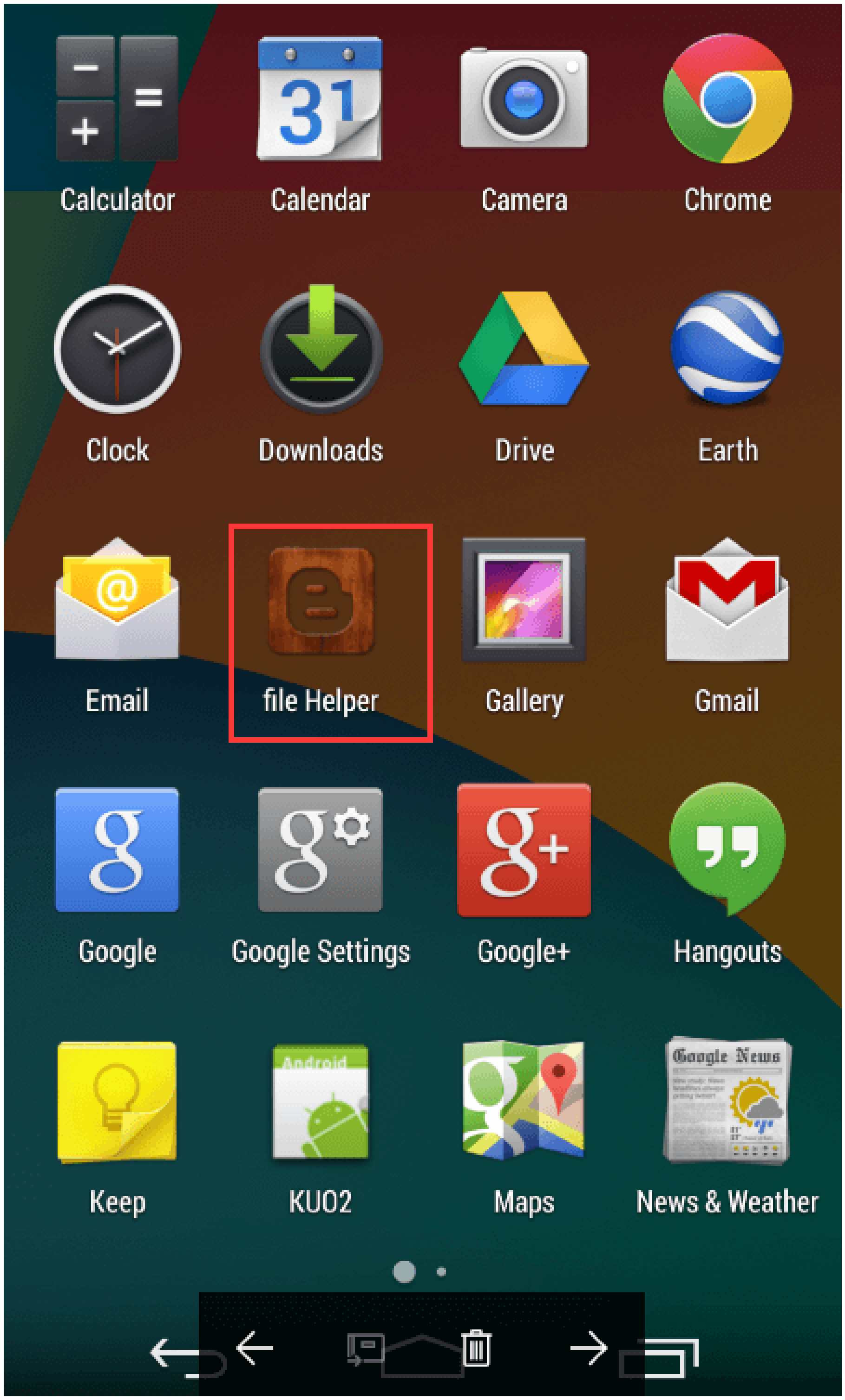
Deep Analysis of Android Rootnik Malware Using Advanced Anti-Debug and Anti-Hook, Part I: Debugging in The Scope of Native Layer

blog.fortinet.com/2017/01/24/deep-analysis-of-android-rootnik-malware-using-advanced-anti-debug-and-anti-hook-part-i-debugging-in-the-scope-of-native-layer

January 26, 2017



Threat Research

By Kai Lu | January 26, 2017

Recently, we found a new Android rootnik malware which uses open-sourced Android root exploit tools and the MTK root scheme from the dashi root tool to gain root access on an Android device. The malware disguises itself as a file helper app and then uses very advanced anti-debug and anti-hook techniques to prevent it from being reverse engineered. It also uses a multidex scheme to load a secondary dex file. After successfully gaining root privileges on the device, the rootnik malware can perform several malicious behaviors, including app and ad promotion, pushing porn, creating shortcuts on the home screen, silent app installation, pushing notification, etc. In this blog, I'll provide a deep analysis of this malware.

A Quick Look at the Malware

The malware app looks like a legitimate file helper app that manages your files and other resources stored on your Android phone.

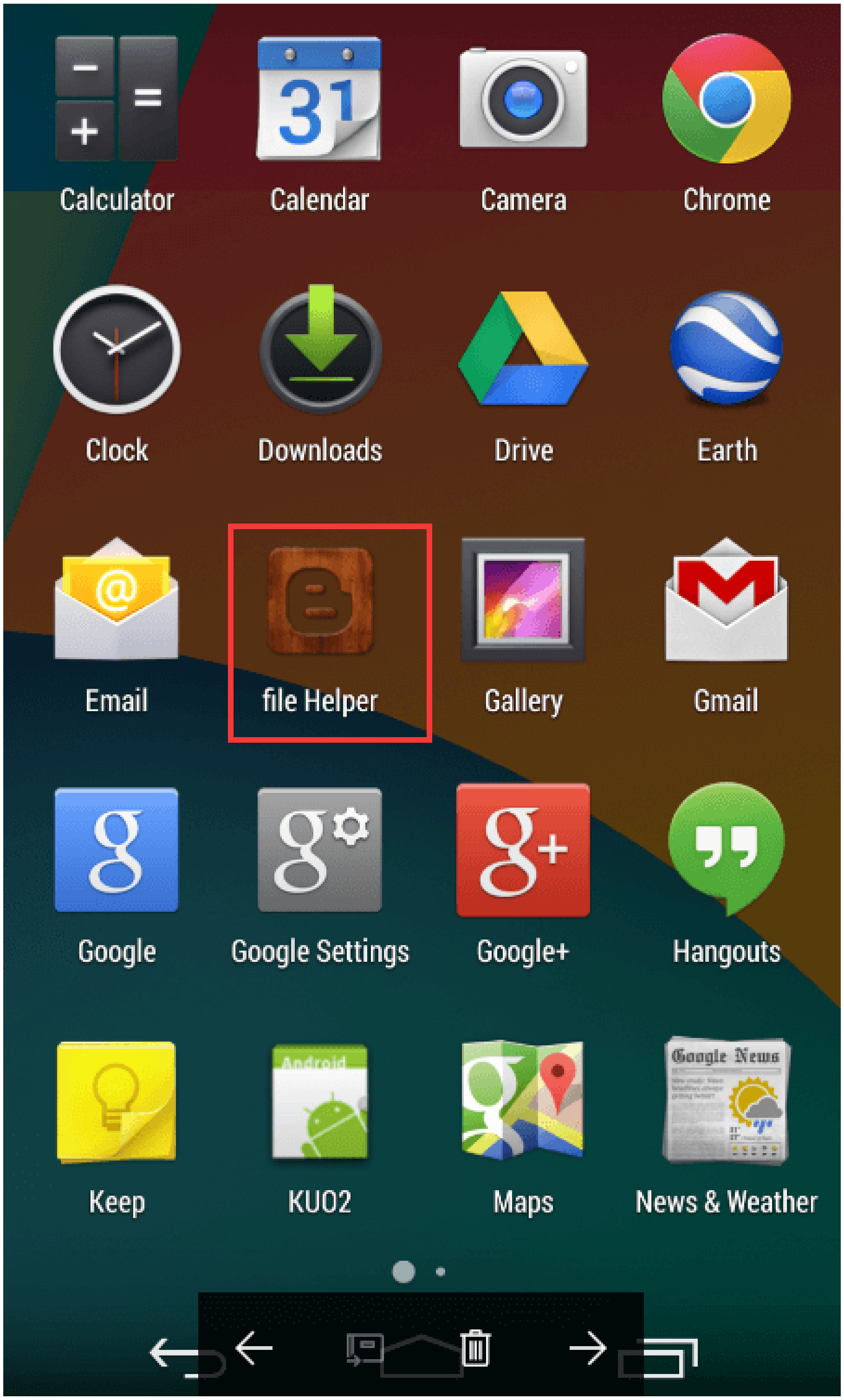


Figure 1. The malware app icon installed

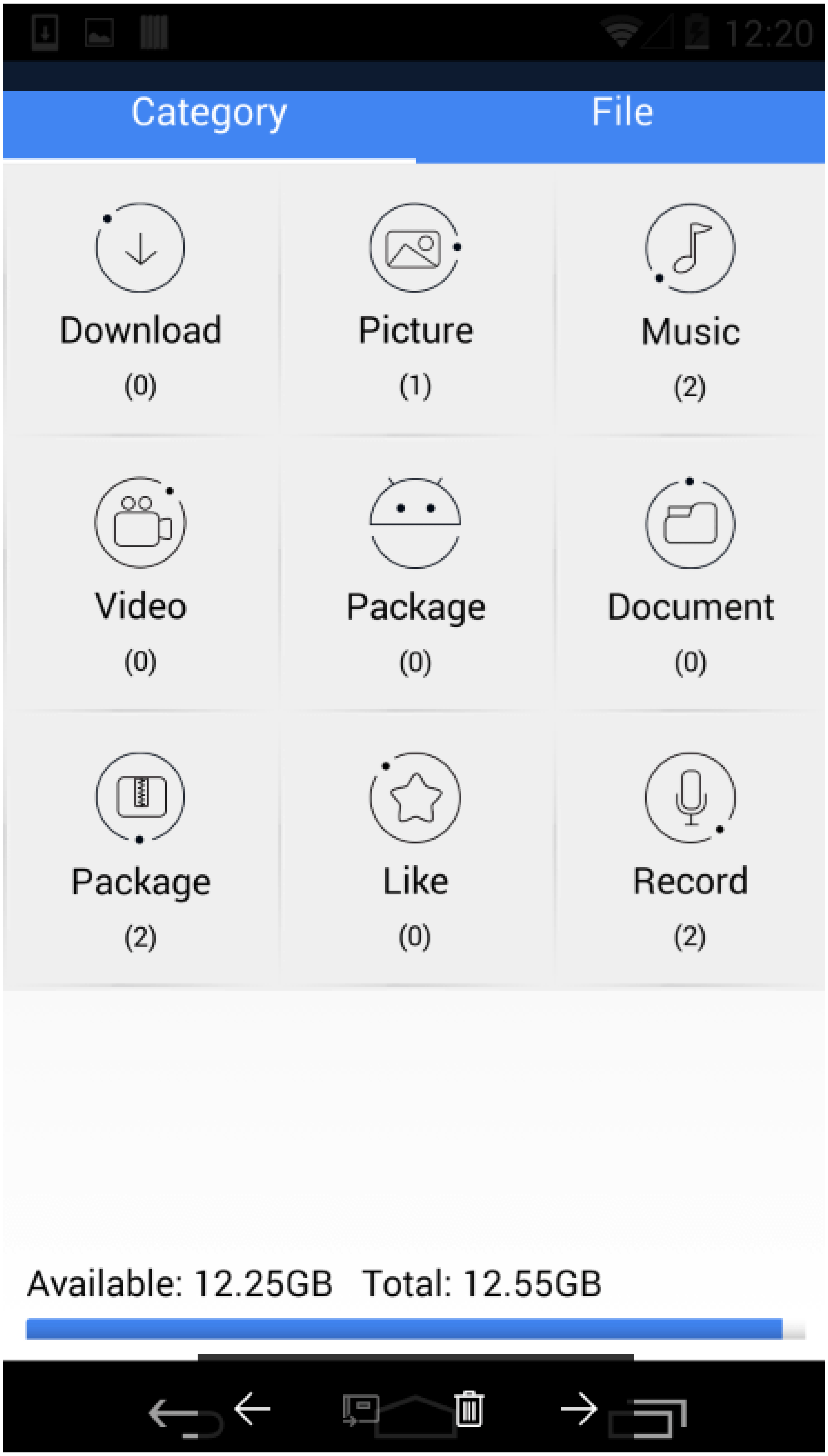


Figure 2. A view of the malware app

We decompiled the APK file, as shown in Figure 3.

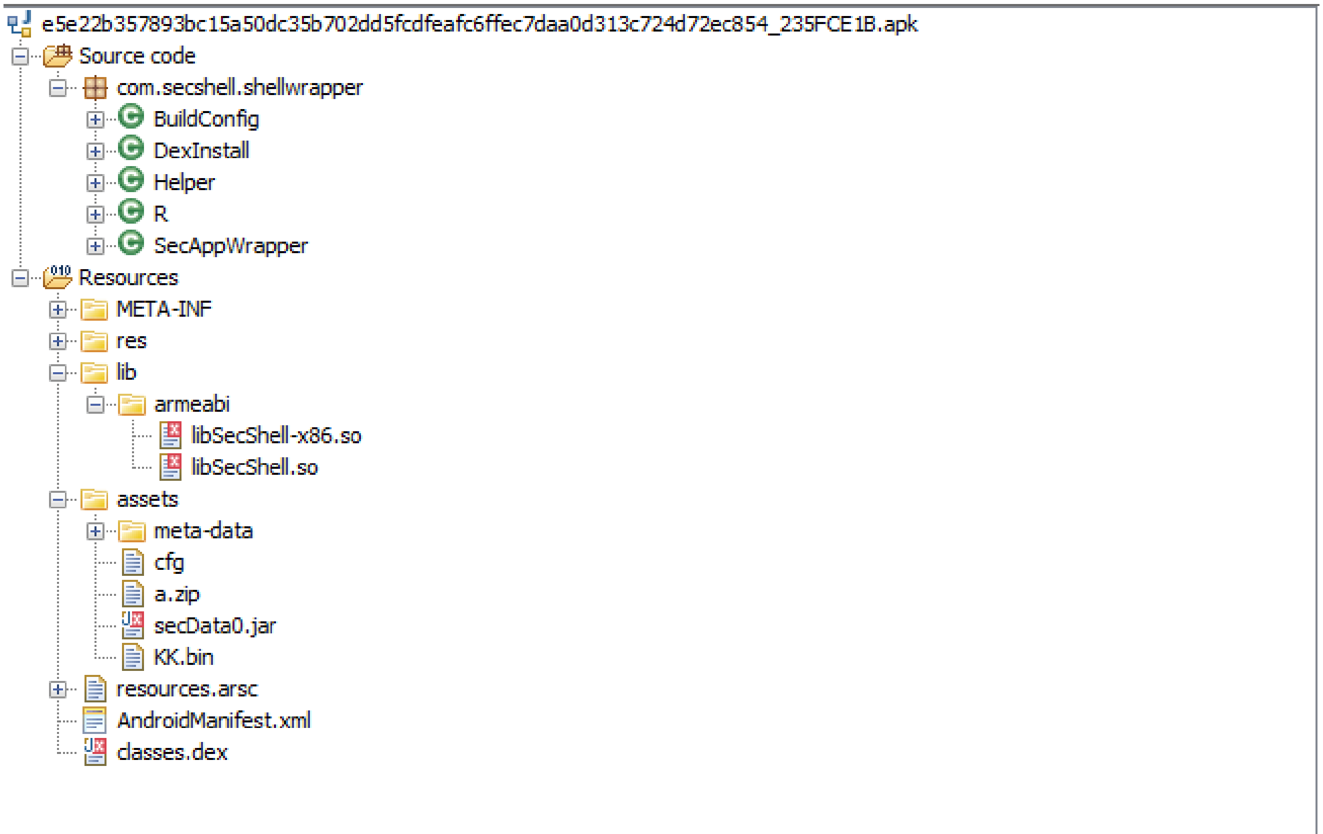


Figure 3. Decompile the malware app's apk file

Its package name is com.web.sdfile. First, let's look at its AndroidManifest.xml file.

Figure 4. AndroidManifest.xml file inside the malware app's apk file

We can't find the main activity com.sd.clip.activity.FileManagerActivity, service class, or broadcast class in Figure 4. Obviously, the main logic of the file helper app is not located in the classes.dex. After analysis, it was discovered that the malware app uses the multidex scheme to dynamically load a secondary dex file and execute it.

How Rootnik Works

I. Workflow of Rootnik

The following is the workflow of the android rootnik malware.

Figure 5. An overview of the android rootnik malware's workflow

II. Going deep into the first dex file

The following is a code snippet of the class SecAppWrapper.

```java
public class SecAppWrapper extends Application {
    public static Application realApplication;

    static {
        SecAppWrapper.realApplication = null;
        System.loadLibrary("SecShell");
        if(Helper.PPATH != null) {
            System.load(Helper.PPATH);
        }
    }

    public SecAppWrapper() {
        super();
    }

    protected void attachBaseContext(Context arg4) {
        super.attachBaseContext(arg4);
        try {
            SecAppWrapper.realApplication = this.getClassLoader().loadClass(Helper.APPNAME).newInstance();  // Helper.APPNAME = "com.sd.clip.base.MyApplication"
            Helper.attach(SecAppWrapper.realApplication, arg4);
        }
        catch(Exception v0) {
            SecAppWrapper.realApplication = null;
        }
    }
  public void onCreate() {
      super.onCreate();
      try {
          this.huawei_share();
      }
      catch(Exception v0) {
      }

      if(SecAppWrapper.realApplication != null) {
          Helper.attach(SecAppWrapper.realApplication, null);
          SecAppWrapper.realApplication.onCreate();
      }
  }
```

Figure 6. A code snippet of the class SecAppWrapper

The execution flow is shown below.

> Static code block -> attachBaseContext -> onCreate

The static code block loads the dynamic link library libSecShell.so into the folder assets, and the program enters into the native layer, performs several anti-debug operations, decrypts the secondary dex file, and then uses a multidex scheme to load the decrypted secondary dex file, which is actually the main logic of the real application.

The class DexInstall is actually the class MultiDex, and it refers to

https://android.googlesource.com/platform/frameworks/multidex/+/d79604bd38c101b54e41745f85ddc2e04d978af2/library/src/android/support/multidex/MultiDex.java

The program then invokes the method install of DexInstall to load the secondary dex file. The invoking of the method install of DexInstall is executed in native layer.

```java
public static void install(ClassLoader arg4, String arg5) {
    try {
        File v0 = new File(arg5);
        ArrayList v2 = new ArrayList();
        ((List)v2).add(v0);
        DexInstall.installSecondaryDexes(arg4, v0.getParentFile(), ((List)v2));
    }
    catch(Exception v1) {
        v1.printStackTrace(System.out);
    }
}
```

Figure 7. Installing the secondary dex file

In function attachBaseContext, the program loads the class com.sd.clip.base.MyApplication, which is the execution entry of the secondary dex. The method attach of Helper is a native method.

In function onCreate, the program executes the method onCreate of the class com.sd.clip.base.MyApplication.

That’s it. The first dex file is rather simple. Next, we’ll do a deep analysis of the native layer code, which is very complicated and tricky.

III. The scope of the native layer code

As described above, the native layer code uses some advanced anti-debug and anti-hook techniques, and also uses several decryption algorithms to decrypt some byte arrays to get the plain text string.

The following is part of the export functions in libSecShell.so. It becomes harder to analyze due to obfuscated function names.

Name	Address	Ordinal
p26102C6A8CA45031EDFB8281477FD563	0000D1CC	
pDA5271CF4CCFFA81F92CF1EC0F435889	0000D890	
p2286289E2C87B5C9202230FFC7E702D4	0000D9C8	
pA4D41BAEAB1450AEEE683A84FD262552	0000DA84	
p642BD1F4A1685C3EDB2B3B920C0EDEA8	0000DDC8	
strlen	0000E624	
JNI_OnLoad	0000E6F4	
p335401FB7DC09C99176C4C11FDE1C9B7	00013028	
p6DF4491A867F48BF1E59730C1F1F97D9	00013034	
p1DE6DB1B7827CE7CFB8D139ED15D7F90	0001305C	
pFA66310A80C2F8BF84C3CCDBBF3C0BA2	000132E4	
pD012AB537A0A2792B1AAF69613FA61A0	00013450	
p40AD7D7CEE164CBEC48795C7820948EA	00013534	
p5AD5D0264D7C8E1AC95496D47885442C	000137BC	
p60AC462B25DA2C75C55F0EC013654EF5	00013BCC	
p4852ABA9A0FB64247021C8D4A4AC24BB	00013D00	
std::allocator<char>::_M_allocate(uint,uint &)	000141A8	
pD3667BED240792A5F1BA435623D9B215	00014E80	
pFB0E7CFB98C1AC8AEDD90B1EAA975993	00015B2C	
pD3E953E17B431824F310DF3381EBDE3E	00015F4C	
pD8F3FA10EEF02923410B2987925759A0	000160EC	
artOatFileOatMethodLinkMethodStub	0001651C	
artClassLinkerDefineClassStub(void *,char const*,void *,void *,void *)	00016608	
pB480AE69EF75206D239B81E62C4C5C10	00016628	
p22E61FD3F3B19CAC04EC7767A8A1756A	00016EE8	
artMOatFileOatMethodLinkMethodStub	00017C54	
p45C8619F918523ED498753806FC08904	00017C68	
p453979B388BECB0D0A8350CC47FCFC13	000185F0	
std::priv::_String_base<char,std::allocator<char> >::_M_deallocate_block(void)	000192F0	
p16DB731B80EE4B088152BBAC874D1494(void *,char const*,char const*,void *,void *)	00019314	
artm_OpenDexFilesFromOat_stub(void *,char const*,char const*,void *)	0001939C	
pB5516CAC797AEBE879DDB9A474472558	0001AE54	
fork_execute_dex2oat	0001B488	
fork_execute_dex2opt	0001B554	
p6BECCA499822B6083186BD481EAF40B3	0001B5D8	
pC0F901BB7A6D1B669B72D78F6861439F	0001B710	

Figure 8. Part of export functions in libSecShell.so

All anti-debug native code is located in function JNI_Onload.

As described in the last section, the method attach of class Helper in java scope is a native method. The program dynamically registers this method in native layer. The following is a snippet of the ARM assembly code that registers native method in native layer.

```
:0000F6F8    STRB    R6, [R4,#0x1C]
:0000F6FA    BL      sub_BF60 ; it's a decryption of char array. com/secshell/shellwrapper/Helper
:0000F6FE    MOVS    R2, #0xD7
:0000F700    LDR     R3, [SP,#0x1C]
:0000F702    LSLS    R2, R2, #2
:0000F704    MOVS    R1, R4
:0000F706    LDR     R3, [R3]
:0000F708    LDR     R0, [SP,#0x1C]
:0000F70A    LDR     R6, [R3,R2]
:0000F70C    LDR     R3, [R3,#0x18]
:0000F70E    BLX     R3      ; findClass(JNIEnv *env, const char *name)
:0000F710    MOVS    R2, R5
:0000F712    MOVS    R1, R0
:0000F714    ADDS    R2, #0x14
:0000F716    MOVS    R3, #1
:0000F718    LDR     R0, [SP,#0x1C]
:0000F71A    BLX     R6      ; jint RegisterNatives(JNIEnv *env,jclass clazz, const JNINativeMethod* methods,jint nMethods)
:0000F71C    LDR     R3, [SP,#0x20]
:0000F71E    LDR     R2, =0xFFFFFD84
:0000F720    LDR     R1, [SP,#0x14]
:0000F722    LDRB    R3, [R3,#2]
:0000F724    LDR     R4, [R1,R2]
:0000F726    CMP     R3, #0
:0000F728    BNE     loc_F72C
:0000F72A    STR     R3, [R4]
```

Figure 9. Dynamically register native method in native layer

The function RegisterNatives is used to register a native method. Its prototype is shown below.

> *jint RegisterNatives(JNIEnv *env,jclass clazz, const JNINativeMethod* methods,jint nMethods)*

The definition of JNINativeMethod is shown below.

> *typedef struct {*
> *const char* name;*
> *const char* signature;*
> *void* fnPtr;*
> *} JNINativeMethod;*
>
> The first variable name is the method name in Java. Here, it's the string "attach". The third variable, fnPtr, is a function pointer that points to a function in C code.

We next need to find the location of the anti-debug code and bypass it, analyze how the secondary dex file is decrypted, and the dump the decrypted secondary dex file from memory.

Let's look at the following code in IDA:

```
.text:0000F81E                 BLX             sprintf
.text:0000F822                 MOVS            R0, R5
.text:0000F824                 BL              pC0E901BB7A6D1B669B72D78E6861439F
.text:0000F828                 SUBS            R1, R0, #0
.text:0000F82A                 BNE             loc_F834
.text:0000F82C                 LDR             R0, [SP,#0x1C]
.text:0000F82E                 BL              sub_D334 ; trace
.text:0000F832                 B               loc_F924 ; after step some code, you can found the anti-debug code.
.text:0000F834 ; ---------------------------------------------------------------------------
.text:0000F834
.text:0000F834 loc_F834                                ; CODE XREF: .text:0000F82A↑j
.text:0000F834                 MOVS            R4, #0
.text:0000F836                 LDR             R3, [R6]
.text:0000F838                 STR             R4, [SP,#0x90]
.text:0000F83A                 CMP             R3, R4
.text:0000F83C                 BEQ             loc_F8D8 ; continue...
.text:0000F83E                 BL              p45C8619F918523ED498753806FC08904
.text:0000F842                 B               loc_F908
```

Figure 10. Code snippet around anti-debug code

Based on our deep analysis, the instruction at address 0xF832 is a jump to address loc_F924.

After tracing some code, we found the anti-debug code.

```
.text:0000F924 loc_F924                                ; CODE XREF: .text:0000F51A↑j
.text:0000F924                                         ; .text:0000F832↑j
.text:0000F924                 LDR             R3, [SP,#0x20] ; from 0xf832 in dynamic debugging,R3 points p599E9330AD7F8A212DE1663B683F8BF4 |00 00 7E 49 5
.text:0000F926                 LDRB            R3, [R3,#8]
.text:0000F928                 CMP             R3, #0
.text:0000F92A                 BEQ             loc_F930
.text:0000F92C                 BL              loc_103BA ; jump

.text:000103BA
.text:000103BA loc_103BA                               ; CODE XREF: .text:0000F92C↑j
.text:000103BA                                         ; .text:0000FE10↑j ...
.text:000103BA                 LDR             R1, [SP,#0x28]
.text:000103BC                 LDR             R2, [SP,#0x3C]
.text:000103BE                 LDR             R0, [SP,#0x2C]
.text:000103C0                 BL              pFBF8EA28AB5406DC5CFADBC7CE32467F
.text:000103C4                 LDR             R0, [SP,#0x28]
.text:000103C6                 BL              p071ADBC73D8008F1BE158FD0441DC741
.text:000103CA                 LDR             R3, [SP,#0x20]
.text:000103CC                 LDRB            R0, [R3,#0xC]
.text:000103CE                 BL              p7E7056598F77DFCC42AE68DF7F0151CA ; F8 on this instruction, the debugging processing in IDA exits.Anti-debuggin code...
.text:000103D2
.text:000103D2 loc_103D2                               ; CODE XREF: .text:0000E742↑j
.text:000103D2                                         ; DATA XREF: .text:0000E778↑o
.text:000103D2                 BL              loc_10E22
```

Figure 11. The location of the anti-debug code

The function p7E7056598F77DFCC42AE68DF7F0151CA() performs the anti-debug operations.

The following is its graphic execution flow, which is very complicated.

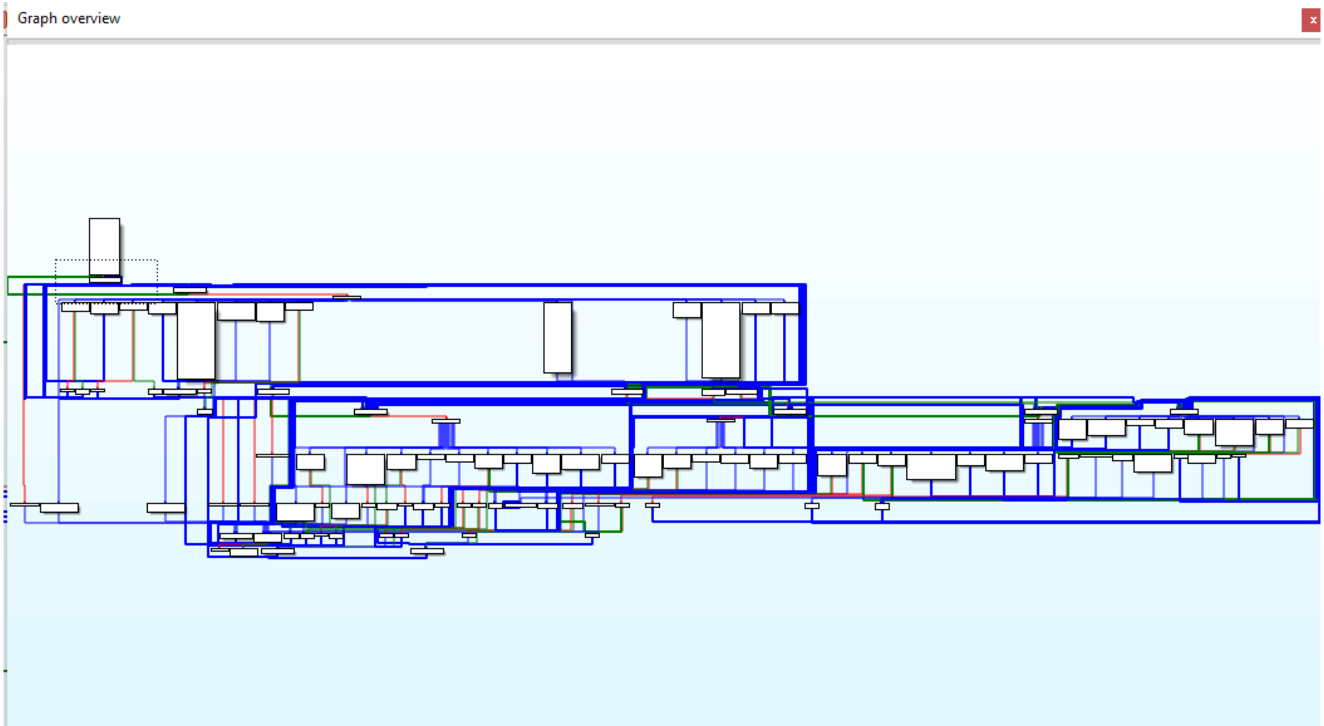


Figure 12. The graphic execution flow of anti-debug code

The following are some methods of anti-debug and anti-hook used in the malware.

1. Detect some popular hook frameworks, such as Xposed, substrate, adbi, ddi, dexposed. Once found, hook it using these popular hook frameworks. It then kills the related process.

```
int __fastcall is_xposed_att(int a1)
{
  int v1; // r4@1
  int result; // r0@2
  unsigned int v3; // r0@4

  v1 = a1;
  if ( strcasestr(a1, "xposedbridge") || strcasestr(v1, ".xposed.") )
  {
    result = 1;
  }
  else
  {
    v3 = strcasestr(v1, "xposed_art");
    result = v3 - (v3 - 1 + (v3 < 1));
  }
  return result;
}
```

Figure 13. Detection of Xposed hook framework

```
    sub_2362C((int)&s);
    sub_235FC(&s);
    if ( v4 == 120 && s && so_filter((int)&s, (const char *)&v7) && find_hook_feature((int)&s) == 1 )
      pA1C2F587169935B2DA9F1DEC35C8270D(v1, 9);
  }
```

```
signed int __fastcall find_hook_feature(int a1)
{
  int v1; // r5@1
  int v2; // r4@1
  signed int result; // r0@1
  int v4; // r5@3
  const char **v5; // r6@3
  int i; // r4@4
  int v7; // r5@14
  int v8; // [sp+4h] [bp-24h]@3
  void *ptr; // [sp+8h] [bp-20h]@2
  int v10; // [sp+Ch] [bp-1Ch]@2

  v1 = a1;
  v2 = strcasestr(a1, "substrate");
  result = 1;
  if ( !v2 )
  {
    ptr = 0;
    v10 = 0;
    if ( !read_elf_file(v1, &ptr, &v10) )
    {
      v4 = 0;
      v5 = (const char **)ptr;
      v8 = v10;
      while ( 1 )
      {
        for ( i = 0; ; ++i )
        {
          if ( i >= v8 )
          {
            v2 = 0;
            goto LABEL_10;
          }
          if ( !strcmp(off_41260[v4], v5[3 * i]) )
            break;
```

```
.data.rel.ro.local:00041260 off_41260       DCD aMshookfunction     ; DATA XREF: find_hook_feature+32↑o
.data.rel.ro.local:00041260                                         ; .text:off_23E08↑o
.data.rel.ro.local:00041260                                         ; "MSHookFunction"
.data.rel.ro.local:00041264                 DCD aSubstrate          ; "substrate"
.data.rel.ro.local:00041268                 DCD aMsfindsymbol       ; "MSFindSymbol"
.data.rel.ro.local:0004126C                 DCD aSubstrate          ; "substrate"
.data.rel.ro.local:00041270                 DCD aMsclosefunctio     ; "MSCloseFunction"
.data.rel.ro.local:00041274                 DCD aSubstrate          ; "substrate"
.data.rel.ro.local:00041278                 DCD aHook_postcall      ; "hook_postcall"
.data.rel.ro.local:0004127C                 DCD aAdbi_hook          ; "adbi_hook"
.data.rel.ro.local:00041280                 DCD aHook_precall       ; "hook_precall"
.data.rel.ro.local:00041284                 DCD aAdbi_hook          ; "adbi_hook"
.data.rel.ro.local:00041288                 DCD aDalvik_java_me     ; "dalvik_java_method_hook"
.data.rel.ro.local:0004128C                 DCD aAllinones_arth     ; "ALLINONEs_arthook"
.data.rel.ro.local:00041290                 DCD aArt_java_metho     ; "art_java_method_hook"
.data.rel.ro.local:00041294                 DCD aAllinones_arth     ; "ALLINONEs_arthook"
.data.rel.ro.local:00041298                 DCD aArt_quick_call     ; "art_quick_call_entrypoint"
.data.rel.ro.local:0004129C                 DCD aAllinones_arth     ; "ALLINONEs_arthook"
.data.rel.ro.local:000412A0                 DCD aArtquicktodisp     ; "artQuickToDispatcher"
.data.rel.ro.local:000412A4                 DCD aAllinones_arth     ; "ALLINONEs_arthook"
.data.rel.ro.local:000412A8                 DCD aDexstuff_defin     ; "dexstuff_defineclass"
.data.rel.ro.local:000412AC                 DCD aDdi_hook           ; "ddi_hook"
.data.rel.ro.local:000412B0                 DCD aDexstuff_loadd     ; "dexstuff_loaddex"
.data.rel.ro.local:000412B4                 DCD aDdi_hook           ; "ddi_hook"
.data.rel.ro.local:000412B8                 DCD aDexstuff_resol     ; "dexstuff_resolv_dvm"
.data.rel.ro.local:000412BC                 DCD aDdi_hook           ; "ddi_hook"
.data.rel.ro.local:000412C0                 DCD aDexposedbridge     ; "DexposedBridge"
.data.rel.ro.local:000412C4                 DCD aDexposed           ; "dexposed"
.data.rel.ro.local:000412C8                 DCD aDexposedishook     ; "dexposedIsHooked"
.data.rel.ro.local:000412CC                 DCD aDexposed           ; "dexposed"
.data.rel.ro.local:000412D0                 DCD aDexposedcallha     ; "dexposedCallHandler"
.data.rel.ro.local:000412D4                 DCD aDexposed           ; "dexposed"
.data.rel.ro.local:000412D4 ; .data.rel.ro.local ends
```

Figure 14. Finding the hook feature

2. It then uses an kind of multi-process ptrace to implement anti-debug, which is tricky a little. Here we don't plan to provide a detailed analysis of the anti-debugging implementation mechanism, but only give some simple explanations.

We can see that there are two processes named com.web.sdfile.

```
root@hammerhead:/ # ps | grep com.web.sdfile
u0_a126   28773 181   942644 58276 ffffffff 4011e73c S com.web.sdfile
u0_a126   28799 28773 879220 33308 ffffffff 4011cdf0 S com.web.sdfile
u0_a126   28832 181   881620 42364 ffffffff 4011e73c S com.web.sdfile:dys
u0_a126   28848 28832 878732 32796 ffffffff 4011cdf0 S com.web.sdfile:dys
```

Figure 15. Two processes named com.web.sdfile

The following is a snippet of multi-process anti-debug code.

```
int __fastcall anti_thread_of_process_debug(int a1, pthread_t a2)
{
  int v2; // r4@1
  signed int v3; // r5@1
  void *v4; // r6@1
  int result; // r0@2
  pthread_t newthread; // [sp+4h] [bp-14h]@1

  newthread = a2;
  v2 = a1;
  v3 = 31;
  v4 = malloc(4u);
  *(_DWORD *)v4 = v2;
  while ( 1 )
  {
    result = pthread_create(&newthread, 0, (void *(*)(void *))anti_thread_body, v4);
    if ( !result )
      break;
    if ( !--v3 )
      break;
    sleep(1u);
  }
  return result;
}
```

Figure 16. A snippet of anti-debug code

3. The program also uses inotify to monitor the memory and pagemap of the main process. It causes the memory dumping to be incomplete. The two processes use pipe to communicate with each other.

In short, these anti-debug and anti-hook methods create a big obstacle for reversing engineering. So bypassing these anti- methods is our first task.

Let's try to bypass them.

As described in Figure 10, the instruction at offset 0x0000F832 jumps to loc_F924, and then the program executes these anti-debug codes. We can dynamically modify the values of some registers or some ARM instructions to change the execution flow of the program when dynamically debugging. When the program executes the instruction "SUBS R1, R0, #0" at offset 0xF828, we modify the value of register R0 to a non-zero value, which makes the condition of the instruction "BNE loc_F834" become true. This allows the program to jump to loc_F834.

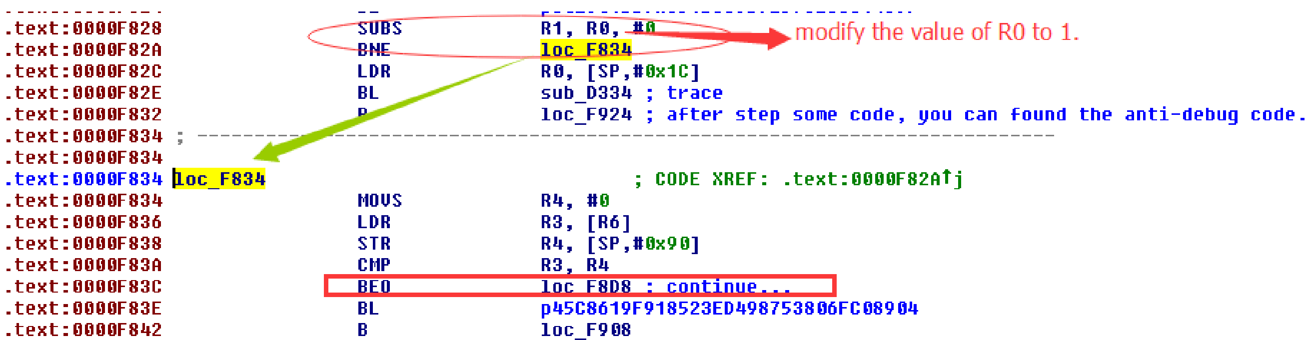


Figure 17. How to bypass the anti-debug code

Next, we dynamically debug it, bypass the anti-debug, and then dump the decrypted secondary dex file. The dynamic debugging is shown below.

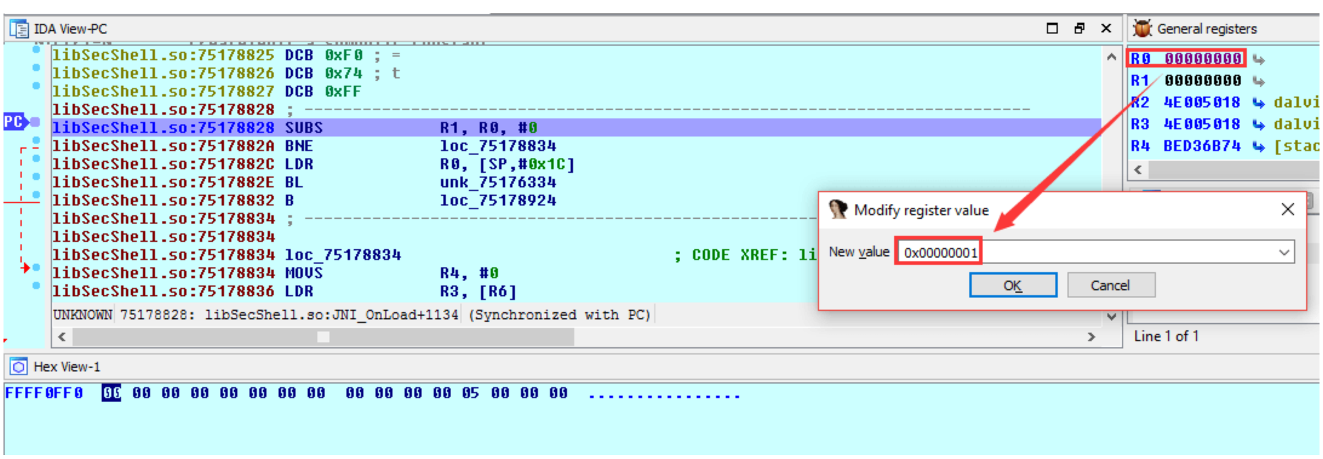


Figure 18. Modifying the value of R0 to non-zero

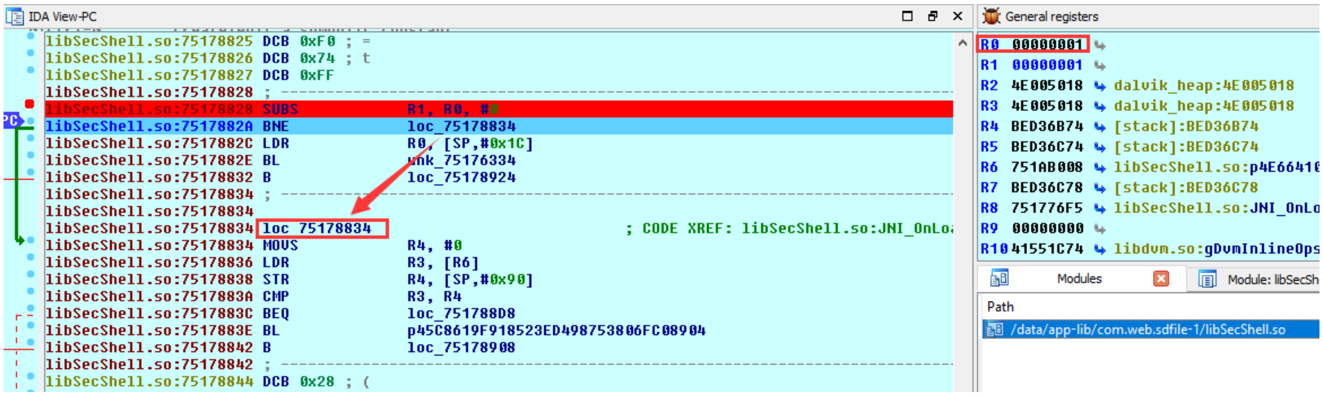


Figure 19. Jump to local_75178834

Next, jump to local_751788D8, as follows.

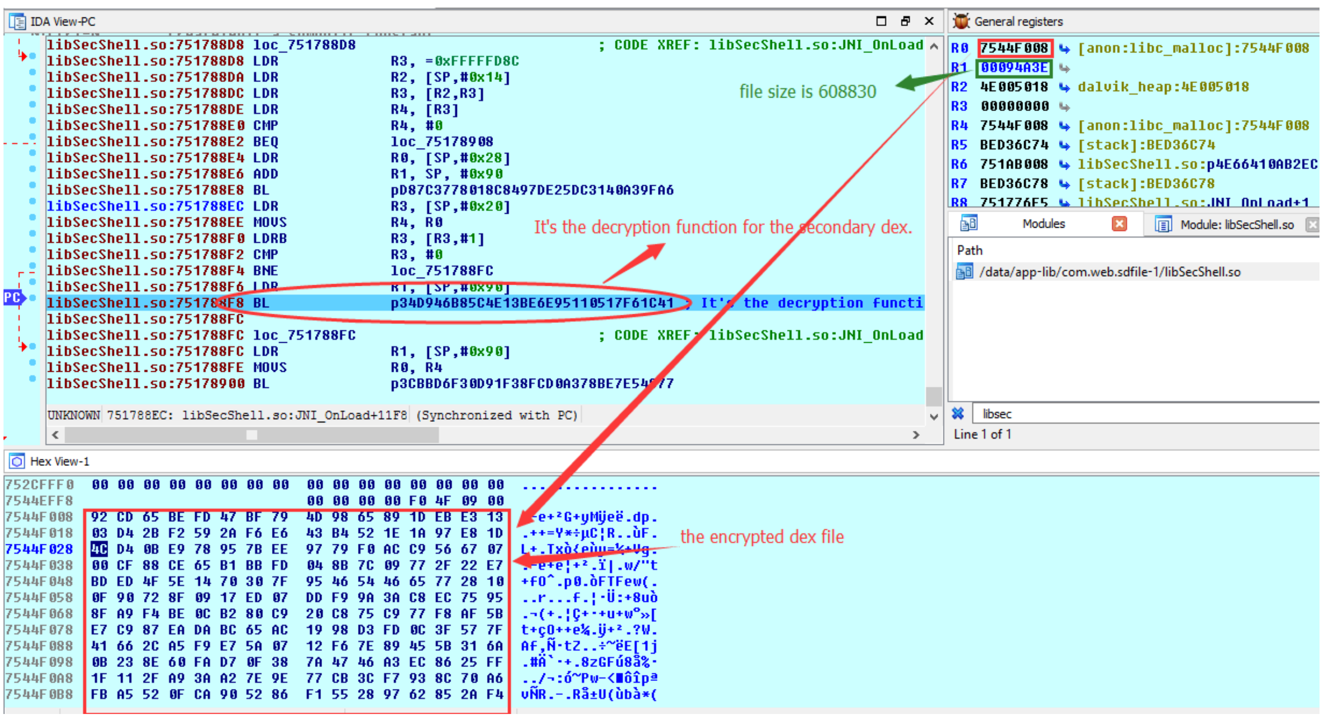


Figure 20. Decryption function for the secondary dex

The function p34D946B85C4E13BE6E95110517F61C41 is the decryption function. The register R0 points to the memory storing the encrypted dex file, and the value of R1 is the size of file and is equal to 0x94A3E(608830). The encrypted dex file is secData0.jar in the folder assets in the apk package. The following is the file secData0.jar.

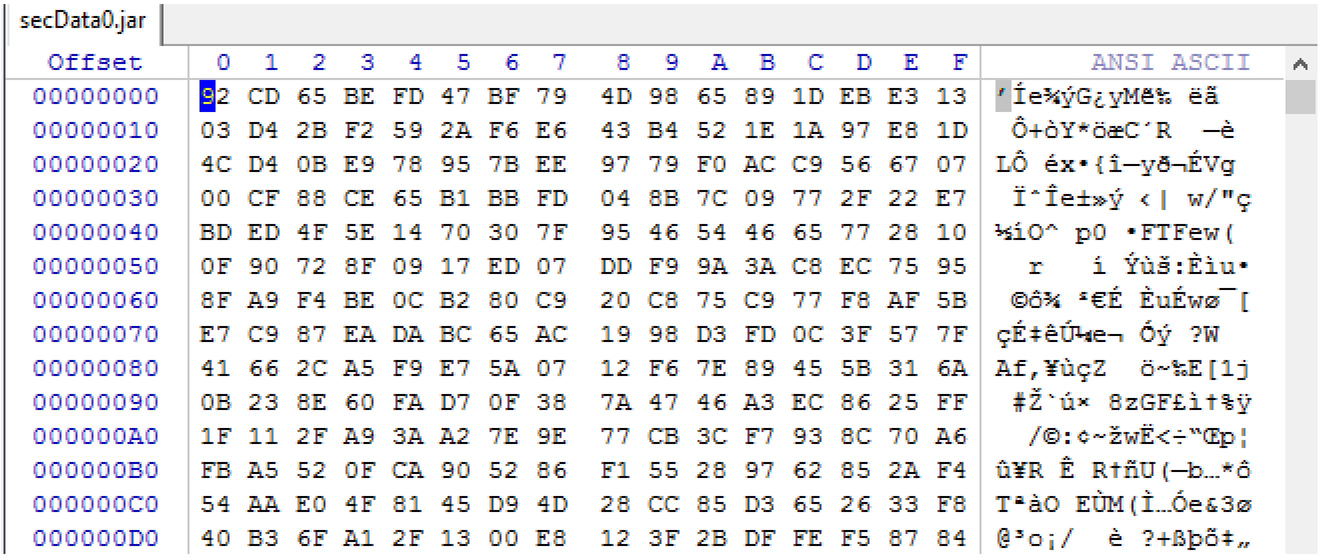


Figure 21. The file secData0.jar in the folder assets in the apk package

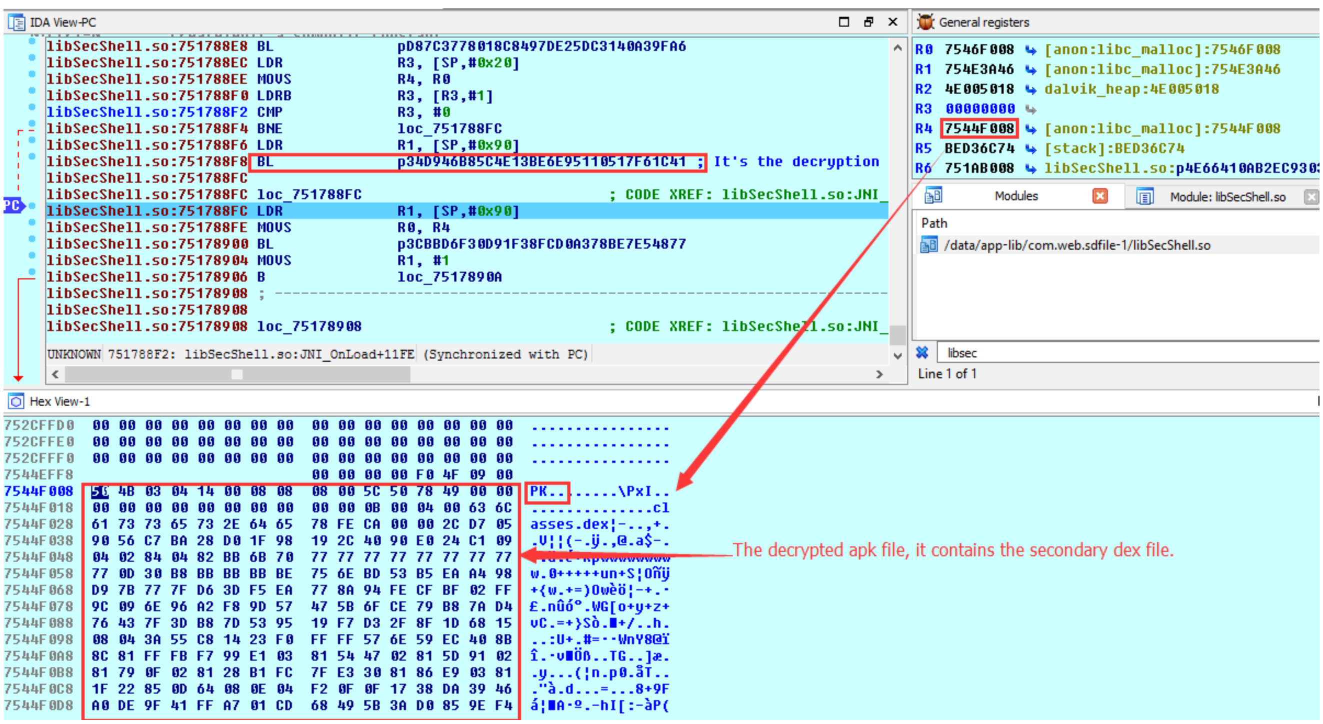


Figure 22. The content of the decrypted secondary apk file in memory

We can now dump the memory of the decrypted file to the file decrypt.dump.

The decrypted file is a zip format file, and it contains the secondary dex file. After decryption, the program decompresses the decrypted secondary apk to a dex file. The function p3CBBD6F30D91F38FCD0A378BE7E54877 is used to decompress the file.

Next, the function unk_75176334 invokes the java method install of class com.secshell.shellwrapper.DexInstall to load the secondary dex.

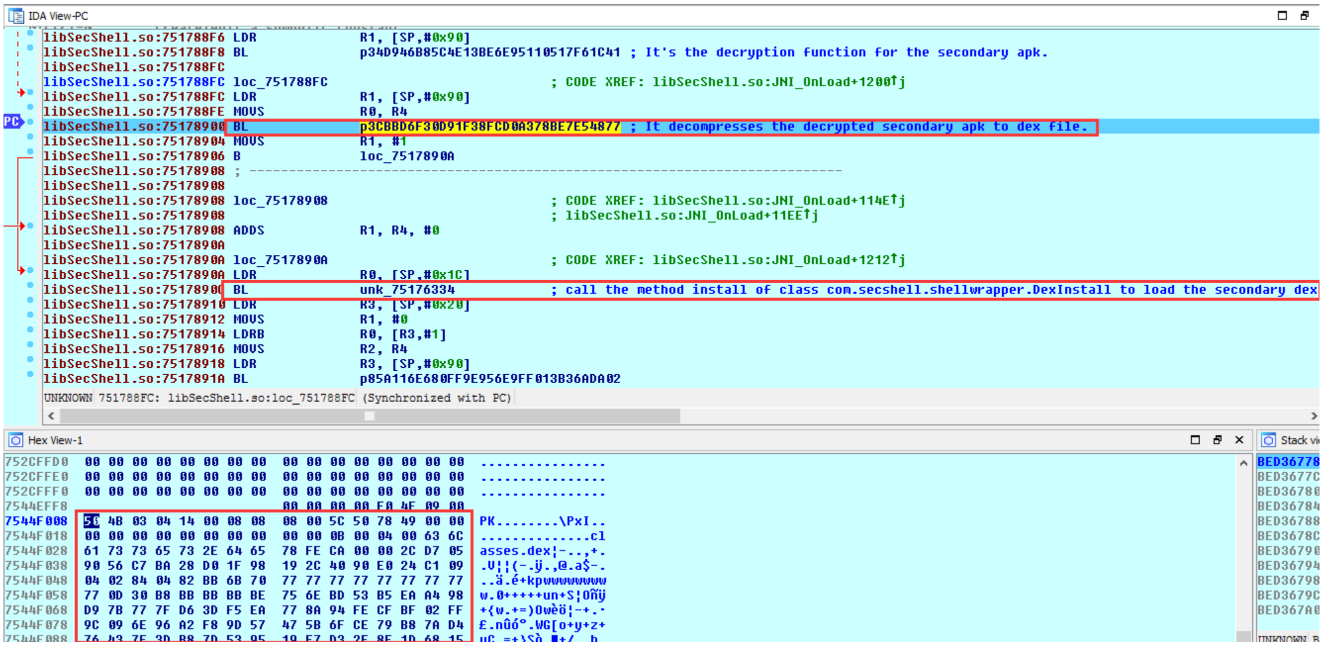


Figure 23. Decompressing the decrypted secondary apk and loading the secondary dex file

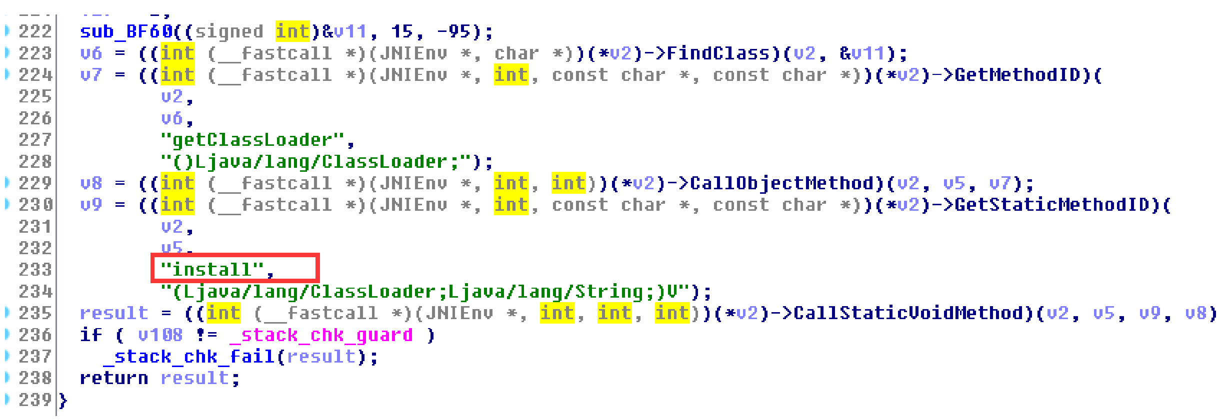


Figure 24. Calling the method install via Jni

Here we finish the analysis of native layer and get the decrypted the secondary apk file, then will analyze the apk file in the part II of this blog.

The Decryption Function That Decrypts secData0.jar in Native Layer:

Related Posts