Architekturelle Sicherheitsanalyse für Android Apps

Bernhard J. Berger[1] · Karsten Sohr[1] · Udo H. Kalinna[2]

Technologie-Zentrum Informatik
und Informationstechnik (TZI)[1]
Universität Bremen
{berber | sohr}@tzi.de

ifAsec GmbH Institut für Applikations-Sicherheit[2]
udo.kalinna@ifasec.de

Zusammenfassung

Architekturelle Sicherheitsanalyse ist ein wichtiger Bestandteil eines sicherheitsbewussten Entwicklungsprozesses. Dieser Prozess erfordert jedoch fundiertes Wissen über Sicherheitskonzepte, deren korrekte Umsetzung und potenzielle Schwachstellen in verwendeten Software-Rahmenwerken. Im Rahmen unserer Forschungsarbeit haben wir diesen Prozess mit Hilfe von statischen Analysen und einer Wissensdatenbank, die architekturelle Sicherheitsregeln enthält, automatisiert. Dies ermöglicht es uns, automatisch eine Sicherheitsarchitektur zu extrahieren und in dieser nach Sicherheitsproblemen zu suchen, die bisher nicht durch andere statische Analysen detektiert werden konnten. Dies ermöglicht insbesondere kleinen und mittleren Unternehmen, die diese Expertise nicht besitzen, einen Einstieg in die architekturelle Sicherheitsanalyse. Eine Auswertung dieser automatisierten Technik für hybride Android-Apps zeigt, dass bei ungefähr zwei Drittel der Anwendungen potenzielle Sicherheitsschwächen enthalten sind, die es einem Angreifer sogar ermöglichen, eigenen Schadcode einzuschleusen und so im Rahmen der App-Berechtigungen beliebigen Programmcode auszuführen.

1 Motivation

Statische Analysewerkzeuge wie Coverity Prevent [Cov14] oder HP-Fortify [Soft13] werden vermehrt in der Industrie eingesetzt, um Sicherheitsprobleme auf der Implementierungsebene zu identifizieren. Mit Hilfe dieser Werkzeuge lassen sich erfolgreich Verwundbarkeiten aufdecken, die auf unvollständiger Eingabeüberprüfung beruhen, wie zum Beispiel SQL-Injection und Cross-Site-Scripting sowie Buffer Overflows. Der Einsatz dieser Tools ist relativ einfach und die Ergebnisse sind für einen Programmierer nachvollziehbar. Der Nachteil bei diesen Tools ist jedoch die Menge an gefundenen potenziellen Verwundbarkeiten, die schnell dazu führt, dass solche Tools nur noch sporadisch eingesetzt werden.

Wer seine Anwendung auf konzeptionelle Sicherheitsprobleme untersuchen möchte, kann architekturelle Risikoanalyse [McGr06] oder Microsoft Threat Modeling [SwSn04] verwenden. Hierbei wird ein Architekturmodell einer Software aufgestellt und dieses auf mögliche Sicherheitsrisiken untersucht. Die Ergebnisse dieser manuellen Analyse hängen sehr stark von dem Wissen des Durchführenden ab. Dies ist besonders für kleine und mittlere Unternehmen

P. Schartner · P. Lipp (Hrsg.) · D•A•CH Security 2014 · syssec (2014) 287-297.

(KMU) problematisch, da diese nicht über das notwendige Personal mit dem entsprechenden Wissen verfügen. Dennoch ist die architekturelle Risikoanalyse wichtig für die Sicherheit, da sie insbesondere konzeptionelle Sicherheitsprobleme aufdeckt, die durch existierende Werkzeuge nicht gefunden werden können. Die Ergebnismenge der architekturellen Risikoanalyse ist in der Regel nicht so umfangreich, wie die der oben genannten statischen Analysen. Dafür sind die gefundenen Probleme schwerwiegender und können leicht das gesamte Sicherheitskonzept einer Anwendung außer Kraft setzen. Bei der architekturellen Risikoanalyse versucht ein Sicherheitsexperte das Gesamtsystem aus der Sicht eines Angreifers zu betrachten und auf diese Weise potenzielle Schwachstellen zu identifizieren.

2 Ansatz

Im Rahmen verschiedener Projekte, unter anderem BMBF gefördert aber auch Industrieaufträge, haben wir einen Ansatz entwickelt, der einen Einsatz der architekturellen Risikoanalyse auch bei KMUs ermöglicht. Wir konnten ihn bereits erfolgreich für Java-Enterprise-Systeme einsetzen und erweitern ihn zur Zeit auf Android-Apps und hybride Android-Apps.

2.1 Anwendungskontext

Ein wesentlicher Nachteil von architekturellen Sicherheitsanalysen gegenüber dem Einsatz von statischen Analysetools ist die Notwendigkeit von teurem Sicherheitswissen. Aus diesem Grund haben wir die Verantwortlichkeiten in unserem Ansatz aufgeteilt und setzen Reverse-Engineering-Techniken, also die Extraktion von abstrakten Modellen aus dem Quelltext, ein. Hiermit erstellen wir automatisch eine Sicherheitsarchitektur der analysierten Anwendung und können anschließend eine automatische Erkennung von architekturellen Sicherheitsproblemen durchführen. Abbildung 1 zeigt die verschiedenen Beteiligten im Rahmen unserer automatisierten architekturellen Risikoanalyse.

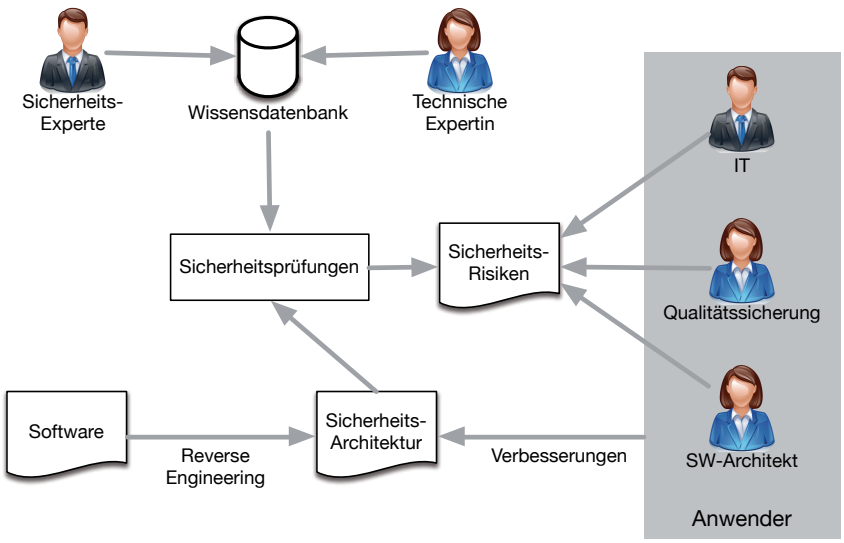

Abb. 1: Aufgabenverteilung bei der automatischen architekturellen Sicherheitsprüfung

Die zentrale Wissensdatenbank, die Informationen über Sicherheitsprobleme enthält, wird von Sicherheitsexperten und Technikexperten befüllt. Hierdurch werden diese zum Zeitpunkt der Analyse einer Software nicht benötigt. Bei der Analyse einer konkreten Software wird zunächst

mit Hilfe von Reverse-Engineering-Techniken eine Sicherheitsarchitektur extrahiert, die anschließend gegen die Regeln der Wissensdatenbank geprüft wird. Die Prüfung identifiziert Sicherheitsschwächen und mögliche existierende Gegenmaßnahmen. Diese werden zusammen mit einer ersten Einschätzung ihrer Relevanz in ein Risikomodell übertragen. Die Sicherheitsverantwortlichen, die den Ansatz anwenden, können die Ergebnisse begutachten. Auf Basis der Priorisierung und der detaillierten Beschreibung der Probleme können die nächsten Schritte zur Absicherung der Anwendung geplant werden. Wo es möglich ist, enthält die Wissensbasis auch Informationen über mögliche Gegenmaßnahmen.

Um mögliche Ungenauigkeiten der Reverse-Engineering-Techniken zu beheben, oder um die extrahierte Sicherheitsarchitektur um Systemeteile zu erweitern, für die wir keine Reverse-Engineering-Unterstützung bereitstellen, haben die Anwender die Möglichkeit, die extrahierte Sicherheitsarchitektur manuell zu verbessern. Es ist auch möglich, eine Sicherheitsarchitektur ganz von Hand zu erstellen und diese gegen die Wissensdatenbank prüfen zu lassen. Damit kann unser Ansatz auch im normalen Entwicklungsprozess angewendet werden, wenn noch keine Software existiert.

2.2 Sicherheitsarchitektur

Die Sicherheitsarchitektur wird in Form eines erweiterten Datenflussdiagramms erhoben. Ein Datenflussdiagramm besteht aus verschiedenen Prozessen, Datenspeichern, externen Akteuren, Datenflüssen und Vertrauensgrenzen. Sie werden zum Beispiel schon bei Microsofts Threat-Modeling verwendet, um eine Sicherheitsarchitektur darzustellen. Allerdings eignen sich diese Diagramme nicht sehr gut für eine automatische Analyse, weshalb wir sie im Rahmen unserer Arbeiten erweitert haben.

Erweiterte Datenflussdiagramme bestehen aus vier Modellierungselementen: Datenflüsse, Komponenten, Vertrauensbereiche und Daten. Jede dieser vier Modellelemente ist typisier- und attributierbar. Ein Typ beschreibt genauere Eigenschaften eines Elements, wie zum Beipiel Prozess, Android-App, Browser oder Software-Bibliothek bei den Komponenten. Attribute beschreiben sicherheitsrelevante Eigenschaften einer Komponente oder eines Datenflusses, wie zum Beispiel, ob eine Kommunikation verschlüsselt ist.

Komponenten stellen jegliche Elemente in unseren erweiterten Datenflussdiagrammen dar und können durch Datenflüsse miteinander verbunden werden. Vertrauensbereiche enthalten eine Menge an Komponenten und ersetzen Vertrauensgrenzen, da sie in unserem automatischen Prüfungsprozess besser verarbeitet werden können, ohne jedoch die Ausdruckskraft einzuschränken.

Die Daten, die ein System verarbeitet, werden in den erweiterten Datenflussdiagrammen explizit modelliert. Dies geschieht in Datenflussdiagrammen nicht. Hier werden sie nur durch Labels an den Datenflüssen dargestellt. Dies kann zu Mehrdeutigkeiten führen, wenn mehrere Datenflüsse in eine Komponente hinein fließen, aber auf unterschiedlichen Wegen wieder hinaus.

Durch die bereits erwähnte Typisierung ersetzen die Komponenten der erweiterten Datenflussdiagramme die Elemente Prozesse, Datenspeicher und externe Akteure der traditionellen Datenflussdiagramme. Außerdem kann ein Typ eine Menge an Annotationen implizieren. Hiermit können wir zum Beispiel automatisch einem Datenfluss, dem der Typ *HTTPS-Kommunikation* zugeordnet ist, die Annotation *Transport-Verschlüsselung* zuordnen. Dies hat bei einer manuellen Erstellung eines erweiterten Datenflussdiagramms den Vorteil, dass der

Ersteller nicht die Sicherheitseigenschaften von *HTTPS* kennen muss. Diese werden durch ein vordefiniertes Typ- und Annotationsschema vorgegeben.

2.3 Architekturrekonstruktion

Der Ablauf der Architekturrekonstruktion ist unabhängig von der analysierten Programmiersprache und des analysierten Software-Rahmenwerks. Abbildung 2 zeigt den allgemeinen Ablauf. Zunächst wird die Implementierung nach Softwaremustern durchsucht, die einer Komponente in der Software entspricht. Diese Muster sind natürlich abhängig von den Rahmenwerken, und es muss für jedes ein eigener Satz an Detektoren entwickelt werden. Diese detektierten Komponenten werden auf mögliche Ein- und Ausgänge untersucht, durch die Daten in die Komponente hinein oder hinaus fließen können. Im dritten Schritt wird der Intra-Komponenten-Datenfluss berechnet und auf dieser Basis die Ausgänge von Komponenten mit den Eingängen anderer Komponenten verbunden. Anschließend werden die Datenflüsse innerhalb von Komponenten verfolgt und von den Eingängen zu den Ausgängen propagiert. Somit ergibt sich nach dem vierten Schritt der Fluss der Daten durch das gesamte System. Im nächsten Schritt werden die Komponenten in Vertrauensbereiche eingeteilt. Abschließend annotiert die Analyse die Elemente der Architektur mit weiteren Sicherheitsinformationen.

Abb. 2: Ablauf der Architekturrekonstruktion

Obwohl der Analyseprozess übertragbar ist, müssen die einzelnen Analyseschritte für die verschiedenen Softwarerahmenwerke, wie zum Beispiel Android oder Java-Enterprise, separat umgesetzt werden. Dennoch können einige Analysen wiederverwendet werden. Außerdem ist zu erwähnen, dass die Extraktion der Vertrauensbereiche schwierig ist, da diese sehr unterschiedliche Formen annehmen können. Zur Zeit stützen wir uns auf die Umsetzung einer Zugriffskontrolle, wie zum Beispiel Rollen bei Java-Enterprise-Systemen oder Berechtigungsprüfungen bei Android-Apps.

2.4 Wissensdatenbank

Die Wissensdatenbank besteht aus einer Menge von Sicherheitsregeln und vordefinierten Regelsätzen. Die Regeln werden in einer domänenspezifischen Sprache beschrieben und enthalten Informationen über das geschätzte Schadensausmaß, die Eintrittswahrscheinlichkeit, verschieden detaillierte Beschreibungen des Problems und Referenzen auf weitere Ressourcen. Darüber hinaus enthält eine Regel noch eine Menge an Problemmustern, die Beschreiben, wie das Problem im erweiterten Datenflussdiagramm aussieht. Zu jedem Problemmuster kann es dann eine

beliebige Anzahl von Gegenmaßnahmen geben, die wieder durch ein eigenes Muster beschrieben sind.

McGraw definiert in [McGr06] drei verschiedene Problemgruppen, die architekturelle Risikoanalyse aufdecken können: anwendungsspezifische Sicherheitsrisiken, allgemeine Sicherheitsrisiken und Sicherheitsrisiken durch verwendete externe Software-Komponenten. Unsere Wissensdatenbank enthält Regeln aus den letzten beiden Bereichen. Darüber hinaus sind wir zudem in der Lage, weitere anwendungsspezifische Regeln aufzunehmen.

2.5 Automatisierte Sicherheitsprüfung

Die Problem- und Gegenmaßnahmenmuster in der Wissensdatenbank werden durch Object-Constraint-Language-Ausdrücke notiert. Die Object-Constraint-Language (kurz OCL) wurde von der Object-Management-Group spezifiziert, um Laufzeitbedingungen von Instanzen eins UML-Klassendiagramme anzugeben [Obj14b]. Wir nutzen einen Standard-OCL-Interpreter, um die Regeln gegen ein konkretes erweitertes Datenflussdiagramm zu prüfen.

Für jedes gefundene Problemmuster wird ein Eintrag in dem zu erstellenden Risikomodell erzeugt. Hierfür werden alle relevanten Informationen aus der Wissensdatenbank zu dem konkreten Problem kopiert. Wenn eine der angegebenen Gegenmaßnahmen ebenfalls gefunden wird, wird das Problem als behoben markiert.

3 Implementierung

In diesem Abschnitt beschreiben wir den aktuellen Stand der Umsetzung. Wir erläutern die verwendete Infrastruktur, die es uns erlaubt, die Analysen transparent für den Anwender auf leistungsfähige Server zu verschieben.

3.1 Implementierung der Architekturrekonstruktion

Die Analysen sind als OSGI-Services umgesetzt und erweiterbar, so dass Analysatoren für neue Rahmenwerke einfach hinzugefügt werden können. In einem ersten Schritt wird ein domänenspezifisches Anwendungsmodell gemäß Abbildung 2 extrahiert. Das auf diese Weise entstandene Anwendungsmodell ist detaillierter als das erweiterte Datenflussdiagramm, das in einem nächsten Schritt mittels Modell-zu-Modell-Transformation in ein erweitertes Datenflussdiagramm überführt wird. Die Überführung ist in der Transformationssprache QVT (siehe [Obj14a]) spezifiziert.

Zur Durchführung der Java-Analysen verwenden wir das statische Analyserahmenwerk Soot [RaCo99, LBLH11]. Dieses bietet verschiedene statische Basisanalysen, die zur Extraktion der notwendigen Fakten benötigt werden. Soot extrahiert aus kompiliertem Java-Bytecode alle notwendigen Typinformationen und für Methodenrümpfe eine einfach zu analysierende Zwischendarstellung. Darüber hinaus unterstützt es verschiedene Algorithmen mit unterschiedlichen Präzisionsstufen zur Aufrufgrapherzeugung.

Auf Soot basiert auch Heros, ein sehr präzises Datenflussanalyseframework. Dieses wird von FlowDroid verwendet, ein Werkzeug zur präzisen Berechnung von sensitiven Datenflüssen unter anderem in Android-Anwendungen [FARB+14].

Die Verwendung von Soot als Basis für die Analysen bietet sich an, da es zum einen bereits seit mehr als 15 Jahren entwickelt wird und daher einen großen Umfang an Standardanalysen

bietet. Zum Anderen enthält es auch bereits Unterstützung für den Android-spezifischen Dalvik-Bytecode. Dies ist besonders hilfreich, wenn nicht für alle Teile des analysierten Systems der Java Sourcecode zur Verfügung steht.

3.2 Infrastruktur

Die Analysen sind in eine Verteilungsinfrastruktur eingebunden, die es ermöglicht, Analyseprozesse nicht nur lokal auszuführen, sondern auch transparent für den Anwender auf leistungsfähige Server zu verschieben. Diese Infrastruktur wird auch dazu verwendet, Analyseprozesse aus unserem Android- oder Web-Frontend zu starten.

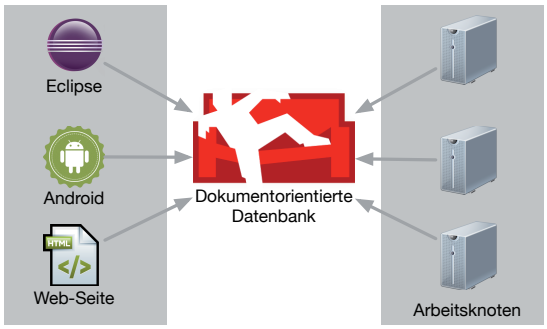

Abb. 3: Infrastruktur

In Abbildung 3 ist die Laufzeitumgebung unserer Analyseinfrastruktur abgebildet. Die verschiedenen Anwenderoberflächen erzeugen über die HTTP-Schnittstelle der zentralen dokumentorientierten Datenbank *CouchDB* die gewünschten Analyseaufträge. Im Hintergrund laufen Arbeitsknoten, die sich wartende Aufträge aus der Datenbank holen und diese entsprechend abarbeiten. Die Ergebnisse werden wieder zurück in die Datenbank geschrieben und stehen im Folgenden für die Klienten bereit.

In der Datenbank werden neben dem Analyseauftrag auch Metadaten, wie zum Beispiel Log-Ausgaben und Zwischenergebnisse, gesammelt, um eine spätere Fehleranalyse zu ermöglichen. Das Datebankdesign wurde so ausgelegt, dass die Datensicherheit der Nutzer gewährleistet ist. Jeder Nutzer besitzt seinen eigenen Datenbereich, und die Datenbankschicht stellt bereits sicher, dass kein anderer Nutzer auf diese Daten zugreifen kann. Zudem sichern wir das System selbst durch eine architekturelle Sicherheitssanalyse ab.

3.3 Anwenderoberfläche

Zum aktuellen Zeitpunkt existieren drei verschiedene grafische Oberflächen für verschiedene Nutzergruppen. Zu Beginn entstanden eine Reihe von Eclipse-Erweiterungen, um den in Abbildung 1 dargestellten Prozess zu unterstützen. Im weiteren Verlauf der Projektarbeiten ergaben sich aber auch interessante weitere Anwendungsszenarien, die eigene Darstellungsmöglichkeiten der Ergebnisse erforderten.

3.3.1 Android-App

Wir haben eine einfache Android-App entwickelt, mit der sich ein Endanwender die Apps, die auf seinem Gerät installiert sind, analysieren lassen kann. Die Ergebnisse werden hierbei nicht

detailliert angezeigt, sondern in Kategorien zusammengefasst und die Auswirkung der Probleme beschrieben. Diese Darstellung richtet sich an unerfahrene Endanwender, die hauptsächlich am Schutz ihrer Daten interessiert sind und einen schnellen Überblick erhalten wollen.

Die App bietet eine Übersicht über die angeforderten Berechtigungen, enthaltene Implementierungsfehler und die sensitiven Informationen, die die App versendet. Am wichtigsten bei dieser Darstellung ist eine Reduktion der Information und eine verständliche Kommunikation, was bei Sicherheitsproblemen üblicherweise eine Herausforderung darstellt.

3.3.2 Web-Frontend

Wir haben ein Web-Frontend entwickelt, mit dem Entwickler selbst erstellte Apps untersuchen können. Die Ergebnisdarstellung ist detaillierter als die der Android-App und versucht, wenn möglich, dem Entwickler auch Wege aufzuzeigen, wie die Probleme gelöst werden können. Die Beschreibungen der Probleme sind technischer und enthalten auch Informationen über mögliche Exploits.

3.3.3 Eclipse-Erweiterung

Die Eclipse-Erweiterung richtet sich an Sicherheitsexperten und Sicherheitsevaluatoren. Diese können die entstandenen erweiterten Datenflussdiagramme (siehe Abbildung 4) und das berechnete Gefahrenmodell von Apps untersuchen. Gerade für Evaluatoren, die eine fremde App verstehen wollen, ist die grafische Darstellung interessant, weil sie einen schnellen Überblick geben kann.

Die grafischen Eclipse-Erweiterungen integrieren sich in das gewohnte Bedienkonzept der Eclipse-Plattform und verwenden zum Beispiel Graphiti [The14b] zur Darstellung der erweiterten Datenflussdiagramme. Den Elementen der Sicherheitsarchitektur wird ein kleines Icon hinzugefügt, wenn eine potenzielle Schwachstelle identifiziert wurde. Das resultierende Gefahrenmodell wird als einfache nach Priorität sortierte Liste dargestellt und auf Wunsch kann man sich die Details zu einem Problem anschauen.

Die Nutzerschnittstelle enthält auch eine Integration der serverseitigen Analyseinfrastruktur, so dass die Analysen einfach gestartet werden können und langlebige Analyseläufe, die in der Regel hohe Speicheranforderungen haben, transparent auf leistungsfähige Server ausgelagert werden.

Das System bietet zudem die Möglichkeit, eine Sicherheitsarchitektur für eine geplante Software manuell zu erstellen. Die Wissensdatenbank kann auch auf eine manuell erstellte Sicherheitsarchitektur angewendet werden. Somit kann auch eine neu erstellte Software von unserem Ansatz profitieren. Dennoch ist der Reverse-Engineering-Ansatz sinnvoll, da es in der Regel eine Abweichung zwischen der geplanten und der tatsächlich implementierten Software gibt.

3.4 Wissensdatenbank

Die Wissensdatenbank ist zum einen mit Regeln gefüllt, die nach Problemen suchen, die aus der Common Weakness Enumeration (CWE) [MIT12] bekannt sind. Ein Beispiel ist der unverschlüsselte Transport von sensitiven Daten, wie zum Beispiel Passwörtern.

Darüber hinaus befinden sich auch Android-bezogene Regeln in der Wissensdatenbank. Die Android-Middleware ist zwar prinzipiell auf Sicherheit ausgelegt, es gibt jedoch einige Standardeinstellungen, die zu Sicherheitsproblemen führen können. Hierzu zählen zum Beispiel

appspezifische Datenbanken, die bis vor wenigen Android-Versionen ohne zusätzliche Einstellung für alle Apps zugänglich sind.

Darüber hinaus haben wir auch Detektoren implementiert, die bei ausgehenden Verbindungen prüfen, ob diese eine Transportverschlüsselung verwenden und ob sie, wie von Georgiev et al. [GIJA+12] und Fahl et al. [FHMB+12] berichtet, unter Umständen unsicher sind.

Des Weiteren sind Regeln für hybride Android-Apps enthalten, die mit Apache Cordova (siehe [The14a]) umgesetzt sind. Dieses Rahmenwerk bietet verschiedene Möglichkeiten, die Sicherheit der App und der Nutzerdaten zu gefährden. Mehr Details hierzu geben wir in Abschnitt 4.

3.5 Unterstützte Rahmenwerke

Auf Grund der durchgeführten Projekte existieren zur Zeit Analyse-Werkzeuge für JavaEE-basierte Anwendungen, Android Apps und hybride Anwendungen auf Basis des HTML-Frameworks Apache Cordova. Hierfür analysieren wir die Deployment-Informationen und den Bytecode von JavaEE-Anwendungen und Android Apps mit Hilfe von Soot. Im Fall von hybriden Anwendungen berücksichtigen wir auch die verwendeten HTML- und JavaScript-Dateien, indem wir sie mit jsoup[Jon14] und Mozilla Rhino[Moz14] analysieren. Die Ergebnisse der einzelnen Analysen werden zu einer einheitlichen und plattformunabhängigen Sicherheitsarchitektur zusammengefasst.

Abbildung 4 zeigt einen grafischen Überblick über die automatisch extrahierte Sicherheitsarchitektur der Wikipedia Mobile App für Android. Auf der linken Seite sieht man einen extrahierten Knoten, der die gesamte Anwendung darstellt. Er enthält einen Knoten für eine Visualisierungkomponente (Activity) mit dem Namen Wikipedia und eine unbenannte lokale Datenbank (Content-Provider). Im unteren Bereich der Anwendung sieht man die Dateien, die in der Anwendung enthalten sind. Es handelt sich um verschiedene HTML, JavaScript und CSS-Dateien. Auf der rechten Seite der Abbildung sind zwei Domains zu sehen, denen die Implementierung vertraut und von denen Daten nachgeladen werden dürfen. Die Pfeile stehen für Datenflüsse. Es ist zu sehen, dass von der externen Domain *http://*.en.m.wikipedia.org* Daten geladen werden. Die geladenen HTML-Seiten sind nicht verschlüsselt, was einem Angreifer ermöglicht, sie auf dem Transport zu manipulieren.

4 Ergebnisse

In einer bereits veröffentlichten Arbeit haben wir Java-Enterprise-Systeme mit Hilfe unseres Ansatzes analysiert [BeSK13]. Hierbei konnten wir in einem Teil der untersuchten Anwendungen architekturelle Sicherheitsprobleme identifizieren. Im Fall einer Anwendung aus dem E-Government-Bereich war uns dies nicht möglich, was nicht verwunderlich war, da die Anwendung gerade nach dem Common Criteria Standard evaluiert wurde und dementsprechend schon auf Sicherheitsprobleme untersucht worden war.

In Rahmen einer größeren Untersuchungsreihe haben wir im Februar 2014 mehr als 850 akutelle Android Apps aus dem Google Play Store untersucht. Der Corpus bestand aus Apps, die auf dem hybriden Framework Apache Cordova basieren. Apache Cordova erlaubt dem Entwickler HTML5- und JavaScript-basierte Anwendungen zu entwickeln, die auf die Funktionen von Smartphones, wie zum Beispiel Positionsdaten, zugreifen können. Dies macht diese Apps zu einem interessanten Angriffsziel, da hier mit bekannten Angriffen aus der Internet-Welt, wie

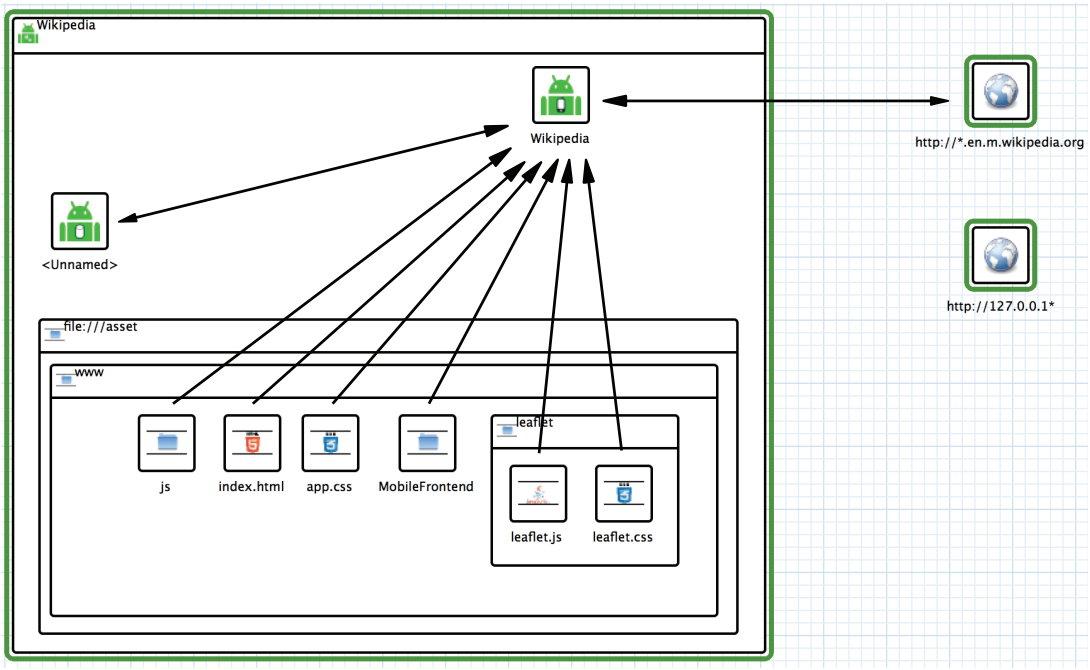

Abb. 4: Grafische Darstellung der Zwischendarstellung

zum Beispiel Cross-Site-Scripting, mehr erreicht werden kann, als bei normalen Browsern. Die größte Gefahr ist wohl, dass ein Angreifer es schafft, beliebigen JavaScript-Code einzuschleusen, der dann ausgeführt wird und mit dem man unter Umständen neue Prozesse starten kann, die erst beim Systemneustart beendet werden.

Die Anwendungen waren teilweise Open-Source-Anwendungen, aber auch sehr viele von großen kommerziellen Anbietern wie Intel, Siemens, SAP, Telekom und HP. Es stellte sich heraus, dass 19% der analysierten Anwendungen Daten gänzlich unverschlüsselt von Servern aus dem Internet nachladen und damit dem ausgespäht werden oder der Manipulation preisgeben. Darüber hinaus verwendeten zwar 9% der Anwendungen ein verschlüsseltes Protokoll, machten allerdings Fehler bei der Verwendung des Frameworks, was dazu führt, dass die Daten trotz der Verschlüsselung gelesen und manipuliert werden können. 53% der Anwendungen aktivierten ein Feature des Android-Frameworks, nämlich den Zugriff auf Java-Objekte aus dem JavaScript-Code, auf eine Weise, die es erlaubt, beliebigen Code auf dem Android-Gerät auszuführen. Alles in allem wurden lediglich 32% der Apps in diesem ersten Analyselauf als nicht offensichtlich angreifbar eingestuft.

In der Android-Fallstudie konnten die Ergebnisse ohne weitere Hilfe durch einen Sicherheits- oder Anwendungsexperten erhoben werden. Eine manuelle Validierung der Ergebnisse zeigt eine sehr hohe Rate an richtigen Treffern. Trotzdem können Ergebnisse übersehen worden sein, da applikationsspezifische Besonderheiten, wie sensitive Daten nicht berücksichtigt wurden.

5 Zusammenfassung und Ausblick

Durch die allgegenwärtige Vernetzung und Integration von Softwaresystemen ist die Sicherheit von Software eine wichtige Produkteigenschaft. Diese Eigenschaft wird durch mangelndes Sicherheitswissen und immer kürzere Veröffentlichungszyklen gefährdet. Architekturelle Si-

cherheitsanalyse ist ein hilfreicher Bestandteil bei der Entwicklung und Absicherung von Softwaresystemen, um die Sicherheit von Anwendungen zu gewährleisten. Sie ist eine sinnvolle Ergänzung zu bestehenden Werkzeugen zur Identifikation von Sicherheitsproblemen, die auf Implementierungsfehlern basieren.

Mit Hilfe unserer Wissensdatendank von architekturellen Sicherheitsproblemen können wir Softwareherstellern helfen, die nicht die notwendige Sicherheitsexpertise besitzen. Durch die Verwendung von Reverse-Engineering-Techniken zur Extraktion der Sicherheitsarchitektur aus einer bestehenden Implementierung ist der Aufwand für den Einsatz unseres Ansatzes auf der Seite des Anwenders gering. Dies macht auch die Durchführung in kleinen und mittleren Unternehmen möglich und kann dabei helfen, schwerwiegende Sicherheitsprobleme zu vermeiden.

Die Extraktion einer Sicherheitsarchitektur aus einer bestehenden Implementierung vermeidet falsche Ergebnisse, die auf einer Abweichung der geplanten und der umgesetzten Architektur beruhen. Allerdings kann mit den entwickelten Werkzeugen auch manuell eine Sicherheitsarchitektur zur Prüfung erstellt werden.

Die architekturelle Sicherheitsanalyse findet weniger Fehler als klassische Analyseverfahren zur Erkennung von Implementierungsfehlern. Die gefundenen Fehler sind jedoch fundamentaler und können das Design einer Anwendung beeinflussen. Die ausführlichen Erklärungen, die in der Wissensdatenbank enthalten sind, helfen Sicherheitslaien, geeignete Gegenmaßnahmen zu identifizieren.

Mit der prototypischen Android-App geben wir Endanwendern die Möglichkeit Einblick in die Sicherheit der von ihnen verwendeten Apps zu erhalten. Hierdurch ergeben sich interessante Einblicke für Endanwender, aber auch für die Forschung, da wir ihr Verhalten studieren können. Im Rahmen einer ersten Studie wollen wir die App den Studenten der Universität Bremen zur Verfügung stellen, um in einer empirischen Studie herauszufinden, welche Informationen für sie relevant sind und wie sie mit diesen umgehen.

Für die Analyse der anbieterseitigen Komponenten planen wir zudem eine Integration des IO-Frameworks der Universität Bremen. Dieses kann mit Hilfe von dynamischen Analysen den Aufbau von Netzen und die verwendeten Hardware-Komponenten und Softwarestände identifizieren [BSSB12]. Auf dieser Basis werden die erweiterten Datenflussdiagramme mit zusätzlichen Informationen angereichert. Die Common Vulnerabilities and Exposures Liste, eine Liste, die Verwundbarkeiten von Standardkomponenten aufzeigt, gibt uns die Möglichkeit, automatisch nach Verwundbarkeiten zu suchen, die sich durch externe Bibliotheken, Programme oder Hardware-Komponenten ergeben.

Danksagung

Die Arbeit an diesem Ansatz wird vom Bundesministerium für Bildung und Forschung gefördert (ZertApps-Projekt – Förderkennzeichen 16KIS0074). Darüber hinaus möchten wir uns bei den anonymen Gutachtern bedanken, die wesentlich dazu beigetragen haben, diesen Beitrag zu verbessern.

Literatur

[BeSK13] B. Berger, K. Sohr, R. Koschke: Extracting and Analyzing the Implemented Security Architecture of Business Applications. *In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on* (2013), 285–294.

[BSSB12] H. Birkholz, I. Sieverdingbeck, K. Sohr, C. Bormann: IO: An Interconnected Asset Ontology in Support of Risk Management Processes. *In: ARES*, IEEE Computer Society (2012), 534–541.

[Cov14] Coverity, Inc: Coverity Prevent (2014), http://www.coverity.com

[FARB+14] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, P. McDaniel: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *In: Proceedings of the 35th ACM SIGPLAN conference on Programming language design and implementation (PLDI)* (2014), to appear.

[FHMB+12] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, M. Smith: Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. *In: Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM, New York, NY, USA (2012), 50–61.

[GIJA+12] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, V. Shmatikov: The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. *In: Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM, New York, NY, USA (2012), 38–49.

[Jon14] Jonathan Hedley: jsoup HTML parser (2014), http://jsoup.org

[LBLH11] P. Lam, E. Bodden, O. Lhoták, L. Hendren: The Soot framework for Java program analysis: a retrospective. *In: Cetus Users and Compiler Infrastructure Workshop*, Galveston Island, TX (2011).

[McGr06] G. McGraw: Software Security: Building Security In. Addison-Wesley (2006).

[MIT12] MITRE Corporation: The Common Weakness Enumeration (CWE) Initiative (2012). http://cwe.mitre.org

[Moz14] Mozilla Developer Network: Rhino (2014). https://developer.mozilla.org/de/docs/Rhino

[Obj14a] Object Management Group: Meta Object Facility (MOF) 2.0 Query / View / Transformation (QVT) (2014). http://www.omg.org/spec/QVT.

[Obj14b] Object Management Group: OCL (2014). http://www.omg.org/spec/OCL.

[RaCo99] V. S. P. L. E. G. Raja Vallée-Rai, Laurie Hendren, P. Co: Soot – a Java Optimization Framework. *In: Proceedings of CASCON 1999* (1999), 125–135. www.sable.mcgill.ca/publications.

[Soft13] F. Software: Fortify Source Code Analyser (2013). http://www.fortify.com/products

[SwSn04] F. Swiderski, W. Snyder: Threat Modeling. Microsoft Press, Redmond, WA, USA (2004).

[The14a] The Apache Software Foundation: Apache Cordova (2014). http://cordova.apache.org

[The14b] The Eclipse Foundation: Graphiti (2014). http://projects.eclipse.org/projects/modeling.gmp.graphiti