25. August 2022 Studie · Mobile Sicherheit · 12 Min. Lesezeit

Erstellung eines iOS-Automationsframeworks zur Sicherheitsanalyse

Archiviert. Dieser Inhalt wird nicht mehr aktualisiert. Er bleibt zur Information verfügbar.

Stand 2022-07-31
Quelle https://technology2.a-sit.at/veroeffentlichungen/erstellung-eines-ios-automationsframeworks-zur-sicherheitsanalyse/

Durch automatisierte Analysetools konnten bereits zahlreiche Schwachstellen und Probleme in mobilen Betriebssystemen gefunden werden. Die bestehenden Analysetools konzentrieren sich jedoch hauptsächlich auf das Betriebssystem Android. In diesem Bericht wird deswegen erläutert, wie ein solches automatisiertes Analysetool auch unter iOS technisch umgesetzt werden kann. Dazu wurde ein Prototyp erstellt, welcher automatisiert Klassen erstellen, Methoden aufrufen und Properties auslesen kann. Im Bericht wird darauf eingegangen, welche Probleme spezifisch unter iOS gelöst werden mussten, um ein solches Analysetool zu erstellen. Vor allem Abstürze durch fehlerhafte Aufrufe machten es notwendig, die Vorgehensweise im Vergleich zu Analysetools unter Android zu adaptieren.

1. Einleitung - 1 2. Vorwissen - 2 2.1. Verbindung - 2 2.1.1. Simulator - 2 2.1.2. iOS-Gerät - 2 2.2. Verwandte Arbeiten - 3 3. Methodik - 3 3.1. Aufbau - 3 3.1.1. Kontrollapplikation - 4 3.1.2. Mobile Analyseapplikation - 5 4. Evaluierung - 6 5. Fazit - 7 1. Einleitung Bei der Analyse der Sicherheit von Betriebssystemen auf Smartphones haben sich automatisierte Analysetools als vorteilhaft erwiesen. So konnten mit solchen Tools Schwachstellen in der Programmierschnittstelle gefunden sowie die Fingerprintbarkeit eines Betriebssystems analysiert werden [1] [2]. Ein Großteil dieser Tools konzentriert sich jedoch auf das Betriebssystem Android. Dies liegt unter anderem an der größeren Offenheit des Betriebssystems, wodurch eine Automatisierung aufgrund der Dokumentation sowie der Einsehbarkeit des Quellcodes leichter umsetzbar ist. Unter iOS ist uns zum gegenwärtigen Zeitpunkt kein solches Analysetool bekannt. Da jedoch auch iOS eine große Nutzerbasis hat, wäre ein solches Analysetool von Vorteil um ähnliche Bedrohungen finden zu können. Android und iOS unterscheiden sich jedoch in der technischen Umsetzung, wodurch ein Konzept zur automatischen Analyse unter Android nicht 1:1 auf iOS umgelegt werden kann. In diesem Bericht wird dieses Problem angegangen. So wird erläutert, wie ein solches automatisiertes Analysetool technisch für iOS umgesetzt werden kann. Dazu wurde ein Prototyp erstellt, welcher automatisiert Instanzen von Klassen erstellen und diese verwenden kann. Mit diesen Instanzen können die Properties der Klassen

ausgelesen sowie Methoden aufgerufen werden. Ebenso wird im Bericht darauf eingegangen, wie sich die Automatisierung der Analyse im Vergleich zu Android verhält und welche Herausforderungen es zusätzlich gegeben hat. Abschließend gibt der Bericht eine Übersicht, wie viele Methoden aufgerufen und wie viele Properties erfolgreich ausgelesen werden konnten.

2. Vorwissen Der folgende Abschnitt gibt darüber Auskunft welche Tools verwendet wurden, um ein automatisches Analyseframework für iOS zu entwickeln. Weiters geht der darauffolgende Abschnitt auf verwandte Arbeiten ein.

2.1. Verbindung Zur automatischen Analyse von iOS-Geräten müssen bestimmte Schritte wie das Installieren und Starten der Analyseapplikation sowie das Auslesen der Resultate automatisiert werden. Dazu greift das Framework auf verschiedene Tools zurück. Diese werden in den folgenden Unterabschnitten genauer beleuchtet. Das Framework unterstützt dabei sowohl iOS-Simulatoren als auch physische iOS-Geräte.

2.1.1. Simulator Um die Analyse im iOS-Simulator zu automatisieren werden die Entwicklertools von Apple verwendet. Die benötigten Tools sind Teil der integrierten Entwicklungsumgebung XCode und können über das Kommandozeilenprogramm xcrun aufgerufen werden. Das Automatisierungsframework verwendet vor allem die Komponente simctl. Da die Kontrollapplikation in Python geschrieben ist, wird zum vereinfachten Aufruf der isim-Wrapper [3] verwendet.

2.1.2. iOS-Gerät Zur Steuerung von iOS-Geräten wird die Open Source-Bibliothek libimobiledevice [4] verwendet. Diese Bibliothek wurde ursprünglich ins Leben gerufen um iOS-Geräte auch unter dem von Apple nicht unterstütztem Betriebssystem Linux zu verwalten. Inzwischen beinhaltet die Bibliothek jedoch eine umfangreiche Toolsammlung, welche auch für die iOS-Entwicklung sowie die hier benötigte Automatisierung nützlich ist. Diese Sammlung ist in einzelne Komponenten aufgeteilt. Für das hier beschriebene Automatisierungsframework ist vor allem die Hauptkomponente libimobiledevice relevant. Diese übernimmt die Kommunikation mit dem Gerät und ermöglicht über den Befehl idevicesyslog das automatisierte Auslesen des Systemlogs. Weiters wird die Komponente ideviceinstaller verwendet. Diese ermöglicht es Applikationen auf dem iOS-Gerät zu installieren und zu verwalten.

2.2. Verwandte Arbeiten Auf dem Betriebssystem Android wurden automatische Analysetools erfolgreich eingesetzt, um Schwachstellen in der Programmierschnittstelle zu erkennen. So wurde ein solches Tool erstellt und eingesetzt, um Seitenkanäle in der Programmierschnittstelle zu detektieren [5]. Dabei wurde erkannt, dass vor allem Methoden, welche Zugang zu Statistiken wie Netzwerkaktivität oder Speicherplatzverbrauch ermöglichen, problematisch sind. Durch Analyse von über einen Zeitraum abgerufenen Informationen kann so die Aktivität der Nutzerin bzw. des Nutzers von eigentlich unberechtigten Applikationen nachverfolgt werden.

In [2] wurde ein automatisiertes Framework eingesetzt, um zeitbasierte Seitenkanäle in der Programmierschnittstelle von Android zu erkennen. Dazu wurden Methoden mit unterschiedlichen Parametern aufgerufen und die Ausführungszeit dieser Aufrufe verglichen. Unterschieden sich diese signifikant, so wurde ein potentieller Seitenkanal gefunden. In dieser Arbeit wurden mehrerer solcher Seitenkanäle gefunden. Diese erlaubten es beispielsweise zu erkennen, ob bestimmte Applikationen installiert, Konten eingerichtet oder gewisse Dateien am Smartphone vorhanden waren.

Zusätzlich zu Schwachstellen wurde ein solches Tool auch eingesetzt um die Fingerprintbarkeit von Android-Smartphones zu bewerten [1]. Dabei wurden die Methoden der API aufgerufen und die Rückgabewerte analysiert. Ebenso wurden Felder abgerufen und die Android-spezifischen Content Provider ausgelesen. Durch Vergleiche der erhaltenen Werte von verschiedenen Geräten konnte so festgestellt werden, welche Informationsquellen sich eignen, um ein Gerät zu identifizieren. Ebenso wurde im Zuge der Analyse herausgefunden, dass Herstelleranpassungen eindeutige Rückschlüsse auf Nutzerinnen bzw. Nutzer ermöglichen und so das Sicherheitsmodell von Android unterwandern.

3. Methodik 3.1. Aufbau Das Automatisierungsframework besteht aus zwei Komponenten. Zum einen gibt es die Kontrollapplikation und zum anderen die mobile Analyseapplikation. Die Kontrollapplikation läuft auf einem macOS-Gerät, während die mobile Analyseapplikation, wie der Name bereits vermuten lässt, auf dem iOS-Gerät (bzw. iOS-Simulator) läuft. Der generelle Aufbau sowie die Kommunikation zwischen den beiden Komponenten sind in Abbildung 1 ersichtlich. In den folgenden beiden Unterabschnitten wird auf die Aufgaben der beidem Komponenten im Detail eingegangen.

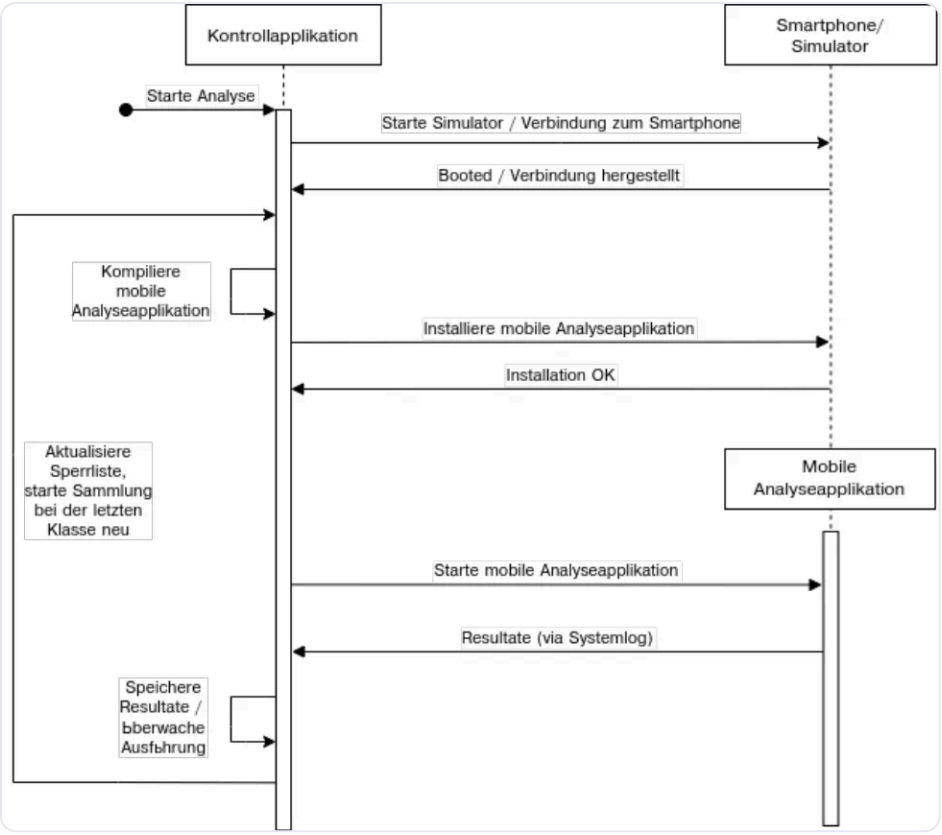


Abbildung 1: Übersicht über das Zusammenspiel der Kontrollapplikation und der mobilen Analyseapplikation.

3.1.1. Kontrollapplikation Die Kontrollapplikation übernimmt das Installieren und Starten der mobilen Analyseapplikation sowie das Sammeln der erhobenen Informationen. Sie wurde in Python geschrieben. Vor dem Start der Analyse kompiliert die Kontrollapplikation die letzte Version der Analyseapplikation. Danach wird sie auf dem Smartphone oder dem Simulator automatisch installiert. In weiterer Folge wird der Analysevorgang gestartet. Dabei werden von der Smartphoneapplikation die Resultate in das Systemlog des Geräts bzw. des Simulators geschrieben. Dieses Systemlog wird von der Kontrollapplikation in zweierlei

Hinsicht überwacht. Dazu werden von der Kontrollapplikation zwei Threads gestartet, welche das Systemlog mittels idevicesyslog bzw. xcrun simctl auslesen. Die Ausgabe des Systemlogs wird dabei Zeile für Zeile ausgewertet.

Einer der beiden gestarteten Überwachungsthreads protokolliert die erhaltenen Informationen. Diese haben ein vorab spezifiziertes Format. Dadurch können zum einen Systemausgaben, welche durch den Aufruf von Methoden generiert werden, ignoriert werden. Zum anderen können aber auch die Resultate in unterschiedliche Kategorien unterteilt werden. Die gesammelten Resultate werden vom Überwachungsthread in verschiedenen Dateien zur weiteren Untersuchung gespeichert. Der Überwachungsthread teilt hierbei die Resultate in Rückgabewerten von Methoden, Werte von Properties und gefangene Fehler auf. Weiters werden Methoden, die NULL zurückgeben bzw. Instanzvariablen die NULL sind in eine eigene Datei protokolliert. Dadurch enthalten die anderen Dateien nur Werte, die nicht NULL sind und so potentiell aussagekräftige Werte beinhalten.

Der zweite der gestarteten Überwachungsthreads überwacht die ordnungsgemäße Ausführung der mobilen Analyseapplikation. Dies wird benötigt, da es sich bei Objective-C um eine von C abgewandelte Sprache handelt. Dadurch kann es zu Speicherzugriffsfehlern oder ähnlichen Problemen kommen, welche die Ausführung der mobilen Analyseapplikation stoppen. Dies ist ein großer Unterschied zu Android, wo die meisten fehlerhaften Aufrufe durch die integrierte Fehlerbehandlung abgefangen und in weiterer Folge ignoriert werden können. Zwar gibt es auch bei Objective-C ein Error-Handling, diese Funktionalität wird jedoch in der Praxis in der Programmierschnittstelle kaum verwendet. Sollte die mobile Analyseapplikation also aufgrund eines Fehlers vom Betriebssystem beendet werden, so wird dies vom Überwachungsthread erkannt. Die Methode oder Klasse, welche die Probleme bereitete, wird daraufhin in die Sperrliste aufgenommen und die Applikation neu gestartet. Alle Methoden und Klassen der Sperrliste werden in weiterer Folge ignoriert. Auch bereits vollständig verarbeitete Klassen werden nicht erneut aufgerufen.

Weiters erkennt der Überwachungsthread, ob die mobile Analyseapplikation noch ordnungsgemäß Daten ausgibt. Sollte dies für einen spezifizierten Zeitraum (in der späteren Evaluierung wurde dieser Zeitraum auf eine Minute festgelegt) nicht der Fall sein, so wird davon ausgegangen, dass die Analyseapplikation hängt, weil sie beispielsweise eine Methode aufgerufen hat, welche auf etwas wartet (z.B. auf Benutzereingaben, Netzwerkpakete oder ähnliches). In diesem Fall wird die Applikation beendet und die Methode (oder Klasse) in die Sperrliste aufgenommen. Danach wird die Analyseapplikation neu gestartet und die blockierende Methode nicht erneut aufgerufen.

3.1.2. Mobile Analyseapplikation Die mobile Applikation sammelt die Daten, welche die API bereitstellt. Dazu werden Methoden aufgerufen und die Rückgabewerte gesammelt. Ebenso wird der Inhalt der Properties der Klasse abgerufen und gespeichert. Die technische Umsetzung erfolgt mit Hilfe von Reflection. Deshalb ist die mobile Analyseapplikation in Objective-C geschrieben, da nur Objective-C eine umfangreiche Reflection-API bietet, welche es erlaubt Methoden aufzurufen und Properties auszulesen. Dazu wird zuerst mittels der Methode objc_getClassList die Anzahl der zugreifbaren Klassen geholt. Auf Basis dieser Information kann ein Buffer erstellt werden, in den mit Hilfe derselben Methode alle Klassen als Class-Objekt gespeichert werden können. Auf Basis dieses Buffers können dann mit der alloc-Methode Instanzen aller dieser Klassen erstellt werden. Klassen in Objective-C werden in der Regel von NSObject abgeleitet. Dadurch können Properties mittels Reflection ausgelesen und Methoden aufgerufen werden. Nach der Instanziierung der Klasse wird das resultierende Objekt zuerst verwendet, um die Properties auszulesen. Danach werden die Methoden aufgerufen.

Um die Properties des Objekts abzurufen, wird zuerst eine Liste aller Properties mit der Methode class_copyPropertyList abgerufen. Diese Liste wird dann nach der Reihe abgearbeitet. Für jedes Property wird zuerst der Name des Properties mit property_getName abgerufen. Danach wird auf die Instanzvariable des Properties zugegriffen. Ist diese NULL, so ist das Property nicht gesetzt und es kann mit dem nächsten Property fortgefahren werden. Andernfalls wird versucht den Wert des Properties auszulesen. Dazu wird auf dem Klassenobjekt die Methode valueForKey aufgerufen. Der Rückgabewert dieser Methode ist der Wert des Properties. Dieser Wert wird daraufhin ausgegeben. Für eine korrekte Ausgabe ist der Typ des Properties erforderlich. Dieser wird über property_getAttributes eruiert und danach in die für die Ausgabe benötigte Form umgewandelt. Die Ausgabe des erhaltenen Wertes erfolgt über das Systemlog. Dieses wird wie im vorherigen Abschnitt beschrieben von der Kontrollapplikation überwacht.

Nach dem Abruf der Properties wird mit dem Aufruf der Methoden der Klasse fortgefahren. Dazu wird, analog zu den Properties, zuerst die Liste der Methoden der Klasse abgerufen. Dazu wird die Methode class_copyMethodList verwendet. Die abgerufene Liste enthält Objekte vom Typ Method. Diese ermöglichen es, in weitere Folge über die Methode method_getName einen Selektor der Methode abzurufen. Dieser Selektor erlaubt es dann, die Methode aufzurufen. Dazu wird auf dem Instanzobjekt der Klasse die Methode performSelector mit dem vorab abgerufenen Selektor ausgeführt. Um die Komplexität des Prototyps zu verringern, werden Methoden mit Parametern nicht unterstützt und deswegen vorerst ignoriert. Für Methoden ohne Parameter gibt die Funktion performSelector schließlich den Wert der eigentlich aufgerufenen Methode zurück. Schlussendlich erfolgt die Ausgabe des erhaltenen Rückgabewertes in das Systemlog. Dazu wird, wie bei den Properties, der Typ des Rückgabewertes benötigt. Dieser wird mit der Methode method_getReturnType abgerufen und dann in die benötigte Form umgewandelt. Sind alle Methoden behandelt, so wird mit der nächsten Klasse fortgefahren. Methoden, welche keinen Rückgabewert haben, werden ignoriert.

Wie bereits im vorherigen Unterabschnitt beschrieben wurde, kann es zu Problemen wie Speicherzugriffsfehlern kommen, welche zur Beendigung der mobilen Analyseapplikation führen. Ein solches Problem kann jedoch von der mobilen Analyseapplikation erkannt werden. Es ist zwar nicht möglich, die Applikation am letzten Punkt weiter auszuführen, es kann jedoch noch eine Nachricht an die Kontrollapplikation übertragen werden. Dazu wird in der mobilen Analyseapplikation ein SignalHandler installiert. Dieser wird vor dem Start der Analyse für alle möglichen Signale gesetzt. Kommt es dann zu einem Problem, so wird vor der Beendigung der Applikation dieser SignalHandler aufgerufen. Darin wird im Fall der mobilen Analyseapplikation der Crash inklusive der Signalnummer an die Kontrollapplikation gemeldet. Danach beendet sich die Applikation selbst und wird von der Kontrollapplikation mit einer aktualisierten Sperrliste neu gestartet.

```
#import <signal.h>
static void SignalHandler (int signo) {
NSLog(@"CRASH %d", signo);
exit(-1);
}
signal(signo, SignalHandler);
```

4. Evaluierung Um die grundlegende Funktionalität des Automatisierungsframeworks zu zeigen, wurde dieses getestet. In diesem Abschnitt werden die Ergebnisse dieser Analyse dargestellt. Für die Evaluierung des Analyseframeworks wurde ein iOS-Simulator verwendet. Dieser verwendete die zum Zeitpunkt der Analyse aktuelle Version 15.5 von iOS.

Die Programmierschnittstelle von iOS hat insgesamt 32.418 Klassen, welche über die Reflection-API erkannt und potentiell instanziiert werden können. Diese Klassen haben insgesamt 545.921 Methoden. Ebenso beinhalten sie 169.952 Properties. Bei der Evaluierung wurde versucht, Objekte aller dieser Klassen zu erstellen und daraufhin die Methoden aufzurufen und die Properties auszulesen.

Es wurden insgesamt 6512 Methoden und 254 Klassen zur Sperrliste hinzugefügt. Ein Großteil der Methoden wurde aufgenommen, da durch ihren Aufruf die mobile Analyseapplikation per Signal beendet wurde. Die meisterhaltenen Signale waren dabei Signal Nummer 11 (SIGSEGV, Speicherzugriffsfehler) und Signal Nummer 6 (SIGABRT, abort()). Weiters traten die Signale mit der Nummer 4 (SIGILL, illegal instruction), Nummer 5 (SIGTRAP, trace trap) sowie Nummer 10 (SIGBUS, bus error) auf. Zusätzlich konnten manche Methoden nicht aufgerufen werden, da der mobilen Analyseapplikation die dafür nötigen Rechte fehlten. Ein Beispiel hierfür ist die fehlende Berechtigung CloudKit zu verwenden:

```
** Terminating app due to uncaught exception
'NSInternalInconsistencyException', reason: 'The application is
missing required value "CloudKit" in entitlement
com.apple.developer.icloud-services. Has entitlements:
CKEntitlements<0x60000082af10>: entitlements = {
"application-identifier" = "AN332FG336.appid";
}, error = (null)'
** First throw call stack:
<…>
```

In diesem Fall wurde zwar kein Signal ausgelöst, die Kontrollapplikation konnte jedoch durch fehlende weitere Ausgabe erkennen, dass etwas schiefgegangen war. So wurde nach dem Timeout von einer Minute die Methode zur Sperrliste hinzugefügt und der Sammelvorgang neu gestartet.

Insgesamt konnten 212.826 Methoden erfolgreich aufgerufen werden. 116.596 Methoden gaben dabei einen Wert zurück, die restlichen 96.230 returnierten NULL. Bei den Properties konnten 161.399 erfolgreich durch die Applikation ausgelesen werden. Bei 3.458 konnte ein Wert eruiert werden, während bei den restlichen 157.941 Properties die Instanzvariable NULL war und somit keinen Wert gesetzt hatte. 1609 der Methodenaufrufe führten zu einem Fehler, der jedoch durch die interne Fehlerbehandlung gefangen werden konnte und so die weitere Ausführung der Applikation nicht beeinflusste.

5. Fazit In diesem Bericht wurde eine mögliche Methodik erläutert wie eine automatisierte Analyse der Programmierschnittstelle des Betriebssystems iOS technisch umgesetzt werden kann. Dazu wurde ein Prototyp entwickelt, welcher automatisiert Instanzen von Klassen erstellen kann um mit diesen die Properties der Klassen auszulesen sowie Methoden aufzurufen. Im Vergleich zu Android wurde die Umsetzung durch ein fehlendes (bzw. kaum verwendetes) einheitliches Fehlermanagement sowie mögliche Speicherzugriffsfehler erschwert. Um diese Probleme zu umgehen wurde eine

Kontrollapplikation eingeführt, welche die Ausführung der mobilen Analyseapplikation überwacht. Dadurch können Abstürze erkannt und problematische Methoden bzw. Klassen automatisch aus der Analyse entfernt werden.

Referenzen

[1] G. Palfinger und B. Prünster, „Systematic Analysis of the Fingerprintability of Android," 6 4 2020. [Online]. Available: https://technology.a-sit.at/en/publications/systematic-analysis-of-the-fingerprintability-of-android/.

[2] G. Palfinger, B. Prünster und D. J. Ziegler, „AndroTIME: Identifying Timing Side Channels in the Android API," 19th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom, pp. 1849-1856, 2020.

[3] D. Myers und weitere, „isim," 03 08 2022. [Online]. Available: https://github.com/dalemyers/xcrun.

[4] M. Szulecki, N. Bassen und weitere, „libimobiledevice," 03 08 2022. [Online]. Available: https://libimobiledevice.org/.

[5] R. Spreitzer, G. Palfinger und S. Mangard, „SCAnDroid: Automated Side-Channel Analysis of Android APIs," WiSec '18: Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, p. 224–235, 2018.