Zentrum für sichere Informationstechnologie – Austria
Secure Information Technology Center – Austria

A-SIT

Seidlgasse 22 / 9, 1030 Wien
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

Inffeldgasse 16a, 8010 Graz
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461
ZVR: 948166612
UID: ATU60778947

SICHERHEITSVERGLEICH VON APPS AUF ANDROID & IOS

Version 1.0 vom 21.12.2018
Johannes Feichtner – johannes.feichtner@a-sit.at

Eingebettet in signifikant unterschiedliche Laufzeitumgebungen werden mobile Anwendungen oft für die beiden dominierenden Mobilplattformen Android und iOS unabhängig voneinander entwickelt. Unterschiedliche Sicherheitskonzepte und Entwicklungsmuster führen dazu, dass gleiche Anwendungen nicht auf beiden Plattformen notwendigerweise dieselben Sicherheitsmerkmale aufweisen.

Im Rahmen dieses Projekts wurde untersucht, ob bzw. welchen Einfluss die Mobilplattform auf die Existenz sicherheitskritischer Probleme in Anwendungen hat. Im Zuge dessen soll anhand von praktischen Beispielen erhoben werden, ob eine Plattform unter Umständen die Präsenz von Entwicklungsfehlern begünstigen könnte.

Inhaltsverzeichnis

1. Einleitung

Welches Betriebssystem ist das „sicherste“? Diese salopp formulierte Frage wird seit einigen Jahren medial nicht nur in Bezug auf Betriebssysteme bei klassischen PCs diskutiert, sondern auch im Mobilbereich. Typischerweise[1] stellen Medienberichte dabei das unter Führung von Google entwickelte Android mit dem von Apple produzierten iOS gegenüber und nehmen für den angestellten Vergleich einzelne Eigenschaften heraus, die sie als wesentlich erachten. In vielen Artikeln werden schließlich die Häufigkeiten der festgestellten Pro- und Contra-Argumente zusammengezählt und ein „Gesamtgewinner“ abgeleitet. Je nach beurteilten Eigenschaften wird am Ende oft Android oder iOS als das jeweils „sicherere“ Betriebssystem für Mobilgeräte verkündet. Wird dabei Bezug auf Spezifika in aktuellen Versionen der jeweiligen Betriebssysteme genommen, so handelt es sich um eine Momentaufnahme, basierend auf ausgewählten Evaluierungsfaktoren.

Wenngleich es naheliegt, einen Sicherheitsvergleich auf Basis einzelner, selektiv gewählter Features in Android und iOS durchzuführen, gelingt ein direkter Vergleich nur auf einer abstrahierten Ebene, die Details der beiden Plattformen ausklammert. Typischerweise[2] werden hierfür Meta-Indikatoren wie die Häufigkeit von bereitgestellten Updates, die Anzahl der in den jeweiligen Plattformen bekannt gewordenen Schwachstellen, die Anzahl an unterschiedlich ausgestatteten Geräten („Device Fragmentation“) und Plattformversionen, sowie ähnliche Faktoren herangezogen. Ob und inwiefern einzelne dieser Faktoren sich tatsächlich als Maß für Sicherheit eignen, lässt sich nur schwer abschätzen, da jegliche Auswirkung nur indirekt zu tragen kommt und nicht unmittelbar ein Gerät „sicher“ oder „unsicher“ macht.

Die in den Plattformen realisierten Sicherheitskonzepte unterscheiden sich nicht nur herstellerbedingt signifikant voneinander, sondern auch darüberhinausgehend. So beschränkt Apple durch Maßnahmen in Hard- und Software, wie auf verbaute Sensoren und andere Peripherie zugegriffen werden kann. Zurzeit ist es beispielsweise nicht möglich, dass iOS-Apps den Sensor zur Erkennung von Fingerabdrücken außerhalb des von Apple vorgegebenen Rahmenwerks einsetzen. Gleichermaßen kann der NFC-Chip aktuell[3] nur von Systemdiensten verwendet werden, die von Apple bereitgestellt werden. Die Verwendung von Zahlungsdiensten über NFC wird bei iOS-Geräten dadurch auf die hauseigene Lösung Apple Pay beschränkt. Android hingegen ist seit jeher als offenes Ökosystem konzipiert, bei dem der Hersteller für alle Anwendungen verwendbare Schnittstellen anbietet, die Zugriff auf Hardwarekomponenten ermöglichen. Ausnahmen davon finden sich nur bei Komponenten, die von eigenständigen Prozessoren verwaltet werden, wie z.B. dem Baseband oder dem „Secure Element“, das zur Speicherung von Schlüsselmaterial in einem isolierten Bereich dient. Darüber hinaus ist es bei Android gängige Praxis, dass Hersteller von Mobilgeräten eigene Lösungen hinzufügen, die potentiell neue Probleme einführen[4] oder welche, die das Maß an Sicherheit erhöhen sollen[5].

Die Vielfalt an verbauten Komponenten und unterschiedliche Sicherheitskonzepte, Indikatoren für Sicherheit auf einer Meta-Ebene und die schiere Vielfalt an Geräten und Versionen von Plattformen führen dazu, dass Anwendungen auf Android und iOS an diese Verhältnisse laufend angepasst werden müssen, um überhaupt zu funktionieren. Unter der Prämisse, dass es die Aufgabe einer Anwendung ist, eingegebene Daten sicher zu verarbeiten und abzuspeichern, müssen EntwicklerInnen nicht nur mit den jeweiligen Sicherheitskonzepten der Plattformen vertraut sein, sondern auch unterschiedlichste Ausprägungen davon berücksichtigen. Vor Android 6.0 war es beispielsweise nicht möglich, im „Secure Element“ kryptographisches Schlüsselmaterial für AES abzulegen. Ebenso sind Hersteller von Android-Geräten seit Erscheinen von Android 6.0 seitens Google verpflichtet[6], die Verschlüsselung von Geräten standardmäßig zu aktivieren, sofern es die Performance des Geräts erlaubt, kryptographische Operationen innerhalb eines gewissen

[1] https://www.digitaltrends.com/mobile/android-vs-ios/
[2] https://medium.com/@AppInventiv/android-vs-ios-which-platform-is-more-secure-in-2018-33b3108270d
[3] https://www.paymentsjournal.com/has-apple-opened-up-nfc-on-iphones/
[4] https://www.nowsecure.com/blog/2017/11/14/oneplus-device-root-exploit-backdoor-engineermode-app-diagnostics-mode/
[5] https://www.samsungknox.com/de
[6] https://source.android.com/compatibility/android-cdd.pdf

Zeitfensters durchzuführen. EntwicklerInnen von Anwendungen stehen schließlich vor dem Dilemma, sich entscheiden zu müssen, wem die Verantwortung über die sichere Ablage sensitiver Informationen zuteilwird: Soll die App sich selbst darum kümmern oder ist es Aufgabe der darunterliegenden Plattform? In der Praxis zeigt sich, dass die eigene Implementierung von sicherheitsrelevanter Funktionalität, z.B. der Verschlüsselung von Dateien, oft mit Fehlern behaftet ist. Ein Delegieren dieser Verantwortung an die Plattform könnte jedoch wiederum implizieren, dass kein oder nur ein sehr geringes Maß an Datensicherheit gewährleistet werden könnte. Wird der Schutz sensitiver Daten an die Plattform abgegeben, ist es für Anwendungen daher essentiell, die verwendeten Schutzmechanismen so „musterhaft" einzusetzen, dass der angestrebte Effekt auch tatsächlich erreicht wird.

Wie zuvor exemplarisch angemerkt, gibt es viele Vielzahl unterschiedlicher Faktoren, die Einfluss auf die „sichere Ausführung" von Apps haben können. Selbst wenn Anwendungen bestmöglich implementiert sind, kann ein sicherheitskritisches Problem plattformbedingt vorliegen. Das Paradigma, dass Sicherheit durch eine korrekt implementierte App gegeben ist, wäre also falsch. Als EntwicklerIn von Anwendungen ist es in der Praxis nicht möglich, das Vorhandensein einer Laufzeitumgebung zu erzwingen, die alle Sicherheitsfunktionen gewährleistet. In Anlehnung daran hat Google mit Android 8.0 die „SafetyNet" API[7] vorgestellt, die es Apps ermöglicht, zu prüfen, ob gewisse Bedingungen, wie z.B. kein Rooting, Verwendung eines Betriebssystems des Geräteherstellers (kein custom ROM), kein Hooking von API-Methoden, etc. zur Laufzeit erfüllt sind. Angesichts der Heterogenität von Geräten und Versionen von Betriebssystemen, sind Apps jedoch in der Regel so gestaltet, auf unterschiedlichen Geräten zu funktionieren, und nicht nur mit einem einzigen Modell oder einer gewissen Plattformversion. Da zurzeit[8] noch fast 80% der Android-Geräte eine ältere Android-Version einsetzen, ist „SafetyNet" keine Universallösung für die geschilderte Problemstellung.

Im vorliegenden Projekt wird die Sicherheit von Anwendungen unter Android und iOS bei Einsatz von Plattformfunktionalität zum Schutz sensitiver Daten untersucht. Wie eingangs beschrieben, definiert das Betriebssystem die Rahmenbedingungen für Apps. Unter der Annahme, dass bei den jeweils eingesetzten Geräten übliche Sicherheitsfunktionalität vorhanden und aktiviert ist, wie z.B. Geräteverschlüsselung, nicht modifizierte Bootloader und Kernel (kein Rooting bzw. Jailbreaking), keine Anwendungen aus Drittquellen, etc. verbleibt die **zentrale Frage, ob für Anwendungen gleiche Bedingungen unter Android und iOS bestehen.**

Dazu gehören im Wesentlichen folgende Forschungsfragen:

- Können bei der Entwicklung von Apps für Android und iOS konzeptionell die gleichen Fehler passieren?
    - Wenn ja, worin sind sie verankert: In falscher Verwendung von Systemschnittstellen, in fehlender Dokumentation, in falschen Grundeinstellungen (Default-Werte)?
    - Wenn nein, kann es sein, dass eine Plattform sicherheitskritische Probleme ggf. begünstigt?
- Wo treten sicherheitskritische Fehler üblicherweise auf? Dies impliziert auch die Frage, in welchen Bereichen die entsprechende Funktionalität prinzipiell eingesetzt wird.

Der Fokus wird in diesem Projekt auf Einsatzbereiche von Plattformfunktionalität gelegt, bei denen im Problemfall ein vergleichsweise hoher „Impact" gegeben ist. Üblicherweise handelt es sich dabei auch um Problemstellen, die normalerweise geprüft werden, wenn Apps für Android und iOS im Zuge von statischer oder dynamischer Analyse ausgewertet werden. In den nachfolgenden Abschnitten wird anhand praktischer Beispiele auf typisch gefundene Probleme bei der Analyse von Android- und iOS-Anwendungen eingegangen. Obwohl die Betriebssysteme in sich sehr unterschiedlich sind, sind z.B. die sichere Ablage von Dateien oder der Schutz von Netzwerkverbindungen übergeordnete Ziele, für die die Plattformen APIs bereitstellen. Nachfolgend wird erhoben, was beim konkreten Einsatz der jeweiligen Schnittstellen unter Android und iOS misslingen kann.

[7] https://developer.android.com/training/safetynet/attestation
[8] https://developer.android.com/about/dashboards/

2. Angriffsfläche in Apps

Sowohl unter Android, als auch unter iOS gibt es potentielle Problemfelder in Anwendungen, die durch die konkrete Umsetzung von ähnlicher Funktionalität bedingt sind. So stellen beispielsweise beide Plattformen Schnittstellen bereit, um Eingaben von BenutzerInnen zu verarbeiten oder um Dateien im Dateisystem abzulegen. Die nachfolgende Übersicht fasst sicherheitsrelevante Angriffsvektoren zusammen, gegen die sowohl Android- als auch iOS-Anwendungen anfällig sein können. Unter Verweis auf die in der Einleitung genannten Faktoren, ist die nachfolgende Liste eine Sammlung von in der Praxis sehr häufig beobachtbarer Probleme.

- Datenblage bzw. Lesen von Dateien aus nicht vertrauenswürdigen Quellen.
- Fehlerhaft umgesetzte oder fehlende Validierung von
    - Eingaben im Zuge von Interprozesskommunikation (IPC) oder URL-Schemata.
    - Inhalten in Eingabefelder, die von BenutzerInnen ausgefüllt werden.
- Nicht sichere Verwendung von WebViews.
- Potentiell unsichere Netzwerkkommunikation mit Servern im Internet oder Anfälligkeit für Man-in-the-Middle (MITM)-Angriffe.
- Falsche Konfiguration von Schnittstellen für kryptographische Anwendungen.

In den nachfolgenden Abschnitten wird auszugsweise auf die beschriebenen Punkte unter Veranschaulichung durch praktische Beispiele eingegangen. Nach Erläuterung des individuellen Problems und der daraus resultierenden Konsequenz auf die Sicherheit einer Anwendung, wird auf die unterschiedlichen Ausprägungen in Android und iOS Bezug genommen.

3. Datenablage

Der Schutz sensibler Informationen, wie Token für Authentifizierungszwecke, kryptographische Schlüssel, persönliche Dateien oder Passwörter ist wesentlich, um ihre Vertraulichkeit zu gewährleisten. Apps unter Android und iOS können grundsätzlich frei bestimmen, wo Dateien und Daten abgelegt sind – passiert das ohne die entsprechende Schutzfunktionalität, kann eine Datei oder ein Schlüssel möglicherweise ausgelesen und in die Hände von unberechtigten Dritten gelangen.

3.1. Android

Durch die Vielzahl an Schnittstellen und den im Vergleich mit iOS relativ vielfältigen Zugriff auf das Dateisystem, können Apps unter Android aus mehreren Varianten wählen, um Daten abzulegen. Nachfolgend werden die Möglichkeiten unter Betrachtung der Sicherheitsaspekte erörtert.

3.1.1. Shared Preferences

Die „SharedPreferences“ API bietet Anwendungen die Möglichkeit, Kombinationen aus Key-Value permanent abzuspeichern. Intern werden die über die „SharedPreferences“-Schnittstelle gespeicherten Daten in einer unverschlüsselten XML-Datei abgelegt, die je nach gewähltem Modus privat (nur für die aktuelle App zugreifbar) oder lesbar für alle Apps ist. Ein Beispiel wäre wie folgt:

```
SharedPreferences sharedPref = getSharedPreferences("secretinfo", MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "asituser");
editor.putString("password", "asitpassword");
editor.commit();
```

Bei Ausführen dieses Codefragments wird eine XML-Datei namens „*secretinfo.xml*“ im Ordner angelegt, der zur aktuellen Applikation gehört. Die Datei wäre demnach im Verzeichnis *„/data/data/at.asitapp/shared_prefs/secretinfo.xml“* zu finden. Das vorliegende Codefragment führt zu zwei wesentlichen Problemen:

1. Username und Passwort werden im Klartext in einer Datei abgelegt. Jeder mit Zugriff auf diese Datei kann ohne weitere Einschränkungen den hinterlegten Benutzernamen und das Passwort auslesen. Besonders wenn ein Gerät gerootet wurde oder ein Backup des

Dateisystems angefertigt wurde, ist ein Zugriff auf diese Datei ohne Umwege möglich. Das Grundproblem ist also, dass sensitive Daten in einem Bereich abgelegt wurden, der hierfür nicht geeignet ist.

2. Der Modus „MODE_WORLD_READABLE“ ist mittlerweile zwar offiziell als nicht mehr zur Verwendung empfohlen („deprecated“) markiert[9], findet sich aber immer noch in Anwendungen. Der XML-Datei werden dadurch Dateirechte zugewiesen, die es ermöglichen, dass beliebige andere Android-Anwendungen ebenso lesend auf die Datei zugreifen. Der Umweg über ein Backup oder ein gerootetes Gerät entfällt somit.

3.1.2. SQLite-Datenbanken

Von der Plattform bereitgestellt, können Android-Anwendungen Tabellen und Relationen in SQLite-Datenbanken abbilden. Analog zur „SharedPreferences“-API ist es dabei die Entscheidung von EntwicklerInnen, was in diesen Datenbanken permanent gespeichert werden soll. Das Resultat wird in einer Datei im Schema *„/data/data/at.asitapp/databases/meinesqlitedb*“ im Dateisystem abgelegt. Werden also sensitive Informationen in SQLite-Datenbanken gespeichert, unterliegen sie den gleichen Risiken wie Daten, die in „SharedPreferences“ abgelegt werden.

Das zunehmende Bewusstsein, dass Daten in SQLite-Datenbanken nicht verschlüsselt sind, befördert die Popularität von Lösungen wie SQLCipher[10], die die Inhalte der Datei symmetrisch verschlüsseln. Das in der Praxis oft beobachtbare Problem dieses Ansatzes ist, dass das Passwort, das intern als Basis für die Ableitung eines kryptographischen Schlüssels verwendet wird, als Konstante im Quellcode der Anwendung, als Eintrag in den „SharedPreferences“ oder ohne weiteren Schutz in einer Datei hinterlegt wird. Durch das Speichern des Passworts in einem Bereich, der vergleichbare Zugangshürden hat wie die Datei mit der SQLite-Datenbank selbst, kann kein adäquater Schutz gegeben sein, um die Vertraulichkeit der Inhalte sicherzustellen. Abhilfe würde schaffen, wenn die Anwendung BenutzerInnen bei Bedarf des Zugriffs auf die Datenbank zur Eingabe eines als allgemein als sicher geltenden Passworts auffordern würde.

3.1.3. Dateiablage am internen oder externen Speicher

Von Anwendungen verwendete Dateien werden üblicherweise am internen Speicher des Geräts abgelegt und gelöscht, sobald eine Anwendung entfernt wird. Ein Zugriff auf die entsprechende Ordnerstruktur kann nur von Apps des gleichen Herstellers erfolgen, oder wenn erweiterter Zugriff auf das Dateisystem gegeben ist, z.B. durch Rooting oder Backup. Vergleichbar mit der bei „SharedPreferences“ beschriebenen Situation ist es möglich, Dateien im Modus „MODE_WORLD_READABLE“ bzw. „MODE_WORLD_WRITABLE“ abzulegen, sodass auch andere Apps darauf zugreifen können. Obwohl offiziell „deprecated“, finden sich noch immer Apps, die diese Eigenschaft implementieren und daher potentiell vertrauliche Daten veruntreuen.

Werden Dateien am externen Speicher (in der Regel eine MicroSD-Karte) abgelegt, sind sie per se lesbar für alle. In der Praxis zeigt sich, dass Apps insbesondere dann externe Speichermedien ansprechen, wenn sie mit größeren Volumina an Daten arbeiten. Dazu gehören beispielsweise Messenger-Apps, die Videos, Fotos und Audio-Mitschnitte tendenziell auf den externen Speicher verschieben, um das interne Dateisystem nicht zu erschöpfen. Diese Dateien werden auch nicht entfernt, sobald eine Anwendung gelöscht wird. Zweifelsfrei ist der externe Speicher somit nicht geeignet, um sensible Informationen unverschlüsselt im Dateisystem abzulegen.

3.2. iOS

Apples Betriebssystem setzt zur Verwaltung von Dateien auf die sog. „Data Protection API“[11]. Im Gegensatz zu Android sind prinzipiell alle Dateien, die am System abgelegt werden, durch die standardmäßig aktive Verschlüsselung des Dateisystems geschützt. Jede Datei wird mit einer eignen Schüssel verschlüsselt, der zusammen mit den Metadaten durch einen „File System Key“ verschlüsselt ist. Die Metadaten wiederum sind durch einen sog. „Class Key“ geschützt, welcher angibt, zu welchem Zeitpunkt eine Datei zur Verwendung unverschlüsselt bereitgestellt wird.

[9] https://developer.android.com/reference/android/content/Context.html#MODE_WORLD_READABLE
[10] https://www.zetetic.net/sqlcipher/sqlcipher-for-android/
[11] https://www.apple.com/business/docs/iOS_Security_Guide.pdf

In der Praxis ist ein Umstand sehr oft beobachtbar, der mit dem jeweiligen „Class Key“ zusammenhängt. iOS kennt mehrere Schutzklassen für Dateien, die von EntwicklerInnen für den jeweiligen Einsatzzweck selbst festgelegt werden können. Bei der sichersten Variante „NSFileProtectionComplete“ wird der Schlüssel aus dem Passcode und einer gerätespezfiischen ID abgeleitet. Sobald das Gerät gesperrt wird, wird auch der Schlüssel aus dem Speicher entfernt. Bei der schwächsten Variante „NSFileProtectionNone“ findet die Verschlüsselung von Dateien nur die gerätespezifische ID statt, ohne den Passcode von BenutzerInnen miteinzubeziehen. Praktisch bedeutet das, dass eine Datei folglich immer entschlüsselt wird, sobald das Dateisystem geladen wird. Eine zusätzliche Interaktion oder die Eingabe eines Geheimnisses wird nicht benötigt.

Die Tatsache, dass es EntwicklerInnen obliegt, eine geeignete Schutzklasse selbst zu wählen, erfordert ein entsprechendes Verständnis der jeweiligen Unterschiede und Notwendigkeiten. Mit „NSFileProtectionNone“ sind potentiell sensible Dateien nicht nur im Backup, sondern können auch ohne Eingabe des Passcodes entschlüsselt werden.

Ein weiteres, in der Praxis sehr häufig auftretendes Problem ist, dass die Wahl der Schutzklasse optional ist. Wird beim Erstellen von Dateien und Ordnern keine angegeben, wird seit iOS 7 die Klasse „Protected Until First User Authentication“ gewählt, die die Schlüssel für Dateien im Speicher hält, sobald das Gerät erstmalig nach dem Bootvorgang entschlüsselt wurde. Davor wurde automatisch die vergleichsweise unsicherere Klasse „NSFileProtectionNone“ gewählt. Wesentlich ist, dass die Klasse, wenn nicht auf den Standardwert zurückgefallen werden soll, wirklich in jedem Fall explizit spezifiziert werden muss:

```
filemanager.createFile(..., attributes:
[FileAttributeKey.protectionKey.rawValue:FileProtectionType.complete])
...
try data.write(to: fileURL, options: .completeFileProtection)
```

Die korrekte Verwendung bzw. Spezifikation einer Schutzklasse wird in der Praxis dadurch erschwert, dass iOS-Anwendungen in mehreren Programmiersprachen (Objective-C und Swift) entwickelt werden können und es für die Schutzklasse keine einheitliche Schnittstelle gibt. Wie im Codefragment dargestellt, wird die Klasse je nach API-Methode als Option oder Attribut festgelegt. Im Fall der Verwendung der Methode createFile() der FileManager-Schnittstelle sieht die notwendige Definition fundamental anders aus als bei Verwendung der Data-Schnittstelle. Die offizielle Dokumentation[12] hält aber nur einen der beiden Fälle als Referenz fest.

Eine sehr ähnliche gelagerte Situation stellt auf iOS die Ablage von kryptographischem Schlüsselmaterial dar. Über die „KeyChain“-API kann die Verantwortung über die Verwaltung beliebiger Arten von Schlüssel an die Plattform abgetreten werden. Analog zu den Schutzklassen bei Dateien, gibt es bei der KeyChain ebenso Klassen[13], die festlegen, wann Schlüsselmaterial im Speicher vorgehalten wird, und wann es entfernt wird. „kSecAttrAccessibleAlways“ wäre vergleichbar mit „NSFileProtectionNone“ d.h. ein Schlüssel ist immer lesbar, unabhängig davon, ob das Gerät mithilfe eines Passcodes zuvor entsperrt wurde oder nicht. Bei der restriktivsten Variante „kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly“ wird die Verfügbarkeit von Schlüsseln auf das aktuelle Gerät limitiert d.h. ein Export der Schlüssel als Backup verunmöglicht und ein Zugriff im Speicher ist nur möglich, wenn das Gerät entsperrt wurde.

Standardmäßig wird bei der KeyChain die Schutzklasse „kSecAttrAccessibleWhenUnlocked“ eingesetzt. Einträge sind also lesbar, sobald das Gerät zum ersten Mal entsperrt wurde. Die standardmäßige Auswahl ist nur unter gewissen Bedingungen als sicher zu betrachten. „WhenUnlocked“ kann nur funktionieren, wenn ein Passcode gesetzt ist. Ist dies nicht der Fall,

12 https://developer.apple.com/documentation/uikit/core_app/protecting_the_user_s_privacy/encrypting_your_app_s_files

13 https://developer.apple.com/documentation/security/keychain_services/keychain_items/item_attribute_keys_and_values#1679100

fordert ein iOS-Gerät vor Anlegen eines Schlüssels mit der entsprechenden Klasse nicht, einen Passcode zu setzen, sondern betrachtet das Gerät einfach als immer „unlocked“. Das bedeutet, dass in Fällen, wo kein Passcode festgelegt ist, die standardmäßige Schutzklasse nicht besser ist, als die schlechteste „kSecAttrAccessibleAlways“. Jede App, die Einträge der KeyChain mit dieser Klasse schützt, müsste somit, um die Effektivität zu wahren, vor Ablage prüfen, ob überhaupt ein Passcode am Gerät gesetzt ist. Leider ist diese Überprüfung in der Praxis nicht trivial oder von der Plattform her gewährleistet, sondern muss von EntwicklerInnen selbst implementiert werden[14]. Darüber hinaus führt das fehlende Suffix „thisDeviceOnly“ bei der Schutzklasse dazu, dass KeyChain-Einträge auch im Backup inkludiert werden.

3.3. Diskussion

Durch die Kanalisierung von Dateizugriffen auf wenige vorgegebene APIs, verhindert iOS, dass Daten an Orten abgelegt werden können, wo der Schutz durch die Plattform nicht mehr gegeben ist. So kann beispielsweise eine SD-Karte vergleichsweise einfach von einem Android-Gerät entfernt und an einem anderen eingelesen werden. Die Vielzahl an Möglichkeiten, wo und auf welche Art und Weise Daten unter Android abgelegt werden können, trägt dazu bei, dass es mehrere Wege gibt, die den gleichen Effekt haben können, wenn sie richtig eingesetzt werden. Im Gegensatz zu iOS, wo die Verwendung der Schnittstellen für Dateiablage mit den Schutzklassen vom Hersteller exemplarisch vorgezeigt wird, fehlt etwas Vergleichbares bei Android.

Beiden Plattformen ist gemein, dass es bei der Verwendung der gegebenen Schnittstellen Parameter gibt, die optional zu befüllen sind und bei fehlendem Wert auf einen Standardverweis zurückgefallen, der nicht notwendigerweise die sicherste Implementierungsvariante darstellt. Ebenso bei beiden Plattformen ist klar erkennbar, dass plattformbedingt a priori nicht verhindert wird, dass Schlüsselmaterial oder potentiell sensible Dateien in Backups aufgenommen werden. Sollte dies nicht gewünscht sein, muss eine Anwendung diese Einstellung explizit hinterlegen.

Positiv ist anzuerkennen, dass sowohl bei Android, als auch bei iOS Modi inzwischen „deprecated“ sind oder auf bessere Standardeinstellungen geändert wurden, bei denen lange Zeit bekannt war, dass ihre Verwendung dem erreichbaren Sicherheitsniveau nur abträglich ist. Ungeachtet dessen sticht hervor, dass bei beiden Plattformen die Hersteller die Schnittstellen nicht dahingehend absichern, dass Ablagen von Schlüsselmaterial oder Daten nur unzureichend geschützt werden. Die Verantwortung für die Wahl sicherer Parameter liegt alleinig bei EntwicklerInnen. Dazu kommt, dass die standardmäßig gewählten Eigenschaften nur unter gewissen Voraussetzungen als sicher betrachtet werden können, z.B. bei iOS in Bezug auf die KeyChain dann, wenn ein Passcode gesetzt ist und ein Schlüssel im Backup inkludiert werden soll bzw. bei Android und der Speicherung von Daten auf der SD-Karte, wenn es sich dabei um keine besonders schutzwürdigen Informationen, wie etwa persönlichen Bildern, Videos oder Sprachnachrichten aus Messenger-Apps handelt.

4. Eingabevalidierung

Sowohl Android als auch iOS ermöglichen Interprozesskommunikation über benutzerdefinierte URL-Schemata. Über diese Strukturen ist es möglich, dass Applikationen gewisse Aktionen an andere Apps abtreten und auf das Resultat der Ausführung warten.

Beispielsweise würde der Klick auf einen Link im Schema „skype:echo123?call“ bei beiden Plattformen dazu führen, dass das System zunächst nach einer App sucht, die Unterstützung für den Präfix „skype:“ signalisiert. Die Anfrage würde dann an die gleichnamige Anwendung abgetreten, welche die gegebene URL interpretiert und folglich den Kontakt „echo123“ anruft. Derartige Links können auch in Apps eingebettet sein und ohne manuelle Interaktion eine Form eines „Intents“ losgetreten werden:

```
Uri marketUri = Uri.parse("market://details?id=com.skype.raider");
Intent myIntent = new Intent(Intent.ACTION_VIEW, marketUri);
myIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
myContext.startActivity(myIntent);
```

[14] https://bencoding.com/2017/01/04/checking-if-device-passcode-is-enabled/

Im vorliegenden, für Android konzipierten Codefragment wird eine URL mit Präfix „market“ konstruiert, die den Empfänger anweist, Details des App-Pakets „com.skype.raider“ anzuzeigen.

Benutzerdefinierte URL-Schemata sind überaus gebräuchlich, unterliegen jedoch gewissen Risiken. Bietet beispielsweise eine App für SMS-Versand Interprozesskommunikation über ein entsprechendes Schema an, könnte dies dazu führen, dass Nachrichten ohne Kenntnis oder zuvor zu erteilenden Berechtigungen an beliebige Adressaten verschickt werden könnten:
„sms://compose/to=+4366412345678&message=Testnachricht&sendImmediately=true“.
Um derartige illegitime Verwendungsarten von URL-Schemata zu verhindern, ist es daher essentiell, dass empfangende Apps prüfen, wer die Aktion ausgelöst hat und ob die URL plausibel ist.

Neben Eingaben mittels Interprozesskommunikation sind die meisten Mobilanwendungen auf Eingaben von BenutzerInnen angewiesen, die entsprechend verarbeitet werden müssen. Werden dabei sensible Daten wie z.B. Kreditkarteninformationen, Passwörter oder persönliche Informationen preisgegeben, ist es essentiell, dass diese Angaben nur in dem dafür gedachten Verwendungszweck eingesetzt werden. Die Plattform kann per se aber nicht unterscheiden, welche Art von Daten BenutzerInnen mit den Geräten teilen.

4.1. Android

Für den Einsatz von URL-Schemata sieht Android vor, dass jede App, die ein entsprechendes Schema anbietet, jenes in der Datei “AndroidManifest.xml” spezifiziert:

```
<activity android:name=".MyUriActivity">
  <intent-filter>
     <action android:name="android.intent.action.VIEW" />
     <category android:name="android.intent.category.DEFAULT" />
     <category android:name="android.intent.category.BROWSABLE" />
     <data android:scheme="myapp" android:host="path" />
  </intent-filter>
</activity>
```

Im vorliegenden Beispiel wird etwa das Schema „myapp“ registriert. Die Angabe der Kategorie „BROWSABLE“ legt fest, dass die entsprechende URL auch im Browser geöffnet werden darf. Anwendungen ist es folglich möglich, Anfragen an die App zu schicken, zu der das entsprechende „AndroidManifest.xml“ gehört und auf eine Ausführung zu warten. Im konkreten Fall könnte der Aufruf beispielsweise „myapp://some/path?key1=value1&key2=value2“ lauten. Wie die Parameter schließlich auseinandergenommen werden, ist Aufgabe der jeweiligen App.

Das grundsätzliche Problem unter Android ist, dass potentiell auch mehrere Apps für ein Schema zuständig sein können. Ohne, dass es die Bestätigung von BenutzerInnen bedürfte, kann sich etwa eine weitere App mit einem der zuvor genannten Schemata identifizieren und es anbieten. Besonders heikel ist dieser Umstand, wenn in den in der URL angegebenen Daten sensible Informationen, etwa zur Authentifizierung enthalten sind. Genauso ist es denkbar, dass eine Anwendung die Anfrage eines gewissen Schemas abfängt und Daten austauscht.

4.2. iOS

Interprozesskommunikation (IPC) kann bei iOS über verschiedene Methoden realisiert werden, die gemeinhin als „XPC Services“, „Mach Ports“ und „NSFileCoordinator“ bekannt sind. Die erstgenannte Methode ist jene, die von Apple für die Verwendung empfohlen wird[15] und vergleichsweise die geringsten Berechtigungen oder Dateizugriffe benötigt.

Die Erhebung ob eine Anwendung unter iOS IPC verwendet ist, ist vergleichsweise schwierig, da ein schnell sichtbarer Verweis nach außen, wie bei Android mittels der Datei „AndroidManifest.xml“, fehlt. Verwendet eine App die von Apple empfohlene Lösung, ist es wesentlich, Sicherheitsattribute

15 https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html

zu definieren[16], deren Plausibilität bei Empfang einer Aktion schließlich überprüft wird. Nur so kann verhindert, dass Anwendungen an andere Aktionsanfragen schicken, wenn diese eigentlich nicht dafür berechtigt sind.

4.3. Diskussion

Obwohl es in der Implementierung an sich wesentliche Unterschiede zwischen Android und iOS gibt und IPC bei iOS beispielsweise über drei Varianten umgesetzt werden kann, verwenden beide Plattformen URL-Schemata zum Informationsaustausch mit anderen Anwendungen. Ihnen ist gemein, dass die empfangende App die Plausibilität der gesendeten Daten, den Absender und die Berechtigung der Aktion vor Ausführung prüfen sollte. Ebenso ist es bei beiden möglich, dass mehrere Apps für die gleichen URL-Schemata hinterlegt werden, mitsamt den damit verbundenen Sicherheitsrisiken.

Angesichts der Tatsache, dass Eingabevalidierung bzw. IPC ein potentieller Angriffsvektor bei Applikationen ist, dürften beide Plattformen in etwa ähnliche Konzepte dafür umsetzen. So ist es weder bei Android, noch iOS möglich, festzulegen, a priori schon einzuschränken, dass nur eine gewisse Applikation für ein URL-Schema zuständig sein darf oder, dass der Absender nur spezifisches Applikationspaket ist. In beiden Fällen obliegt es EntwicklerInnen von Apps die entsprechenden Überprüfungen durchzuführen. Die Konsequenzen im Problemfall sind gleich.

5. WebViews

Applikationen unter Android und iOS können WebViews verwenden, um die Browser-Funktionalität in die eigene Anwendung einzubetten. Webseiten können so ohne Umwege über einen Browser direkt dargestellt. Dies schließt alle Eigenschaften ein, die sich auch im Browser finden. WebViews können Cookies setzen, CSS rendern und JavaScript-Code ausführen. Damit einhergehend sind auch Angriffsrisiken, denen man in jeder Art von Desktop- und Mobilbrowser ausgesetzt ist, wie etwa Cross-Site Scripting (XSS) oder Injection von bösartigen Scripts. WebViews sind zwar eingebettet in einer abgesicherten Umgebung (Sandbox), fungieren aber dennoch als Teil der nativen Android- oder iOS-App, die sie aufruft und steuert. Ebenso ist es möglich, dass eine App weitere Funktionalität bereitstellt, die aus der WebView heraus aufgerufen werden kann. Eines der prominentesten Beispiele hierfür ist die Plattform Apache Cordova[17], mit der Mobilanwendungen auf Basis von Webseiten entwickelt werden können. Aus der abgesicherten Umgebung kann es daher auch einen „Rückkanal“ geben d.h. die WebView Aktionen innerhalb der beinhaltenden Anwendung aufrufen.

Gleich wie reguläre Browser können native Anwendungen WebViews instruieren, Webseiten von einem Server (https-Schema) oder dem lokalen Dateisystem zu laden (file-Schema). Wenn Inhalte aus dem Dateisystem geladen werden, sollte es BenutzerInnen nicht möglich sein, den Pfad oder den Inhalt der aufgerufenen Dateien zu bearbeiten. Andernfalls könnte u.U. ein Angreifer dieselbe Methode nutzen, um manipulierte Inhalte im WebView anzuzeigen, eigene Inhalte bzw. Scripts einzubinden und Eingabedaten auszulesen. Durch die Tatsache, dass WebViews transparent in Apps eingebunden werden, sind sie nicht immer als solche erkennbar. Beispielsweise gibt es keine Anzeige, ob die dargestellte Webseite über einen sicheren Übertragungskanal (TLS) abgerufen wurde oder um welche URL sich handelt (Adressleiste fehlt). Umso essentieller ist es daher, dass Apps unter Android und iOS gewisse sicherheitsrelevante Eigenschaften bei der Darstellung von Webseiten sicherstellen:

- Verwendung von JavaScript nur, wenn tatsächlich benötigt. Durch Deaktivieren können Cross-Site-Script und Script Injection verhindert werden.
- Verhindern, dass WebViews neue Fenster oder Popups öffnen können. Dies ist insbesondere gängige Praxis bei Werbung, die sich dann über andere Fenster legt.
- Beschränkung des Kontexts von URL-Schemata. Wenn Webseiten nur aus lokaler oder entfernter Quelle angezeigt werden sollen, so sollen die entsprechenden Webseiten-URLs unabänderlich festgelegt werden.
- Verifizierung von Serverzertifikaten mithilfe der Vertrauensanker, die im jeweiligen Betriebssystem hinterlegt sind.

[16] https://www.objc.io/issues/14-mac/xpc/#security-attributes-of-the-connection
[17] https://cordova.apache.org/

5.1. Android

Bei der programmatischen Erstellung einer WebView-Komponente sieht Android vor, dass weitere Optionen explizit hinterlegt werden. Dazu gehört beispielsweise die Einstellung, ob JavaScript erlaubt sein soll oder nicht. Standardmäßig ist dies nicht der Fall. De facto benötigt aber ein Gros der Webseiten aktiviertes JavaScript zur korrekten Darstellung. Bei Mobilanwendungen kommt hinzu, dass sie oft ein natives Plattformlayout mittels JQuery Mobile[18] imitieren und darum auf JavaScript angewiesen sind.

```
WebView webview = new WebView(this);
setContentView(webview);
webview.loadUrl("https://www.a-sit.at/");
webview.getSettings().setJavaScriptEnabled(true);
```

Durch zusätzliche Optionen kann eingeschränkt werden, von welchen Quellen Seiten geladen werden können. Konkret kann z.B. durch die Methoden setAllowFileAccessFromFileURLs(), setAllowFileAccess() und setAllowUniversalAccessFromFileURL() eingeschränkt werden, ob Seiten aus lokalen Quellen geladen und JavaScript in ihnen ausgeführt werden kann oder nicht. Dies impliziert auch die Funktionalität, dass Webseiten mithilfe von JavaScript Dateien einer Anwendung lesen, erstellen und bearbeiten können. Die vorgegebene Sandbox-Umgebung wird somit verlassen. Diese Einstellungen sind standardmäßig aktiv und sollten deaktiviert werden, sofern nicht benötigt.

WebView-Komponenten verwenden die Vertrauensanker, die in der Plattform hinterlegt sind zur Verifikation von Zertifikaten. Wird also mit einem entfernten Server eine TLS-Verbindung aufgebaut, der kein gültiges Zertifikat präsentiert, erscheint eine Fehlermeldung, die es BenutzerInnen ermöglicht, manuell die Fortsetzung der Verbindung zu bestätigen oder abzubrechen. Durch die Methode onReceivedSslError() der Klasse WebViewClient wird Android-Apps theoretisch die Möglichkeit gegeben, die Fehlermeldung des ungültiges Zertifikats im eigenen Stil visuell anzuzeigen. In der Praxis sehr oft zu beobachten ist jedoch, dass bei Aufruf von onReceivedSslError() in dieser Methode einfach nichts passiert d.h. sie von EntwicklerInnen bewusst „blind“ geschalten wird. Praktisch bedeutet das, dass ein TLS-bezogenes Problem folglich nicht mehr berichtet, sondern unterdrückt wird. Dadurch, dass die WebView-Komponente keine Adressleiste anzeigt, können BenutzerInnen folglich nicht erkennen, ob die dargestellte Webseite über einen sicheren Übertragungskanal geladen wurde.

5.2. iOS

Ähnlich wie bei Android ist es mit iOS möglich, dass WebViews Endpunkte aus dem lokalen Verzeichnis einer Applikation laden oder von einem entfernten Server. Die API ist dabei in wesentlichen Aspekten ähnlich zu Android. Gleich wie dort gibt es mehrere Methoden, um den Zugriff auf Ressourcen zu verwalten: allowUniversalAccessFromFileURLs(), allowFileAccessFromFileURLs() und allowingReadAccessToURL(). Standardmäßig sind diese Optionen deaktiviert d.h. die Basiseinstellungen sind maximal konservativ. Eine wesentliche Ausnahme dazu ist die Unterstützung von JavaScript, die von Haus aus gegeben ist und auch nicht deaktiviert werden kann.

5.3. Diskussion

Angesichts der Tatsache, dass WebViews Inhalte aus entfernen Quellen anzeigen können, spielt die Überprüfung, welche Inhalte es konkret sind, eine besondere Rolle. Beide Plattformen bringen ähnliche Methoden mit, um zu regulieren, ob lokale Dateien als Webseiten dargestellt werden dürfen.

Wesentliche Unterschiede in beiden Plattformen zeigen sich aber bei der Behandlung von JavaScript und dem Umgang mit Fehlermeldungen in TLS-Verbindungen. Wohingegen bei Android die Unterstützung für JavaScript explizit aktiviert werden muss, ist sie bei iOS von Haus aus gegeben. Bei Android können EntwicklerInnen Fehlermeldungen in Bezug auf TLS unterdrücken, was erfolgreiche Man-in-the-Middle (MITM)-Angriffe wesentlich begünstigt.

[18] https://jquerymobile.com/

6. Netzwerkkommunikation

Die Verwendung von TLS zur geschützten Übertragung von Informationen ist heutzutage Standard. Mobilanwendungen für Android und iOS können plattformgegebene Schnittstellen verwenden, um Verbindungen zu beliebigen Servern aufzubauen. In den Standardeinstellungen sind die dabei verwendeten Einstellungen vernünftig. Das Betriebssystem übernimmt in seiner Rolle als Mittler der Verbindung die Aushandlung der Verbindung und verifiziert die Gültigkeit des vom Server präsentierten Zertifikats. Um die Vertrauenskette zu prüfen, wird dabei auf eine Liste von Vertrauensanker zugegriffen, die im Betriebssystem fest hinterlegt sind. Weder unter Android, noch unter iOS besteht daher a priori kein Risiko, dass Man-in-the-Middle-Angriffe erfolgreich sind.

6.1. Android

In ähnlicher Situation wie bei TLS-Verbindungen mithilfe von WebViews (siehe Abschnitt 5.1), können Anwendungen unter Android eigene Funktionen implementieren, die eine Zertifikatsprüfung implementieren. Die vom System eigentlich vorgenommene Überprüfung kann somit ausgeschaltet werden, wenn etwa die Methode checkServerTrusted() in einem TrustManager-Objekt überschrieben wird. Apps können so theoretisch die vom System vorgenommenen Überprüfungen um eigene Routinen, z.B. um zu verifizieren, ob gewisse Erweiterungen im Zertifikat vorkommen, oder das Zertifikat gewisse Attribute enthält.

In der Praxis ist sehr oft zu beobachten, dass Produktivversionen von Apps eine eigene Implementierung eines TrustManagers inkludieren, bei dem die Methode checkServerTrusted() leer, also funktionslos, ist. Die Gültigkeitsprüfung wird dadurch umgangen und eine App akzeptiert prinzipiell jedes Serverzertifikat, unabhängig davon ob es von einem offiziell akkreditierten Zertifikatsaussteller unterschrieben wurde oder nicht.

Im Bewusstsein dieses Missbrauchspotentials gibt es seitens Android eine vergleichsweise sehr umfangreiche Dokumentation[19], die mithilfe von Codebeispielen aufzeigt, wie eine Prüfung vorgenommen werden soll und welche Probleme dabei häufig auftreten.

6.2. iOS

Unter iOS sorgt der Dienst „App Transport Security“ (ATS)[20] dafür, dass das Betriebssystem sämtliche Verbindungen prüft, die über die Schnittstellen NSURLConnection, NSURLSession und CFURL zu Domains aufgebaut werden. ATS ist grundsätzlich aktiv und fordert z.B. die Verwendung gewisser Cipher suites, einen SHA256-Fingerabdruck bei Zertifikaten und mindestens TLS 1.2.

Wesentlich ist, dass ATS nur dann aktiv ist, wenn Apps Verbindungen zu offiziell registrieren Domainnamen aufbauen. Bei Verbindungen zu IP-Adressen und Hostnamen im lokalen Netz findet kein Schutz durch ATS statt. Unzweifelhaft ergibt sich hieraus ein wesentliches Sicherheitsproblem, das für BenutzerInnen nicht erkennbar ist. In der Praxis sind die Auswirkungen überschaubar, wenn man bedenkt, dass fast alle im Internet erreichbaren Rechner ohnehin „Server Name Indication“ (SNI) verwenden, um mehrere Domainnames unter der gleichen IP-Adresse erreichbar zu machen. Dies reduziert das Risiko, dass EntwicklerInnen möglicherweise versehentlich eine IP-Adresse anstelle des Domainnamens angeben.

6.3. Diskussion

Sowohl Android als auch iOS behalten bei der Prüfung von Serverzertifikaten die Verantwortung innerhalb der Plattform. Unter Android ist es jedoch möglich, dieses Standardverhalten zu überschreiben und dadurch im schlimmsten Fall die Zertifikatsprüfung zu deaktivieren. Dies basiert jedoch auf der bewussten Entscheidung von EntwicklerInnen. In den Standardeinstellungen begünstigt keine der beiden Plattform ein erhöhtes Risiko für erfolgreiche Man-in-the-Middle Angriffe.

[19] https://developer.android.com/training/articles/security-ssl

[20] https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html

7. Falsche Verwendung von Kryptographie-APIs

Android und iOS stellen Schnittstellen bereit, um die Komplexität kryptographischer Operationen zu abstrahieren. Apps können diese APIs nutzen, um beliebige Daten zu ver- und entschlüsseln, Schlüsselmaterial aus Passwörtern abzuleiten (Key derivation), sowie sichere Zufallszahlen zu generieren. Damit diese Operationen „sicher“ funktionieren, müssen EntwicklerInnen einige Regeln beachten, wie z.B. keinen Electronic Code Book (ECB) Modus für Blockchriffen zu verwenden, keine Schlüssel konstant und wiederauffindbar in Apps zu hinterlegen, sowie den „Zufall“ bei Zufallsgeneratoren nicht negativ zu beeinflussen. Werden diese Kriterien missachtet, kann es dazu führen, dass sensible Daten, die eigentlich verschlüsselt sind, nicht mehr ausreichend geschützt sind und im schlimmsten Fall ein Angreifer Kenntnis über vertrauliche Informationen erlangt.

Zu diesen, von der Plattform unabhängigen Regeln der sicheren Anwendung zählen etwa folgende:

- **Kein ECB-Modus für Verschlüsselung unter Einsatz von Blockchiffren**
  In diesem Modus werden Datenblöcke unabhängig voneinander verschlüsselt. Gleiche Blöcke von Nachrichten können somit zu gleichlautenden Blöcken von verschlüsseltem Text werden. Muster in verschlüsselten Daten sind folglich erkennbar und unter Umständen kann dadurch auch auf den verschlüsselten Inhalt rückgeschlossen werden.

- **Kein nicht-zufälliger Initialisierungsvektor für Verschlüsselung im CBC-Modus**
  Konstante oder vorhersagbare Initialisierungsvektoren (IV) führen zu einem deterministischen und zustandslosen Verschlüsselungssystem, das anfällig für sog. „Chosen Plaintext Attacks“ ist. Wird also der CBC-Modus eingesetzt, sollten EntwicklerInnen einen zufällig gewählten Initialisierungsvektor spezifizieren. Wird kein IV angegeben, wird eine Serie von NULL-Bytes als IV verwendet, was gleichwertig ist, wie ein konstanter IV.

- **Keine konstant in der App hinterlegten Schlüssel**
  Die Geheimhaltung von kryptographischen Schlüssel ist eine wesentliche Anforderung, um zu verhindern, dass Dritte auf vertrauliche Daten zugreifen können. Statisch definierte Schlüssel verletzen diese Regel und führen die Verschlüsselung ad absurdum.

- **Keine konstanten Salt-Werte bei der Ableitung von Schlüssel aus Passwörtern**
  Ein zufällig gewählter Salt-Wert stellt sicher, dass ein Schlüssel, der auf einem Passwort basiert, eindeutig ist und Brute-Force-Angriffe erschwert werden.

- **Nicht weniger als 1000 Iterationen bei der Ableitung von Schlüsseln**
  Eine niedrige Zahl an Iterationen bei der Ableitung von Schlüsseln reduziert die Kosten und Komplexität erfolgreicher Angriffe auf Schlüssel, die von Passwörtern abgeleitet werden.

7.1. Android

Android orientiert sich an der „Java Cryptography Architecture“ (JCA) und stellt über mehrere nativ implementierte Provider verschiedene Algorithmen zur Verfügung. Wie in Java-Anwendungen wird dabei üblicherweise ein Objekt der Klasse „Cipher“ unter Angabe des gewünschten Algorithmus instanziiert. Darüber hinaus können der zu verwendende Modus und ein Padding angegeben werden. Wird hierbei bewusst ECB spezifiziert, z.B. durch Angabe von „AES/ECB/PKCS5Padding“, ist die zuvor angeführte Regel verletzt. Alternativ dazu ist es möglich, nur den Algorithmus zu nennen, woraufhin für Modus und Padding Standardwerte gewählt werden. Im Falle von Blockchiffren wie AES oder Twofish wählt JCA automatisch den ECB-Modus und verletzt somit die genannte Regel. In der Praxis ist sehr häufig zu beobachten, dass nur der Algorithmus ohne zusätzlichen Modus angegeben wird. Dies führt implizit zur Verletzung der Regel, dass der ECB-Modus nicht für Verschlüsselung unter Einsatz von Blockchiffren verwendet werden sollte.

Die Klasse „Cipher“ nimmt als Parameter ein Objekt IVParameterSpec, das einen IV spezifiziert:

```
// Wrong: Constant IV
byte[] staticIv = new byte[] { 0x0f , 0x01 , 0x02 , 0x03 , 0x04 , 0x02 , 0x01 };
IvParameterSpec ivParameterSpec = new IvParameterSpec(staticIv);
```

```
Cipher cipher1 = Cipher.getInstance("AES/CBC/PKCS7Padding");
cipher1.init(Cipher.ENCRYPT_MODE, key, ivParameterSpec);

// Correct approach
Cipher cipher2 = Cipher.getInstance("AES/CBC/PKCS7Padding");
cipher2.init(Cipher.ENCRYPT_MODE, key);
byte[] randomIv = cipher2.getIV();
```

Wie im Codefragment darstellt, kann über IVParameterSpec theoretisch ein konstanter IV übergeben werden. Dies würde die zweitgenannte Sicherheitsregel verletzten. Der korrekte Ansatz hingegen wäre, den IV von der Plattform über einen sicheren Zufallsgenerator zu erstellen. Geben EntwicklerInnen keinen IV an, übernimmt die Plattform die Generierung.

Analog zum IV können EntwicklerInnen einen Schlüssel konstant in der App hinterlegen, der die zuvor spezifizierte Regel verletzen würde:

```
// Hard-coded key
SecretKey secretKey1 = new SecretKeySpec("secretkeywithfictive16bytes".getBytes(), "AES");

// Better approach
KeyGenerator kg = KeyGenerator.getInstance("AES", "BC");
kg.init(256);
SecretKey secretKey2 = kg.generateKey();
```

Die sichere Alternative wäre die dynamische Generierung eines Schlüssels in der gewünschten Stärke, wofür unter Java/Android z.B. die Methoden der Klasse KeyGenerator verwendet werden können. Dass die Möglichkeit besteht, einen Schlüssel selbst anzugeben, ist nicht per problematisch. Für die Entschlüsselung von zuvor chiffrierten Inhalten würde sich der KeyGenerator nicht eignen. Die Verantwortung obliegt hierbei dennoch EntwicklerInnen, den zu verwendenden Schlüssel aus sicheren Quellen herzuleiten und nicht konstant zu hinterlegen.

Ein weiterer Regelverstoß wäre gegeben, wenn EntwicklerInnen bei der Ableitung von Schlüsseln aus Passwörtern den zu verwendenden Salt-Wert statisch hinterlegen:

```
char[] password = "secretkey".toCharArray();
KeySpec keySpec = new PBEKeySpec(password , "salt".getBytes(), 123, 256);
```

Ebenso sehr häufig zu beobachten ist ein Verstoß der zuletzt angeführten Regel, die besagt, dass mindestens 1000 Iterationen für Schlüsselableitung (KDF) verwendet werden sollen. Diese seit 2001 genannte und in RFC 8018[21] im Jänner 2017 erneut formulierte „Untergrenze“, soll die Kosten für Brute-Force-Angriffe hochhalten. Da sich seit erstmaliger Nennung von 1000 Iterationen die Kosten und Performance von Rechenleistung für Brute-Force-Angriffe drastisch reduziert hat, würde die Verwendung von deutlich mehr Iterationen Angriffe entsprechend verteuern. Auch in diesem Fall gibt es bei Android keinen systembedingten unteren Grenzwert. Die Verantwortung über die sichere Verwendung von Schlüsselableitung obliegt somit alleinig EntwicklerInnen.

Die von Android für die Verwendung von kryptographischen Algorithmen bereitstellte Dokumentation[22] bzw. die Beschreibung einzelner Klassen[23] sparen Informationen über falsche Verwendung aus. Da das von Android verwendete Fundament die JCA ist, könnten als Referenz für EntwicklerInnen auch vergleichbare Dokumentationen zur Verwendung herangezogen werden. Bedauerlicherweise finden sich jedoch auch dort[24] keine Hinweise, die auf die Konsequenzen falscher Verwendung hinweisen.

[21] https://tools.ietf.org/html/rfc8018
[22] https://developer.android.com/guide/topics/security/cryptography
[23] https://developer.android.com/reference/javax/crypto/spec/PBEKeySpec
[24] https://docs.oracle.com/javase/7/docs/api/javax/crypto/spec/PBEKeySpec.html

7.2. iOS

Das für Anwendungen unter iOS offiziell bereitgestellte Rahmenwerk heißt CommonCrypto[25]. Über die Verwendung von Methoden wie CCCrypt() oder CCCryptorCreate() können Apps die Plattform unter Angabe entsprechender Parameter anweisen, gewisse kryptographische Algorithmen, z.B. für Verschlüsselung, auf Daten anzuwenden. Bei Blockchiffren kommt dabei standardmäßig der CBC-Modus zum Einsatz. Die Verwendung des unsicheren ECB-Modus, müsste durch die Option kCCOptionECBMode manuell bzw. von EntwicklerInnen bewusst erfordert werden.

Ein IV zur Verwendung mit CCCrypto(), CCCryptorCreate() oder CCCryptorCreateWithMode() muss zuvor über einen kryptographisch sicheren Zufallszahlengenerator erstellt worden sein. Dies kann entweder CCRandomGeneateBytes() in CommonCrypto sein oder SecRandomCopyBytes() in der Security-Bibliothek. In der Praxis kann beobachtet werden, dass der Parameter mit dem IV einfach ausgelassen wird – was implizit dazu führt, dass ein NULL-IV (nur NULL-Bytes im IV) generiert wird und die zuvor angeführte Regel verletzt.

Beim Einsatz von kryptographischem Schlüsselmaterial mit CCrypto, CCCryptorCreate() oder CCCryptorCreateWithMode() müssen EntwicklerInnen darauf achten, dass verwendet wird, darf nicht konstant hinterlegt sein. Dies gilt ebenso für die Angabe von Salt-Werten bei der Verwendung von CCKeyDerivationPBKDF() zur Schlüsselableitung.

Für die Zahl der Iterationen, die für die Schlüsselableitung mit CCKeyDerivationPBKDF() angegeben wird, gibt es von der Plattform her keine Untergrenze. Die Angabe einer Zahl, die die Kosten für Angriffe möglichst hochhält, obliegt somit der Verantwortung von EntwicklerInnen.

Die von Apple bereitgestellte Dokumentation[26] ist vergleichsweise abstrakt und geht nicht auf die sichere Verwendung von CommonCrypto ein. Erklärt werden hingegen prinzipielle Begriffe von Kryptographie und deren Ablauf.

7.3. Diskussion

Android und iOS stellen jeweils unterschiedliche Frameworks für die Anwendung kryptographischer Algorithmen bereit. Die gegebenen Werkzeuge schützen in keinem Fall vor der Verwendung von Parametern, die gemeinhin als unsicher gelten. Die Dokumentation der Plattformen beschränkt sich auf die Erklärung der jeweiligen Schnittstellen, klammert jedoch Informationen über die sichere Verwendung aus. EntwicklerInnen sind somit auf Quellen Dritter angewiesen, um zu erfahren, welche Parameter in der Verwendung als sicher gelten und welche Konsequenzen auftreten, wenn z.B. der IV nicht spezifiziert wird.

Wesentliche Unterschiede in den Plattformen finden sich in den Standardwerten, die für die Verschlüsselung mit Blockchiffren verwendet werden. Bei Android kommt hier der unsichere ECB-Modus zum Einsatz, bei iOS das unbedenkliche CBC. Analog dazu kommt man bei Android nicht darum umhin, einen IV zu spezifizieren oder nach der Verschlüsselung einen generierten zu erhalten. Bei iOS wiederum wird implizit ein IV bestehend aus NULL-Bytes angenommen, sofern kein Wert explizit definiert wird.

Die Wahl der Anzahl der Iterationen bei der Ableitung von kryptographischen Schlüsseln aus Passwörtern, sowie die Angabe des Salt-Werts ist in beiden Betriebssystemen frei möglich. Ebenso ist es bei beiden Plattformen möglich, standardmäßig sicher geltende Werte zu ändern, wenn entsprechende Optionen für ECB-Modus oder dem IV angegeben werden. A priori ist daher kein Schutz gegeben, um eine Verletzung der in Abschnitt 7 eingangs angeführten Regeln zu verhindern. Die Verantwortung über die korrekte Implementierung liegt somit bei EntwicklerInnen.

---

[25] https://opensource.apple.com/source/CommonCrypto/CommonCrypto-36064/CommonCrypto/CommonCryptor.h

[26] https://developer.apple.com/library/archive/documentation/Security/Conceptual/cryptoservices/Introduction/Introduction.html

8. Fazit

Im Zuge dieses Projekts wurde der Fokus auf die Sicherheit von Anwendungen für Android und iOS gelegt. Es sollte untersucht werden, ob eine der Plattformen Sicherheitsprobleme in Apps womöglich begünstigt oder durch Schutzmechanismen Fehler von EntwicklerInnen erkennen kann. Hierfür wurden zuerst die sicherheitsrelevanten Eigenschaften der Plattformen von jenen in Apps abgegrenzt. Trotz der Heterogenität der Plattformen und Schnittstellen hat sich herausgestellt, dass es Angriffsvektoren gibt, die prinzipiell beide Welten betreffen.

Beim Vergleich von in der Praxis häufig auftretenden Sicherheitsproblemen war erkennbar, dass die unterschiedlichen Konzepte in Android und iOS implizit auch zu individuellen Problemen führen, die die Plattform verursacht. In den meisten Fällen wurde die Verantwortung über die korrekte Bedienung von Schnittstellen jedoch an EntwicklerInnen abgetreten. Im Falle von WebViews hat sich beispielsweise gezeigt, dass beide Plattformen in etwa die gleichen Risiken für falsche Verwendung bergen. In den Bereichen Netzwerkkommunikation und der Verwendung kryptographischer APIs war erkennbar, dass Android und iOS jeweils eigene Werkzeuge bereitstellen, um die Funktionalität nach Wünschen von EntwicklerInnen in Apps einzusetzen. Eine unsichere Verwendung wurde in keinem Fall technisch unterbunden. Darüber hinaus konnte festgestellt werden, dass die offiziellen Dokumente für Android und iOS zwar die Funktionalität ihrer Schnittstellen erklären, jedoch keine Akzente auf sichere Verwendung legen. Dies kann Situationen begünstigen, bei denen die unsichere Implementierung auf einen Mangel an Kenntnis sicherer Praktiken zurückzuführen ist.

Die Erkenntnisse aus diesem Projekt zeigen, dass, abgesehen von den Spezifika der einzelnen Plattformen, sowohl Android als auch iOS einen Nährboden für sicher implementierte Apps darstellen. Die Sicherstellung der korrekten Wahl von Parametern für Schnittstellen und die Notwendigkeit, über sicherheitsrelevante Aspekte Bescheid wissen zu müssen, obliegt in jedem Fall EntwicklerInnen der jeweiligen Anwendung.