Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 2425

Verschlüsselte E-Mails auf mehreren Endgeräten

Jakob Bode[1], Matthias Sekul[1] und Tobias Schülke[1]

Abstract: Trotz der bereits langen Existenz von E-Mail-Verschlüsselungsverfahren wie z.B. PGP verwenden wenige E-Mail-Nutzer diese Lösungen. Vielfach wird damit argumentiert, dass die Anwendung dieser Verfahren zu kompliziert sei. Dies bezieht sich häufig auf die Benutzerschnittstelle. Wir sehen ebenfalls ein Problem in der komplizierten Verwaltung der Schlüssel. Um die Verwaltung der Schlüssel zwischen mehreren Geräten zu ermöglichen, haben wir eine Bibliothek geschrieben, die einen verschlüsselten und authentifizierten Kanal zwischen zwei Geräten aufbaut. Darüber kann auf sehr einfache Weise ein privater Schlüssel übertragen werden. Diese Lösung haben wir in die OpenKeychain App[2], die zusammen mit K-9-Mail[3] zum weit verbreiteten Kombination für PGP unter Android gehört, eingebaut. Die von uns implementierte Möglichkeit der Übertragung haben wir mit bereits existierenden Möglichkeiten mittels Cognitive Walkthrough [Po92] verglichen und festgestellt, dass sie die Übertragung deutlich vereinfacht.

Keywords: Privater Schlüssel; Kryptographie; Device Pairing; E-Mail; Smartphone

1 Einleitung

Seit einigen Jahren erfreuen sich Cryptomessenger immer größerer Beliebtheit. Dies liegt auch an der weiten Verbreitung von Smartphones und Tablets, auf denen diese vorrangig genutzt werden. Dagegen besteht nur eine geringe Nutzung von verschlüsselter E-Mail auf diesen Geräten. Vergleicht man die Anzahl der Downloads von Whatsapp (1 bis 5 Milliarden)[4] mit denen von OpenKeychain (100 bis 500 Tausend)[5], einem Schlüsselmanager, der zusammen mit K-9-Mail zu einem der am häufigsten verwendeten Möglichkeiten zur E-Mailverschlüsselung auf Android zählt, wird das Missverhältnis zwischen Cryptomessengern und verschlüsselter E-Mail sehr deutlich. Unserer Hypothese nach liegt das auch in der Schwierigkeit begründet, verschlüsselte E-Mails auf mobilen Endgeräten zu verwenden. Zwar wurden viele Nutzungsaspekte verschlüsselter E-Mail in den letzten Jahren deutlich verbessert, jedoch bezog sich dies oft auf Desktop-Anwendungen. Wenig Beachtung fand die Problematik, dass E-Mails häufig nicht nur auf einem, sondern auf mehreren Geräten gelesen und geschrieben werden. Dadurch ist ein Schlüsselmanagement von privaten Schlüsseln

[1] Freie Universität Berlin, Fachbereich Informatik, Takustraße 9, 14195 Berlin, Deutschland vorname.nachname@fu-berlin.de

[2] https://www.openkeychain.org/

[3] https://k9mail.github.io/ E-Mail App mit OpenKeychain Unterstützung

[4] https://play.google.com/store/apps/details?id=com.whatsapp&hl=de

[5] https://play.google.com/store/apps/details?id=org.sufficientlysecure.keychain&hl=de

doi:10.18420/in2017_247

zwischen mehreren Geräten nur wenig umgesetzt worden.
Um diesen Zustand zu verbessern, haben wir im Rahmen eines Softwareprojektes an der Freien Universität Berlin OpenKeychain um eine für Nutzer einfach verwendbare Möglichkeit erweitert, private Schlüssel zwischen verschiedenen Geräten auszutauschen.
Im Folgenden werden wir uns auf den Transfer von privaten Schlüsseln vom Desktop zum Smartphone fokussieren. Dabei werden wir erst einen Einblick in vorhandene Literatur geben und im Anschluss kurz beschreiben, gegen welche Angreifer wir uns schützen möchten, um danach die bisher in OpenKeychain vorhandenen Verfahren zu analysieren. Dies tun wir in Form eines Cognitive Walkthroughs. Danach stellen wir unsere Lösung vor und vergleichen sie mit den existierenden Verfahren. Abschließend folgt unser Fazit.

2 Verwandte Arbeiten

Seit vielen Jahren wird nach einer Möglichkeit gesucht, E-Mail-Verschlüsselung für eine breite Masse nutzbar zu machen. Besonder Fokus wurde dabei auf die Benutzerschnittstelle in der Hinsicht gelegt, dem Nutzer ein besseres Verständnis für sein Handeln zu ermöglichen. Hierbei wurde u.a. darauf geachtet, dass dem Nutzer bewusst ist, wann er verschlüsselt, wann nicht, ob eine erhaltene E-Mail von einem gültigen, vertrauten Schlüssel signiert wurde und welche Gefahren von einem sich ändernden Schlüssel ausgehen können. Nachdem diese Aspekte anfänglich in Implementierungen kaum umgesetzt wurden [WT99], wurden im Laufe der Zeit bessere Darstellungen gefunden, dem Nutzer einen Überblick zu gewähren [GM05, Ru16]. Es ist immer noch Entwicklungsbedarf vorhanden [Ru15].
Ein anderer Zweig der Forschung beschäftigt sich intensiver mit dem Austausch von öffentlichen Schlüsseln und der Verifizierung derer [Fa13]. Hier wird die Nutzung vereinfacht, indem andere Partner, mit denen verschlüsselt kommuniziert werden soll, leicht zu finden bzw. hinzuzufügen sind.
Wir betrachten die Thematik aus einer anderen Perspektive und fokussieren uns auf die Schlüsselsynchronisierung zwischen den eigenen Geräten. Denn heute in Zeiten von kurzlebigen Smartphones und mehreren anderen Geräten, die von einem Nutzer verwaltet werden, ist dies ein Hemmnis für die weite Verbreitung von verschlüsselter Kommunikation. Schließlich müssen Nutzer bei jedem Gerät umständlich ihre Schlüssel aufspielen. Eine Lösungsmöglichkeit ist die Verwendung eines zentralen Servers, auf dem sich die Schlüssel befinden. Dieser Ansatz ist auch bei der E-Mail[6] vorhanden. Das Problem hierbei ist, dass einem weiteren Server vertraut wird und dieser auch erreichbar sein muss. Ebenfalls ist eine zusätzliche Anwendung notwendig.
Unsere Lösung basiert auf Device Pairing [NR10], dabei wird über einen für Angreifer nicht veränderbaren, aber u.U. abhörbaren Kanal mit geringer Bandbreite der jeweilige Kommunikationspartner beim Verbindungsaufbau authentifiziert. Hier gibt es verschiedene Möglichkeiten, wie ein solcher Kanal aussehen kann [KFR09]. Wir haben uns auf einen visuellen Kanal in Form eines QR-Codes, der von einem Teilnehmer angezeigt und vom anderen gescannt werden muss, entschieden.

[6] https://keybase.io/

3 Angreifermodell

Wir gehen davon aus, dass beide Geräte nicht kompromittiert sind. Weiterhin kann ein Angreifer die Verbindung zwischen den Geräten bzw. Anwendungen nicht belauschen oder mögliche Nachrichten verändern. Eine Ausnahme bildet der visuelle Kanal zwischen den Geräten, der für den Angreifer jedoch ebenfalls weder lesbar noch beschreibbar ist.

4 Bereits existierende Verfahren

Wir haben die Erweiterung in OpenKeychain mit dem Ziel implementiert, dass ein privater Schlüssel sehr einfach vom Desktop-Rechner auf ein Smartphone mit OpenKeychain importiert werden kann. Deshalb fokussieren wir uns in unserer Auswertung auf diese Übertragung. Das Verfahren lässt sich ebenfalls dazu nutzen, zwischen zwei Smartphones Schlüssel auszutauschen oder einen Schlüssel vom Smartphone auf einen Desktop-Rechner zu spielen.
In den offiziellen OpenKeychain FAQ[7] werden mehrere Möglichkeiten genannt, einen bereits existierenden privaten Schlüssel auf das Smartphone in die OpenKeychain zu transferieren. Dabei wird besonders darauf hingewiesen, dass ein Mitschneiden des Schlüssels durch Dritte unterbunden werden muss. Hierfür wird insbesondere auf die in GnuPG[8] (kurz: gpg) eingebaute symmetrische Verschlüsselung hingewiesen. Im Wesentlichen werden hier drei Transfermöglichkeiten genannt: Übertragung per SD-Karte, Übertragung per E-Mail und Übertragung über Filehosting-Dienste (im Folgenden „Cloud“).
Ergänzen ließe sich in dieser Liste noch der Transport per USB-Kabel, mit dem das Smartphone direkt am Rechner angeschlossen wird. Diese Möglichkeit ähnelt stark der SD-Karten-Variante, weshalb wir sie gemeinsam unter dem Punkt SD-Karte/USB-Kabel behandeln.
Jedes der besprochenen Verfahren folgt einem groben Schema. Zuerst wird der private Schlüssel auf dem Desktop-Rechner exportiert und verschlüsselt. Im Anschluss wird dieser auf das Smartphone übertragen und dort in OpenKeychain importiert. In jedem Fall gehen wir davon aus, dass der Nutzer bereits einen existierenden Schlüssel auf seinem Desktop-Rechner besitzt, den er aus gpg exportieren möchte.

4.1 SD-Karte/USB-Kabel

Der Import privater Schlüssel mittels SD-Karte verlangt das Vorhandensein von entsprechender Hardware (Karte und Slots bzw. USB-Kabel).
Der Nutzer muss auf dem Desktop-Rechner seinen Schlüssel in eine Datei exportieren. Das

[7] https://www.openkeychain.org/faq/#what-is-the-best-way-to-transfer-my-own-key-to-openkeychain
[8] https://gnupg.org/index.html

lässt sich über die in den FAQ beschriebenen gpg Befehle erledigen. Für viele wird das zu kompliziert sein.
Als Alternative gibt es Programme mit graphischer Schnittstelle wie das Enigmail-Plugin[9] für Thunderbird, die einen Export der Schlüssel ermöglichen. Bei dem Export wird im besten Fall der Schlüssel des Nutzers mit einem sicheren Passwort verschlüsselt und in einer Datei abgelegt. Das ist allerdings nicht immer der Fall. Kann der Nutzer selbst ein Passwort festlegen, wird er vermutlich eines wählen, das für einen Angreifer leicht zu erraten oder zu errechnen ist [Ur15]. Im schlimmsten Fall wird der private Schlüssel unverschlüsselt abgelegt.
Die Schlüsseldatei muss im Anschluss auf die SD-Karte übertragen und, nachdem die Karte in das Smartphone gesteckt wurde, in OpenKeychain importiert werden. Dabei gibt es zwei Vorgehensweisen: Das Importieren der Datei direkt aus OpenKeychain heraus und das Importieren über den Dateimanager des Smartphones. Wir betrachten zuerst den letzteren Fall. Über den Dateimanager wird von dem Nutzer die Datei ausgewählt, woraufhin ein Dialog (siehe Abbildung 1) erscheint, der ihn vor die Wahl stellt, entweder die Schlüsseldatei zu entschlüsseln (oben rechts) oder sie zu importieren (unten links). Wurde die Datei verschlüsselt, muss „entschlüsseln“ ausgewählt werden. Wird stattdessen der Import des Schlüssels gewählt, schlägt der Vorgang fehl. Das ist aus Nutzersicht schlecht. Der Import des Schlüssels ist das eigentliche Ziel, weshalb ein Auswählen dieses Buttons logisch erscheint.

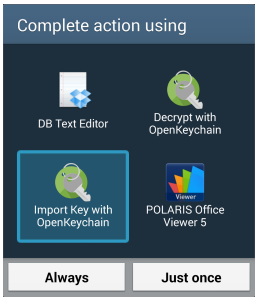

Abb. 1: Auswahldialog nach Selektieren der Datei

Nachdem der richtige Button gewählt wurde, muss bei vorheriger Verschlüsselung der Datei das entsprechende Passwort eingegeben werden. Im Anschluss bestätigt der Nutzer noch, dass er den Schlüssel importieren möchte.
Der Import aus OpenKeychain heraus umgeht dabei den Dialog aus Abbildung 1. Nach der Auswahl der Datei muss der Nutzer das Passwort eingeben und den Import des Schlüssels bestätigen.
Hervorzuheben ist bei diesem Verfahren, dass die Schlüsseldatei sich für einige Zeit im allgemeinen Speicher des Smartphones befindet, auf den alle Apps Zugriff haben. Um zu verhindern, dass unintendierte Apps Zugriff auf den Schlüssel erhalten, ist eine sichere

[9] https://www.enigmail.net/index.php/en/

Verschlüsselung vonnöten. Diese ist, wie oben gezeigt, durch kritische Nutzerfehler nicht immer vorhanden. Ebenso sind viele Interaktionsschritte notwendig.

4.2 E-Mail

Möchte der Nutzer seinen Schlüssel per E-Mail übertragen, muss er diesen, wie im vorherigen Abschnitt beschrieben, exportieren und verschlüsseln. Anschließend schreibt er eine E-Mail, die auf dem Smartphone empfangen wird. Die Schlüsseldatei fügt er als Anhang hinzu. Nach der Öffnung der E-Mail auf dem Smartphone wählt er den Anhang aus. Es erscheint der gleiche Dialog, wie in Abbildung 1 dargestellt. Das heißt, dass der Nutzer hier ebenfalls vor dasselbe Problem gestellt wird und womöglich die falsche Aktion wählt. Die nachfolgenden Schritte sind die gleichen wie bei der SD-Karte.
Hat der Nutzer ein schlechtes oder gar kein Passwort zum Verschlüsseln der Schlüsseldatei gewählt, gibt er seinen Schlüssel beim Versenden per E-Mail vielen potentiellen Angreifern preis. Seine E-Mail befindet sich mindestens auf den Mailservern der Absender- und Empfängerseite. Außerdem durchläuft die E-Mail während des Versandes womöglich weitere Server, die somit wenigstens temporär zur E-Mail Zugang erhalten.
Gelingt es einem Angreifer, in diese E-Mail-Konten oder -Server einzudringen, erhält er dadurch Zugang zum Schlüssel und kann die E-Mails, die u.U. gerade zum Schutz vor einem solchen Angreifer verschlüsselt wurden, lesen.

4.3 Cloud

Zur Verwendung einer Cloud, die zu Fliehosting-Zwecken dient (z.B. Dropbox[10]), muss diese auf Desktop-Rechner und Smartphone eingerichtet sein. Der Export des Schlüssels läuft wie in 4.1 beschrieben. Anstatt auf die SD-Karte wird die Schlüsseldatei hier auf den Fileserver übertragen.
Nun bieten sich dem Nutzer zwei Möglichkeiten: Die Datei kann entweder aus OpenKeychain importiert werden. Dabei gleicht der Vorgang dem der SD-Karte bis auf den Unterschied, dass nicht auf den lokalen, sondern auf den Online-Speicher zugegriffen wird. Andererseits kann die Datei aus der App des entsprechenden Cloud-Anbieters geladen werden. Hier gleicht das Vorgehen dem des Importierens via Dateimanager in 4.1. Der Nutzer bekommt auch den Dialog aus 1 zu sehen und verwendet dieselben Schritte im Anschluss.
Ein größeres Problem als bei der SD-Karte oder der E-Mail stellt hier ein schlecht gewähltes Passwort dar. Ist die Schlüsseldatei in die Cloud geladen, können alle damit verknüpften Geräte auf die Datei zugreifen. Dies sind folglich potentiell mehr Anwendungen. Ebenso hat der Cloudanbieter Zugriff auf die Datei. Das erhöht die Gefahr, dass ein Angreifer Zugang zu der Datei und damit zu dem Schlüssel erhält, insbesondere wenn die Schlüsseldatei nicht verschlüsselt wurde.

[10] https://www.dropbox.com/

5 Eigene Lösung

Im folgenden Kapitel geben wir zuerst einen Überblick über die Theorie, wie zwei beliebige Geräte eine sichere Verbindung aufbauen können. Danach gehen wir konkret auf die Implementierung in Java ein.

5.1 Protokoll

Damit die übermittelten Daten symmetrisch verschlüsselt und authentifiziert werden können, müssen beide Geräte einen gemeinsamen privaten Schlüssel kennen. Dazu sollen zwei Arten unterstützt werden, eine sichere Verbindung aufzubauen, die je nach Hardwarespezifikation der Geräte gewählt werden. Besitzt ein Gerät eine Kamera und das andere Gerät ein Display, so kann das gemeinsame Geheimnis alleine dadurch übertragen werden, dass eines der beiden Geräte einen QR-Code anzeigt und dieser von dem anderen Gerät eingescannt wird. In dem QR-Code befindet sich ein geheimer Schlüssel, mit dem die Daten, die über das Netzwerk gesendet werden, verschlüsselt und authentifiziert werden. Zusätzlich befindet sich auch noch die IP-Adresse und der Port in dem QR-Code, sodass die Geräte ohne weitere Nutzerinteraktion einen sicheren Kanal aufbauen können. Dazu verbindet sich das Gerät, das den QR-Code gescannt hat, mit dem Server des anderen Geräts. Der Server überprüft danach, ob der Client die Nachrichten entschlüsseln kann, in dem der Server eine zufällige Zahl verschlüsselt und dem Client schickt. Der Client muss die Nachricht entschlüsseln, die Zahl um eins erhöhen, verschlüsseln und an den Server senden. Der Server überprüft danach, ob der Client die richtige Nachricht geschickt hat. Ist dies nicht der Fall, bricht der Server die Verbindung ab, andernfalls konnte eine sichere Verbindung aufgebaut werden.
Wenn keines der Geräte eine Kamera besitzt, aber bei beiden Geräten ein Display vorhanden ist, kann ein sicherer Schlüsselaustausch durchgeführt werden. Hierbei muss der Nutzer allerdings die IP-Adresse und den Port des anderen Geräts, zu dem er sich verbinden möchte, manuell eingeben. Diese Informationen zeigt ihm das andere Gerät an.
Um ein gemeinsames Geheimnis zu generieren, müssen die beiden Geräte einen Diffie-Hellman-Schlüsselaustausch [DH76] vollziehen. Dazu generien Alice und Bob jeweils mittels der Schlüsselerzeugungsfunktion ($Gen()$) einen öffentlichen und privaten Schlüssel. Danach tauschen sie ihre öffentlichen Schlüssel aus und berechnen ein gemeinsames Geheimnis. Dies stellt Abbildung 2 dar. Der Diffie-Hellman-Schlüsselaustausch ist nicht

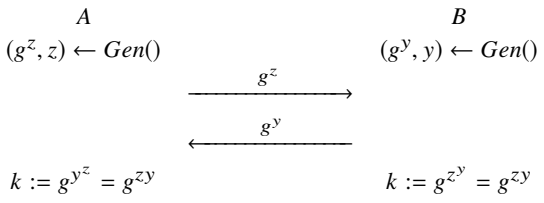

Abb. 2: DH-Schlüsselaustausch

gegen einen Mittelsmannangriff geschützt, bei dem der Angreifer jeweils Alice und Bobs öffentlichen Schlüssel abfängt und stattdessen seinen öffentlichen Schlüssel schickt. Deshalb verwenden wir Short-Authentication-Strings (SAS) [Va05], um sicherzustellen, dass kein solcher Angriff stattgefunden hat. Short-Authentication-Strings benutzt ein Commitment-Scheme. Ein Commitment beinhaltet eine Nachricht, die erst durch Bekanntgabe eines Decommitments lesbar wird. Das ermöglicht es Alice und Bob jeweils eine Nachricht zu senden, die sie erst zu einem späteren Zeitpunkt preisgeben, aber nicht verändern können. Unser gewähltes Commitment-Scheme [HM96] basiert auf kollisionsresistenten Hashfunktionen (H) und benutzt eine 2-universal Hashfunktion (h(x):= (Ax + B) mod p), um den Inhalt des Commitments erst später zu veröffentlichen.
Zusätzlich benutzen wir noch eine HMAC-Funktion (tag t := mac(x,m), wobei m die Nachricht und x ein geheimer Schlüssel ist). In Abbildung 3 ist unser Protokoll dargestellt,

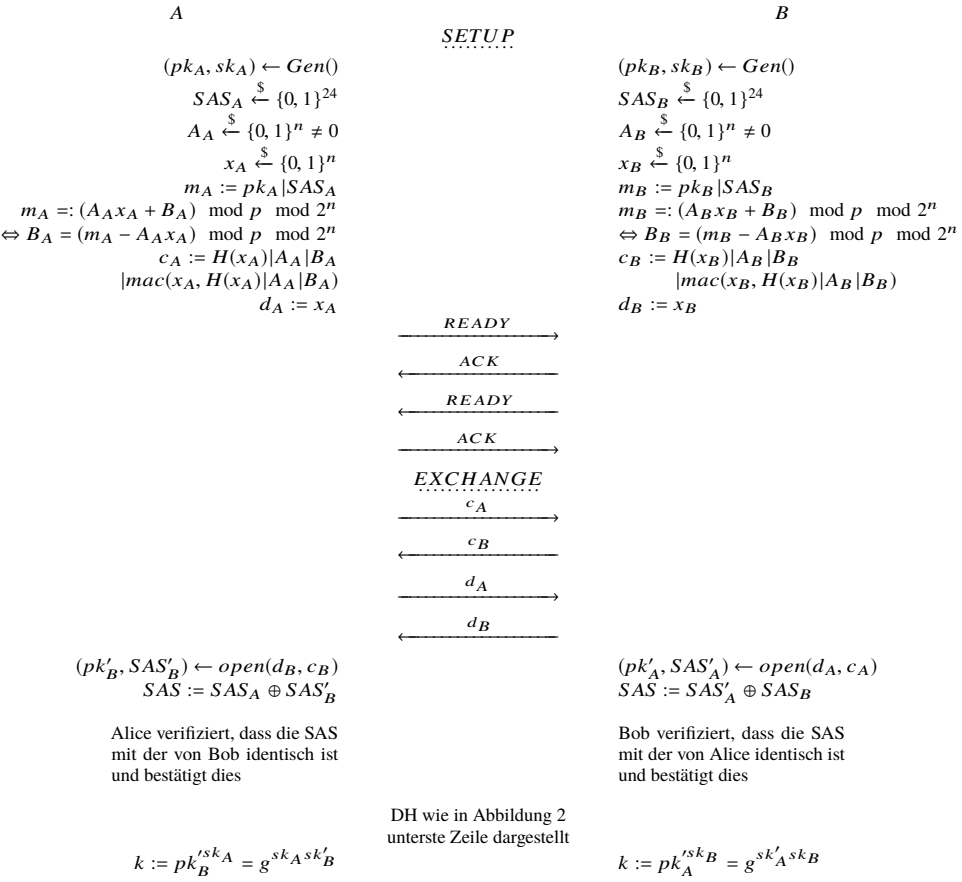

Abb. 3: Sicherer DH-Schlüsselaustausch

dabei ist m die Nachricht, c das Commitment und d das Decommitment. p ist eine Primzahl

mit $p \geq 2^n$, wobei n die Bitlänge der Nachricht (Bitlänge des öffentlichen Schlüssels plus 24 bit für die SAS) ist. Der Operator | konkateniert die Operanden miteinander.
Das vorherige $READY$, ACK dient nur dazu, dem anderen Gerät zu signalisieren, dass das Commitment erstellt wurde, da das Protokoll beim EXCHANGE einen harten Timeout von 5 Sekunden hat, während der Timeout beim SETUP bei 5 Minuten liegt.
Nachdem Alice bzw. Bob das Decommitment des anderen erhalten hat, wird zuerst überprüft, ob der Tag der vorherigen Nachricht korrekt ist. Ist dies nicht der Fall, wird der Schlüsselaustausch sofort abgebrochen. Andernfalls, wird das Commitment geöffnet, um die Nachricht zu erhalten.
Die resultierenden 24 Bit werden in sechs 4-Bit Blöcke aufgeteilt, die einen englischen Satz bilden. Jeder Block stellt einen Index in einem Array (Indizes 0-15) dar, der zu einem anderen englischen Wort aufgelöst wird. Jeder Block wird aus einer anderen Kategorie von Wörtern gezogen. Der Satz „Max owns nine heavy red books“ ergibt sich aus den 24 bit, indem die ersten 4 bit den Namen, die nächsten 4 bit das Verb, usw. bestimmen. Insgesamt gibt es $2^{4^6} = 16^6$ mögliche englische Sätze. Die englischen Wörter, die den Satz bilden, sind so gewählt, dass keine ähnlichen Sätze entstehen. Nachdem der Satz auf beiden Geräten angezeigt wird, muss der Nutzer auf beiden Geräten bestätigen, dass sie den selben Satz anzeigen.
Sind die Sätze nicht identisch, so gab es einen Mittelsmannangriff. Sind die Sätze identisch, war der Schlüsselaustausch erfolgreich und k ist der private Schlüssel, der den sicheren Kanal zwischen den Geräten ermöglicht.

5.2 SecureDataSocket

Um einen sicheren Kanal zwischen zwei Geräten aufzubauen, haben wir die Java-Bibliothek SecureDataSocket[11] entwickelt. Diese ist plattformübergreifend und lässt sich damit gut in bestehende Lösungen integrieren. Dabei soll SecureDataSocket die normalen Operationen (read/write), die auf einem Netzwerksocket möglich sind, zur Verfügung stellen und sicherstellen, dass alle Daten nur verschlüsselt und authentifiziert über das Netzwerk geschickt werden. Die Verwendung ist vorrangig für das lokale Netzwerk gedacht, da die Kommunikation direkt zwischen beiden am Austausch beteiligten Geräten passiert. SecureDataSocket implementiert die im vorherigen Abschnitt beschrieben Protokolle, um einen sicheren Kanal zwischen den Geräten aufzubauen.

5.3 OpenKeychain

Zur Übertragung von Schlüsseln in der Android-App OpenKeychain wurde die beschriebene Bibliothek SecureDataSocket verwendet. Dabei sollte der Schlüsselaustausch sowohl zwischen zwei Android-Geräten als auch zwischen einem Android-Gerät und einem

[11] https://github.com/MrJabo/SecureDataSocket/

Computer möglich sein. Die Übertragung mit einem Computer erfolgt mit Hilfe von Keylivery. Beide Geräte müssen sich dafür im gleichen lokalen Netzwerk befinden. Zum Verbindungsaufbau zur Übertragung eines Schlüssels wird ein QR-Code verwendet.

5.4 Keylivery

Keylivery[12] ist eine Java-Desktop-Anwendung, die dem Nutzer hilft, seine Schlüssel von seinem Desktop-Rechner auf sein Smartphone zu importieren. Es bildet eine grafische Darstellung um gpg[13] und lässt den Nutzer seinen gpg-Schlüssel zum Exportieren auswählen. Dazu zeigt es nach der Auswahl einen QR-Code an, der Verbindungsdetails wie IP-Addresse, Port und ein Geheimnis enthält. Den QR-Code muss der Nutzer einscannen, um eine Verbindung zu Keylivery aufzubauen. Der Schlüssel wird durch die SecureDataSocket-Bibliothek verschlüsselt und authentifiziert übertragen.

5.5 Cognitive Walkthrough

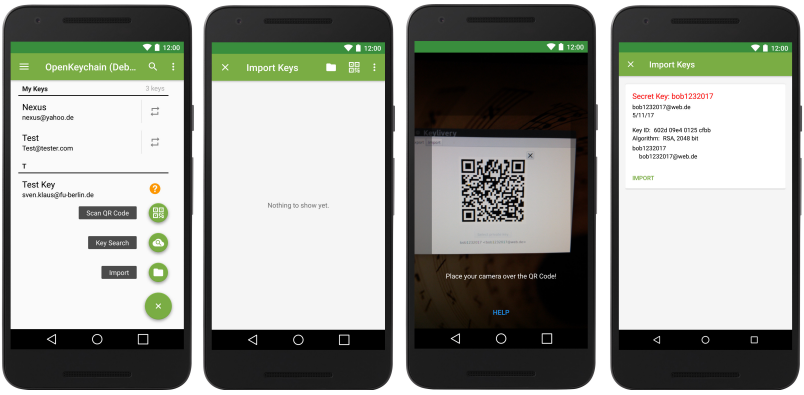

Abb. 4: Importieren eines Schlüssels mit QR-Code zum Verbindungsaufbau in OpenKeychain

Zur Verwendung unserer Methode muss zunächst auf einem Desktop-Rechner Keylivery installiert und gestartet werden. Nun kann der Button zur Schlüsselauswahl angeklickt werden. Danach öffnet sich ein Dropdown-Menü, aus dem der zu exportierende Schlüssel ausgewählt wird. Daraufhin wird ein QR-Code angezeigt, der nun aus der OpenKeychain-App heraus gescannt werden muss. Hier ist kritisch anzumerken, dass dem Nutzer deutlicher gemacht werden muss, dass er nun OpenKeychain zu öffnen hat. Dies ließe sich beispielsweise durch mehr Hinweistexte verbessern.
Anschließend wächselt der Nutzer zu seinem Smartphone und öffnet OpenKeychain.

[12] `https://github.com/svenklaus/keylivery`
[13] `https://www.gnupg.org/`

Zum Importieren eines neuen Schlüssels kann man im Hauptmenü der App den Button „Import“ auswählen, wie im linken Bild von Abbildung 4 dargestellt. In dieser Ansicht gab es bereits vor unserer Implementierung die Möglichkeit, als Datei gespeicherte Schlüssel und Schlüssel aus der Zwischenablage zu importieren. Über den neu hinzugefügten Button in der Menüleiste, auf dem ein QR-Code abgebildet ist, kann der Schlüssel auch aus dem lokalen Netzwerk importiert werden (siehe zweites Bild von links). Wird dieser Button ausgewählt, öffnet sich eine Kameraansicht zum Scannen eines QR-Codes. Damit wird nun der von Keylivery präsentierte Code gescannt. Sobald der QR-Code eingelesen wurde, schließt sich die Kameraansicht und der importierte Schlüssel wird angezeigt (siehe rechtes Bild). Es wird sowohl der öffentliche als auch der private Schlüssel übertragen. In der Ansicht kann der Benutzer sich davon überzeugen, dass der richtige Schlüssel übertragen wurde und hat die Möglichkeit, diesen in OpenKeychain zu importieren oder den Vorgang abzubrechen.

6 Zusammenfassung und Diskussion

6.1 Vergleich von Verfahren

Betrachtet man die länger vorhandenen Verfahren, stellt man fest, dass sie alle am Export aus gpg scheitern können. Wird dabei ein unsicheres oder gar kein Passwort gewählt, ist es einem Angreifer möglich, den privaten Schlüssel zu erlangen. Das von uns implementierte Verfahren vermeidet hingegen Schritte, die kritisch für den Nutzer sein können. Das wird sowohl durch die vom Nutzer unabhängige Verschlüsselung erreicht, als auch durch direkte Kommunikation der beiden Endgeräte im lokalen Netzwerk, die weitere Parteien von der Kommunikation ausschließt.
Gleichzeitig ist das neue Verfahren deutlich besser nutzbar. In den vorher vorhandenen Verfahren sah sich der Nutzer mit dem Übertragen eines Passwortes konfrontiert, dass, um als sicher zu gelten, lang und möglichst zufällig gewählt sein sollte. Ebenso hatte er in einigen Fällen mit verwirrenden Dialogen zu tun. Stellt man dem das neue Verfahren gegenüber, ist dieses deutlich leichter zu nutzen. Der Nutzer muss auf Desktop-Seite einzig Keylivery aufrufen und einen Schlüssel auswählen. Auf der Smartphone-Seite muss er die Importfunktion in OpenKeychain aufrufen und den von Keylivery angezeigten QR-Code einscannen. Die Eingabe des Passworts entfällt.
Als Hemmnis für den Nutzer ist anzumerken, dass Keylivery auf dem Desktop-Rechner erst installiert werden muss. Dagegen lässt sich einwenden, dass Keylivery nur einen Prototypen darstellt. Unsere Hoffnung ist es, dass die Funktionalität von Keylivery direkt in bestehende Verschlüsselungsanwendungen eingebaut wird. Dadurch ließen sich unter diesen Programmen leicht Schlüssel austauschen, auch ohne Beteiligung von OpenKeychain.
Zum Zeitpunkt der Einreichung besteht für den Nutzer die Möglichkeit den falschen Scanvorgang zu wählen, da neben unserem ein weiterer zum Verifizieren von öffentlichen Schlüsseln existiert. Das beabsichtigen wir jedoch durch einen Scanvorgang, der für beide Anwendungsfälle genutzt werden kann, zu beheben.

Auch ließe sich als Kritik anführen, dass in einer Studie [KFR09] die Verwendung von QR-Codes im Verhältnis zu anderen Methoden des Device Pairings sehr schlecht abschnitt. Das wird allerdings von den Autoren der Studie auch mit einer unzureichenden Implementierung, die häufig zu Versagen führte, begründet. Allerdings führen sie auch an, dass ihre Probanden häufig irritiert von den QR-Codes waren und nicht wussten, warum sie diese scannen sollten. Da die Studie aus dem Jahr 2009 stammt und sich durch die steigende Verbreitung von Smartphones seitdem auch die Verwendung von QR-Codes gestiegen ist, sind wir der Meinung, dass heutige Nutzer besser mit ihnen umzugehen wissen.

6.2 Zukünftige Arbeit

In künftigen Arbeiten ist es notwendig, die Resultate, die hier durch den Cognitive Walkthrough ermittelt wurden, in einer Studie mit Probanden zu überprüfen. Hierbei ist es insbesondere wichtig, unsere Hypothese zum Umgang von Nutzern mit QR-Codes zu prüfen.
Zukünftig kann die Implementierung in anderen Anwendungen integriert werden, um auch hier die Nutzbarkeit zu verbessern. Dabei bieten sich auch Anwendungen an, die nicht mit E-Mail in Verbindung stehen. So ließen sich beispielsweise auch die privaten Schlüssel von Cryptomessengern mit dieser Methode leicht transferieren.
Eine Implementierung des Protokolls in Desktop-Anwendungen wie dem Enigmail-Plugin erhöht die Benutzbarkeit. Damit muss der Nutzer keine neue Anwendung installieren und kann leichter, evtl. ohne vorher in die OpenKeychain FAQ sehen zu müssen, seine Schlüssel exportieren.
Mit der SecureDataSocket-Bibliothek kann auch ein grundsätzlicher Dateitransfer umgesetzt werden. Dies kann das Sichern von Fotos auf dem Smartphone per Backup auf einem anderen Gerät leicht ermöglichen und sicher gestalten. Ebenso lassen sich damit Kontaktdaten sehr einfach austauschen.
In künftigen Anwendungsfällen lässt sich die Benutzbarkeit auch sehr einfach erhöhen, falls die beiden Geräte keinen QR-Code Scanner besitzen, indem sich die Geräte über die Broadcast-IP-Adresse automatisch finden und das ebenfalls im SecureDataSocket verwendete Commitmentscheme verwendet wird.
Abschließend sei erwähnt, dass die Entwickler von OpenKeychain unsere Idee mittels tls-psk implementiert haben.

Danksagung

Wir danken Oliver Wiese und Prof. Dr. Volker Roth für die Organisation des Softwareprojektes, sowie die wissenschaftliche Betreuung. Ebenfalls danken wir Sven Klaus.

Literaturverzeichnis

[DH76] Diffie, W.; Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory, 22(6):644–654, Nov 1976.

[Fa13] Farb, Michael; Lin, Yue-Hsun; Kim, Tiffany Hyun-Jin; McCune, Jonathan M.; Perrig, Adrian: Safeslinger: Easy-To-Use And Secure Public-Key Exchange. In: MOBICOM. 2013.

[GM05] Garfinkel, Simson L.; Miller, Robert C.: Johnny 2: A User Test of Key Continuity Management with S/MIME and Outlook Express. In: Proceedings of the 2005 Symposium on Usable Privacy and Security. SOUPS '05, ACM, New York, NY, USA, S. 13–24, 2005.

[HM96] Halevi, Shai; Micali, Silvio: Practical and Provably-Secure Commitment Schemes from Collision-Free Hashing. In (Koblitz, Neal, Hrsg.): Advances in Cryptology — CRYPTO '96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, S. 201–215, 1996.

[KFR09] Kainda, Ronald; Flechais, Ivan; Roscoe, A. W.: Usability and security of out-of-band channels in secure device pairing protocols. In (Cranor, Lorrie Faith, Hrsg.): Proceedings of the 5th Symposium on Usable Privacy and Security, SOUPS 2009, Mountain View, California, USA, July 15-17, 2009. ACM International Conference Proceeding Series. ACM, 2009.

[NR10] Nguyen, Long Hoang; Roscoe, A. W.: Authentication Protocols Based on Low-Bandwidth Unspoofable Channels: A comparative survey. Journal of Computer Security, 19:139–201, 2010.

[Po92] Polson, Peter G; Lewis, Clayton; Rieman, John; Wharton, Cathleen: Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. International Journal of man-machine studies, 36(5):741–773, 1992.

[Ru15] Ruoti, S.; Andersen, J.; Zappala, D.; Seamons, K.: Why Johnny Still, Still Can't Encrypt: Evaluating the Usability of a Modern PGP Client. ArXiv e-prints, Oktober 2015.

[Ru16] Ruoti, Scott; Andersen, Jeff; Hendershot, Travis; Zappala, Daniel; Seamons, Kent: Private Webmail 2.0: Simple and Easy-to-Use Secure Email. In: Proceedings of the 29th Annual Symposium on User Interface Software and Technology. UIST '16, ACM, New York, NY, USA, S. 461–472, 2016.

[Ur15] Ur, Blase; Noma, Fumiko; Bees, Jonathan; Segreti, Sean M; Shay, Richard; Bauer, Lujo; Christin, Nicolas; Cranor, Lorrie Faith: i added '!'at the end to make it secure": Observing password creation in the lab. In: Proc. SOUPS. 2015.

[Va05] Vaudenay, Serge: Secure Communications over Insecure Channels Based on Short Authenticated Strings. In: Proceedings of the 25th Annual International Conference on Advances in Cryptology. CRYPTO'05, Springer-Verlag, Berlin, Heidelberg, S. 309–326, 2005.

[WT99] Whitten, Alma; Tygar, J. D.: Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In: Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8. SSYM'99, USENIX Association, Berkeley, CA, USA, S. 14–14, 1999.