MASTERARBEIT

SICHERE APP ENTWICKLUNG

Secure App Development

ausgeführt am

Studiengang

Informationstechnologien und Wirtschaftsinformatik

Von: Sandra Gschwend

Personenkennzeichen: 1510320031

Graz, am 27. März 2017 .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....

Unterschrift

DANKSAGUNG

An dieser Stelle möchte ich mich besonders bei meinem Betreuer DI (FH) Christian Schmid, M.Sc. bedanken, der mich während meiner Diplomarbeit betreut und unterstützt hat.

Außerdem möchte ich mich herzlich bei meinen Eltern für die Unterstützung bedanken. Mein ganz besonderer Dank geht aber an Dominik, der mir bei vielen Formulierungen und auch bei der Korrektur der Arbeit sehr hilfreich zur Seite stand.

KURZFASSUNG

Sicherheit in mobilen Applikationen ist ein wichtiger Punkt in der mobilen Softwareentwicklung. Diese Arbeit hat sich das Ziel gesetzt, bekannte Sicherheitsrisiken aufzuzeigen und welche Angriffsvektoren sich im speziellen für die mobilen Apps und die dazugehörigen Applikationsserver ergeben, darzustellen. Der Fokus liegt auf das Erarbeiten sicherer Frameworks und Designregeln, die zur Wahrung der IT-Sicherheit eingehalten werden sollen.

Der erste Teil bietet einen Einblick in die sichere Applikationsentwicklung. Es wird gezeigt, dass sich Sicherheitsprobleme über die Jahre verändert haben und das neue Angriffsvektoren hinzugekommen sind. Zusätzlich werden aktuelle Sicherheitsrisiken und die Sicherheitsmodelle der verwendeten Plattformen behandelt. Es wird gezeigt, wie Applikationen gegen Angriffe geschützt werden können.

Der zweite Teil widmet sich unterschiedlichen Übertragungsmöglichkeiten, Angriffs-möglichkeiten und Verteidigungsmöglichkeiten. Es geht hervor, wie eine mobile Applikation attackiert werden kann und wie Abwehrversuche aussehen können. Diese Themen werden mithilfe von Beispielen verständlich dargestellt.

Im praktischen Kapitel wird anhand von Tests erörtert, wie wichtig grundlegende Sicherungen von mobilen Anwendungen sind. Sie weisen auch darauf hin, dass nicht jede Verteidigung einen vollständigen Schutz bietet, aber Angriffe verlangsamen können. In Verbindung mit der theoretischen Ausarbeitung und den Erkenntnissen der Tests entstanden Frameworks und dazugehörige Designregeln. Diese bieten Anfängerinnen und Anfänger eine Grundlage für die sichere Applikationsentwicklung und sie dienen als Checkliste für erfahrene Entwicklerinnen und Entwickler.

ABSTRACT

Security in mobile applications is a crucial issue in mobile software development. The purpose of this thesis is to present known security risks and show vulnerabilities in mobile applications and the corresponding application servers. The focus is on the development of secure frameworks and design rules which can ensure IT security.

The thesis begins with an insight into secure application development. This part shows that security problems have changed over the years and new attack methods discovered. It also presents current security risks and the security models of the platforms and how to protect the applications against attacks.

The second part addresses various data transmission possibilities, attack possibilities, and defence possibilities. It shows how to attack a mobile application and possible defensive measures.

The practical chapter is based on practical tests and discusses the importance of security for mobile applications. The tests show that defensive measures are imperfect, but can slow down attacks. In conjunction with the theoretical elaboration and the findings of the tests, frameworks and related design rules are developed. They offer a basis for secure application development as a beginner and serve as a checklist for experienced developers.

INHALTSVERZEICHNIS

1 EINFÜHRUNG

Dieses Kapitel widmet sich der Problemstellung und die daraus abgeleitete Forschungsfrage mit Hypothesen. Des Weiteren werden die Ziele und Nicht-Ziele behandelt und die verwendete Methoden und Vorgehensweisen erläutert. Zudem wird der Aufbau der Masterarbeit grafisch dargestellt.

1.1 Problemstellung

Der mobile Markt ist ein stark wachsender Bereich. Unzählige Anwendungen werden in App Stores veröffentlicht. Diese Applikationen werden von Nutzerinnen und Nutzern heruntergeladen und verwendet. Zum Teil werden die Anwendungen von großen Unternehmen entwickelt, es befinden sich dort aber auch Apps von Freelancern oder Programmieranfängerinnen und Programmieranfängern.

Jeder App Store hat dabei seine eigenen Regeln, die für die Veröffentlichungen eines Programmes eingehalten werden müssen. Diese Regeln beinhalten aber hauptsächlich Punkte in Bezug auf Layout, Schriftgrößen und Umgang mit persönlichen Daten. Auf Sicherheitsrisiken wird nicht ausreichend eingegangen. Deshalb ist es wichtig, noch zusätzliche Sicherheitskriterien zu beachten. Wird bei der Entwicklung die Sicherheit vernachlässigt, können Lücken entstehen, die ein Risiko für Anwenderinnen und Anwender darstellen.

1.2 Forschungsfrage

Welche Designregeln und Security Frameworks können für die Wahrung der IT Sicherheit bei mobilen Applikationen angewendet werden?

1.3 Hypothese

Forschungshypothese: Das Einsetzen von Designregeln und Security Frameworks beeinflusst die Sicherheit von mobilen Anwendungen.

H1: Designregeln und Security Frameworks beeinflussen die Sicherheit von mobilen Anwendungen.

H0: Designregeln und Security Frameworks haben keinen Einfluss auf die Sicherheit von mobilen Anwendungen.

1.4 Ziele und Nicht-Ziele

Das Ziel dieser Masterarbeit ist, bekannte Sicherheitsrisiken aufzuzeigen und welche Angriffsvektoren sich im speziellen für die mobilen Apps und die dazugehörigen Applikationsserver ergeben. Des Weiteren soll gezeigt werden, wie ein sicheres Framework aussehen kann und welche Designregeln zur Wahrung der IT-Sicherheit eingehalten werden sollen.

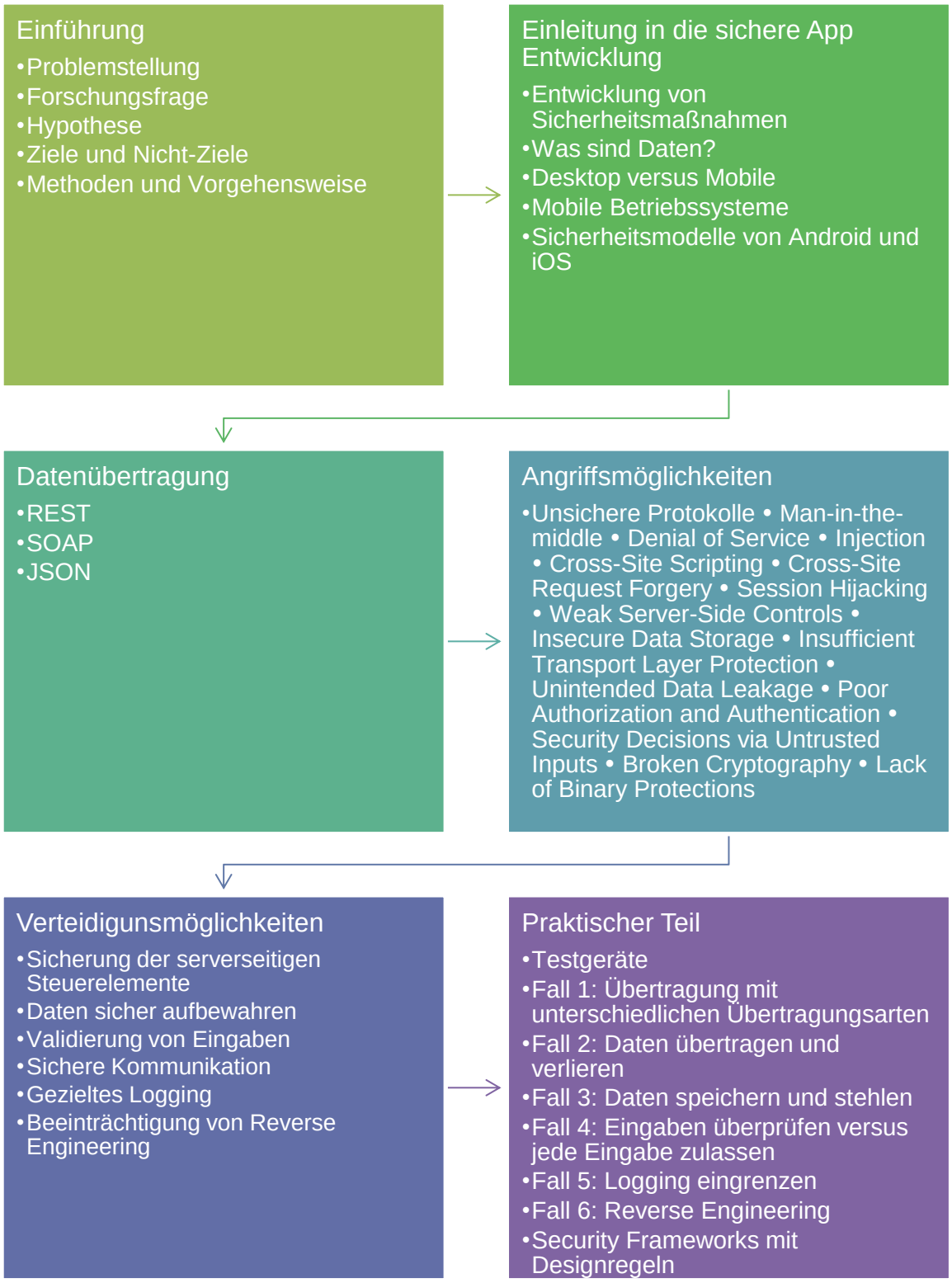
Der Sinn dieser Ausarbeitung ist nicht eine komplette Sammlung von allen existierenden Angriffsmöglichkeiten aufzuzeigen, sondern einen Überblick über relevante Attacken geben.

1.5 Methoden und Vorgehensweise

Zur Bearbeitung des oben genannten Forschungsthemas wird eine Testimplementierung erstellt. Um eine Implementierung erstellen zu können, wird eine theoretische Ausarbeitung benötigt. Anhand der theoretischen Ausarbeitung und der Testimplementierung werden sichere Frameworks mit Designregeln angefertigt und definiert.

Die Testumsetzung soll zeigen, wie das Sicherheitsrisiko bekannter Gefahren gemindert werden kann und wie eine sichere Datenübertragung möglich ist. Des Weiteren soll sie Aufschluss darüber geben, ob ein sicheres Framework in Verbindung mit Designregeln, die Sicherheit von mobilen Anwendungen und deren Applikationsserver beeinflusst.

1.6 Gliederung der Arbeit



2 EINLEITUNG IN DIE SICHERE APP ENTWICKLUNG

Die Beliebtheit von mobilen Endgeräten führt dazu, dass die Programmierung von mobilen Anwendungen genauso entscheidend ist, wie es bei Desktop-Anwendungen der Fall war. Vor allem soziale Medien gehen Hand in Hand mit mobilen Geräten. Es wurde immer mehr zu einem Rennen um bessere Anwendungen für kleinere Bildschirme. Das bedeutet konkret, Daten von unterschiedlichen Orten im Internet zu sammeln, sie lesbar aufzubereiten und wieder an unterschiedliche Orte im Internet zu senden. Nur, was sind Daten? Woher kommen diese? Was passiert mit ihnen? Das ist ein Sicherheitsproblem. (Glaser, 2014)

Mobile Anwendungen zu erstellen, bedeutet in erster Hinsicht, einen Service zu erstellen, der mit einem Webserver kommunizieren kann. Das wird benötigt um wichtige Prozessschritte abzuhandeln und gibt die Möglichkeit, dass der Client das Layout rendern kann. Von dem neu organisierten Stream der an den Client gesendet wird, entsteht dann ein Datenchaos. Ein Chaos, das zu verstehen und zu ordnen ist. Diese Aufgabe erhält die Entwicklerin oder der Entwickler. Sie verwenden alle Werkzeuge um es zu beseitigen, oder zu bändigen. Es ist eine große Aufgabe. Sicherheit bedeutet, das Richtige zur richtigen Zeit zu tun, und das konsequent. (Glaser, 2014)

2.1 Entwicklung von Sicherheitsmaßnahmen

In den vergangenen Jahren sind Sicherheitsprobleme stark gewachsen. In den Anfängen von C und C++, als der meiste Code noch binär war, war der Hauptangriffsvektor der Buffer Overflow. Die Entwicklerinnen und Entwickler versuchten sicherzustellen, dass alle Daten des Buffers auch innerhalb des Buffers einen Platz finden. Es war ein einfaches Problem, mit einer einfachen Lösung. Vor allem da ein Buffer Overflow immer ein Eingabeproblem ist und im Gegensatz zu heute, war es einfacher sich auf diese Problematik zu fokussieren. (Glaser, 2014)

Heutzutage beinhalten Anwendungen Web-Technologien, die zuerst interpretiert werden müssen. Das verändert den Angriffsvektor, indem versucht wird, diese Interpretation zu umgehen. Das bedeutet, dass es nicht nur Angriffe auf Eingabefelder gibt, wie zum Beispiel mittels Injection (wird im Kapitel 4.4 näher erläutert), sondern es werden auch die Ausgaben angegriffen. Wie zum Beispiel was und wie etwas dargestellt wird oder ob es sich um aktiven oder nicht aktiven Inhalt handelt. Neuen Code zu entwickeln benötigt Zeit. Zu Zeiten des Buffer Overflow war mehr Zeit vorhanden, um die Problematik zu verstehen und Lösungen zu implementieren. Gegenwärtig ist es anders. Zusätzlich zeigen viele Fragen die in Foren gestellt werden, dass Entwicklerinnen und Entwickler nicht wissen, wie eine Filterung für die Eingabe konkret aussehen soll. Es werden zum Beispiel Fragen gestellt wie: „Was ist der beste Weg um einen Filter zu erstellen?“ oder „Ist mein Filter gut genug?“. Es gibt zwar eine große Anzahl an Antworten, aber es kann passieren, dass sie sich widersprechen, es eine persönliche Meinung ist oder dass es kein Versuch ist die Frage zu beantworten, sondern es eher zur Beleidigung des Fragestellers kommt. Nicht hilfreiche Antworten sind ein Beleg, dass viele Sicherheitsprobleme durch das Fehlen von Verständnis entstehen. (Glaser, 2014)

2.2 Was sind Daten?

Um zu verstehen, was geschützt werden soll, ist es wichtig zu wissen, was Daten sind. Wenn in der Informatik von Daten gesprochen wird, werden meist alle Informationen die im System dargestellt werden können, damit bezeichnet. In einer sicheren Anwendungsentwicklung sollte besonders stark auf personenbezogene und sensible Daten Rücksicht genommen werden. (Gunasekera, 2012)

Personenbezogene Daten sind private Daten, die mit Freunden oder Familienmitgliedern geteilt werden. Zum Beispiel sind das Mobiltelefonnummern, Adressen oder E-Mail-Adressen. Werden diese Informationen veröffentlicht, kommt es selten zu einem physischen Schaden oder psychischen Leid. In Folge dessen könnte ein Stalker Zugriff auf Telefonnummern und Adressen erhalten und der betroffenen Person Schaden zufügen. (Gunasekera, 2012)

Sensible Daten sind mehr Wert, als personenbezogene Daten. Sensible Daten werden unter normalen Umständen mit niemanden geteilt. Beispiele sind Passwörter, Bankanmeldedaten, PIN-Codes, Mobiltelefongeheimnummern oder die Sozialversicherungsnummer. Werden sensible Daten veröffentlich, können die Auswirkungen zu einen physischen Schaden oder ein psychisches Leid führen. Diese Daten sind deshalb besonders zu schützen. Zum Beispiel der Verlust der Bankdaten führt zum physischen Schaden, da Geld gestohlen wurde und zum psychischen Leid, aufgrund des Diebstahls. (Gunasekera, 2012)

2.3 Desktop versus Mobile

Daten können sich überall befinden, wie zum Beispiel auf einem Server, Desktop, Notebook oder Smartphone. Vor allem das Smartphone wurde zu einem persönlichen Gegenstand des täglichen Lebens. Der alltägliche Einsatz reicht weit über Fotografieren, Nachrichten senden oder News lesen. Die Fragen die sich stellen sind: Warum auf den mobilen Markt konzentrieren? Wieso sollten auch Daten von mobilen Endgeräten geschützt und gesichert werden?

Die Abbildung 2-1 von KPCB mobile technology trends von Mary Meeker zeigt den Trend zum mobilen Gerät. 2008 wurden digitale Medien noch zu 80 Prozent mittels Desktop erkundet, 2015 sind es nur mehr 42 Prozent. Der Desktop wurde von dem mobilen Gerät eingeholt, das 2015 zu 51 Prozent am Tag verwendet wird, um digitale Medien zu konsumieren. (Chaffey, 2016)

Mobile Endgeräte verdrängen den Desktop. Somit werden viele Tätigkeiten, die früher mit dem Computer getätigt wurden, auch am Smartphone erledigt. Wie zum Beispiel Einkäufe. Diese können von überall getätigt werden. (Chaffey, 2016) Dabei können Zahlungsinformationen und personenbezogene Daten verwendet werden. Diese Daten und alle anderen sensiblen Informationen sind zu schützen.

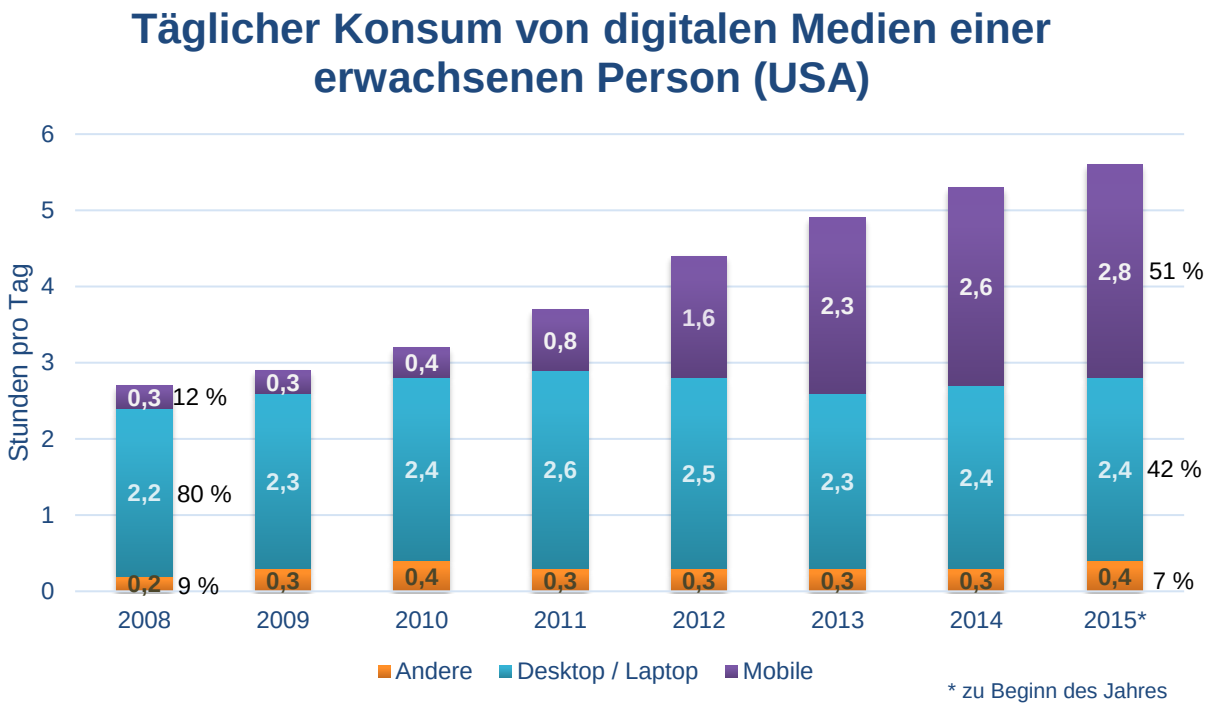


Abbildung 2-1: Täglicher Konsum von digitalen Medien einer erwachsenen Person (Chaffey, 2016)

Yahoo's Flurry analytics zeigt in der Abbildung 2-2, dass 90 Prozent der Nutzerinnen und Nutzer mobile Anwendungen dem mobilen Web Browser vorziehen. Aus diesem Grund sollten sich Unternehmen überlegen, ob sie nur eine Webapplikation erstellen oder es von Vorteil wäre, eine mobile App zu entwickeln. Diese Abbildung ist mit Vorsicht zu genießen, da viel Zeit auf Facebook, Nachrichtensystemen oder mit Spielen verbracht wird. Die Anwenderin oder der Anwender benötigen einen Mehrwert, um eine mobile Anwendung zu installieren. (Chaffey, 2016)

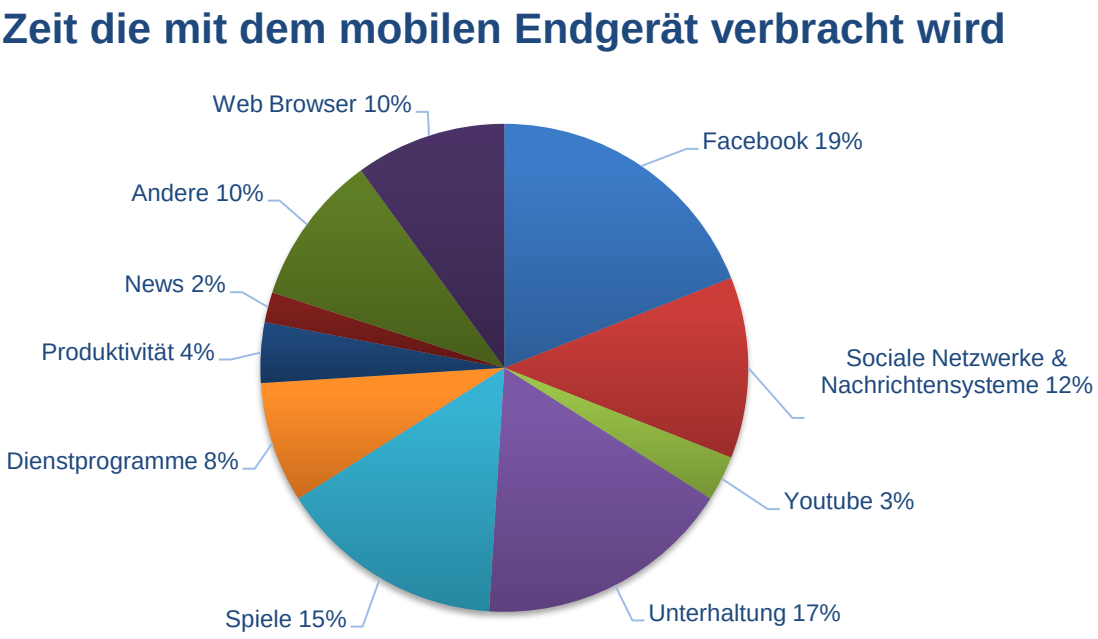


Abbildung 2-2: Zeit die mit dem mobilen Gerät verbracht wird (Chaffey, 2016)

2.4 Mobile Betriebssysteme

Das vorherige Unterkapitel zeigt, wie sich das Nutzerinnen- und Nutzerverhalten in den letzten Jahren verändert hat. Der Trend geht zum mobilen Gerät. Mehr und mehr Tätigkeiten, wie Einkäufe, Kommunikation usw., werden auf das Smartphone ausgelagert. Umso wichtiger ist es, sich um die Sicherheit einer mobilen Anwendung zu kümmern.

Nicht alle mobilen Geräte sind gleich. Zum einen unterscheiden sie sich durch ihre technischen Spezifikationen, zum anderen durch ihre Betriebssysteme (OS = Operating System). Welche mobilen OS gerade aktuell sind, wird in diesem Kapitel erläutert.

In der Abbildung 2-3 sind die Prozentwerte der OS-Verbreitung bei Smartphone-Neukäufen, die von Kantar World Panel herausgegeben wurden, zu sehen. Diese beziehen sich auf Europa, im Zeitraum von Dezember 2015 bis Februar 2016. Android von Google hat in dieser Zeit einen neuen Jahreshöchstwert von 74,3 Prozent aufgestellt. Auf Platz zwei befindet sich iOS von Apple. Es liegt mit 19,3 Prozent weit hinter Android. Mit 5,9 Prozent befindet sich Microsoft mit seinem Windows auf Platz drei. Vor allem Microsoft hat mit der Verbreitung seines aktuellen OS Windows 10 Mobile Schwierigkeiten. Es verzeichnet fast auf allen Märkten Verluste und hat mit einem Marktanteil von 0,9 Prozent in China oder 0,5 Prozent in Japan kaum noch Bedeutung. (Schäfer, 2016) Aufgrund des hohen Marktanteiles werden in dieser Arbeit Android und iOS betrachtet.

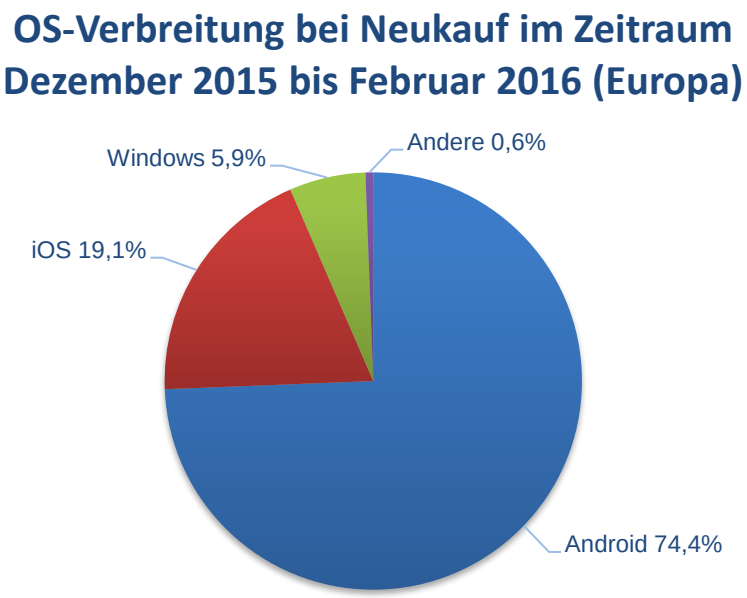


Abbildung 2-3: OS-Verbreitung bei Neukauf im Zeitraum Dezember 2015 bis Februar 2016 (Schäfer, 2016)

2.5 Sicherheitsmodelle von Android und iOS

Vor dem Beginn der Entwicklung einer mobilen Anwendung, sollten zuerst die Sicherheitseigenschaften des gewünschten OS ermittelt werden. Im Folgenden werden die Sicherheitsmodelle von Android und iOS näher erläutert. Diese Arbeit baut auf ein grundlegendes Wissen der Anwendungsentwicklung auf.

2.5.1 Android

Das Android Sicherheitsmodell baut darauf auf, dass unterschiedliche Anwendungen nicht die Möglichkeit haben, auf fremde Daten, ohne Autorisierung, zuzugreifen. Des Weiteren kann eine Anwendung, ohne entsprechende Berechtigungen, den Betrieb einer anderen Anwendung nicht nachteilig beeinflussen. Das bildet das Grundkonzept für die Anwendungs-Sandbox. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Code-Signierung

Klingt theoretisch einfach. Das Problem liegt bei der Definition, was eine autorisierte Handlung ist und was nicht. Anwendungen müssten wissen, ob eine andere Anwendung berechtigt ist, gewisse Aktionen durchzuführen. Zum Beispiel könnten Herstellerinnen und Hersteller behaupten, dass ihre Anwendungen von Google entwickelt wurden. Aus diesem Grund wird die Identität von jeder und jedem durch die Code-Signierung verwaltet. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Die Signierung eines Android-Pakets erfolgt kryptografisch, durch die Verwendung von digitalen Zertifikaten, deren privater Schlüssel von den Anwendungsentwicklerinnen und -entwicklern gehalten wird. Die Code-Signierung wird verwendet, um die Identität einer Entwicklerin oder eines Entwicklers einer Anwendung zu belegen. Die Signierung eines Pakets ist zwingend, auch wenn es sich um das Standard-Debug-Zertifikat handelt, das während der Entwicklung verwendet wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Berechtigungen

Ein weiterer wichtiger Sicherheitsmechanismus sind Berechtigungen. Hat jede Anwendung, die auf dem Gerät installiert ist Zugriff auf Kontakte, Nachrichtensysteme, GPS-Standort oder andere Informationen, könnte das fatale Folgen für die Privatsphäre haben. Aus diesem Grund verwendet Android ein Permission Model für Anwendungen. Anwendungen müssen eine Berechtigung anfordern, um auf bestimmte Informationen und Ressourcen auf einem Gerät zugreifen zu können. Eine Benutzerin oder ein Benutzer, die oder der eine Anwendung aus dem Play Store installiert, wird ein Hinweis angezeigt. Dieser Hinweis zeigt die Arten von Informationen und Hardware, auf die die Anwendung auf dem Gerät zugreifen kann. Diese Informationen werden jedoch in neueren Versionen des Play Store von den technischen Details abstrahiert und zeigen nicht die vollständigen Details der aktuellen Berechtigung an. Jede definierte Berechtigung hat einen eindeutigen Namen, der im Code referenziert wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Protection Levels

Des Weiteren besitzt jede Berechtigung, die definiert wird, ein zugehöriges Attribut, das als Protection Level bezeichnet wird. Protection Levels regeln, unter welchen Umständen Anwendungen gewisse Berechtigung anfordern können. Einige Berechtigungen sind gefährlicher als andere. Dies sollte sich in dem zugewiesenen Protection Level widerspiegeln. Beispielsweise sollten Anwendungen von Drittanbietern niemals die Möglichkeit erhalten, neue Anwendungen zu installieren. Entwicklerinnen oder Entwickler von einer Anzahl von Anwendungen, möchten Informationen zwischen den Anwendungen gemeinsam nutzen oder die Funktionalität zwischen ihren Anwendungen auf einer sicheren Weise aufrufen. Beide Szenarien können durch die Auswahl der richtigen Protection Levels erreicht werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Festplattenverschlüsselung

Eine weitere Sicherheitsfunktion ist die vollständige Festplattenverschlüsselung (FDE = full disk encryption). FDE verschlüsselt den Inhalt eines gesamten Laufwerks oder Datenträgers und nicht nur ausgewählte einzelne Dateien. Das ist sinnvoll, weil es beim Start das Kennwort vom Benutzer anfordert und von da an transparent verschlüsselt und entschlüsselt. Dabei werden alle Daten, die auf den Datenträger geschrieben und gelesen werden verschlüsselt. Dies dient zum Schutz vor gestohlenen oder verloren gegangenen Festplatten, die ausgeschaltet wurden. Dadurch sollen bekannte forensische Techniken, wie Disk Imaging und Booten der Festplatte mit einem anderen Betriebssystem (um den Inhalt zu durchsuchen) verhindert werden. Vor Android 5.0 (Lollipop) ist FDE nicht standardmäßig aktiviert und muss von der Benutzerin oder dem Benutzer aktiviert werden. Der PIN oder das Passwort für das Entsperren des Displays ist dasselbe, das zum Verschlüsseln des FDE-Passworts verwendet wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Keine Standard Root-Rechte

Ein weiterer Sicherheitsmechanismus unter Android ist, dass es keine standardmäßige Möglichkeit gibt, eine Anwendung oder eine Aufgabe als Root auszuführen. Das hat dazu geführt, dass Communities versuchen Wege zu finden, um Root-Zugriff auf verschiedenen Android Geräten zu erhalten. (Chell, Erasmus, Colley, & Whitehouse, 2015)

2.5.2 iOS

Folgende Sicherheitstechnologien werden von Apple für Hardware- oder Softwarekomponenten kombiniert, um die Gesamtsicherheit von iPhone-, iPad- und iPod-Geräten zu verbessern. Diese Sicherheitsfunktionen sind auf allen Geräten vorhanden, außer auf denen, die durch einen Jailbreak verändert wurden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Die grundlegenden Sicherheitsfunktionen sind:

- Secure boot chain
- Code signing
- Process-level sandboxing
- Data-at-rest encryption
- Generic native language exploit mitigations:
    1. Address space layout randomization
    2. Non-executable memory
    3. Stack-smashing protection

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Secure boot chain

Die Abbildung 2-4 zeigt die Secure Boot Chain. Sie ist die erste Verteidigungsschicht.

Abbildung 2-4: iOS Secure Boot Chain (Chell, Erasmus, Colley, & Whitehouse, 2015)

Dabei wird die Firmware beim Hochfahren initialisiert und auf das iOS Gerät geladen. In jedem weiteren Schritt der Secure Boot Chain, werden alle relevanten Komponenten, die von Apple kryptografisch signiert wurden, überprüft. Das dient zur Sicherstellung, dass nichts verändert wurde. Wenn ein iOS-Gerät eingeschaltet wird, führt der Prozessor den Boot-ROM aus, der ein schreibgeschützter Teil des Codes ist. Dieser wird während der Herstellung auf dem Chip gebrannt und auf Grund dessen vertraut. Der Boot-ROM enthält den öffentlichen Schlüssel für die Apple Root CA. Mit ihm wird die Integrität des nächsten Schrittes der Secure Boot Chain, des Low-Level-Bootloaders (LLB), überprüft. Der LLB führt eine Reihe von Setup-Routinen aus, wie zum Beispiel die Lokalisierung des iBoot Image im Flash-Speicher, von dem gebootet wird. Der LLB versucht, die in Abbildung 2-4 gezeigte Secure Boot Chain einzuhalten, indem die Signatur des iBoot Images überprüft wird. Stimmt sie nicht mit dem zu erwarteten Wert überein, wird im recovery mode gebootet. iBoot ist der second-stage Bootloader. Er ist für die Verifizierung und das Laden des iOS Kernels verantwortlich. Der iOS Kernel ladet wiederum die Usermode-Umgebung und das OS. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Eine weitere Sicherung ist der Secure Enclave. Er ist ein Coprozessor, der mit den A7 und A8 Chip Geräten ausgeliefert wird. Er verwendet seinen eigenen secure Boot und seine eigenen unabhängigen Softwareupdate Prozesse. Der Secure Enclave erledigt die kryptografischen Operationen auf dem Gerät. Diese sind die Schlüsselverwaltung für die Data Protection API und die Touch ID-Fingerabdruckdaten. Der Secure Enclave verwendet eine benutzerdefinierte Version des ARM TrustZone, um sich vom Hauptprozessor zu trennen und die Datenintegrität zu gewährleisten. Auch dann, wenn das Kernel des Gerätes gefährdet wird. Das bedeutet, wenn das Gerät mittels Jailbreak verändert wurde oder anderweitig gefährdet wird, können die kryptografischen Materialien, wie biometrischen Fingerabdruckdaten, von dem Gerät unmöglich extrahiert werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Code-Signierung

Wie Google bei Android, verwendet Apple die Code-Signierung bei iOS. Sie ist eines der wichtigsten Sicherheitsmerkmale der iOS Plattform. Sie versucht nicht autorisierte Anwendungen, auf einem Gerät beim Starten zu verhindern, indem sie die Anwendungssignatur bei jeder Ausführung überprüft. Darüber hinaus stellt die Code-Signierung sicher, dass Anwendungen nur Code ausführen können, der eine gültige, vertrauenswürdige Signierung besitzt. Zum Beispiel lehnt der Kernel jede Anfrage ab, die von nicht signierten Quellen stammen. Damit eine Anwendung auf einem iOS Gerät ausgeführt werden kann, muss sie zuerst von einem vertrauenswürdigen Zertifikat signiert werden. Entwickler können vertrauenswürdige Zertifikate auf einem Gerät über ein Provisioning-Profil installieren, das von Apple signiert wurde. Das Provisioning-Profil enthält das eingebettete Entwicklerzertifikat und die Anzahl der Berechtigungen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Sandbox

Vergleichbar mit Android verwendet iOS eine Sandbox. Sie ist eine eigenständige Umgebung, in der alle Anwendungen von Drittanbietern laufen. Dabei werden Anwendungen nicht nur von anderen Anwendungen, sondern auch vom OS isoliert. Sandboxing stellt eine bedeutende Sicherheit für die Plattform dar und begrenzt den Schaden, den Malware verursachen kann. Vorausgesetzt, dass eine schädliche Anwendung den App Store Überprüfungsprozess überstanden hat. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Data-at-Rest Encryption

Standardmäßig werden alle Daten im iOS Dateisystem durch die blockbasierte Verschlüsselung mit dem Dateisystemschlüssel verschlüsselt. Dieser wird beim ersten Start erzeugt und im Block 1 des NAND Flash-Speichers gespeichert. Das Gerät verwendet diesen Schlüssel während des Startvorgangs, um die Partitionstabelle und die Systempartition zu entschlüsseln. Das Dateisystem wird nur im Ruhezustand verschlüsselt. Wenn das Gerät eingeschaltet wird, entsperrt der hardwarebasierte crypto Accelerator das Dateisystem. Neben der Hardwareverschlüsselung können einzelne Dateien und keychain Items mit der Data Protection API verschlüsselt werden, die einen vom Gerätepasswort abgeleiteten Schlüssel verwendet. Third-party Applikationen, die sensible Daten verschlüsseln müssen, sollten die Data Protection API verwenden. Allerdings sollte berücksichtigt werden, dass Prozesse die im Hintergrund

gestartet werden, nicht auf die nötigen Dateien zugreifen können, wenn das Gerät gesperrt ist. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Schutz gegen Angriffe mit Exploit Mitigation Features

Die iOS-Plattform nutzt eine Reihe moderner Technologien, um Angriffe auf das Gerät zu erschweren. Eine der wichtigsten Schutzvorrichtungen ist die Implementierung der Schreib- und Lesezugriffsrichtlinie. Sie besagt, dass Speicherseiten nicht gleichzeitig als schreibbar (writeable) und ausführbar (executable) markiert werden können. Als Teil dieser Richtlinie können ausführbare Speicherseiten, die als beschreibbar markiert sind, nicht später zu ausführbaren Dateien zurückgesetzt werden. In vielerlei Hinsicht ähnelt dies den in den Microsoft Windows-, Linux- und Mac OS X-Desktopsystemen implementierten Data Execution Protection (DEP). Obwohl non-executable Memory protections alleine unter der Verwendung von return-oriented programming (ROP)-basierten Nutzlasten leicht umgangen werden können, wird die Komplexität der Ausnutzung signifikant erhöht, wenn sie mit Address Space Layout Randomization (ASLR) und zwingender Code-Signierung kombiniert wird. Die ASLR versucht dort, wo Daten und Code in einem Prozess-Adressbereich abgebildet werden, zu randomisieren. Durch die Randomisierung von Codestellen wird die Ausnutzung von Speicherbeschädigungen komplexer. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Ein weiterer Schutzmechanismus, den iOS Anwendungen nutzen können, ist der "stack-smashing"-Schutz. Dieser ist compiler-basierend und bietet eine Abwehr gegen traditionelle Stack basierende Overflow Exploits, durch die Einführung von Stack Canaries. Stack Canaries sind pseudozufällige Doppelwort-Werte, die hinter lokalen Variablen eingefügt werden. Stack Canaries werden nach der Funktion erneut überprüft. Wenn ein Überlauf aufgetreten ist und der Stack Canarie vollständig beschädigt oder überschrieben wurde, wird die Anwendung zwangsweise beendet. Das dient, um ein unbeabsichtigtes Verhalten zu vermeiden, das durch die Speicherkorruption verursacht werden könnte. (Chell, Erasmus, Colley, & Whitehouse, 2015)

2.6 Zusammenfassung

Dieses Kapitel bietet einen Einblick in die sichere Applikationsentwicklung. Es wird gezeigt, dass sich Sicherheitsprobleme über die Jahre verändert haben und das neue Angriffsvektoren hinzugekommen sind. Des Weiteren haben mobile Anwendungen in den vergangenen Jahren den Desktop eingeholt, was zu neuen Sicherheitsrisiken geführt hat. Für Entwicklerinnen und Entwickler ist es aus diesem Grund wichtig, sich über aktuellen Sicherheitsrisiken, die Sicherheitsmodelle der verwendeten Plattformen und wie sie ihre Applikationen gegen Angriffe schützen können, zu informieren. Das nachfolgende Kapitel gibt Informationen über Übertragungsmöglichkeiten zwischen mobilen Endgeräten und Servern.

3 DATENÜBERTRAGUNG

Damit Daten auf jedem persönlichen Gerät verfügbar sind, müssen diese Daten in irgendeiner Form übertragen werden. Dabei stellen sich die Fragen, welche Übertragungsarten es gibt und wie sie eingesetzt werden können. Dieses Kapitel widmet sich den diversen Übertragungsmöglichkeiten.

3.1 Representational State Transfer

Representational State Transfer (REST) arbeitet mit Ressourcen, die nie direkt angezeigt werden, sondern als Repräsentationen dargestellt sind. Dabei kann eine Ressource mehrere Repräsentationen haben, die in einem definierten Format dargestellt werden. Des Weiteren muss eine Ressource eindeutig identifizierbar sein. REST bietet eine Anzahl von Methoden, die von allen Ressourcen unterstützt werden sollten. Bei der HTTP 1.1-Version sind GET, HEAD, PUT, POST, DELETE, OPTIONS, TRACE und CONNECT definiert. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

Durch ein genaueres Betrachten von REST, zeichnen sich folgende fünf Kernprinzipien ab:

1. Ressourcen mit eindeutiger Identifikation
2. Verknüpfungen/Hypermedia
3. Standardmethoden
4. Unterschiedliche Repräsentationen
5. Statuslose Kommunikation

(Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

3.1.1 Ressourcen mit eindeutiger Identifikation

Jede REST-konforme Anwendung hat als eindeutige Adresse den Uniform Resource Identifier (URI). Dieser ermöglicht es, alle Instanzen von Abstraktionen der Anwendungen zu identifizieren. Beispiele sind:

- http://example.com/customers/1234
- http://example.com/orders/2007/10/776654
- http://example.com/products/
- http://example.com/processes/salary-increase-234

  (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

URIs, die für Menschen lesbar sind, sind aus Sicht von REST irrelevant. Dabei sollte beachtet werden, ein einheitliches und global gültiges Namenschema zu verwenden. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

3.1.2 Verknüpfungen/Hypermedia

Hypermedia wird verwendet um Beziehungen zwischen den Ressourcen herzustellen. Dabei wird der Applikationsfluss, soweit es möglich ist, gesteuert. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015) Das Listing 3-1 zeigt ein Beispiel einer Beziehung:

```
{
    "amount": 23,
    "customer": {
          "href": "http://example.com/customers/1234"
    },
    "product": {
          "href": "http://example.com/products/4554"
    },
    "cancel": {
          "href": "./cancellations"
    }
}
```

Listing 3-1: REST – Beziehung zwischen Ressourcen (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

Eine Anwendung kann diesen Links folgen, um weitere Informationen zu erhalten. Der Vorteil vom Hypermedia-Ansatz ist, dass Verknüpfungen anwendungsübergreifend funktionieren. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

3.1.3 Standardmethoden

Standardmethoden wie zum Beispiel HTTP müssen korrekt implementiert werden, da Clients die Möglichkeit haben müssen, mit den Ressourcen zu kommunizieren. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

3.1.4 Unterschiedliche Repräsentationen

Um unterschiedliche Anforderungen zu bewältigen, müssen unterschiedliche Repräsentationen der Ressourcen zur Verfügung stehen. Eine Repräsentation kann in einem bestimmen Format angefordert werden, wie das Beispiel vom Listing 3-2 zeigt. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

Listing 3-2: REST – Format einer Repräsentation (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

Hier könnte im vCard-Format eine Adresse von Customers zurückgeliefert werden. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

3.1.5 Statuslose Kommunikation

Bei der statuslosen Kommunikation muss entweder vom Client der Status gehalten werden oder der Server wandelt ihn in einen Ressourcenstatus um. Wichtig ist es serverseitig, dass nur dann kommuniziert wird, wenn ein Request gerade bearbeitet wird. Dabei wird kein Bezug auf den Status des Servers genommen, da dieser den gleichen Zustand behalten kann. (Tilkov, Eigenbrodt, Schreier, & Wolf, 2015)

In Android werden REST-Services vollständig unterstützt. Dazu wird der Apache `HttpClient` verwendet, der für Android angepasst wurde. Dieser bietet volle Unterstützung für das HTTP-Protokoll. (MacLean, Komatineni, & Allen, 2015) Auch iOS bietet eine vollständige Unterstützung von REST an. HTTP-Implementierungen können im `CFNetwork`-Framework gefunden werden. (Turner, 2011)

3.2 Simple Object Access Protocol

Das Simple Object Access Protocol (SOAP) ist eine nachrichtenbasierte Kommunikationstechnologie, mit der Daten im XML-Format ausgetauscht werden können. Es kann verwendet werden, um Informationen zu teilen und Nachrichten zwischen Systemen zu übertragen, auch wenn diese auf verschiedenen Betriebssystemen und Architekturen laufen.

Seine primäre Verwendung ist in Web-Services. SOAP wird häufig in großen Unternehmensanwendungen eingesetzt, wo einzelne Aufgaben von verschiedenen Computern ausgeführt werden, um die Leistung zu verbessern. Es wird eingesetzt, wo eine Webanwendung als Front-End für eine bestehende Anwendung erstellt wurde. In dieser Situation kann die Kommunikation zwischen verschiedenen Komponenten unter Verwendung von SOAP implementiert werden, um Modularität und Interoperabilität sicherzustellen. (Stuttard & Pinto, 2011)

Es gibt viele SOAP-basierte Web-Dienste für Android im Internet, bis jetzt hat Google aber keine direkte Unterstützung für den Aufruf von SOAP-Web-Services bereitgestellt. Google zieht stattdessen REST-ähnliche Webdienste vor. Allerdings haben die Entwicklerinnen und Entwickler mehr Arbeit, um Daten zu senden und die empfangenen Daten zu parsen. (MacLean, Komatineni, & Allen, 2015)

Wie bei Android gibt es für iOS keine SOAP-basierende Web-Dienste direkt von Apple. Wird SOAP benötigt, muss auf einer Drittanbieterin oder einem Drittanbieter zurückgegriffen werden. Dabei ist zu beachten, dass die Anbieterinnen oder Anbieter des Service die Entwicklung jederzeit einstellen könnten oder dass durch ein Update von XCode oder iOS der Service nicht mehr funktionieren könnte. (Turner, 2011)

3.3 JavaScript Object Notation

JavaScript Object Notation (JSON) ist ein einfaches Datenübertragungsformat. Mit JSON können beliebige Daten serialisiert werden und es kann direkt von JavaScript-Interpretern verarbeitet werden. Es kann in Ajax-Anwendungen als Alternative zu dem ursprünglich für die Datenübertragung verwendeten XML-Format verwendet werden. In einer Situation, wo eine Benutzerin oder ein Benutzer eine Aktion ausführt, verwendet das clientseitige JavaScript einen `XMLHttpRequest`, um die Aktion an den Server zu übermitteln. Der Server gibt eine Antwort mit Daten im JSON-Format zurück. Das clientseitige Skript verarbeitet dann diese Daten und aktualisiert die Benutzeroberfläche entsprechend. (Stuttard & Pinto, 2011)

Zum Beispiel kann eine Ajax-basierte Webmail-Anwendung ein Feature enthalten, um die Details eines ausgewählten Kontakts anzuzeigen. Wenn eine Benutzerin oder ein Benutzer auf einen Kontakt klickt, verwendet der Browser `XMLHttpRequest`, um die Details des ausgewählten Kontakts abzurufen, die mit JSON zurückgegeben werden. (Stuttard & Pinto, 2011) Das Listing 3-3 zeigt Kontaktdetails im JSON-Format

```
{
    "name": "Mike Kemp",
    "id": "8041148671",
    "email": "fkwitt@layerone.com"
}
```

Listing 3-3: JSON – Kontaktdetails (Stuttard & Pinto, 2011)

Das clientseitige Skript verwendet den JavaScript-Interpreter, um die JSON-Antwort zu verarbeiten, und aktualisiert den relevanten Teil der Benutzeroberfläche auf der Grundlage des Inhalts. Eine weitere Stelle, an der JSON-Daten in Anwendungen auftreten können, besteht darin, Daten in herkömmlichen Anforderungsparametern einzukapseln. (Stuttard & Pinto, 2011) Wenn der Benutzer beispielsweise die Details eines Kontakts aktualisiert, können die neuen Informationen dem Server, unter Verwendung des Codes vom Listing 3-4, mitgeteilt werden.

```
POST /contacts HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 89
Contact={"name":"Mike Kemp","id":"8041148671","email":"pikey@
clappymonkey.com"}
&submit=update
```

Listing 3-4: JSON – Aktualisierung von Kontaktinformationen (Stuttard & Pinto, 2011)

In Android wird JSON unterstützt. Es ist eine verbreitete Methode, um Daten zwischen einem Webserver und einem Client zu transferieren. Mithilfe der JSON-Parsing-Klassen können Daten aus einem Response verarbeitet werden. Google bietet zusätzlich die Gson-Java-Library an, um einen JSON-Input einfacher und schneller in ein Java-Object zu parsen und umgekehrt. Zusätzlich stellt Android einen XML-Parser bereit, um die Responses aus den HTTP-Calls zu verarbeiten. (MacLean, Komatineni, & Allen, 2015)

JSON wird ab iOS 5 mit der `NSJSONSerialization`-Klasse komplett unterstützt. Diese Klasse kann ein `NSDictionary` oder `NSArray` von JSON-Daten erstellen oder ein `NSDictionary` oder `NSArray` in JSON codieren. Die einfachste Methode, um Daten von einem Server zu einem iOS-Gerät zu erhalten, ist mittels einem `UIWebView` eine Webseite anzuzeigen. Es besteht aber die Möglichkeit, die Daten vom Server mit nativen Tools, wie `UITableView` am mobilen Gerät anzuzeigen. (Richter & Keeley, 2015)

3.4 Zusammenfassung

In diesem Kapitel wurden unterschiedliche Übertragungsarten aufgezeigt. Welche Übertragungsart gewählt wird, hängt von dem zweiten System ab. Wird auf einem Server SOAP verwendet, muss entweder ein Framework entwickelt oder auf ein Third-Party-Framework zurückgegriffen werden. Dabei sollte darauf geachtet werden, dass das Framework regelmäßig Updates erhält und ein Notfallplan sollte erstellt werden, falls die Entwicklung eingestellt wird. Besteht auf dem zweiten System noch keine Bindung zu einer Übertragungsart, bietet sich ein REST-Webdienst und JSON an, da sie von Android und iOS unterstützt werden. Um einen Überblick über bekannte Attacken zu erhalten, widmet sich das nachfolgende Kapitel den unterschiedlichen Angriffsmöglichkeiten.

4 ANGRIFFSMÖGLICHKEITEN

Das vorherige Kapitel hat gezeigt, dass Daten schnell an Personen überall auf der Welt übertragen werden können. Bei einer Übertragung kann es aber passieren, dass Informationen abgehört oder gestohlen werden. Aus diesem Grund wird bei personenbezogenen und sensiblen Daten versucht, dass sie so gut wie möglich diebstahlsicher übertragen werden. Dabei ist es zuerst wichtig, unterschiedliche Attacken zu kennen. Dieses Kapitel widmet sich den verschiedenen Angriffsmöglichkeiten.

4.1 Unsichere Protokolle

Daten im Klartext zu übertragen ist die unsicherste Variante, da bei der Übertragung Dritte die Möglichkeit haben, diese Daten auszulesen. Nicht verschlüsselte Protokolle sind zum Beispiel HTTP, FTP oder für den E-Mail-Versand IMAP, POP und SMTP. Für all diese Protokolle gibt es ein sicheres Gegenstück, wie zum Beispiel HTTPS für HTTP. Dabei handelt es sich um eine Verbindung von HTTP und SSL. Zwar bietet HTTPS auch keine vollkommene Sicherheit, es erschwert jedoch den Zugriff auf sensible Daten wie zum Beispiel Passwörter. (Ziegler, 2008)

4.2 Man-in-the-middle

Man-in-the-middle beschreibt eine Vielzahl von Techniken um die Kommunikation zwischen zwei Systemen im Netzwerk auf ein fremdes System umzuleiten. Die Abbildung 4-1 zeigt eine grafische Darstellung des Angriffes:

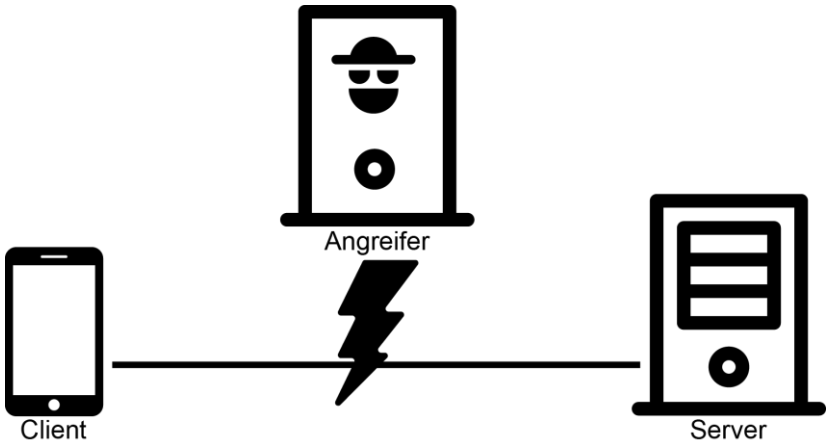


Abbildung 4-1: Man-in-the-middle

Eine Technik ist, Anwenderinnen und Anwender auf eine gefälschte Webseite zu locken. Dazu wird zuerst ein zusätzlicher Server benötigt. Dann müssen mögliche Opfer benachrichtigt werden, diesen Server zu verwenden. Dieser Angriff ist für Anwenderinnen und Anwender leicht ersichtlich und dadurch nicht effektiv. (Ziegler, 2008)

Ein Beispiel für einen Man-in-the-middle-Angriff bei mobilen Geräten ist Tapjacking. Tapjacking ist das mobile Äquivalent zum Web-Angriff Clickjacking. Dabei wird eine gefälschte Anwendungsoberfläche über eine Aktivität einer anderen Anwendung gelegt. Das dient, um Benutzerinnen und Benutzer zu bewegen, Links zu klicken, die nicht beabsichtigt waren. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Im Intranet ist, im Gegensatz dazu, eine Reihe von unbemerkten Angriffen möglich. Einige Attacken benötigen hierbei keine Interaktion des Clients, sodass Einbrüche schwer wahrgenommen werden. (Ziegler, 2008)

4.3 Denial of Service

Als Denial of Service (DoS) wird jeder Angriff bezeichnet der ein System bremst oder komplett lahmlegt. Dabei wird versucht, die Ressourcen auf dem Server mit einer Reihe von Zugriffen aufzubrauchen. Das Ziel ist, dass keine ordentliche Kommunikation mehr möglich ist. Distributed Denial of Service (DDoS) ist, wenn die Attacke mit mehreren Systemen ausgeführt wird. (Ziegler, 2008)

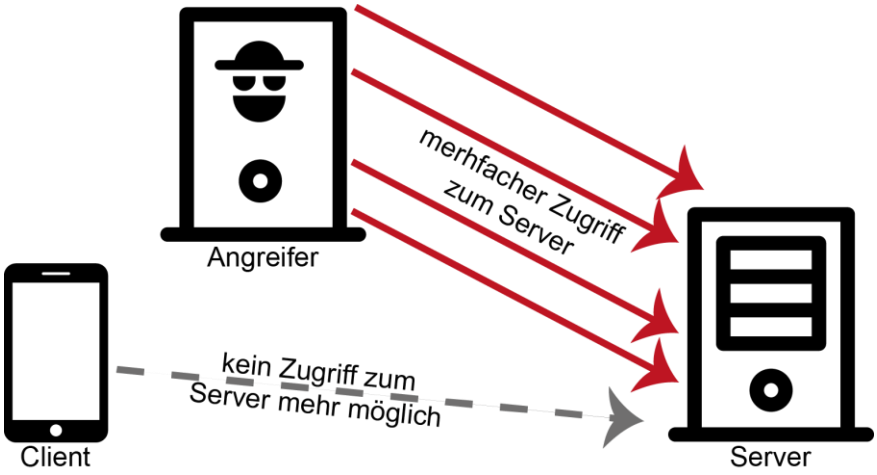


Abbildung 4-2: Denial of Service

Die Abbildung 4-2 veranschaulicht einen DoS-Angriff grafisch. Verbraucht ein Gerät alle Ressourcen des Servers, kann kein weiteres Gerät auf diesem Server zugreifen.

4.4 Injection

Bei einer Injection werden Eingabefelder ausgenutzt, um Befehle einzugeben und anzuwenden. Diese sollen das Zielsystem ausspähen, verändern oder schädigen. Injections können durch eine schlechte Überprüfung der Eingabefelder oder durch unvorsichtige Programmierung von Queries ermöglicht werden. Wie zum Beispiel bei einer ungefilterten Verkettung von Eingaben. Eine Injection kann in unterschiedlichen Bereichen angewendet werden, wie bei Datenbanken durch SQL-Injection oder bei Webservern mittels Directory Traversal. (Drake, et al., 2014)

Auch beim OWASP Mobile Security Project erscheint die Client-Side Injection in der Top 10 der Mobile Risks. Bei mobilen Anwendungen kann dieser Angriff auftreten, wenn Eingaben von einer nicht vertrauenswürdigen Quelle akzeptiert werden. Zum Beispiel kann mit einer Social-Networking-Anwendung Beiträge gepostet oder Statusaktualisierungen von anderen Personen abgerufen werden. Wird dann ein böswilliger Status von einer Angreiferin oder einem Angreifer erstellt, kann dieser Status dann ohne Probleme auf anderen Geräten synchronisiert werden und einen Schaden verursachen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.5 Cross-Site Scripting

Eine weitere große Sicherheitslücke ist das Cross-Site Scripting (XSS). Bei diesem Angriff wird die Ausgabe einer Anwendung attackiert. Hierbei ist die eingesetzte HTML Parsing-Engine wichtig, da sie für die Ausgabe verantwortlich ist. Dieser Angriff zeigt, dass sich das Sicherheitsdenken stark verändert hat. Es muss zusätzlich ein Fokus auf die angezeigten Daten gelegt werden. (Glaser, 2014)

XSS versucht die HTML Parsing-Engine des Browsers auszutricksen, indem sie Programmcode ausführt, wo normalerweise nur Daten angezeigt werden sollten. Diese Attacke ist gefährlich, da sie einfach auf einer Seite gespeichert werden kann. Die Abbildung 4-3 zeigt ein Beispiel wo ein Script bei einem Blog Eintrag hinterlegt wird. (Glaser, 2014)

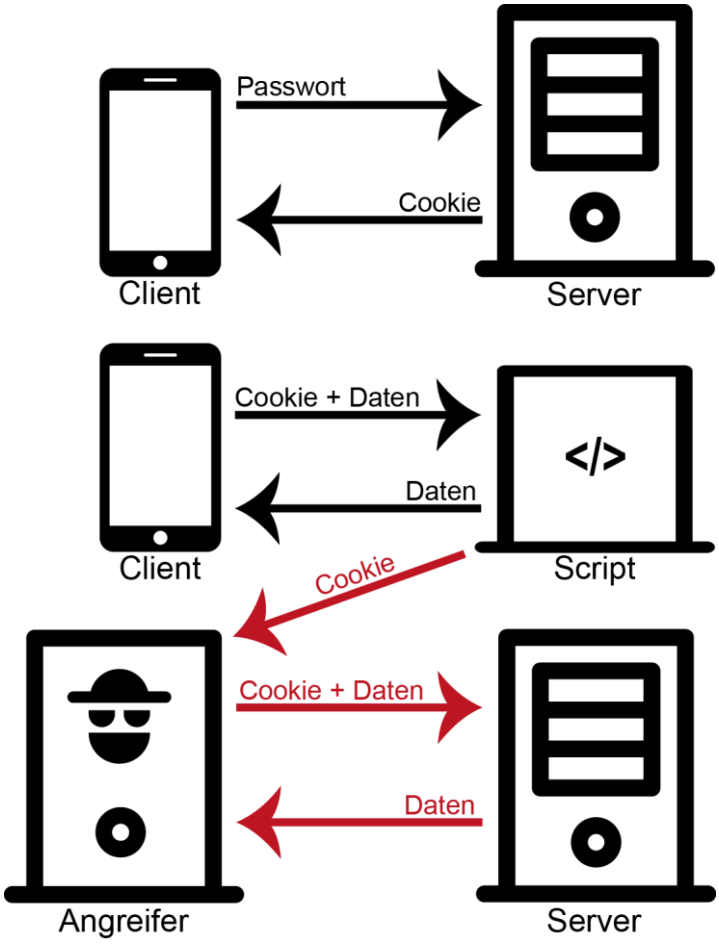


Abbildung 4-3: Cross-Site Scripting

Jedes Mal, wenn eine Webseitenbesucherin oder ein Besucher den Eintrag liest, startet das Script im Hintergrund. Die Besucherinnen und Besucher der Seite führen nun bei jedem Aufruf der Webseite das Script aus. Je nach Script senden sie zum Beispiel alle Cookies, die im Browser sind, weiter. Befinden sich dabei auch aktive Session Cookies, wie die einer Internet Banking Webseite, besteht die Möglichkeit diese zu übernehmen und sich damit einzuloggen. (Glaser, 2014)

4.6 Cross-Site Request Forgery

Bei Cross-Site Request Forgery (XSRF) wird der Anwenderin oder dem Anwender eine gefälschte Anfrage gesendet, die im Hintergrund einen Angriff startet. Das kann unbemerkt passieren, indem zum Beispiel die Anwenderin oder der Anwender auf der Webseite einer Bank angemeldet ist und auf einen Link klickt, der schädlichen JavaScript Code beinhaltet. Dieser Code verwendet das aktuelle Session Cookie, um eine Anfrage zu dieser Bank zu senden. (Glaser, 2014)

Ein Angriff könnte zum Beispiel wie in Abbildung 4-4 aussehen.

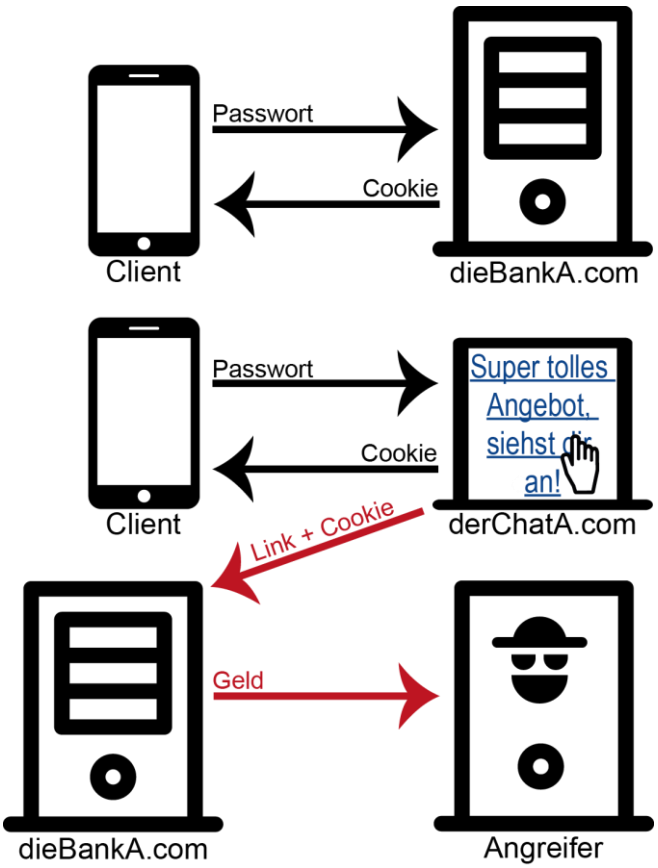


Abbildung 4-4: Cross-Site Request Forgery

Der Ausgangspunkt ist, dass Anwenderinnen oder Anwender gleichzeitigt auf den Webseiten dieBankA.com und auf derChatA.com eingeloggt sind. Sie erhalten dann eine Nachricht mit einem Link auf derChatA.com, die besagt: „Super tolles Angebot, sieh es dir an!“. Wird dieser Link dann angeklickt, enthält er schädlichen JavaScript Code der zur Seite dieBankA.com

verweist. Die gefälschte Anfrage findet dann bei dem Account statt, wo der Link geklickt wurde. Das ebnet den Weg, um Überweisungen an das Angreiferinnen- oder Angreifer-Konto zu tätigen. Das funktioniert, weil das aktive Session Cookie verwendet wird und dieses dann die Anfrage authentifiziert. Überweist die Angreiferin oder der Angreifer nur kleinere Beträge, kann es sein, dass es die Person die bestohlen wird, nicht bemerkt. (Glaser, 2014)

Dieser Angriff kann nur dann funktionieren, wenn dieBankA.com dem authentifizierenden Cookie vertraut und Befehle blind ausgeführt werden. Es kann verhindert werden, indem am Server noch weitere Verifizierungen durchgeführt werden. (Glaser, 2014)

4.7 Session Hijacking

Die vorherigen Beispiele zeigten Diebstähle von Session Cookies. Angriffe dieser Art werden Session Hijacking genannt. Ein Session Cookie ist da, um eine Person eindeutig zu identifizieren. Wenn es aber möglich ist, dieses Cookie in einem anderen Browser zu übernehmen, kann die Session übernommen werden. Das bietet die Möglichkeit, Accounts von anderen Personen zu übernehmen. (Glaser, 2014) Auch vom OWASP Mobile Security Project erscheinen Informationen über „Improper Session Handling“ in den Top 10 der Mobile Risks. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Die einfachste Attacke ist die Session Fixation. Eine Anwenderin oder ein Anwender kann durch eine Anmeldung eine Session ID erstellen und der Server akzeptiert diese ID. Eine Hackerin oder ein Hacker erstellt diese ID und probiert sie der Anwenderin oder dem Anwender anzuhängen. (Glaser, 2014)

Wie in Abbildung 4-5 dargestellt, wird zum Beispiel eine E-Mail mit einem Link der Webseite dieBankA.com, mit einer selbst generierten Session ID, an die Anwenderin oder dem Anwender gesendet. Sie oder er muss dann diesen Link anklicken und wird dann aufgefordert sich auf dieBankA.com anzumelden. Der Server akzeptiert die von der Hackerin oder des Hackers erstellte ID. Nun hat sich eine Ermächtigte oder ein Ermächtigter mit der erfundenen ID authentifiziert und die Hackerin oder der Hacker haben Zugriff. Es besteht nur so lange Zugriff, bis sich die ermächtigte Person abmeldet. Die Lösung für das Problem ist, dass der Server nur die von ihm generierten IDs zulässt und Session IDs neu erstellt, wenn eine Änderung es erfordert. (Glaser, 2014)

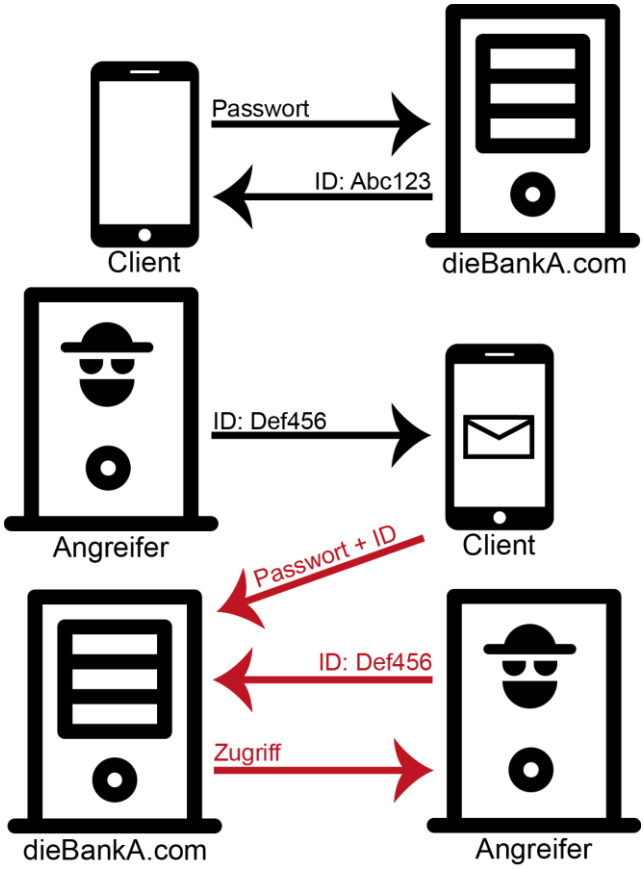


Abbildung 4-5: Session Hijacking

4.8 Weak Server-Side Controls

Weak Server-Side Controls wurde vom OWASP Mobile Security Project von den Top 10 der Mobile Risks, als kritischstes Problem eingestuft. Berechtigt, da die Folgen für ein Unternehmen gravierend sein können. Dieser Angriffspunkt beschreibt jede Schwachstelle, die serverseitig auftaucht, inklusive der mobilen Webservices, Webserverkonfigurationen oder herkömmlichen Webapplikationen. Da dieses Problem nicht auf einem mobilen Gerät stattfindet, ist es umstritten, ob es seine Daseinsberichtigung in der Mobile Risks Liste hat, weil es eine eigene Top 10 Liste für Webapplikationen gibt. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.9 Insecure Data Storage

Wenn Anwendungen sensible Daten als Klartext oder in einem reversiblen Format speichern, wird von Insecure Data Storage gesprochen. Das ermöglicht Malware oder Personen, die Zugriff auf das Dateisystem haben, diese Daten zu stehlen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.10 Insufficient Transport Layer Protection

Insufficient Transport Layer Protection beschreibt Situationen, wo der Datenaustausch nicht vollständig gesichert wird. Zu einem durch unsichere Protokolle oder in Szenarien, wo der Datenverkehr zwar verschlüsselt wurde, aber unsichere Methoden verwendet wurden. Zum Beispiel durch die Anwendung von:

- self-signed Certificates,
- Zertifikate wurden kaum oder mangelhaft validiert oder
- durch Verwendung unsicherer Chiffrensammlungen.

Angriffe können über das Netzwerk gemacht werden, da ein physischer Zugriff auf das Gerät nicht erforderlich ist. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.11 Unintended Data Leakage

Werden Informationen oder sensible Daten irrtümlich so auf einem mobilen Gerät abgespeichert, dass andere Applikationen leicht Zugriff darauf haben, handelt es sich um einen Unintended Data Leakage. Häufige Beispiele für unabsichtliche Datenverluste sind Caching, Snapshots oder Anwendungsprotokolle. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.12 Poor Authorization and Authentication

Weitere Probleme sind Authentifizierungs- und Autorisierungsfehler, die entweder in der mobilen Anwendung oder in der serverseitigen Implementierung auftreten können. Eine lokale Authentifizierung ist innerhalb von mobilen Anwendungen vor allem dann häufig, wenn im Offlinemodus auf Daten zugegriffen werden muss. Werden entsprechende Sicherheitskontrollen ausgelassen, kann die Authentifizierung dadurch umgangen werden, um vollständigen Zugriff auf die Anwendung zu erhalten. Dieses Problem betrifft auch Berechtigungsfehler auf der serverseitigen Anwendung. Es können ohne entsprechende Berechtigungen, Funktionen ausgeführt oder unerlaubter Zugriff zugelassen werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.13 Security Decisions via Untrusted Inputs

Werden Eingaben von einer vertrauenswürdigen Quelle prinzipiell erlaubt, kann es passieren, dass eine Hintertür geöffnet wird. In den meisten Fällen bezieht sich das Risiko auf einen Interprozesskommunikationsmechanismus (inter-process communication = IPC). Zum Beispiel verwendet ein Unternehmen ein Back-End für eine Reihe von Anwendungen und diese Anwendungen kommunizieren miteinander mittels dem gleichem Session Token. Ein IPC-Mechanismus wird benötigt, sodass jede Anwendung Zugriff auf diesen Token hat. Wird dieser

IPC-Mechanismus nicht ordnungsmäßig gesichert, kann eine schädliche Anwendung auf dem Gerät die IPC-Schnittstelle abrufen. Dadurch kann der Session Token aufgerufen und die Sessions der Anwenderin oder des Anwenders gestohlen werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

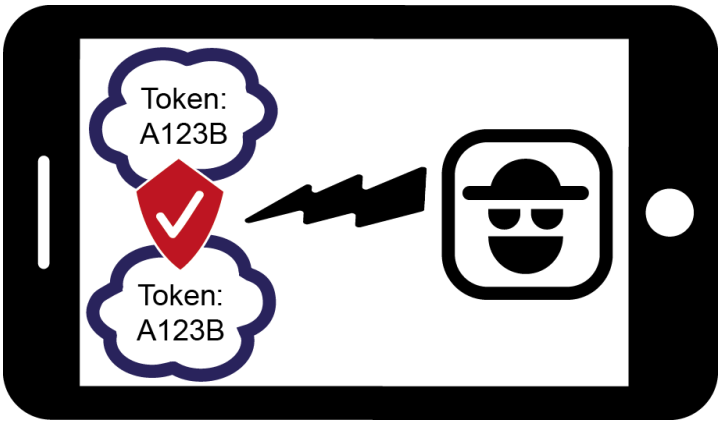


Abbildung 4-6: Security Decisions via Untrusted Inputs

4.14 Broken Cryptography

Werden Daten auf einem mobilen Endgerät gespeichert, sollten diese so gut wie möglich verschlüsselt sein. Ist die Verschlüsselung schlecht implementiert, könnte eine Angreiferin oder ein Angreifer Zugriff auf die Daten erhalten. Dieses Problem kann zum Beispiel durch einen schlechten Key-Management-Prozess, wie das Einbetten des privaten Schlüssels in die Anwendung, auftreten. Aber auch die Verwendungen eines statischen Schlüssels oder eines trivialen Schlüssels, der leicht vom Gerät abgeleitet werden kann, könnte zum Problem werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.15 Lack of Binary Protections

Binary Protections zielen darauf ab, dass der binäre Code schwerer zu analysieren, nach zu konstruieren (Reverse Engineering) oder zu verändern ist. Dabei wird versucht eine Angreiferin oder einen Angreifer bei dem Versuch zu verlangsamen oder wenn möglich zu hindern. (Chell, Erasmus, Colley, & Whitehouse, 2015)

4.16 Zusammenfassung

Dieses Kapitel zeigt, wie unterschiedlich Angriffe sein können. Es gibt zum Beispiel Attacken, die nur an den Server abgerichtet sind, wie "Weak Server-Side Control“. Es gibt jedoch auch Angriffe, die sich auf das mobile Endgerät spezialisieren, wie bei „Insecure Data Storage“ gezeigt wurde. Bei vielen von ihnen werden beide Seiten ins Visier genommen. Das nächste Kapitel befasst sich mit unterschiedlichen Verteidigungsmöglichen, um diese Angriffe abzuschwächen oder zu verhindern.

5 VERTEIDIGUNGSMÖGLICHKEITEN

Welche bekannten Angriffsmöglichkeiten bestehen, wurde im vorherigen Kapitel gezeigt. Zusätzlich zu dieser Information, ist es wichtig zu wissen, wie eine Anwendung geschützt werden kann. Dieses Kapitel widmet sich den unterschiedlichen Verteidigungsmöglichkeiten.

5.1 Sicherung der serverseitigen Steuerelemente

Das Hauptaugenmerk dieser Arbeit liegt zwar im mobilen Bereich, es müssen jedoch auch die serverseitigen Angriffe beachtet werden. Die folgenden Punkte können für die Schwächung von Attacken berücksichtig werden.

5.1.1 Authentifizierung von Datei-Uploads

Nur authentifizierte Benutzerinnen oder Benutzer sollten Uploads erlaubt werden. Dies sichert zwar nicht, dass die Datei harmlos ist, es ermöglicht aber eine Nachverfolgung, wenn nötig. (Glaser, 2014)

5.1.2 White-List für zulässige Datentypen erzeugen

Eine White-List beinhaltet alle Datentypen, die für den Upload zugelassen werden, wie zum Beispiel PDF, GIF oder PNG. Es hindert Benutzerinnen und Benutzer jede Art von Datei hochzuladen, außer die Anwendung ist darauf ausgelegt, jeden Datentyp zu akzeptieren. (Glaser, 2014)

5.1.3 Dateierweiterungen und Datentypen nicht vertrauen

Trotz der White-List können sich schadhafte Dateien in den Uploads befinden. Jede Datei könnte ein Virus sein, da eine Änderung des Datentyps oder der Datenerweiterungen möglich ist. Eine Filterung des Datentyps dient deshalb nur als erste Kontrolle und nicht als sicherer Schutz. (Glaser, 2014)

5.1.4 Systemgenerierten Dateinamen erstellen

Die hochgeladenen Dateien der Benutzerinnen und Benutzern sollten einen systemgenerierten zufälligen Dateinamen am Server erhalten. Eine Angreiferin oder ein Angreifer sollte nicht die Möglichkeit haben, einen Dateinamen mit einem vordefinierten Prozess zu erreichen. (Glaser, 2014)

5.1.5 Hochgeladene Dateien außerhalb des Web Root speichern

Hochgeladene Dateien sollten nicht direkt über ein Web-Request im Webroot-Verzeichnis aufgerufen werden können. Das Speichern von hochgeladenen Dateien mit einem anderen Namen, außerhalb des Web-Root, sorgt dafür, dass Angreiferinnen und Angreifer die Dateien nicht über externe Requests abrufen können. (Glaser, 2014)

5.1.6 Dateigrößen einschränken

Eine maximale Dateigröße sollte immer verwendet werden, sowohl im Upload-Formular als auch am Server. Es dient um DoS-Angriffe zu vermeiden und zu viel Speicherplatz am Server zu verbrauchen. (Glaser, 2014)

5.1.7 Zweistufige Confirmation oder Reauthentication verwenden

Cross-site request forgery und andere Session-Attacken können erschwert werden, indem eine zweistufige Confirmation oder Reauthentication verwendet wird. Es sollte vor allem bei kritischen Handlungen, wie zum Beispiel Geldtransfers, durchgeführt werden. (Stuttard & Pinto, 2011)

5.1.8 Nicht nur auf HTTP-Cookies verweisen, um Session-Token zu übertragen

Cross-site request forgery Angriffe können abgewehrt werden, indem nicht nur auf HTTP-Cookies vertraut wird, um Session Tokens zu übertragen. Die Verwendung von Cookies stellt ein Sicherheitsrisiko dar, da Cookies automatisch vom Browser übermittelt werden, unabhängig davon, wer für den Request verantwortlich ist. Eine Angreiferin oder ein Angreifer kann kein autorisiertes Formular erstellen, wenn Tokens immer in einem verstecken Feld eines HTML-Formulars mit übertragen werden. Die Submission würde immer zu einer unautorisierten Handlung führen, außer der Wert des Tokens ist bekannt. In diesem Fall kann ein Hijacking-Angriff durchgeführt werden. (Stuttard & Pinto, 2011)

5.2 Daten sicher aufbewahren

In Android können Daten mittels Shared Preferences, SQLite-Datenbank oder direkt als Datei gespeichert werden. Darüber hinaus kann jeder dieser Speichertypen, auf verschiedene Arten erstellt und abgerufen werden. Die häufigsten Fehler bei der Speicherung sind Klartextspeicherungen von sensiblen Daten, ungeschützte Content Provider oder unsichere Dateizugriffsrechte. (Drake, et al., 2014)

Beim Erstellen einer Datei ist deshalb eine explizite Angabe der Dateiberechtigung besser, als auf das vom System definierte Umask-Set zu verweisen. Folgendes Beispiel zeigt eine explizite Angabe von Berechtigungen unter Android, sodass nur die Anwendung auf Dateien Zugriff hat, die diese erstellt hat:

```
FileOutputStream secretFile = openFileOutput("secret", Context.MODE_PRIVATE);
```
(Chell, Erasmus, Colley, & Whitehouse, 2015)

Darüber hinaus kann ein Ordner im privaten Datenverzeichnis der Anwendung erstellt werden, der mit gesicherten Berechtigungen festgelegt wird:

```
File newdir = getDir("newdir", Context.MODE_PRIVATE);
```
(Chell, Erasmus, Colley, & Whitehouse, 2015)

Es besteht auch die Möglichkeit, das Gleiche durch die Verwendung der statischen Ganzzahlen, zu verwenden:

```
FileOutputStream secretFile = openFileOutput("secret", 1);
```
(Chell, Erasmus, Colley, & Whitehouse, 2015)

Das direkte Anzeigen von Ganzzahlen, die die Berechtigungen darstellen, ist nicht empfehlenswert, da dieser Code schwer lesbar wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Wenn nativer Code zum Erstellen einer Datei verwendet wird, kann dieser mit expliziten Berechtigungen versehen werden. Dieses Beispiel zeigt, wie dies in der `open`-Funktion geschieht:

```
FILE * secretFile = open("/data/data/com.myapp/files/secret", O_CREAT|O_RDWR,
                    S_IRUSR|S_IWUSR);
```
(Chell, Erasmus, Colley, & Whitehouse, 2015)

Dadurch wird eine Datei mit Berechtigungen erstellt, die dem Anwendungseigentümer nur lesen und schreiben erlaubt. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Es besteht auch die Möglichkeit die Datei zu verschlüsseln, sodass keine andere Anwendung die Datei auslesen kann. Dabei wird die Datei verschlüsselt, bevor sie auf der SD-Karte gespeichert wird. Eine Angreiferin oder ein Angreifer muss ein Passwort verwenden, um die Datei zu entschlüsseln und zu lesen. (Gunasekera, 2012)

Für das folgende Beispiel wird der Advanced Encryption Standard (AES) verwendet, um die Daten mit einem Passwort oder Key zu verschlüsseln. Ein Key ist erforderlich, um die Daten zu verschlüsseln und zu entschlüsseln. Dies wird auch als symmetrische Key-Encryption bezeichnet. Im Gegensatz zur Public-Key- Encryption, ist dieser Key der einzige, der sowohl zum

Verschlüsseln als auch zum Entschlüsseln von Daten verwendet wird. Aufgrund dessen muss er sicher gespeichert werden, da er sonst von einer Angreiferin oder einem Angreifer verwendet werden kann, um die Daten zu entschlüsseln. Das Listing 5-1 zeigt eine Implementierung einer Verschlüsselung. (Gunasekera, 2012)

```
private static byte[] encrypt(byte[] key, byte[] data){
    SecretKeySpec sKeySpec = new SecretKeySpec(key,"AES");
    Cipher cipher;
    byte[] ciphertext = null;
    try {
          cipher = Cipher.getInstance("AES");
          cipher.init(Cipher.ENCRYPT_MODE, sKeySpec);
          ciphertext = cipher.doFinal(data);
    } catch (NoSuchAlgorithmException e) {
          Log.e(TAG,"NoSuchAlgorithmException");
    } catch (NoSuchPaddingException e) {
          Log.e(TAG,"NoSuchPaddingException");
    } catch (IllegalBlockSizeException e) {
          Log.e(TAG,"IllegalBlockSizeException");
    } catch (BadPaddingException e) {
          Log.e(TAG,"BadPaddingException");
    } catch (InvalidKeyException e) {
          Log.e(TAG,"InvalidKeyException");
    }
    return ciphertext;
}
```

Listing 5-1: Verschlüsselung von Daten in Android mittels AES (Gunasekera, 2012)

Beim Listing 5-2 wird die `SecretKeySpec`-Klasse initialisiert und eine neue Instanz der `Cipher`-Klasse erstellt, um die Erstellung eines AES-Geheimschlüssels vorzubereiten. (Gunasekera, 2012)

```
SecretKeySpec sKeySpec = new SecretKeySpec(key,"AES");
Cipher cipher;
byte[] ciphertext = null;
```

Listing 5-2: Initalisierung der SecretKeySpec-Klasse (Gunasekera, 2012)

Des Weiteren wird ein `byte`-Array initialisiert, um den Cipher-Text zu speichern. Der Code vom Listing 5-3 bereitet die `Cipher`-Klasse vor, um den AES-Algorithmus zu verwenden. (Gunasekera, 2012)

```
cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, sKeySpec);
```

Listing 5-3: Vorbereitung der Cipher-Klasse für den AES-Algorithmus (Gunasekera, 2012)

Die Funktion `cipher.init()` initialisiert das Cipher-Objekt, so dass es die Verschlüsselung mit dem generierten geheimen Key ausführen kann. Die nächste Codezeile verschlüsselt die Klartextdaten und speichert die verschlüsselten Inhalte im `byte`-Array `ciphertext`:

`ciphertext = cipher.doFinal(data);` (Gunasekera, 2012)

Damit die vorhergehende Routine funktionieren kann, sollte sie immer einen Encryption-Key haben. Es ist wichtig, dass der gleiche Schlüssel für die Decryption-Routine verwendet wird, da es sonst nicht funktioniert. Es ist von Vorteil einen eigenen Key-Generator zu erstellen, der einen zufälligen nummernbasierten Key erzeugt, statt eines normalen Passworts, da dieser schwieriger zu erraten ist. (Gunasekera, 2012) Das folgende Listing 5-4 zeigt einen Key-Generator-Algorithmus:

```
public static byte[] generateKey(byte[] randomNumberSeed) {
    SecretKey sKey = null;
    try {
          KeyGenerator keyGen = KeyGenerator.getInstance("AES");
          SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
          random.setSeed(randomNumberSeed);
          keyGen.init(256,random);
          sKey = keyGen.generateKey();
    } catch (NoSuchAlgorithmException e) {
          Log.e(TAG,"No such algorithm exception");
    }
    return sKey.getEncoded();
}
```

Listing 5-4: Key-Generator-Algorithmus (Gunasekera, 2012)

Beim Listing 5-5 wird zuerst die die `KeyGenerator`-Klasse initialisiert, damit ein AES-spezifischer Key erzeugt werden kann und danach wird die `SecureRandom`-Klasse initialisiert, um zufällige Zahlen zu generieren. (Gunasekera, 2012)

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
```

Listing 5-5: Initialisierung der KeyGenerator-Klasse und der SecureRandom-Klasse (Gunasekera, 2012)

Diese Zufallszahlen werden mit Secure Hash Algorithmus 1 (SHA1) codiert. SHA1 ist eine kryptografische Hashfunktion. Der Algorithmus wird auf einem Datenelement mit einer beliebigen Länge angewandt und erzeugt einen kurzen String mit einer bestimmten Größe. Wenn ein gehashter Teil der Daten verändert wird, kann der resultierende Hash abweichen. Dies ist ein

Hinweis darauf, dass ein Teil der Daten manipuliert wurde. Das Code-Snippet vom Listing 5-6 generiert einen 256-Bit-Key. (Gunasekera, 2012)

```
random.setSeed(randomNumberSeed);
keyGen.init(256,random);
sKey = keyGen.generateKey();
```

Listing 5-6: Generierung eines 256-Bit-Key (Gunasekera, 2012)

Für die Verwendung wird zuerst der Key-Generator-Algorithmus einmal ausgeführt und gespeichert. Danach wird dieser Key für die Decryption-Routine verwendet. (Gunasekera, 2012)

Apple hat erkannt, dass eine sichere Speicherung von Daten wichtig ist und hat aus diesem Grund eine Data-Protection-API zu Verfügung gestellt. Diese verwendet eine integrierte Hardwareverschlüsselung, sodass eine grundlegende Dateiverschlüsselung ermöglicht wird, um einen Encryption Key pro Datei zu generieren. Jeder Encryption Key wird durch die `protection`-Klasse zu den jeweiligen Entwicklerinnen oder Entwicklern zugeteilt. Die `protection`-Klasse steuert, wann ein Class-Key im Speicher bleibt und zum Verschlüsseln oder Entschlüsseln von Encryption-Keys verwendet wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Mittels der Data-Protection-API kann die jeweilige `protection`-Klasse zu den Dateien zugeordnet werden, wodurch vier Levels zum Schutz des Dateisystems entstehen. Diese Klassen können durch Übergabe eines erweiterten Attributs an die `NSData` oder `NSFileManager` Klassen konfiguriert werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

- `NSFileProtectionNone`: Kein Schutz – Die Datei wird nicht auf dem Dateisystem verschlüsselt.
- `NSFileProtectionComplete`: Vollständiger Schutz – Die Datei ist auf dem Dateisystem verschlüsselt und ist nicht verfügbar, wenn das Gerät gesperrt ist.
- `NSFileProtectionCompleteUnlessOpen`: Komplett, wenn nicht geöffnet – Die Datei ist im Dateisystem verschlüsselt und nicht verfügbar, wenn sie geschlossen ist. Wenn ein Gerät entsperrt ist, kann eine Anwendung einen Zugriff zu der Datei erhalten, selbst wenn diese gesperrt wurde. Während dieser Zeit wird die Datei nicht verschlüsselt.
- `NSFileProtectionCompleteUntilFirstUserAuthentication`: Vollständiger Schutz bis First User Authentication – Die Datei ist auf dem Dateisystem verschlüsselt und unzugänglich, bis das Gerät zum ersten Mal entsperrt wurde. Dies bietet Schutz gegen Angriffe, die einen Neustart des Geräts erfordern.

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Das folgende Listing 5-7 zeigt ein Beispiel, wie das Attribut der `protection`-Klasse für eine Datei festgelegt, heruntergeladen und im `documents`-Verzeichnis gespeichert wird.

```
-(BOOL) getFile
{
      NSString *fileURL = @"https://www.mdsec.co.uk/pdfs/wahh-live.pdf";
      NSURL *url = [NSURL URLWithString:fileURL];
      NSData *urlData = [NSData dataWithContentsOfURL:url];
      if ( urlData )
      {
            NSArray *paths =
            NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
            NSUserDomainMask, YES);
            NSString *documentsDirectory = [paths objectAtIndex:0];
            NSString *filePath = [NSString stringWithFormat:@"%@/%@",
            documentsDirectory,@"wahh-live.pdf"];
            NSError *error = nil;
            [urlData writeToFile:filePath
            options:NSDataWritingFileProtectionComplete error:&error];
            return YES;
      }
      return NO;
}
```

Listing 5-7: Beispiel für die protection-Klasse (Chell, Erasmus, Colley, & Whitehouse, 2015)

In diesem Beispiel ist das Dokument dann erreichbar, wenn das Gerät entsperrt ist. Unter iOS kann auf eine Datei innerhalb von zehn Sekunden nach dem Sperren zugegriffen werden, bis der Zugriff verweigert wird. (Chell, Erasmus, Colley, & Whitehouse, 2015) Der Ausschnitt vom Listing 5-8 zeigt einen Versuch auf die Datei nach dieser Zeit zuzugreifen, während das Gerät gesperrt ist.

```
$ ls -al Documents/ total 372
drwxr-xr-x 2 mobile mobile 102 Jul 20 15:24 ./
drwxr-xr-x 6 mobile mobile 204 Jul 20 15:23 ../
-rw-r--r-- 1 mobile mobile 379851 Jul 20 15:24 wahh-live.pdf
$ strings Documents/wahh-live.pdf
strings: can't open file: Documents/wahh-live.pdf
(Operation not permitted)
```

Listing 5-8: Zugriff auf Datei, mehr als zehn Sekunden nach dem Sperren (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.3 Validierung von Eingaben

iOS und Android Anwendungen können Eingaben von unterschiedlichen Eintrittspunkten verarbeiten, wie zum Beispiel:

- Web-Anwendungen
- Datentypen (zum Beispiel Bilder oder Dokumente)
- Bluetooth
- Wi-Fi
- Anwendungserweiterungen

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Injection-Angriffe können in jedem Bereich auftreten, wo eine User-Eingabe akzeptiert wird. Aus diesem Grund ist es wichtig, dass alle Eintrittspunkte einer Anwendung überprüft werden.

5.3.1 SQL Injection

Einer der häufigsten Injection-Angriffe sind SQL-Injections. Dieser Angriff kann jederzeit zu Stande kommen, wenn schädigende Daten in eine SQL-Abfrage eingetragen werden. Die Konsequenzen können gemindert werden, wenn geeignete Vorkehrmaßnahmen getroffen wurden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Wie bei Webapplikationen können diese Angriffe abgewehrt werden, indem SQL-Abfragen parametrisiert werden. Dabei werden statt den Strings, Platzhalter für die Abfrage eingesetzt. Bei iOS Applikationen wird am häufigsten SQLite eingesetzt. SQLite bietet `sqlite3_prepare`, `sqlite3_bind_text` und ähnliche Funktionen, um Abfragen zu parametrisieren und zugehörige Werte an die Parameter zu binden. Das Listing 5-9 zeigt, wie eine Abfrage erstellt werden kann, in der parametrisiert wird und die User-Controller-Werte an die Abfrage gebunden werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

```
NSString* safeInsert = @"INSERT INTO messages(uid, message, username)
VALUES(?, ?, ?)";
if(sqlite3_prepare(database, [safeInsert UTF8String], -1, &statement,
NULL) != SQLITE_OK)
{
    // Unable to prepare statement
}
if(sqlite3_bind_text(statement, 2, [status.message UTF8String], -1,
SQLITE_TRANSIENT) != SQLITE_OK)
{
    // Unable to bind variabless
}
```

Listing 5-9: Parametrisierte SQL-Abfrage in iOS (Chell, Erasmus, Colley, & Whitehouse, 2015)

Dieses Beispiel zeigt, wie die Variable `status.message` an eine Textspalte in einer Abfrage gebunden werden kann. Um weitere Variablen hinzuzufügen, wird ein ähnlicher Code und eine angepasste Funktion für den jeweiligen Spaltentyp an dem es gebunden werden soll, benötigt. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Wenn Eingaben von Benutzerinnen und Benutzern direkt innerhalb eines SQL-Statement verwendet werden, können sie anfällig für Injection-Angriffe werden. Werden mittels der `rawQuery()`-Methode von der `SQLiteDatabase` die Eingaben direkt verkettet, ist ein Angriff möglich. Wie in iOS können SQL-Injection-Angriffe in Android mit vorgefertigten Statements verhindert werden. Das Listing 5-10 zeigt ein Beispiel, wie ein Statement aussehen könnte. (Chell, Erasmus, Colley, & Whitehouse, 2015)

```
String[] userInput = new String[] {"book", "wiley"};
Cursor c = database.rawQuery("SELECT * FROM Products WHERE type=?
AND brand=?", userInput);
```

Listing 5-10: Vorgefertigte SQL-Statements in Android (Chell, Erasmus, Colley, & Whitehouse, 2015)

Die Verwendung von vorgefertigten Statements stellt sicher, dass Eingaben von Benutzerinnen und Benutzern korrekt abbrechen und nicht Teil der SQL-Abfrage werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.3.2 Interprozesskommunikation

Ein Angriff auf eine Anwendung über das Android-IPC-System setzt voraus, dass exportierte Komponenten einer Anwendung gefunden und missbräuchlich verwendet werden. Das beinhaltet Activities, Broadcast Receivers, und Services. Jeder Code der mit Intents arbeitet, sollte auf eine mögliche missbräuchliche Verwendung überprüft werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

In iOS ist aufgrund der Sandbox, in der eine Anwendung isoliert arbeitet, eine Interprozesskommunikation verboten. Die Möglichkeit einen Angriff zu starten, besteht durch folgende Ausnahmen:

- Pasteboard
- Registrierte Protokollhandler
- Anwendungserweiterungen
  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Wie in Android sollte deshalb jeder IPC-Endpunkt einer Sicherheitsüberprüfung unterzogen werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.3.3 Directory Traversal

Die Basis für die Überprüfung ob eine andere Anwendung einen Directory Traversal Angriff auf einen Content Provider in Android startet, ist den Resultatordner gegen einen bekannten Wert zu überprüfen. Das bedeutet, dass die angeforderte Datei innerhalb eines „allowed“-Ordner bleibt. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Mithilfe der Methode `getCanonicalPath()` der Klasse `File` kann das ermöglicht werden. Diese übersetzt einen Pfad in einen, der die resultierenden „.“- und „..“-Zeichen entfernt und arbeitete in dem resultierenden Pfad weiter. Um diesen Angriff zu verhindern wird zuerst diese Prüfung durchgeführt und dann wird sie mit einer Liste der zulässigen Dateien in einem bestimmten Verzeichnis oder gegen den Speicherort des Verzeichnisses verglichen. (Chell, Erasmus, Colley, & Whitehouse, 2015) Der folgende Code im Listing 5-11 beschränkt andere Anwendungen auf das Lesen von Dateien im Verzeichnis `/files/` im privaten Datenverzeichnis der eigenen Anwendung:

```
@Override
public ParcelFileDescriptor openFile (Uri uri, String mode)
{
     try
     {
     String baseFolder = getContext().getFilesDir().getPath();
     File requestedFile = new File(uri.getPath());
     //Only allow the retrieval of files from the /files/
     //directory in the private data directory
     if (requestedFile.getCanonicalPath().startsWith(baseFolder))
          return ParcelFileDescriptor.open(requestedFile,
                ParcelFileDescriptor.MODE_READ_ONLY);
     else
          return null;
     }
     catch (FileNotFoundException e)
     {
          return null;
     }
     catch (IOException e)
     {
          return null;
     }
}
```

Listing 5-11: Leserechte von Dateien im Verzeichnis /files/ (Chell, Erasmus, Colley, & Whitehouse, 2015)

Unter iOS sind Directory Traversal Angriffe durch das Berechtigungsschema schwieriger durchzuführen. Dennoch können bei der Änderung von Dateinamen durch Benutzerinnen und Benutzern Schwachstellen auftreten. Aus diesem Grund sollte beim Erstellen von Dateinamen darauf geachtet werden, dass keine schadhaften Namen zugelassen werden. Es werden zwei Hauptklassen für die Dateiverarbeitung in der iOS-SDK verwendet:

- `NSFileManager`
- `NSFileHandle`
  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Wenn eine Angreiferin oder ein Angreifer einen Teil des Dateinamens steuern kann, können diese Klassen von einem Directory Traversal-Problem betroffen sein. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.4 Sichere Kommunikation

Da bei mobilen Applikationen eine Netzwerkkommunikation möglich ist, stehen durch das Empfangen und Senden von Daten mehr Möglichkeiten offen, als bei einer Anwendung die keinen Zugriff auf ein Netzwerk hat. Um eine Netzwerkverbindung zu erhalten, können von Anwenderinnen und Anwendern auch nicht vertrauenswürdige oder unsichere Netzwerke verwendet werden, wie zum Beispiel in Hotels oder Cafés. Aus diesen Grund sollte auf eine sichere Kommunikation in mobilen Anwendungen geachtet werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

In der Webentwicklung wird auf ein sicheres Übertragen der Daten Rücksicht genommen, in der mobilen Welt wird es noch vernachlässigt. Sensible Daten werden bei der Übertragung nicht ausreichend geschützt. Dabei kann die Suche nach nicht sicheren Übertragungen einfach sein, indem der Datenverkehr eines ausgewählten Gerätes überwacht wird. (Drake, et al., 2014) Im Folgenden wird gezeigt, wie einige Angriffe und sichere Kommunikationen aussehen können.

5.4.1 Cross-Site Scripting vermeiden

Cross-Site-Scripting (XSS) kann in iOS jederzeit auftreten, wenn schädigende Daten in ein `UIWebView` eingetragen werden. Je nachdem wie die Webansicht geladen wird, welche Berechtigungen die Anwendung hat oder ob die Anwendungen zusätzliche Funktionen wie eine JavaScript zu Objective-C-Bridge verwendet, können die Konsequenzen variieren. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Folgende Ansätze helfen nicht nur Auswirkungen durch XSS-Angriffe zu mindern, sie können auch dabei helfen diese Angriffe zu vereiteln:

- Es soll darauf geachtet werden, dass `UIWebView` als Original geladen wird, indem ein laden von `file:// protocol handler` vermieden wird.
- Es soll vorsorglich darauf geachtet werden, dass keine schädigenden Daten in JavaScript Strings eingetragen und in der Webansicht ausgeführt werden können. Das passiert, wenn vom `UIWebView` die Methode `stringByEvaluatingJavaScriptFromString` eingesetzt wird.
- Es soll bei dynamisch generierten HTML Code für das `UIWebView` auf schädigende Daten geachtet werden. Bevor das HTML in die Webansicht geladen wird, sollten alle Inhalte vollständig überprüft und kodiert werden. Dieses Problem tritt vor allem bei der `UIWebView`-Methode `loadHTMLString` auf.

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Android Applikationen können nativ, HTML5 oder Hybrid-Anwendungen mit einigen HTML Elementen sein. Sie bestehen aber meistens aus nativen Java Code. Wird eine WebView-Applikation erstellt, sollte abgewogen werden, ob `setJavaScriptEnabled` in den WebViews auf `true` gesetzt wird oder nicht. Wird JavaScript zugelassen, kann das einen XSS-Angriff ermöglichen. Es ist zum Beispiel möglich mittels JavaScript Zugriff zu den Shared Preferences

zu erhalten, indem der Befehl `file:///` verwendet wird. Des Weiteren können unerwünschte SMS-Nachrichten mithilfe von `smsJSInterface.launchSMSActivity` vom Telefon gesendet werden. (Nolan, 2015) Folgender Code zeigt das Setzen von `setJavaScriptEnabled` auf `false`, sodass kein JavaScript-Code mehr ausgeführt werden kann:

```
myWebView.getSettings().setJavaScriptEnabled(false);
```
(Nolan, 2015)

Wird in mobilen Anwendungen mit HTML oder XML gearbeitet, kann es passieren, dass schädigende Daten in eine Webansicht dynamisch eingetragen werden. Um einen XSS-Angriff zu vermeiden, sollten alle Daten vorher kodiert werden, die zu einer Schädigung führen könnten. (Chell, Erasmus, Colley, & Whitehouse, 2015) Folgende Regeln helfen dabei Metazeichen zu codiert:

- Weniger als `(<)` wird mit `&lt` überall ersetzt
- Größer als `(>)` wird mit `&gt` überall ersetzt
- Das Et-Zeichen `(&)` wird mit `&amp` überall ersetzt
- Das doppelte Anführungszeichen `(")` wird mit `&quot` innerhalb der Attributwerte ersetzt
- Das einfache Anführungszeichen `(')` wird mit `&apos` innerhalb der Attributwerte ersetzt

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.4.2 Sichere Übertragung von Daten

Jede Kommunikation im Netzwerk sollte vor Abhörangriffen oder Manipulation gesichert werden, unabhängig davon, ob Daten gesendet oder empfangen werden. Ein gängiges Missverständnis ist, dass nur sensible Daten verschlüsselt werden sollten, wie zum Beispiel die Authentifizierung. Jede Datenübertragung als Klartext, wie bei HTTP, ist anfällig für fremde Änderungen. Eine Angreiferin oder ein Angreifer kann überall zwischen dem Gerät und dem Server einen Angriff starten, um Inhalte abzufangen und zu modifizieren. Vor allem sind Authentifizierungen kritisch, da Benutzerinnen und Benutzer dieselben Anmeldedaten bei mehreren Webseiten verwenden könnten. Somit wäre ein Zugriff auf einen E-Mail Account oder sensible Services möglich. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Zum Beispiel wird unter iOS `UIWebView` verwendet, um eine Anfrage ohne sensible Daten zu einer Webapplikation zu senden. Eine Angreiferin oder ein Angreifer könnte einen Man-in-the-middle-Angriff starten und einen schädlichen JavaScript Code mittels Cross-Side Scripting einspeisen. Um herauszufinden, wann eine Anwendung einen Klartext-Request sendet, könnten zusätzliche Anwendungen verwendet werden. Zuerst kann die Anwendung passiv mittels eines Packet Capturing Tools wie Wireshark getestet werden. Es besteht auch die Möglichkeit einer Überprüfung durch einen Proxy mittels der Burp Suite. Mit dieser Methode ist es nur möglich HTTP-basierenden Verkehr zu erkennen. Um das Risiko eines unverschlüsselten Abhörens zu vermeiden, kann der Secure Socket Layer (SSL) oder die Transport Layer Security (TLS) verwendet werden. Das SSL-Protokoll und sein Nachfolger, das TLS-Protokoll, werden als Standard für sichere Netzwerkkommunikation akzeptiert und als sicheres Transportmedium für

HTTP verwendet. (Chell, Erasmus, Colley, & Whitehouse, 2015) Apple bietet drei Möglichkeiten zur Implementierung von SSL und TLS:

- URL Loading System – Diese API enthält eine Anzahl von High-Level Helper-Klassen und Methoden, wie `NSURLConnection` und `NSURLSession`, die für sichere HTTP-Request verwendet werden können. Das URL Loading System ist die einfachste Methode für die Erstellung von URL-Requests.
- Carbon Framework – Diese API ist granularer als das URL Loading System und gibt Entwicklerinnen und Entwicklern eine größere Kontrolle über Netzwerk-Requests. Es wird in der Regel mit der `CFNetwork`-Klasse implementiert.
- Secure Transport API – Diese Low-Level API bietet die größte Kontrolle über den Transport und ist komplex zu implementieren. Aus diesem Grund wird sie selten direkt verwendet.

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Unabhängig von der API, die die Anwendung verwendet, kann eine SSL- oder TLS-Verbindung in einer Reihe von Angriffen geschwächt werden. Folgende Beispiele zeigen bekannte Implementierungsfehler die bei der Entwicklung auftreten können. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Zertifikatsvalidierung

SSL und TLS basieren auf dem grundlegenden Konzept der Certificate-Based Authentifizierung. Durch dieses Verfahren wird sichergestellt, dass mit dem korrekten Server kommuniziert wird und es verhindert auch Abhörangriffe und Manipulationsangriffe. Das kritischste beim Einrichten einer SSL-Sitzung ist das Vertrauen eines Zertifikates, das nicht von der Certificate Authority (CA) zertifiziert wurde. Eine Anwendung die ein Self-Signed Zertifikat akzeptiert, kann nicht bestätigen, ob der Server derjenige ist, für den er sich ausgibt. Wodurch die Applikationen anfällig für Abhörangriffe und Manipulationen von jeder Angreiferin oder Angreifer in geeigneter Position im Netzwerk ist. (Chell, Erasmus, Colley, & Whitehouse, 2015)

SSL Session Security

Zusätzlich dazu bestehen bei der Apple-API weitere Möglichkeiten eine SSL-Session zu untergraben. Wird die High-Level URL Loading API verwendet, besteht keine Bedrohung, da diese nicht granular genug für Änderungen der Eigenschaften von SSL/TLS-Sessions ist. Wird das Carbon Framework oder die Secure Transport API verwendet, sollte Folgendes beachtet werden:

- **Protocol Versions:** Die `CFNetwork`- und die Secure Transport APIs ermöglichen es einer Entwicklerin oder einem Entwickler, die Protokollversion für SSL- oder TLS-Sitzungen zu ändern. Die Versionen SSLv2 und SSLv3 sind anfällig für Schwachstellen. Zum Beispiel für eine Anzahl verschiedener Angriffe, die es einer Angreiferin oder einen Angreifer von einer geeigneten Position im Netzwerk erlauben, den verschlüsselten Ciphertext als Klartext zu erhalten. Bei der `CFNetwork`-API kann die Protokollversion mittels dem `kCFStreamPropertySSLSettings`-Dictionary konfiguriert werden. Bei der Secure

Transport API kann sie mit den Funktionen `SSLSetProtocolVersion()` oder `SSLSetProtocolVersionEnabled()` konfiguriert werden.

- **Cipher Suite Negotiation:** Die Cipher-Suite ist eine Kombination aus Authentifizierung, Verschlüsselung, Message Authentication Code (MAC) und Key Exchange Algorithmen. Diese werden verwendet, um ein sicheres Netzwerk mit SSL/TLS zu erstellen. Bei der Secure Transport API und `CFNetwork`-API ist es möglich die Cipher-Suit für eine SSL/TLS-Session zu konfigurieren. Das bedeutet, dass durch falsche Konfiguration oder durch Unwissenheit eine Cipher-Suite verwendet wird, die nicht kryptographisch sicher ist. Die Liste der verfügbaren Cipher-Suites ist umfangreich. Die von der `CFNetwork`-API und der Secure Transport API unterstützt werden, haben alle Einträge in der `SSLCipherSuite`-Enum, die von Apple unter der folgenden URL dokumentiert wird: https://developer.apple.com/reference/security/secure_transport

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Android bietet APIs an, mit denen sicherere Kommunikationskanäle erstellt werden können. Wie bei iOS besteht die Möglichkeit SSL zu verwenden. Das Problem bei SSL ist, dass es abhängig von einer Reihe von vertrauenswürdigen CAs für die Validierung ist. Einer einzelnen zu vertrauen, kann sich auf die Sicherheit der Clients auswirken, die nur dieser Zertifizierungsstelle vertrauen. Wird das Signaturzertifikat kompromittiert und für diese eine Quelle betrügerische Zertifikate ausgestellt, würden diese Zertifikate von einem Client trotzdem als vertrauenswürdig eingestuft werden. Das Kompromittieren von vertrauenswürdigen CA-Zertifikaten ist ein bekannter Schwachpunkt. Um sich vor diesem Angriff zu schützen kann SSL-Certificate-Pinning verwendet werden. Eine Verbindung ist nur dann möglich, wenn bestimmte Attribute des Zertifikates vom Server gegen gespeicherte Werte auf dem Gerät positiv übereinstimmen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Ein weiterer wichtiger Punkt ist, dass bevor eine Anwendung veröffentlicht wird, die Bereiche des Codes überprüft werden, die mit der SSL-Verbindung verknüpft sind. Das wird benötigt, um sicherzustellen, dass das Zertifikat nicht umgangen werden kann. Dazu kann ein benutzerdefinierter `HostnameVerifier` und `TrustManager` implementiert werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Sowie für iOS ist es für Android hilfreich einen Proxy (wie zum Beispiel Burp Suite) einzurichten. Dieser ermöglicht den gesamten Inhalt eines Web-Traffics zwischen Applikation und Internet einzusehen und die Änderung von Requests und Responses. (Chell, Erasmus, Colley, & Whitehouse, 2015)

WebView Security

Wie in Kapitel „5.4.1 Cross-Site Scripting vermeiden" erwähnt, können durch WebViews Sicherheitslücken entstehen. Der Unterschied von Webseiten im Webbrowser und von denen in WebViews liegt darin, dass ein WebView im Kontext einer Applikation eingebettet ist. Das ermöglicht unterschiedliche Angriffsmöglichkeiten, um das Verhalten während der Laufzeit zu ändern oder um bestimmte Events während des Ladens einer Seite aufzufangen. Werden Inhalte

zum Beispiel als Klartext geladen, können verschiedene Formen von Man-in-the-Middle-Angriffen ermöglicht werden. Während der Entwicklung einer Applikation kann es nötig sein, SSL-Zertifikate vorübergehen zu übergehen. Wird dieser Code vor der Veröffentlichung nicht entfernt und getestet, ist es einfach möglich SSL zu übergehen. Aus diesem Grund sollte ein Developer Zertifikat eingerichtet werden oder vor der Veröffentlichung alle Entwicklercodes und Logs entfernt werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

WebViews haben einige Funktionen die eine Angreiferin oder ein Angreifer für den eigenen Vorteil ausnützen könnten. Um diese Angriffsfläche zu limitieren, sollten eine Informationswebseite mittels Link geöffnet werden, statt als eingebettete WebView. Diese Methode ist sicherer, da im Android-Browser der Inhalt innerhalb vom Context der Sandbox geladen wird. Wird der Browser kompromittiert, hat das keinen Einfluss auf die Daten innerhalb der der eigenen Applikation. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Eine weitere Angriffsfläche ist, das WebViews standardmäßig Dateien aus dem Dateisystem laden können. Dies stellt ein Problem dar, wenn böswillige Anwendungen Zugriff auf lokale Dateien innerhalb einer anderen Anwendungs-WebView haben. Der Dateisystemzugriff von einem WebView kann wie folgt deaktiviert werden:

`webview.getSettings().setAllowFileAccess(false);` (Chell, Erasmus, Colley, & Whitehouse, 2015)

Dies verhindert nicht, dass Laden aus dem Applikationsordnern `resources` oder `assets` mithilfe von `file:///android_res` und `file:///android_asset`. Um das WebView noch weiter zu sperren, sollten geladene Seiten nicht aus dem Dateisystem auf andere Dateien im Dateisystem zugreifen können. Das folgende Beispiel zeigt eine Möglichkeit:

`webview.getSettings().setAllowFileAccessFromFileURLs(false);` (Chell, Erasmus, Colley, & Whitehouse, 2015)

Darüber hinaus kann der Zugriff einer WebView auf einem Content Provider auf dem Gerät gesperrt werden, indem folgende Einstellung verwendet wird:

`webview.getSettings().setAllowContentAccess(false);` (Chell, Erasmus, Colley, & Whitehouse, 2015)

Soll eine WebView sich zu einer vordefinierten Zusammenstellung von bekannten Seiten verbinden, dann können Überprüfungen erstellt werden, sodass keine andere Seite innerhalb der WebView geladen wird. Das kann mittels überschreiben des WebViewClient's `shouldInterceptRequest`-Methode erfolgen, wie im Listing 5-12 zu sehen ist. (Chell, Erasmus, Colley, & Whitehouse, 2015)

```
@Override
public WebResourceResponse shouldInterceptRequest (final WebView view,
String url){
    Uri uri = Uri.parse(url);
    if (!uri.getHost.equals("www.mysite.com") &&
    !uri.getScheme.equals("https"))
```

```
    {
        return new WebResourceResponse("text/html", "UTF-8",
            new StringBufferInputStream("alert('Not happening')")
    }
    else
    {
        return super.shouldInterceptRequest(view, url);
    }
}
```

Listing 5-12: Überschreiben der WebViewClient's shouldInterceptRequest-Methode (Chell, Erasmus, Colley, & Whitehouse, 2015)

Das vorherige Beispiel lädt Seiten von www.mysite.com, wenn sie über HTTPS geladen werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.4.3 Man-in-the-middle vermeiden

Die meisten Man-in-the-middle-Angriffe sind auf Self-Signed Zertifikate oder auf das Täuschen von Browsern konzentriert. Es kann aber passieren, dass es Vorfälle bei den CAs gibt. Bei einer CA wird von einem hohen Maß an Sicherheit ausgegangen. Im Juni 2011 wurde die CA „DigiNotar“ angegriffen. Dabei wurden über 500 bösartige SSL Zertifikate ausgestellt und von DigiNotar signiert. Da DigiNotar vertraut wurde, befand sich das Root-Zertifikat in allen modernen Browsern. Das bedeutet, dass die Angreiferin oder der Angreifer zulässige SSL-Zertifikate hatte und damit Man-in-the-middle-Angriffe begehen konnte. (Gunasekera, 2012)

Aus diesem Grund sollten zusätzliche Maßnahmen ergriffen werden, um die Sicherheit zu wahren. Mittels OAuth und Challenge/Response können Anmeldeinformationen geschützt werden, auch wenn die Transportkanäle kompromittiert wurden.

Das OAuth-Protokoll ermöglicht Third-Party-Seiten oder -Anwendungen, Enduser-Daten in einer Webanwendung (Service Provider) zu verwenden. Die Endbenutzerin und der Endbenutzer haben dabei die Kontrolle über das Ausmaß des Zugriffs, ohne Anmeldedaten preiszugeben oder zu speichern. OAuth funktioniert mittels Request-Tokens. Webseiten, die Zugriff auf Daten einer Webapplikation benötigen, sind auf einen Token der gleichen Applikation angewiesen, bevor sie den Zugriff auf diese Daten erhalten. (Gunasekera, 2012)

Der zweite Mechanismus um Anmeldeinformationen zu schützen ist die Challenge/Response-Technik. Bei dieser Technik werden wie bei OAuth keine Anmeldeinformationen übertragen. Stattdessen fordert eine Partei die andere Partei heraus. Die andere Partei muss dann eine zufällige Information mittels einem speziell ausgewählten Algorithmus und einer kryptographischen Funktion verschlüsseln. Der Key der zur Verschlüsselung verwendet wird, ist das Passwort. Die verschlüsselten Informationen werden wieder zurückgesendet und von der anderen Partei mit dem Passwort am Ende verschlüsselt. Der Ciphertext wird danach verglichen. Stimmt er überein, erhält die Benutzerin oder der Benutzer den Zugriff. (Gunasekera, 2012)

5.5 Gezieltes Logging

In Android ist das Logging eine große Quelle von ungewollter Informationsfreigabe. Durch die Verwendung von Logging-Methoden, wie zum Beispiel für Debugging-Zwecke, können Anwendungen allgemeine Meldungen zu Anmeldeinformationen oder anderen sensiblen Daten preisgeben. Systemprozesse wie der ActivityManager protokollieren ausführliche Meldungen zum Aktivitäts-Aufruf. Anwendungen mit der Berechtigung READ_LOGS erhalten Zugriff auf diese Protokollmeldungen. Auch wenn die Berechtigung READ_LOGS ab Android 4.1 nicht mehr für Drittanbieteranwendungen verfügbar ist, können ältere Versionen und Root Geräte weiterhin darauf zugreifen. (Drake, et al., 2014)

Das Logging kann während der Entwicklung unerlässlich sein. Den Überblick über alle Logging Funktionen zu behalten, kann für Entwicklerinnen und Entwickler schwierig sein. Anstatt bis zur Anwendungsveröffentlichungen zu warten, um Logs zu überprüfen oder zu deaktivierten, können sie auch mittels einer zentralen Logging Klasse verwaltet werden. Dabei sollte diese Klasse ein Flag enthalten, das eingeschaltet und ausgeschaltet werden kann. Je nachdem, ob das Logging während der Entwicklung aktiviert werden soll, oder ob es für die Produktveröffentlichung deaktiviert sein sollte. Die Logging-Funktion kann auch mit einem Check für `BuildConfig.DEBUG` verknüpft werden. Dieser Ansatz kann anfällig für Fehler sein. Die Verwendung einer benutzerdefinierten Protokollierungsklasse beseitigt alle potenziellen Fehlerpunkte in Bezug auf das Protokollieren von sensiblen Informationen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Ein gutes Beispiel für exzessive Protokollierung wurde im Firefox Browser für Android gefunden. Firefox loggte Browseraktivität, einschließlich der URLs, die besucht wurden. In einigen Fällen waren Session-ID's dabei, wie im Bug-Eintrag von dem Mozilla-Bugtracker gezeigt wurde. Das Listing 5-13 zeigt einen Auszug des Bug-Eintrags. (Drake, et al., 2014)

```
I/GeckoBrowserApp(17773): Favicon successfully loaded for URL =
https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41
DC2C3AB
I/GeckoBrowserApp(17773): Favicon is for current URL =
https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41
DC2C3AB
E/GeckoConsole(17773): [JavaScript Warning: "Error in parsing value for
'background'. Declaration dropped."
{file:https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD
172D41DC2C
3AB?wicket:bookmarkablePage=:com.wm.mobile.web.rx.privacy.PrivacyPractices
line: 0}]
```

Listing 5-13: Auszug des Mozilla-Bugtrackers von einer exzessiven Protokollierung (Drake, et al., 2014)

In diesem Fall könnte eine schädliche Anwendung (mit Protokollzugriff) diese Session-ID sammeln, um die Sitzung des Opfers zu übernehmen. (Drake, et al., 2014)

Bei iOS kann es zu Fällen kommen, dass sensible oder proprietäre Informationen durch Logging veröffentlicht und bis zum nächsten Neustart zwischengespeichert werden. Das Logging einer iOS-Anwendung erfolgt in der Regel mithilfe der NSLog-Methode, die Nachrichten an das Apple System Log (ASL) sendet. Diese Konsolen-Logs können mittels der Xcode Geräteanwendung betrachtet werden. Seit iOS 7 werden von ASL nur Daten der zugehörigen Anwendung returniert. Das soll Angreiferinnen und Angreifern hindern, fremde Logs nach Informationen zu durchsuchen. Bei einem Jailbreak kann es passieren, dass ein Gerät den NSLog auf den Syslog umleitet. Dadurch besteht die Möglichkeit, dass sensible Informationen im Dateisystem im Syslog gespeichert werden. Aus diesem Grund sollten Entwicklerinnen und Entwickler NSLog nicht verwenden, um sensitive oder proprietäre Informationen zu loggen. Die einfachste Möglichkeit um NSLog im Produktionsrelease zu vermeiden, ist ihn mit einem dummy pre-processor macro wie „`#define NSLog (...)`“ neu zu definieren. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.6 Beeinträchtigung von Reverse Engineering

Folgende Maßnahmen für Binary Protections sind kein Ersatz für zusätzliche Methoden zur Anwendungssicherheit. Das Umgehen dieser Maßnahmen wird immer möglich sein, wie zum Beispiel statisch oder währender Laufzeit, durch einen privilegierten User-Context mittels Patchen der Anwendung. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.6.1 Obfuscation

Eine kompilierte Android Anwendung kann leicht zu lesbaren Quellcode dekompiliert werden, der dem Original ähnelt. Um ein Reverse-Engineering komplexer zu gestalten, können Obfuscators eingesetzt werden, sodass der dekompilierte Code weniger lesbar und schwerer zu folgen ist. Abhängig von dem Grad der Verschleierung kann es zu einem erheblichen Zeitaufwand für die Angreiferin oder dem Angreifer kommen. Es gibt viele Tools für Obfuscation, wie zum Beispiel ProGuard, das frei verfügbar ist, sowie weitere kostenpflichtige Optionen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Bei iOS-Anwendungen ist es möglich, Klassen-, Methoden- und Variablennamen einer Mach-O Binary abzurufen. Aus diesem Grund ist es hier wichtig Obfuscators einzusetzen. Es gibt wenige Optionen, um nativen Code zu verschleiern. Eine Möglichkeit ist das Obfuscator-LLVM-Projekt (LLVM - Low Level Virtual Machine). Mithilfe dessen können Android-NDK-Anwendungen (Native Development Kit) sowie iOS-Anwendung mittels LLVM-Compiler-Optimization-Pass verschleiert werden. Um die Obfuscator-LLVM mit Xcode verwenden zu können, muss zunächst ein Xcode-Plugin erstellt werden, das auf den neuen Compiler verweist. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Obfuscator-LLVM ist zwar ein nützlicher Obfuscator, es fehlt jedoch die Funktionalität, dass Klassennamen und Methodennamen verschleiert werden. Es besteht aber die Möglichkeit einen zusätzlichen Obfuscator einzusetzen. Zum Beispiel könnte iOS-Class-Guard als Erweiterung eingesetzt werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.6.2 Root Detection

Anwendungen können die Information benötigen, ob ein Android Gerät gerootet ist oder nicht. Um diese Information zu erhalten gibt es mehrere Möglichkeiten. Die bekannteste Technik ist die Überprüfung ob die `su`-Binary im Path existiert. Das könnte zum Beispiel bewerkstelligt werden, indem „`which su`“ ausgeführt und geparst wird. Wenn Root auf dem Gerät verfügbar ist, wird der vollständige Pfad zu `su` bereitgestellt. Das Which-Tool zählt aber nicht zu den Standard-Binärdateien, deshalb ist es nicht sichergestellt, ob es vorhanden ist. Aufgrund dessen sollte eine Funktion erstellt werden, die wie das Which-Tool funktioniert. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Dazu muss die PATH-Umgebungsvariablen in eigene Verzeichnisse zerlegt werden und die Verzeichnisse müssten nach der Binary durchsucht werden. Obwohl die Suche nach dem `su`-

Binary sicher zulässig ist, reicht es allein nicht aus, um einen Root zu erkennen. Dazu kann folgende zusätzliche Überprüfung durchgeführt werden:

- Auslesen der Datei `default.prop` im Root-Verzeichnis des Android-Dateisystems. Ein Attribut in dieser Datei namens `ro.secure` gibt an, welche Berechtigungen mit einer Android-Debug-Bridge(ADB)-Shell verknüpft sind, wenn die Verbindung von einem Computer erfolgt. Wenn dieser Wert gleich 0 ist, startet ADB mit Root-Rechten und das ist ein Hinweis darauf, dass zur Root-Shell verbunden werden kann, indem `adb shell` verwendet wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)
- Überprüfung ob ein Programm namens `adbd` vom Root-User gestartet wurde. Das ist möglich, indem der Standard `ps`-Binary ausgeführt und der Output geparst wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)
- Überprüfung von bekannten Emulator-Build-Properties mithilfe von der `android.os.Build`-Klasse. Folgende System Properties können gegen den regulären Ausdruck überprüft werden, um zu sehen ob eine Applikation innerhalb eines Emulators gestartet wurde:
  - `Build.TAGS = "test-keys"`
  - `Build.HARDWARE = "goldfish"`
  - `Build.PRODUCT = "generic" oder "sdk"`
  - `Build.FINGERPRINT = "generic.*test-keys"`
  - `Build.DISPLAY = ".*test-keys"`

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

  Die Existenz einer oder mehrere dieser Werte würde anzeigen, dass die Anwendung innerhalb eines Emulators ausgeführt wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)
- Iteration über die Labels der installierten Anwendungen mithilfe der `PackageManager`-Klasse und Überprüfung ob sie Wörter wie „`SuperSU`“, „`Superuser`“ und weitere gängige Anwendungen zur Steuerung des Root-Zugriffs enthalten. Der Vorteil gegenüber der Suche ob sich eine APK-Datei in einem bestimmten Verzeichnis befindet, liegt darin, dass die Möglichkeit besteht die Datei umzubenennen oder in ein anderes Verzeichnis zu verschieben. Die Datei ist dann nicht mehr unter `/system/app/` auffindbar. Des Weiteren können die Package-Namen dieser Anwendungen gesucht werden, wie zum Beispiel nach „`com.noshufou.android.su`“ und „`eu.chainfire.supersu`“. Diese Überprüfung ist aber am wenigsten zuverlässig, da eine Root-Manager-Anwendung auf dem Gerät installiert sein könnte, ohne den tatsächlichem Root-Zugriff. Wenn sich die Root-Manager-APK innerhalb des `/system`-Ordners befindet, zeigt das an, dass das Gerät zu einem bestimmten Zeitpunkt über einen privilegierten Zugriff verfügt hat. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.6.3 Jailbreak Detection

Ähnlich zur Root-Überprüfung in Android ist ein weiterer wichtiger Schutz die Überprüfung, ob ein iOS-Gerät mittels Jailbreak manipuliert wurde. Wird ein Jailbreak festgestellt, können folgende Maßnahmen angewendet werden:

- Besitzerinnen oder Besitzer des Gerätes werden gewarnt und aufgefordert Haftung zu übernehmen
- Verhindern, dass die Anwendung abstürzt
- Löschen aller sensiblen Daten vom Gerät
- Bericht zum Management-Server senden und Besitzerin oder Besitzer als Betrugsrisiko kennzeichnen
- Anwendung beenden

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Nachfolgende Methoden zeigen Möglichkeiten auf, wie ein Jailbreak aufgedeckt werden kann.

Bei einem Jailbreak können Spuren im Dateisystem sichtbar bleiben. Um diese zu finden, kann eine Mischung aus File-Handling-Routinen von den SDK-APIs verwendet werden, wie `NSFileManager fileExistsAtPath` und Standard-POSIX-ähnliche Funktionen wie `stat()`. Die Verwendung von unterschiedlichen Funktionen, um die Existenz von Dateien oder Verzeichnissen zu bestimmen, hilft vor allem dann, wenn Angreiferinnen und Angreifer einen Teil der Funktionen ausführen. Bekannte Pfade eines jailbreak/root sind:

- /bin/bash
- /usr/sbin/sshd
- /Applications/Cydia.app
- /private/var/lib/apt
- /pangueaxe
- /System/Library/LaunchDaemons/io.pangu.axe.untether.plist
- /Library/MobileSubstrate/MobileSubstrate.dylib
- /usr/libexec/sftp-server
- /private/var/stash

  (Chell, Erasmus, Colley, & Whitehouse, 2015)

Um von Reverse Engineering nicht entdeckt zu werden, sollten Verschlüsselungen oder Obfuscation verwendet werden, sodass die Pfade die validiert werden, verschleiert sind. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Viele Anwenderinnen oder Anwender eines Jailbreak-Gerätes verwenden Fernzugriffssoftware. Dass kann dazu führen, dass ein nicht standardmäßiger Port eines Gerätes geöffnet wird. Zum

Beispiel wird OpenSSH verwenden, wozu der TCP-Port 22 benötigt wird. Es kann davon ausgegangen werden, dass wenn SSH oder andere nicht standardmäßige Ports geöffnet sind, das Gerät manipuliert wurde. Als weitere Methode um einen Jailbreak zu erkennen, kann das Gerät nach nicht standardmäßig geöffneten Ports durchsucht werden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Die Sandbox soll eine Anwendung vom OS und anderen Anwendungen schützen. Ein Jailbreak beschädigt die Sandbox, sodass es zum Beispiel möglich ist `fork()` anzuwenden. Bei einem Gerät ohne Jailbreak würde es nicht funktionieren, da Third-Party-Anwendungen keinen neuen Prozess starten dürfen. Bei einem Gerät mit Jailbreak ist dieser Befehl aber ausführbar. Das kann ausgenutzt werden, um ein manipuliertes Gerät zu erkennen. (Chell, Erasmus, Colley, & Whitehouse, 2015) In folgendem Listing 5-14 wird ein einfaches Beispiel gezeigt:

```
inline int checkSandbox() __attribute__((always_inline));
int checkSandbox() {
    int result = fork();
    if (result >= 0) return 1;
    return 0;
}
```

Listing 5-14: Anwendung von fork(), um ein Jailbreak zu erkennen (Chell, Erasmus, Colley, & Whitehouse, 2015)

Es besteht auch die Möglichkeit, eine Datei außerhalb der Sandbox zu erstellen. Bei einem nicht manipulierten Gerät ist das nicht möglich. Der Code vom Listing 5-15 zeigt ein Beispiel für eine Dateierstellung. (Chell, Erasmus, Colley, & Whitehouse, 2015)

```
inline int checkWrites() __attribute__((always_inline));
int checkWrites()
{
    FILE *fp;
    fp = fopen("/private/shouldnotopen.txt", "w");
    if(!fp) return 1;
    else return 0;
}
```

Listing 5-15: Erstellung einer Datei außerhalb der Sandbox, um ein Jailbreak zu erkennen (Chell, Erasmus, Colley, & Whitehouse, 2015)

Auf iOS Geräten ist die `read-only`-Systempartitionen meist kleiner als die `data`-Partition und vorinstallierte Systemanwendungen befindet sich standardmäßig in der Systempartition im `/Applications`-Ordner. Bei einem Jailbreak-Prozess kann dieser Ordner verschoben worden sein, sodass zusätzliche Anwendungen installiert werden können, ohne den limitierten Speicherplatz aufzubrauchen. Dazu wird ein symbolischer Link erstellt, um das `/Applications`-Verzeichnis zu ersetzen und ein neues Verzeichnis wird innerhalb der `data`-Partition angelegt. Wird das Dateisystem auf diese Weise verändert, kann ein Jailbreak nachgewiesen werden, indem `/Applications` auf einem symbolischen Link überprüft wird. (Chell, Erasmus, Colley, &

Whitehouse, 2015) Der Programmcode vom Listing 5-16 zeigt wie ein Pfad, wie zum Beispiel `/Applications`, überprüft werden kann.

```
inline int checkSymLinks (char *path) __attribute__((always_inline));
int checkSymLinks(char *path)
{
    struct stat s;
    if (lstat(path, &s) == 0)
    {
          if (S_ISLNK(s.st_mode) == 1)
          return 1;
    }
    return 0;
}
```

Listing 5-16: Überprüfung von Pfaden, um ein Jailbreak nachzuweisen (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.6.4 Debugger Detection

Debugging ist eine beliebte Technik, um bei mobilen Anwendungen, Reverse Engineering anzuwenden. Es bietet einen Einblick in die interne Funktionsweise einer Anwendung und ermöglicht es einem Angreifer, den Kontrollfluss oder interne Codestrukturen zu verändern. Um einen Code innerhalb einer Anwendung zu manipulieren, wird ein Debugger am Gerät benötigt. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Das ist unter Android möglich, wenn die Anwendung als Debuggable markiert ist. Es besteht aber die Möglichkeit das Manifest der Anwendung zu ändern, um „`debugable = true`“ hinzuzufügen oder ein Laufzeitmanipulationstool einzusetzen, das ein Debuggen ermöglicht. Mit folgendem Code kann überprüft werden, ob bei der Anwendung debuggen aktiviert ist:

```
boolean debuggable = (getApplicationInfo().flags &
ApplicationInfo.FLAG_DEBUGGABLE) != 0;
```
(Chell, Erasmus, Colley, & Whitehouse, 2015)

Eine weitere Maßnahme ist regelmäßig zu prüfen, ob eine Anwendung über einen Debugger verbunden ist. Zum Beispiel durch die Verwendung von der `isDebuggerConnected()`-Methode in der `android.os.Debug`-Klasse. Das ist wieder als Verlangsamung zu sehen, da eine Angreiferin oder ein Angreifer die Überprüfung entfernen kann. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Bei iOS wird das Debugging in der Regel mit dem Systemaufruf `ptrace()` durchgeführt. Diese Funktion kann innerhalb der Anwendung aufgerufen werden und mittels einer weiteren Funktion kann die Verfolgung durch einen Debugger verhindert werden. Wird ein Prozess gerade verfolgt, kann er mit dem `ENOTSUP`-Status beendet werden. Wie bei Android ist es wieder als Verlangsamung zu sehen, da immer die Möglichkeit besteht die Überprüfung zu entfernen. (Chell, Erasmus, Colley, & Whitehouse, 2015) Folgendes Listing 5-17 zeigt diese Implementierung.

Dieser Code sollte nicht nur während der Anwendung gestartet werden, sondern so nah wie möglich beim Prozessstart ausgeführt werden, wie zum Beispiel in der `main`-Funktion:

```
inline void denyPtrace () __attribute__((always_inline));
void denyPtrace()
{
    ptrace_ptr_t ptrace_ptr = dlsym(RTLD_SELF, "ptrace");
    ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);
}
```

Listing 5-17: Debuggererkennung von ptrace() in iOS (Chell, Erasmus, Colley, & Whitehouse, 2015)

Falls der `PT_DENY_ATTACH`-Vorgang überwunden wurde, könnte eine zusätzliche Maßnahme implementiert werden. Zum Beispiel kann die Funktion `sysctl()`verwendet werden, um zu erkennen ob ein Debugger an der Anwendung angehängt wurde oder nicht. Die Funktion hindert nicht das Anhängen eines Debuggers, aber gibt Aufschluss darüber, ob die Anwendung debuggt wird oder nicht. Die Funktion `sysctl()` gibt eine Struktur mit dem Flag `kp_proc.p_flag` zurück. Das zeigt Informationen über den Prozess, wie zum Beispiel ob er debuggt wird. Der Code vom Listing 5-18 zeigt eine Implementierung. (Chell, Erasmus, Colley, & Whitehouse, 2015)

```
inline int checkDebugger () __attribute__((always_inline));
int checkDebugger()
{
    int name[4];
    struct kinfo_proc info;
    size_t info_size = sizeof(info);
    info.kp_proc.p_flag = 0;
    name[0] = CTL_KERN;
    name[1] = KERN_PROC;
    name[2] = KERN_PROC_PID;
    name[3] = getpid();
    if (sysctl(name, 4, &info, &info_size, NULL, 0) == -1) {
    return 1;
}
    return ((info.kp_proc.p_flag & P_TRACED) != 0);
}
```

Listing 5-18: Debuggererkennung mit `sysctl()` in iOS (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.6.5 Tamper Detection

Tamperproofing versucht Manipulationsversuche an einer Software zu erschweren oder zu verhindern. Dabei überprüft eine Integritätsprüfung ob statische Anwendungsressourcen, wie HTML oder Shared Libraries und interne Codestrukturen nicht verändert wurden. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Die Integritätsvalidierung kann mittels einer Checksumme, wie CRC32, implementiert werden. Um statische Anwendungsressourcen wie HTML oder Shared Libraries zu validieren, kann die Checksumme für jede einzelne Ressource oder für alle Ressourcen vereint berechnet werden. Danach wird sie in die Anwendung mit einer Validierungsroutine eingebettet und periodisch während der Anwendungslaufzeit mit der gespeicherten Checksumme verglichen. Eine weitere Möglichkeit ist die Verwendung eines LLVM-Compilers, um den nativen Code in iOS und Android-Anwendungen eine Selbstvalidierung zu ermöglichen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Unter Android besteht die Möglichkeit einen Anwendungsstart zu verhindern, wenn es Anzeichen für die Manipulation der APK-Datei gibt. Das Listing 5-19 zeigt, wie eine Anwendung auf die Änderung der APK überprüft werden kann. Es prüft die Signatur des Signaturzertifikats, das gegen einen bekannten guten Wert geprüft wird. (Chell, Erasmus, Colley, & Whitehouse, 2015)

```
public boolean applicationTampered(Context con)
{
    PackageManager pm = con.getPackageManager();
    try
    {
       PackageInfo myPackageInfo = pm.getPackageInfo(con.getPackageName(),
             PackageManager.GET_SIGNATURES);
       String mySig = myPackageInfo.signatures[0].toCharsString();
       //Compare against known value
       return !mySig.equals("3082...");
    }
    catch (NameNotFoundException e)
    {
       e.printStackTrace();
    }
    return false;
}
```

Listing 5-19: Überprüfung des Signaturzertifikats, um ein unerlaubten Start zu verhindern (Chell, Erasmus, Colley, & Whitehouse, 2015)

Diese Abfrage ist auch kein vollständiger Schutz gegen diesen Angriff. Es besteht die Möglichkeit, dass wenn die Abfrage fehlschlägt, die Anwendung Informationen über das Gerät an die Entwicklerin oder den Entwickler sendet. Das dient als Warnung, dass jemand die Anwendung verändert oder knackt, um sie dann im Internet zu veröffentlichen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

Trotz aller Sicherheitsvorkehrungen wird es immer möglich sein, diese Maßnahmen zu umgehen, da sie sich innerhalb der Binärdatei befinden. Die Angreiferin oder der Angreifer hat somit die Möglichkeit mittels eines Patches alle Routinen zu entfernen. (Chell, Erasmus, Colley, & Whitehouse, 2015)

5.7 Zusammenfassung

Dieses Kapitel widmete sich den unterschiedlichen Verteidigungsmöglichkeiten. Dabei wurde gezeigt, wie serverseitigen Steuerelemente gesichert werden können und wie eine sichere Aufbewahrung von Daten erfolgen kann. Des Weiteren wurde eine sichere Kommunikation, sowie ein gezieltes Logging behandelt. Um Daten sicher zu übertragen sollte auf eine sichere Kommunikation mit einem Server geachtet werden. Zum Schutz der Anwendung und des eigenen Programmcodes sollten Maßnahmen gegen Reverse Engineering getroffen werden. Das nachfolgende Kapitel beschäftigt sich mit unterschiedlichen Testfällen der theoretischen Ausarbeitung.

6 PRAKTISCHER TEIL

Die vorangegangenen Kapitel haben gezeigt, wie eine Datenübertragung aussehen kann, welche Angriffsvektoren es gibt und wie eine Verteidigung erfolgen kann. Mithilfe dieser Informationen werden sechs Testfälle ausgearbeitet, sowie die Vor- und Nachteile der jeweiligen Implementierung gezeigt. Dieses Kapitel kombiniert das Wissen der theoretischen Ausarbeitung und der Testfälle, um Security Frameworks mit Designregeln zu entwickeln.

6.1 Testgeräte

Für die Teststellung werden zwei Testgeräte mit mobilen Betriebssystem Android verwendet.

6.1.1 Testgerät 1

Als Testgerät 1 kommt ein OnePlus3 zum Einsatz, welches für die Durchführung der Tests in keiner Form manipuliert oder verändert wurde. In folgender Tabelle 6-1 werden die Spezifikation des Testgeräts 1 angeführt:

Modellnummer:	ONEPLUS A3003
Betriebssystem:	OxygenOS 4.0.3
Android-Version:	7.0
Build-Nummer:	ONEPLUS A3003_16_170208

Tabelle 6-1: Spezifikation von Testgerät 1

Auf dem Gerät ist die Android-Version 7.0 vom August 2016 installiert (Pakalski, 2017). Die einzelnen Spezifikationen wurden über „Einstellungen → Über das Telefon“ direkt am Smartphone abgerufen. Die folgende Abbildung 6-1 veranschaulicht ein Bild, welches für Mock-ups verwendet wird:

Abbildung 6-1: OnePlus 3 (OnePlus, 2017)

6.1.2 Testgerät 2

Bei diesem Gerät handelt es sich um ein LG Optimus 2X. Für Testzwecke wurde diese Smartphone gerootet und eine CyanogenMod-Version für das Modell installiert. Die folgende Tabelle 6-2 veranschaulicht die technischen Eigenschaften des zweiten Testgeräts:

Modellnummer:	LG-P990
Betriebssystem:	CyanogenMod 11-20140918-UNOFFICIAL-p990
Android-Version:	4.4.4
Build-Nummer:	cm_p990-userdebug 4.4.4 KTU84Q 1a006b233c test-keys

Tabelle 6-2: Spezifikation von Testgerät 2

Auf dem Gerät befindet sich die Android-Version 4.4.4 vom Dezember 2014 (Whitwam, 2017). Die Gerätedetails wurden über „Settings → About phone“ direkt am Smartphone abgerufen.

Abbildung 6-2: LG Optimus 2X (LGEPR, 2017)

Die Grafik, welche in Abbildung 6-2 ersichtlich ist, wird in weiterer Folge für Mock-ups verwendet.

6.2 Fall 1: Übertragung mit unterschiedlichen Übertragungsarten

Der Fall 1 widmet sich der Datenübertragung mittels REST und Soap. Eine Datenübertragung kann für viele Szenarios benötigt werden, wie zum Beispiel für einen Log-in oder zur Bereitstellung von Information. Dabei wird ein Client, der die Daten sendet und ein Server der diese Daten empfängt (oder umgekehrt), benötigt.

Für das folgende Beispiel werden die Webservices unter der Verwendung von Hypertext Preprocessor (PHP) und dem Datenbankverwaltungssysteme MySQL erstellt. In der Datenbank wird eine Tabelle mit den Namen `persons` erstellt, die folgende Spalten enthält:

- **`id`** `int(11) NOT NULL AUTO_INCREMENT`
- **`first_name`** `varchar(250) DEFAULT NULL`
- **`last_name`** `varchar(255) DEFAULT NULL`
- **`created_at`** `timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP`

Der REST-Service wird mit der Unterstützung von Slim (Lockhart, Smith, Allen, & SlimFrameworkTeam, 2017) erzeugt und enthält folgende Requests:

- `GET /persons` → Gibt alle Datensätze der Tabelle `persons` zurück
- `POST /create` → Die Werte von `first_name` und `last_name` werden in einem neuen Datensatz eingefügt.

Für den Soap-Service wird eine Web Services Description Language (WSDL)-Datei erstellt, die folgende Funktionen enthält:

- `selectPerson` → Gibt einen Datensatz anhand der `id` oder alle Datensätze von `persons` zurück.
- `createPerson` → Erstellt einen Eintrag in `persons`, mit den Werten von `first_name` und `last_name`

Für den Client wird eine Anwendung mit zwei Eingabefeldern, zwei Buttons und zwei Textfeldern erstellt. Als Testgerät wird das Testgerät 1 herangezogen. Für dieses Szenario werden folgende Klassen definiert:

- `JsonParser` → Parst ein JSON-Object, sodass es als `String` „`first_name + " " + last_name`“ ausgegeben werden kann.
- `Get` → Fragt alle Personen mittels http://host/persons ab und parst diese Informationen mit der `JsonParser`-Klasse. Danach wird jeder einzelne Datensatz in die `List<String>` geschrieben und als komplette Liste retourniert. Das Listing 6-1 zeigt die Implementierung dieser Funktionen.

```
String line, newjson = "";
URL urls = new URL("http://host/persons");
try (BufferedReader reader = new BufferedReader(new
```

```
InputStreamReader(urls.openStream(), "UTF-8"))) {
    while ((line = reader.readLine()) != null) {
        newjson += line;
    }
    String json = newjson.toString();
    JSONObject jsonObject = new JSONObject(json);
    List<String> persons = JsonParser.parseJson(jsonObject);

    return persons;
```

Listing 6-1: GET-Methode und Parsen von REST

- `Post` → Das Listing 6-2 zeigt die Erstellung eines Datensatzes mit `first_name` und `last_name` über http://host/create. Falls keine Werte übergeben werden, wird ein Datensatz mit `NULL` angelegt.

```
String urlString = params[0];
String firstName = params[1];
String lastName = params[2];
URL url = null;
InputStream stream = null;
HttpURLConnection urlConnection = null;
try {
    url = new URL(urlString);
    urlConnection = (HttpURLConnection) url.openConnection();
    urlConnection.setRequestMethod("POST");
    urlConnection.setDoOutput(true);

    String data = URLEncoder.encode("first_name", "UTF-8") + "=" +
URLEncoder.encode(firstName, "UTF-8");
    data += "&" + URLEncoder.encode("last_name", "UTF-8") + "=" +
URLEncoder.encode(lastName, "UTF-8");
    urlConnection.connect();

    OutputStreamWriter wr = new
OutputStreamWriter(urlConnection.getOutputStream());
    wr.write(data);
    wr.flush();
    stream = urlConnection.getInputStream();
    BufferedReader reader = new BufferedReader(new
InputStreamReader(stream, "UTF-8"), 8);
    String result = reader.readLine();

    return result;
}
```

Listing 6-2: POST-Methode von REST

- `SoapParser` → Parst ein `Vector<SoapObject>` zu einer `List<String>` wo ein `String`, wie bei `JsonParser`, wie folgt aussieht: „`first_name + " " + last_name`“
- `Soap` → Im Listing 6-3 wird zuerst ein SoapObject mit `first_name` und `last_name` erstellt und danach an den Soap-Server http://host/SoapSrv.php mittels `HttpTransportSE` gesendet. Nach dem Senden werden alle Personen von der Datenbank ausgelesen, mit dem `SoapParser` geparst und als `List<String>` retourniert.

```
SoapObject create = new SoapObject(NAMESPACE, METHOD_NAME_createPerson);
SoapSerializationEnvelope envelopeCreate = new
SoapSerializationEnvelope(SoapEnvelope.VER11);
create.addProperty("first_name", params[0]);
create.addProperty("last_name", params[1]);
envelopeCreate.setOutputSoapObject(create);
HttpTransportSE httpTransport = new HttpTransportSE(URL);
try {
    httpTransport.call(SOAP_ACTION_createPerson, envelopeCreate);
} catch (IOException e) {
    e.printStackTrace();
} catch (XmlPullParserException e) {
    e.printStackTrace();
}

List<String> result = new ArrayList<String>();
SoapObject soapObject = new SoapObject(NAMESPACE,
METHOD_NAME_selectPerson);
PropertyInfo propertyInfo = new PropertyInfo();
propertyInfo.setName(PARAMETER_NAME);
propertyInfo.setType(String.class);
soapObject.addProperty(propertyInfo);
SoapSerializationEnvelope envelope = new
SoapSerializationEnvelope(SoapEnvelope.VER11);
envelope.setOutputSoapObject(soapObject);
envelope.env = "http://schemas.xmlsoap.org/soap/envelope/";
HttpTransportSE httpTransportSE = new HttpTransportSE(URL);
try {
    httpTransportSE.call(SOAP_ACTION_selectPerson, envelope);
    Vector<SoapObject> response = (Vector<SoapObject>)
envelope.getResponse();
    result = SoapParser.parseSoap(response);
    return result;
} catch (Exception e) {
    e.printStackTrace();
}
return result;
```

Listing 6-3: SOAP-Methode

Die Klassen `Get`, `Post` und `Soap` werden mit der `AsyncTask`-Klasse erweitert, da Netzwerkzugriffe aufgrund von möglichen Wartezeiten, nicht während der Laufzeit ausgeführt werden sollen.

Für Android gibt es keine Soap-Lösung, deshalb wurde das ksoap2-Projekt (simpligility technologies inc., 2017) mittels build.gradle importiert:

```
buildTypes {
        repositories {
            maven { url
'https://oss.sonatype.org/content/repositories/ksoap2-android-releases' }
        }
    }
dependencies {
    compile 'com.google.code.ksoap2-android:ksoap2-android:3.6.2'
}
```

Listing 6-4: Importierung des ksoap2-Projekt in der build.gradle-Datei

Im Listing 6-4 wird erkennbar, dass unter den `buildTypes` die zugehörige `maven`-URL importiert und unter `dependencies` das ksoap2-Projekt hinzugefügt wird.

In der `MainActivity`-Klasse wird ein `setOnClickListener` für die Buttons „REST SPEICHERN“ und „SOAP SPEICHERN“ erstellt. Beim Klick des Buttons „REST SPEICHERN“ werden die Inhalte der Eingabefelder zum REST-Service übertragen.

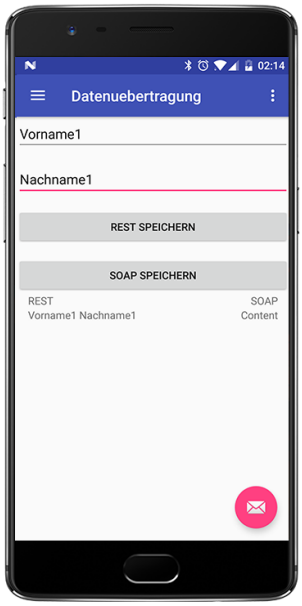


Abbildung 6-3: Ausführung von REST

Wie in der Abbildung 6-3 zu erkennen ist, werden nach dem Senden im Textfeld von REST, alle Personen vom Server geladen. Da es sich um den ersten Eintrag handelt, wird nur ein Datensatz angezeigt. Wird der Button „SOAP SPEICHERN“ betätigt, werden die Werte der Eingabefelder mittels Soap versendet.

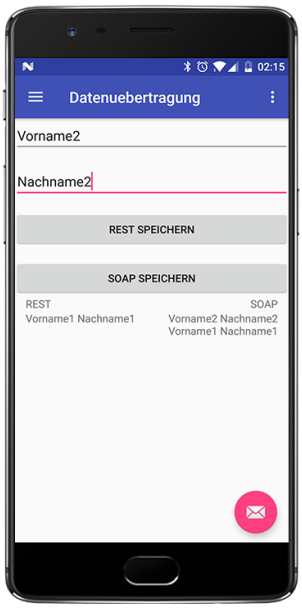


Abbildung 6-4: Ausführung von SOAP

Wie auch bei REST, werden nach dem Senden alle Personen vom Server heruntergeladen. Die Abbildung 6-4 zeigt, wie zwei Einträge ausgegeben werden, da der mittels REST gespeichert Datensatz, auch angezeigt wird.

6.3 Fall 2: Daten übertragen und verlieren

Der erste Fall hat gezeigt wie eine Übertragung von Daten funktionieren kann. Der zweite Fall setzt sich mit dem unsicheren Datenverkehr auseinander und zeigt, wie TSL die Datenübertragung beeinflusst. Für dieses Szenario wird die Anwendung vom ersten Fall übernommen, analysiert und bearbeitet.

Zu Beginn wird die Anwendung von Fall 1 in ein neues Projekt migriert. Vor dem Ausführen der neuen Anwendung wird Wireshark gestartet, um den Web-Traffic zu überwachen (Wireshark Foundation, 2017). Nach dem Start der Anwendung wird der Button „REST SPEICHERN“ betätigt und die Ausgabe in Wireshark überprüft:

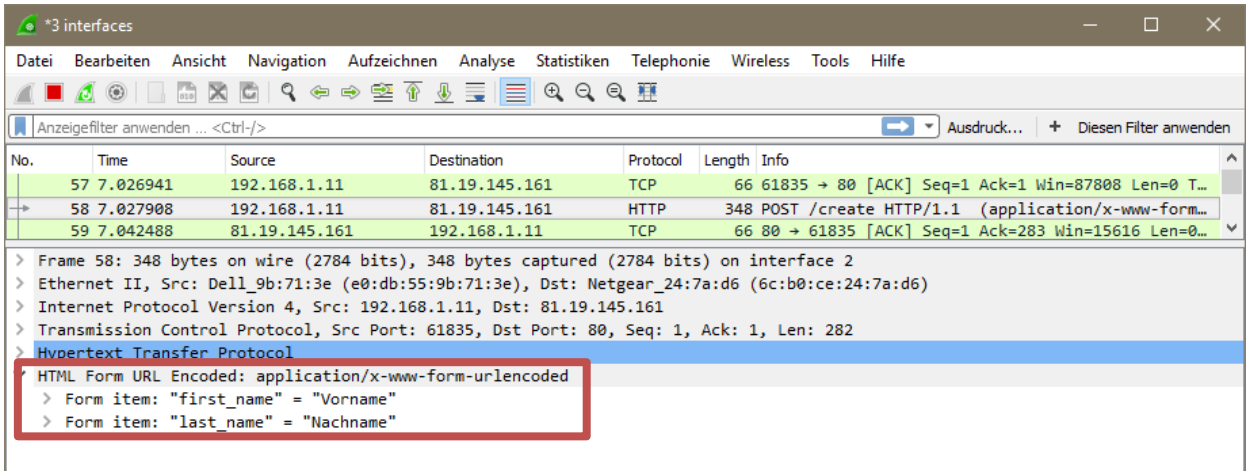


Abbildung 6-5: Screenshot Wireshark von REST POST (Wireshark Foundation, 2017)

Wie in Abbildung 6-5 zu erkennen ist, können die Werte der Anfrage `/create` mitgesnifft werden. Es ist zu sehen, dass für „first_name“ der Wert „Vorname“ und für „last_name“ der Wert „Nachname“ übergeben wurde. Des Weiteren werden alle Personen von `/persons` als JSON-Object sichtbar:

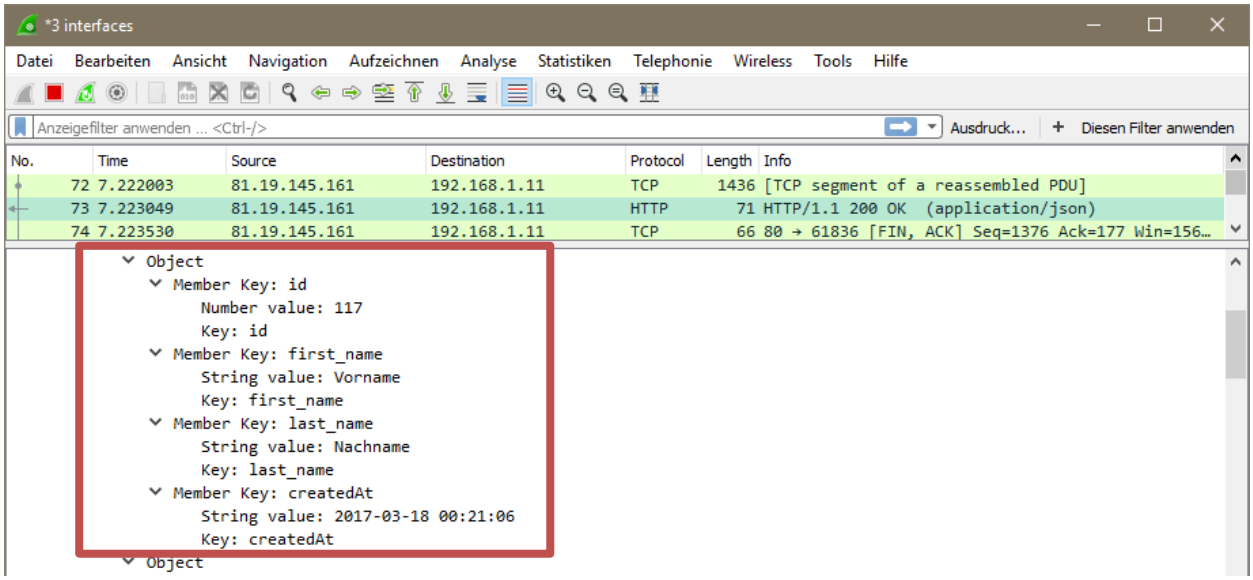


Abbildung 6-6: Screenshot Wireshark von REST GET (Wireshark Foundation, 2017)

Die Abbildung 6-6 zeigt alle Werte einer Person an, auch Informationen, wie die `id` und `createdAt`, die in der Anwendung nicht ersichtlich sind. Nach dieser Auswertung wird der Button „SOAP SPEICHERN“ betätigt. Wie auch bei der Ausführung mit REST, wird der Web-Traffic mitgesnifft:

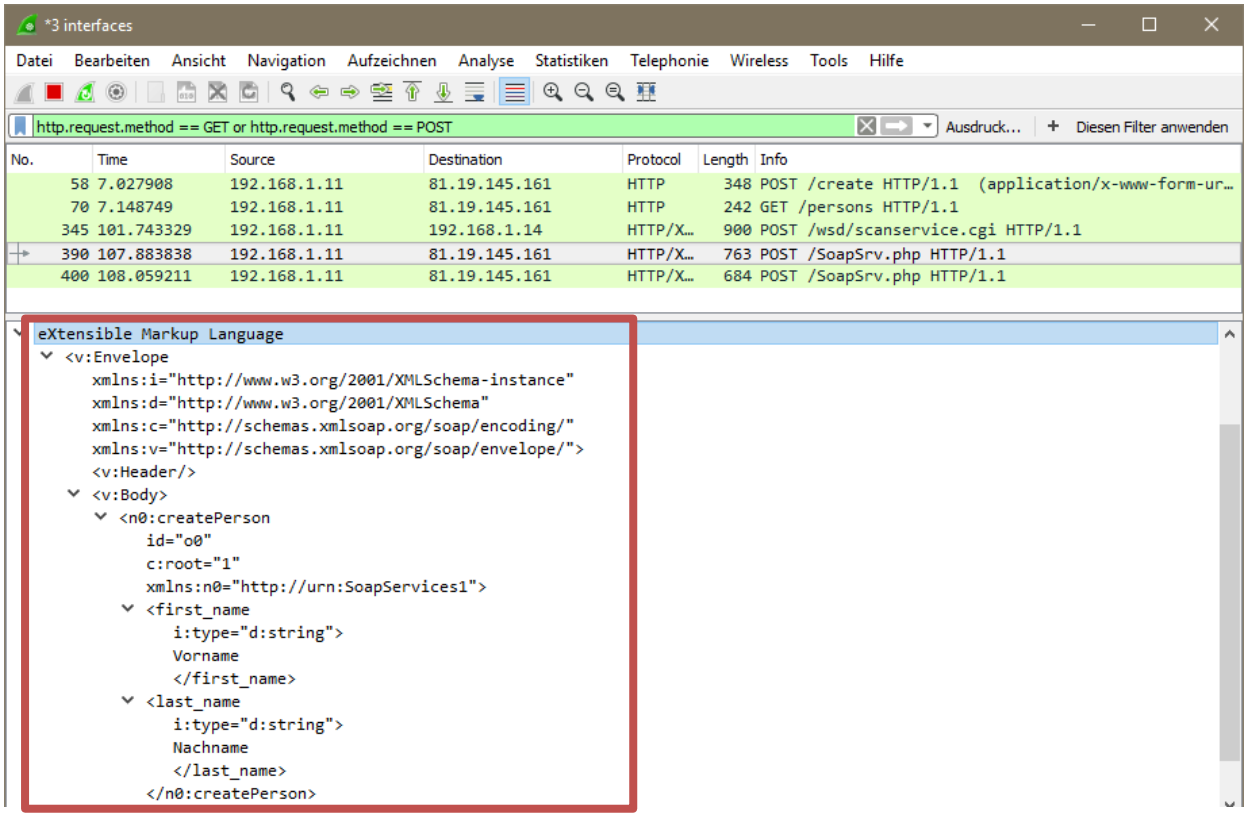


Abbildung 6-7: Screenshot Wireshark von SOAP POST (Wireshark Foundation, 2017)

Die Abbildung 6-7 zeigt den `Envelope` von `createPerson` mit den übergebenen Parametern „first_name“ mit „Vorname“ und „last_name“ mit „Nachname“. Nach dem Senden der Werte an den Server, werden alle Personen mittels Soap abgefragt:

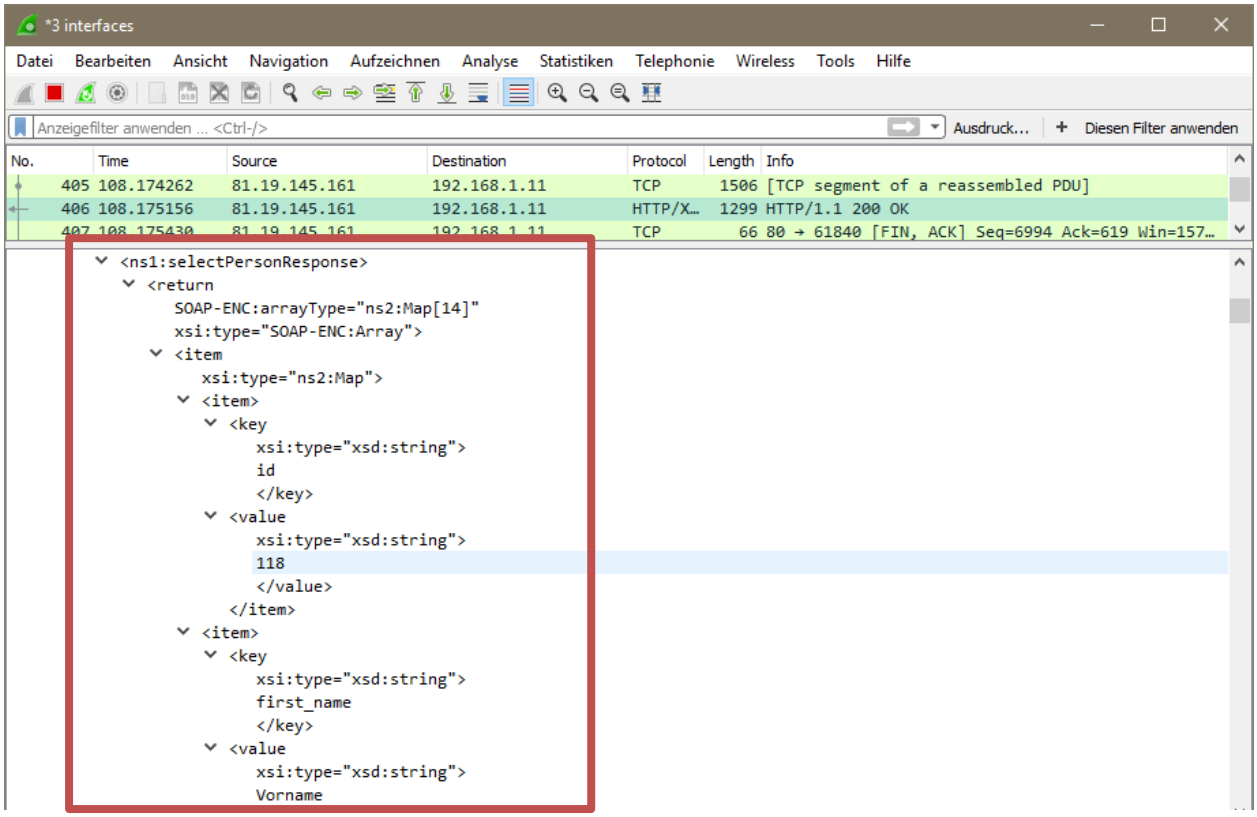


Abbildung 6-8: Screenshot Wireshark von SOAP GET (Wireshark Foundation, 2017)

Die Abbildung 6-8 zeigt, dass wie auch bei REST, alle Informationen zu einer Person angezeigt werden. In diesem Szenario wird ersichtlich, dass nicht nur sensible Daten gesichert werden müssen. Es ist schon bei der Entwicklung darauf zu achten, dass nur Informationen bereitgestellt werden, die notwendig sind.

Im nachfolgendem Szenario wird am Server TLS aktiviert und der Web-Traffic wird wieder mit Wireshark mitgesnifft. Des Weiteren werden folgende Anpassungen in der Anwendung benötigt, sodass eine Übertragung durch TLS möglich ist:

- Die URL muss von http://host nach https://host geändert werden
- In der `Post`-Klasse muss `(HttpURLConnection)` auf `(HttpsURLConnection)` geändert werden
- Da bei Android ab dem 20. API Level TLS standardmäßig aktiviert ist, sind keine weiteren Einstellungen notwendig

Nach dem Start der modifizierten Anwendung, wird wieder der Button „REST SPEICHERN“ ausgeführt. Dabei wird mit Wireshark mitgesnifft, ob Veränderungen im Output existieren. Im Web-Traffic konnte POST oder GET nicht mehr gefunden werden, deshalb werden nur mehr TCP und TLSv1.0 Zeilen angezeigt. Die Datenübertragung läuft nun verschlüsselt ab. Die Zeile für die Übertragung ist schwer zu finden, da sie nicht im Klartext aufscheint und keine Referenzwerte bestehen.

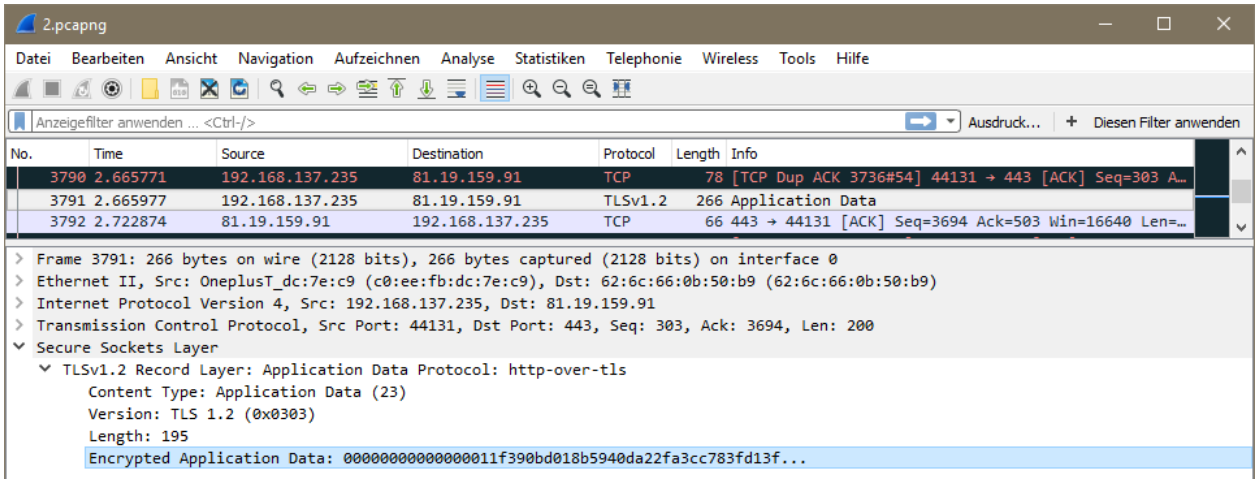


Abbildung 6-9: Screenshot Wireshark vom verschlüsselten REST /create (Wireshark Foundation, 2017)

Die Abbildung 6-9 zeigt die verschlüsselten Werten von REST /create. Es sind keine Texte oder Informationen erkennbar.

Bei der Abbildung 6-10 werden die Personen von /persons returniert. Auch hier sind keine Informationen erkennbar.

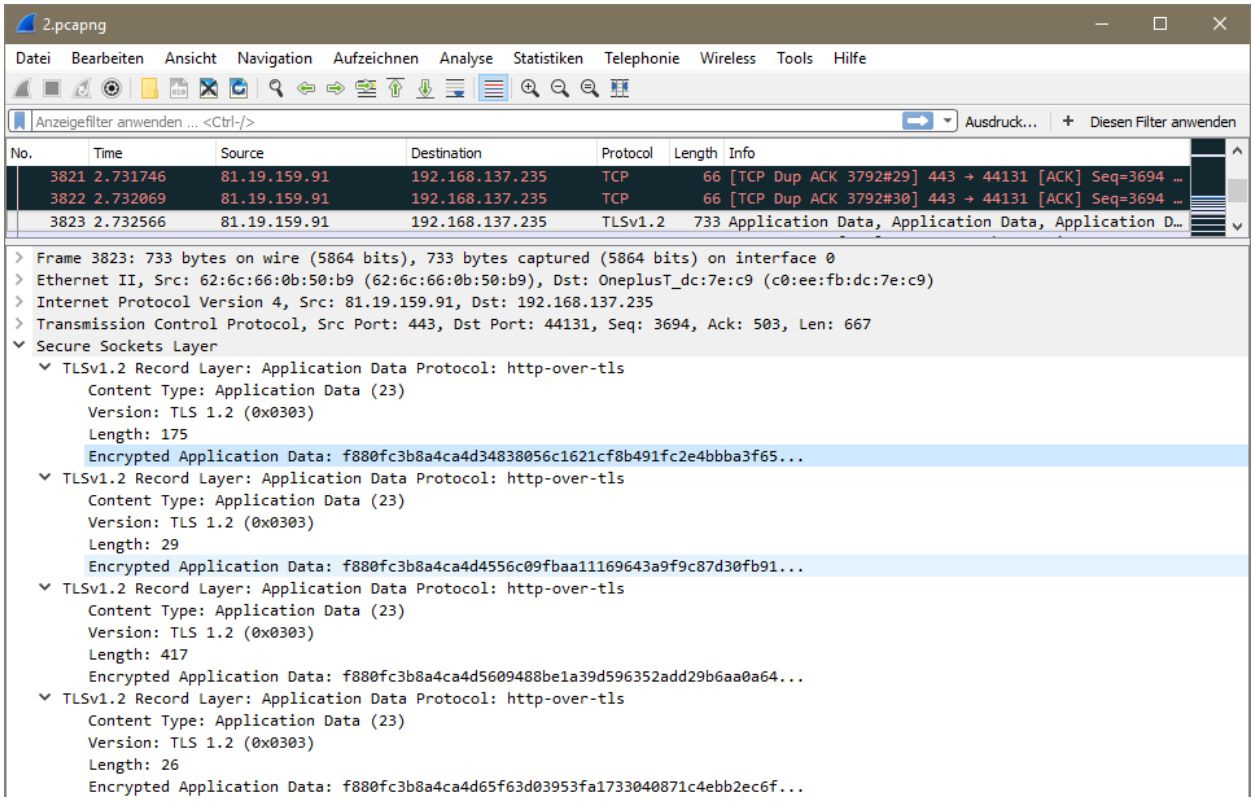


Abbildung 6-10: Screenshot Wireshark vom verschlüsselten REST /persons (Wireshark Foundation, 2017)

Bei der Übertragung von SOAP ist das Ergebnis gleich wie bei REST. Es gibt keine Anzeichen eines `Envelopes` oder ähnliches, es wird nur die Zeile „Encrypted Application Data“ angezeigt.

Mit TLS können Daten sehr schnell und ohne großen Aufwand verschlüsselt werden und durch die Unterstützung von Android ist dies sehr einfach zu implementieren.

6.4 Fall 3: Daten speichern und stehlen

Daten von Benutzerinnen und Benutzern können in unterschiedlichen Formen gespeichert werden. Dabei ist zu beachten, dass andere Anwendungen nur auf diese Daten Zugriff erhalten, die auch benötigt werden. Alle anderen sollten in einer Form gespeichert werden, sodass ein unbefugter Zugriff nicht möglich ist.

Im dritten Fall werden Daten in Form einer Textdatei(TXT)-Datei gespeichert. Für das vorliegende Szenario werden zwei Dateien an unterschiedlichen Orten und mit unterschiedlichen Berechtigungen gespeichert. Danach wird versucht, die Dateien von einer eigenen Anwendung und einer „fremden“ Anwendung auszulesen. Als Testgerät wird das Testgerät 1 verwendet. Die erste Datei „Test1.txt“ wird im internen Speicher der Anwendung gespeichert und hält die Berechtigung `Context.MODE_PRIVATE`. Das Listing 6-5 zeigt den Code der für diese Speicherung verwendet wird:

```
public void saveToInternalStorage(Context context) {
    String fileName1 = "Test1.txt";
    String content = "save to internal storage";
    FileOutputStream outputStream = null;
    try {
        outputStream = context.openFileOutput(fileName1,
Context.MODE_PRIVATE);
        outputStream.write(content.getBytes());
        outputStream.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Listing 6-5: Speicherung einer Datei im internen Speicher einer Anwendung

Der Inhalt der Datei lautet: "save to internal storage". Wenn die Berechtigungen für read oder write bestehen, kann dieser Text ausgelesen werden. Mittels des Programmcodes welcher in Listing 6-6 ersichtlich ist, wird die zweite Datei „Test2.txt“ in den externen Speicher mit den Standardberechtigungen (read/write) gespeichert:

```
public void saveToExternalStorage() {
    String fileName2 = "Test2.txt";
    File exFile;
    String content = "save to external storage";
    FileOutputStream outputStream;
    try {
        exFile = new
File(Environment.getExternalStorageDirectory().getAbsolutePath(),
fileName2);
        outputStream = new FileOutputStream(exFile);
```

```
            outputStream.write(content.getBytes());
            outputStream.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Listing 6-6: Speicherung einer Datei im externen Speicher einer Anwendung mit Standardberechtigungen

Sind die notwendigen Berechtigungen vorhanden, wird der Inhalt „save to external storage“ ausgelesen. Die zwei Testdateien werden mit dem Code ausgelesen der im Listing 6-7 ersichtlich ist.

```
public String getFileFromStorage(Context context, String path, String
fileName) {
    try {
        File file = new File(path, fileName);
        BufferedReader input = new BufferedReader(new
InputStreamReader(new FileInputStream(file)));
        String line;
        StringBuffer buffer = new StringBuffer();
        while ((line = input.readLine()) != null) {
            buffer.append(line);
        }
        return buffer.toString();
    } catch (FileNotFoundException e) {
        Log.e("Save", "getFileFromStorage: ", e);
        return "";
    } catch (UnsupportedEncodingException e) {
        Log.e("Save", "getFileFromStorage: ", e);
        return "";
    } catch (IOException e) {
        Log.e("Save", "getFileFromStorage: ", e);
        return "";
    }
}
```

Listing 6-7: Auslesen von Dateien nach Namen und Speicherort

Ist das Auslesen einer Datei nicht möglich, wird eine Exception im Log angezeigt. Wie im Listing 6-8 ersichtlich, wird für den ersten Durchlauf die Dateien mit der eigenen Anwendung erstellt und ausgelesen.

```
Save save = new Save();
save.saveToInternalStorage(getApplicationContext());
Log.d("MainActivity", "onCreatet: " +
save.getFileFromStorage(getApplicationContext(),
"/data/user/0/com.exampletest.test.datenspeichern/files/", "Test1.txt"));

save.saveToExternalStorage();
Log.d("MainActivity", "onCreate: " +
save.getFileFromStorage(getApplicationContext(),
Environment.getExternalStorageDirectory().getAbsolutePath(),
"Test2.txt"));
```

Listing 6-8: Speicherung und auslesen im internen Speicher mittels eigener Anwendung

Im Log wird angezeigt, ob das Erstellen und Auslesen der Dateien möglich ist:

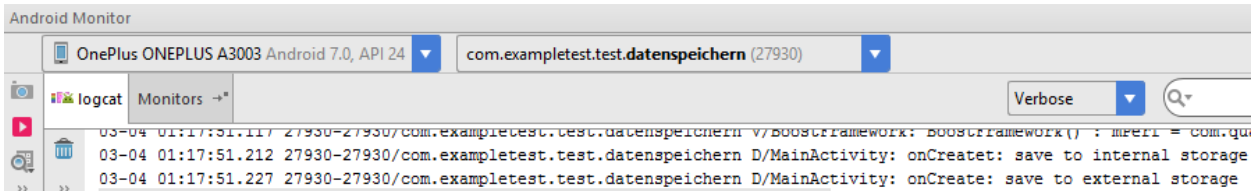


Abbildung 6-11: Daten erfolgreich speichern und im Log auslesen

Wie in Abbildung 6-11 gezeigt wird, konnten beide Dateien erfolgreich gespeichert und ausgelesen werden. Für die zweite Abfrage wird ein neues Projekt mit einer anderen Company Domain erstellt. Wie schon bei der ersten Applikation wird `getFileFromStorage` für das Auslesen verwendet. In Listing 6-9 wird die Implementation veranschaulicht.

```
Read read = new Read();
Log.d("MainActivity", "onCreate: " +
read.getFileFromStorage(getApplicationContext(),
"/data/user/0/com.exampletest.test.datenspeichern/files/","Test1.txt"));

Log.d("MainActivity", "onCreate: " +
read.getFileFromStorage(getApplicationContext(),
Environment.getExternalStorageDirectory().getAbsolutePath(),
"Test2.txt"));
```

Listing 6-9: Speicherung und auslesen im internen Speicher mittels externer Anwendung

Des Weiteren wird versucht, die angelegten Dateien auszulesen:

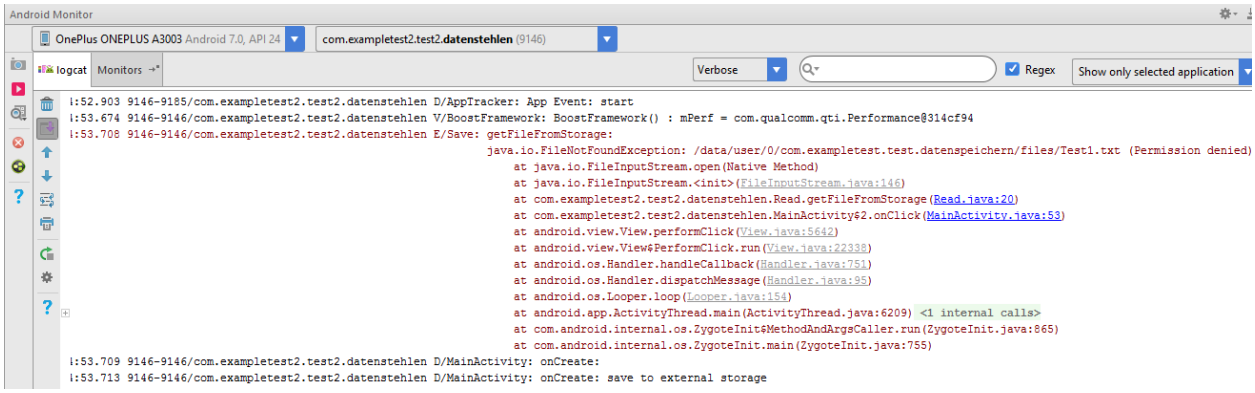


Abbildung 6-12: Keine Zugriffberechtigung auf „Test1.txt"

Die Abbildung 6-12 zeigt, dass die Datei „Test1.txt“ aufgrund von „Permission denied“ nicht ausgelesen werden konnte. Da die Anwendung mit einer anderen Company Domain gestartet wurde, hat sie keinen Zugriff auf diese Datei. Bei der Datei „Test2.txt“ konnte der Inhalt ohne Probleme ausgelesen werden.

6.5 Fall 4: Eingaben überprüfen versus jede Eingabe zulassen

Eingaben von Benutzerinnen und Benutzer können in unterschiedlichen Bereichen benötigt werden. Jedoch stellt jede übermittelte Eingabe einen Angriffspunkt für Injection-Angriffe dar. Deshalb widmet sich der Fall 4 der SQL-Injection und wie diese verhindert werden kann.

Für diesen Fall werden konkret zwei Tests durchgeführt. Der erste Test zeigt ein direktes `INSERT`-Statement. Bei dem zweiten Test wird das `INSERT`-Statement parametrisiert. Für diese Tests werden die Klassen `Person` und `DatabaseHandler` erstellt. Wie im Listing 6-10 ersichtlich, stellt `person` eine Person mit einer eindeutigen `_id`, einen `_first_name` und einen `_last_name` dar. Des Weiteren werden für die Klasse `Constructors`, `Getters` und `Setters` erstellt.

```
int _id;
String _first_name;
String _last_name;
```

Listing 6-10: Tabelle person mit Datentypen

Die Klasse `DatabaseHandler` ist für dir Datenbankabwicklung verantwortlich. Das Listing 6-11 zeigt den Code der Klasse.

```
private static final int DATABASE_VERSION = 1;
private static final String DATABASE_NAME = "personsDB";
private static final String TABLE_PERSONS = "persons";
private static final String KEY_ID = "id";
private static final String KEY_FIRST_NAME = "first_name";
private static final String KEY_LAST_NAME = "last_name";

public DatabaseHandler(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
    String CREATE_PERSONS_TABLE = "CREATE TABLE " + TABLE_PERSONS + "("
            + KEY_ID + " INTEGER PRIMARY KEY," + KEY_FIRST_NAME + " TEXT,"
            + KEY_LAST_NAME + " TEXT" + ")";
    db.execSQL(CREATE_PERSONS_TABLE);
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_PERSONS);

    onCreate(db);
}
```

```
public void addPerson1(Person person) {
    SQLiteDatabase db = this.getWritableDatabase();
    String sql = "INSERT INTO " + TABLE_PERSONS + "(" + KEY_FIRST_NAME +
"," + KEY_LAST_NAME +
            ") VALUES ('" + person.getFirstName() + "','" +
person.getLastName() + "')";
    db.beginTransaction();
    SQLiteStatement statement = db.compileStatement(sql);
    try {
        statement.execute();
        db.setTransactionSuccessful();
    } finally {
        db.endTransaction();
    }
    db.close();
}

public void addPerson2(Person person) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(KEY_FIRST_NAME, person.getFirstName());
    values.put(KEY_LAST_NAME, person.getLastName());

    db.insert(TABLE_PERSONS, null, values);
    db.close();
}

public List<Person> getAllPersons() {
    List<Person> personList = new ArrayList<>();
    String selectQuery = "SELECT  * FROM " + TABLE_PERSONS;

    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery, null);

    if (cursor.moveToFirst()) {
        do {
            Person person = new Person();
            person.setID(Integer.parseInt(cursor.getString(0)));
            person.setFirstName(cursor.getString(1));
            person.setLastName(cursor.getString(2));
            personList.add(person);
        } while (cursor.moveToNext());
    }
    return personList;
}
```

Listing 6-11: DatabaseHandler-Klasse

Die im Code ersichtliche Methode `addPerson1` wird für den Test 1 herangezogen, zudem die Methode `addPerson2` für den Test 2 verantwortlich ist. In `addPerson1` wird ein Insert-Statement direkt mit den Werten befüllt und ausgeführt. Bei `addPerson2` wird jeder Wert individuell behandelt. Die Methode `getAllPersons` liest alle Personen in der Datenbank aus. Sie dient als Kontrolle, welche Werte gespeichert wurden. In der `MainActivity`-Klasse werden zwei Textfelder und zwei Buttons eingefügt. Bei einem Klick des ersten Buttons wird der Test 1, wie im Listing 6-12 dargestellt, ausgeführt.

```
db.addPerson1(new Person(firstNameEdit.getText().toString(),
lastNameEdit.getText().toString()));

List<Person> persons = db.getAllPersons();
for (Person ps : persons) {
    String log = "Id: " + ps.getID() + " , First Name: " +
ps.getFirstName() + " , Last Name: " + ps.getLastName();
    Log.d("Name: ", log);
}
```

Listing 6-12: Hinzufügen einer Person indem Werte direkt befüllt werden

Zuerst wird versucht, die Informationen der Textfelder in die Datenbank zu speichern. Danach erfolgt die Überprüfung ob das Speichern erfolgreich abgeschlossen wurde. Bei einem Klick des zweiten Buttons wird ähnlich zu Test 1, der zweite Test ausgeführt.

```
db.addPerson2(new Person(firstNameEdit.getText().toString(),
lastNameEdit.getText().toString()));

List<Person> persons = db.getAllPersons();
for (Person ps : persons) {
    String log = "Id: " + ps.getID() + " , First Name: " +
ps.getFirstName() + " , Last Name: " + ps.getLastName();
    Log.d("Name: ", log);
}
```

Listing 6-13: Hinzufügen einer Person indem Werte parametrisiert wurden

Nach dem Einfügen wird überprüft ob der Datensatz korrekt eingetragen wurde.

Bei Abbildung 6-13 wird der ersten Versuch Werte in die Datenbank zu schreiben gezeigt, indem gültige Strings verwendet werden.

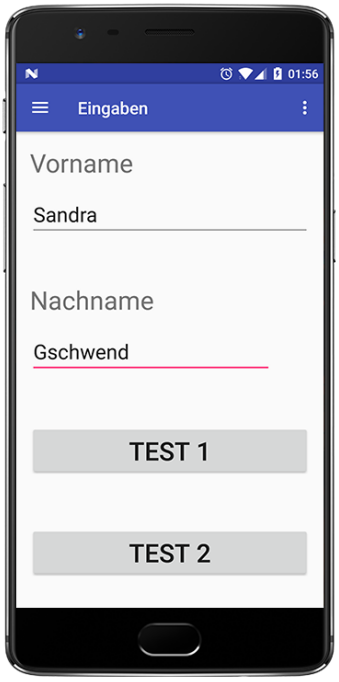


Abbildung 6-13: Eingaben mit gültigen Strings

Der erste Schritt ist die Ausführung des Tests 1. In der Abbildung 6-14 wird der dazugehörige Log Eintrag gezeigt:

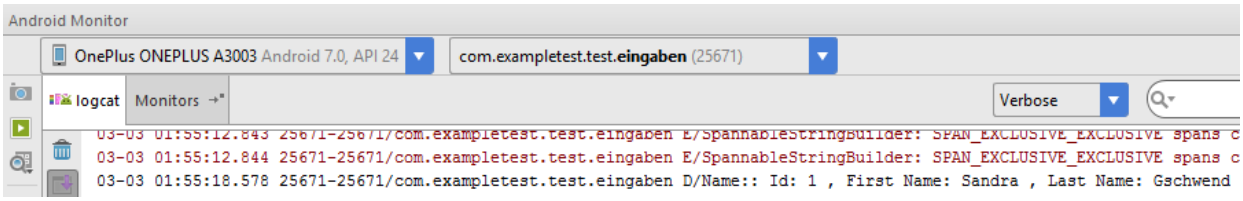


Abbildung 6-14: Logs von Test 1 der gültigen Eingaben

In den Logs wird sichtbar, dass ein Eintrag mit den angegebenen Werten erstellt wurde. Mit den gleichen Werten wird Test 2 ausgeführt, welcher die folgenden Logeinträge in Abbildung 6-15 ausgibt:

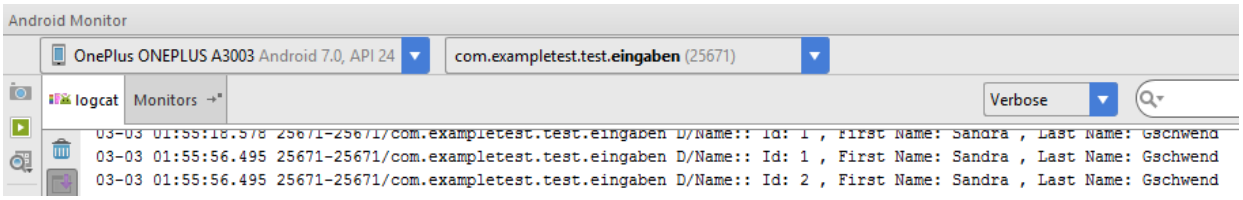


Abbildung 6-15: Logs von Test 2 der gültigen Eingaben

Nach diesem Test wurde mit den gleichen Werten ein Eintrag in der Datenbank erstellt. Zusätzlich wird in der Ausgabe auch der Eintrag von Test 1 angezeigt.

Die Abbildung 6-16 zeigt den zweiten Versuch, bei dem ein direkter Zugriff auf die Datenbank erzwungen wird. Dies wird erreicht, indem im Feld „Vorname“ das Statement zu Ende geführt wird.

Abbildung 6-16: Eingaben mit Injection

Wird das SQL-Statement als ein solches erkannt, erhält die Angreiferin oder der Angreifer direkten Zugriff auf die Datenbank. Als Vorname würde „test1“ stehen und als Nachname „test2“. Wird es nicht als SQL-Statement erkannt, wird der sichtbare Text als Vorname und Nachname in die Datenbank geschrieben. Für diesen Eintrag wird wieder Test 1 und Test 2 durchgeführt:

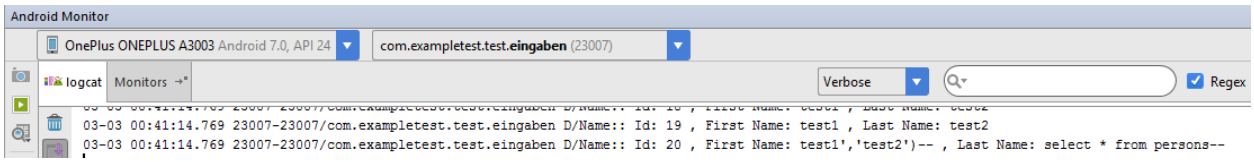


Abbildung 6-17: Log von Injection

In Abbildung 6-17 ist zu erkennen, dass bei einem direkten Statement, eine Injection möglich ist. Test 1 trägt für den Vornamen „test1“ und für den Nachnamen „test2“ ein. Test 2 befüllt die Datenbank mit dem kompletten Statement. Es besteht keine Möglichkeit bei Test 2 aus einem `INSERT` auszubrechen.

6.6 Fall 5: Logging eingrenzen

Logging kann für die Entwicklung von Anwendungen sehr nützlich sein. Vor der Veröffentlichung einer Anwendung sollten, aber keine sensiblen Informationen durch Logging preisgegeben werden. Fall 5 beschäftigt sich mit Logging und wie es vor der Veröffentlichung entfernt werden kann.

Für die Ausarbeitung werden zwei Tests durchgeführt. Bei Test 1 wird eine Klasse `Log` erstellt, die als Buffer zwischen der originalen Log-Klasse und der Ausgabe, anzusehen ist. Test 2 veranschaulicht wie mit ProGuard Logs entfernt werden können.

Für Test 1 wird eine neue Klasse namens `Log` benötigt. Dabei werden die vorhandenen Methoden von `android.util.Log` verwendet und mittels einem Flag abgefragt. Das Listing 3-1 zeigt einen Ausschnitt der Implementierung.

```
public class Log {
    static final boolean LOG = false;

    public static void v(String tag, String string) {
        if (LOG) android.util.Log.v(tag, string);
    }

    public static void d(String tag, String string) {
        if (LOG) android.util.Log.d(tag, string);
    }

    public static void i(String tag, String string) {
        if (LOG) android.util.Log.i(tag, string);
    }

    public static void w(String tag, String string) {
        if (LOG) android.util.Log.w(tag, string);
    }

    public static void e(String tag, String string) {
        if (LOG) android.util.Log.e(tag, string);
    }

    public static void wtf(String tag, String string) {
        if (LOG) android.util.Log.wtf(tag, string);
    }
}
```

Listing 6-14: Neue Klasse von Log

Während der Entwicklung kann das Flag `LOG` auf `true` gesetzt werden, sodass alle Logs angezeigt werden. Wird das Flag auf `false` gesetzt wird kein Log mehr angezeigt, ausgenommen

`android.util.Log` wird statt der eigenen Klasse verwendet. Für diesen Test wurden kritische Informationen abgefragt, um die Wichtigkeit von richtigem Logging hervorzuheben. Das Listing 6-15 zeigt die Methoden die für die Testimplementierung erstellt wurden.

```
public static String getDeviceId(Context context) {
    final TelephonyManager telephonyManager = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
    String deviceId = telephonyManager.getDeviceId();
    return deviceId;
}

public static String getSerialNr(Context context) {
    final TelephonyManager telephonyManager = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
    String serialNr = telephonyManager.getSimSerialNumber();
    return serialNr;
}

public static String getandroidId(Context context) {
    String androidId =
android.provider.Settings.Secure.getString(context.getContentResolver(),
android.provider.Settings.Secure.ANDROID_ID);
    return androidId;
}

public static String getPhoneNr(Context context) {
    final TelephonyManager telephonyManager = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
    String phoneNr = telephonyManager.getLine1Number();
    return phoneNr;
}

public static String getModel(Context context) {
    String model = Build.MODEL;
    return model;
}

public static String getManufacturer(Context context) {
    String manufacturer = Build.MANUFACTURER;
    return manufacturer;
}
```

Listing 6-15: Beispiele für schlechtes Logging

Bei `getDeviceId` handelt sich um eine eindeutige Identifikation des Gerätes. Mit `getandroidId` kann der Unique Identifier (UID) des Gerätes angezeigt werden. Falls eine Subscriber Identity Module-Karte (SIM-Karte) vorhanden ist, lesen die Methoden `getPhoneNr` die Telefonnummer

und `getSerialNr` die Seriennummer aus. Mit `getModel` wird das Model ausgelesen und mit `getManufacturer` kann der Hersteller eruiert werden. Diese Methoden benötigen die Berechtigung `READ_PHONE_STATE`, die wie folgt im AndroidManifest.xml hinterlegt wird:

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
```

Listing 6-16: Berechtigung READ_PHONE_STATE setzen

Die Implementierung vom Logging in der `MainActivity`-Klasse in der `onCreate`-Methode wird im Listing 6-17 gezeigt.

```
Log.v("MainActivity", "onClick 1: " + getDeviceId);
Log.d("MainActivity", "onClick 2: " + getSerialNr);
Log.i("MainActivity", "onClick 3: " + getandroidId);
Log.w("MainActivity", "onClick 4: " + getPhoneNr);
Log.e("MainActivity", "onClick 5: " + getModel);
Log.wtf("MainActivity", "onClick 6: " + getManufacturer);
```

Listing 6-17: Impementation der Logging-Beispiele

Wie im Listing 6-17 zu erkenn ist, werden unterschiedliche Log-Methoden, wie zum Beispiel `Log.d()` für DEBUG-Log-Nachrichten oder `Log.wtf()` für „What a Terrible Failure“-Log-Nachrichten, verwendet.

Für diesen Test wurde das Testgerät 1 verwendet. Zuerst wird das Flag auf `true` gesetzt:

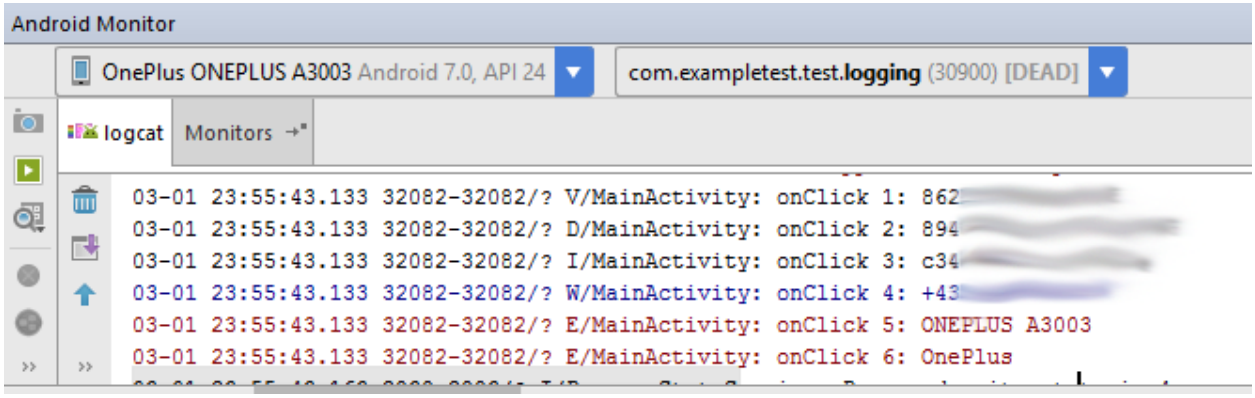


Abbildung 6-18: Logging ist aktiviert

In der Abbildung 6-18 ist ersichtlich, dass alle Informationen für jede Person sichtbar angezeigt werden. Die Logs werden nach einem Release mittels einer signierten APK weiterhin angezeigt. Der Test wird erneut mit dem Flag, welches auf `false` gesetzt ist, ausgeführt. Im Log-File befinden sich keine Entwickler-Logs der Anwendung.

Für Test 2 muss ProGuard in der Anwendung aktiviert werden. ProGuard ist ein Obfuscator und löscht nicht verwendeten Code vor der Veröffentlichung. Da ProGuard mit dem Android Studio geliefert wird, ist nur eine Änderung der `build.gradle`-Datei und der `proguard-rules.pro`-Datei nötigt. Dabei muss in der `build.gradle`-Datei `buildTypes` → `release` → `minifyEnabled`

auf `true` gesetzt werden und in `buildTypes` → `release` → `proguardFiles getDefaultProguardFile` auf `('proguard-android-optimize.txt')` geändert werden. Im Listing 6-18 werden die Änderung der `build.gradle`-Datei gezeigt.

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
    }
}
```

Listing 6-18: Aktivierung von ProGuard in der build.gradle-Datei

In der `proguard-rules.pro`-Datei müssen Zeilen vom Listing 6-19 hinzugefügt werden.

```
-assumenosideeffects class android.util.Log {
    public static int v(...);
   public static int d(...);
    public static int i(...);
    public static int w(...);
    public static int e(...);
   public static int wtf(...);
}
```

Listing 6-19: Hinzufügen der Logging-Methoden in der proguard-rules.pro-Datei

Mit diesen Zeilen werden die angegeben Methoden der Klasse `android.util.Log` entfernt. Diese Änderung wird nach dem Erstellen einer signierten APK sichtbar. Im Debug-Modus werden weiterhin Logging-Informationen angezeigt.

6.7 Fall 6: Reverse Engineering

Fall 6 beschäftigt sich mit Reverse Engineering. Speziell wird dabei der Bereich Root-Zugriff behandelt und wie Programmcode aus einer signierten Anwendung exportiert werden kann.

6.7.1 Root Detection

Die Überprüfung für die Root Detection teilt sich in zwei Tests auf. Test 1 zeigt eine Root Detection mittels `adb shell`, sobald das Testgerät direkt mit einem Computer verbunden ist. Bei Test 2 wird mittels einer Applikation überprüft, ob Root-Rechte bestehen.

Der Ablauf für Test 1 sieht wie folgt aus:

1. Auslesen der `default.prop` mittels `adb -d shell getprop` im Terminal. Ist der Wert von `ro.secure` gleich 0 startet ADB mit Root-Rechten. Das kann ein Hinweis auf ein gerootetes Gerät sein.
2. Starten der Shell mittels `adb shell`
    a. Gestartete Prozesse mittels `ps` aufrufen. Wurde der Prozess `adbd` als Root gestartet, kann das ein Hinweis auf Root-Rechte sein.
    b. Root-Rechte mittels `su` erhalten. Ist dieser Befehl möglich, ist das Gerät gerootet. Bestehen keine Berechtigungen oder wird dieser Befehl nicht gefunden, kann nach ihm im Ordner `system/xbin/` oder `system/bin/` gesucht werden. Befindet sich `su` innerhalb der Ordner und kann ausgeführt werden, bestehen Root-Rechte.

Hinweis: Bei einem gerootetem Gerät kann der Root-Zugriff mittels ADB in den Einstellungen gesperrt sein. Diese Berechtigung muss vor dem Test in den Superuser-Einstellungen aktiviert werden.

Ausführung von Test 1 mit dem Testgerät 1:

1. `[ro.secure]:[1]`
   Das Ergebnis von default.prop zeigt den Wert 1. Es bestehen keine Anzeichen von Root-Rechten.
2. `adb shell` konnte ohne Probleme gestartet werden
    a. Das Listing 6-20 zeigt, dass der Prozess `adbd` nicht als Root gestartet wurde.

```
USERPID   PPID  VSIZE RSS   WCHAN   PC              NAME
Shell     6113  1     15732 1120    0000000000 R    /sbin/adbd
```

Listing 6-20: Ausführung von `adb shell` bei dem Testgerät 1

    b. `/system/bin/sh: su: not found`

    `su` ist nicht ausführbar und auch nicht in den Ordner `system/xbin/` oder `system/bin/` vorhanden.

Auf dem Testgerät 1 wurden keine Hinweise für Root-Rechte gefunden.

Ausführung von Test 1 mit dem Testgerät 2:

Bevor der Test 1 ausgeführt wird, müssen die Einstellungen der Root-Rechte für ADB kontrolliert werden. Diese befinden sich bei Testgerät 2 unter: Settings → Superuser → 3-Punkt-Menü → Settings → Superuser Access → Apps and ADB

1. `[ro.secure]:[1]`
   Das Ergebnis von default.prop zeigt den Wert 1. Es bestehen keine Anzeichen das Root-Rechten vorhanden sind.
2. `adb shell` konnte ohne Probleme gestartet werden
   a. Der Prozess `adbd` wurde nicht als Root gestartet.

```
USERPID   PPID  VSIZE RSS   WCHAN    PC                  NAME
shell     109   1     5684  264      ffffffff 00000000 S  /sbin/adbd
```

Listing 6-21: Ausführung von `adb shell` bei dem Testgerät 2

   b. `su: permission denied`

      Es existieren zwar keine Berechtigungen `su` vorerst auszuführen, `su` befindet sich aber im Ordner `system/xbin/`. Direkt in diesem Ordner kann `su` ausgeführt werden.

Der Test 1 wird erneut ausgeführt. Die Ergebnisse bleiben gleich, außer dass bei Punkt 2.b der `su`-Befehl durch den einmaligen Start ohne Fehlermeldung ausführbar ist. Das Testgerät wird neu gestartet und der Test 1 wird erneut ausgeführt. Wieder mit dem Ergebnis, dass Root-Rechte vorhanden sind. Auf dem Testgerät 2 konnten Root-Rechte nachgewiesen werden.

Test 2 besteht aus der Klasse `RootDetection` und einer Abfrage innerhalb der `onCreate`-Funktion in der `MainActivity`-Klasse. Folgend wird die `RootDetection`-Klasse gezeigt. Wie im Listing 6-22 ersichtlich, wurden für diese Klasse neun Methoden erstellt.

```
public class RootDetection {
    public static boolean isDeviceRooted() {
        return checkWhichSu() || checkTag() || checkHardware() ||
checkProductGeneric()  || checkProductSdk() || checkFingerprint() ||
                checkDisplay() || checkPath();
    }

    private static boolean checkWhichSu() {
        Process process = null;
        try {
            process = Runtime.getRuntime().exec(new String[] {
"/system/xbin/which", "su" });
            BufferedReader in = new BufferedReader(new
```

```
InputStreamReader(process.getInputStream()));
            if (in.readLine() != null) return true;
            return false;
        } catch (Throwable t) {
            return false;
        } finally {
            if (process != null) process.destroy();
        }
    }

    private static boolean checkTag() {
        String buildTags = Build.TAGS;
        return buildTags != null && buildTags.contains("test-keys");
    }

    private static boolean checkHardware() {
        String buildHardware = Build.HARDWARE;
        return buildHardware != null &&
buildHardware.contains("goldfish");
    }

    private static boolean checkProductGeneric() {
        String buildProductGeneric = Build.PRODUCT;
        return buildProductGeneric != null &&
buildProductGeneric.contains("generic");
    }

    private static boolean checkProductSdk() {
        String buildProductSdk = Build.PRODUCT;
        return buildProductSdk != null && buildProductSdk.contains("sdk");
    }

    private static boolean checkFingerprint() {
        String buildFingerprint = Build.FINGERPRINT;
        return buildFingerprint != null &&
buildFingerprint.contains("generic.*test-keys");
    }

    private static boolean checkDisplay() {
        String buildDisplay = Build.DISPLAY;
        return buildDisplay != null && buildDisplay.contains(".*test-
keys");
    }

    private static boolean checkPath() {
        String[] paths =  "/sbin/su", "/system/bin/su", "/system/xbin/su",
"/data/local/xbin/su", "/data/local/bin/su", "/system/sd/xbin/su",
"/system/bin/failsafe/su", "/data/local/su", "/su/bin/su",
```

```
"/system/app/Superuser.apk"};
        for (String path : paths) {
            if (new File(path).exists()) return true;
        }
        return false;
    }
}
```

Listing 6-22: RootDetection-Klasse

Jede Methode der Klasse `RootDetection` gibt einen `boolean`-Wert zurück. Die Methode `isDeviceRooted` überprüft, ob eine von den folgenden Methoden `true` ist. Diese Methode kann aufgrund des Zugriffsmodifizierers `public` global aufgerufen werden. Die weiteren Methoden dieser Klasse haben den Zugriffsmodifizierer `private` und können nicht außerhalb der Klasse aufgerufen werden. Die Methode `checkWhichSu` überprüft, ob „`which su`“ ausführbar ist. Mit den Methoden `checkTag`, `checkHardware`, `checkProductGeneric`, `checkProductSdk`, `checkFingerprint` und `checkDisplay`, werden Systemproperties gegen einen regulären Ausdruck getestet. Mit `checkPath` wird nach bekannte `su`-Pfaden gesucht.

Das Listing 6-23 zeigt, wie für diesen Test die Überprüfung in die `onCreate`-Methode der `MainActivity`-Klasse eingefügt wird. Ist einer dieser Überprüfungen positiv, wird der Text „Not Root!“ mit „Root!“ überschrieben. Im folgendem Codebeispiel wird die Abfrage angezeigt:

```
if (RootDetection.isDeviceRooted()) {
    TextView text = (TextView) findViewById(R.id.root);
    text.setText("Root!");
}
```

Listing 6-23: Ausführung von der RootDetection-Klasse in der MainActivity-Klasse

Deployment des Test 2 auf dem Testgerät 1:

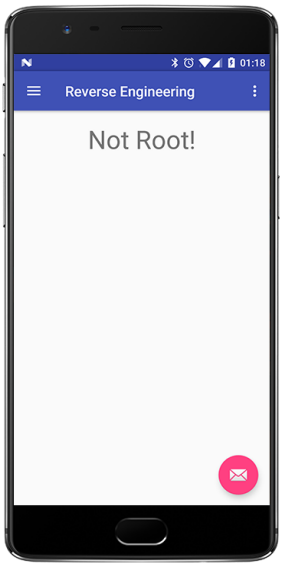


Abbildung 6-19: Root Detection Test 2 mit Testgerät 1

Die Abbildung 6-19 zeig, dass der Text „Not Root!“ nicht überschrieben wurde. Die Anwendung konnte keine Hinweise auf Root-Rechte finden.

Deployment des Test 2 auf dem Testgerät 2:

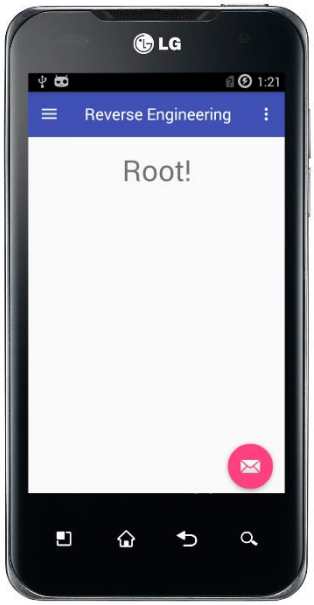


Abbildung 6-20: Root Detection Test 2 mit Testgerät 2

Der Text „Not Root!“, wie in Abbildung 6-20 ersichtlich, wurde überschrieben. Eine oder mehrere Methoden haben Hinweise auf Root-Rechte gefunden.

6.7.2 Debugger Detection

Für die Debugger Detection werden zwei Tests implementiert. Bei Test 1 wird überprüft ob die Anwendung als `debug` deployed wurde. Der zweite Test überprüft ob die Anwendung beim Starten debuggt wird. Ist eine Überprüfung positiv, wird der Text „No Debugger“ mit „Debugger!“ überschrieben. Für die Teststellung wird das Testgerät 1 verwendet.

Das Listing 6-24 zeigt die Abfrage für den Test 1.

```
boolean debuggable = (getApplicationInfo().flags &
ApplicationInfo.FLAG_DEBUGGABLE) != 0;
if (debuggable) {
     TextView text = (TextView) findViewById(R.id.debugger1);
     text.setText("Debug");
}
```

Listing 6-24: Abfrage ob die Anwendung mit FLAG_DEBUGGABLE gestartet wurde

Zuerst wird die Build-Variante `debug` verwendet und die Anwendung deployed.

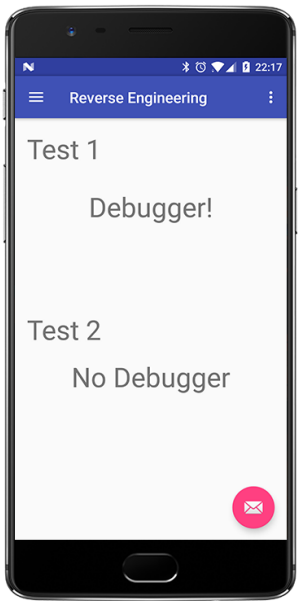


Abbildung 6-21: Debugger Detection Test 1 und Build-Variante debug

In Abbildung 6-21 wird gezeigt, dass es sich nicht um ein Release handelt. Der Test wird weitergeführt, indem die Anwendung veröffentlicht wird. Dazu ist das Signieren der APK und die Build-Variante `release` notwendig. Folgendes Bild zeigt die Anwendung nach dem Start der veröffentlichten APK:

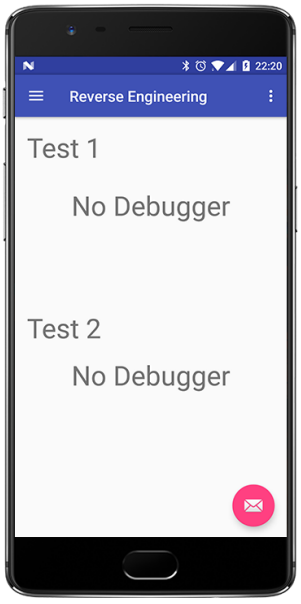


Abbildung 6-22: Debugger Detection Test 1 und Build-Variante Release

Wie in der Abbildung 6-22 ersichtlich ist, zeigt der Test zeigt keine Hinweise eines Debuggers.

Die Abfrage für Test 2 wird im Listing 6-25 veranschaulicht.

```
if (Debug.isDebuggerConnected()) {
    TextView text = (TextView) findViewById(R.id.debugger2);
    text.setText("Debugger!");
}
```

Listing 6-25: Abfrage ob eine Anwendung im Debug-Modus ausgeführt wird

Die Anwendung wird im Debug-Modus gestartet:

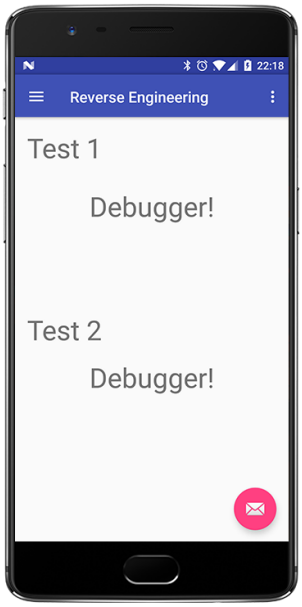


Abbildung 6-23: Debugger Detection Test 2 im Debug-Modus

Abbildung 6-23 veranschaulicht die Erkennung des Tests, dass die Anwendung im Debug-Modus gestartet wurde.

6.7.3 Tamper Detection und Obfuscation

Der Test für Tamper Detection und Obfuscation läuft wie folgt ab:

1. Implementierung und Ausführung der Tamper Detection
2. Generierung und Umbenennung der signierten APK
3. Anwendung mit ProGuard verschleiern
4. Generierung und Umbenennung der signierten APK
5. Anwendung von Reverse Engineering auf die APK-Dateien und diese vergleichen
6. Erstellung eines neuen Projektes und Code einfügen
7. Generierung der signiertes APK mit neuem Key-Store

Zuerst wird die Tamper Detection entwickelt. Dazu wird die Klasse `TamperDetection` erstellt und die Methode vom Listing 6-26 implementiert.

```
public static boolean applicationTampered(Context con)
{
    PackageManager pm = con.getPackageManager();
    try
    {
        PackageInfo myPackageInfo =
pm.getPackageInfo(con.getPackageName(), PackageManager.GET_SIGNATURES);
        String mySig = myPackageInfo.signatures[0].toCharsString();
    Log.d("WICHTIG: wieder entfernen! Signatur: ",mySig);
    // Wert aufgrund der Länge gekürzt
        return !mySig.equals("308203……1a74eb");
    }
    catch (PackageManager.NameNotFoundException e)
    {
        e.printStackTrace();
    }
    return false;
}
```

Listing 6-26: Vergleich der Signatur mit einem bekannten Wert

Um den Signaturwert für die Applikation zu erhalten, wird für `mySig` ein Log-Eintrag erstellt. Wenn die signierte APK ausgeführt wird, muss der Eintrag ausgewertet, die Signatur eingetragen und die Anwendung erneut signiert werden. Der Log-Eintrag sollte vor der Veröffentlichung entfernt werden.

In der Klasse `MainActivity` wird die Abfrage laut Listing 6-27 implementiert.

```
if (TamperDetection.applicationTampered(context)) {
    TextView text = (TextView) findViewById(R.id.tamper);
    text.setText("Tamper!");
}
```

Listing 6-27: Ausführung der Tapmper Detection in der MainActivity-Klasse

Stimmen die Werte der Signatur nicht mehr überein, wird der Text „No Tamper!“ mit „Tamper!“ überschrieben. Es wird eine signierte APK generiert und auf „app-release-ohne.apk“ umbenannt. Die Abbildung 6-24 zeigt die Ausführung der Anwendung nach der Implementierung.

Abbildung 6-24: Tamper Detection hat keine Änderung erkannt

Im dritten Schritt gilt es die Anwendung mit ProGuard zu verschleiern. ProGuard ist ein Bestandteil des Android Studios und kann mit einer Änderung in der `build.gradle`-Datei aktiviert werden. Dabei muss der Parameter `minifyEnabled` in `buildTypes` → `release` auf `true` gesetzt werden. Im Listing 6-28 wird ein Ausschnitt der `build.gradle`-Datei dargestellt:

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
'proguard-rules.pro'
    }
}
```

Listing 6-28: Aktivierung von ProGuard in der build.gradle-Datei

Nach dieser Änderung wird eine signierte APK generiert und auf „app-release-mit.apk“ umbenannt.

Für das Reverse Engineering werden zwei Anwendungen benötigt. Die erste Anwendung „ex2jar“ verwandelt APK-Dateien in Java Archive (JAR)-Dateien um (dex2jar, 2017). Die zweite Anwendung „Java Decompiler“ ermöglicht das Anzeigen dieser JAR-Dateien (Java Decompiler, 2017). Um die APK-Datei in eine JAR-Datei umzuwandeln, muss der Befehl vom Listing 6-29 im Verzeichnis von ex2jar im Terminal (unter Windows) ausgeführt werden:

```
d2j-dex2jar.bat "C:\Speicherort\app-release-ohne.apk"
d2j-dex2jar.bat "C:\Speicherort\app-release-mit.apk"
```

Listing 6-29: Ausführung von dex2jar, um die APK-Datei in eine JAR-Datei umzuwandeln (dex2jar, 2017)

Die umgewandelten Dateien werden mit dem Java Decompiler geöffnet. Folgender Screenshot zeigt die „app-release-ohne.apk“ nach dem Reverse Engineering:

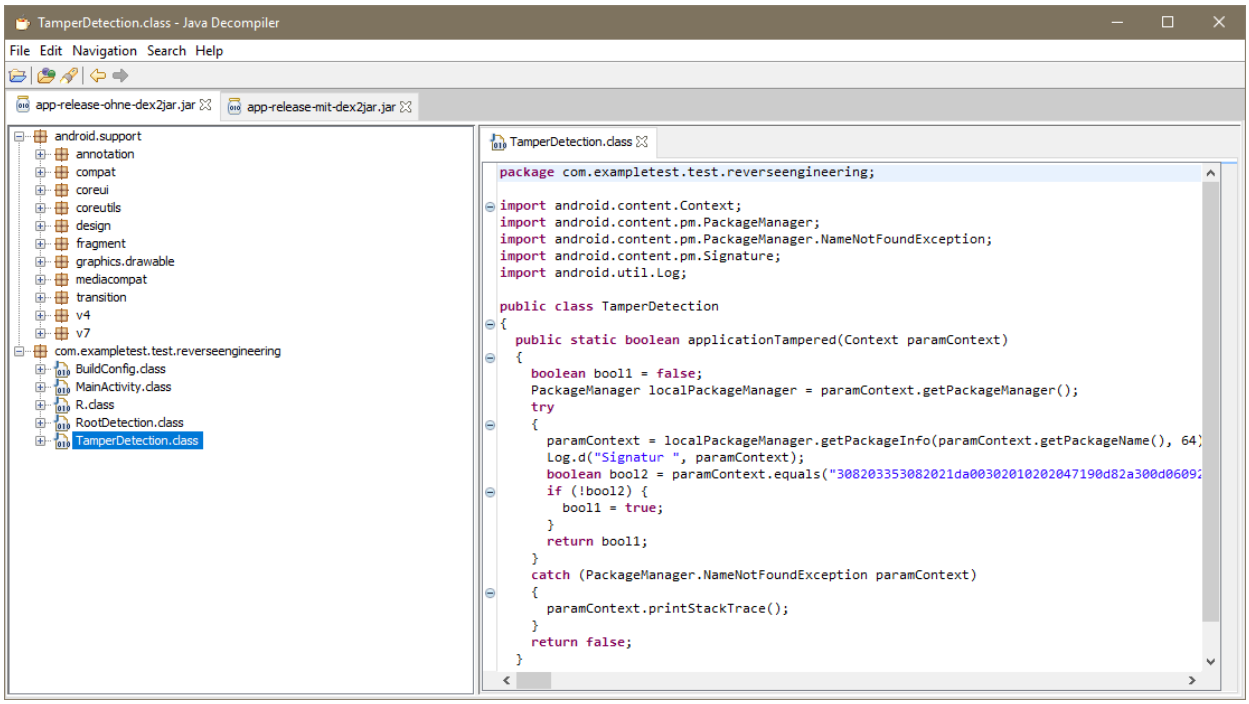


Abbildung 6-25: Screenshot – Reverse Engineering mit Java Decompiler ohne ProGuard (Java Decompiler, 2017)

In Abbildung 6-25 sind alle Klassen sichtbar, die für den „Fall 6: Reverse Engineering“ erstellt wurden. Die `RootDetection.class` wurde zwar in der `MainActivity.class` auskommentiert, ist jedoch nach dem Reverse Engineering weiterhin sichtbar. Der nächste Screenshot zeigt die „app-release-mit.apk“ nach dem Reverse Engineering:

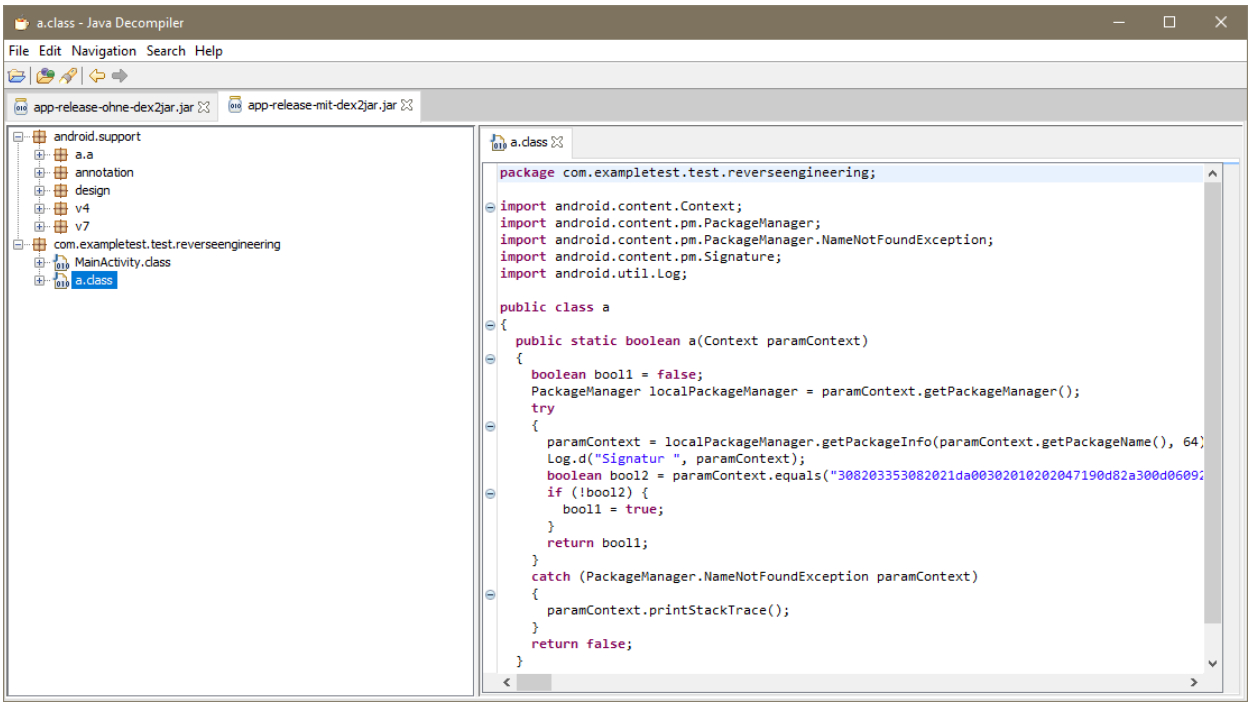


Abbildung 6-26: Screenshot – Reverse Engineering mit Java Decompiler und ProGuard (Java Decompiler, 2017)

Wie in Abbildung 6-26 ersichtlich, sind weniger Klassen sichtbar, als bei der „app-release-ohne.apk“. Der Grund dafür ist, dass ProGuard nur Klassen kompiliert die auch in der Anwendung verwendet werden. Die `a.class` stellt die `TamperDetection.class` dar. Des Weiteren wurde die Methode `applicationTampered` in `a` umbenannt. Werden alle Klassen und Methoden in dieser Form verschleiert, wird es für Angreiferinnen und Angreifer schwieriger den Code zu lesen und zu manipulieren.

Der letzte Schritt ist es diesen Code zu verwenden und eine neue Anwendung zu erstellen. Die `content_main.xml` konnte nicht aus der APK-Datei wiederhergestellt werden und musste neu erstellt werden. In der `MainActivity`-Klasse und in der `TamperDetection`-Klasse mussten IDs, Namen und Datentypen neu eingefügt werden. Der neue Code unterscheidet sich vom Original-Code, indem INT-Werte als Zahl und nicht mehr als Referenz dargestellt werden. Ein Beispiel dafür wäre die Referenz `PackageManager.GET_SIGNATURES`, die als `64` angezeigt wird. Im Listing 6-30 ist der Code ersichtlich, der nachdem die Referenzen eingetragen wurden, wieder ausgeführt werden kann.

```
public static boolean applicationTampered(Context paramContext)
{
    boolean bool1 = false;
    PackageManager localPackageManager = paramContext.getPackageManager();
    try
    {
        String sig =
localPackageManager.getPackageInfo(paramContext.getPackageName(),
PackageManager.GET_SIGNATURES).signatures[0].toCharsString();
        Log.d("Signatur ", sig);
        boolean bool2 = sig.equals("308203……020204");
        if (!bool2) {
            bool1 = true;
        }
        return bool1;
    }
    catch (PackageManager.NameNotFoundException a)
    {
        a.printStackTrace();
    }
    return false;
}
```

Listing 6-30: Code nach dem Reverse Engineering

Mit dieser Anwendung wird eine neue signierte APK, mit neuem Key-Store erstellt. Folgendes Bild zeigt die neu erstellte Anwendung mit Reverse Engineering:

Abbildung 6-27: Tamper Detection hat Änderungen erkannt

Wien Abbildung 6-27 zu entnehmen ist, wurden die Änderungen durch die Tamper Detection erkannt. Da jedoch der Code von der Angreiferin oder dem Angreifer sichtbar ist, kann er ohne Probleme entfernt werden.

6.8 Security Frameworks mit Designregeln

Die Abbildung 6-28 zeigt die Security Frameworks die anhand der theoretischen und praktischen Ausarbeitung erstellt wurden. Sie gliedern sich in „Lokale Sicherheit“, „Kommunikationssicherheit“ und „Reverse Engineering“. Jedes Framework enthält Designregeln die zur Wahrung der mobilen Sicherheit eingehalten werden sollen.

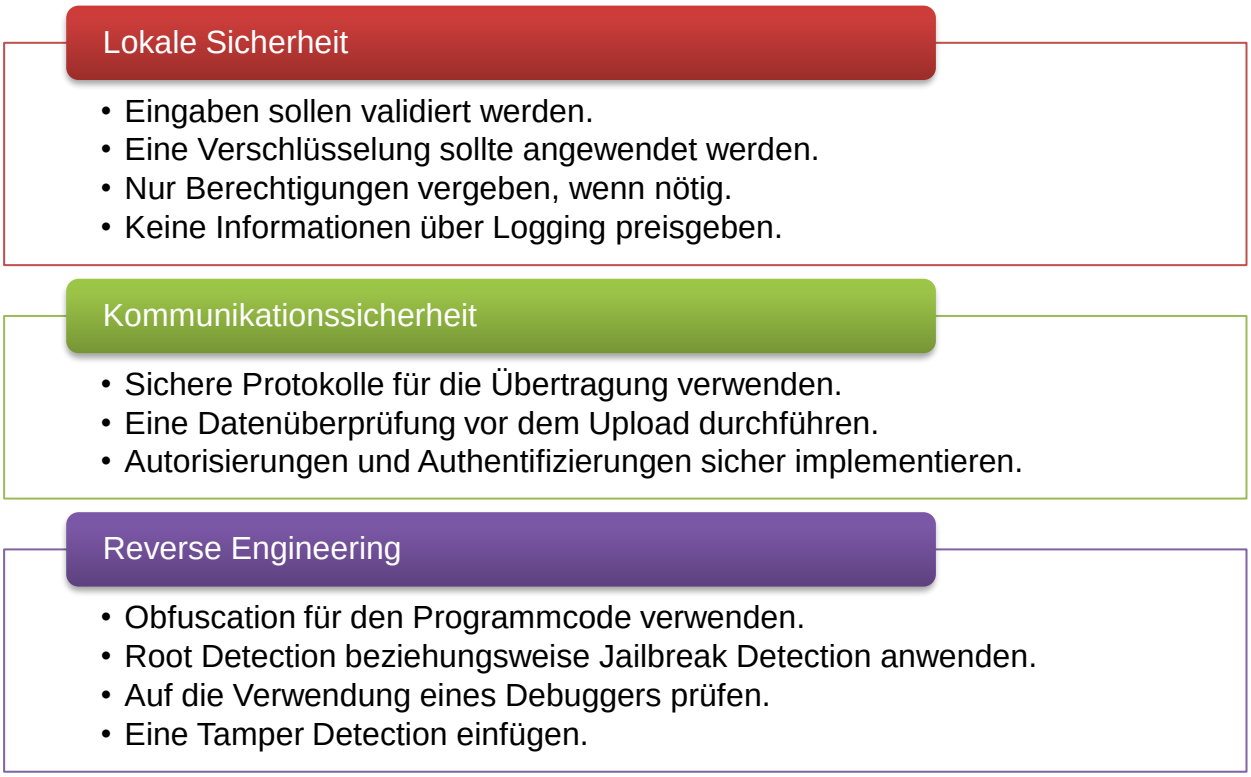


Abbildung 6-28: Security Frameworks mit Designregeln

Im Folgendem werden die Frameworks mit den Designregeln genauer beschrieben.

6.8.1 Lokale Sicherheit

Werden Daten nur lokal gespeichert, sollte die Sicherheit nicht vernachlässigt werden. Es besteht immer die Möglichkeit, dass Anwendung das Dateisystem auf Daten ausspähen und diese dann stehlen. Des Weiteren sollte jede Eingabe auf mögliche Gefahren überprüft werden, sodass kein Kompromittieren der Anwendung möglich ist. Die folgenden Designregeln sollten für das Implementieren jeder Anwendung beachtet werden.

Eingaben sollen validiert werden.

Wie im Kapitel 5.3 beschrieben, können Eingaben von unterschiedlichen Eintrittspunkten verarbeitet werden. Jede Eingabe kann die Gefahr eines Injection-Angriffs (Kapitel 4.4) mit sich bringen. Des Weiteren sollte darauf geachtet werden, welchen Quellen vertraut wird, da die Möglichkeit von „Security Decisions via Untrusted Inputs“ (Kapitel 4.13) besteht. Der „Fall 4: Eingaben überprüfen versus jede Eingabe zulassen“ hat gezeigt, wie schnell mittels SQL-Injection Daten kompromittiert werden können. Deshalb ist es wichtig, dass jede Eingabe vor der Weiterverarbeitung überprüft wird.

Eine Verschlüsselung sollte angewendet werden.

Daten die für keine weitere Anwendung oder Person lesbar sein sollen, sollten verschlüsselt werden. Werden sensible Daten im Klartext gespeichert, wird von einem „Insecure Data Storage“ (Kapitel 4.9) gesprochen. Werden Daten zwar verschlüsselt, aber die Verschlüsselung ist nicht sicher, handelt es sich um „Broken Cryptography“ (Kapitel 4.14). Unter „5.2 Daten sicher aufbewahren“ wird gezeigt, wie eine AES-Verschlüsselung funktioniert.

Nur Berechtigungen vergeben, wenn nötig.

Können Daten nicht verschlüsselt werden, weil sie zum Beispiel von einer anderen Anwendung benötigt werden, sollten nur die notwendigen Berechtigungen vergeben werden. Der „Fall 3: Daten speichern und stehlen“ und das Kapitel 5.2 zeigen, wie ein Berechtigungsschema, den Zugriff auf eine Datei verändern kann.

Keine Informationen über Logging preisgeben.

Für ein Unintended Data Leakage (siehe Kapitel 4.11) gibt es mehrere Möglichkeiten, wie zum Beispiel Logging. Logging ist eine Quelle von ungewollter Informationsfreigabe, wie in Kapitel 5.5 beschrieben. Im „Fall 5: Logging eingrenzen“ wurden Möglichkeiten aufgezeigt, wie Logging-Informationen vor dem Deployment entfernt werden können. Um nicht ungewollt Daten freizugeben, sollte beim Logging, Caching oder bei sonstige Aufzeichnung auf die Freigabe von Informationen geachtet werden.

6.8.2 Kommunikationssicherheit

Eine sichere Kommunikation ist nicht nur für die Anwenderinnen oder Anwender wichtig, sondern soll auch den Server oder die Anwendung vor Angriffen schützen. Folgende Designregeln geben einen Überblick über die wichtigsten Sicherheitsaspekte bei der Kommunikation.

Sichere Protokolle für die Übertragung verwenden.

Jede Kommunikation soll vor Abhörangriffen und Manipulationen gesichert werden, unabhängig davon ob es sich um sensible Daten handelt oder nicht. Das Kapitel 5.4 zeigt Methoden um „Insufficient Transport Layer Protection“ (Kapitel 4.10), „Man-in-the-middle“ (Kapitel 4.2) und „Cross-Site Scripting“ (Kapitel 4.5) zu vermeiden. Im „Fall 2: Daten übertragen und verlieren“ wird gezeigt wie leicht „Unsichere Protokolle“ (Kapitel 4.1) abgehört werden können. Daten sollten deshalb nie im Klartext, sondern immer verschlüsselt übertragen werden.

Eine Datenüberprüfung vor dem Upload durchführen.

Die serverseitige Sicherung liegt zwar bei der mobilen Entwicklung nicht an erste Stelle, sie sollte aber dennoch beachtet werden. Eine Nicht-Beachtung kann schwere Folgen haben, indem Angriffe wie „Denial of Service“ (Kapitel 4.3) oder „Session Hijacking“ (Kapitel 4.7) mittels „Cross-Site Request Forgery“ (Kapitel 4.6) erleichtert werden. Des Weiteren sollten alle Services auf Schwachstellen überprüft werden, die im Kapitel 4.8 „Weak Server-Side Controls“ beschrieben sind. Das Kapitel 5.1 zeigt wie die Angriffe geschwächt oder sogar vermieden werden können.

Autorisierungen und Authentifizierungen sicher implementieren.

Autorisierungen oder Authentifizierungen können für unterschiedliche Bereiche benötigt werden. Wie in Kapitel 5.1.1 „Authentifizierung von Datei-Uploads“ beschrieben wird, kann eine Authentifizierung für die Nachverfolgung einer Datei verwendet werden. Um Anmeldeinformationen zu schützen, können OAuth und Challenge/Response (siehe Kapitel 5.4.3) verwendet werden. Werden Autorisierungen oder Authentifizierungen nicht korrekt implementiert, wird von „Poor Authorization and Authentication“ (Kapitel 4.12) gesprochen. Fehlerhafte Implementierungen können dazu führen, dass eine Anwenderin oder ein Anwender einen kompletten Zugriff auf die Anwendung ohne Erlaubnis erhält.

6.8.3 Reverse Engineering

Die Sicherung des eigenen Programmcodes dient nicht nur dem Schutz des geistigen Eigentums, sondern soll auch eine Veränderung der Anwendung erschweren. Das Kapitel 4.15 „Lack of Binary Protections“ beschreibt den Angriff auf binären Code. Der „Fall 6: Reverse Engineering“ widmet sich den Angriffs-, sowie auch den Verteidigungsmöglichkeiten. Die folgenden Designregeln sollten bei der Entwicklung, zum Schutz der binären Dateien, beachtet werden. Ein vollständiger Schutz gegen Reverse Engineering ist aber nicht möglich, da der Code, der zur Sicherung der Anwendung verwendet wird, immer entfernt werden kann.

Obfuscation für den Programmcode verwenden.

Kompilierte Anwendungen können sehr einfach wieder zu lesbaren Code dekompiliert werden. Aufgrund dessen ist es wichtig Obfuscation für die Verschleierung des Codes zu verwenden. Das Kapitel 5.6.1 „Obfuscation“ zeigt, wie mithilfe von Tools dies erreicht werden kann. Im Fallbeispiel von Kapitel 6.7.3 wird gezeigt, wie ProGuard den Programmcode verschleiert.

Root Detection beziehungsweise Jailbreak Detection anwenden.

Ein Smartphone, das mittels Root oder Jailbreak verändert wurde, kann ein Sicherheitsrisiko darstellen, da viele Sicherheitsmechanismen des Betriebssystems bei diesem Prozess ausgehebelt werden. Deshalb ist es wichtig, eine Root oder Jailbreak Erkennung in die Anwendung zu implementieren. Die Kapitel 5.6.2 „Root Detection“ und 5.6.3 „Jailbreak Detection“ zeigen, wie eine Erkennung aussehen kann. Des Weiteren stellt das Fallbeispiel im Kapitel 6.7.1 eine Umsetzung einer „Root Detection“ in Android dar.

Auf die Verwendung eines Debuggers prüfen.

Debugging ermöglicht es, einen Programmcode einfach zu überprüfen. Während der Entwicklung kann das ein großer Vorteil sein, jedoch sollten fremde Personen nicht die Möglichkeit haben, den Code so einfach zu analysieren. Wie eine Überprüfung auf einen Debugger erfolgen kann, wird im Kapitel 5.6.4 „Debugger Detection“ gezeigt. Die Ausarbeitung des Fallbeispiels im Kapitel 6.7.2 zeigt eine Implementierung am Betriebssystem Android.

Eine Tamper Detection einfügen.

Wird eine Anwendung verändert, kann eine Tamper Detection die Veränderung erkennen und das Programm, beim Ausführen hindern. Die Tamper Detection wurde zusammen mit Obfuscation im Kapitel 6.7.3 getestet. Weitere Möglichkeiten und Tools zu Verteidigung werden im Kapitel 5.6.5 „Tamper Detection“ gezeigt.

7 ZUSAMMENFASSUNG UND AUSBLICK

Im diesem Kapitel werden aus den Ergebnissen dieser Arbeit Schlussfolgerungen gezogen, sowie ein Ausblick gegeben. Diese Arbeit hat sich mit mobilen Sicherheitsrisiken und deren Verteidigungsmöglichkeiten auseinandergesetzt. Es wurde der Forschungsfrage nachgegangen, welche Designregeln und Security Frameworks zur Wahrung der IT Sicherheit, bei mobilen Applikationen angewendet werden können. Dabei wurde mithilfe der theoretischen Ausarbeitung und den praktischen Testfällen, Security Frameworks mit zugehörigen Designregeln erstellt.

Die Ausarbeitung hat drei Frameworks ergeben, die sich in „Lokale Sicherheit“, „Kommunikationssicherheit“ und „Reverse Engineering“ aufteilen. Zu jedem Framework wurden wiederum Designregeln ausgearbeitet. Diese Designregeln sollen bei der Entwicklung einer Anwendung helfen, die wichtigsten Sicherheitsrisiken aufzuspüren. Sie zeigen die zugehörigen Angriffsvektoren an und wie die Verteidigung gegen diese aussehen kann. Die Fallbeispiele zeigen dabei die Unterschiede, die durch die Verwendung oder der Nicht-Verwendung von Sicherheitsmaßnahmen entstehen.

Das Ergebnis dieser Masterarbeit ist, dass durch Verwendung von Security Frameworks und Designregeln die IT-Sicherheit verbessert werden kann. Es ist aber zu beachteten, dass nicht jeder Angriff abgewehrt werden kann und das immer wieder neue Angriffsvektoren entstehen können. Aus diesem Grund kann die aufgestellte Hypothese H1 „Designregeln und Security Frameworks beeinflussen die Sicherheit von mobilen Anwendungen.“ angenommen werden und H0 verworfen werden.

Für die Zukunft sollen die Entwicklungen in der mobilen Softwareentwicklung beobachtet und beim Auftreten neuartiger Angriffsvektoren, diese Frameworks weiterentwickelt und ausgebaut werden. Der nächste Testlauf sollte sich in einem praxis-orientierten Umfeld befinden, um ein genaueres und klareres Ergebnis zu erhalten. Werden diese als positiv beurteilt, könnten die Frameworks als Grundlage für die sichere mobile Entwicklung aufgenommen werden.

Die Autorin erhofft sich durch diese Arbeit, besonders Anfängerinnen und Anfänger in der mobilen Entwicklung, einen Überblick über bekannte Gefahren zu geben und sie bei der Entwicklung von sicheren Anwendung zu unterstützen. Des Weiteren soll diese Arbeit helfen, mögliche Angriffspunkte in bestehenden Applikationen aufzuzeigen und Beispiele zur Entfernung dieser Gefahren und zur Sicherung der Anwendung, anbieten.

ABKÜRZUNGSVERZEICHNIS

ADB ..... Android-Debug-Bridge
AES ..... Advanced Encryption Standard
API ..... Application Programming Interface
APK..... Android Package
ASL ..... Apple System Log
ASLR ..... Address Space Layout Randomization
CRC32 ..... Cyclic Redundancy Check (Polynom 0x104C11DB7)
DDoS ..... Distributed Denial of Service
DEP ..... Data Execution Protection
DoS ..... Denial of Service
FDE ..... Festplattenverschlüsselung
HTTP ..... Hypertext Transfer Protocol
IPC..... Interprozesskommunikation
JAR ..... Java Archive
JSON ..... JavaScript Object Notation
LLB ..... Low-Level-Bootloaders
LLVM ..... Low Level Virtual Machine
OS..... Operating System / Betriebssystem
MAC..... Message Authentication Code
NDK ..... Native Development Kit
PHP ..... Hypertext Preprocessor
REST ..... Representational State Transfer
ROP ..... Return-oriented programming
SIM ..... Subscriber Identity Module
SMS ..... Short Message Service
SHA1 ..... Secure Hash Algorithmus 1
SQL..... Structured Query Language
SOAP..... Simple Object Access Protocol
SSL ..... Secure Socket Layer
TLS ..... Transport Layer Security
TXT ..... Textdatei
UID..... Unique Identifier
URI..... Uniform Resource Identifier
WSDL ..... Web Services Description Language
XML ..... Extensible Markup Language
XSRF ..... Cross-Site Request Forgery
XSS ..... Cross-Site Scripting

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

Chaffey, D. (18. Oktober 2016). *Mobile Marketing Statistics compilation*. Von Smart Insights (Marketing Intelligence) Ltd: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/ abgerufen

Chell, D., Erasmus, T., Colley, S., & Whitehouse, O. (2015). *The Mobile Application Hacker's Handbook.* Indianapolis, IN: John Wiley & Sons, Inc.

dex2jar. (27. Februar 2017). *SourceForge*. Von https://sourceforge.net/projects/dex2jar/ abgerufen

Drake, J. J., Fora, P. O., Lanier, Z., Mulliner, C., Ridley, S. A., & Wicherski, G. (2014). *Android™ Hacker's Handbook.* Indianapolis, Ind.: Wiley.

Glaser, J. D. (2014). *Secure Development for Mobile Apps: How to Design and Code Secure Mobile Applications with PHP and JavaScript.* Boca Raton, FL: CRC Press.

Gunasekera, S. (2012). *Android Apps Security: create apps that are safe from hacking, attacks, and security breaches.* New York, NY : LinkAPress .

Java Decompiler. (27. Februar 2017). *Java Decompiler*. Von http://jd.benow.ca/ abgerufen

LGEPR. (24. Februar 2017). *Wikimedia Commons*. Von https://commons.wikimedia.org/wiki/File:LG_Optimus_2X.jpg abgerufen

Lockhart, J., Smith, A., Allen, R., & SlimFrameworkTeam. (16. März 2017). *Slim a micro framework for PHP*. Von Slim: https://www.slimframework.com/ abgerufen

MacLean, D., Komatineni, S., & Allen, G. (2015). *Pro Android Five .* New York, NY : Apress .

Nolan, G. (2015). *Bulletproof Android: practical advice for building secure apps.* Addison Wesley.

OnePlus. (24. Februar 2017). *OnePlus*. Von OnePlus: https://oneplus.net/press-photos abgerufen

Pakalski, I. (24. Februar 2017). *Golem Media GmbH*. Von https://www.golem.de/news/android-7-0-google-veroeffentlicht-fertiges-nougat-1608-122838.html abgerufen

Richter, K., & Keeley, J. (2015). *Mastering iOS Frameworks: Beyond the Basics .* Pearson Education, Inc.

Schäfer, M. (13. Oktober 2016). *Smartphone-Marktanteile: Android mit Zuwachs auf allen wichtigen Märkten*. Von ComputerBase GmbH: https://www.computerbase.de/2016-04/smartphone-marktanteile-android-mit-zuwachs-auf-allen-wichtigen-maerkten/ abgerufen

simpligility technologies inc. (17. März 2017). *ksoap2-android - lightweight, efficient SOAP on Android*. Von ksoap2-android: http://simpligility.github.io/ksoap2-android/ abgerufen

Stuttard, D., & Pinto, M. (2011). *The web application hacker's handbook: finding and exploiting security flaws.* Indianapolis, Ind.: Wiley.

Tilkov, S., Eigenbrodt, M., Schreier, S., & Wolf, O. (2015). *REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web, 3. Auflage.* Heidelberg: dpunkt.verlag GmbH.

Turner, J. (2011). *Developing Enterprise iOS Applications: iPhone and iPad Apps for Companies and Organizations.* O'Reilly Media.

Whitwam, R. (24. Februar 2017). *Illogical Robot LLC*. Von AndroidPolice: http://www.androidpolice.com/2014/06/19/google-rolling-out-android-4-4-4-update-ktu84p-with-a-security-fix-factory-imagesbinaries-up-for-nexus-devices/ abgerufen

Wireshark Foundation. (18. März 2017). *Wireshark · Go Deep.* Von Wireshark: https://www.wireshark.org/ abgerufen

Ziegler, P. S. (2008). *Netzwerkangriffe von innen.* Köln: O'Reilly Verlag GmbH & Co. KG.