android动态加载so破解,[原创] 记一次so文件动态解密

转载

weixin_39840733 于 2021-05-26 18:45:27 发布 284 收藏 2

写在前面

整个程序基本上就是一个 动态注册 + so函数加密 的逻辑，中间加了一些parser的东西

主要考察了elf文件结构的一些知识以及在攻防对抗中防止IDA静态分析的姿势

题目描述

找到flag

WriteUp

360加固,先脱壳，看入口函数MainActivity

```
    }

    public void onClick(View view) {
        boolean test = MainActivity.this.test(this.val$inputEditText.getText().toString());
        String str = BuildConfig.FLAVOR;
        String str2 = "result";
        if (test) {
            new Builder(this.val$tmpcontext).setTitle(str2).setMessage("Congratulations!").show();
            this.val$inputEditText.setText(str);
            return;
        }
        new Builder(this.val$tmpcontext).setTitle(str2).setMessage("Sorry,try again?").show();
        this.val$inputEditText.setText(str);
    }
}

public native void onCreate(Bundle bundle);

public native String stringFromJNI();

public native boolean test(Object obj);

static {
    StubApp.interface11(1345);
    System.loadLibrary("native-lib");
}
```

具体的逻辑写到so里了， 使用IDA打开so文件,先看有没有.init和.init_array,发现只有.init_array节，

```
.init_array:0001BC84                ; Segment type: Pure data
.init_array:0001BC84                AREA .init_array, DATA
.init_array:0001BC84                ; ORG 0x1BC84
.init_array:0001BC84 41 9A 00 00    DCD .datadiv_decode4192348989750430380+1
.init_array:0001BC88 05 89 00 00    DCD unk_8905
.init_array:0001BC88                ; .init_array   ends
.init_array:0001BC88
LOAD:0001BC8C                ; ELF Dynamic Information
```

跟进去一看又是字符串解密函数，解密之后，代码如下(这里我根据解密后的数据进行了重命名)

```
unsigned int datadiv_decode4192348989750430380()
{
v29 = 0;
do
{
v0 = v29;
```

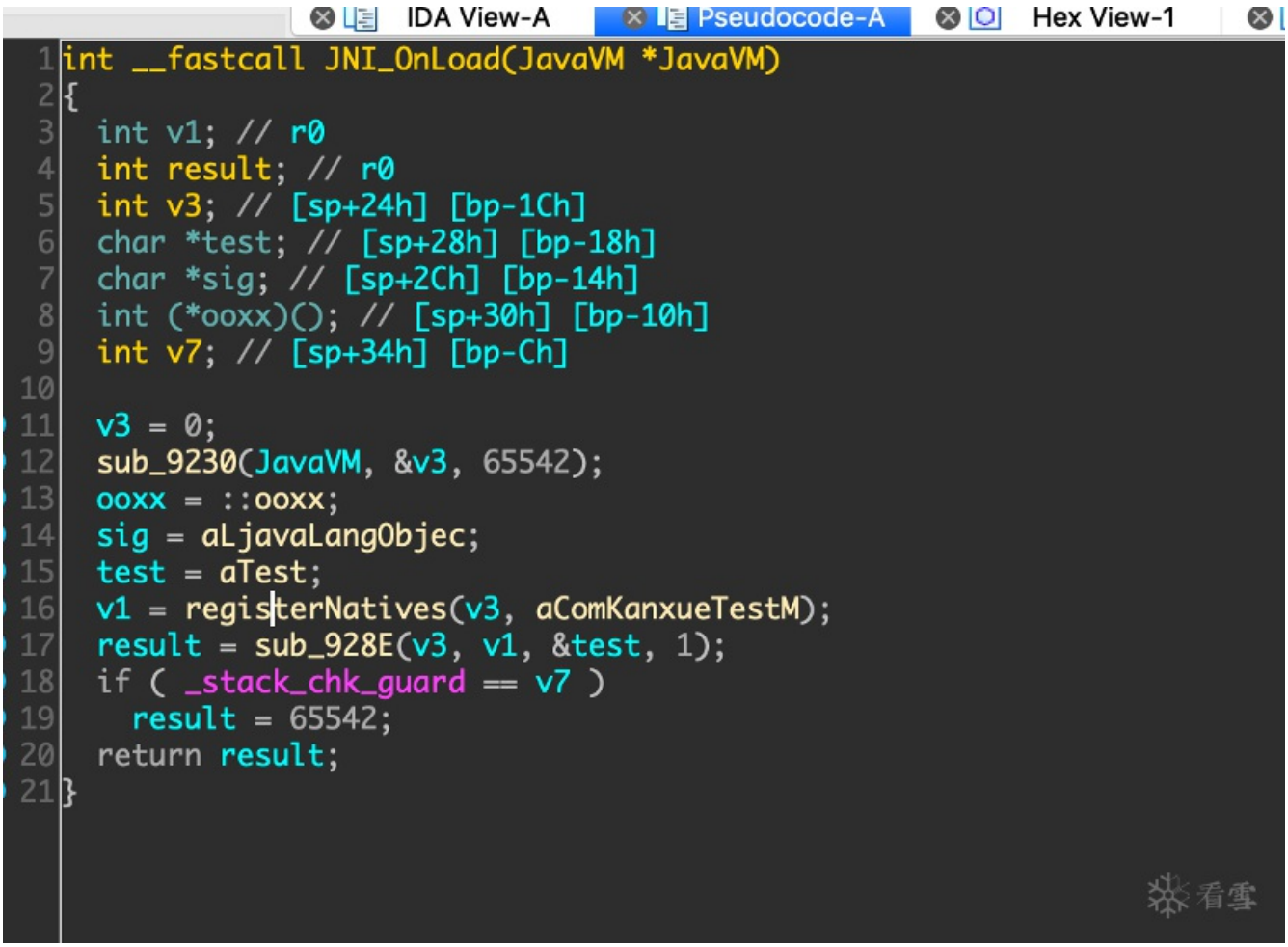
```
Find_ooxx_failed[v29++] ^= 0x14u;
}
while ( v0 < 0x10 );
v28 = 0;
do
{
v1 = v28;
mem_privilege_change_failed[v28++] ^= 0xD3u;
}
while ( v1 < 0x1B );
v27 = 0;
do
{
v2 = v27;
kanxuetest[v27++] ^= 0x63u;
}
while ( v2 < 0xA );
v26 = 0;
do
{
v3 = v26;
Hello_from_Cjiajia[v26++] ^= 0x3Fu;
}
while ( v3 < 0xE );
v25 = 0;
do
{
v4 = v25;
test[v25++] ^= 0xF3u;
}
while ( v4 < 4 );
```

```
v24 = 0;
do
{
v5 = v24;
sig_Ljava_lang_Object_Z[v24++] ^= 0xFAu;
}
while ( v5 < 0x15 );
v23 = 0;
do
{
v6 = v23;
com_kanxue_test_MainActivity[v23++] ^= 0x2Du;
}
while ( v6 < 0x1C );
v22 = 0;
do
{
v7 = v22;
maps[v22++] ^= 0xF5u;
}
while ( v7 < 0xD );
v21 = 0;
do
{
v8 = v21;
r[v21++] ^= 0xF8u;
}
while ( !v8 );
v20 = 0;
do
{
```

```
v9 = v20;
open_failed[v20++] ^= 0xE6u;
}
while ( v9 < 0xB );
v19 = 0;
do
{
v10 = v19;
heng[v19++] ^= 0x66u;
}
while ( !v10 );
v18 = 0;
do
{
v11 = v18;
Find__dynamic_segment[v18++] ^= 0x2Du;
}
while ( v11 < 0x15 );
v17 = 0;
do
{
v12 = v17;
Find_needed__section_failed[v17++] ^= 9u;
}
while ( v12 < 0x1C );
v16 = 0;
do
{
v13 = v16;
basic_string[v16++] ^= 0x9Eu;
}
```

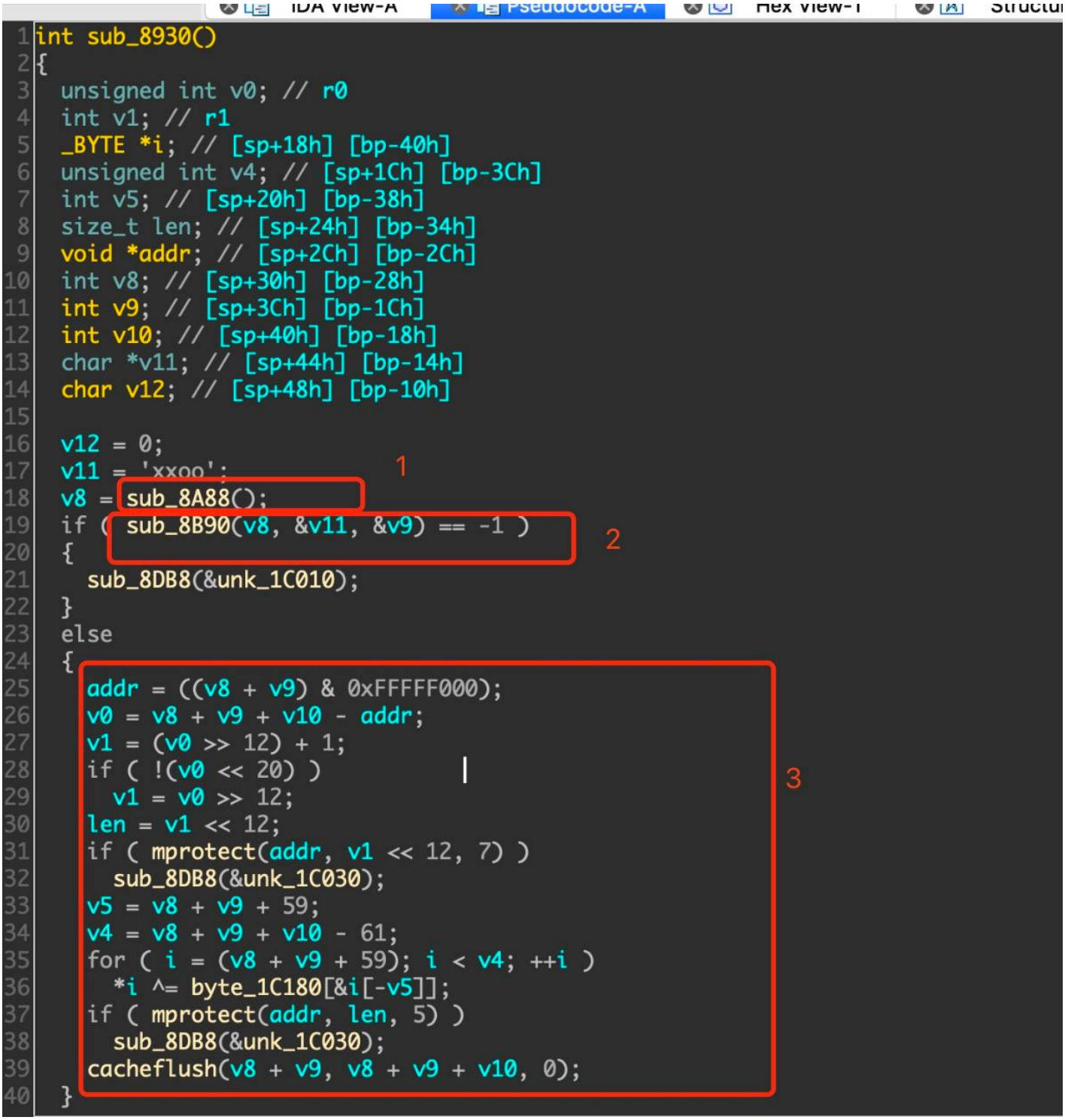
```
while ( v13 < 0xC );
v15 = 0;
do
{
result = v15;
allocate_exceeds_maximum_supported_size[v15++] ^= 0xDBu;
}
while ( result < 0x43 );
return result;
}
```

回过来看JNI_Onload函数,



其实就是将native函数test函数动态注册到ooxx函数，直接看ooxx函数

可以发现除了调用了sub_8930之外，就是一堆垃圾代码，先跟进sub_8930函数



这里我把函数分为三块，先看第一块

```
1  int get_so_base_addr()
2  {
3    __pid_t v0; // ST1C_4
4    const char *nptr; // ST20_4
5    int base_addr; // r0
6    FILE *stream; // [sp+18h] [bp-1040h]
7    unsigned int v4; // [sp+24h] [bp-1034h]
8    char s; // [sp+3Bh] [bp-101Dh]
9    char v6; // [sp+103Bh] [bp-1Dh]
10   int v7; // [sp+104Ch] [bp-Ch]
11
12   v4 = 0;
13   _aeabi_memcpy(&v6, "libnative-lib.so", 17);
14   v0 = getpid();
15   sprintf(&s, maps, v0);
16   stream = fopen(&s, r);
17   if ( stream )
18   {
19     while ( fgets(&s, 4096, stream) )
20     {
21       if ( strstr(&s, &v6) )
22       {
23         nptr = strtok(&s, heng);                // 以-分割字符串
24                                                 //
25         v4 = strtoul(nptr, 0, 16);              // 获取libnative-lib.so的加载基地址
26         break;
27       }
28     }
29   }
30   else
31   {
32     puts(open_failed);
33   }
34   base_addr = fclose(stream);
35   if ( _stack_chk_guard == v7 )
36     base_addr = v4;
37   return base_addr;
38 }
```

经过分析，实际上就是读/proc/self/maps的标准输出，从而获取到对应于libnaitve-lib.so的那一行，然后以-分割字符串，并将分割后的第一段解析为16进制的数，实际上就是获取libnaitve-lib.so的加载基地址。

```
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
root@hammerhead:/ # cat /proc/20704/maps | grep native-lib
b4115000-b412e000 r-xp 00000000 b3:1c 162904     /data/app/com.kanxue.test-1/lib/arm/libnative-lib.so
b412f000-b4131000 r--p 00019000 b3:1c 162904     /data/app/com.kanxue.test-1/lib/arm/libnative-lib.so
b4131000-b4132000 rw-p 0001b000 b3:1c 162904     /data/app/com.kanxue.test-1/lib/arm/libnative-lib.so
root@hammerhead:/ #
```


再看第二块，也就是sub_8B90函数的实现

int __fastcall find_symbol_value_and_size(int base_addr, char *a2, _DWORD *a3)

{

int v3; // ST38_4

_DWORD *ELF_Hash_Table; // ST28_4

unsigned int v5; // ST20_4

int elf_hash_chain; // [sp+14h] [bp-5Ch]

int ELF_Symbol_Table; // [sp+24h] [bp-4Ch]

int elf_hash_table; // [sp+28h] [bp-48h]

int string_table; // [sp+2Ch] [bp-44h]

```
int elf_symbol_table; // [sp+30h] [bp-40h]
_DWORD *v12; // [sp+34h] [bp-3Ch]
int dynamic_segment_base_addr; // [sp+40h] [bp-30h]
_DWORD *header_table; // [sp+44h] [bp-2Ch]
signed int i; // [sp+4Ch] [bp-24h]
unsigned int j; // [sp+4Ch] [bp-24h]
int elf_hash_bucket; // [sp+4Ch] [bp-24h]
char v18; // [sp+57h] [bp-19h]
char v19; // [sp+57h] [bp-19h]
char v20; // [sp+57h] [bp-19h]
_DWORD *value; // [sp+58h] [bp-18h]
char *s2; // [sp+5Ch] [bp-14h]
int so_base_addr; // [sp+60h] [bp-10h]
so_base_addr = base_addr;
s2 = a2;
value = a3;
v18 = -1;
header_table = (base_addr + *(base_addr + 0x1C));// header_table_offset
for ( i = 0; i < *(base_addr + 0x2C); ++i ) // *(base_addr + 0x2C) = 8
{
if ( *header_table == 2 )
{
v18 = 0;
puts_0(); // find_dynamic_segment
break;
}
header_table += 8;
}
if ( v18 )
goto LABEL_27;
dynamic_segment_base_addr = header_table[2] + so_base_addr;// 找到dynamic_segment的虚拟地址
```

```
v19 = 0;
for ( j = 0; j < header_table[4] >> 3; ++j )
{
v12 = (dynamic_segment_base_addr + 8 * j);
if ( *(dynamic_segment_base_addr + 8 * j) == 6 )
{
elf_symbol_table = v12[1]; // 0x1f0
++v19;
}
if ( *v12 == 4 )
{
elf_hash_table = v12[1]; // 0x46e0
v19 += 2;
}
if ( *v12 == 5 )
{
string_table = v12[1]; // 0x1d00
v19 += 4;
}
if ( *v12 == 10 )
{
v3 = v12[1]; // 0x1eb6
v19 += 8;
}
}
if ( (v19 & 0xF) != 0xF )
{
puts_0();
LABEL_27:
return -1;
}
```

```
ELF_Hash_Table = (so_base_addr + elf_hash_table);// v4 =elf_hash_table

v5 = turn_ooxx(s2); // v5 = 0x766f8

ELF_Symbol_Table = so_base_addr + elf_symbol_table;// ELF Symbol Table

elf_hash_chain = &ELF_Hash_Table[*ELF_Hash_Table + 2];

v20 = -1;

for ( elf_hash_bucket = ELF_Hash_Table[v5 % *ELF_Hash_Table + 2];// ELF_Hash_Table[v5 %
*ELF_Hash_Table + 2] = 0x4918

elf_hash_bucket;

elf_hash_bucket = *(elf_hash_chain + 4 * elf_hash_bucket) )

{

if ( !strcmp((so_base_addr + string_table + *(ELF_Symbol_Table + 16 * elf_hash_bucket)), s2) )//
string_table[] = "ooxx"

{

v20 = 0;

break;

}

}

if ( v20 )

goto LABEL_27;

*value = *(ELF_Symbol_Table + 16 * elf_hash_bucket + 4);

value[1] = *(ELF_Symbol_Table + 16 * elf_hash_bucket + 8);

return 0;

}
```

这个地方你仔细地去分析对比，会发现其实就是一个读so文件的对应于symbol name为ooxx的symbol table表项中的value和size,其实就是读ooxx的函数起始地址以及函数大小。其实也就是一个parser的过程之一

对了，这个函数中的一行，也就是v5 = turn_ooxx(s2);这里调用的turn_ooxx函数中的伪代码直接copy出来跑一跑，就可以得到v5的值。我也没有分析这个过程，直接跑的。。

接着看sub_8930函数的第三块。

```
{
  addr = ((base_addr + value) & 0xFFFFF000);
  v0 = base_addr + value + size - addr;
  v1 = (v0 >> 12) + 1;
  if ( !(v0 << 20) )
    v1 = v0 >> 12;
  len = v1 << 12;
  if ( mprotect(addr, v1 << 12, 7) )
    puts_0();
  v5 = base_addr + value + 59;                  // 0x8e00
  v4 = base_addr + value + size - 61;           // 0x8fd0
  for ( i = (base_addr + value + 59); i < v4; ++i )
    *i ^= byte_1C180[&i[-v5]];
  if ( mprotect(addr, len, 5) )
    puts_0();
  cacheflush(base_addr + value, base_addr + value + size, 0);
}
return _stack_chk_guard;
```

经过分析会发现，围绕mprotect函数将这个部分再次分成三块，分别实现功能为

第一块，设置ooxx函数所在内存页为rwx

第二块，还原ooxx函数中code

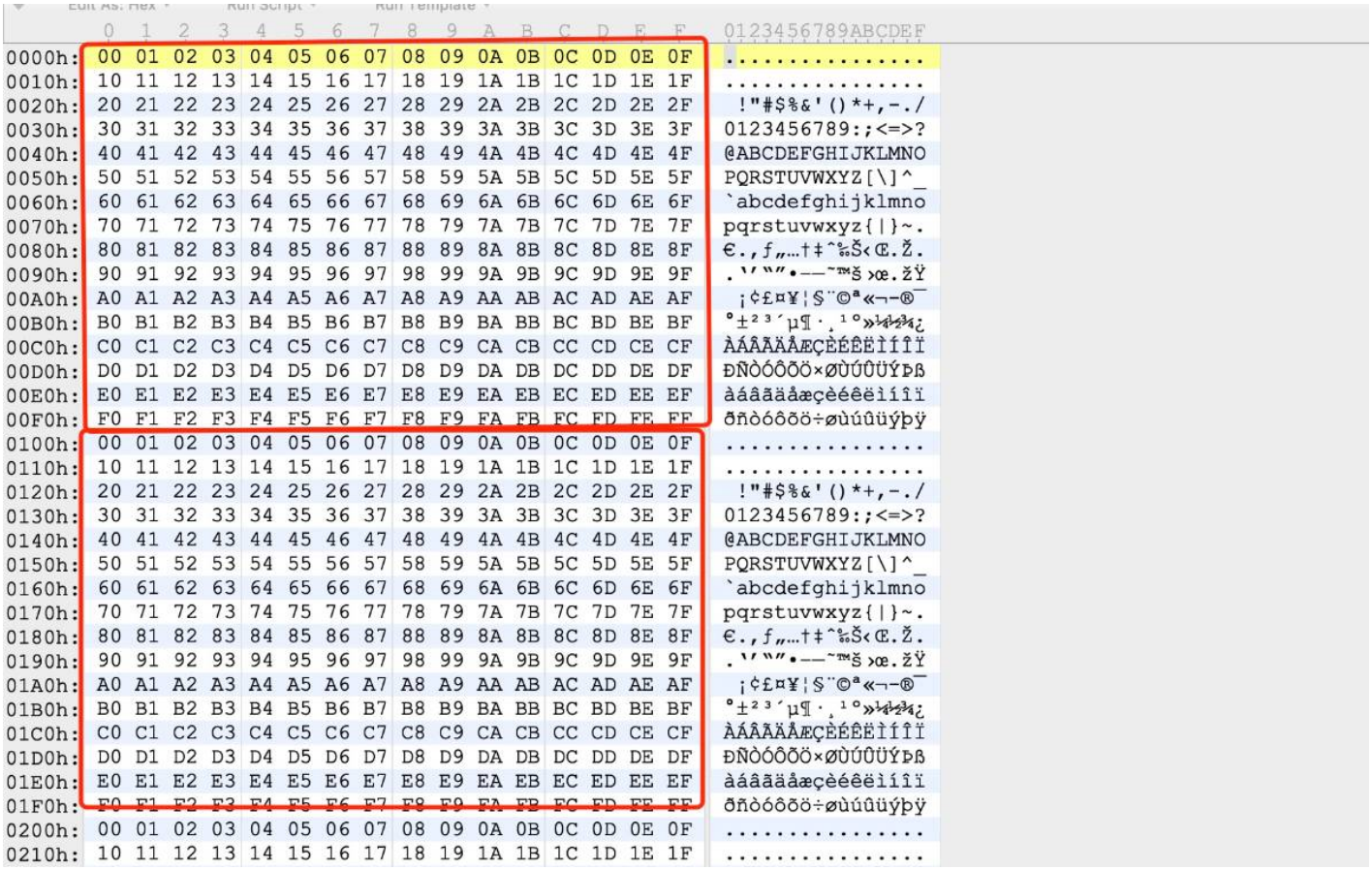
第三块，恢复内存页为r-x

这里第二块中的*i ^= byte_1C180[&i[-v5]];这个部分，再加上byte_1C180实际上在bss段，不想再去分析了，直接动态吧。

这里使用objection在动态运行时dump出对应内存中的数据，

```
libjiagu.so                  0xdbe9e000  548864 (536.0 KiB)   /data/data/com.kanxue.test/.jiagu/libjiagu.so
libnative-lib.so             0xdbf90000  18784 (116.0 KiB)   /data/app/com.kanxue.test-QB3Os_I7ai5723rWcKzTjA==/lib/ar
gralloc.msm8992.so           0xd706c000  57344 (56.0 KiB)    /vendor/lib/hw/gralloc.msm8992.so
libmemalloc.so               0xd708c000  32768 (32.0 KiB)    /system/lib/libmemalloc.so
libqdMetaData.so             0xd700f000  20480 (20.0 KiB)    /system/lib/libqdMetaData.so
libqdutils.so                0xd70db000  49152 (48.0 KiB)    /system/lib/libqdutils.so
libqservice.so               0xd7124000  40960 (40.0 KiB)    /system/lib/libqservice.so
frida-agent-32.so            0xd3d60000  14798848 (14.1 MiB) /data/local/tmp/re.frida.server/frida-agent-32.so
linker                       0xf477b000  557056 (544.0 KiB)  /system/bin/linker
com.kanxue.test on (google: 8.1.0) [usb] # memory dump from_base 0xdbfac180 1000 byte_1C180
Dumping 1000.0 B from 0xdbfac180 to byte_1C180
Memory dumped to file: byte_1C180
com.kanxue.test on (google: 8.1.0) [usb] # memory dump from_base 0xdbfac180 1000 byte_1C180
Destination file byte_1C180 already exists
Override? [y/N]: y
Dumping 1000.0 B from 0xdbfac180 to byte_1C180
Memory dumped to file: byte_1C180
com.kanxue.test on (google: 8.1.0) [usb] # memory dump from_base 0xdbf98dc5 464  code
Dumping 464.0 B from 0xdbf98dc5 to code
Memory dumped to file: code
```

使用010 editor查看对应文件



很明显那就是0-255的字节咯，继续看伪码，会发现实际上这里的&i[-v5]实际上就相当于i-v5,而v5为i的初值，那么patch脚本就有了

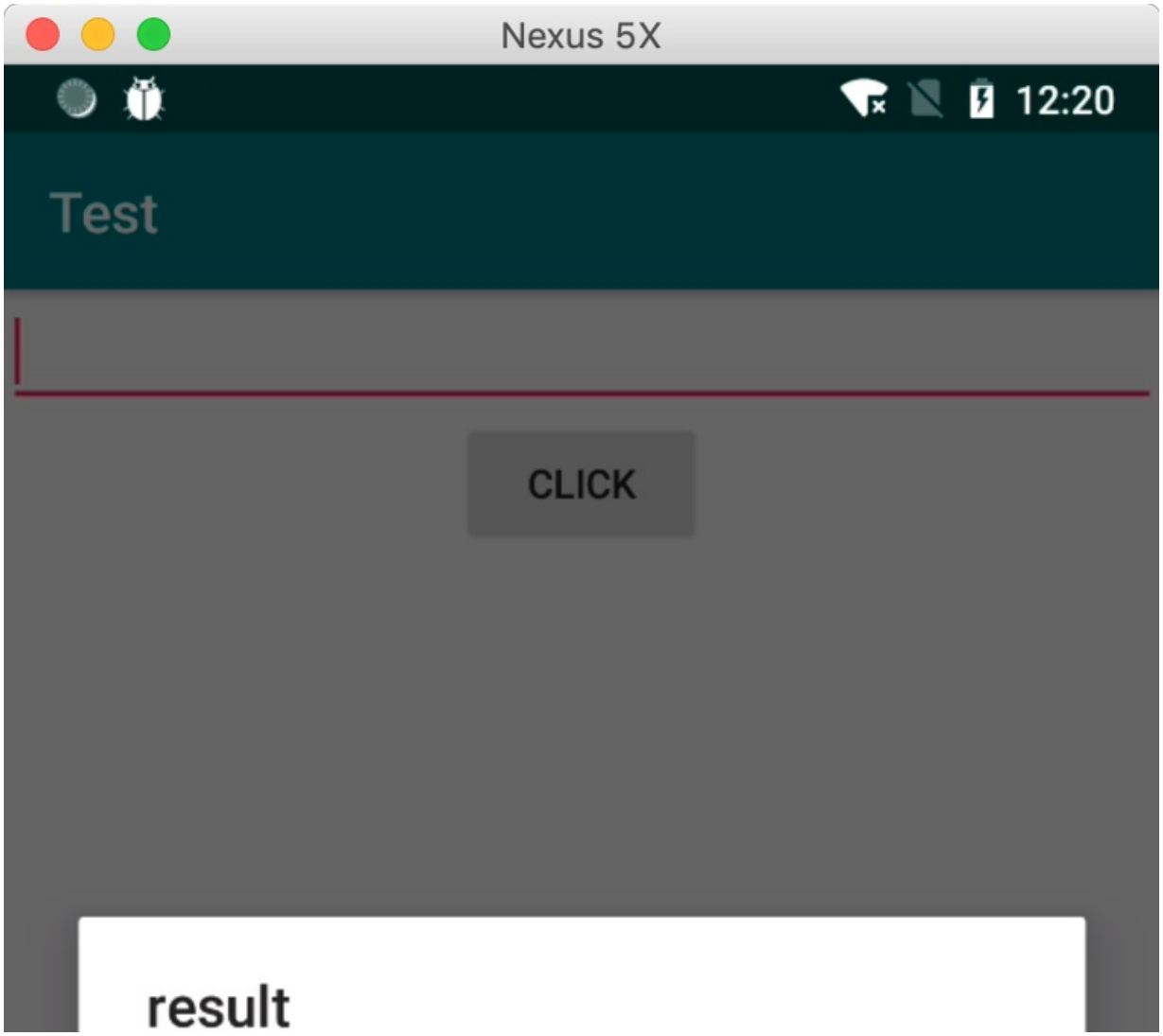
```
def patchBytes(addr,length):

for i in range(0,length):

byte=get_bytes(addr + i,1)

byte = ord(byte) ^ (i%0xff)

patch_byte(addr+i,byte)

patchBytes(0x8e00,0x8fd0-0x8e00)
```

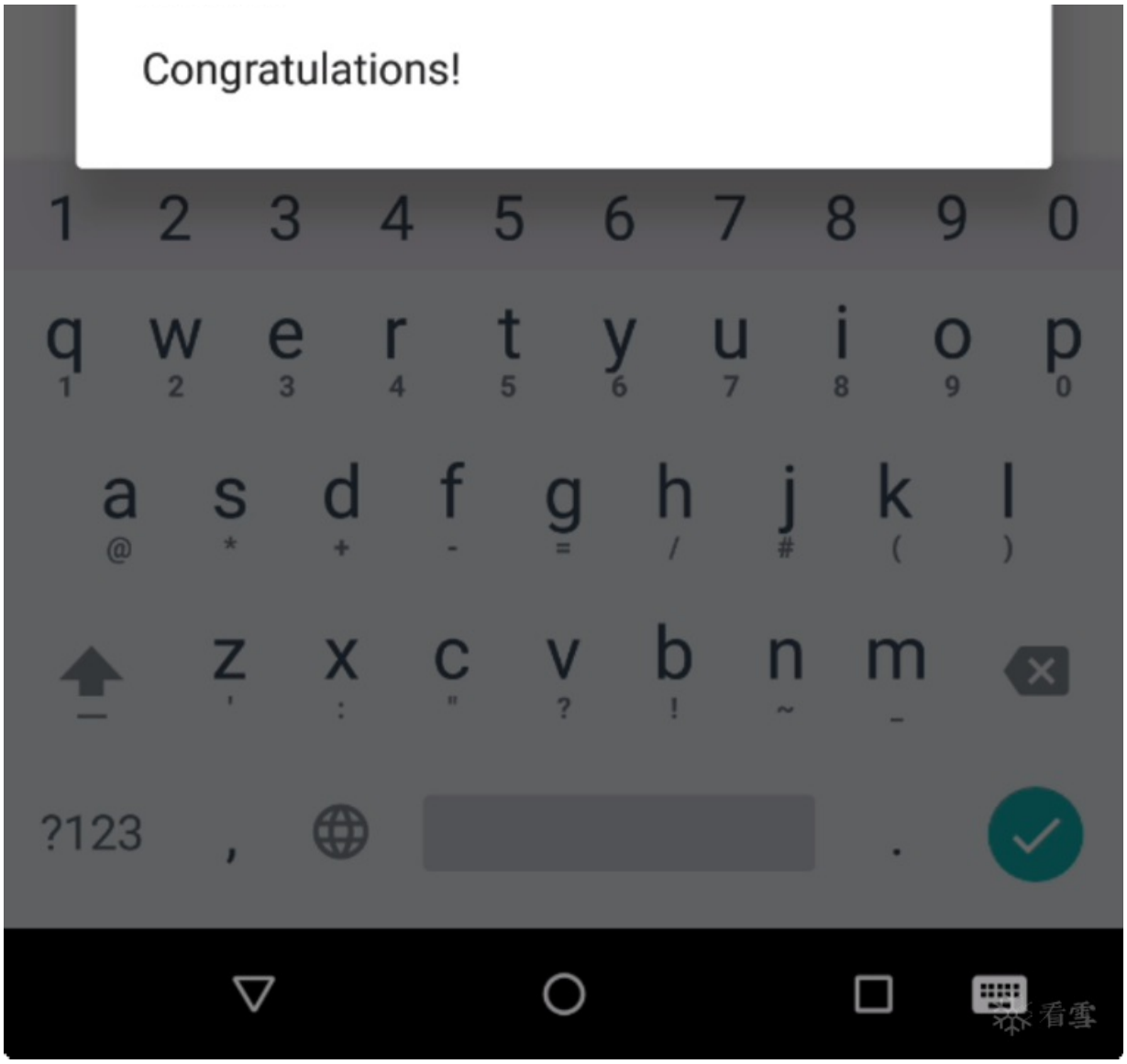
执行这个脚本之后,查看ooxx函数内容

```
int __fastcall ooxx(JNIEnv *a1, int a2, int a3)

{

JNIEnv *v3; // ST20_4

int input; // r0

int v5; // r0

unsigned __int8 v7; // [sp+17h] [bp-19h]

v3 = a1;

sub_8930(); //

v7 = 0;
```

```
input = getStringUtf(v3);
if ( input )
{
input = strcmp(aKanxuetest, input);
if ( !input )
{
input = 1;
v7 = 1;
}
}
v5 = *(input + 8);
sub_8930();
return v7;
}
```

最终会发现，实际上ooxx就是拿我的输入和kanxuetest进行对比。。验证下





拿到flag

后记

整个程序实际上真正难的地方在于看出parser的过程，不过我猜如果写过parser相信会很容易的看出来，还有

另外，这个程序有点类似于之前寒冰师傅说的在函数执行开始之前对函数内容进行恢复，函数执行结束时再还原回加密状态，再加上插入了一堆MOV R0, R0这种无效代码，让我感觉真像so层的"函数抽取壳"的实现。。神奇的题目，最后，附上附件

最后于 2020-7-8 11:32

被Simp1er编辑

，原因：

上传的附件：

附件.zip

(2.14MB，76次下载)