Digital Investigation 26 (2018) S67–S76

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2018 USA — Proceedings of the Eighteenth Annual DFRWS USA

DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps

Benjamin Taubmann[*], Omar Alabduljaleel, Hans P. Reiser

University of Passau, Germany

Keywords:
Memory forensics
Semantic gap
TLS
Android

A B S T R A C T

Fast extraction of ephemeral data from the memory of a running process without affecting the performance of the analyzed program is a problem when the location and data structure layout of the information is not known. In this paper, we introduce DroidKex, an approach for partially reconstructing the semantics of data structures in order to minimize the overhead required for extracting information from the memory of applications. We demonstrate the practicability of our approach by applying it to 86Android applications in order to extract the cryptographic key material of TLS connections.

1. Introduction

Live forensics on mobile devices becomes more and more important in the day-to-day jobs of forensic investigators (Casey and Turnbull, 2011). One common use case is the decryption of encrypted communication channels such as TLS, which is the most frequently used protocol for that purpose in the Internet (Krawczyk et al., 2013).

In this paper, we discuss the problems that arise when ephemeral TLS session keys of an application should be extracted at runtime when the exact layout of the data structures and their position in memory is unknown. The most important questions we address are: Where is the data located, how can we extract it with minimal overhead and when is it available in memory?

The concrete use case of this paper is the extraction of the cryptographic key material required during a TLS connection from Android applications. One important aspect of our discussion is that we address the problem of extracting the cryptographic key material as efficient and generic as possible so that the performance and usability of an application should not be affected.

There are different approaches to address the problem of decrypting TLS connections. The most important approaches are: using a *man-in-the-middle approach*, *manipulation of the control flow* and *extraction of the key from main memory*. All three approaches have advantages and disadvantages.

The man-in-the-middle approach requires that the client application do not implement certificate pinning, i.e., that it communicates with the server even if the server does not present a valid certificate. However, many Android applications already use this technique (Fahl et al., 2012). To bypass certificate pinning, the certificate of the proxy can be installed in the application key store. With Android Nougat it gets even harder to use this approach since applications do not trust user or admin-added CAs unless the applications allow it (Brubaker, 2016). The application of these tools can lower the security of the inspected connections (Durumeric et al., 2017; US Cert, 2017; Carnavalet et al., 2016).

The control flow of an application can be intercepted to extract information or manipulated to accept any certificate. To extract the raw communication, the data can be accessed by intercepting encrypt and decrypt functions of the crypto library. Alternatively, the control flow can be patched statically (manipulation of the binary) or dynamically (interception of function calls) so that the crypto library accepts any certificate even if it is not valid for a certain domain so that the application can be used with a proxy (Cipolloni, 2017). These approaches manipulate the normal routine of an application and make it vulnerable to attacks (Durumeric et al., 2017; US Cert, 2017; Carnavalet et al., 2016). Only extracting the encrypted communication data without being able to decrypt the network stream might lower the forensic soundness of the data.

Another method is to extract the cryptographic keys from the main memory of a process (Maartmann-Moe et al., 2009). One possible approach is to test every byte sequence in the memory of an application as a potential key to decrypt the first TLS message of

[*] Corresponding author.
E-mail addresses: bt@sec.uni-passau.de (B. Taubmann), alabdu01@gw.uni-passau.de (O. Alabduljaleel), hr@sec.uni-passau.de (H.P. Reiser).

https://doi.org/10.1016/j.diin.2018.04.013

a connection (Caragea, 2016; Taubmann et al., 2016). In our evaluation, we measured an average time of 1.35 s for taking a snapshot of an Android application on the mobile device and 15.08 s to locate the key in it with that approach (see Section 6). When the data acquisition process requires pausing the application while the snapshot is taken, this approach does not work on common Android applications that use multiple TLS connections simultaneously. Each snapshot would disrupt the usability of an application even when the key extraction process is executed on a PC or in the cloud since transferring the snapshot to an external entity can be costly and time intensive.

The contribution of the paper is twofold. First, we present an approach for locating information quickly in a memory snapshot by partially reconstructing the semantics of data structures in memory instead of scanning the full memory. Thus, this approach narrows the *semantic gap*, i.e., the difference between high-level information and its low-level representation in memory. In order to achieve that, we analyze a memory snapshot of an application in the *training phase* and try to find a path of pointers that link data structures which allows the extraction of information with minimal memory access. Afterwards, in the *normal mode*, we use that knowledge to extract information by following the pointers in corresponding data structures. Second, we provide a proof-of-concept implementation – DroidKex – that uses this approach for fast key extraction of TLS connections at run-time of Android applications and we evaluate it on Android 6 with 86different applications.

The structure of this paper is as follows: Section 2 provides brief background knowledge about SSL/TLS and the Android crypto libraries. Section 3 describes the DroidKex architecture and the interaction of its components. Section 4 describes the process of extracting the key from applications by intercepting the control flow and Section 5 describes the algorithm to find a path to the data structures that hold the information. Section 6 measures the performance of DroidKex and Section 7 compares our approach to related work on decrypting TLS communication. Finally, Section 8 concludes the paper.

2. Transport layer security

The Transport Layer Security (TLS) protocol is the successor of the Secure Sockets Layer (SSL) protocol and one of the most frequently used cryptographic protocols on the Internet (Krawczyk et al., 2013). In this paper, we will use the abbreviation TLS for both TLS and SSL. TLS provides a standardized way for exchanging cryptographic key material for establishing symmetric encrypted communication.

To initiate a new TLS connection, the client and the server exchange “Hello” messages that contain client random (CR) and server random (SR) byte sequences. After the “Hello” messages, further details may be exchanged including digital certificates depending on the selected cipher suite. Furthermore, both parties generate a pre-master secret (PS) using the key exchange algorithm (such as Diffie Hellmann) specified in the cipher suite. PS, CR, and SR are used to compute the master secret (MS). MS, CR, and SR are then used with a pseudo-random function (PRF) to derive the symmetric keys that are used to encrypt the communication and to verify the integrity of a TLS. After having the key material exchanged, both parties send a change cipher spec (CSP) TLS message to notify each other to start encrypting messages. The byte sequences MS, CR, and SR are the cryptographic key material that is extracted by DroidKex.

In order to increase the speed of negotiating new sessions with the same server, TLS implements the concept of session resumption. Therefore, the negotiated parameters, i.e., the session state, are stored either by the client (session ticket) or the server (session ID) (Salowey et al., 2008). By using one of these methods the overhead for exchanging new cryptographic key material can be reduced. DroidKexis able to handle all connections of these three cases, i.e., negotiation of new material, usage of session tickets and session IDs.

2.1. Android and SSL

Android uses at least three different layers which are important for the use of SSL/TLS functionality in applications. The first layer on top is the application itself. Each Android application is written in Java and executed by the Dalvik/ART run-time environment. Applications can either come with their own crypto routines, e.g., in a separate native library or they can use the libraries provided by Android.

The core concept behind Android's crypto system is the *Java cryptography architecture (JCA)*, an interface used by different crypto libraries and implementations. A *cryptographic service provider* implements the interface of the JCA and provides the implementation of cryptographic routines. The most common cryptography providers for Android are Bouncy Castle and AndroidOpenSSL (Elenkov, 2014).

Bouncy Castle is a pure Java implementation of cryptographic algorithms and protocols whereas AndroidOpenSSL uses Java native interface (JNI) calls to access the OpenSSL library on the native level. In Android 6, Google replaced OpenSSL with BoringSSL, which is a fork of OpenSSL with Android specific patches. The default crypto provider of Android is the “AndroidOpenSSL” provider.[1]

3. The DroidKex architecture

This section discusses the general approach of the DroidKex architecture, the assumptions under which it works and the components that are required for the proof-of-concept implementation and the requirements on the target device.

3.1. Approach and goals

The goal of the DroidKex architecture is to extract the ephemeral information required for decrypting a TLS connection of an Android application from its main memory, namely the MS which is required to derive symmetric session keys. The key extraction is executed synchronous to the control flow of the application, i.e., the MS is extracted during a TLS session. Otherwise, there is no guarantee that the MS is still in main memory because an application might free or overwrite it directly after the connection terminates. Thus, we intercept all network related *send* and *receive* functions of an application. If they are handling a TLS connection (the interception framework resolves the remote address of a file descriptor), we extract the corresponding MS by dereferencing pointers that point to a structure holding the cryptographic key material that are passed to the functions of the crypto library. Even though the layout of the data structures is known since the implementation of OpenSSL and BoringSSL is open source, the exact layout of the data structures is unknown. This is caused by the fact that it changes based on the used compiler and compiler settings as well as the version of the library.

To address this problem, we follow a precomputed path starting with pointers on the calling stack to a data structure holding the

[1] https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html.