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

By Kai Lu

Published: 2017-01-26 · Archived: 2026-04-05 20:14:17 UTC

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.

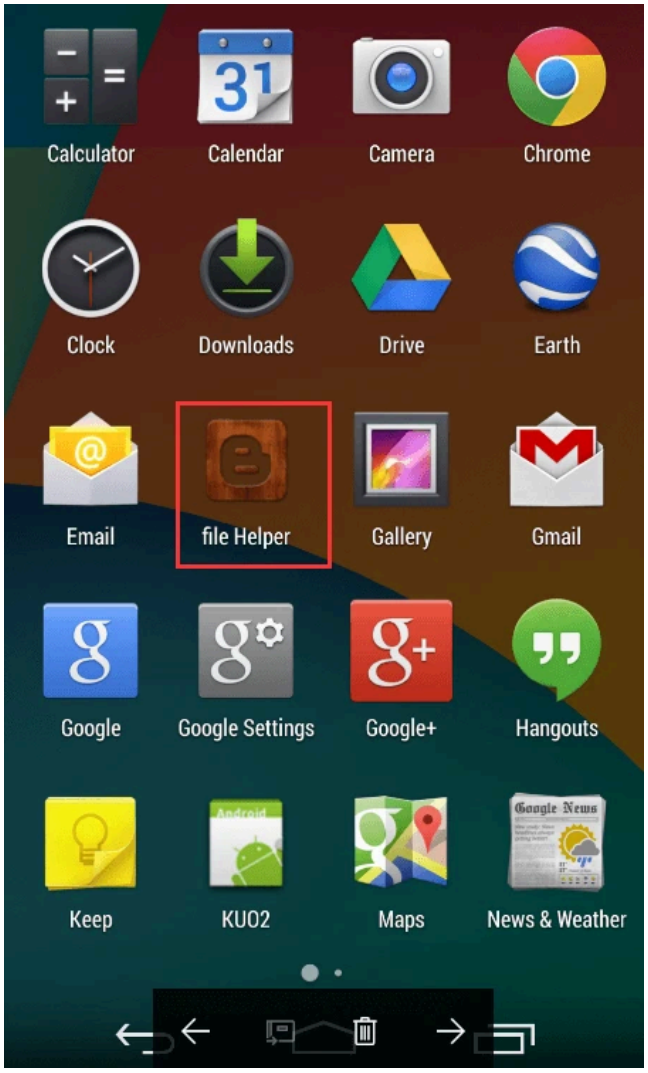

Figure 1. The malware app icon installed

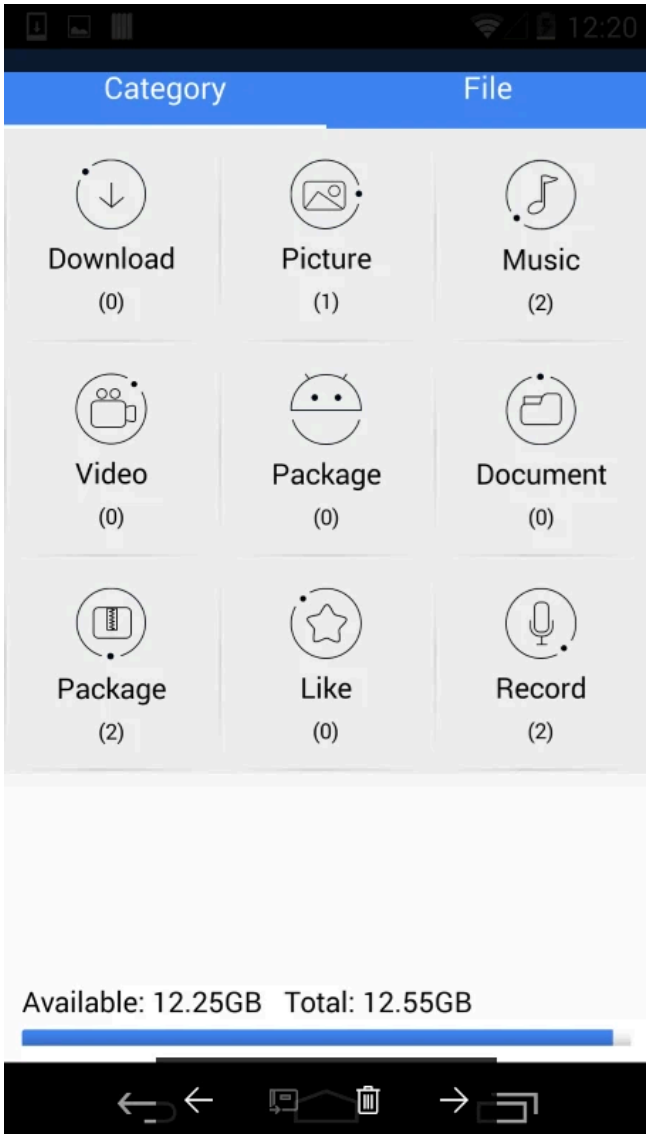

Figure 2. A view of the malware app

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

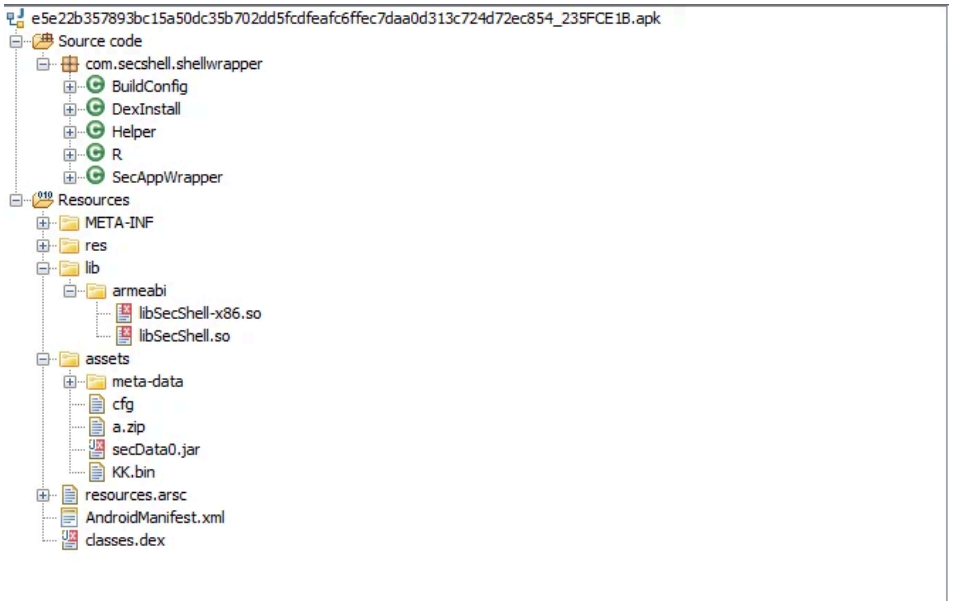

Figure 3. Decompile the malware app's apk file

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

```
<application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/app_icon" android:label="@string/app_name" android:name="com.secshell.shell.ShellApplication" and
    <activity android:exported="true" android:label="@string/app_name" android:launchMode="singleTask" android:name="com.sd.clip.activity.SDManagerActivity" android:screenOrientation="portrai
    <activity android:name="com.sd.clip.activity.FileManagerActivity" android:screenOrientation="portrait" android:theme="@android:style/Theme.Light.NoTitleBar">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name="com.sd.clip.activity.FileDeleteActivity" android:screenOrientation="portrait" android:theme="@android:style/Theme.Light.NoTitleBar" />
    <service android:exported="false" android:name="com.sd.clip.ur.UploadErrorInfoService" />
    <service android:name="com.hg.mer.PG" android:process=":dys" />
    <receiver android:name="com.hg.mer.Nws">
        <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.USER_PRESENT" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
        </intent-filter>
    </receiver>
    <meta-data android:name="adchannel" android:value="jgm161115102" />
    <meta-data android:name="UMENG_CHANNEL" android:value="jgm161115102" />
    <meta-data android:name="UMENG_APPKEY" android:value="582aa12c7666134ece0006c9" />
    <receiver name="com.bangcle.everisk.stub.AlarmReceiver" />
    <activity background="\#ffffff" label="EVERISK" name="com.bangcle.everisk.stub.NewActivity" />
</application>
```

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.

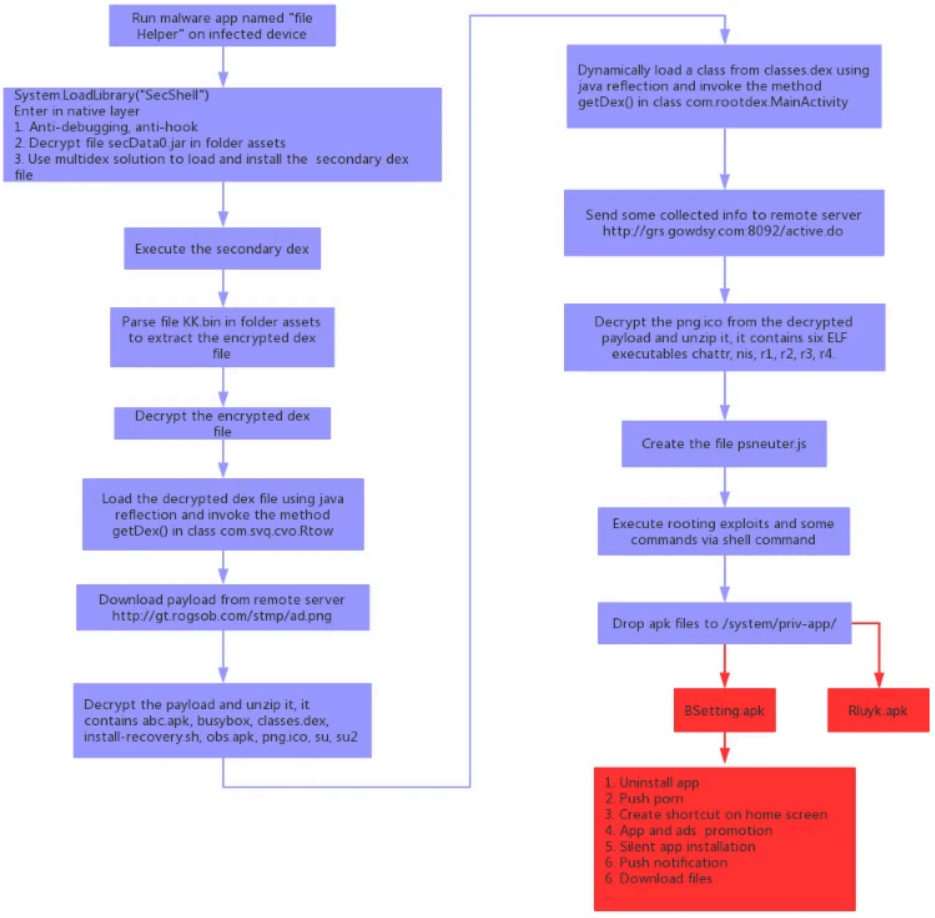

Figure 5. An overview of the android rootnik malware's workflow

II. Going deep into the first dex file

The following is a code snippet of the class SecAppWrapper.

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/mul

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.

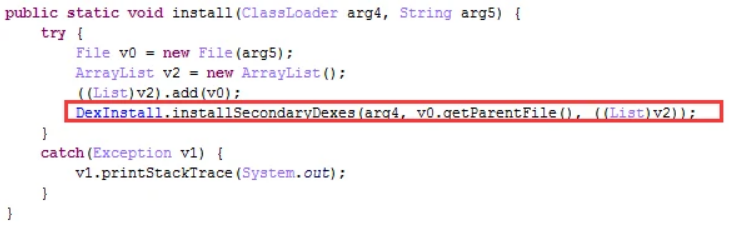

Figure 7. Installing the secondary dex file

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.

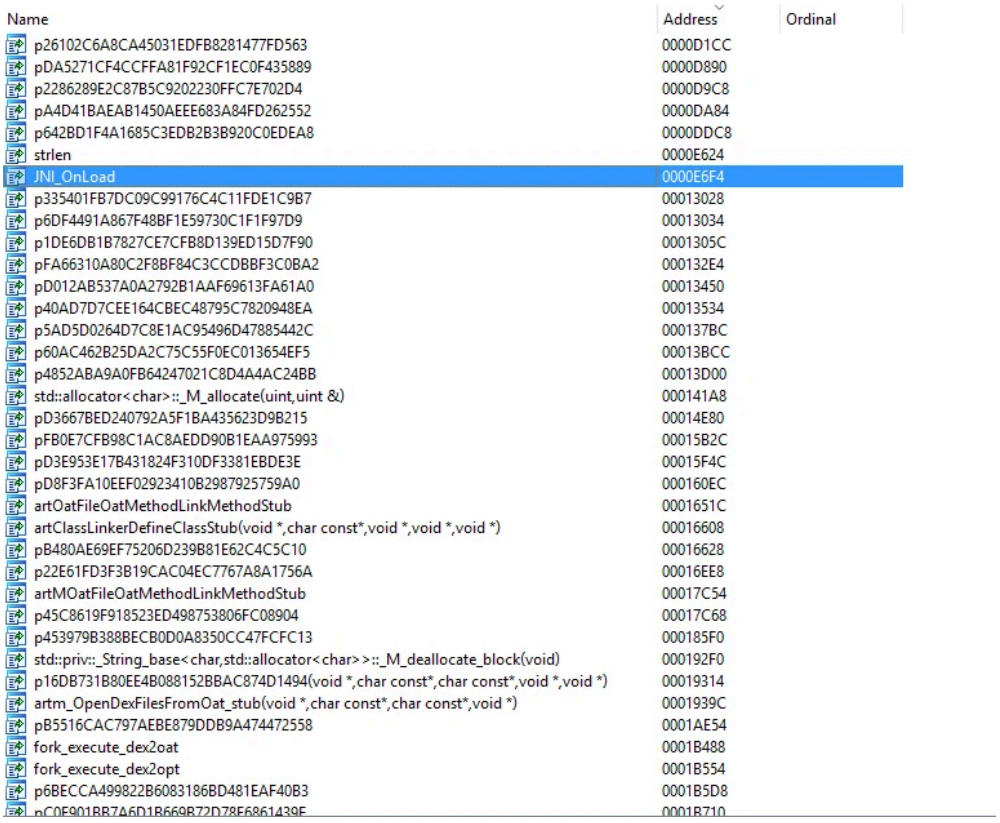

Figure 8. Part of export functions in libSecShell.so

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              pF8F8EA28AB5406DC5CFAD8C7CE32467F
.text:000103C4                  LDR             R0, [SP,#0x28]
.text:000103C6                  BL              p071ADBC73D8008F18E158FD0441DC741
.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.

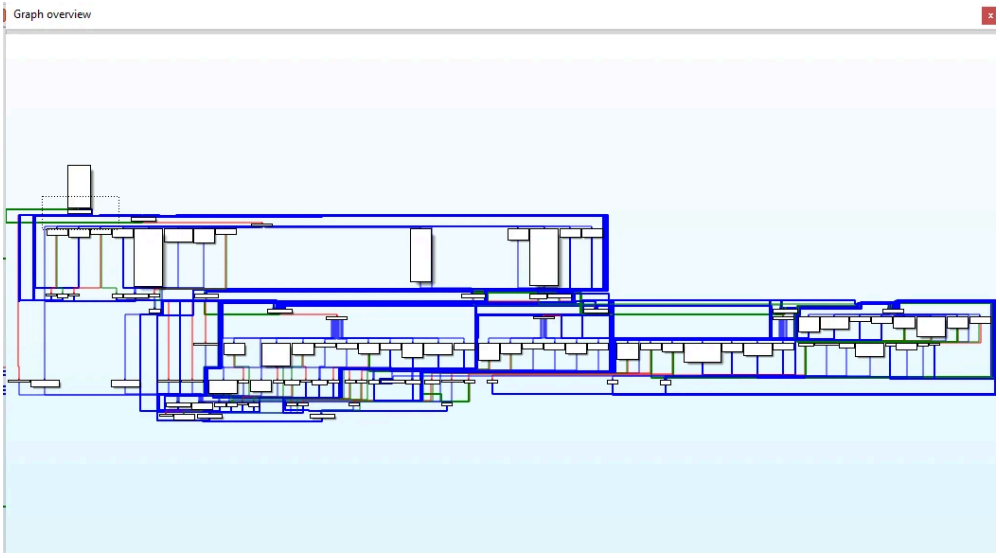

Figure 12. The graphic execution flow of anti-debug code

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 ( tread_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:000412600 off_41260   DCD aMshookfunction   ; DATA XREF: find_hook_feature+32To
.data.rel.ro.local:000412600                                   ; .text:off_23E0BTo
.data.rel.ro.local:000412600                                   ; "MSHookFunction"
.data.rel.ro.local:000412604             DCD aSubstrate        ; "substrate"
.data.rel.ro.local:000412608             DCD aMsfindsymbol     ; "MSFindSymbol"
.data.rel.ro.local:00041260C             DCD aSubstrate        ; "substrate"
.data.rel.ro.local:000412610             DCD aMscloseFunctio   ; "MSCloseFunction"
.data.rel.ro.local:000412614             DCD aSubstrate        ; "substrate"
.data.rel.ro.local:000412618             DCD aHook_postcall    ; "hook_postcall"
.data.rel.ro.local:00041261C             DCD aAdbi_hook        ; "adbi_hook"
.data.rel.ro.local:000412620             DCD aHook_precall     ; "hook_precall"
.data.rel.ro.local:000412624             DCD aAdbi_hook        ; "adbi_hook"
.data.rel.ro.local:000412628             DCD aDalvik_java_me   ; "dalvik_java_method_hook"
.data.rel.ro.local:00041262C             DCD aAllinones_arth   ; "ALLINONEs_arthook"
.data.rel.ro.local:000412630             DCD aArt_java_metho   ; "art_java_method_hook"
.data.rel.ro.local:000412634             DCD aAllinones_arth   ; "ALLINONEs_arthook"
.data.rel.ro.local:000412638             DCD aArt_quick_call   ; "art_quick_call_entrypoint"
.data.rel.ro.local:00041263C             DCD aAllinones_arth   ; "ALLINONEs_arthook"
.data.rel.ro.local:000412640             DCD aArtquicktodisp   ; "artQuickToDispatcher"
.data.rel.ro.local:000412644             DCD aAllinones_arth   ; "ALLINONEs_arthook"
.data.rel.ro.local:000412648             DCD aDexstuff_defin   ; "dexstuff_defineclass"
.data.rel.ro.local:00041264C             DCD aDdi_hook         ; "ddi_hook"
.data.rel.ro.local:000412650             DCD aDexstuff_loadd   ; "dexstuff_loaddex"
.data.rel.ro.local:000412654             DCD aDdi_hook         ; "ddi_hook"
.data.rel.ro.local:000412658             DCD aDexstuff_resol   ; "dexstuff_resolv_dvm"
.data.rel.ro.local:00041265C             DCD aDdi_hook         ; "ddi_hook"
.data.rel.ro.local:000412660             DCD aDexposedbridge   ; "DexposedBridge"
.data.rel.ro.local:000412664             DCD aDexposed         ; "dexposed"
.data.rel.ro.local:000412668             DCD aDexposedishook   ; "dexposedIsHooked"
.data.rel.ro.local:00041266C             DCD aDexposed         ; "dexposed"
.data.rel.ro.local:000412670             DCD aDexposedcallha   ; "dexposedCallHandler"
.data.rel.ro.local:000412674             DCD aDexposed         ; "dexposed"
.data.rel.ro.local:000412678 ; .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.

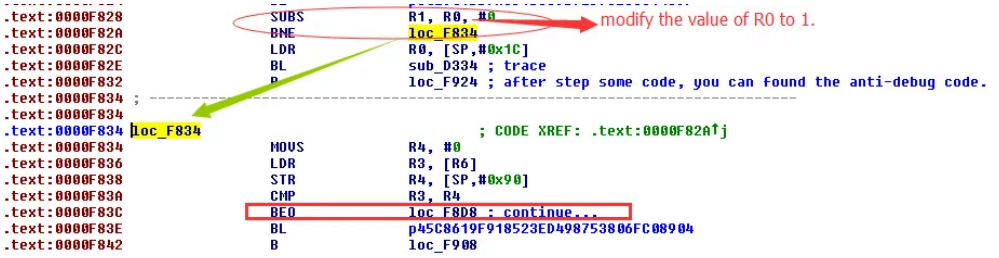

Figure 17. How to bypass the anti-debug code

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

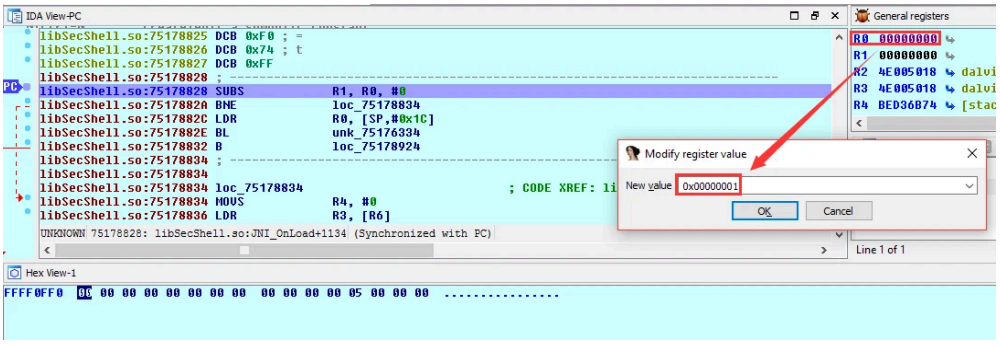

Figure 18. Modifying the value of R0 to non-zero

Figure 19. Jump to local_75178834

Next, jump to local_751788D8, as follows.

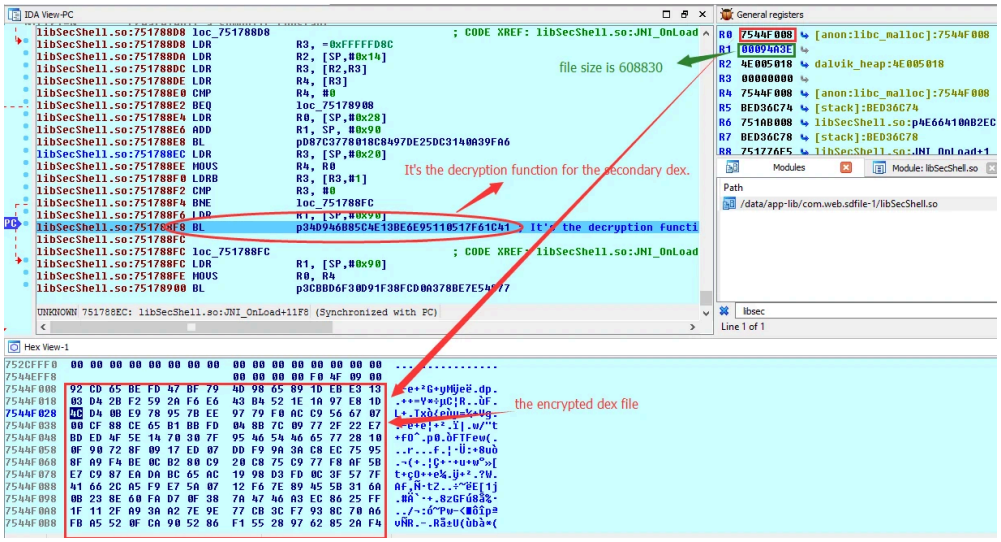

Figure 20. Decryption function for the secondary dex

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.

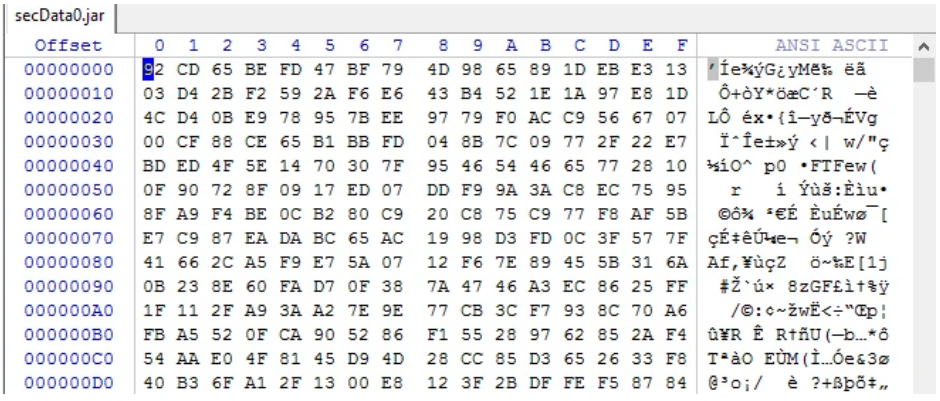

Figure 21. The file secData0.jar in the folder assets in the apk package

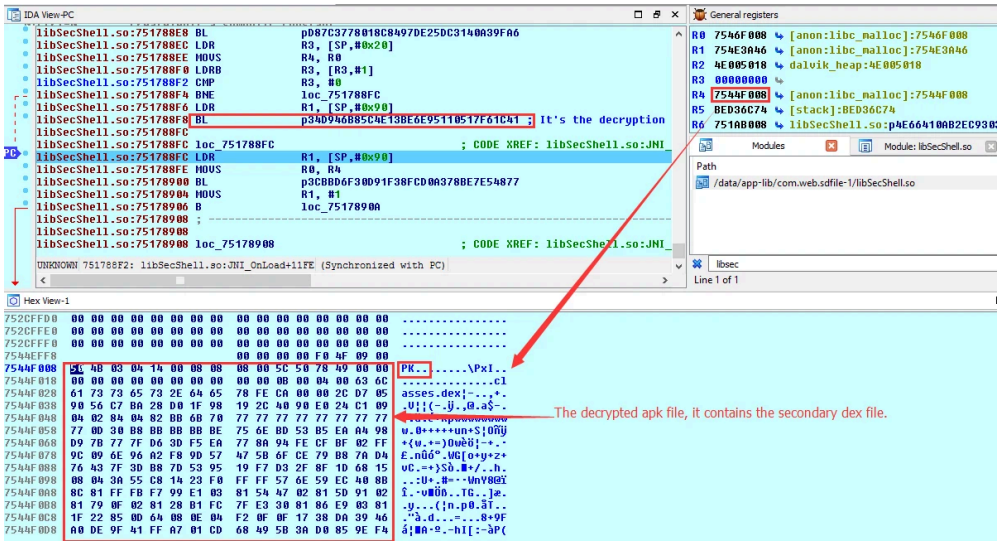

Figure 22. The content of the decrypted secondary apk file in memory

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.

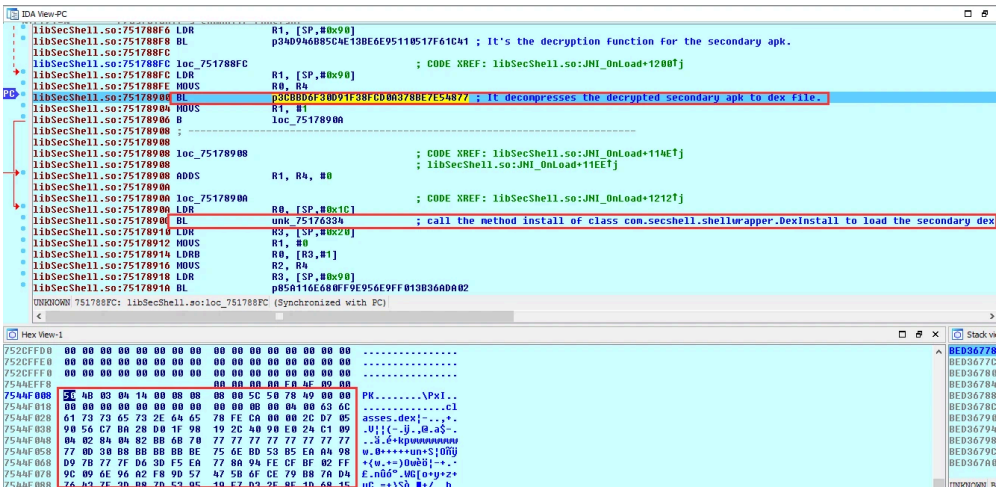

Figure 23. Decompressing the decrypted secondary apk and loading the secondary dex file

```
222  sub_BF60((signed int)&v11, 15, -95);
223  v6 = ((int (__fastcall *)(JNIEnv *, char *))(*v2)->FindClass)(v2, &v11);
224  v7 = ((int (__fastcall *)(JNIEnv *, int, const char *, const char *))(*v2)->GetMethodID)(
225         v2,
226         v6,
227         "getClassLoader",
228         "()Ljava/lang/ClassLoader;");
229  v8 = ((int (__fastcall *)(JNIEnv *, int, int))(*v2)->CallObjectMethod)(v2, v5, v7);
230  v9 = ((int (__fastcall *)(JNIEnv *, int, const char *, const char *))(*v2)->GetStaticMethodID)(
231         v2,
232         v5,
233         "install",
234         "(Ljava/lang/ClassLoader;Ljava/lang/String;)V");
235  result = ((int (__fastcall *)(JNIEnv *, int, int, int))(*v2)->CallStaticVoidMethod)(v2, v5, v9, v8);
236  if ( v108 != _stack_chk_guard )
237    _stack_chk_fail(result);
238  return result;
239}
```

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:

```
int __fastcall sub_7518394C(int result, _BYTE *a2, int a3)
{
int v3; // r1@1
int v4; // r3@5
unsigned int v5; // r3@7
int v6; // r6@7
int v7; // r5@7
char v8; // r2@8
int v9; // r4@9
int v10; // r3@9
int v11; // r7@11
_BYTE *v12; // r6@12
```

```
int v13; // r4@13
_BYTE *v14; // r1@15
int v15; // [sp+0h] [bp-138h]@1
int v16; // [sp+4h] [bp-134h]@1
_BYTE *v17; // [sp+8h] [bp-130h]@1
int v18; // [sp+10h] [bp-128h]@5
char v19[256]; // [sp+1Ch] [bp-11Ch]@6
int v20; // [sp+11Ch] [bp-1Ch]@1

v17 = a2;
v16 = result;
v20 = _stack_chk_guard;
v15 = a3;
v3 = 0;
if ( result <= 0x1FFFF )
{
v3 = 0x20000 - result;
if ( 0x20000 - result > a3 )
v3 = a3;
v15 = a3 - v3;
if ( v3 > 0 )
{
v18 = dword_751AF650;
v4 = 0;
do
{
v19[v4] = v4;
++v4;
}
while ( v4 != 256 );
v5 = 0;
v6 = 0;
v7 = 0;
do
{
v6 = (*(_BYTE *)(v18 + v5) + (unsigned __int8)v19[v7] + v6) & 0xFF;
v8 = v19[v7];
v5 = (v5 + 1) & -((v5 + 1 <= 0xF) + ((v5 + 1) >> 31));
```

```
v19[v7] = v19[v6];
v19[v6] = v8;
++v7;
}
while ( v7 != 256 );
v9 = 0;
result = 0;
v10 = 0;
while ( v9 != v16 )
{
v10 = (v10 + 1) & 0xFF;
v11 = (unsigned __int8)v19[v10];
++v9;
result = (v11 + result) & 0xFF;
v19[v10] = v19[result];
v19[result] = v11;
}
v12 = v17;
do
{
v10 = (v10 + 1) & 0xFF;
v13 = (unsigned __int8)v19[v10];
result = (result + v13) & 0xFF;
v19[v10] = v19[result];
v19[result] = v13;
*v12++ ^= v19[(v13 + (unsigned __int8)v19[v10]) & 0xFF];
}
while ( v12 != &v17[v3] );
}
}
if ( v15 > 0 )
{
v14 = &v17[v3];
result = (int)v14;
do
*v14++ ^= 0xACu;
while ( (signed int)&v14[-result] < v15 );
}
```

	if (v20 != _stack_chk_guard)
	result = ((int (*)(void))unk_75173E48)();
	return result;
	}

Source: https://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