[原创]360加固保分析 优

[軍] 1

大侠

2020-6-11 20:08 12695

背景：

调试手机是Android 6.0的32位的手机。样本是自己写的一个经过360加固的小程序。加固时间为今年的5月份左右。

步骤：

总体来说就是分为两大步，首先是分析libjiagu.so，从中dump出第二个so，也就是解释器so，第二步就是修复分析解释器so，找到指令映射的对应关系，得出指令映射表。

第一步：

首先看一下.init_array做了什么事

```
init_array:0000FD78 ; ===========================================================
init_array:0000FD78
init_array:0000FD78 ; Segment type: Pure data
init_array:0000FD78                 AREA .init_array, DATA
init_array:0000FD78                 ; ORG 0xFD78
init_array:0000FD78                 DCD sub_1CE0
init_array:0000FD7C                 ALIGN 0x10
init_array:0000FD7C ; .init_array   ends
init_array:0000FD7C
```

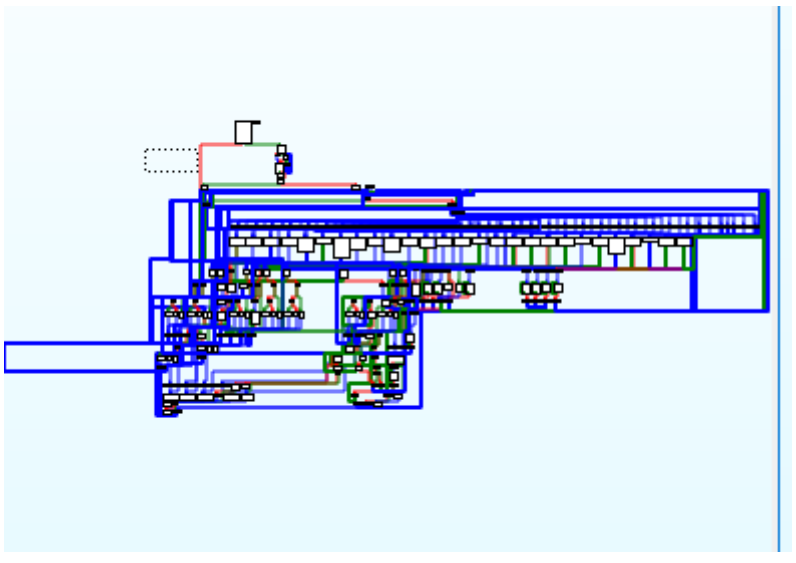
可以看到这里只有一个函数，经过我的分析这里并没有做什么重要的事。

然后继续看JNI_OnLoad区域。

```
.04 JNI_OnLoad                              ; DATA XREF: LOAD:0(
.04                                         ; sub_9684+10↑o ...
.04 ; __unwind {
.04                 STMFD   SP!, {R0-R7,LR}
.08                 LDR     R3, =(sub_2DAC - 0xA114)
.0C                 ADD     R3, R3, PC      ; sub_2DAC
.10                 LDR     R0, =3
.14                 BLX     R3              ; sub_2DAC
.18                 MOV     R12, R0
.1C                 LDMFD   SP!, {R0-R7,LR}
.20                 MOV     PC, R12
20 ; End of function JNI_OnLoad
```

发现是这样子的，然后我们的静态分析就变得困难了，只能开始我们的动态调试之旅了。

通过动态调试我从JNI_OnLoad一路跟到这个重要的函数，_Z10__fun_a_18Pcj。



可以看到这个函数被混淆的非常严重。但是经过我的分析有用的分支基本只有case31和case35两个分支。这里可以说一下case31是一个反调试的分支，case35是进入第二个so的入口，别的分支有的也做了一些事，但是对于我

反调试：

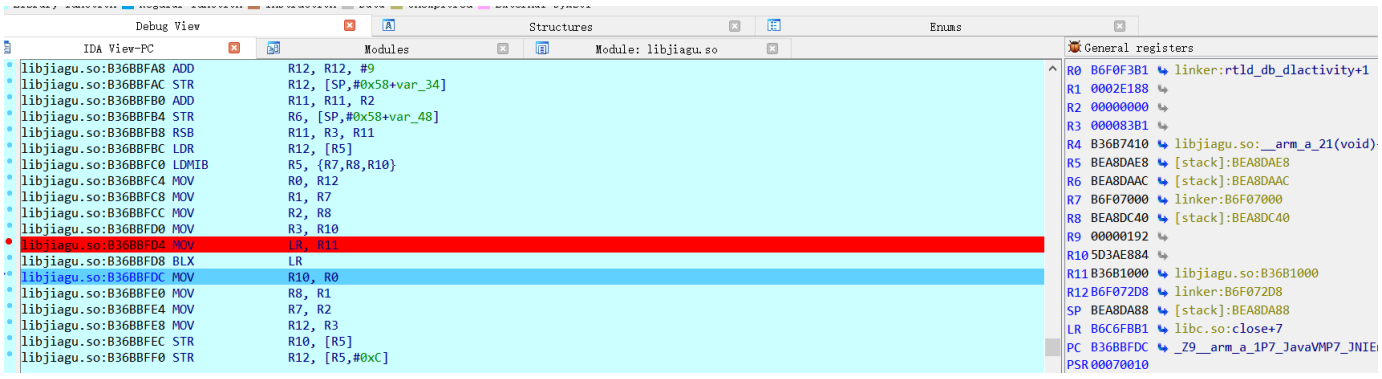
我们可以在这里下断，然后跟随R0寄存器。如图

```
.so:B36BBF98 ADD             R3, R4, R12             ; jumptable B36BBC24 case 31
.so:B36BBF9C LDR             R11, [R3,#5]
.so:B36BBFA0 LDR             R2, [R5,#0x38]
.so:B36BBFA4 LDR             R3, [R3,#1]
.so:B36BBFA8 ADD             R12, R12, #9
.so:B36BBFAC STR             R12, [SP,#0x58+var_34]
.so:B36BBFB0 ADD             R11, R11, R2
.so:B36BBFB4 STR             R6, [SP,#0x58+var_48]
.so:B36BBFB8 RSB             R11, R3, R11
.so:B36BBFBC LDR             R12, [R5]
.so:B36BBFC0 LDMIB           R5, {R7,R8,R10}
.so:B36BBFC4 MOV             R0, R12
.so:B36BBFC8 MOV             R1, R7
.so:B36BBFCC MOV             R2, R8
.so:B36BBFD0 MOV             R3, R10
.so:B36BBFD4 MOV             LR, R11
.so:B36BBFD8 BLX             LR
.so:B36BBFDC MOV             R10, R0
```

这个版本的一共有三个反调试的点，一个是符号，一个是端口，一个是时间。

```
3  ·····.$.k./sys
E  tem/bin/linker..
9  rtld_db_dlactivi
E  ty.^......k.....
```

此处为第一个反调试了。



我们可以在内存窗口修改指令，改成无用指令。

第一个反调试之后，两次f9就是第二个反调试的点了，也就是时间反调试。

```
B36BBFD4 MOV             LR, R11
B36BBFD8 BLX             LR
B36BBFDC MOV             R10, R0
```

从这个跳转进入，即为时间调试。

```
:B36B7A18 CMP             R0, #0
:B36B7A1C STMFD           SP!, {R3,LR}
:B36B7A20 BNE             loc_B36B7A4C
:B36B7A24 BL              unk_B36B0ADC
:B36B7A28 LDR             R3, =(dword_B3734758 - 0xB36B7A34)
:B36B7A2C ADD             R3, PC, R3              ; dword_B3734758
:B36B7A30 LDR             R3, [R3]
:B36B7A34 RSB             R0, R3, R0
:B36B7A38 CMP             R0, #3
```

```
R0 0000002E
```

在这里我们只需把R0寄存器置0就可以。

还有一个是端口检测，我们只需更改ida的默认调试端口就可以了。

```
# ./as -p31928
```

到此，反调试告一段落。

接下来我们继续分析，sub_5CE8为第二个so加载的关键地方。

如图，这个是ida f5的结果。

```
int __fastcall sub_5CE8(int a1, int a2)
{
  int v2; // r5
  _DWORD *v3; // r4
  int v4; // r3

  v2 = a2;
  v3 = a1;
  if ( !a2 )
    return 0;
  *(a1 + 96) = 0;
  v4 = *(a2 + 176);
  if ( !v4 )
  {
    *(a1 + 80) = 0;
    if ( sub_5610(a1, a2) && sub_5708(v3, v2) && sub_5984(v3, v2) && sub_59A8(v3, v2) )
      return sub_5C34(v3);
    return 0;
  }
  if ( v4 != 1 )
    return 0;
  *(a1 + 80) = 1;
  if ( !sub_540C(a1, a2) || !sub_5984(v3, v2) )
    return 0;
  return sub_59A8(v3, v2);
}
```

但是我们会发现一个有趣的地方。

```
  if ( sub_5610(a1, a2) && sub_5708(v3, v2) && sub_5984(v3, v2) && sub_59A8(v3, v2) )
    return sub_5C34(v3);
  return 0;
}
```

有没有觉得这个地方与Android源码中加载so的函数有点相似。

```
bool ElfReader::Load() {
  return ReadElfHeader() &&
         VerifyElfHeader() &&
         ReadProgramHeader() &&
         ReserveAddressSpace() &&
         LoadSegments() &&
         FindPhdr();
}
```

在这个函数断下就可以dump第二个so。

但是可惜的是它并没有走这个分支，而是这个分支

```
  if ( !sub_540C(a1, a2) || !sub_5984(v3, v2) )
    return 0;
```

sub_540C这个函数一共要解密四个部分，分别为程序头表、JMPREL、RELA、DYNAMIC。

```
50  if ( v8 )
51  {
52    v10 = &v9[v8];
53    do
54      *v9++ ^= v4;
55    while ( v9 != v10 );
56    v11 = v3[16];
57  }
58  else
59  {
60    v11 = 0;
61  }
62  v12 = v5 + v11 + 4;
63  memcpy(&dest, v12, 4u);
64  v13 = calloc(1u, dest);
65  v3[21] = v13;
66  if ( !v13 )
67    return 0;
68  v14 = dest;
69  memcpy(v13, (v12 + 4), dest);
70  v15 = v3[21];
71  if ( v14 )
72  {
73    v16 = 0;
74    do
75      *(v15 + v16++) ^= v4;
76    while ( v14 != v16 );
77  }
78  v17 = v12 + v14 + 4;
```

一个循环即为一部分。我们把这四部分dump出来。

```
R0  00000100
R1  AEEC0105 [anon:libc_malloc]:AEEC0105
R2  00000000
R3  B06BF300 [anon:libc_malloc]:B06BF300
R4  000000F7
R5  BEA8DAE0 [stack]:BEA8DAE0
R6  AEEC0001 [anon:libc_malloc]:AEEC0001
R7  AEEC0000 [anon:libc_malloc]:AEEC0000
R8  00000100
R9  BEA8DB90 [stack]:BEA8DB90
R10 E1A00004
R11 B36B88EC libjiagu.so:__arm_a_2(char *,unsi
R12 00000000
SP  BEA8DA98 [stack]:BEA8DA98
LR  B36B446C libjiagu.so:__self_open+198C
PC  B36B4478 libjiagu.so:__self_open+1998
PSR 20070010
```

R7寄存器地址即为整个第二个so的开始地址，我们可以把整个malloc的区域全dump下来。

[anon:libc_malloc]	AEEC0000	AEFC0000	R	W	.	D	.	byte	00	public DATA

需要注意的是

```
if ( v8 )
{
  v10 = &v9[v8];
  do
    *v9++ ^= v4;
  while ( v9 != v10 );
  v11 = v3[16];
}
```

一定要在这个循环执行完之后dump，即解密完。

在这里说一下如何修复，因为这个so没有elf_header，所以需要我们自己修复，并且把其他三部分放到原来的地方。在这里特别说一下，R7寄存器为so的开始地址，R1寄存器为此部分在so的偏移，R0为要解密的数据大小。如

总结一下，我们把dump出来的四部分按照偏移直接覆盖到dump出来的so中就可以，因为我们dump出来的so的那四部分并未解密。但是需要注意先修复elf_header。

第二步：

按惯例来分析init_array区域，经过分析init_array区域并没有做什么重要的事情。

然后分析JNI_OnLoad，这个函数做了很多事情，最重要的就是注册stub方法和dex的加载。

```
DCD sub_E3CC+1
DCD aInterface14        ; DATA XREF: JNI_OnLoad+66↑o
                        ; JNI_OnLoad:off_F3A8↑o
                        ; "interface14"
DCD aILjavaLangStri_0   ; "(I)Ljava/lang/String;"
DCD sub_102A0+1
DCD aMark               ; "mark"
DCD aV_0                ; "()V"
DCD sub_47C38+1
DCD aInterface5         ; "interface5"
DCD aLandroidAppApp_1   ; "(Landroid/app/Application;)V"
DCD sub_10300+1
DCD aInterface11        ; "interface11"
DCD aIV                 ; "(I)V"
DCD sub_1CE60+1
DCD aInterface12        ; "interface12"
DCD aLdalvikSystemD_4   ; "(Ldalvik/system/DexFile;)Ljava/util/Enu"..
DCD sub_10688+1
DCD aInterface21        ; "interface21"
DCD aLandroidAppApp_1   ; "(Landroid/app/Application;)V"
DCD sub_120D0+1
DCD aInterface7         ; "interface7"
DCD aLandroidAppApp_2   ; "(Landroid/app/Application;Landroid/cont"..
DCD sub_14174+1
DCD aInterface8         ; "interface8"
DCD aLandroidAppApp_2   ; "(Landroid/app/Application;Landroid/cont"..
DCD sub_10B1C+1
DCD aInterface22        ; "interface22"
DCD aILjavaLangStri_1   ; "(I[Ljava/lang/String;[I)V"
DCD sub_11E34+1
EXPORT getSoName2
```

Dex的加载在这里完成

```
201   v21 = operator new(0x14u);
202   v22 = sub_17564(v21, env);
203 }
204 if ( v22 && (*(*v22 + 8))() )
205 {
206   sub_221E0(130);
207   v25 = env;
208   s[0] = 0x69;
```

通过动态调试我们进入这个函数sub_1A104。这个函数很长，完成dex的一系列操作。

对了忘记说了，我们可以通过case35分支进入第二个so

```
ibjiagu.so:B36BBEBC                             ; CODE XREF: _Z9__arm_a_1P7_JavaVMP7_
ibjiagu.so:B36BBEBC                             ; _Z9__arm_a_1P7_JavaVMP7_JNIEnvPvRi:
ibjiagu.so:B36BBEBC ADD             R3, R12, #2 ; jumptable B36BBC24 case 35
ibjiagu.so:B36BBEC0 LDR             R2, [R4,R3]
ibjiagu.so:B36BBEC4 STR             R3, [SP,#0x58+var_34]
ibjiagu.so:B36BBEC8 ADD             R12, R12, #6
ibjiagu.so:B36BBECC LDR             R11, [R5,R2,LSL#2]
ibjiagu.so:B36BBED0 STR             R12, [SP,#0x58+var_34]
ibjiagu.so:B36BBED4 STR             R6, [SP,#0x58+var_48]
ibjiagu.so:B36BBED8 LDR             R12, [R5]
ibjiagu.so:B36BBEDC LDMIB           R5, {R7,R8,R10}
ibjiagu.so:B36BBEE0 MOV             R0, R12
ibjiagu.so:B36BBEE4 MOV             R1, R7
ibjiagu.so:B36BBEE8 MOV             R2, R8
ibjiagu.so:B36BBEEC MOV             R3, R10
ibjiagu.so:B36BBEF0 MOV             LR, R11
ibjiagu.so:B36BBEF4 BLX             LR
ibjiagu.so:B36BBEF8 MOV             R12, R0
```

因为我们的目标不是dump dex 这里dex的加载就不多说了，直接分析被native化的函数如何还原。

分析我们dump的dex会发现这样的现象，

```
/* access modifiers changed from: protected */
public native void onCreate(Bundle bundle);

public native String stringFromJNI();

static {
    StubApp.interface11(1347);
    System.loadLibrary("native-lib");
}
```


这里有个interface11 那么这个动态注册的函数就是我们的切入点。

```
DCD aInterface11        ; "interface11"
DCD aIV                 ; "(I)V"
DCD sub_1CE60+1
```

但是我们发现这个函数是这个样子的，

```
void __fastcall __noreturn sub_1CE60(JNIEnv *a1, int a2, int a3)
{
  int v3; // r4
  int v4; // r0

  v3 = a3;
  v4 = sub_59100();
  JUMPOUT(__CS__, *(&off_BE188 + (sub_59150(v4, v3) == 0)));
}
```

这个jumpout我确实解决的不是很好，就是静态分析和动态调试硬着头皮看的，我也试过网上分享的一些方法，但是效果都不怎么好。如果跳转的地方是连在一起的还好，可以进行patch并p成一个函数，那样参数，变量还可以看，如果中间夹杂着别的东西，不能p成一个函数，那就是一团糟，索性我就这样看了。希望大佬能够在这里给点意见。

在这里说一下360 进行native化的函数都是绑定了同一个函数，类似于分发的作用。然后，经过分析找到这个函数，sub_BF75C。其实这个函数就是调用了sub_1C480。

到这里我们进入正题，这个最重要的函数sub_1C480的分析。

我们悲剧的发现，这里跟前面一样，都被处理了。

```
void __fastcall __noreturn sub_1C480(int a1)
{
  int v1; // r6
  int v2; // r0
  int v3; // [sp+48h] [bp-30h]

  v1 = a1;
  v2 = sub_59100();
  sub_1CC44(&v3, v2, v1);
  JUMPOUT(__CS__, *(&off_BE100 + (v3 == 0)));
}
```

幸好这个函数不是很长，我就一步步跟出来了。

然后进行jni接口的赋值，然后根据注册vmp方法时的描述信息执行哪条分支，

一种是直接将指令复制到原地方，然后用jni函数调用，调用完成后再将其清空。另一种方式是自己实现的解释器，边解密边解释执行，且解密后的指令也是替换过的。

```
sub_1C4CC                               ; DATA XREF: .data:off_BE108↓o

arg_10          =  0x10
arg_14          =  0x14
arg_40          =  0x40

000             ADD             R0, SP, #arg_40
000             MOV             R1, R5
000             BL              sub_1CC80 ; JNI接口
000             LDR             R1, [R7,#0x10]
000             LDR             R0, [R7]
000             LDR             R2, [R6]
000             LDR             R3, [R1,#0x3C]
000             LDR             R5, [R1,#0x4C]
000             LDR             R1, [R1,#0x5C]
000             ADD             R1, R0
000             ADD.W           R8, R1, R2,LSL#3
000             ADDS            R2, R0, R5
000             ADD             R0, R3
000             LDRH.W          R1, [R8,#2]
000             ADD.W           R1, R1, R1,LSL#1
000             LDR.W           R1, [R2,R1,LSL#2]
000             ADD.W           R1, R0, R1,LSL#2
000             MOV             R0, R7
000             BL              sub_1E328
000             MOV             R9, R0
000             BLX             strlen
000             MOV             R11, R0
000             LDR             R0, =(off_BE110 - 0x1C510)
000             LDR             R1, [R6,#0x18]
000             ADD             R0, PC  ; off_BE110 ; 判断第五个字段是否为0
000             CMP             R1, #0
000             IT NE
```

描述信息是这样的，

4F 3C 00 00 01 00 00 00 5C 60 1C 00 03 00 00 00

00 00 00 00 00 00 00 00 00 00 00 00 6D 07 00 00

00 00 00 00 00 00 00 00 00 00 00 00 65 78 00 00

4F 3C 00 00 methon_id

01 00 00 00 0为实例方法，1为静态方法

5C 60 1C 00 code_item在dex中的偏移

03 00 00 00 第几个方法

00 00 00 00 如果为0表示直接解密原来code_item中的指令执行，不为0则执行的时候， 把该处的指令复制到原来的code_item，执行完后又清除。

其他字段意义就不是很大了。

接下来执行sub_1C51E，解析参数，分析类名和包名

```
000             BL              sub_1E328
000             MOV             R1, R0
000             ADD             R0, SP, #arg_34
000             BL              sub_58B70 ; 包名
000             LDR             R2, [R7,#0x10]
000             LDR.W           R0, [R8,#4]!
000             LDR             R1, [R7]
000             LDR             R2, [R2,#0x3C]
000             STR             R7, [SP,#arg_C]
000             ADD             R1, R2
000             ADD.W           R1, R1, R0,LSL#2
000             MOV             R0, R7
000             BL              sub_1E328 ; oncreate
000             MOV             R5, R0
000             BLX             strlen
000             ADD.W           R10, SP, #arg_28
```

再就是执行sub_1C5D0，根据是否为静态方法来分别走两个分支。

```
sub_1C5D0                               ; DATA XREF: .data:off_BE14
                                        ; .data:off_BE148↓o

arg_10          =  0x10

000             STRD.W          R0, R6, [SP,#4]
000             LDR             R6, [SP,#arg_10]
000             LDRB.W          R0, [R10,#0x20]
000             LDRH            R2, [R5]
000             LDR             R1, [R6,#0x1C]
000             LDR.W           R3, [R8]
000             EORS            R0, R1
000             EORS            R0, R2
000             EORS            R0, R3
000             STRB.W          R0, [R7,#0x24]
000             LDRB.W          R0, [R10,#0x20]
000             STRB.W          R0, [R7,#0x25]
000             LDR             R0, [R6]
000             LDR             R1, =(off_BE150 - 0x1C600)
000             STR             R0, [R7,#0x28]
000             LDRB            R0, [R6,#4] ; 第二个字段
000             ADD             R1, PC  ; off_BE150
000             LDRH            R2, [R5,#2]
000             LDRH            R3, [R5]
000             ADDS            R5, R4, #4
000             LDR.W           R0, [R1,R0,LSL#2]
000             SUB.W           R10, R3, R2
000             MOV             PC, R0
    ; End of function sub_1C5D0
```

但是，最后都会执行到这个地方

```
.C7D4     loc_1C7D4                             ; CODE XREF: sub_1C60E+A↑j
.C7D4 000               MOVS            R4, #1
.C7D6 000               LDR             R6, =(off_BE158 - 0x1C7E4)
.C7D8 000               MOVS            R0, #0
.C7DA 000               CMP             R4, R11
.C7DC 000               IT CC
.C7DE 000               MOVCC           R0, #1
.C7E0 000               ADD             R6, PC  ; off_BE158
.C7E2 000               MOV.W           R8, #1
.C7E6 000               LDR.W           R0, [R6,R0,LSL#2]
.C7EA 000               MOV             PC, R0
.C7EA     ; END OF FUNCTION CHUNK FOR sub_1C60E
.C7EC
.C7EC     ; =============== S U B R O U T I N E =======================================
.C7EC
.C7EC
.C7EC     sub_1C7EC                             ; CODE XREF: sub_1C60E+1DC↑j
.C7EC                                           ; sub_1C60E+292↓j ...
.C7EC 000               LDRB.W          R0, [R9,R4]
.C7F0 000               CMP             R0, #74
.C7F2 000               BCS             loc_1C806
.C7F4 000               CMP             R0, #70
.C7F6 000               BCS             loc_1C878
.C7F8 000               CMP             R0, #68
.C7FA 000               BCS             loc_1C8CC
.C7FC 000               AND.W           R0, R0, #0xFE
.C800 000               CMP             R0, #66
.C802 000               BEQ             loc_1C8AA
.C804 000               B               loc_1C92A
```

这里是申请寄存器空间，将参数解析出来，最后调用sub_2ABC8。

经过分析sub_37652函数为重点。

```
; FUNCTION CHUNK AT .text&.ARM.extab:0003880C SIZE 00000034 BYT
; FUNCTION CHUNK AT .text&.ARM.extab:000388B6 SIZE 0000008A BYT
; FUNCTION CHUNK AT .text&.ARM.extab:00038964 SIZE 00000030 BYT

                LDR.W           R0, [R9] ; 多次经过
                LDRB.W          R1, [R9,#4]
                SUBS            R2, R0, R6
                LDRH            R0, [R0]
                ORR.W           R1, R1, R1,LSL#8
                ASRS            R7, R2, #1
                EOR.W           R8, R1, R0
                MOV             R0, R9
                UXTH.W          R11, R8
                MOV             R1, R11
                BL              sub_389F2 ; 解密opcode
                MOV             R10, R0
                CMP             R0, #127
                BGE             loc_376D0
                CMP.W           R10, #64
```

此处的描述信息为

54 B2 E5 A1 04 00 00 00 00 5D B1 AE C1 3B

54 B2 E5 A1 指向指令

04 00 00 00 这个字段还不知道代表什么

00 5D B1 AE 执行dex

C1 3B method_id

C1 3B 这个是我的apk的oncreate方法的method_id

现在内存窗口是这样的

```
54 B2 E5 A1 04 00 00 00  00 5D B1 AE C1 3B 00 00
```

下一条指令

```
58 B2 E5 A1 04 00 00 00  00 5D B1 AE C1 3B 00 00
```

58-54=4

4即为上一条指令的长度。

后面就是根据op来对应指令执行了。因为是自己写的apk，oncreate方法的smali指令都知道，所以分析指令映射表就比较容易点，到这里，我们就可以对应出指令映射表来了。

到此为止，我对这个数字壳的加固就分析完了。有感兴趣的大佬可以分析一下指令映射表。

最后，再奉上附件。

【公告】看雪·众安 2021 KCTF秋季赛 【最受欢迎战队奖】评选开始！

最后于 2020-6-11 20:08 被[軍]编辑，原因:

上传的附件:

360.zip（9.82MB，340次下载）

收藏 · 55　点赞 · 11　打赏　分享

最新回复 (30)

1 2 ▶

endlif 1 2020-6-11 23:14　2 楼　0

感谢分享

最新回复 (30)

极客 **哆啦噩梦** 5 2020-6-12 08:09 3 楼 0 •••

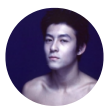
感谢分享，jumpout咋对抗的呢

极客 **hixhi** 6 2020-6-12 11:36 4 楼 0 •••

楼主这功力，简直了，佩服。

极客 **战马** 6 2020-6-12 15:10 5 楼 0 •••

这是免费版的吗



临时 **感谢你曾来过** 0 2020-6-12 15:58 6 楼 0 •••

这才是我遇到的加固so，之前那些大佬分析的 我都找不到样本

大侠 **[重]** 8 1 2020-6-12 17:20 7 楼 0 •••

是的 免费版的

大侠 **[重]** 8 1 2020-6-12 17:21 8 楼 0 •••

> 哆啦噩梦 感谢分享，jumpout咋对抗的呢

暂时还没有找到好的对抗方法

极客 **哆啦噩梦** 5 2020-6-12 18:10 9 楼 0 •••

> [重] 暂时还没有找到好的对抗方法

楼主jumpout咋处理的

大侠 **[重]** 8 1 2020-6-12 19:06 10 楼 0 •••

> 哆啦噩梦 楼主jumpout咋处理的

可以patch处理 但是对于参数的识别不是太友好

极客 **caijunfan** 3 2020-7-1 19:15 11 楼 0 •••

老师，请教一下。
“R7寄存器地址即为整个第二个so的开始地址，我们可以把整个malloc的区域全dump下来。”这个需要在哪条指令处dump?

大侠 **[重]** 8 1 2020-7-2 10:59 12 楼 0 •••

> caijunfan 老师，请教一下。“R7寄存器地址即为整个第二个so的开始地址，我们可以把整个malloc的区域全dump下来。”这个需要在哪条指令处dump?

把整个malloc区域全dump下来就可以，然后进行so修复

最新回复 (30)

caijunfan 2020-7-2 17:34 13 楼 0 ...

极客

> [重] 把整个malloc区域全dump下来就可以，然后进行so修复

是把四次malloc的区域dump下来吗~

[重] 1 2020-7-3 18:48 14 楼 0 ...

大侠

> caijunfan 是把四次malloc的区域dump下来吗~[em_85]

那四个区域肯定要dump的，这四个区域是so文件中加密的四部分 等它解密完后dump下来 用来修复so文件 除此之外还要dump整个so文件，就是截图中这部分 AEEC0000 AEFC0000 这是完整的so文件，只不过里面其中的四部分加密了，当然，除了这四部分之外，还有别的很多部分。

yezheyu 2020-7-5 11:13 15 楼 0 ...

极客

感谢分享

初音未来__ 2020-7-7 20:07 16 楼 0 ...

极客

感谢大佬分享教程

caijunfan 2020-7-16 17:18 17 楼 0 ...

极客

> [重] 那四个区域肯定要dump的，这四个区域是so文件中加密的四部分 等它解密完后dump下来 用来修复so文件 除此之外还要dump整个so文件，就是截图中这部分 AEEC0000 AEFC0000 ...

感谢大佬~我分析的可能版本不太一样 有差别

mb_jgzwnuxx 2020-7-17 15:23 18 楼 1 ...

临时

感谢分享

mb_evshufvg 2020-8-2 21:20 19 楼 0 ...

临时

跟大佬请教一下，整个malloc区域全dump下来文件大小不到100K，比你提供的的dump.so 765k文件小很多,是什么原因

[重] 1 2020-8-3 11:51 20 楼 0 ...

大侠

> mb_evshufvg 跟大佬请教一下，整个malloc区域全dump下来文件大小不到100K，比你提供的的dump.so 765k文件小很多,是什么原因

可能是dump错了地方 也可能是没有dump下来的原因

mb_evshufvg 2020-8-4 09:08 21 楼 0 ...

临时

谢谢，我忘了按照十六进制转换了

最新回复 (30)

1 2 ▶

mb_evshufvg 2020-8-4 21:20 22 楼 0 ...

临时

再跟大佬请教一下，我如何在加载第二个so里面下断点了，这个断点对应的地址怎么确定

[重] 1 2020-8-5 11:43 23 楼 0 ...

大侠

> mb_evshufvg 再跟大佬请教一下，我如何在加载第二个so里面下断点了，这个断点对应的地址怎么确定

上面提到过，从case35可以进入第二个so

mb_evshufvg 2020-8-5 17:15 24 楼 0 ...

临时

请教一下，从case35的.engine:0000CEF4 BLX LR，进入，加载的第二个so在模块列表中是什么名称，怎么找到调试中的sub_1CE60的断点位置，如何确定第二个so的调试基地址了

mb_evshufvg 2020-8-5 18:26 25 楼 0 ...

临时

已经找到了，大佬不用答复了，谢谢

wx_8954哈哈

内容

回帖 表情 高级回复

1 2 ▶

返回