Bypassing SSL Pinning on Android via Reverse Engineering

Denis Andzakovic – Security-Assessment.com

15 May 2014

Table of Contents

Bypassing SSL Pinning on Android via Reverse Engineering 1

Introduction 3

Tools Used 3

The Victim 3

The Approach 4

Reversing 5

- Retrieving and Disassembling the APK 5
- Patching 6
  - Patch at class instantiation 6
  - Patch the Class 7
  - Hijacking the Keystore 8
- Repacking and Running 10

Tricks 11

- Information in Stack Traces 11
- Decompiling into Java Code 12

References 12

Introduction

Android applications may implement a feature called ‘SSL Pinning’ (1). This effectively allows the developer to specify the certificate(s) that an application will consider valid. In normal circumstances, one can add their custom certificate chain to the trusted store of their device and subsequently intercept SSL traffic. SSL pinning prevents this from being possible, as the “pinned” certificate will not match the user-supplied certificate. This introduces an issue when one wishes to intercept traffic between the application and whatever backend systems it may be communicating with.

In many cases it is possible to bypass the SSL pinning with publicly available solutions, such as iSEC Partners ‘android-ssl-bypass’ (2). In the event that this is not suitable for the target application, a more hands-on approach is required to disable the pinning. This whitepaper will detail the steps taken to unpack an application, locate the pinning handler, patch and repack. The techniques detailed in this whitepaper may also be used to achieve other goals when hacking Android applications.

The aim of this exercise is to remove the pinning in order to be able to intercept the requests with a Web proxy. The Burp Proxy (3) is used in this example.

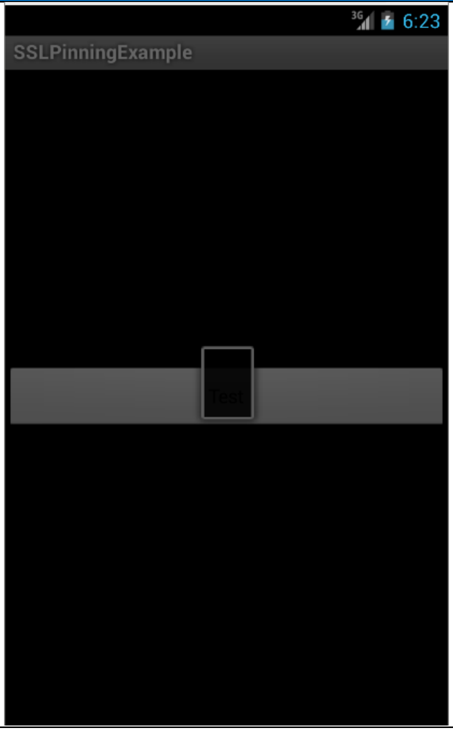
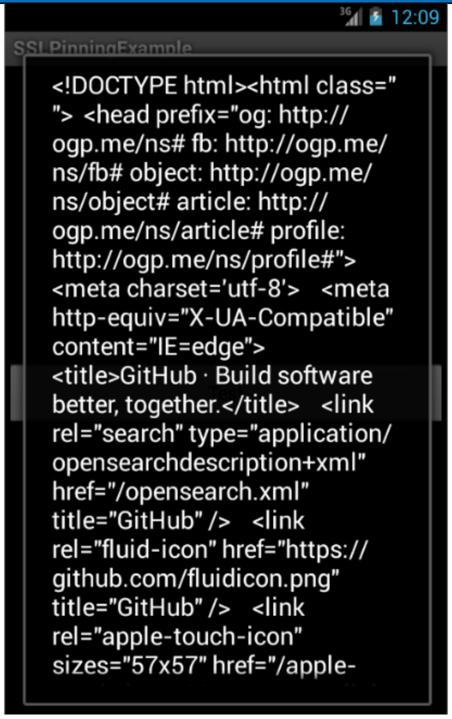
Tools Used

- Android SDK - http://developer.android.com/sdk/index.html
- Smali/baksmali - https://code.google.com/p/smali/w/list
- Signapk – https://code.google.com/p/signapk/
- Burp - http://portswigger.net/burp/

The Victim

For the purpose of this whitepaper, will we be using the SSLPinningExample application, which can be found here (http://security-assessment.com/files/documents/whitepapers/SSLPinningExample.zip). This is a very simple application that performs an HTTP request to https://github.com and displays the result of the request – in general operation, this will be the HTTP response returned from GitHub. As SSL Pinning is enabled, a Man-In-The-Middle with Burp results in an SSL error.

Attempting to use an intercepting proxy with the default application results in no response, where as a successful request returns the response data:

Failure	Success
	 <!DOCTYPE html><html class=""> <head prefix="og: http://ogp.me/ns# fb: http://ogp.me/ns/fb# object: http://ogp.me/ns/object# article: http://ogp.me/ns/article# profile: http://ogp.me/ns/profile#"> <meta charset='utf-8'> <meta http-equiv="X-UA-Compatible" content="IE=edge"> <title>GitHub · Build software better, together.</title> <link rel="search" type="application/opensearchdescription+xml" href="/opensearch.xml" title="GitHub" /> <link rel="fluid-icon" href="https://github.com/fluidicon.png" title="GitHub" /> <link rel="apple-touch-icon" sizes="57x57" href="/apple-

The Approach

In order to disable the pinning, one can disassemble the application, locate the method responsible for pinning control and remove the check. The aim is to have the client accept your own SSL certificate as valid. **Note: the proxy's certificate should be installed in the Android trusted certificate store.**

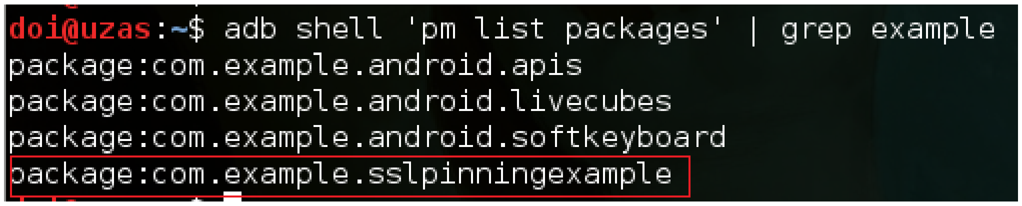
The application needs to be disassembled into Smali code (based on Jasmin syntax) (4).

The Smali code will then be patched to remove the SSL Pinning check and the application reassembled and repacked.

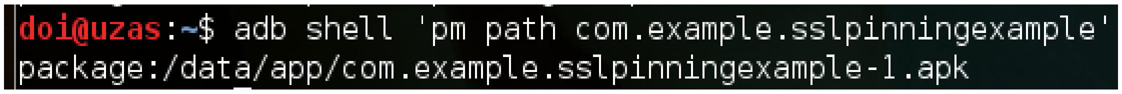
Reversing

Retrieving and Disassembling the APK

After the correct package is located, the APK file can be pulled from the device. List all installed packages, find the relevant APK for your desired package and then pull said package off the device.

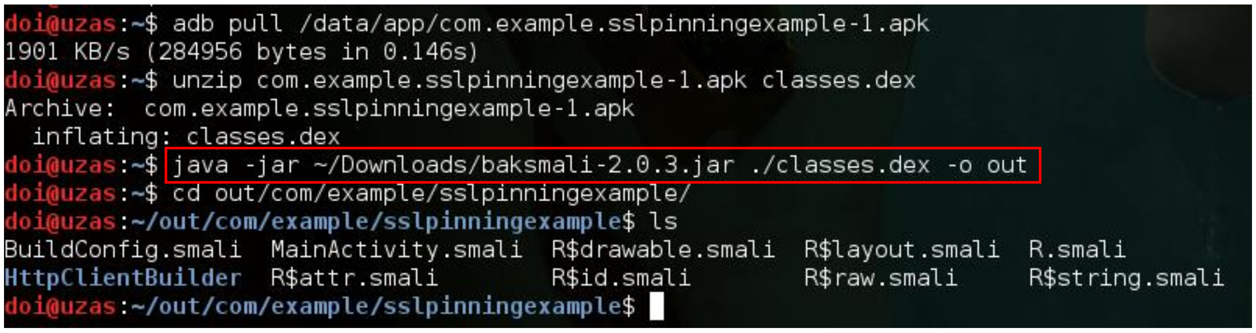


```
doi@uzas:~$ adb shell 'pm list packages' | grep example
package:com.example.android.apis
package:com.example.android.livecubes
package:com.example.android.softkeyboard
package:com.example.sslpinningexample
```



```
doi@uzas:~$ adb shell 'pm path com.example.sslpinningexample'
package:/data/app/com.example.sslpinningexample-1.apk
```

After pulling '/data/app/com.example.sslpinningexample-1.apk', unzip the APK to retrieve the classes.dex file. Baksmali is then run to disassemble the Dalvik bytecode into Smali.

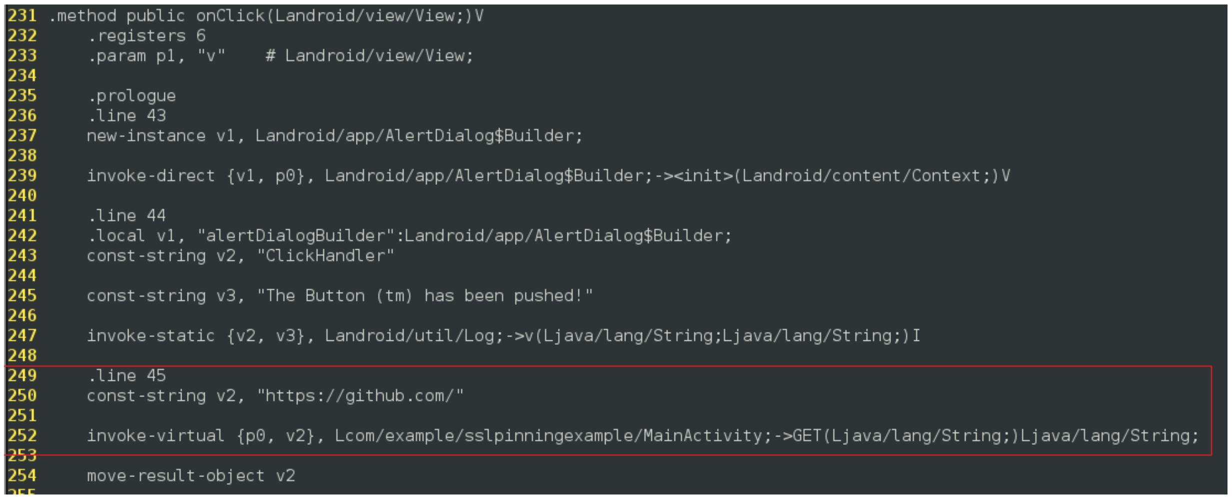


```
doi@uzas:~$ adb pull /data/app/com.example.sslpinningexample-1.apk
1901 KB/s (284956 bytes in 0.146s)
doi@uzas:~$ unzip com.example.sslpinningexample-1.apk classes.dex
Archive:  com.example.sslpinningexample-1.apk
  inflating: classes.dex
doi@uzas:~$ java -jar ~/Downloads/baksmali-2.0.3.jar ./classes.dex -o out
doi@uzas:~$ cd out/com/example/sslpinningexample/
doi@uzas:~/out/com/example/sslpinningexample$ ls
BuildConfig.smali  MainActivity.smali  R$drawable.smali  R$layout.smali  R.smali
HttpClientBuilder  R$attr.smali        R$id.smali        R$raw.smali     R$string.smali
doi@uzas:~/out/com/example/sslpinningexample$
```

Patching

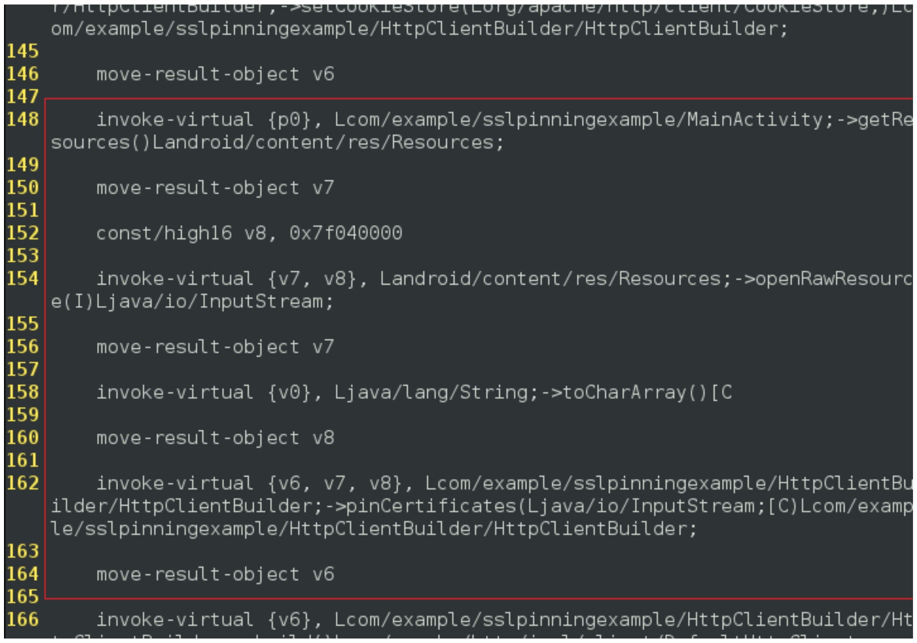
Patch at class instantiation

The first method we will discuss is patching the file that spawns the HTTPS client. Starting with `MainActivity.smali` and working back from the onClick method, we can see that the GET method is responsible for retrieving https://github.com:



```
.method public onClick(Landroid/view/View;)V
    .registers 6
    .param p1, "v"    # Landroid/view/View;

    .prologue
    .line 43
    new-instance v1, Landroid/app/AlertDialog$Builder;

    invoke-direct {v1, p0}, Landroid/app/AlertDialog$Builder;-><init>(Landroid/content/Context;)V

    .line 44
    .local v1, "alertDialogBuilder":Landroid/app/AlertDialog$Builder;
    const-string v2, "ClickHandler"

    const-string v3, "The Button (tm) has been pushed!"

    invoke-static {v2, v3}, Landroid/util/Log;->v(Ljava/lang/String;Ljava/lang/String;)I

    .line 45
    const-string v2, "https://github.com/"

    invoke-virtual {p0, v2}, Lcom/example/sslpinningexample/MainActivity;->GET(Ljava/lang/String;)Ljava/lang/String;

    move-result-object v2
```

Further analysis shows the GET method calls the `HttpClientBuilder` class to build the `DefaultHttpClient`. `DefaultHttpClient` expects a number of parameters when it is created, so our goal is to remove the parameter containing the keystore responsible for SSL Pinning (5). Working through the GET method, we see that the `pinCertificates` method of the `HttpClientBuilder` class is called. We will remove this, as well as the `pinCertificates` call's relevant variables.



```
r/HttpClientBuilder;->setCookieStore(Lorg/apache/http/client/CookieStore;)Lc
om/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;

    move-result-object v6

    invoke-virtual {p0}, Lcom/example/sslpinningexample/MainActivity;->getRe
sources()Landroid/content/res/Resources;

    move-result-object v7

    const/high16 v8, 0x7f040000

    invoke-virtual {v7, v8}, Landroid/content/res/Resources;->openRawResourc
e(I)Ljava/io/InputStream;

    move-result-object v7

    invoke-virtual {v0}, Ljava/lang/String;->toCharArray()[C

    move-result-object v8

    invoke-virtual {v6, v7, v8}, Lcom/example/sslpinningexample/HttpClientBu
ilder/HttpClientBuilder;->pinCertificates(Ljava/io/InputStream;[C)Lcom/examp
le/sslpinningexample/HttpClientBuilder/HttpClientBuilder;

    move-result-object v6

    invoke-virtual {v6}, Lcom/example/sslpinningexample/HttpClientBuilder/Ht
```

As can be seen above, lines 148 through 164 need to be removed. The application is then re-assembled and reinstalled on the device.

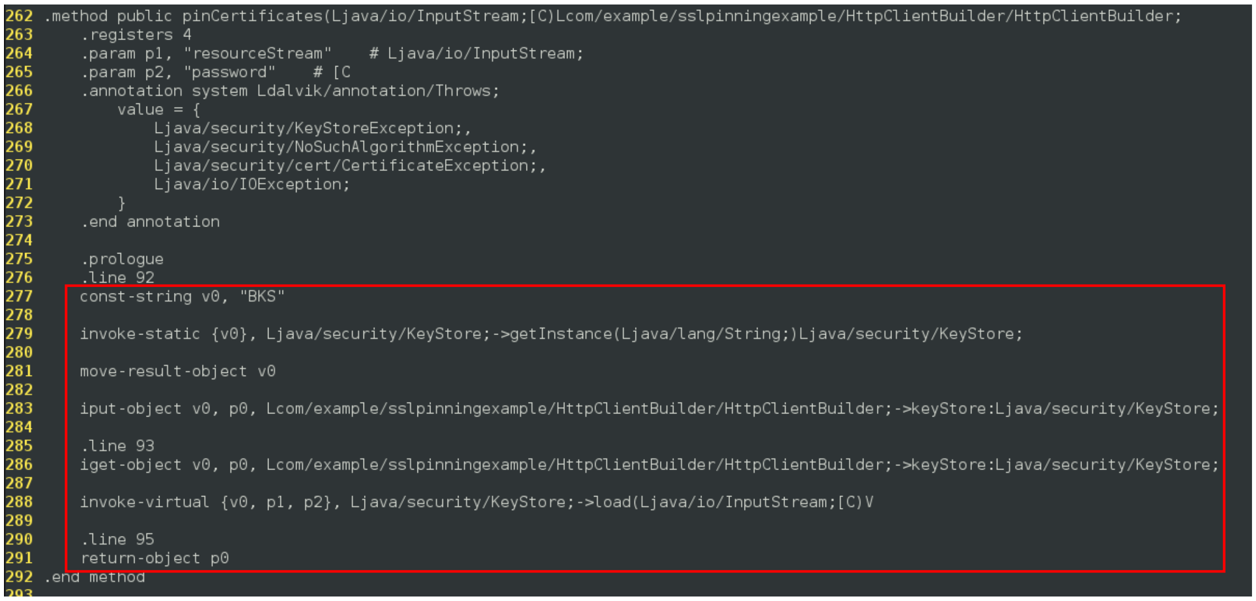
Patch the Class

A more elegant solution would be to patch the `HttpClientBuilder` class itself. The application may call this class multiple times from different locations in the code, so patching the library guarantees that we won't get stung by SSL Pinning further down the track.

The `MainActivity.smali` shows which method we need to patch within the `HttpClientBuilder` class. This is indicated by the invoke-virtual on line 162:

```
invoke-virtual {v6, v7, v8},
Lcom/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;-
>pinCertificates(Ljava/io/InputStream;[C
)Lcom/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;
```

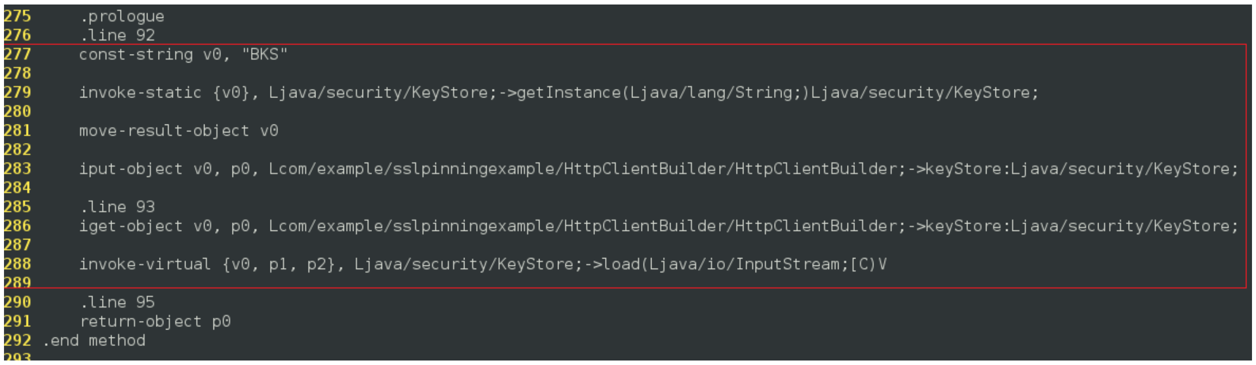
Looking at the above, we can see that the `pinCertificates` method expects an `InputStream` and a Char array as its parameters. This is enough information to locate the method within the `HTTPClientBuilder.smali` file and patch it:



```
262 .method public pinCertificates(Ljava/io/InputStream;[C)Lcom/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;
263     .registers 4
264     .param p1, "resourceStream"    # Ljava/io/InputStream;
265     .param p2, "password"    # [C
266     .annotation system Ldalvik/annotation/Throws;
267         value = {
268             Ljava/security/KeyStoreException;,
269             Ljava/security/NoSuchAlgorithmException;,
270             Ljava/security/cert/CertificateException;,
271             Ljava/io/IOException;
272         }
273     .end annotation
274
275     .prologue
276     .line 92
277     const-string v0, "BKS"
278
279     invoke-static {v0}, Ljava/security/KeyStore;->getInstance(Ljava/lang/String;)Ljava/security/KeyStore;
280
281     move-result-object v0
282
283     iput-object v0, p0, Lcom/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;->keyStore:Ljava/security/KeyStore;
284
285     .line 93
286     iget-object v0, p0, Lcom/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;->keyStore:Ljava/security/KeyStore;
287
288     invoke-virtual {v0, p1, p2}, Ljava/security/KeyStore;->load(Ljava/io/InputStream;[C)V
289
290     .line 95
291     return-object p0
292 .end method
```

The above method is what we need to patch to bypass the check. Looking at lines 264 and 265, we see this is the `pinCertificates` method expecting an `InputStream` and a character array. We don't particularly care what the method does, we just want it to return p0 (this).

This modification means that even though the `pinCertificates` method is called, the keystore information is never added to the parameters passed to the `DefaultHttpClient`.



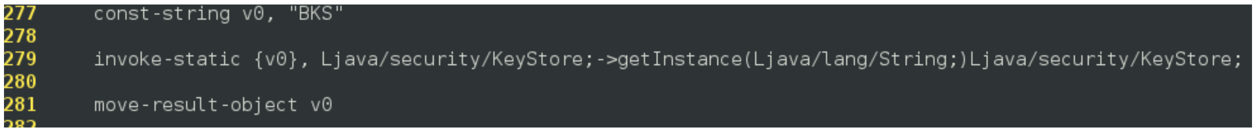
```
275     .prologue
276     .line 92
277     const-string v0, "BKS"
278
279     invoke-static {v0}, Ljava/security/KeyStore;->getInstance(Ljava/lang/String;)Ljava/security/KeyStore;
280
281     move-result-object v0
282
283     iput-object v0, p0, Lcom/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;->keyStore:Ljava/security/KeyStore;
284
285     .line 93
286     iget-object v0, p0, Lcom/example/sslpinningexample/HttpClientBuilder/HttpClientBuilder;->keyStore:Ljava/security/KeyStore;
287
288     invoke-virtual {v0, p1, p2}, Ljava/security/KeyStore;->load(Ljava/io/InputStream;[C)V
289
290     .line 95
291     return-object p0
292 .end method
```

The above lines are removed, the application re-assembled and then reinstalled on the device.

Hijacking the Keystore

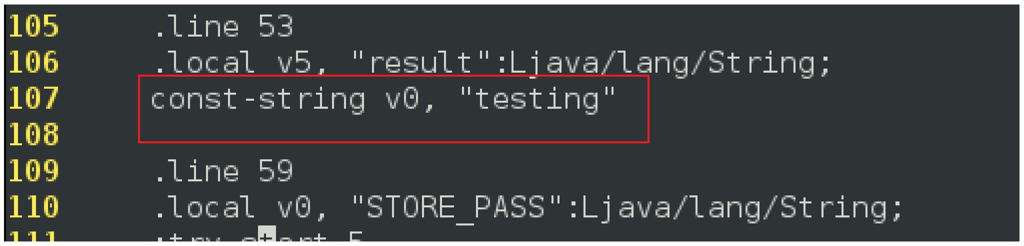
Another method is hijacking the keystore. The `DefaultHttpClient` can be passed a keystore file which contains the pins as a parameter. In order for the application to be able to be able to access this keystore, it must know the password. If the location of the keystore and the password is known, then one can add their own CA into the keystore and leave the pinning in place. This can be especially useful when the application is large, convoluted or obfuscated.

The first step is the location of the keystore. The `HttpClientBuilder` class specifies the string ‘BKS”, which is then passed to java/security/Keystore, indicating we are dealing with a BKS format keystore file. This can be seen on line 277 of the `HttpClientBuilder.smali` file:



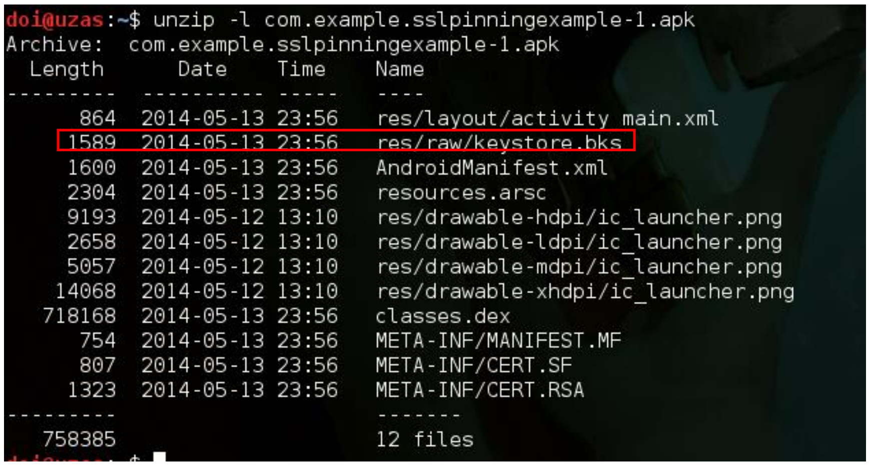
```
277     const-string v0, "BKS"
278
279     invoke-static {v0}, Ljava/security/KeyStore;->getInstance(Ljava/lang/String;)Ljava/security/KeyStore;
280
281     move-result-object v0
```

In the `MainActivity.smali` file, under the GET method, we see the STORE_PASS variable be set to “testing”, giving us the key:



```
105     .line 53
106     .local v5, "result":Ljava/lang/String;
107     const-string v0, "testing"
108
109     .line 59
110     .local v0, "STORE_PASS":Ljava/lang/String;
```

By inspecting the APK archive, we can determine the location of the keystore (in this case, `res/raw/keystore.bks`):

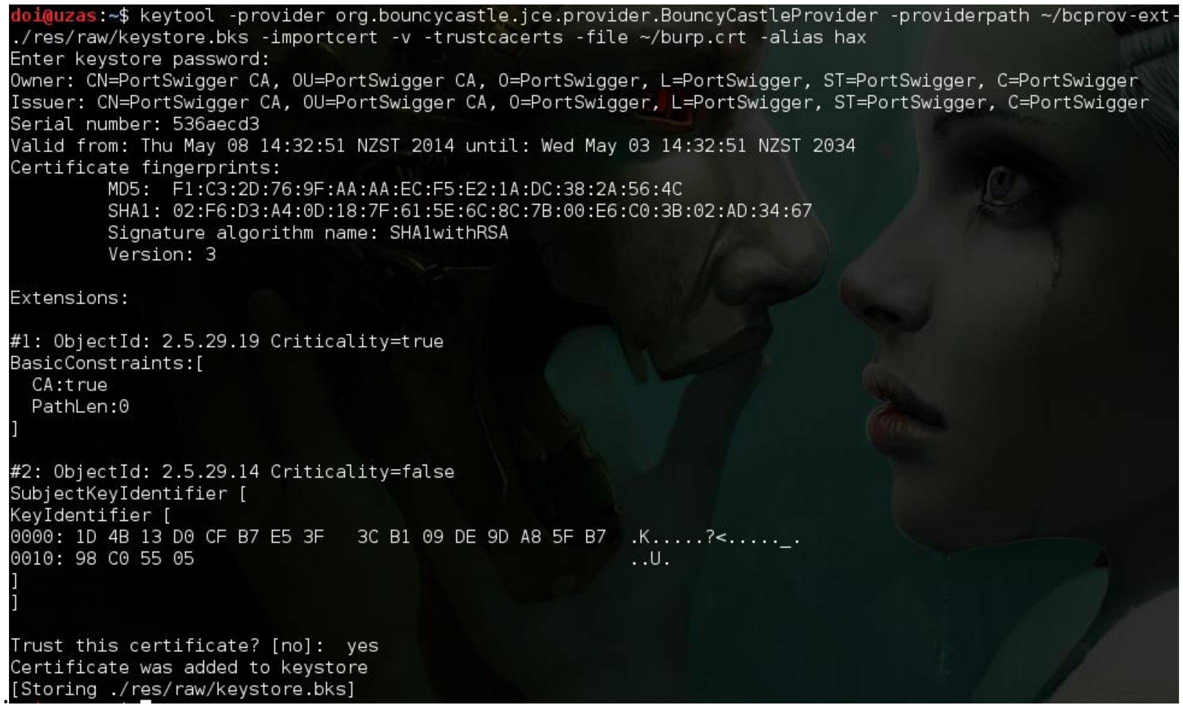


```
doi@uzas:~$ unzip -l com.example.sslpinningexample-1.apk
Archive:  com.example.sslpinningexample-1.apk
  Length      Date    Time    Name
---------  ---------- -----   ----
      864  2014-05-13 23:56   res/layout/activity_main.xml
     1589  2014-05-13 23:56   res/raw/keystore.bks
     1600  2014-05-13 23:56   AndroidManifest.xml
     2304  2014-05-13 23:56   resources.arsc
     9193  2014-05-12 13:10   res/drawable-hdpi/ic_launcher.png
     2658  2014-05-12 13:10   res/drawable-ldpi/ic_launcher.png
     5057  2014-05-12 13:10   res/drawable-mdpi/ic_launcher.png
    14068  2014-05-12 13:10   res/drawable-xhdpi/ic_launcher.png
   718168  2014-05-13 23:56   classes.dex
      754  2014-05-13 23:56   META-INF/MANIFEST.MF
      807  2014-05-13 23:56   META-INF/CERT.SF
     1323  2014-05-13 23:56   META-INF/CERT.RSA
---------                     -------
   758385                     12 files
```

After extraction, we can add our own certificate file, repack and reinstall. Due to the BKS format, we need to use the Bouncy Castle APIs (6). The keytool command is used to add the certificate:

`keytool -provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath` **`path-to-bcprov.jar`** `-storetype BKS -keystore` **`keystore-file`** `-importcert -v -trustcacerts -file` **`certificate-file`** `-alias` **`hax`**

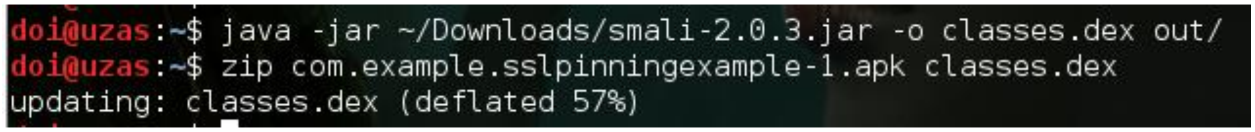
The end result should look similar to this:



```
doi@uzas:~$ keytool -provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath ~/bcprov-ext-
./res/raw/keystore.bks -importcert -v -trustcacerts -file ~/burp.crt -alias hax
Enter keystore password:
Owner: CN=PortSwigger CA, OU=PortSwigger CA, O=PortSwigger, L=PortSwigger, ST=PortSwigger, C=PortSwigger
Issuer: CN=PortSwigger CA, OU=PortSwigger CA, O=PortSwigger, L=PortSwigger, ST=PortSwigger, C=PortSwigger
Serial number: 536aecd3
Valid from: Thu May 08 14:32:51 NZST 2014 until: Wed May 03 14:32:51 NZST 2034
Certificate fingerprints:
         MD5:  F1:C3:2D:76:9F:AA:AA:EC:F5:E2:1A:DC:38:2A:56:4C
         SHA1: 02:F6:D3:A4:0D:18:7F:61:5E:6C:8C:7B:00:E6:C0:3B:02:AD:34:67
         Signature algorithm name: SHA1withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.19 Criticality=true
BasicConstraints:[
  CA:true
  PathLen:0
]

#2: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 1D 4B 13 D0 CF B7 E5 3F   3C B1 09 DE 9D A8 5F B7  .K.....?<....._.
0010: 98 C0 55 05                                        ..U.
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
[Storing ./res/raw/keystore.bks]
```

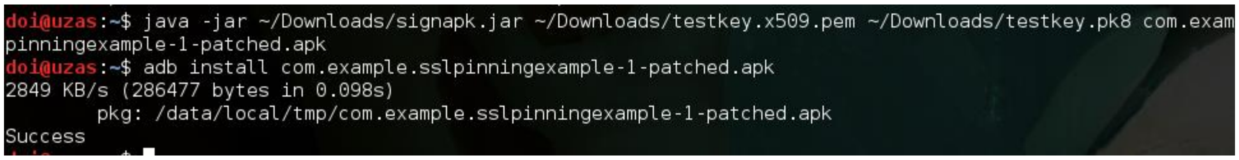
Repacking and Running

Once the modifications are made, the Smali code needs to be reassembled back into Dalvik bytecode, the new classes.dex file needs to be zipped and the signatures need to be updated. Smali is used to achieve the reassembly:



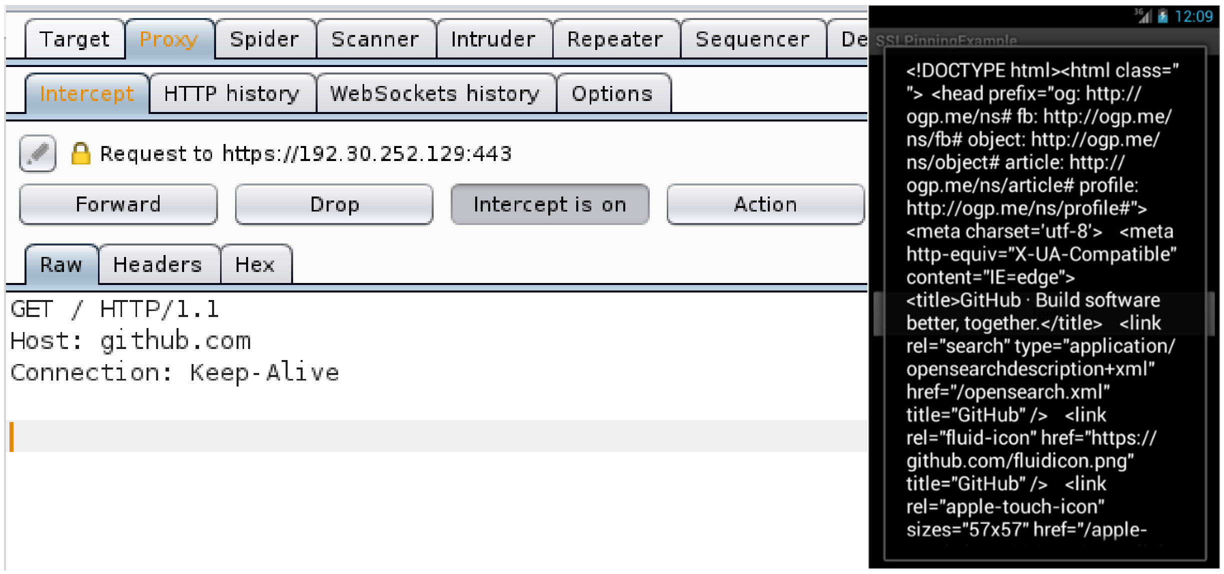
```
doi@uzas:~$ java -jar ~/Downloads/smali-2.0.3.jar -o classes.dex out/
doi@uzas:~$ zip com.example.sslpinningexample-1.apk classes.dex
updating: classes.dex (deflated 57%)
```

SignAPK is then used to update the signature, however any tool used for signing APKs will suffice. After the signing is completed, the app is installed onto the device:



```
doi@uzas:~$ java -jar ~/Downloads/signapk.jar ~/Downloads/testkey.x509.pem ~/Downloads/testkey.pk8 com.exam
pinningexample-1-patched.apk
doi@uzas:~$ adb install com.example.sslpinningexample-1-patched.apk
2849 KB/s (286477 bytes in 0.098s)
        pkg: /data/local/tmp/com.example.sslpinningexample-1-patched.apk
Success
```

SignAPK expects a certificate, a key, the original APK and the output APK as its parameters. After signing and reinstallation, the application is run. We can then intercept requests with Burp (again - make sure Burp's CA is in the OS trust store):



Request to https://192.30.252.129:443

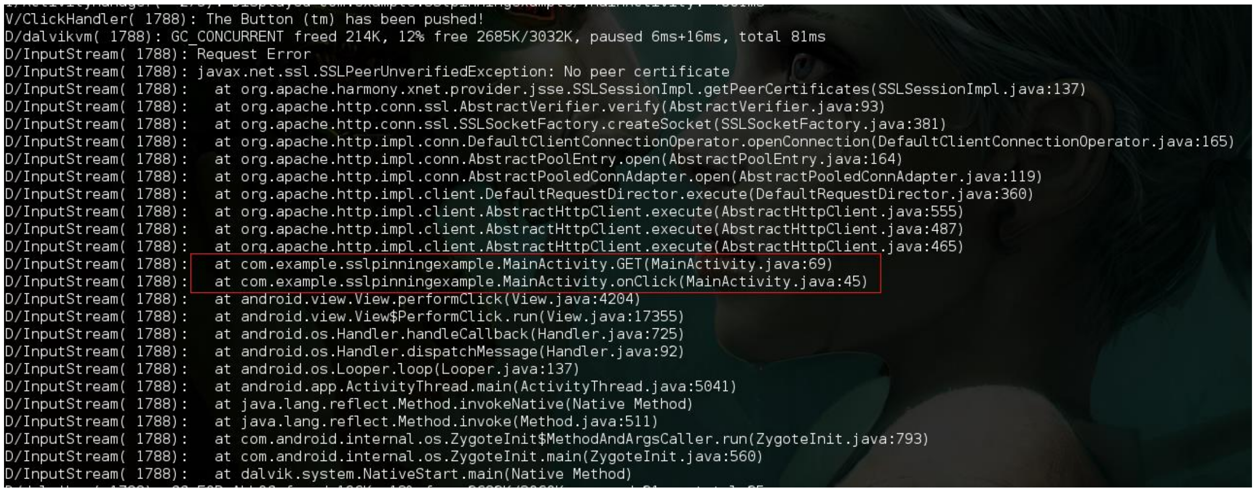
```
GET / HTTP/1.1
Host: github.com
Connection: Keep-Alive
```

<!DOCTYPE html><html class=" "> <head prefix="og: http://ogp.me/ns# fb: http://ogp.me/ns/fb# object: http://ogp.me/ns/object# article: http://ogp.me/ns/article# profile: http://ogp.me/ns/profile#"> <meta charset='utf-8'> <meta http-equiv="X-UA-Compatible" content="IE=edge"> <title>GitHub · Build software better, together.</title> <link rel="search" type="application/opensearchdescription+xml" href="/opensearch.xml" title="GitHub" /> <link rel="fluid-icon" href="https://github.com/fluidicon.png" title="GitHub" /> <link rel="apple-touch-icon" sizes="57x57" href="/apple-

Tricks

Information in Stack Traces

In the event that the application logs its stack traces, an attacker may use the stack trace to determine the location of the function responsible for the pinning. This trick can be particularly useful when faced with applications that have been obfuscated with a tool such as ProGuard (7). The SSLPinningExample application throws the following when the SSL connection fails:



```
V/ClickHandler( 1788): The Button (tm) has been pushed!
D/dalvikvm( 1788): GC_CONCURRENT freed 214K, 12% free 2685K/3032K, paused 6ms+16ms, total 81ms
D/InputStream( 1788): Request Error
D/InputStream( 1788): javax.net.ssl.SSLPeerUnverifiedException: No peer certificate
D/InputStream( 1788):   at org.apache.harmony.xnet.provider.jsse.SSLSessionImpl.getPeerCertificates(SSLSessionImpl.java:137)
D/InputStream( 1788):   at org.apache.http.conn.ssl.AbstractVerifier.verify(AbstractVerifier.java:93)
D/InputStream( 1788):   at org.apache.http.conn.ssl.SSLSocketFactory.createSocket(SSLSocketFactory.java:381)
D/InputStream( 1788):   at org.apache.http.impl.conn.DefaultClientConnectionOperator.openConnection(DefaultClientConnectionOperator.java:165)
D/InputStream( 1788):   at org.apache.http.impl.conn.AbstractPoolEntry.open(AbstractPoolEntry.java:164)
D/InputStream( 1788):   at org.apache.http.impl.conn.AbstractPooledConnAdapter.open(AbstractPooledConnAdapter.java:119)
D/InputStream( 1788):   at org.apache.http.impl.client.DefaultRequestDirector.execute(DefaultRequestDirector.java:360)
D/InputStream( 1788):   at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:555)
D/InputStream( 1788):   at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:487)
D/InputStream( 1788):   at org.apache.http.impl.client.AbstractHttpClient.execute(AbstractHttpClient.java:465)
D/InputStream( 1788):   at com.example.sslpinningexample.MainActivity.GET(MainActivity.java:69)
D/InputStream( 1788):   at com.example.sslpinningexample.MainActivity.onClick(MainActivity.java:45)
D/InputStream( 1788):   at android.view.View.performClick(View.java:4204)
D/InputStream( 1788):   at android.view.View$PerformClick.run(View.java:17355)
D/InputStream( 1788):   at android.os.Handler.handleCallback(Handler.java:725)
D/InputStream( 1788):   at android.os.Handler.dispatchMessage(Handler.java:92)
D/InputStream( 1788):   at android.os.Looper.loop(Looper.java:137)
D/InputStream( 1788):   at android.app.ActivityThread.main(ActivityThread.java:5041)
D/InputStream( 1788):   at java.lang.reflect.Method.invokeNative(Native Method)
D/InputStream( 1788):   at java.lang.reflect.Method.invoke(Method.java:511)
D/InputStream( 1788):   at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:793)
D/InputStream( 1788):   at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:560)
D/InputStream( 1788):   at dalvik.system.NativeStart.main(Native Method)
```

The trace points us to line 69 in the `MainActivity` file, saving the headache of reversing the application from scratch.

Decompiling into Java Code

Working with Smali may not be the easiest way to quickly understand what the application is doing if one is not familiar with Smali. Another approach is; after extraction, Dex2Jar (8) can be used to convert the classes.dex file into a JAR file. The resulting JAR file can then be decompiled with JD-Gui or similar (9).

References

1. Google. (n.d.). Retrieved from http://developer.android.com/training/articles/security-ssl.html
2. iSEC Partners. (n.d.). Retrieved from https://github.com/iSECPartners/android-ssl-bypass
3. Portswigger. (n.d.) Retrieved from http://portswigger.net/burp/proxy.html
4. Ben Gruver. (n.d.) Retrieved from https://code.google.com/p/smali/w/list
5. Google. (n.d.). Retrieved from http://developer.android.com/reference/org/apache/http/impl/client/DefaultHttpClient.html
6. Bouncy Castle. (n.d.). http://www.bouncycastle.org/
7. Google. (n.d.). http://developer.android.com/tools/help/proguard.html
8. pxb1...@gmail.com. (n.d.). https://code.google.com/p/dex2jar/
9. Dupuy, E. (n.d.). http://jd.benow.ca/