Alte Lücken in neuen Kleidern

Mobile (Un-)Sicherheit

Markus Maria Miedaner

Smartphones, Tablets und mobile Anwendungen nehmen immer mehr Raum in unserem Alltag und unserer Arbeitswelt ein. Sicherheitsprüfungen von Apps zeigen: Betriebssysteme wie Android oder iOS liefern zwar einen soliden Sicherheitsstandard, er kann jedoch durch Schwachstellen in Apps umgangen werden. Diese Situation ist aus der Webapplikationswelt hinreichend bekannt und auch die damit einhergehenden Sicherheitslücken. Doch was bedeutet dies für die berufliche und private Nutzung derselben mobilen Geräte (Bring Your Own Device)?

Android und iOS beherrschen den Smartphone-Markt. Ihre Software Development Kits (SKDs) sind frei verfügbar, die darauf aufbauenden Applikationen meist nicht. Allein im Jahr 2012 wurden über 45 Milliarden Apps heruntergeladen [Gartner]. Eine derart hohe Nachfrage weckt natürlich auch das Interesse von Kriminellen. Wie in der Webapplikationswelt haben wir bei mobilen Applikationen den Stand erreicht, dass Sicherheitslücken in den Apps gravierender sind als in den darunter liegenden Betriebssystemen.

Welchen Schutz bieten Smartphones?

Ein Smartphone besteht aus mehreren kleinen „Computern" für Bluetooth, WIFI, Base Band, Near Field Communication (NFC) usw. Diese Mikrocontroller besitzen jeweils ein eigenes Betriebssystem und sind über vertrauenswürdige Kanäle verbunden. Hier stellt sich die Frage, wie die Integrität dieser Komponenten über die Zeit sichergestellt wird. Die hohe Vertrauensstellung, die die Komponenten genießen, ist durchaus skeptisch zu sehen. Der Angreifer hat hier leichtes Spiel, sobald er eine kontrollieren kann.

Neben den Mikrocontrollern besitzt das Smartphone ein alles koordinierendes Betriebssystem. Dieses schottet eine Applikation mit sogenannten „Sandbox"-Technologien ab und kontrolliert die Kommunikation mit anderen Apps und Hardwarekomponenten. In den letzten Versionen mobiler Betriebssysteme hielten Schutzmaßnahmen, wie *A*ddress *S*pace *L*ayout *R*andomization (ASLR) und *D*ata *E*xecution *P*revention (DEP), Einzug. Unter Windows, Mac oder Linux sind diese Technologien bereits seit Langem im Einsatz.

Android erlaubt verschiedene Architekturen, um eine Applikation zu entwickeln:

- Diese kann nativ mit dem Native Development Kit in C/C++ und gegebenenfalls JavaScript geschrieben werden.
- Das klassische Android-SDK beruht auf Java und nutzt seine eigene Runtime (Dalvik).
- Als dritte Möglichkeit kann der Entwickler auf HTML5 zurückgreifen und seine Applikation in einer abgespeckten Browserumgebung – der WebView – starten.

Klassische Schwachstellen sind seit Jahren bekannt

Das *O*pen *W*eb *A*pplication *S*ecurity *P*roject (OWASP) stellte dieses Jahr erneut eine TOP 10-Liste mit mobilen Sicherheitslücken zusammen [OWASP_M]. Interessanterweise kennen wir genau solche Lücken schon seit Jahren von Webapplikationen und Cloud-Lösungen.

Also – alte Lücken in neuen Kleidern? Tabelle 1 gibt einen Überblick über bekannte Lücken. Die nächsten Absätze beleuchten die einzelnen Schwachstellen genauer und unterlegen sie mit Beispielen.

M1	Weak Server Side Controls
M2	Insecure Data Storage
M3	Insufficient Transport Layer Protection
M4	Unintended Data Leakage
M5	Poor Authentication and Authorization
M6	Broken Cryptography
M7	Client Side Injection
M8	Security Decision Via Untrusted Input
M9	Improper Session Handling
M10	Lack Of Binary Protections

Tabelle 1: OWASP Mobile Top 10

M1 – Weak Server Side Controls

Apps kommunizieren häufig mit einer Backend-/Serverseite. Diese stellt einen Teil der Angriffsoberfläche dar. Sie ist in diversen Abhandlungen über sichere Webapplikationen und Cloud-Lösungen beschrieben worden, sodass ich hier nur auf die OWASP-TOP 10 für Webapplikationen [OWASP_T] verweise.

M2 – Insecure Data Storage

Die sichere Ablage von Daten auf Mobiltelefonen ist nicht einfach zu lösen.

Gängige Praxis ist immer noch die Ablage im Dateisystem ohne weiteren Schutz. Prekär wird dies, wenn der Entwickler sich für Temp-Verzeichnisse oder andere weltweit lesbare Bereiche (Bsp.: SD-Karte) entscheidet. Diese Bereiche sind anders zu bewerten, wenn die Daten vor der Ablage verschlüsselt werden.

Kryptografische Ansätze werden selten gewählt [Elcom] und gern falsch implementiert [Ege13], obwohl es einfache Lösungen wie SQLCipher [SqlC] gibt. Letztere stellt eine kryptografisch solide gesicherte SQLite-Datenbank dar. SQLite ist der Standard auf mobilen Endgeräten. Eine verschlüsselte Datenbank stellt den Entwickler allerdings vor die Frage, wo er den Schlüssel ablegen soll. Das Smartphone selbst ist mit allen Speichermöglichkeiten hierfür sicher keine Lösung. Deshalb wird eine verschlüsselte Ablage von Daten häufig zugunsten der Usability verworfen. Welcher Benutzer möchte schon ein

langes Passwort eingeben, bevor er seine Applikation nutzen kann?

M3 – Insufficient Transport Layer Protection

Die Übertragung von Daten vom Mobiltelefon zum Server über einen unverschlüsselten Kanal ist nicht selten. Wird eine SSL-Verbindung (HTTPS) aufgebaut, finden sich immer wieder Apps, die das erhaltene Serverzertifikat gar nicht oder nicht ausreichend prüfen [Apple]. Somit wird einer „Man in the Middle"-Attacke im wahrsten Sinne des Wortes Tür und Tor geöffnet.

Öffentliche, kostenlose Internetzugänge sind geradezu prädestiniert für dieses Szenario: Der Angreifer fängt den Verbindungsaufbau zum Server ab und präsentiert der Applikation stattdessen sein eigenes SSL-Zertifikat. Parallel dazu baut er eine Verbindung zum eigentlichen Zielserver auf. Dadurch ist sowohl die Übertragung zwischen der mobilen Anwendung und dem Angreifer als auch zwischen dem Angreifer und dem Backendservice verschlüsselt. Auf dem Rechner des Angreifers hingegen ist die Kommunikation im Klartext ersichtlich.

Ein klassisches Codebeispiel für eine SSL-Verbindung ohne Validierung des Zertifikats findet sich in Listing 1. Die Methoden des `TrustManager`s prüfen nicht, welches oder ob sie ein Zertifikat erhalten.

```
TrustManager tm = new X509TrustManager() {
  @Override
  public void checkClientTrusted(X509Certificate[] chain,
                                 String authType)
      throws CertificateException {
    // do nothing here
  }
  @Override
  public void checkServerTrusted(X509Certificate[] chain,
                                 String authType)
      throws CertificateException {
    // do nothing here
  }
  @Override
  public X509Certificate[] getAcceptedIssuers() {
    return null;
  }
};
sslContext.init(null, new TrustManager[] { tm }, null);
```

Listing 1: Codebeispiel – fehlende SSL-Validierung

M4 – Unintended Data Leakage

Logdateien enthalten häufig wichtige oder gar personenbezogene Daten, wie beispielsweise den Benutzernamen und dessen AccessTokens. Bis Android, Version „JellyBean 4.1" war es Applikationen möglich, die Logdateien aller anderen Applikationen zu lesen.

Das ist problematisch, denn viele Applikationen sammeln zu viele Daten über ihre Nutzer. Glücklicherweise findet man immer häufiger sogenannte „Privacy Settings". Diese sind jedoch gerne so tief innerhalb der Applikation versteckt, dass der normale Benutzer auf halber Strecke aufgibt zu suchen oder er erfährt erst gar nicht von ihnen. Zeichnet eine Applikation diese Informationen auf und überträgt sie auf einem unsicheren Kanal, kann die Privatsphäre der Benutzer schnell verloren gehen.

Einen weiteren Fallstrick bietet Android durch seine nachrichtenbasierte Kommunikationsmöglichkeit (genannt „Intent") zwischen einzelnen Applikationskomponenten. Diese können auch als Broadcast beziehungsweise an einen zu offenen Empfängerkreis verschickt werden. Das ermöglicht es einer bösartigen App, diese Nachricht zu lesen. Abhängig vom Inhalt kann Schadsoftware mehr oder weniger anrichten. Mehr dazu im Abschnitt „M8 – Security Decisions Via Untrusted Input".

M5 – Poor Authentication and Authorization

Authentifizierung und Autorisierung sind in vielen Applikationen nur rudimentär implementiert. Diesem Problem scheint durch die steigende Popularität von OAuth2.0 und ähnlichen Protokollen langsam Abhilfe geschaffen zu werden. Dennoch ist die Ablage von Zugangsdaten des Nutzers in einer lokalen Datenbank oder Datei – wenn möglich noch weltweit lesbar – keine Seltenheit, vergleiche Listing 2.

```
Schreiben der Daten
SharedPreferences preferences =
  getSharedPreferences("AUTHENTICATION_FILE_NAME",
                       Context.MODE_WORLD_WRITEABLE);
SharedPreferences.Editor editor = preferences.edit();
editor.putString("Authentication_Id",userid.getText().toString());

Auslesen:
SharedPreferences prfs =
  getSharedPreferences("AUTHENTICATION_FILE_NAME",
                       Context.MODE_PRIVATE);
String Astatus = prfs.getString("Authentication_Id", "");
```

Listing 2: Unsichere Ablage von Daten

Bei Aufrufen von Backend-Funktionalitäten darf auch die Zuordnung zwischen Nutzer und Applikation beziehungsweise Endgerät nicht verloren gehen. Durch das Abändern der UserId können ansonsten die Daten anderer Kunden verändert werden. Bei der Analyse mobiler Applikationen stößt man nicht selten auf derartige Fehler.

M6 – Broken Cryptography

Viele Applikationen implementieren oder integrieren kryptografische Algorithmen fehlerhaft [Ege13]. Ebenso verwechseln manche Entwickler ein Encoding wie Base64 mit einer soliden Verschlüsselung. Dabei beinhaltet das Android-SDK alle bekannten kryptografischen Bibliotheken, die man aus der Java-Welt kennt. Doch anscheinend reicht dies nicht aus, denn unsichere Algorithmen wie RC2, MD4, MD5 oder SHA1 werden weiterhin bevorzugt.

M7 – Client Side Injection

Applikationen validieren selten vollständig die Eingaben ihrer Benutzer. Dies ist ein gängiges Einfallstor für Injektionsangriffe. An Stelle eines Wortes kann auch Code eingetippt werden, der umgehend ausgeführt wird. Bildet die Applikation ihre Datenbankabfragen durch Konkatenation von Strings anstatt mit parametrisierten Abfragen, kann ein Angreifer eigene Abfragen ausführen lassen. Listing 3 zeigt dies exemplarisch.

```
public String getItemById(String itemId){
  SQLiteDatabase database =
    SQLiteDatabase.openOrCreateDatabase(databaseFile);
  String query = "SELECT * FROM item WHERE Id = " + itemId;
  ResultSet rs = database.execSQL(
  ...
```

Listing 3: Input-Validierung und SQL-Injection

Auch die Eingabe von HTML- und Javascript-Code anstelle eines geforderten Benutzernamens, Kommentars oder ähnlicher Texteingaben ist geläufig. Dies kann in WebViews dazu führen, dass Funktionalitäten im Backendservice ohne Zustimmung des Benutzers ausgelöst werden. Bietet die Applikation dem Nutzer die Möglichkeit, Dateinamen anzugeben, und validiert diese nicht, können damit durchaus Dateien, die nicht im Fokus des Entwicklers waren, erreicht werden.

M8 – Security Decisions Via Untrusted Inputs
Android benutzt „Intents“ zur Übertragung von Nachrichten zwischen den Applikationskomponenten. Ist der Empfängerkreis zu offen gehalten, kann eine andere App diese Kommunikation abfangen. Einmal im Besitz dieser Informationen können sie über diverse Kanäle aus dem Telefon heraus transportiert und aggregiert werden.

Eine bösartige Applikation kann diesen Weg ebenfalls nutzen, um eine entsprechende Nachricht zu schicken. Da der Angreifer den Inhalt des Intents bestimmt, kann er das Verhalten der betroffenen Applikation fernsteuern.

M9 – Improper Session Handling
Session Handling wird sowohl innerhalb der App als auch auf Seite des Backendservice implementiert. Werden keine Session-Timeouts oder nur ein einseitiger Logout-Mechanismus auf Seite der Applikation realisiert, kann ein Angreifer im Namen des Opfers die Backendservices weiter nutzen. Dazu braucht er nur den Netzwerkverkehr mitzuschneiden und daraus die Authentifizierungsdaten zu extrahieren.

Autorisierungstoken werden häufig sehr einfach erzeugt (z. B. Base64-Encodierung der Rechte und der UserId). Dies ist alles andere als fälschungssicher, was der Header eines Http-Requests zum Backendservice in Listing 4 zeigt.

```
Authorization: VUlEPTEwMDEwMTAwMDI7Um9sZT1QcmVtaXVt
Base64 decodiert:
Authorization: UID=1001010002;Role=Premium
```

Listing 4: Unsicheres Session-Management

An dieser Stelle fehlt auch jegliche Information über den Absender. Damit ist der Angreifer nicht mehr an ein Smartphone gebunden, sobald er den Mechanismus verstanden hat.

M10 – Lack Of Binary Protections
Die Authentizität eines Binärpakets zu prüfen, ist eine Herausforderung für den normalen Nutzer. Er ist kaum in der Lage, die Signatur eines legitimen Entwicklers von der Signatur eines Hackers zu unterscheiden. Besonders prekär ist diese Situation für Android, da es neben dem GooglePlayStore weitere Marketplaces gibt, deren Betreiber unterschiedlichen Philosophien folgen. Ein Angreifer kann durchaus eine bekannte App entpacken und sie mit Malware gebündelt erneut in einem solchen Marketplace veröffentlichen. Apple ist hier mit seinem erzwungenen CodeSigning und einem zentralen AppStore einen großen Schritt weiter.

Wie kann man diese Lücken frühzeitig in der Entwicklung erkennen und beseitigen?

Source Code Analysis Tools liefern einen guten Beitrag zur statischen Code-Analyse. Derartige Werkzeuge betrachten allerdings weniger sicherheitsrelevante, sondern eher klassische Programmierfehler. Damit wird beispielsweise eine fehlerhafte Prüfung von SSL-Zertifikaten nicht gefunden, da sie keine Programmierfehler enthält.

Auf Sicherheit fokussierte Analyse-Tools wie beispielsweise von Firmen wie Checkmarx, Tennable (Nessus) oder HP (Fortify) haben sich diesen Markt bereits erschlossen. Der Einsatz solcher Tools ist in der Regel recht selten.

Ein Entwickler kann auf andere Weise mit Tools wie der BurpSuite oder ZAP bereits viel erreichen. Ebenso gibt es mehr und mehr Open-Source-Analyse-Werkzeuge. Der Kasten „Source Code Analysis Tools “ zeigt eine Auswahl an kostenpflichtigen und kostenlosen Analysetools. Manche der hier genannten Hilfsmittel bieten eine Schnittstelle zur Integration in Continuous Integration Server wie Atlassian Bamboo, Hudson oder Jenkins. Eine Sicherheitsprüfung des zu erstellenden Produkts ist leicht automatisierbar und mit hoher Frequenz wiederholbar. Mit wenig Aufwand kann man Import-Skripte erstellen und die Ergebnisse der Sicherheitsprüfung in das vom Entwicklerteam genutzte Ticketsystem übertragen. Aber Achtung: Derartige Prüfwerkzeuge melden auch mal den ein oder anderen „falschen Treffer” (false positive).

Source Code Analysis Tools

HP Fortify – `https://www.fortify.com`
Nessus – `http://www.tenable.com/de`
Checkmarx – `http://www.checkmarx.com`
FlowDroid – `http://sseblog.ec-spride.de/tools/flowdroid/`
Santoku – `https://santoku-linux.com`
Yasca – `http://www.scovetta.com/yasca.html`
SciTools – `http://www.scitools.com/index.php`

Neben den technischen Möglichkeiten bieten neue Zertifizierungen wie der CSSLP von ISC² die Möglichkeit, Sicherheitslücken erst gar nicht entstehen zu lassen. Die Ausbildung dafür ist allerdings kostspielig. Weitere Möglichkeiten sind hier beispielsweise die OWASP-CheetSheet-Serie oder diverse Blogs, Foren und Mailinglisten.

BYOD – Hier kommt alles zusammen (Smartphones + Apps + Sicherheitslücken + Business Impact)

Viele Schwachstellen in mobilen Applikationen gewinnen erst richtig an Bedeutung, wenn das private Mobiltelefon dienstlich genutzt wird. Denn dann hat das Gerät Zugriff auf Geschäftsdaten und ist in das Firmennetz integriert. *Bring Your Own Device (BYOD)* hat in den letzten Jahren Einzug in viele Unternehmen gehalten – oft zum Leidwesen der CIOs und CSOs. So stellt die Verwendung dieser Geräte zu privaten und beruflichen Zwecken große Herausforderungen an die Firmenrichtlinien und deren Umsetzung. Zusätzlich erweitern sie die Angriffsoberfläche um ein Vielfaches. Jailbreaks werden mittlerweile so weit automatisiert, dass sie keine Kenntnisse über das Betriebssystem mehr erfordern. Damit bekommen Nutzer, aber auch Angreifer deutlich mehr Möglichkeiten, denn die Sicherheitsmaßnahmen des Betriebssystems wurden ausgehebelt.

Klassische Maßnahmen wie eine starke Netzwerksegmentierung sind häufig nicht im nötigen Umfang möglich, da diese die berufliche Nutzung der Geräte zu sehr einschränken.

Auf Netzwerkebene bieten Continuous-Monitoring-Lösungen eine gute Ergänzung zu bereits bestehenden Sicherheitslösungen. Sie zeigen deutlich, welche Geräte von Malware befallen sind oder sich abweichend von den Unternehmensrichtlinien verhalten.

Ein weiterer Schutz gegen Malware und bösartige Applikationen sind Remote Wipes. Durch das vollständige Zurücksetzen des Geräts werden diese entfernt. Marketplaces beziehungsweise ein AppStore ermöglichen des Weiteren eine schnelle und einfache Wiederherstellung der Applikationslandschaft auf dem Smartphone.

Mobile-Device-Management-Lösungen erlauben es, schnell zu erkennen, welches Gerät einen Jailbreak hinter sich hat. Man kann über diese ganz einfach den Softwarepool für Smart-

Bring Your Own Device – Checkliste

- Welche Geräte werden bereits eingesetzt?
- Wie hoch ist der aktuelle Grad der Kompromittierung? Bsp.: Jailbreaks
- Welche Nutzergruppen haben wir? Angestellte, Management, CEO?
- Welcher Schutzbedarf gilt für welche Geräte?
- Welche Software wird benötigt (VPN, Mail + PGP usw.)?
- Welches Mobile-Device-Management-Produkt wollen wir nutzen?
- Sind Remote Wipes in regelmäßigen Abständen für das gehobene Management möglich und akzeptabel?
- Schließt das firmeninterne Sensibilisierungsprogramm BYOD ein?
- Existiert ein Continuous Monitoring im Firmennetz?
- Wie soll sich das BOYD-Programm entwickeln?

phones regulieren. Ein zu stark einschränkender Rahmen gefährdet hierbei jedoch das BYOD-Programm.

Ebenso wie Laptops verfügen Smartphones mittlerweile über VPN-Clients, Mailprogramme und Speichermöglichkeiten für interne Dokumente und stellen damit auch das gleiche Risiko im Falle eines Verlustes dar. Eine vollständige Verschlüsselung aller Speichermedien ist bei diesen Geräten nicht nutzerfreundlich, da es diesen dazu zwingt, jedes Mal ein langes Passworts zum Entsperren des Geräts einzugeben. Wer will das schon.

Der Kasten „Bring Your Own Device – Checkliste“ listet wesentliche Punkte für die Prüfung eines bestehenden beziehungsweise zur Auswahl eines BYOD-Programms auf.

Fazit

Die sichere Entwicklung von mobilen Applikationen steckt noch immer in den Kinderschuhen. Dieser Artikel soll Ihnen den ein oder anderen Impuls geben, worauf bei der Programmierung von Apps zu achten ist. Viele dieser Punkte können behoben werden, bevor die Applikation den Marketplace oder gar Kunden erreicht. Setzt der Endkunde sein Smartphone bedacht ein – ist ein Bring Your Own Device einfach umsetzbar, ohne die Sicherheitsansprüche zu vernachlässigen.

Literatur und Links

[Apple] Apple Security Updates, `http://support.apple.com/kb/HT1222`

[Ege13] M. Egele, D. Brumley, Y. Fratanonio, C. Kruegel, An Empirical Study of Cryptographic Misuse in Android Applications, in: CCS'13 Proc. ACM SIGSAC Conference on Computer & Communications Security

[Elcom] A. Belenko, D. Skyarov, „Secure Password Managers“ and „Military-Grade Encryption“ on Smartphones: Oh, Really?, Elcomsoft Co. Ltd., 2012, `http://www.elcomsoft.com/WP/BH-EU-2012-WP.pdf`

[Gartner] Gartner Says Free Apps Will Account for Nearly 90 Percent of Total Mobile App Store Downloads in 2012, `http://www.gartner.com/newsroom/id/2153215`

[OWASP_M] `https://www.owasp.org/index.php/OWASP_Mobile_Security_Project`

[OWASP_T] `https://www.owasp.org/index.php/Top_10_2013`

[SqlC] `http://sqlcipher.net/`

Dr. Markus Maria Miedaner ist als IT-Architekt mit mehrjähriger Erfahrung für die SYRACOM Consulting AG tätig. Sein Schwerpunkt liegt auf Applikationssicherheit und Softwareentwicklungsmodellen. Er ist ein aktives Mitglied des Open Web Application Security Project.
E-Mail: markus.miedaner@syracom.de