6.7, Zeile 14 - 16) übergeben. Diese Attribute Kennzeichen die Systemversion in welcher, die Sicherheitslücke bereits geschlossen wurde. Aus diesem Grund muss die Systemversion welche die Sicherheitslücken betrifft neuer als die der Systemversion (siehe Listing, 6.7, Zeile 11).

```
1  private func versionEvaluation(sVMajor: Int, sVMinor: Int,
       sVPatch: Int) -> Bool {
2  var version: Bool
3  version = false
4
5  var deviceVersion: String
6  var securityVulnerabilityVersion: String
7
8  deviceVersion = "\(deviceMajor)" + "\(deviceMinor)" + "\(
       devicePatch)"
9  securityVulnerabilityVersion = "\(sVMajor)" + "\(sVMinor)" + "\(
       sVPatch)"
10
11 if Int(securityVulnerabilityVersion)! > Int(deviceVersion)! {
12 version = true
13 }
14 return version
15 }
16 }
```

Listing 6.8: *versionEvaluation*-Methode (LocalSystemAnalysis-Klasse)

6.4.2 Server Systemüberprüfung

Bei einer Serverseitige Systemüberprüfung wird eine Anfrage zu Sicherheitslücken, bezüglich einer Betriebssystemversion, an den Server gesendet. Der Server Antwort darauf mit einem JSON-Response (siehe 5.2), welcher in der Struktur mit den Lokalen JSON-Datei (vgl. Listing 6.7) bis auf die Ausnahmen das keine Major, Minor- und Patchversion enthalten sind, identisch ist. Anschließend wird die Antwort geparst und im Lokalen Klassenattribut (searchResults) gesichert.

Die Serverseitge Systemenüberprüfung wird durch die *connectToServer*-Methode (siehe Listing 6.9) und *updateSearchResults*-Methode realisiert. Hierfür ruft die MainViewController-Klasse über die Instanz zur *ServerSystemAnalysis*-Klasse die *connectToServer*-Methode auf. Dabei wird die Systemversion als Suchparameter (siehe Listing 6.9, Zeile 1) übergeben. Die URL zum Server ist bis auf den Suchparameter hart codiert (Zeile 5) und wird mit der im Konzept (siehe 5.3.2) vorgestellten URLSession (Zeile 9) aufgerufen. War der Verbindungsaufbau erfolgreich (Zeile 16) wird zunächst die *updateSearchResults*-Methode aufgerufen und dieser die Daten (Sicherheitslücken) der Serverantwort übergeben (Zeile 17). Die *updateSearchResults*-Methode parst die Daten (Sicherheitslücken) genau wie die *jsonParsing*-Methode (siehe Listing 6.7), mit dem unterschied das keine Auswertung der Daten stattfindet. Dies ist nicht nötig da dies durch den Server erledigt wird. Anschließend wird die *successful*-Methode der *MainViewController*-Klasse (siehe Listing 6.6) durch die *ServerSystemAnalysis*-Klasse aufgerufen (Zeile 18).

```
1  public func connectToServer(searchTerm: String) {
2
3  let serverUrl = URL(string: "http://192.168.178.72:8080/
       RestServerWeb/api/v1/service/ios/\(searchTerm)")
4  var request = URLRequest(url:serverUrl!);
5  request.httpMethod = "GET";
6
7  dataTask= URLSession.shared.dataTask(with: request as URLRequest)
8        {data, response, error in
9
10   if let httpResponse = response as? HTTPURLResponse,
       httpResponse.statusCode == 200{
11     self.updateSearchResults(data: data)
12     self.delegate?.successful()
13   }else{
14     self.delegate?.failed(error: error)
15     }
16   }
17
18 dataTask?.resume()
19 }
```

Listing 6.9: *connectToServer*-Methode (ServerSystemAnalysis-Klasse)

6.4.3 Netzwerk-Schwachstellen Scanner

Das Web-Interface (GSA) des Netzwerk-Schwachstellen Scanner wird über den Safari aufgerufen. Des Weiteren werden Informationen (Logindaten und IP-Adresse) zur Verwendung des Netzwerk-Schwachstellen Scanner durch die NetworkScanViewController-Klasse bereitgestellt.

```
1 @IBAction func startNetworkScanner(_ sender: UIButton) {
2 UIApplication.shared.openURL(URL(string: "http://192.168.178.21")
    !)
3 }
```

Listing 6.10: *startNetworkScanner*-Methode (NetworkScanVieController-Klasse)

Um das Web-Interface zu öffnen wurde eine Methode zur Ereignisbehandelung implementiert (siehe 6.3), diese wird durch die Betätigung der *Analyse-Starten*-Taste ausgelöst (siehe Listing 6.10, Zeile 1). Anschließend wird die im Konzept (siehe 5.3.2) vorgestellte *openURL*-Methode verwendet um die URL im Browser zu öffnen (Zeile 2). Ab diesen Zeitpunkt wird die App verlassen und der restliche Ablauf (siehe 5.6.2) wird durch den Netzwerk-Schwachstellen Scanner im Browser abgewickelt.

7 Test

7.1 Tests

Um den Erfolg der Implementierung zu prüfen, wurden Unit-Tests geschrieben um die wichtigsten Methoden auf ihr Funktionalität zu prüfen. Es werden die verwendeten Testwerkzeuge vorgestellt die für das Testen der App verwendet wurden.

7.1.1 Testen auf iPhone Simulator

Der iPhone Simulator ermöglicht das schnelle Testen während des Entwicklungsprozesses ohne das Hardware benötigt wird. Dabei wird unter Mac OS X das Verhalten von iOS weitestgehend analog zum iPhone simuliert. Der Simulator ist als ein Teil von Xcode installiert und ermöglicht das Testen und Debuggen von Prototypen und Test-Builds. Zudem ist es mögliche aktuelle und einige ältere Betriebssystemversionen zu simulieren. Was zu beachten ist das der Simulator nur ein vorläufiges Testwerkzeug ist bevor die App auf echter Hardware getestet wird.

7.1.2 Testen auf Hardware

Für das Testen der App auf Hardware wurde ein iPhone SE mit der Systemversion iOS 9.3.3 verwendet, welches durch HAW Hamburg zur Verfügung gestellt wurde. Um Apps zu Testzwecken auf den iPhone auszuführen, wird von Xcode ein Provisoning Profile benötigt. Das Provisoning Profile wurden ebenfalls durch die HAW Hamburg mit der Aufnahme meiner Apple ID, in das Apple Developer Programm, bereitgestellt. Anschließend muss das iPhone noch im Entwickler Portal anhand des Unique Device Identifier (UDID) registriert werden.

7.2 Unit-Tests

Die Lokale und serverseitige Systemüberprüfung wird anhand von Unit-Tests auf ihre Funktionalität geprüft. Dabei werden die wichtigsten Methoden zur Verwendung dieser getestet.

Testszenario: *UI Verhalten*
Das Folgende Szenario wurde zur Evaluierung des UI Verhalten beim Aufruf der Lokalen und serverseitigen Systemüberprüfung eingesetzt.

Kriterien:
Der Benutzer startet die Lokale oder serverseitige Systemüberprüfung im App Startbildschirm, dabei wird im Segemented Control entweder Local (Index: 0) oder Server (Index: 1) ausgewählt und die *Analyse-Starten*-Taste betätigt. Die *Analyse-Starten*-Taste ist über einen @IBAction mit der *startAnalysis*-Methode verknüpft. Es wird geprüft ob die entsprechenden Funktion (Lokale (*jsonParsing*-Methode) oder servseitiger Systemüber-

prüfung (*connectToServer*-Methode)) korrekt aufgerufen wird.

Testablauf:
Es werden zwei Mock-Klassen erstellt welche die Schnittstellen der Klassen *LocalSystemAnalysis* und *ServerSystemAnalysis* implementieren. Mit den Mock-Klassen wird das Verhalten der Klassen (siehe Listing 7.1, Zeile 2 und 8) simuliert und die jeweiligen Methoden (siehe Listing 7.1, Zeile 4 und 10) der Klassen überschrieben und die zum Testfall benötigten Werte festgelegt (siehe Listing 7.1, Zeile 5 und 11).

```
1  //Connection Mock Local
2  class ConnectionMockLocal: LocalSystemAnalysis {
3    var calledJsonParsing = false
4    override func jsonParsing() {
5      calledJsonParsing = true
6  }
7  //Connection Mock Server
8  class ConnectionMockServer: ServerSystemAnalysis {
9    var calledConnectToServer = false
10   override func connectToServer(searchTerm: String) {
11     calledConnectToServer = true
12   }
13 }
```

Listing 7.1: Connection Mock (Lokal und Server)

Die Klassenattribute der MainViewController-Klasse (controller) werden entsprechend des benötigten Testfalls angepasst (siehe Listing (7.2), Zeile 2-5 und 10-13).

Auswertung:
Die XCTAsserts (siehe Listing (7.2, Zeile 7 und 15) lieferten das erwartete Ergebnis und waren somit Erfolgreich.

```
1  //Mock Server
2  controller.selectedSource.selectedSegmentIndex = 1
3  let mockServer = ConnectionMockServer()
4  controller.serverscan = mockServer
5  controller.startAnalysis(controller.startAnalysis)
6  // test
7  XCTAssert(mockServer.calledConnectToServer == true)
8
9  //Mock Local
10 controller.selectedSource.selectedSegmentIndex = 0
11 let mockLocal = ConnectionMockLocal()
12 controller.localScan = mockLocal
13 controller.startAnalysis(controller.startAnalysis)
14 // test
15 XCTAssert(mockLocal.calledJsonParsing == true)
```

Listing 7.2: Unit-Test *startAnalysis*-Methode

Testszenario: Auswertung
Im Rahmen dieses Szenario wird die *versionEvaluation*-Methode zur Auswertung von Sicherheitslücken bei der Lokalen Systemüberprüfung getestet.

Kriterien:
Die *jsonParsing*-Methode parst eine Sicherheitslücke und übergibt diese der versionEvaluation-Methode. Die *versionEvaluation*-Methode muss überprüfen ob die Sicherheitslücke die Systemversion des Gerätes betrifft.

Testablauf:
Um die Systemversion eines Gerätes zu simulieren wurde die createDeviceStub-Methode implementiert. Diese ermöglicht ein Gerät mit beliebiger Systemversion zu erzeugen, unter der Angabe von Major-, Minor- und Patchversion. Für die Testfälle wurden jeweils ein Gerät mit der Systemversion 9.3.3 erzeugt (siehe Listing 7.3, Zeile 2 und 7). Es wurden zwei Testfälle der *versionEvaluation*-Methode simuliert. Zum einen wurde ein Sicherheitslückenversion übergeben die größer (siehe Listing 7.3, Zeile 3) der Systemversion ist und zum anderen eine die kleiner der Systemversion (siehe Listing 7.3, Zeile 8) ist.

Auswertung:
Die XCTAsserts (siehe Listing (7.2, Zeile 7 und 15) lieferten das erwartete Ergebnis und waren somit Erfolgreich.

```
1 func testTrueRequestVersionGreaterThanDeviceVersion() {
2 let metadata = LocalSystemAnalysis(device: createDeviceStub(major
    : 9, minor: 3, patch: 3))
3 XCTAssert(metadata.versionEvaluation(sVMajor: 10, sVMinor: 0,
    sVPatch: 0) == true)
4 }
5
6 func testFalseRequestVersionSmallerThanDeviceVersion() {
7 let metadata = LocalSystemAnalysis(device: createDeviceStub(major
    : 9, minor: 3, patch: 3))
8 XCTAssert(metadata.versionEvaluation(sVMajor: 9, sVMinor: 3,
    sVPatch: 0) == false)
9 }
```

Listing 7.3: Unit-Test *versionEvaluation*-Methode

Testszenario: Parser
Im Rahmen dieses Szenario wird die *updateSearchResults*-Methode zum Parsen von Sicherheitslücken bei Serverseitiger Systemüberprüfung getestet.

Kriterien:
Die vom*connectToServer*-Methode erhaltene Serverantwort wird an die *updateSearchResults*-Methode übergeben um diese zu Parsen.

Testablauf:
Es wird eine Test JSON-Datei angelegt und eingelesen (siehe Listing (7.4, Zeile 1). Zu Testzwecken wird *updateSearchResults*-Methode modifiziert, damit diese das geparste Objekt als String zurück gibt. Der Rückgabewert wird dann mit dem Erwarteten Wert verglichen.

Auswertung:
Die XCTAssertEqual (siehe Listing (7.4, Zeile 5) ist mit dem erwarteten Wert identisch und damit Erfolgreich.

```
1 let file = Bundle.main.path(forResource: "test", ofType: "json")
2 let url = URL(fileURLWithPath: file!)
3 let data = try! Data(contentsOf: url)
4
5 XCTAssertEqual(updateSearchResults(data: data), "CVE-2015-5924")
```

Listing 7.4: Unit-Test *updateSearchResults*-Methode

8 Fazit

8.1 Zusammenfassung

Die immer weiter fortschreitende digitale Vernetzung der Welt hat durch das Smartphone eine neue Dimension erreicht. Dieses verfügt nicht nur über gute Vernetzungsmöglichkeiten und den Zugriff auf eine stetig zunehmende Zahl von Applikationen (Apps), sondern hat auch und gerade eine von Jahr zu Jahr steigende Nutzerschaft. Die gleichzeitig wachsende Zahl an gefundenen Sicherheitslücken bei mobilen Geräten ist genau darauf zurückzuführen: Cyberkriminelle haben seit ca. 2014 das Smartphone als neues Ziel auserkoren, was kaum verwunderlich ist, denn Smartphones bieten Angreifern schließlich Unmengen an Daten jeglicher Art: Zugangsdaten zu privaten und geschäftlichen E-Mails, Finanzdienstleistungen, Käuferkonten und vieles mehr. Der Hersteller Apple versucht, durch regelmäßige Updates, Anpassungen der Sicherheitsarchitektur sowie die Entwicklung von neuen Sicherheitsmerkmalen die Benutzer des iPhones besser zu schützen. Letzten Endes gilt diesbezüglich jedoch Folgendes:

> *„Die Organisationen stecken Millionen von Dollars in Firewalls und Sicherheitssysteme und verschwenden ihr Geld, da keine dieser Maßnahmen das schwächste Glied der Sicherheitskette berücksichtigt: Die Anwender und Systemadministratoren.“-Kevin Mitnick (Hacker)*

Wie dieses Zitat zeigt, können von Organisationen zwar Unmengen von Geld für Sicherheitssysteme investiert werden, wenn der Benutzer nachher jedoch ein so einfaches Passwort wie „Hamburg1234“ verwendet, darf sich dieser nicht wundern, zum Opfer von Cyberkriminellen zu werden. Vor diesem Hintergrund gilt das Interesse des Autors der vorliegenden Arbeit nicht nur der Entwicklung einer Applikation für iOS zur Überprüfung eines mobilen Endgerät auf bekannte Sicherheitslücken, sondern der Leser wird zugleich darüber aufgeklärt, wie wichtig die Sicherheit auf bzw. von mobilen Geräten ist.

Zu diesem Zweck wurde zunächst in den Grundlagen auf die Sicherheitsarchitektur von iOS eingegangen. Dabei wurde ein Einblick in die von iOS verwendeten Sicherheitsmechanismen für die System-, Applikations- und Datensicherheit gewährt. Anschließend wurden relevante Begriffe rund um das Thema Schwachstellen geklärt, beispielweise, welche Arten hiervon es gibt, und, wie gefundene Schwachstellen von CVEs eindeutig gekennzeichnet werden.

Im Anschluss daran wurde eine Sicherheitsanalyse des Betriebssystems iOS vorgenommen. Dafür wurde zunächst auf die Bedeutung von Sicherheit in mobilen Betriebssystemen eingegangen und darauf, welche Auswirkung diese für den Einzelnen haben. Dann wurden die verwendeten Sicherheitspraktiken und Risiken von iOS vorgestellt, wie der Walled Garden (geschlossenes Ökosystem) und Jailbreaks (Umgehung von Nutzerbeschränkung). Des Weiteren wurde exemplarisch der Ablauf für die Datensicherheit erläutert und, auf welche Sicherheitsmechanismen diese zurückgreift. Dann wurde ein Vergleich zwischen den verschiedenen relevanten mobilen Betriebssystemen (iOS, Android, Windows Phone)

in Bezug auf deren Sicherheitsmerkmale durchgeführt. Zum Schluss wurde ein Einblick in die aktuellen Sicherheitslücken in iOS als auch plattformunabhängig vermittelt.

In Bezug auf die Sicherheitsanalyse wurden sowohl funktionale als auch nicht funktionale Anforderungen zur Umsetzung einer mobilen Anwendung ermittelt. Basierend auf der Anforderungsanalyse wurden spezifische Abläufe der Applikation in Form von konkreten Anwendungsfällen dargestellt.

Im nächsten Schritt wurden zunächst die Systemarchitektur und die darin enthaltenen Komponenten wie Client oder Server vorgestellt. Es wurde in Bezug auf die in der vorliegenden Arbeit nicht explizit umgesetzten Teile der Systemarchitektur eingegangen und die sich daraus ergebenden Vorgaben wurden vorgestellt. Im Anschluss wurde ein Entwurf zur Umsetzung der Applikation erstellt. Dieser beinhaltet Erläuterungen zu den Entwicklungstechnologien wie Programmbibliotheken und den eingesetzten Frameworks (z. B. JSON und NSURLSession). Anhand des Klassenmodells und von Sequenzdiagrammen wurden Abhängigkeiten der Klassen in den Abläufen verdeutlicht. Schließlich beinhaltete es den Entwurf zur Umsetzung der grafischen Benutzeroberfläche.

Nachfolgend wurde auf die iOS-Systemarchitektur und die darin enthaltenen Frameworks eingegangen und die zur Umsetzung der App verwendete Entwicklungsumgebung wurde vorgestellt. Entsprechend den Anforderungen und dem Konzept wurde dann die Applikation entwickelt. Zunächst wurde in dem Zusammenhang auf die Umsetzung der grafischen Oberfläche durch das Storyboard eingegangen. Anschließend daran wurden die Implementierung der wichtigsten Methoden aufgezeigt.

Zum Schluss wurden die Relevanten Methoden zur Umsetzung der primären Funktionen anhand von Unit-Test auf die Gültigkeit geprüft.

8.2 Ausblick

Zu Beginn dieser Arbeit wurde das Ziel festgelegt, eine Applikation für iOS zu entwickeln, welche dem Nutzer dieser Anwendung Hinweise zu bestehenden Sicherheitslücken in seinem mobilen Endgerät gibt. Ein teil Sicherheitslücken wurde zum einen lokal in der Applikation gesichert und zum anderen, stellt ein Server weitere Sicherheitslücken für die Applikation bereit. Darüber hinaus ist es möglich einen Netzwerk-Schwachstellen-Scanner im Webbrowser zu öffnen und das Endgerät auf Sicherheitslücken im Netzwerk zu prüfen. Hierzu wurden im Rahmen dieser Arbeit Anforderungen (FA-1 bis FA-5.4) ermittelt, welche allesamt erfüllt werden konnten. Bei der Umsetzung des Netzwerk-Schwachstellen-Scanners, welcher durch den Server bereitgestellt wurde, lief hingegen nicht alles optimal. Dieser wurde in einer anderen Thesis entwickelt und war bis zum Zeitpunkt der Abgabe der vorliegenden Thesis (29.11.2016) noch nicht komplett fertiggestellt. Die Walktrough-Anleitung stellte daher einen kleinen Kompromiss dar, um die Benutzererfahrung, im

Hinblick auf die Verwendung des Netzwerk-Schwachstellen-Scanners zu verbessern.

Die Applikation wurde im Hinblick auf die Anforderungen fertig gestellt, jedoch bestehen einige Möglichkeiten zur Erweiterung. Der Aufruf des Netzwerk-Schwachstellen-Scanner wird über den Browser abgewickelt. Eine Verbesserung wäre, die Integration einer WebView in die App. Die Lokale und serverseitige Systemüberprüfung, könnte durch eine einzelne Funktion zur Systemüberprüfung ersetzt werden. Hierfür müsste eine Lokale Datenbank (SQLite, Core Data) angelegt werden, welche dann vom Server mit Aktualisierungen versorgt wird.

Das iOS Betriebssystem wurde auf der Basis der Cocoa API entwickelt, welche im direkten Zusammenhang mit Mac OS X steht. Daher wurden des Öfteren, bei der Recherche, Sicherheitslücken entdeckt die in beiden Plattformen auftraten. Ein Großteil dieser Sicherheitslücken wurde im Safari Browser vorgefunden. Des Weiteren müsste die Relevanz der App auf den iPad und iPodtouch geprüft werden. Dies wären Themen die in einer weiteren Arbeit erörtert werden könnten .

Literatur

[1] Apple. *iOS-Sicherheit*. white paper. Apple, 2015. URL: https://www.apple.com/de/business/docs/iOS_Security_Guide.pdf (besucht am 12. 02. 2016).

[2] muut.com. *Versioning Example*. 25. Okt. 2016. URL: https://muut.com/help/img/semantic.png (besucht am 25. 10. 2016).

[3] Fakultät Informatik TU Dresden. *The L4 microkernal Family*. 2016. URL: https://os.inf.tu-dresden.de/L4 (besucht am 28. 10. 2016).

[4] Chaos Computer Club. *Chaos Computer Club hackt Apple TouchID*. Hrsg. von Frank. 2013. URL: http://www.ccc.de/de/updates/2013/ccc-breaks-apple-touchid (besucht am 12. 08. 2016).

[5] Apple. *About App Sandbox*. 11. Feb. 2014. URL: https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html (besucht am 24. 05. 2016).

[6] John Barker Elaine; Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Hrsg. von National Institute of Standards und Technology Special Publication 800-90A. 2015. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf (besucht am 27. 10. 2016).

[7] Prof. Dr. Claudia Eckert. *IT-Sicherheit*. Konzepte-Verfahren-Protokolle. Oldenbourg Verlag, 2014. ISBN: 978-3-486-72138-6.

[8] Computervirus. *Was ist Malware?* 2016. URL: http://computervirus.de/was-ist-malware/ (besucht am 19. 11. 2016).

[9] Computerbetrug.de. *Spyware: Spione auf dem Computer*. 2016. URL: http://www.computerbetrug.de/spyware-spione-auf-dem-computer (besucht am 19. 11. 2016).

[10] Marvin the Robot. *Was sind Exploits und warum sind sie so gefährlich?* Hrsg. von Kaspersky. 2015. URL: https://blog.kaspersky.de/exploits-problem-explanation/5905/ (besucht am 01. 08. 2016).

[11] CVEDetails. *Apple iPhone OS Vulnerability Statistics*. 2016. URL: http://www.cvedetails.com/product/15556/Apple-Iphone-Os.html?vendor_id=49 (besucht am 01. 11. 2016).

[12] itwissen.info. *DoS (denial of service) DoS-Attacke*. 2016. URL: http://www.itwissen.info/definition/lexikon/denial-of-service-DoS-DoS-Attacke.html (besucht am 10. 05. 2016).

[13] Wikipedia. *Memory corruption*. 5. Mai 2016. URL: https://en.wikipedia.org/wiki/Memory_corruption (besucht am 10. 05. 2016).

[14] Victor van der Veen u. a. “Memory Errors: The Past, the Present, and the Future”. In: *Research in Attacks, Intrusions, and Defenses*. 2012, S. 86–106. DOI: 10.1007/978-3-642-33338-5_5. URL: http://dx.doi.org/10.1007/978-3-642-33338-5_5.

[15] Daniel Schröter u. a. *Das Sicherheitsloch Buffer-Overflows und wie man sich davor schützt*. 2001. URL: http://www.heise.de/ct/artikel/Das-Sicherheitsloch-285320.html (besucht am 10. 05. 2016).

[16] PCTools. *Arbitrary Code Execution*. 2016. URL: http://www.pctools.com/security-news/arbitrary-code-execution/ (besucht am 02. 08. 2016).

[17] Inc. FIRST.org. *Common Vulnerability Scoring System*. 2016. URL: http://www.first.org/cvss (besucht am 09. 05. 2016).

[18] Mitre. *About CVE*. 2016. URL: http://cve.mitre.org/about/ (besucht am 09. 05. 2016).

[19] Axel Pols Andreas Streim Timm Lutter. *Umsätze mit Smartphones gehen in Deutschland erstmals zurück*. Hrsg. von Bitkom Research GmbH. 2016. URL: https://www.bitkom.org/Presse/Presseinformation/Umsaetze-mit-Smartphones-gehen-in-Deutschland-erstmals-zurueck.html (besucht am 27. 07. 2016).

[20] Bitkom Research GmbH. *Industrie im Visier von Cyberkriminellen und Nachrichtendiensten*. Hrsg. von Cornelius Kopke Maurice Shahd. 2016. URL: https://www.bitkom.org/Presse/Presseinformation/Industrie-im-Visier-von-Cyberkriminellen-und-Nachrichtendiensten.html (besucht am 27. 07. 2016).

[21] Mixpanel. *HOW QUICKLY ARE USERS UPDATING TO iOS 10?* 2016. URL: https://mixpanel.com/trends/#report/ios_10/ (besucht am 24. 10. 2016).

[22] Caitlin O’Connell. *One Month Later, iPhone ‘7’ Models Continue to Impress*. Hrsg. von Localytics. 2016. URL: http://info.localytics.com/blog/one-month-later-iphone-7-models-continue-to-impress (besucht am 25. 10. 2016).

[23] Skycure. *Mobile Threat Intelligence Report*. threat report. Skycure, 2016. URL: https://www.skycure.com/wp-content/uploads/2016/06/Skycure-Q1-2016-MobileThreatIntelligenceReport.pdf (besucht am 21. 11. 2016).

[24] Symantec. *Schutz vor Softwarepiraterie*. 10. Nov. 2016. URL: https://www.symantec.com/de/de/about/profile/antipiracy/ (besucht am 10. 11. 2016).

[25] Wikipedia. *Walled Garden*. 2016. URL: https://de.wikipedia.org/wiki/Walled_Garden (besucht am 25. 10. 2016).

[26] Wikipedia. *Jailbreak (iOS)*. 25. Okt. 2016. URL: https://de.wikipedia.org/wiki/Jailbreak_(iOS)#Geschichte (besucht am 25. 10. 2016).

[27] Stephan Wiesend. *Antivirus fürs iPhone: Macht es Sinn?* Hrsg. von Macwelt. 2016. URL: http://www.macwelt.de/ratgeber/Antivirus-iPhone-IOS-VPN-9944311.html (besucht am 26. 10. 2016).

[28] Ivan Kristic. *Behind the Scences with iOS Security*. Hrsg. von Balck Hat. 2016. URL: https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf (besucht am 25. 09. 2016).

[29] IDC's Worldwide Quarterly Mobile Phone Tracker. *Press Release*. 2016. URL: http://www.idc.com/getdoc.jsp?containerId=prUS40980416 (besucht am 22. 11. 2016).

[30] IDC's Worldwide Quarterly Mobile Phone Tracker. *Smartphone OS Market Share, 2016 Q2*. 2016. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (besucht am 10. 11. 2016).

[31] IDC's Worldwide Quarterly Mobile Phone Tracker. *Smartphone Vendor Market Share, 2016 Q2*. 2016. URL: http://www.idc.com/prodserv/smartphone-market-share.jsp (besucht am 11. 11. 2016).

[32] Kaspersky. *Verbreitung von Malware*. 2016. URL: http://www.kaspersky.com/de/internet-security-center/bedrohungen/malware-verbreitung (besucht am 10. 11. 2016).

[33] TechTarget. *Comparing mobile OS security features*. Hrsg. von Lisa Phifer. 2016. URL: http://searchmobilecomputing.techtarget.com/tip/Comparing-mobile-OS-security-features.

[34] Margret Rouse. *Remote Wipe*. 2014. URL: http://www.searchnetworking.de/definition/Remote-Wipe (besucht am 13. 11. 2016).

[35] Fabrizio Ferri-Benedetti. *Android, iOS und Windows Phone im Vergleich: Welches ist das sicherste Betriebssystem?* 2014. URL: https://artikel.de.softonic.com/android-ios-und-windows-phone-im-vergleich-welches-ist-das-sicherste-betriebssystem (besucht am 11. 11. 2016).

[36] Lisa Phifer Robert Sheldon Colin Steele. "Mobile Application Delivery: The Next Frontier". In: (2015). URL: http://searchmobilecomputing.techtarget.com/ehandbook/Mobile-application-delivery-The-next-frontier (besucht am 13. 11. 2016).

[37] Eric Klein. *Mobile network security remains an enterprise challenge*. 2016. URL: http://searchmobilecomputing.techtarget.com/opinion/Mobile-network-security-remains-an-enterprise-challenge (besucht am 11. 11. 2016).

[38] Symantec. *Internet Security Threat Report*. Threat Report. Symantec, 2016, S. 81. DOI: 10.1016/s1353-4858(00)03015-4. URL: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf (besucht am 22. 11. 2016).

[39] MICHAEL CATANZARO. *ON WEBKIT SECURITY UPDATES*. 2016. URL: https://blogs.gnome.org/mcatanzaro/2016/02/01/on-webkit-security-updates/ (besucht am 14. 10. 2016).

[40] Apple. *Informationen zum Sicherheitsinhalt von iOS 10*. 3. Nov. 2016. URL: https://support.apple.com/de-de/HT207143 (besucht am 13. 11. 2016).

[41] Hui Xue, Tao Wei und Yulong Zhang. *MASQUE ATTACK: ALL YOUR IOS APPS BELONG TO US*. Hrsg. von FireEye. 2014. URL: https://www.fireeye.com/blog/threat-research/2014/11/masque-attack-all-your-ios-apps-belong-to-us.html (besucht am 13. 11. 2016).

[42] Luyi Xing u. a. "Cracking App Isolation on Apple". In: *Proceedings of the 22nd Conference on Computer and Communications Security*. Association for Computing Machinery, 2015. DOI: 10.1145/2810103.2813609. URL: http://dx.doi.org/10.1145/2810103.2813609.

[43] Bill Marczak und John Scott-Railton. *The Million Dollar Dissident*. Hrsg. von Citizen Lab. 2016. URL: https://citizenlab.org/2016/08/million-dollar-dissident-iphone-zero-day-nso-group-uae/ (besucht am 13. 11. 2016).

[44] Lookout. *Sophisticated, persistent mobile attack against high-value targets on iOS*. 2016. URL: https://blog.lookout.com/blog/2016/08/25/trident-pegasus/ (besucht am 13. 11. 2016).

[45] Censys Team. *The FREAK Attack*. 2015. URL: https://censys.io/blog/freak (besucht am 14. 11. 2016).

[46] ZeroDay. *Adobe Flash AS2 Sound loadSound Use-After-Free Remote Code Execution Vulnerability*. 2015. URL: http://www.zerodayinitiative.com/advisories/ZDI-15-563/ (besucht am 14. 11. 2016).

[47] Steve Jobs. *Thoughts on Flash*. Hrsg. von Apple. 2010. URL: http://www.apple.com/hotnews/thoughts-on-flash/ (besucht am 26. 10. 2016).

[48] Wikipedia. *SQLite*. 2016. URL: https://de.wikipedia.org/wiki/SQLite (besucht am 17. 11. 2016).

[49] Katerina Roukounaki. *Five popular databases for mobile*. 2014. URL: https://www.developereconomics.com/five-popular-databases-for-mobile (besucht am 17. 11. 2016).

[50] Apple. *What is Core Data?* 2016. URL: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/index.html#//apple_ref/doc/uid/TP40001075-CH2-SW1 (besucht am 17. 11. 2016).

[51] Ophir Prusak. *iOS Databases: SQLLite vs. Core Data vs. Realm*. 2016. URL: https://blog.rollout.io/ios-databases-sqllite-core-data-realm/ (besucht am 17. 11. 2016).

[52] Wikipedia. *JavaScript Object Notation*. 2016. URL: https://de.wikipedia.org/wiki/JavaScript_Object_Notation (besucht am 17. 11. 2016).

[53] Yari D'areglia. *CREATING CUSTOM WALKTHROUGHS FOR YOUR APPS*. 2016. URL: http://www.thinkandbuild.it/creating-custom-walkthroughs-for-your-apps/ (besucht am 27. 10. 2016).

[54] Apple. *About the iOS Technologies*. 2016. URL: https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html (besucht am 30. 10. 2016).

[55] Clemens Wagner Klaus M. Rodewig. *Apps entwicklen für iPhone und iPad*. 2013. 1172 S. ISBN: 978-3-8362-2734-6. URL: http://openbook.rheinwerk-verlag.de/apps_entwickeln_fuer_iphone_und_ipad/apps_01_001.html#dodtp09544318-027e-49d8-b540-d500a5a0466a (besucht am 13. 11. 2016).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Glossar

Apple A7 Ist ein System-on-Chip (SoC) der auf einer 64-Bit ARM Architektur basiert.. 9

Boot-ROM Der erste Code, den der Prozessor des Geräts am Beginn des Startvorgangs ausführt. Als integraler Bestandteil des Prozessors kann er weder von Apple noch von einem Angreifer modifiziert werden. [1, S. 61]. 7

Curve25519 Elliptische Kurve die auch bei Schlüsselaustauschprotokollen zum Einsatz kommt.. 14

D-U-N-S Data Universal Numbering System - Zahlensystem zur eindeutigen Identifizierung von Firmen. 11

DFU-Modus (Device Firmware Upgrade) Modus, bei dem der Boot-ROM des Geräts darauf wartet, über USB wiederhergestellt zu werden. Der Bildschirm bleibt im DFU-Modus schwarz, bis eine Verbindung zu einem Computer mit iTunes hergestellt wird. Daraufhin wird die folgende Eingabeaufforderung angezeigt: „iTunes hat ein iPad im Wartungsmodus erkannt. Sie müssen dieses iPad wiederherstellen, bevor es mit iTunes verwendet werden kann.“[1, S. 61]. 7

DMA Direct Memory Access, damit wird ein direkter Speicher Zugriff über ein Bussystem beschrieben.. 13

ECDH Elliptic curve Diffie-Hellman, ist ein Elliptische Kurven Schlüsselaustauschprotokoll basierend auf dem Diffie-Hellmann Protokoll.. 14

ECID Exklusiv Chip ID , eine 64-Bit Kennung welche einzigartig für jedes Gerät ist.. 7

Effaceable Storage (Auslöschbarer Speicher) Ein bestimmter Bereich des NAND-Speichers, in dem kryptografische Schlüssel gespeichert werden und der direkt abgerufen und sicher gelöscht werden kann. Es ist zwar kein Schutz geboten, wenn ein Angreifer direkt auf das Gerät zugreifen kann, die Schlüssel im Effaceable Storage können aber als Teil einer Schlüsselhierarchie verwendet werden und so eine schnelle Löschung und Folgenlosigkeit (Forward Secrecy) ermöglichen. [1, S. 61]. 13

GID Group ID, ein 256-Bit AES Schlüssel der bei allen Prozessoren einer Geräteklasse gleich ist.. 9

iBoot Code der im Rahmen des sicheren Startvorgangs von LLB geladen wird und selbst XNU lädt.[1, S. 61]. 7

JTAG Standardwerkzeug für Hardwaredebugging, das von Programmierern und Schaltkreisentwicklern verwendet wird.. 9

Low-Level-Bootloader (LBB) Code der im Rahmen des sicheren Startvorgangs vom Boot-ROM abgerufen und selbst iBoot lädt.[1, S. 62]. 7

Metrics Attack Vector (AV), Attack Complexity (AC), Privileges Required (PR), Integrity (I), Availability (A) - hier genauer beschrieben https://www.first.org/cvss/calculator/3.0. 19

Repos Repositories, Standardquellen von Cydia welcher einer Zensur ähnlich wie der bei dem AppStore durch die Quellenbetreiber unterliegen.. 25

UID Unique ID, ein 256-Bit AES Schlüssel der bei jedem Gerät einzigartig ist.. 9, 13

VPN Virtual Private Network, beschreibt ein virtuelle private Netzwerkinfrastruktur, bei welchen Komponenten des privaten Netzwerk über ein Öffentliches Netzwerk Kommunizieren, mit der Einbildung das Netz stehe ihnen allein zur Verfügung. [7, S.765]. 67

XNU Der Kernel im Zentrum der Betriebssysteme iOS und OS X. Er wird als vertrauenswürdig eingestuft und setzt Sicherheitsmaßnahmen wie Codesignierung, Sandbox, Überprüfen von Berechtigungen und ASLR durch.[1, S. 62]. 7

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29. November 2016 Maschood Ahmad