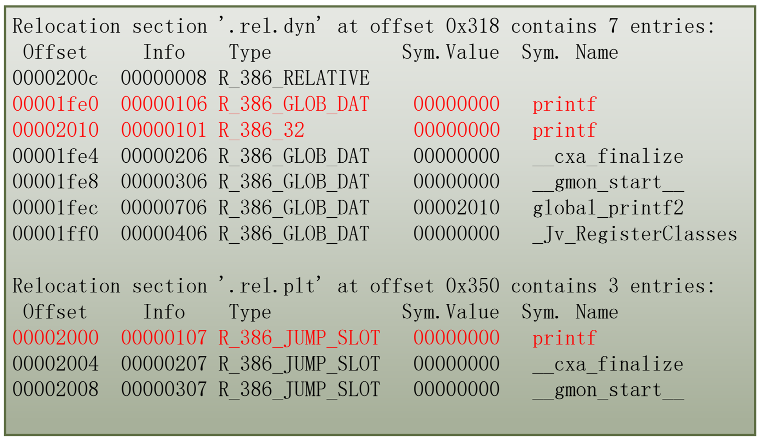
Improve the Security of Android* Applications using Hooking Techniques: Part 2

Contents

Study of the PIC Code in libtest_PIC.so

If the object is compiled in PIC mode, relocation is implemented differently. By observing the sections information of the libtest_PIC.so which is shown in Figure 17, the printf() relocation information is located in two relocation sections: .rel.dyn and .rel.plt. Two new relocation types R_386_GLOB_DAT and R_386_JMP_SLOT are used, and the absolute 32-bit address of the substituted function should be filled in with these offset addresses.



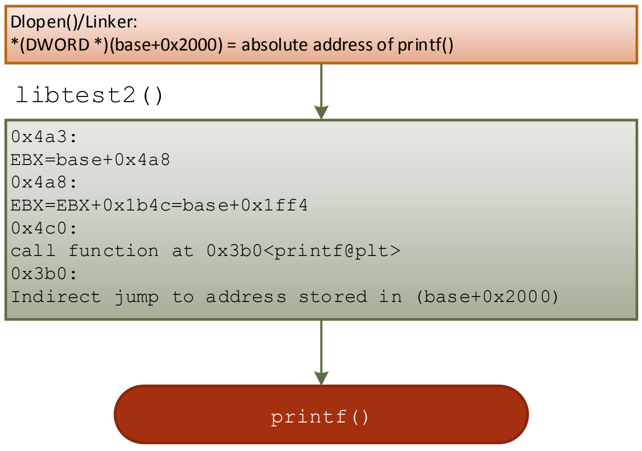
```
Relocation section '.rel.dyn' at offset 0x318 contains 7 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000200c  00000008 R_386_RELATIVE
00001fe0  00000106 R_386_GLOB_DAT    00000000   printf
00002010  00000101 R_386_32          00000000   printf
00001fe4  00000206 R_386_GLOB_DAT    00000000   __cxa_finalize
00001fe8  00000306 R_386_GLOB_DAT    00000000   __gmon_start__
00001fec  00000706 R_386_GLOB_DAT    00002010   global_printf2
00001ff0  00000406 R_386_GLOB_DAT    00000000   _Jv_RegisterClasses

Relocation section '.rel.plt' at offset 0x350 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00002000  00000107 R_386_JUMP_SLOT   00000000   printf
00002004  00000207 R_386_JUMP_SLOT   00000000   __cxa_finalize
00002008  00000307 R_386_JUMP_SLOT   00000000   __gmon_start__
```

Figure 1: *Relocation section of libtest_PIC.so*

The Figure 18 shows the assembly code of function libtest2() which is compiled in non-PIC mode. The entry addresses of printf() marked with red color are specified in the relocation sections .rel.dyn and .rel.plt in Figure 17.

```
...
000003b0 <printf@plt>:
 3b0: ff a3 0c 00 00 00       jmp    *0xc(%ebx)
...
0000049c <libtest2>:
...
 4a3: e8 ef ff ff ff          call   497 <__i686.get_pc_thunk.bx>
 4a8: 81 c3 4c 1b 00 00       add    $0x1b4c,%ebx
 4ae: 8b 83 ec ff ff ff       mov    -0x14(%ebx),%eax
 4b4: 89 45 f4                mov    %eax,-0xc(%ebp)

;printf("libtest2: 1st call to the original printf()\n");
 4b7: 8d 83 60 e5 ff ff       lea    -0x1aa0(%ebx),%eax
 4bd: 89 04 24                mov    %eax,(%esp)
 4c0: e8 eb fe ff ff          call   3b0 <printf@plt>

;printf("libtest2: 1st call to the original printf()\n");
 4c5: 8d 83 90 e5 ff ff       lea    -0x1a70(%ebx),%eax
 4cb: 89 04 24                mov    %eax,(%esp)
 4ce: e8 dd fe ff ff          call   3b0 <printf@plt>

;global_printf2("libtest2: global_printf2()\n");
 4d3: 8b 83 f8 ff ff ff       mov    -0x8(%ebx),%eax
 4d9: 8b 10                   mov    (%eax),%edx
 4db: 8d 83 bd e5 ff ff       lea    -0x1a43(%ebx),%eax
 4e1: 89 04 24                mov    %eax,(%esp)
 4e4: ff d2                   call   *%edx

;local_printf("libtest2: local_printf()\n");
 4e6: 8d 83 d9 e5 ff ff       lea    -0x1a27(%ebx),%eax
 4ec: 89 04 24                mov    %eax,(%esp)
 4ef: 8b 45 f4                mov    -0xc(%ebp),%eax
 4f2: ff d0                   call   *%eax
...
Disassembly of section .got:
00001fe0 <.got>:
      ...
Disassembly of section .got.plt:
00001ff4 <.got.plt>:
    1ff4:   20 1f                   and    %bl,(%edi)
      ...
    1ffe:   00 00                   add    %al,(%eax)
    2000:   b6 03                   mov    $0x3,%dh
    2002:   00 00                   add    %al,(%eax)
      ...
Disassembly of section .data:
      ...
00002010 <global_printf2>:
    2010:   00 00                   add    %al,(%eax)
      ...
```

Figure 2: *Disassemble code of libtest2(), compiled with -PIC parameter*



```
Dlopen()/Linker:
*(DWORD *)(base+0x2000) = absolute address of printf()

libtest2()
0x4a3:
EBX=base+0x4a8
0x4a8:
EBX=EBX+0x1b4c=base+0x1ff4
0x4c0:
call function at 0x3b0<printf@plt>
0x3b0:
Indirect jump to address stored in (base+0x2000)

printf()
```

Figure 3: *Working flow of 'printf("libtest2: 1st call to the original printf()\n");'*

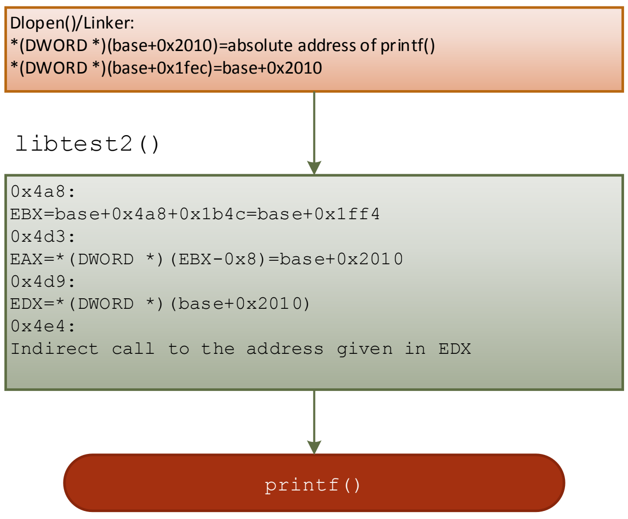

Dlopen()/Linker:
*(DWORD *)(base+0x2010)=absolute address of printf()
*(DWORD *)(base+0x1fec)=base+0x2010
libtest2()
0x4a8:
EBX=base+0x4a8+0x1b4c=base+0x1ff4
0x4d3:
EAX=*(DWORD *)(EBX-0x8)=base+0x2010
0x4d9:
EDX=*(DWORD *)(base+0x2010)
0x4e4:
Indirect call to the address given in EDX
printf()

Figure 4: *Working flow of 'global_printf2("libtest2: global_printf2()\n");'*

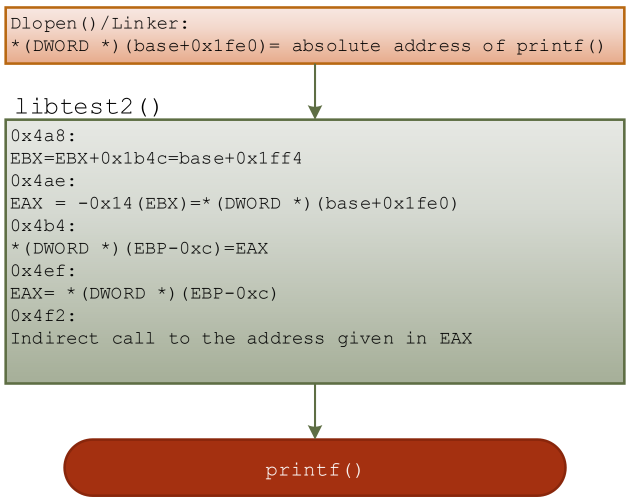

Dlopen()/Linker:
*(DWORD *)(base+0x1fe0)= absolute address of printf()
libtest2()
0x4a8:
EBX=EBX+0x1b4c=base+0x1ff4
0x4ae:
EAX = -0x14(EBX)=*(DWORD *)(base+0x1fe0)
0x4b4:
*(DWORD *)(EBP-0xc)=EAX
0x4ef:
EAX= *(DWORD *)(EBP-0xc)
0x4f2:
Indirect call to the address given in EAX
printf()

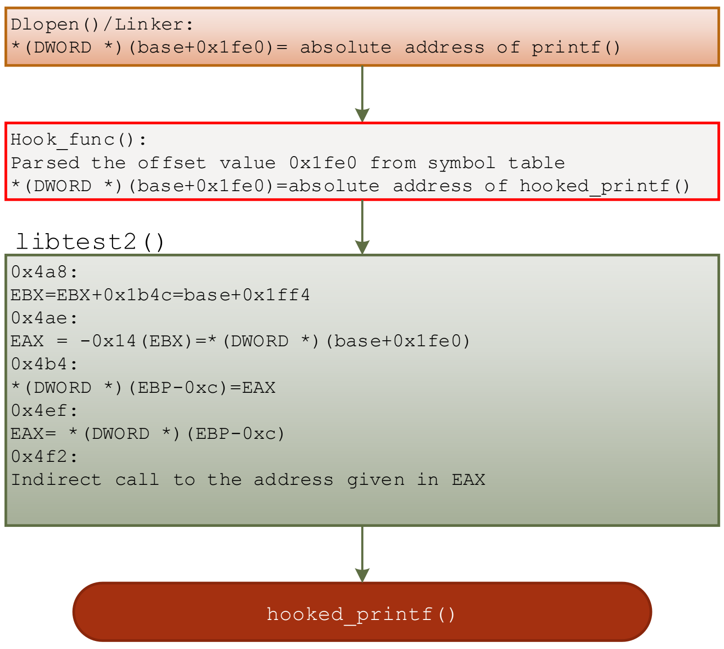
Figure 5: *Working flow of 'local_printf("libtest2: local_printf()\n");'*

From Figures 19-21, it can be seen that when working with the dynamic library generated with the -PIC parameter, the code in libtest2() will jump to the address placed in offset addresses 0x1fe0, 0x2010, and 0x2000, which are the entrances to printf().

Hook Solution

If the hook module wants to intercept the calls to printf() and redirect to another function, it should write the redirected function address to the offset addresses of the symbol 'printf' defined in the relocation sections, after the linker loaded the dynamic library into memory.

To replace the call of the printf() function with the call of the redirected hooked_printf() function, as shown in the software flow diagram in Figure 22, a hook function should be implemented between the dlopen() and libtest() calls. The hook function will first get the offset address of symbol printf, which is 0x1fe0 from the relocation section named .rel.dyn. The hook function then writes the absolute address of hooked_printf() function to the offset address. After that, when the code in libtest2() calls into the printf(), it will enter the hooked_printf() instead.



```
Dlopen()/Linker:
*(DWORD *)(base+0x1fe0)= absolute address of printf()

Hook_func():
Parsed the offset value 0x1fe0 from symbol table
*(DWORD *)(base+0x1fe0)=absolute address of hooked_printf()

libtest2()
0x4a8:
EBX=EBX+0x1b4c=base+0x1ff4
0x4ae:
EAX = -0x14(EBX)=*(DWORD *)(base+0x1fe0)
0x4b4:
*(DWORD *)(EBP-0xc)=EAX
0x4ef:
EAX= *(DWORD *)(EBP-0xc)
0x4f2:
Indirect call to the address given in EAX

hooked_printf()
```

Figure 6: *Example of how the hook function intercepts the call to printf() and reroutes the call to hooked_printf(). The original function calling process is described in Figure 21.*

To consider all the possible cases previously listed, the entire flow chart of the hook function is shown in Figure 23. And the part of the change in main() function is depicted in Figure 24.

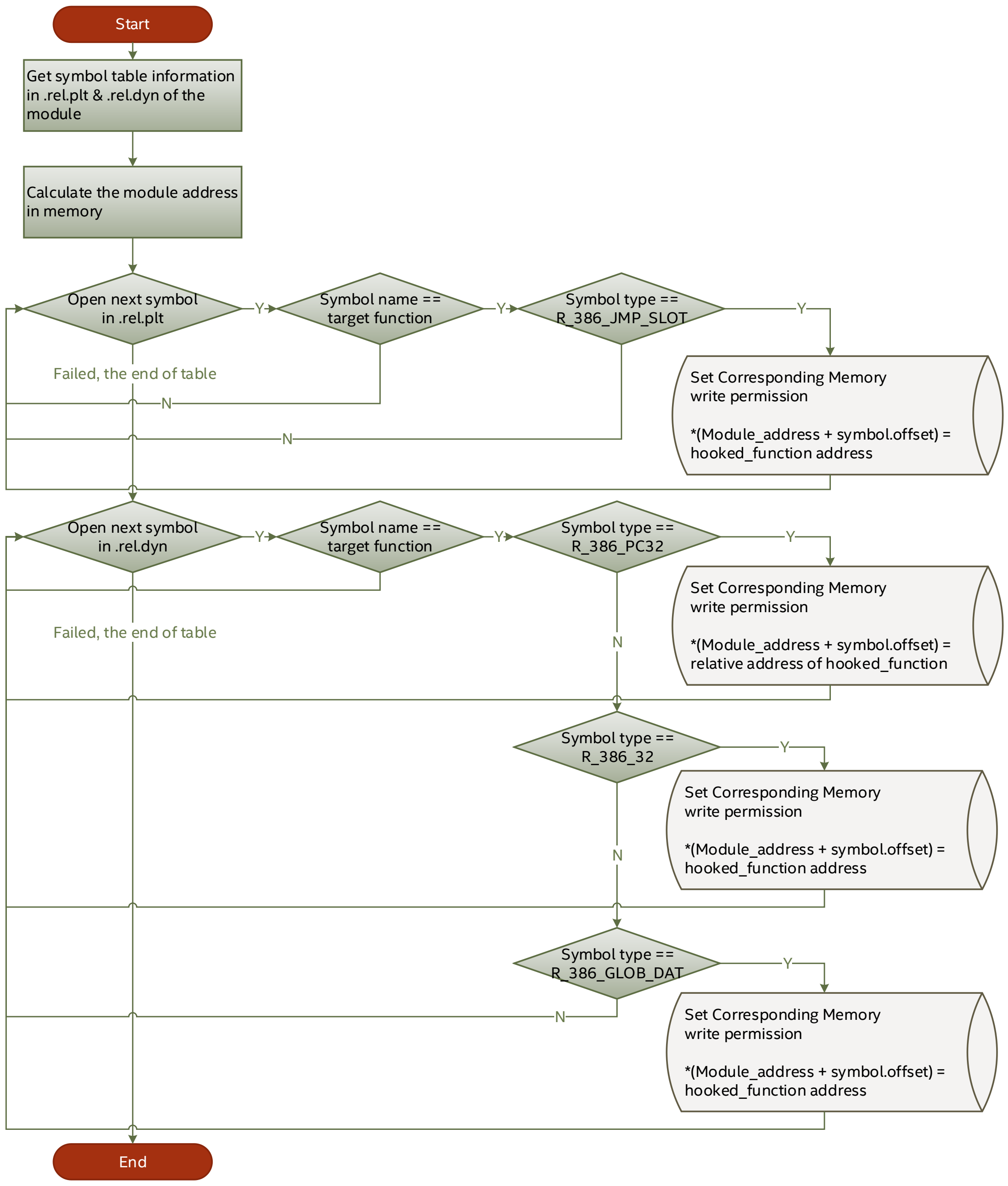
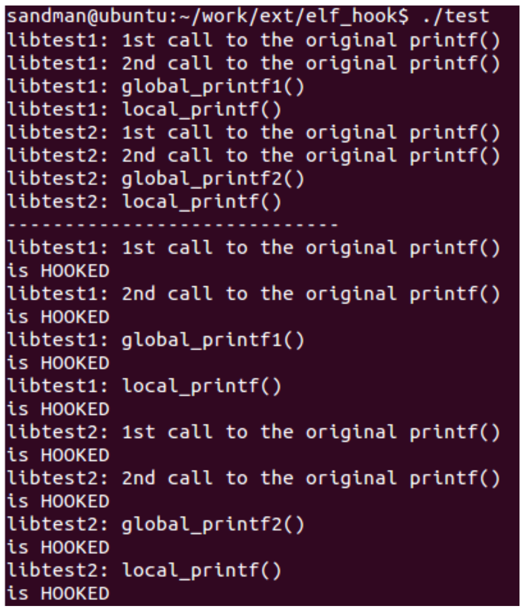

Start
Get symbol table information in .rel.plt & .rel.dyn of the module
Calculate the module address in memory
Open next symbol in .rel.plt
Y
Symbol name == target function
Y
Symbol type == R_386_JMP_SLOT
Y
Set Corresponding Memory write permission
*(Module_address + symbol.offset) = hooked_function address
Failed, the end of table
N
N
Open next symbol in .rel.dyn
Y
Symbol name == target function
Y
Symbol type == R_386_PC32
Y
Set Corresponding Memory write permission
*(Module_address + symbol.offset) = relative address of hooked_function
Failed, the end of table
N
Symbol type == R_386_32
Y
Set Corresponding Memory write permission
*(Module_address + symbol.offset) = hooked_function address
N
Symbol type == R_386_GLOB_DAT
Y
Set Corresponding Memory write permission
*(Module_address + symbol.offset) = hooked_function address
N
End

Figure 7: The flow chart of ELF hook module

```
int hooked_printf(char const *format,...)
{
//printf the original string then attach "is HOOKED" at the end
  va_list arg;
  va_start (arg, format);
  done = vfprintf (stdout, format, arg);
  va_end (arg);

  puts("is HOOKED");
}

int main()
{
...
   void *handle1 = dlopen("libtest_nonPIC.so", RTLD_LAZY);
   void *handle2 = dlopen("libtest_PIC.so", RTLD_LAZY);
...
   libtest1();
   libtest2();
   printf("-----------------------------\n");

//hook printf()
   original1 = elf_hook("libtest_nonPIC.so", handle1, "printf", hooked_printf);
   original2 = elf_hook("libtest_PIC.so", handle2, "printf", hooked_printf);
...
   libtest1();
   libtest2();
...
}
```

Figure 8: *Code in main() after hooking*

The output of the program is shown in Figure 25, you can see that when the first call to libtest1()/libtest2() executes, the printf() is called inside the functions. When calling the two functions again, after the hook functions are executed, the calls to the printf() are redirected to the hooked_printf() function. The hooked_printf() function will attach the string “is HOOKED” at the end of the normal printed string. Figure 26 shows the program running flow after hooking, compare with the original flow shown in Figure 8, the hooked_printf() has been injected into libtest1() and libtest2().



```
sandman@ubuntu:~/work/ext/elf_hook$ ./test
libtest1: 1st call to the original printf()
libtest1: 2nd call to the original printf()
libtest1: global_printf1()
libtest1: local_printf()
libtest2: 1st call to the original printf()
libtest2: 2nd call to the original printf()
libtest2: global_printf2()
libtest2: local_printf()
-----------------------------
libtest1: 1st call to the original printf()
is HOOKED
libtest1: 2nd call to the original printf()
is HOOKED
libtest1: global_printf1()
is HOOKED
libtest1: local_printf()
is HOOKED
libtest2: 1st call to the original printf()
is HOOKED
libtest2: 2nd call to the original printf()
is HOOKED
libtest2: global_printf2()
is HOOKED
libtest2: local_printf()
is HOOKED
```

Figure 9: *Output of the test program, printf() has been hooked*

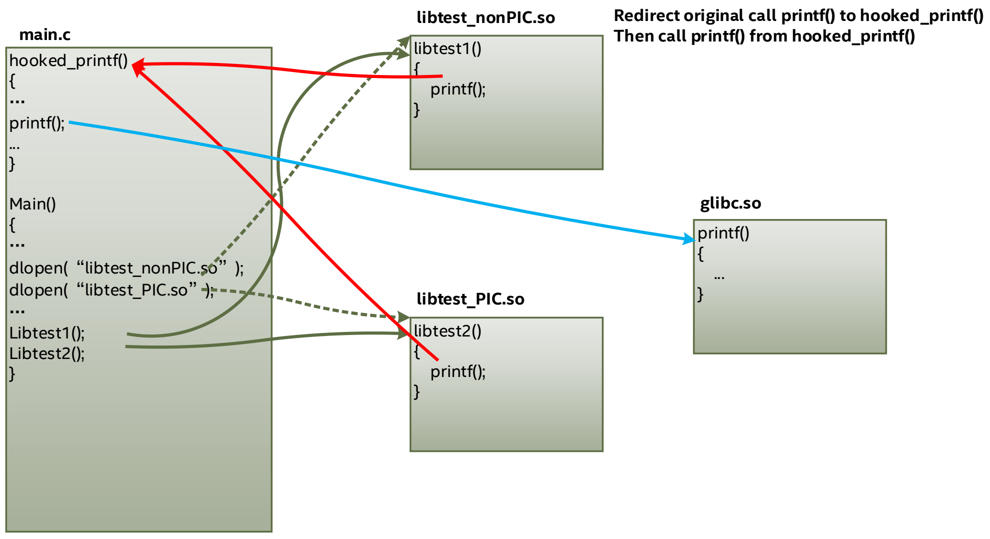


Figure 10: *The running flow of the test project after hooking*

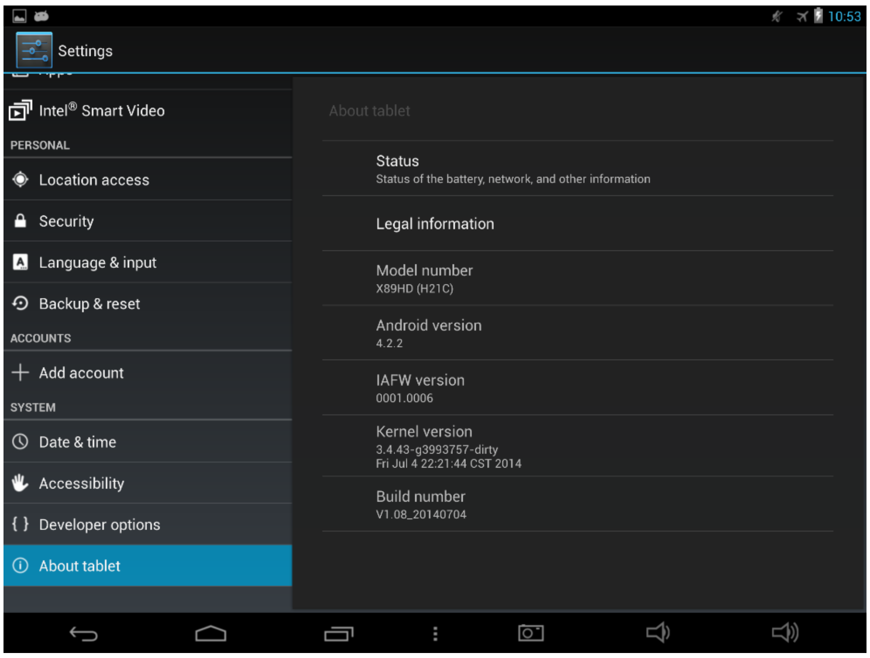
Case Study – a Hook-Based Protection Scheme in Android

Based on the studies of the hooking technique in the previous sections, we developed a plug-in to help Android application developers improve the security of their applications. Developers need to add only one Android native library to their projects and add one line of Java code to load this native library at start-up time. Then this library injects some protection code to other third-party libraries in the application. The protection code will aid encrypting the local file's input/output stream, as well as bypass the function __android_log_print() to avoid some user privacy leakage by printing debugging information through Logcat.

To verify the effectiveness of the protection plug-in, we wrote an Android application to simulate a scene of an application that contains a third-party library. In the test application, the third party library does two things:

1. When an external Java instruction calls the functions in the library, it will print some information by calling __android_log_print().
2. In the library, the code creates a file (/sdcard/data.dat) to save data in local storage without encryption, then reads it back and prints it on the screen. This action is to simulate the application trying to save some sensitive information in the local file system.

Figures 27-30 compare the screenshots of the test program, output of Logcat, and the content of the saving file in the device's local file system before and after hooking.



About tablet

Status
Status of the battery, network, and other information

Legal information

Model number
X89HD (H21C)

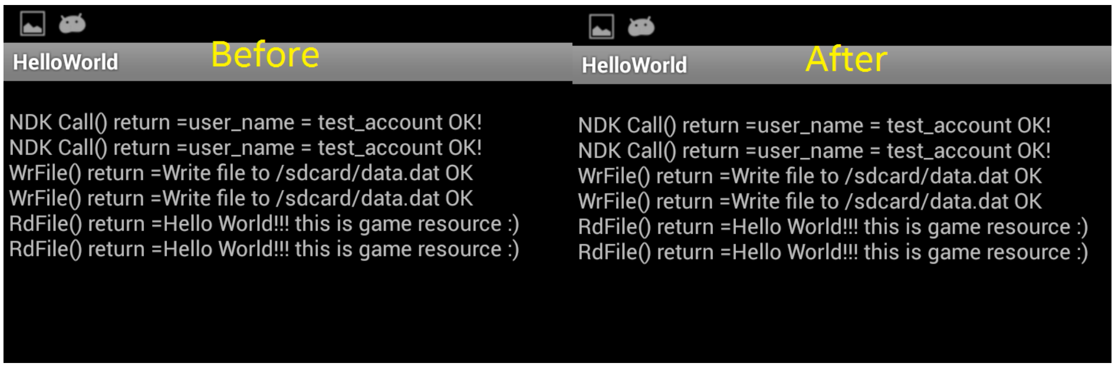
Android version
4.2.2

IAFW version
0001.0006

Kernel version
3.4.43-g3993757-dirty
Fri Jul 4 22:21:44 CST 2014

Build number
V1.08_20140704

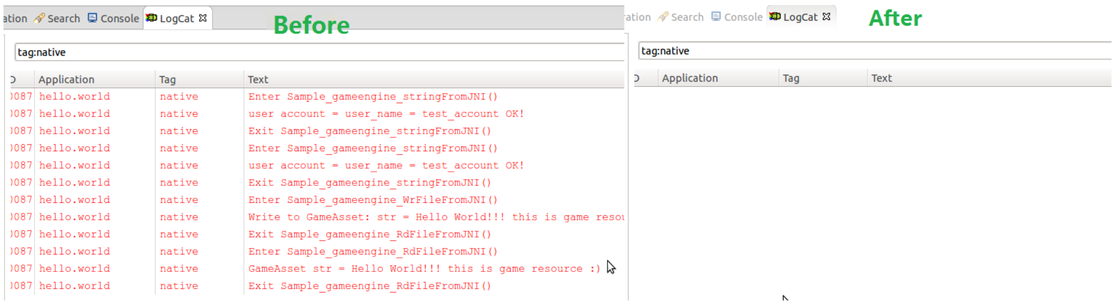
Figure 11: *The Android* platform is Teclast X89HD, Android 4.2.2*



```
NDK Call() return =user_name = test_account OK!
NDK Call() return =user_name = test_account OK!
WrFile() return =Write file to /sdcard/data.dat OK
WrFile() return =Write file to /sdcard/data.dat OK
RdFile() return =Hello World!!! this is game resource :)
RdFile() return =Hello World!!! this is game resource :)
```

```
NDK Call() return =user_name = test_account OK!
NDK Call() return =user_name = test_account OK!
WrFile() return =Write file to /sdcard/data.dat OK
WrFile() return =Write file to /sdcard/data.dat OK
RdFile() return =Hello World!!! this is game resource :)
RdFile() return =Hello World!!! this is game resource :)
```

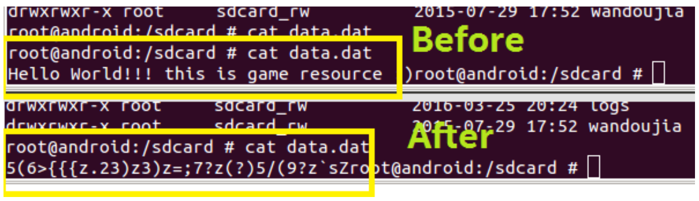
Figure 12: *App output - no change after hooking*



```
tag:native
D    Application   Tag      Text
)087 hello.world   native   Enter Sample_gameengine_stringFromJNI()
)087 hello.world   native   user account = user_name = test_account OK!
)087 hello.world   native   Exit Sample_gameengine_stringFromJNI()
)087 hello.world   native   Enter Sample_gameengine_stringFromJNI()
)087 hello.world   native   user account = user_name = test_account OK!
)087 hello.world   native   Exit Sample_gameengine_stringFromJNI()
)087 hello.world   native   Enter Sample_gameengine_WrFileFromJNI()
)087 hello.world   native   Write to GameAsset: str = Hello World!!! this is game resou
)087 hello.world   native   Exit Sample_gameengine_RdFileFromJNI()
)087 hello.world   native   Enter Sample_gameengine_RdFileFromJNI()
)087 hello.world   native   GameAsset str = Hello World!!! this is game resource :)
)087 hello.world   native   Exit Sample_gameengine_RdFileFromJNI()
```

```
tag:native
D    Application   Tag      Text
```

Figure 13: *Logcat output - empty after hooking*



```
drwxrwxr-x root      sdcard_rw          2015-07-29 17:52 wandoujia
root@android:/sdcard # cat data.dat
root@android:/sdcard # cat data.dat
Hello World!!! this is game resource )root@android:/sdcard #
```

```
drwxrwxr-x root      sdcard_rw          2016-03-25 20:24 logs
drwxrwxr-x root      sdcard_rw          2015-07-29 17:52 wandoujia
root@android:/sdcard # cat data.dat
5(6>{{{z.23)z3)z=;7?z(?)5/(9?z`sZroot@android:/sdcard #
```

Figure 14: *Local file 'data.dat' at /sdcard has been encrypted after hooking*

As the figures show, the running flow of the program after hooking is the same as the one without hooking. However, the Logcat cannot catch any output from the native library after hooking. Further, the content of the local file is no longer stored in a plain text format.

The plug-in helps the test application improve security against malicious attacks on collecting information via Logcat, as well as offline attacks to the local file system.

Conclusion

The hooking technique can be used in many development scenarios, providing seamless security protection to Android applications. Hook-based protection schemes can not only be used on Android, but also can be expanded to other operating systems such as Windows*, Embedded Linux, or other operating systems designed for Internet of Things (IoT) devices. It can significantly reduce the development cycle as well as maintenance costs. Developers can develop their own hook-based security scheme or use the professional third-party security solutions available on the market.

References

Redirecting functions in shared ELF libraries
Apriorit Inc, Anthony Shoumikhin, 25 Jul 2013
http://www.codeproject.com/Articles/70302/Redirecting-functions-in-shared-ELF-libraries

x86 API Hooking Demystified
Jurriaan Bremer
http://jbremer.org/x86-api-hooking-demystified/

Android developer guide
http://developer.android.com/index.html

Android Open Source Project
https://source.android.com/

About the Author

Jianjun Gu is a senior application engineer in the Intel Software and Solutions Group (SSG), Developer Relations Division, Mobile Enterprise Enabling team. He focuses on the security and manageability of enterprise application.

Notices