Mobile Security Report - Android instrumentation using Frida

Lukas Treffner (11807541)
Hans-Jürgen Kleeberger (11814327)

Graz, June 12, 2023

Contents

List of Figures

1 Introduction

First of all we would like to start with a short introduction what Frida actually is and what it can do. Frida is a dynamic instrumentation framework for many platforms (one of those is Android, which we focused on). Frida allows to trace basically any given software and even modify their functionality (e.g. the Frida documentation mentions that you can build a custom logging app for an already existing app [4]).

The easiest way to get started with Frida for Android, although, this addition will be omitted from now an, as we are only covering Frida for Android in this report, is if you have a rooted android smartphone. Then you can launch the Frida-server application on the phone itself and start tracing/injecting into given programs via the command line/android debugging bridge.

If your phone is not rooted, you will need to modify the .apk file of the app you wan't to instrument with Frida. This is an alternative and obviously has the limitation on only working for this specific app.

As we currently have a rooted android phone we chose this route, as it is fairly easy to setup and allows to switch to different projects with ease. So we started thinking about possible projects and came up with a few good ideas. We focused on two ides:

1. certificate pinning fixing (see 2)
2. simple keylogger for the AnySoft Keyboard (see 3)

The first option was chosen as a short introduction to Frida as there are already various different project about disabling certificate pinning and we wanted to do something that we could not find en masse. Secondly this was chosen to connect back a little to the first assignment, to create something "familiar".

The second option was then chosen to challenge us a bit. We thought that the idea of a keylogger is rather simple, but we only could find a few posts (without answers) on how to do that.

As Android hardware we are using a OnePlus One with LineageOS 18.1 with root access.

2 Certificate Pinning fixer

To make our lives a bit easier we created our own simple Android app. This app just created a simple request to a webpage (`https://cat-fact.herokuapp.com/facts`) and displays the results (see Figure 1.

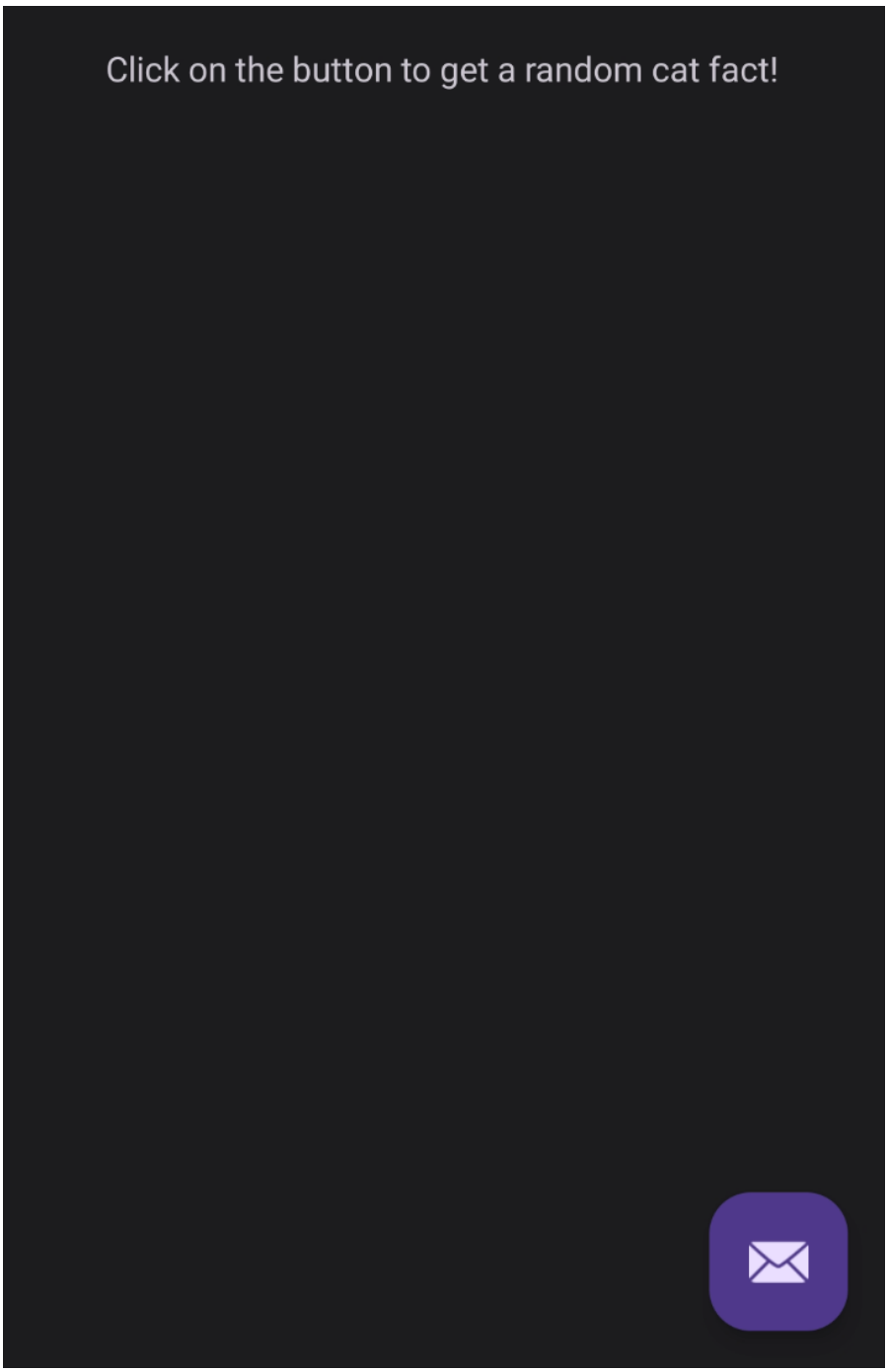

Figure 1: Demo App

The app specifically just uses the default Android TrustManager (without certificate pinning) and does not use a network security config (secure default, but without certificate pinning).

We then reused the setup from assignment 1 (mitmproxy) to test if the app would accept connections with the mitm proxy certificate (the certificate is installed as system ca due to the phone having root access). Note that all of this could have been done without root access as well (we could have just added a network security config trusting user installed certificates).

2.1 First approach

Our first approach was the TrustManager, as we though this would be the easiest option, but it turned out that although we could trace and instrument TrustManagerImpl functions (especiall - "checkTrustedRecursive" see Listing 1 for full code) it did not work all the time (maybe due to some misunderstanding on our side) and was not very cleanly implemented in our opinion. Additionally we'd loose any functionality the original developers might have implemented for their TrustManager. Additionally we did not fully fletch out the implementation due to the concerns mentioned above, so we only compared the pin in Listing 1.

A quick explanation of the code is nevertheless in order we think. Lines one to ten tell Frida which Java classes we would need. After that we register an overload for the checkTrustedRecursive function in line 13. In line 18 we use this overload to change the implementation of this method to our desired implementation (here we do not call the original method, but this can be done if needed as well). This implementation compares the first certificate of the given certificate chain to the "pinned" certificate. If they match we just log a confirmation if they do net match, the application should throw a CertificateException.

Listing 1: First implementation for Android ≥ 7

```
let X509Certificate = Java.use("java.security.cert.X509Certificate");
let MessageDigest = Java.use("java.security.MessageDigest");
let CertificateFactory = Java.use("java.security.cert.CertificateFactory")
let FileInputStream = Java.use("java.io.FileInputStream");
let JavaArrays = Java.use("java.util.Arrays");
let CertificateException =
  Java.use("javax.security.cert.CertificateException");
let Key = Java.use("java.security.Key");

const t =
  Java.use('com.android.org.conscrypt.TrustManagerImpl');

var checkTrustRecursive = t.checkTrustedRecursive.overload(
  '[Ljava.security.cert.X509Certificate;', '[B', '[B',
  'java.lang.String', 'boolean', 'java.util.ArrayList',
  'java.util.ArrayList', 'java.util.Set'
);
checkTrustRecursive.implementation = function(chain, b, c, d, e, f){
    let md = MessageDigest.getInstance("SHA256");
    let cf = CertificateFactory.getInstance("X509");
    let is = FileInputStream.$new("path_to_certificate.cert");
    let cert = Java.cast(cf.generateCertificate(is), X509Certificate);
    let pubKeyPin = Java.cast(cert.getPublicKey(), Key).getEncoded();
    let pin = md.digest(pubKeyPin);

    // this needs to match our pin;
    let certificate = Java.cast(chain[0], X509Certificate);
    let pubKey = Java.cast(certificate.getPublicKey(), Key).getEncoded();
    let fingerprint = md.digest(pubKey);

    if(!JavaArrays.equals(fingerprint, pin)){
        CertificateException.$new("SOMETHING'S␣FISHY"s);
    }else {
        console.log("IT'S␣ALL␣GOOD␣MAN!");
    }
    let trustedList = Java.use("java.util.ArrayList").$new(0);
    trustedList.add(cf);
    return trustedList;
}
```

So we started searching about other options this problem could be solved. One possible solution we came up with is adding the certificate pins to the network security config. We then looked at the Android source code ([2]), which is luckily open source. There we found out, that upon building the network security config the pins from it are added to a member variable, the PinSet (see 2):

```
private NetworkSecurityConfig(boolean cleartextTrafficPermitted, boolean hstsEnforced,
        PinSet pins, List<CertificatesEntryRef> certificatesEntryRefs) {
    mCleartextTrafficPermitted = cleartextTrafficPermitted;
    mHstsEnforced = hstsEnforced;
    mPins = pins;
```

Figure 2: The third paramter is the PinSet

In Figure 3 the PinSet implementation can be seen.

```
public final class PinSet {
    public static final PinSet EMPTY_PINSET =
            new PinSet(Collections.<Pin>emptySet(), Long.MAX_VALUE);
    public final long expirationTime;
    public final Set<Pin> pins;

    public PinSet(Set<Pin> pins, long expirationTime) {
        if (pins == null) {
            throw new NullPointerException("pins must not be null");
        }
        this.pins = pins;
        this.expirationTime = expirationTime;
    }
```

Figure 3: PinSet Implementation

The Pin implementation (see Figure 4) is pretty straight forward containing only the digest (”fingerprint”) of the certificate and the used algorithm to get this fingerprint (in our case this will be SHA256).

```
    public final String digestAlgorithm;
    public final byte[] digest;

    private final int mHashCode;

    public Pin(String digestAlgorithm, byte[] digest) {
        this.digestAlgorithm = digestAlgorithm;
        this.digest = digest;
        mHashCode = Arrays.hashCode(digest) ^ digestAlgorithm.hashCode();
```

Figure 4: Pin Implementation

2.2 Working solution

Our next step then was to tracd the constructor of the network security config object, and check if the pinset of the object would be empty (as we then would know, that the app does not implement certificate pinning via the network security config). If this pinset is empty, we could just create our own pinset and pass it on as the objects ”own” pinset. The certificate needed is loaded from a .cert file on the phone's file system (located in /data/local/tmp/).

The working solution therefore just ”mimics” the behaviour of the app, if the original developers had added the pin into the network security config. A network security config doing this would look something like this:

Listing 2: Sample nsc with pinning ([3])

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">example.com</domain>
        <pin-set expiration="2018-01-01">
            <pin digest="SHA-256">fingerprint</pin>
            <!-- backup pin -->
            <pin digest="SHA-256">fingerprint</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

As we always call the default constructor and only modify the parameters, there is little invasion into the app as well. Not only is this, in our opinion an elegant solution, but it also is really effective. When given the correct certificates this approach does not work only for our own app, but also for willhaben, Amazon and a few other apps we tested in our respective first assignment.

The code in Listing 3 again checks if the passed PinSet is empty and only then creates a new PinSet containing our given certificate. Then we just pass our created PinSet (instead of the passed empty one) to the original constructor and return this constructed object.

Listing 3: CertificatePinning Fixer finished implementation

```
let nsc = Java.use('android.security.net.config.NetworkSecurityConfig');
let X509Certificate = Java.use("java.security.cert.X509Certificate");
let MessageDigest = Java.use("java.security.MessageDigest");
let CertificateFactory = Java.use("java.security.cert.CertificateFactory")
let FileInputStream = Java.use("java.io.FileInputStream");
let Pin = Java.use("android.security.net.config.Pin");
let PinSet = Java.use("android.security.net.config.PinSet");
let Set = Java.use("android.util.ArraySet");
let Long = Java.use("java.lang.Long");
let Key = Java.use("java.security.Key");
const nsc_const = nsc.$init.overload(
  'boolean', 'boolean',
  'android.security.net.config.PinSet',
  'java.util.List'
);
nsc_const.implementation = function(a, b, pinset, l)
{
    console.log("Hooking NSC constructor");
    if(pinset.pins.value.size() === 0){
        console.log("NSC HAS NO DEFAULT PINS SET, ADDING PIN");
        let pins = Set.$new();
        let md = MessageDigest.getInstance("SHA256");
        let cf = CertificateFactory.getInstance("X509");
        let is = FileInputStream.$new("path_to_certificate.crt");
        let cert = Java.cast(cf.generateCertificate(is), X509Certificate);
        let pubKeyPin = Java.cast(cert.getPublicKey(), Key).getEncoded();
        let pin = md.digest(pubKeyPin);
        pin = Pin.$new("SHA-256", pin);
        pins.add(pin);
        pinset = PinSet.$new(pins, Long["MAX_VALUE"].value);
    }
    let NSC_OBJ; = nsc_const.call(this,a, b, pinset, l);
    return NSC_OBJ;
}
```

3 Keylogger

Certainly this task was much more of a challenge than the first one as it did not go as easily as we first planned. We chose AnySoft Keyboard as we had used this keyboard before on our own devices and knew that it was open source [1] (which greatly aided the reverse engineering process of the compiled app). So why a keylogger? We had gone through the Frida example projects listed on the Frida documentation page and many user created Frida projects. Most of them handled http sniffing or disabling certificate pinning. The first of the two we also thought of implementing but then discarded it due to it already being done multiple times. So we came up with keylogging as we thought that this would be a cool demo to show how powerful Frida really is and it hadn't been done before (or at least not in the way we intended it to do).

3.1 First steps

We first started to try tracing some functions we thought would be called from the keyboard upon key presses (onKeyUp and onKeyDown or some variations of those strings). Sadly this did not yield any results so we started consulting the source code of the keyboard. There we quickly found an implementation for keyboard suggestions. This was our first indicator that this whole thing might work. The code to suppress suggestions can be seen in Listing 4.

Listing 4: Suggestions suppression

```
let CandidateView =
  Java.use("com.anysoftkeyboard.keyboards.views.CandidateView");
CandidateView["e"].implementation = function (list, z, z2) {
    var iter = list.iterator();
    while(iter.hasNext()) {
        console.log(iter.next());
    }
    let newList = Java.use("java.util.ArrayList").$new(0);
    newList.add("Look␣at␣me␣I␣am␣the␣keyboard␣now!␣:)");
    console.log(`CandidateView.e is called: list=${list}, z=${z}, z2=${z2}`);
    this["e"](newList, z, z2);
};
```

With this knowledge we started poking around in the source code even more, finally finding the PointerTracker class which contained some logic containing the input of the characters. Sadly none of these functions could be traced with Frida, so we decided to try decompiling the apk with JADX.

3.2 Implementation

In JADX we noticed that many of the functions and classes are renamend to single letters (and or multiple letter combinations). So we first needed to figure out which

decompiled (and traceable) functions correlate to which ”clear” source code functions. Once this was done we could implement our ”logger” by just writing the user typed input to console. With this working we started refining our approach by detecting setting changes and correctly building the user input. As the source code for this project is rather big, we will not include it in this report, but it can be seen in the key_logger.py file.

A few things we needed to consider were how the app handles keyboard interactions and what information we could get from the instrumented functions. Luckily from most of the functions we were also able to get useful member variables (like the mDetector object that allows to check if a key is shifted or not). The biggest challenge was getting the current cursor position of the user (as it can be set freely without any keyboard interaction). Luckily it is possible to get the currentInputConnection from Android which has a built in method that can check how many characters are before the cursor. This can be used as the current cursor position and is currently used to insert the keypresses into the correct position in our output string.

Another challenge was what to do with the built user input? We decided to ”sent” it to a remote host, but unfortunately the AnySoft Keyboard application does not run with network permissions. Luckily Frida provides a method to start the script from python and send the user input back to the python process (host process). This is done once the user closes the keyboard (either leaves the app or triggers an action that hides the window).

One limitation comes from the fact that this logger only detects current keypresses, so if input is typed into already existing input (e.g. user opens a textfield, where something was inputted before this ”session”) the missing characters will be padded with a unique string (a combination of ”[”, ”&” and an emoji). This is done to allow correct function of the insertion and also to signal, that there might be something, that cannot be detected with this approach.

With that this task was completed as well. With some modifications, this logger could easily be embedded into the AnySoft Keyboard apk.

4 Conclusion

Frida is a very powerful tool that can be used for many things. The framework could benefit many people as there are certainly some things that can be implemented with Frida ”afterhand” (if the original developers did not care to implement it). Additionally it also can be used to ease the process of reverse engineering apps, figuring out problems in already compiled apps and many more things.

References

[1] AnySoft. *AnySoft Keyboard source code*. June 11, 2023. URL: https://github.com/AnySoftKeyboard/AnySoftKeyboard.

[2] Google. *Android source code*. June 11, 2023. URL: https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/security/net/config.

[3] Google. *Network Security Config - CertificatePinning*. URL: https://developer.android.com/training/articles/security-config#CertificatePinning (visited on 06/11/2023).

[4] Frida team. *Frida documentation*. URL: https://frida.re/docs/home/ (visited on 06/11/2023).