



[原创]android so文件攻防实战-libDexHelper.so反混淆

houjingyi 9 10

专家

举报

17小时前

718

计划是写一个android中so文件反混淆的系列文章，目前这是第三篇。

第一篇：[android so文件攻防实战-百度加固免费版libbaiduprotect.so反混淆](#)

第二篇：[android so文件攻防实战-某团libmtguard.so反混淆](#)

今天分析的是企业版64位，我用LibChecker查了一下手机上的APP找到的，时间也还比较新。根据其他人的分析可知，libDexHelper.so是指令抽取的实现，libdexjni.so是VMP的实现。

去除混淆

首先因为加密过，肯定是不能直接反编译的，可以在libart.so下断点，进入JNI_OnLoad以后就可以dump下来。

```
art::LogMessage::LogMessage(v125, "art/runtime/java_vm_ext.cc", 856LL, 3LL, 0xFFFFFFFFLL);
v105 = art::LogMessage::stream((art::LogMessage *)v125);
__strlen_chk("[Calling JNI_OnLoad in \"", 25LL);
v106 = std::__put_character_sequence<char, std::char_traits<char>>(v105, (int)"[Calling JNI_OnLoad in \"");
if ( (*a3 & 1) != 0 )
    v107 = *((_QWORD *)a3 + 2);
else
    LODWORD(v107) = (_DWORD)a3 + 1;
v108 = std::__put_character_sequence<char, std::char_traits<char>>(v106, v107);
__strlen_chk("]", 3LL);
std::__put_character_sequence<char, std::char_traits<char>>(v108, (int)"]");
art::LogMessage::~LogMessage((art::LogMessage *)v125);
}
v67 = Symbol(v123, 0LL);
v68 = *((_DWORD *)((*(_QWORD *)v123 + 8) + 1192LL));
if ( v68 && v68 <= 21 )
    art::FaultManager::EnsureArtActionInFrontOfSignalChain((art::FaultManager *)&art::fault_manager);
art::Thread::SetClassLoaderOverride(v121, v66);
if ( v67 == -1 )
{
    v74 = "JNI_ERR returned from JNI_OnLoad in \"%s\"";
    v75 = v120;
    if ( (*a3 & 1) != 0 )
        v76 = (unsigned int)v107 + 1;
}
```

不过此时也不能直接F5，还存在以下混淆方式：

1.垃圾指令

LOAD:000000000006AC2C	;	-----
LOAD:000000000006AC2C		
LOAD:000000000006AC2C	loc_6AC2C	; CODE XREF: LOAD:000000000006A488↑j
LOAD:000000000006AC2C	LDP	Q21, Q9, [X25],#0x250
LOAD:000000000006AC2C	;	-----
LOAD:000000000006AC30	DCB	0xA4
LOAD:000000000006AC31	DCB	0xF0, 0x35, 0xD7
LOAD:000000000006AC34	;	-----
LOAD:000000000006AC34	UBFX	X14, X0, #0x18, #3
LOAD:000000000006AC38	ADRP	X7, #0xFFFFFFFF8132E000
LOAD:000000000006AC3C	STR	X4, [X13,#0x3BD0]
LOAD:000000000006AC3C	;	-----
LOAD:000000000006AC40	DCB	0xAD
LOAD:000000000006AC41	DCB	0x23, 0xD, 0x74, 0x7C, 0x90, 0x74, 0xAA
LOAD:000000000006AC48	DCQ	0x55F1220B965C5AC7, 0xB6972E6917B73799, 0xE9061EAC47231D7E
LOAD:000000000006AC60	DCB	0xFA, 0x2A, 0x8A, 0xC2
LOAD:000000000006AC64	;	-----
LOAD:000000000006AC64	B	loc_6A48C
LOAD:000000000006AC68	;	-----

这些垃圾指令是在switch的一个永远不会被执行到的分支里面，可以直接将IDA不能MakeCode的地方patch成NOP再MakeCode。

2.字符串加密

有好几个解密字符串的函数，0x186C4，0x7783C，0x95B9C。在[android so文件攻防实战-百度加固免费版libbaiduprotect.so反混淆](#)中我们是交叉引用拿到加密后的字符串和它对应的解密函数的表然后frida主动调用得到的解密后的字符串，但是在这里这个方法就不太好用了。因为这里加密后的字符串是在栈上一个byte一个byte拼起来的，和最后调用解密函数之间可能隔了很多条指令，甚至都不在一个block。

我最后用的是下面这种方案：以0x40110处调用0x186C4处的解密函数为例，这里面字符串解密的逻辑

[illegible]

总共有几百处调用，不可能全部人工去这样解出来，我写了另外以一个脚本去调用decstr.py。首先通过交叉引用找到所有调用解密函数的地方，然后把起始地址设为该block的起始地址，结束地址设为调用解密函数的地址，通过unicorn跑出decstr.py需要的三个参数之后调用decstr.py。遇到

unicorn.unicorn.UcError也有两个处理策略，一个是跳过该地址(loop_call_prepare_arg1)，起始地址不变；一个是将起始地址设为下一条地址(loop_call_prepare_arg2)。当然这套方案还有优化的空间，比如生成调用解密函数需要的参数的代码和最后调用解密函数的代码不在一个block，就处理不了。

```
1  import unicorn
2  import binascii
3  import threading
4  import subprocess
5
6  from capstone import *
7  from capstone.arm64 import *
8
9  inscnt = 0
10 start_addr = 0
11 end_addr = 0
12 stop_addr = 0
13 stop_addr_list = []
14
15 def hook_code(uc, address, size, user_data):
16     global inscnt
17     global end_addr
18     global stop_addr
19     global stop_addr_list
20
21     md = Cs(CS_ARCH_ARM64, CS_MODE_ARM)
22
23     for ins in md.disasm(sodata[address:address + size], address):
24         #rint(">>> 0x%x:\t%s\t%s" % (ins.address, ins.mnemonic, ins.op_str))
25         stop_addr = ins.address
26
27         if ins.address in stop_addr_list:
28             #print("will pass 0x%x:\t%s\t%s" %(ins.address, ins.mnemonic, ins.op_str))
29             uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_PC, address + size)
30             return
31
32         inscnt = inscnt + 1
33         if (inscnt > 500):
34             uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_PC, 0xffffffff)
35             return
36
37         if ins.mnemonic.find("b.") != -1:
38             print("will pass 0x%x:\t%s\t%s" %(ins.address, ins.mnemonic, ins.op_str))
39             uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_PC, address + size)
40             return
41
42         if ins.mnemonic.find("bl") != -1:
43             print("will pass 0x%x:\t%s\t%s" %(ins.address, ins.mnemonic, ins.op_str))
44             uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_PC, address + size)
45             return
46
47         if ins.op_str in ["x0", "x1", "x2", "x3"]:
48             X1 = uc.reg_read(unicorn.arm64_const.UC_ARM64_REG_X1)
49             if X1 > 0x105A88:
50                 print("will pass 0x%x:\t%s\t%s" %(ins.address, ins.mnemonic, ins.op_str))
51                 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_PC, address + size)
52                 return
53         if ins.op_str.startswith("#0x"):
54             addr = int(ins.op_str[3:],16)
55             if (addr > 0x14E50 and addr < 0x15820) \
56                 or addr == 0x186C4 \
57                 or addr > 0x105A88:
58                 print("will pass 0x%x:\t%s\t%s" %(ins.address, ins.mnemonic, ins.op_str))
59                 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_PC, address + size)
60                 return
61
62 def call_prepare_arg():
63     global inscnt
64     global start_addr
65     global end_addr
66     global stop_addr
67     global stop_addr_list
68
69     inscnt = 0
70
71     uc = unicorn.Uc(unicorn.UC_ARCH_ARM64, unicorn.UC_MODE_ARM)
72     code_addr = 0x0
73     code_size = 8*0x1000*0x1000
74     uc.mem_map(code_addr, code_size)
75     stack_addr = code_addr + code_size
76     stack_size = 0x1000000
77     stack_top = stack_addr + stack_size - 0x8
78     uc.mem_map(stack_addr, stack_size)
79     uc.hook_add(unicorn.UC_HOOK_CODE, hook_code)
80
81     uc.mem_write(code_addr, sodata)
```

☆

12

👍

4

¥



12



4



```
84 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X26, stack_addr)
85 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X25, stack_addr)
86 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X24, stack_addr)
87 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X23, stack_addr)
88 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X22, stack_addr)
89 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X21, stack_addr)
90 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X20, stack_addr)
91 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X19, stack_addr)
92 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X18, stack_addr)
93 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X17, stack_addr)
94 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X16, stack_addr)
95 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X15, stack_addr)
96 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X14, stack_addr)
97 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X13, stack_addr)
98 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X12, stack_addr)
99 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X11, stack_addr)
100 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X10, stack_addr)
101 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X9, stack_addr)
102 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X8, stack_addr)
103 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X7, stack_addr)
104 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X6, stack_addr)
105 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X5, stack_addr)
106 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X4, stack_addr)
107 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X3, stack_addr)
108 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X2, stack_addr)
109 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X1, stack_addr)
110 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_X0, stack_addr)
111
112 uc.reg_write(unicorn.arm64_const.UC_ARM64_REG_SP, stack_top)
113 uc.emu_start(start_addr, end_addr)
114
115 X0 = uc.reg_read(unicorn.arm64_const.UC_ARM64_REG_X0)
116 decstr = uc.mem_read(X0, 80)
117 end_index = decstr.find(bytearray(b'\x00'), 1)
118 decstr = decstr[:end_index]
119
120 decstr = binascii.b2a_hex(decstr)
121 decstr = decstr.decode('utf-8')
122
123 X1 = uc.reg_read(unicorn.arm64_const.UC_ARM64_REG_X1)
124 X2 = uc.reg_read(unicorn.arm64_const.UC_ARM64_REG_X2)
125
126 pi = subprocess.Popen(['C:\\Python38\\python.exe', 'decstr.py', decstr, hex(X1), hex(X2)], st
127 output = pi.stdout.read()
128 print(output)
129
130 def loop_call_prepare_arg1():
131     global inscnt
132     global end_addr
133     global stop_addr
134     global stop_addr_list
135
136     loopcnt = 0
137     stop_addr_list = []
138
139     while True:
140         try:
141             loopcnt = loopcnt + 1
142             if(loopcnt > 200):
143                 break
144             call_prepare_arg()
145         except unicorn.unicorn.UcError:
146             print("adding...")
147             print(hex(stop_addr))
148             stop_addr_list.append(stop_addr)
149         else:
150             break
151
152 def loop_call_prepare_arg2():
153     global inscnt
154     global end_addr
155     global stop_addr
156     global stop_addr_list
157
158     global start_addr
159
160     loopcnt = 0
161     stop_addr_list = []
162
163     while True:
164         try:
165             loopcnt = loopcnt + 1
166             if(loopcnt > 200):
167                 break
168             call_prepare_arg()
169         except unicorn.unicorn.UcError:
170             start_addr = stop_addr + 4
```


☆

12

👍

4

¥

```
173
174 with open("C:\\Users\\hjy\\Downloads\\out1.fix.so", "rb") as f:
175     sodata = f.read()
176
177 all_addr = []
178 with open('xref_decstr.txt', 'r', encoding='utf-8') as f:
179     for line in f:
180         addr = "0x" + line[2:]
181         addr = int(addr, 16)
182         all_addr.append(addr)
183
184 for i in all_addr:
185
186     print("i:")
187     print(hex(i))
188
189     end_addr = i
190     CODE = sodata[i - 4:i]
191     md = Cs(CS_ARCH_ARM64, CS_MODE_ARM)
192     for x in md.disasm(CODE, i - 4):
193         mnemonic = x.mnemonic
194
195     while mnemonic != "ret" \
196           and mnemonic != "b" \
197           and mnemonic != "br" \
198           and mnemonic != "cbz" \
199           and mnemonic != "cbnz":
200         i = i - 4
201         CODE = sodata[i - 4:i]
202         for x in md.disasm(CODE, i - 4):
203             mnemonic = x.mnemonic
204
205     start_addr = i
206
207     print("start_addr:")
208     print(hex(start_addr))
209     print("end_addr:")
210     print(hex(end_addr))
211
212     loop_call_prepare_arg1()
213     loop_call_prepare_arg2()
214
```

更恶心的是还有很多字符串是自己在函数内解密的，这种情况我也没想到有什么好的方法。

3.控制流混淆

第一种是把正常顺序执行的指令打乱成switch的形式，这个影响倒不是太大：

```
v10 = 0;
v11 = 0;
v12 = 3;
while ( 2 )
{
    switch ( v12 )
    {
        case 0:
            if ( v10 >= 7 )
                v12 = 2;
            else
                v12 = 4;
            continue;
        case 1:
            ++v10;
            v12 = 0;
            continue;
        case 2:
            *(&v18 + v10) = 0;
            v3 = 0LL;
    }
}
```

第二种是动态计算跳转地址，基本上类似于在[android so文件攻防实战-某团libmtguard.so反混淆](#)见过的那种，但是要更复杂。

☆

12

👍

4

¥

```
-----
LOAD:000000000001D9D0      STRB      W2, [X19,#3]
LOAD:000000000001D9D4      MOV       W0, #4
LOAD:000000000001D9D8      STRB      W2, [X19,#4]
LOAD:000000000001D9DC      MOV       W8, #5
LOAD:000000000001D9E0      STRB      W4, [X19,#5]
LOAD:000000000001D9E4      MOV       W7, #0xFFFFFFFF
LOAD:000000000001D9E8      STRB      W2, [X19,#7]
LOAD:000000000001D9EC      STRB      W6, [X19,#0xA]
LOAD:000000000001D9F0      STRB      W5, [X19,#0xB]
LOAD:000000000001D9F4      STRB      W2, [X19,#0xC]
LOAD:000000000001D9F8      STRB      W2, [X19,#0xD]
LOAD:000000000001D9FC      STRB      W4, [X19,#0xE]
LOAD:000000000001DA00      STRB      W2, [X19,#0xF]
LOAD:000000000001DA04      loc_1DA04      ; CODE XREF: sub_1D518+4F0↓j
LOAD:000000000001DA04      loc_1DA04      ; sub_1D518+96C↓j
LOAD:000000000001DA04      CMP       W0, #5
LOAD:000000000001DA08      B.HI      loc_1DA04
LOAD:000000000001DA0C      loc_1DA0C      ; CODE XREF: sub_1D518+960↓j
LOAD:000000000001DA0C      loc_1DA0C      ; sub_1D518+978↓j ...
LOAD:000000000001DA0C      LDR       X2, [X29,#0x190+var_118]
LOAD:000000000001DA10      LDRH      W0, [X2,W0,UXTW#1]
LOAD:000000000001DA14      ADR       X2, loc_1DA20 ; jumtable 000000000001D91C case 2
LOAD:000000000001DA18      ADD       X0, X2, W0,SXTH#2
LOAD:000000000001DA1C      BR        X0
LOAD:000000000001DA20      -----
-----
```

比如这里的指令，在0x1DA0C处给X2赋值，X2此时为.data段中的一个地址，W0为偏移，取出值后在0x1DA18处乘4加上0x1DA20，最后的值就是0x1DA1C处X0的值。那么需要解决这么几个问题：

如何确定0x1DA0C处给X2赋的值

将0x1DA00处的指令改成跳转指令，0x1DA00这个地址又该如何确定

找到所有会跳转到0x1DA1C的指令，将跳转地址改成计算出来的X0的值

第一个问题，其实和字符串解密面临的情况是类似的，比如这里需要找到"LDR XX, [X29,#0x190+var_118]"这条指令，然后再找给XX寄存器赋值的指令，然而这两条指令很可能和BR X0隔了好几个block。我的解决方法是通过IDA提供的idaapi.FlowChar功能，递归前面的block，找到需要的指令。不足之处在于前提条件是IDA正确识别了函数的起始地址，否则会出现我们需要的指令和BR X0不在同一个函数的情况，这样就处理不了。

第二个问题，在递归前面的block的时候就先找到0x1D9D4处这条给W0赋值的指令，然后从0x1D9D4处开始直到0x1DA1C，找到第一个存在交叉引用的地址，也就是0x1DA04。它的前一条指令0x1DA00就是需要改成跳转指令的地方。

第三个问题，确定了0x1DA00之后，那么从0x1DA00到0x1DA1C所有存在交叉引用的地址都要去交叉引用的地方修改跳转地址。不过这里有很多细节。

(1)如果W0是由CSEL, CSET, CSINC这些指令赋值的，像下面这种情况，那么需要把0x1DE80和0x1DE84修改成 B.GE和B.LT。

patch前：

```
LOAD:000000000001DE7C      CMP       W1, #0xE
LOAD:000000000001DE80      CSEL      W0, WZR, W8, LT
LOAD:000000000001DE84      B         loc_1DA04
```

patch后：

```
LOAD:000000000001DE7C      ; sub_1D518+99C↓j
LOAD:000000000001DE7C      CMP       W1, #0xE
LOAD:000000000001DE80      B.GE      loc_1DBDC
LOAD:000000000001DE84      B.LT      loc_1DE60
```

(2)0x1DE80处的CSEL W0, WZR, W8, LT，这里W8的值是在0x1D9DC MOV W8, #5赋值的，所以我的代码中有一个register_value_dict，在改掉0x1DA00处的指令之后会读取0x1DA00所在的block到0x1DA1C所在的block的所有指令，找到给寄存器赋值的指令然后把值存起来。

```
register_list = ["W0","W1","W2","W3","W4","W5","W6","W7","W8","W9","W10",\
               "W11","W12","W13","W14","W15","W16","W17","W18","W19","W20", \
               "W21","W22","W23","W24","W25","W26","W27","W28","W29","W30"]

register_value_dict = {'W0': -1, 'W1': -1, 'W2': -1, 'W3': -1, 'W4': -1, 'W5': -1, 'W6': -1, 'W7': -1, 'W8': -1, 'W9': -1, 'W10': -1, \
                      'W11': -1, 'W12': -1, 'W13': -1, 'W14': -1, 'W15': -1, 'W16': -1, 'W17': -1, 'W18': -1, 'W19': -1, 'W20': -1, \
                      'W21': -1, 'W22': -1, 'W23': -1, 'W24': -1, 'W25': -1, 'W26': -1, 'W27': -1, 'W28': -1, 'W29': -1, 'W30': -1}
```

☆

12

👍

4

¥

```
target_blocks = []
flowchart = idaapi.FlowChart(idaapi.get_func(patch_addr))
for block in flowchart:
    if (block.start_ea <= patch_addr and patch_addr <= block.end_ea) \
    or (block.start_ea > patch_addr and block.end_ea <= brX0_addr):
        target_blocks.append(block)

if target_blocks != []:
    for target_block in target_blocks:
        print("---")
        print(hex(target_block.start_ea))
        print(hex(target_block.end_ea))
        print("---")
        i = target_block.start_ea
        while i < target_block.end_ea:
            if idc.print_insn_mnem(i) == "MOV" \
            and idc.print_operand(i, 0) in register_list \
            and idc.print_operand(i, 1).startswith("#"):
                value = int(idc.print_operand(i, 1)[1:], 16)
                if value < 100:
                    register_value_dict[idc.print_operand(i, 0)] = value
            i = i + 4
```

(3)有些地方还会有一条sub指令，这个也要考虑进去，比如下面这种情况0x33394处跳转的地址就应该按照W8为4计算。

LOAD:00000000000033388	BLR	X2
LOAD:0000000000003338C	STR	X0, [X29,#0x1B0]
LOAD:00000000000033390	MOV	W8, #5
LOAD:00000000000033394	B	loc_3152C

LOAD:0000000000003152C	STR	X0, [X29,#0x1B0+var_10]
LOAD:0000000000003152C		; CODE XREF: sub_31648+14↓j
LOAD:0000000000003152C		; sub_31660+5C↓j ...
LOAD:0000000000003152C	SUB	W8, W8, #1
LOAD:00000000000031530	CMP	W8, #6
LOAD:00000000000031534	B.LS	loc_31570
LOAD:00000000000031538		; CODE XREF: sub_31474+F8↓j
LOAD:00000000000031538	LDR	X0, [X29,#0xA30+file] ; file
LOAD:0000000000003153C	MOV	W2, #0x1B6
LOAD:00000000000031540	MOV	W1, #0xC2 ; oflag
LOAD:00000000000031544	BL	.open
LOAD:00000000000031548	MOV	W19, W0
LOAD:0000000000003154C	LDR	X1, [X29,#0xA30+buf] ; buf
LOAD:00000000000031550	MOV	X2, #4 ; n
LOAD:00000000000031554	BL	.write
LOAD:00000000000031558	MOV	W0, W19 ; fd
LOAD:0000000000003155C	BL	.close
LOAD:00000000000031560	MOV	W8, #0
LOAD:00000000000031568	CMP	W8, #6
LOAD:0000000000003156C	B.HI	loc_31538
LOAD:00000000000031570		
LOAD:00000000000031570		; CODE XREF: sub_31474+C0↑j
LOAD:00000000000031570	LDR	X0, [X29,#0xA30+var_920]
LOAD:00000000000031574	LDRH	W0, [X0,W8,UXTH#1]
LOAD:00000000000031578	ADR	X1, sub_31584
LOAD:0000000000003157C	ADD	X0, X1, W0, SXTH#2
LOAD:00000000000031580	RR	X0

最后的脚本放附件了。当然还有一些脚本处理不了的地方，不过问题已经不算太大了，需要的话可以动态调试确定。

4.函数地址动态计算

☆

12

👍

4

¥

```
277     v44 = *v43 == 49;
278 LABEL_27:
279     if ( v0 )
280     {
281         v34 = sub0(&loc_1E78C, 0x26C);
282         v34();
283         v35 = sub0x17D(off_12EB80[0] + 0x17D);
284         v36 = v35(v0, v44);
285         (path_init)(v36);
286     }
287     return *off_12ECD8;
288 }
```

LOAD:0000000000018828 ; __int64 __fastcall sub0(__int64, int)

LOAD:0000000000018828 sub0 ; CODE XREF: pEAF293EB881CF172A405994F2B8240C8(void)+30C↑p

LOAD:0000000000018828 ; pEAF293EB881CF172A405994F2B8240C8(void)+AA0↑p ...

LOAD:0000000000018828 SUB X0, X0, W1, SXTW

LOAD:000000000001882C RET

LOAD:000000000001882C ; End of function sub0

LOAD:000000000001882C

LOAD:0000000000018830

AD:0000000000012EB78 ; DATA XREF: hook_libart_lib

AD:0000000000012EB78 ; hook_libart_libdexfile+45C

AD:0000000000012EB80 off_12EB80 DCQ p329AAB59961F6410ABA963EF972FE303

AD:0000000000012EB80 ; DATA XREF: env_detect_1+AA

AD:0000000000012EB80 ; env_detect_1+AAC↑p

这个在IDA里面是能看清楚的，v35其实就是off_12EB80[0]，即调用0x80FE0处的p329AAB59961F6410ABA963EF972FE303。

接下来我们就来分析libDexHelper.so，来看看它都干了些什么。精力有限，很多地方没能很详细去分析。有些地方分析的可能也不一定对，将就看吧。

功能分析

JNI_OnLoad(0x3EA68)的分析在最后。

0x15960

读/proc/self/maps，特征字符串：

```
1 libDexHelper.so
2 libDexHelper-x86.so
3 libDexHelper-x86_64.so
4 /system/lib64/libart.so
5 /system/lib64/libLLVM.so
6 /system/framework/arm64/boot-framework.oat
7 /system/lib64/libskia.so
8 /system/lib64/libhwui.so
9 .oat
10 ff c3 01 d1 f3 03 04 aa f4 03 02 aa f5 03 01 aa e8 03 00 aa
11 GumInvocationListener
12 GSocketListenerEvent
```

0x16A30

获取系统属性，读/proc/%d/cmdline，特征字符串：

```
1 ro.yunos.version
2 ro.yunos.version.release
3 persist.sys.dalvik.vm.lib
4 persist.sys.dalvik.vm.lib.2
5 /system/bin/dex2oat
6 LD_OPT_PACKAGENAME
7 LD_OPT_ENFORCE_V1
```

off_12EF10:为2表示yunos，art模式；为1表示yunos，dalvik模式；为0表示非yunos。

0x17A70

md5。

0x186C4

字符串解密函数。

0x19674



0x19778

返回字符串su。

0x1987C

返回字符串mount。

0x19998

写classes.dve文件。

0x19b48

读取目录中的文件。

0x19E08

创建String类型的数组，第一个参数是String列表，第二个参数是数组长度。

0x1A058

调用0x19E08创建数组：

```
1 | /etc
2 | /sbin
3 | /system
4 | /system/bin
5 | /vendor/bin
6 | /system/sbin
7 | /system/xbin
```

0x1A740

调用0x19E08创建数组：

```
1 | com.yellowes.su
2 | eu.chainfire.supersu
3 | com.noshufou.android.su
4 | com.thirdparty.superuser
5 | com.koushikdutta.superuser
6 | com.noshufou.android.su.elite
```

0x1AF1C

调用0x19E08创建数组：

```
1 | com.chelpus.lackypatch
2 | com.ramdroid.appquarantine
3 | com.koushikdutta.rommanager
4 | com.dimonvideo.luckypatcher
5 | com.ramdroid.appquarantinepro
6 | com.koushikdutta.rommanager.license
```

0x1B7D0

调用0x19E08创建数组：

```
1 | com.saurik.substrate
2 | com.formyhm.hideroot
3 | com.amphoras.hidemyroot
4 | com.devadvance.rootcloak
5 | com.formyhm.hiderootPremium
6 | com.devadvance.rootcloakplus
7 | com.amphoras.hidemyrootadfree
8 | com.zachspng.temprootremovejb
9 | de.robv.android.xposed.installer
```

0x1C40C

system_property_get ro.product.cpu.abi和读/system/lib/libc.so判断是不是x86架构。

0x1C61C

查看classes.dve是否存在。

0x1C8D8

调用0x19E08创建数组：





```
1  | /sbin/
2  | /su/bin/
3  | /data/local/
4  | /system/bin/
5  | /system/xbin/
6  | /data/local/bin/
7  | /system/sd/xbin/
8  | /data/local/xbin/
9  | /system/bin/.ext/
10 | /system/bin/failsafe/
11 | /system/usr/we-need-root/
```

0x1D518

初始化一些路径，特征字符串：

```
1  | .cache
2  | oat
3  | .payload
4  | v1filter.jar
5  | classes.odex
6  | classes.vdex
7  | classes.dex
8  | assets/classes.jar
9  | .cache/classes.jar
10 | .cache/classes.dex
11 | .cache/classes.odex
12 | .cache/classes.vdex
```

0x1E520

将libc中的一些函数的地址放到.DATA。

```
1  | 0x137BB0 fopen
2  | 0x137BB8 fclose
3  | 0x137BC0 fgets
4  | 0x137BC8 fwrite
5  | 0x137BD0 fread
6  | 0x137BD8 sprintf
7  | 0x137BE0 pthread_create
```

0x1F250

读/proc/self/cmdline，判断是否含有com.miui.packageinstaller从而判断是否由小米应用包管理组件启动。

0x1F710

先system_property_get ro.product.manufacturer和system_property_get ro.product.model判断是否是samsung，然后system_property_get ro.build.characteristics是否为emulator。

0x1FDC8

注册如下native函数：

```
1  | RegisterNative(com/secneo/apkwrapper/H, attach(Landroid/app/Application;Landroid/content/Context;)
2  | RegisterNative(com/secneo/apkwrapper/H, b(Landroid/content/Context;Landroid/app/Application;)V, RX
3  | RegisterNative(com/secneo/apkwrapper/H, c()V, RX@0x40024c08[libDexHelper.so]0x24c08)
4  | RegisterNative(com/secneo/apkwrapper/H, d(Ljava/lang/String;)Ljava/lang/String;, RX@0x40023d04[lib
5  | RegisterNative(com/secneo/apkwrapper/H, e(Ljava/lang/Object;Ljava/util/List;Ljava/lang/String;)[Lj
6  | RX@0x40035ab0[libDexHelper.so]0x35ab0)
7  | RegisterNative(com/secneo/apkwrapper/H, f()[Ljava/lang/String;, RX@0x4001a740[libDexHelper.so]0x1a
8  | RegisterNative(com/secneo/apkwrapper/H, g()[Ljava/lang/String;, RX@0x4001af1c[libDexHelper.so]0x1a
9  | RegisterNative(com/secneo/apkwrapper/H, h()[Ljava/lang/String;, RX@0x4001b7d0[libDexHelper.so]0x1b
10 | RegisterNative(com/secneo/apkwrapper/H, n()[Ljava/lang/String;, RX@0x4001c8d8[libDexHelper.so]0x1c
11 | RegisterNative(com/secneo/apkwrapper/H, j()[Ljava/lang/String;, RX@0x4001a058[libDexHelper.so]0x1a
12 | RegisterNative(com/secneo/apkwrapper/H, k()Ljava/lang/String;, RX@0x40019778[libDexHelper.so]0x197
13 | RegisterNative(com/secneo/apkwrapper/H, l()Ljava/lang/String;, RX@0x4001987c[libDexHelper.so]0x198
14 | RegisterNative(com/secneo/apkwrapper/H, m()Ljava/lang/String;, RX@0x40019674[libDexHelper.so]0x196
15 | RegisterNative(com/secneo/apkwrapper/H, bb(Landroid/content/Context;Landroid/app/Application;Landr
16 | RX@0x4002921c[libDexHelper.so]0x2921c)
17 | RegisterNative(com/secneo/apkwrapper/H, o(Landroid/content/Context;)I, RX@0x4002f158[libDexHelper.
18 | RegisterNative(com/secneo/apkwrapper/H, p()V, RX@0x4001875c[libDexHelper.so]0x1875c)
   | RegisterNative(com/secneo/apkwrapper/H, q()I, RX@0x40023568[libDexHelper.so]0x23568)
   | RegisterNative(com/secneo/apkwrapper/H, mu()I, RX@0x4001f250[libDexHelper.so]0x1f250)
```

0x218A8

system_property_get ro.build.version.release/ro.build.version.sdk/ro.build.version.codename，最终返回sdkversion。



首页



论坛



课程



招聘



发现

创建一些目录：

```
1 | /data/usr/0/包名/.cache/oat
2 | /data/usr/0/包名/.cache/oat/arm64
3 | /data/usr/0/包名/.payload
```

0x22a90

模拟器检测，特征字符串：

```
1 | vboxsf
2 | /mnt/shared/install_apk
3 | nemusf
4 | /mnt/shell/emulated/0/Music sharefolder
5 | /sdcard/windows/BstSharedFolder
```

0x23568

读proc/pid/cmdline找字符串":bbs"，没搞懂这是什么意思。这个函数名是is_magisk_check_process。

0x247C0

调用setOuterContext。

0x24C08

system_property_get ro.product.brand，针对华为/荣耀机型，调用startLoadFromDisk。

0x26278

getDeclaredFields获取field对象数组之后调用equals，返回查找的指定的field对象。

0x27290

修改mInitialApplication和mClassLoader。

0x2921C

修改mAllApplications(remove和add)。

0x29CE8

模拟器检测，特征字符串：

```
1 | com.bignox.app.store.hd
2 | com.bluestacks.appguidance
3 | com.bluestacks.settings
4 | com.bluestacks.home
5 | com.bluestack.BstCommandProcessor
6 | com.bluestacks.apmart
```

0x2B670

通过FLAG_DEBUGGABLE判断是debug还是release。

0x2CAE0

通过android.content.pm.Signature获取签名的md5。

0x2F158

通过access以下文件判断是否被root：

```
1 | /sbin/.magisk/
2 | /sbin/.core/img
3 | /sbin/.core/mirror
4 | /sbin/.core/db-0/magisk.db
```

0x2F6E4

读/proc/self/cmdline，调用java层的com.secneo.apkwrapper.H.j，调用bindService，获取android_id，调用android.app.Application.attach，如果包名是com.huawei.irportalapp.uat调用setOuterContext。

0x31474

调用java层的com.secneo.apkwrapper.H.f(ff)加载v1filter.jar。

查看/proc/self/maps:

0000000000000000-0000000000000000 r-xp 00000000 00:00000000 00000000 00000000



12



4



首页



论坛



课程



招聘



发现



12



4



hook libcutils.so/liblog.so中的android_log_write和android_log_buf_write，使其返回0。

0x339FC

currentActivityThread-mPackages-LoadedApk-mResources-getAssets。

0x34A00

调用android.content.res.Resources.getAssets，失败再调用0x339FC。

0x351DC

读取assets文件。

0x35AB0

对传入参数调用makeInMemoryDexElements，修改dalvik.system.DexFile.mFileName。

0x3766C

初始化下列字符串:

```
1 | /data/user/0/cn.missfresh.application/.cache/classes.jar
2 | /data/user/0/cn.missfresh.application/.cache/classes.dex
3 | /data/user/0/cn.missfresh.application/.cache/v1filter.jar
```

调用0x80458计算包名hash，调用0x75AA8

调用AAssetManager_open读取assets/resthird.data写入v1filter.jar，调用0x31474

(看别人的分析应该读assets下面两个文件：classes0.jar是被加密的dex，classes.dgc是被加密的抽取后的指令。不过我分析的这个样本中没有classes0.jar和classes.dgc，v1filter.jar看了一下也并不是原始dex，有点没太搞懂这个样本怎么加固的)

0x398F8/0x3A08C

检测dexhunter，dumpclass好像是dexhunter里面的吧。特征字符串：

```
1 | _Z16hprofDumpClassesP15hprof_context_t
2 | _Z12dvmDumpClassPK11ClassObjecti
3 | _Z9dumpClassP7DexFilei
4 | dumpclass
5 | dump_class
```

0x3BF10

参数是文件名，返回文件是否存在。

0x3BF7C

模拟器检测，特征字符串：

```
1 | ueventd.ttVM_x86.rc
2 | init.ttVM_x86.rc
3 | fstab.ttVM_x86
4 | bluestacks
5 | BlueStacks
```

0x3CE14

system_property_get ro.debuggable，调用检测模拟器的函数。

0x3D814

通过android.hardware.usb.action.USB_STATE监听USB状态。

0x42378

md5。

0x44708

hook下列函数(反调试):

```
1 | vmDebug::notifyDebuggerActivityStart(hook后: 0x446C0)
2 | art::Dbg::GoActive(hook后: 0x446E4)
3 | art::Runtime::AttachAgent(hook后: 0x45CF8)
```

0x46194



首页



论坛



课程



招聘



发现

0x4C2F0

hook下列函数(指令抽取还原):

```
1 | art::ClassLinker::DefineClass(hook后: 0x46BB8)
2 | art::ClassLinker::LoadMethod(hook后: 0x46ED4/0x47BB8/0x488C0/0x491F8/0x49B0C)
3 | art::OatFile::OatMethod::LinkMethod(hook后: 0x46BD8/0x46DB0)
```

0x4DB80

md5。

0x50280

读/proc/self/maps找到含有包名的段。

0x5074C

调用java层的com.secneo.apkwrapper.H1.find_dexfile。

0x50B60

调用java.lang.StackTraceElement.getMethodName和java.lang.StackTraceElement.getClassName。

0x57424

加载assets中的classes.dgg。

0x598FC

读/proc/self/maps找到libDexHelper.so。

0x59CE8

设置dex2oat的参数, --zip-fd/--oat-fd/--zip-location/--oat-location/--oat-file/--instruction-set。

0x5C600

hook libdvm.so中的函数(类似于0x67544), 具体没仔细看, 0x5BAA8-0x5BEF8都是被hook后的实现。

0x61E3C

hook libc中的下列函数:

```
1 | fstatat64(hook后: 0x5E778)
2 | stat(hook后: 0x5E858)
3 | close(hook后: 0x5EA20)
4 | openat(hook后: 0x5ED20)
5 | open(hook后: 0x5ED9C)
6 | pread(hook后: 0x5FAB8)
7 | read(hook后: 0x5FC14)
8 | mmap64(hook后: 0x5FDDC)
9 | __openat_2(hook后: 0x5FEF4)
10 | __open_2(hook后: 0x5FF74)
```

0x64AE8

根据不同SDK版本返回Name Mangling之后的art::DexFileLoader::open。

0x65FE4

根据不同SDK版本返回Name Mangling之后的art::OatFileManager::OpenDexFilesFromOat。

0x67544

hook下列函数:

```
1 | art::DexFileLoader::open(hook后: 0x6D39C/0x6D3E8)
2 | art::OatFileManager::OpenDexFilesFromOat(hook后: 0x6A2C0/0x6AF14/0x6B9B0/0x6C188/0x6CB5C)
```

0x6D4A0

patch掉art::Runtime::IsVerificationEnabled。

0x6DAD8

hook art::DexFileVerifier::Verify(hook后: 0x6D38C/0x6D394, 直接返回1)。

0x6E40C

hook art::DexFileLoader::open(hook后: 0x6D39C/0x6D3E8)。



12



4



hook下列函数：

```
1 | art::DexFileVerifier::Verify(hook后: 0x6EB04/0x6EB0C/0x6EB14, 直接返回1)
2 | art::DexFile::OpenMemory(hook后: 0x74EE8/0x74E90/0x74F38)
3 | Art::DexFile(hook后: 0x74E30/0x74F88)
```

0x75054

hook libdvm.so中的函数，具体没仔细看，0x6EB1C/0x74DEC/0x6FFBC都是被hook后的实现。

0x75AA8

读java.lang.DexCache.dexfile(这个dexfile就是解压apk之后根目录的那个classes.dex)。

0x767F8

参数是so文件路径，打开该so文件。

0x76C8C

参数是libart.so中的一个函数，返回该函数地址。

0x76CCC

第一个参数是so中的函数名，第二个参数是so的相对路径，返回该函数在so中的地址。

0x76D90

参数是libdexfile.so中的一个函数，返回该函数地址。

0x76DD4

参数是libjdwp.so中的一个函数，返回该函数地址。

0x76E18

md5。

0x7783C

字符串解密函数。

0x79270

计算传入字符串的hash(不完全是md5)。

0x7A240

热补丁检测，特征字符串：

```
1 | nuwa
2 | andfix
3 | hotfix
4 | .RiskStu
5 | tinker
```

0x80458

调用0x79270。

0x804A8

hook libc中的下列函数：

```
1 | msync(hook后: 0x78470)
2 | close(hook后: 0x7AF50)
3 | munmap(hook后: 0x7A568)
4 | openat64(hook后: 0x7DC48)
5 | __open_2(hook后: 0x7DC80)
6 | _open64(hook后: 0x7DCB8)
7 | _openat_2(hook后: 0x7DCF0)
8 | ftruncate64(hook后: 0x7DD30)
9 | mmap64(hook后: 0x7EF60)
10 | pread64(hook后: 0x7F5D0)
11 | read(hook后: 0x7F7DC)
12 | write(hook后: 0x8022C)
```

0x87F98

hook libdvm.so中的函数(类似于0x44708)，具体没仔细看，0x856C0/0x87F00/0x87F4C都是被hook后的实现





12



4



patch掉art::Runtime::UseJitCompilation。

0x8A794/0x8B71C/0x8B890

hook函数实现。

0x917E8

读/proc/sys/fs/inotify/max_queued_watches。

0x91848

读/proc/sys/fs/inotify/max_user_instances。

0x918A8

读/proc/sys/fs/inotify/max_user_watches。

0x95778

看起来好像是通过判断时间实现的反调试。

0x95A28

字符串查找函数。

0x95B9C

字符串解密函数。

0x95D60

socket连接。

0x96398

frida检测，读/proc/self/task，特征字符串：gum-js-loop；读/proc/self/fd，特征字符串linjector。

0x995D0

xposed检测，特征字符串：

```
1 | .xposed.  
2 | xposedbridge  
3 | xposed_art
```

0x99D28

hook框架检测，特征字符串：

```
1 | frida  
2 | ddi_hook  
3 | dexposed  
4 | substrate  
5 | adbi_hook  
6 | MSFindSymbol  
7 | hook_precall  
8 | hook_postcall  
9 | MSHookFunction  
10 | DexposedBridge  
11 | MSCloseFunction  
12 | dexstuff_loaddex  
13 | dexposedIsHooked  
14 | ALLINONEs_arthook  
15 | dexstuff_resolv_dvm  
16 | dexposedCallHandler  
17 | art_java_method_hook  
18 | artQuickToDispatcher  
19 | dexstuff_defineclass  
20 | dalvik_java_method_hook  
21 | art_quick_call_entrypoint  
22 | frida_agent_main
```

0x9C0BC

调用0x96398检测frida，system_property_get ro.product.model，调用0x9FD88检测xposed和自动脱壳机，hook dlopen(hook后：0x9B89C)和ptrace(hook后：0x95BF8)。

0x9CFCC

通过读取/proc/%d/status判断TracerPid等实现反调试。



首页



论坛



课程



招聘



发现



12



4



通过读取/proc/%d/wchan判断是不是ptrace_stop实现反调试。

0x9DCF4

通过读取/proc/%ld/task/%ld/status判断TracerPid等实现反调试。

0x9ED44

通过java.lang.StackTraceElement.getClassName打印函数调用栈进行xposed检测。

0x9F770

通过java.lang.ClassLoader.getSystemClassLoader.loadClass打印类加载器进行xposed检测。

0x9FD88

调用0x9ED44和0x9F770，通过判断ServiceManager里是否有user.xposed.system进行xposed检测，然后检测自动脱壳机：

fart(<https://github.com/hanbinglengyue/FART>)

FUPK3(<https://github.com/F8LEFT/FUPK3>)

Youpk(<https://github.com/Youlor/Youpk>)

检测方法是判断下列类或者方法是否存在：

```
1 dumpMethodCode
2 fartthread
3 fart
4 android/app/fupk3/Fupk
5 android/app/fupk3/Global
6 android/app/fupk3/UpkConfig
7 android/app/fupk3/FRefInvoke
8 cn/youlor/Unpacker
```

0xA18D4

getInstalledApplications获取系统中安装的APP信息。

0xA7D3C

解密出字符串Java和JNI_OnLoad，hook了几个函数，被hook的原地址未知，新地址：

0xA43A0/0xA485C/0xA48F4/0xA54B0；hook dlsym(hook后：0xA4554)和dlopen(hook后：0xA4D30)。

0xB4B94

hook libc中的下列函数：

```
1 write(hook后：0xAA2CC)
2 pwrite64(hook后：0xAA51C)
3 close(hook后：0xAA774)
4 read64(hook后：0xAAA9C)
5 openat64(hook后：0xAACB8)
6 __openat_2(hook后：0xAB6D4)
7 __open_2(hook后：0xAC0F4)
8 open64(hook后：0xACB10)
9 read(hook后：0xAFE18)
10 mmap64(hook后：0xB1C54)
```

system_property_get debug.atrace.tags.enableflags，hook bionic_trace_begin和bionic_trace_end(hook后：0xA8EF4和0xA8EF8，直接返回)，没有找到则hookg_trace_marker_fd(hook后：0xA8EFC，返回-1)。

0xB9BEC

sha1。

0xBAE64

md5init。

0xC3378

base64。

0xC5DDC

base64。



首页



论坛



课程



招聘



发现



APK签名相关。

0xD024C

sha1。

0xD1C04

sha1init。

0xD2E98

md5。

0xD6484

调用0xD75A0。

0xD5CDC

读/proc/self/cmdline。

0xD6578

hook libdvm.so中的函数(hook后： 0xD6988)。

0xD68A8

根据off_12EF10处的值判断调用0xD6484还是0xD6578。

0xD75A0

hook libaoc.so中的函数(hook后： 0xD69BC)。

0x3EA68

JNI_OnLoad。分析环境pixel4 android10，动态分析过程中一些没有被调用的函数不再分析。

- 1.初始化cpuabi字符串(arm64)于0x12E7C8
- 2.初始化so名字字符串(libDexHelper)于0x12EC38
- 3.初始化字符串com/secneo/apkwrapper/H于0x137B10
- 4.调用0x1E520
- 5.

```
1 | JNIEnv->FindClass(com/secneo/apkwrapper/H)
2 | JNIEnv->GetStaticFieldID(com/secneo/apkwrapper/H.PKGNAMELjava/lang/String;)
3 | JNIEnv->GetStaticObjectField(class com/secneo/apkwrapper/H, PKGNAME Ljava/lang/String; => "cn.missf
4 | JNIEnv->GetStringUtfChars("cn.missfresh.application")
```

6.将包名存于0x138040

7.

```
1 | JNIEnv->FindClass(android/app/ActivityThread)
2 | JNIEnv->GetStaticMethodID(android/app/ActivityThread.currentActivityThread()Landroid/app/ActivityT
3 | JNIEnv->CallStaticObjectMethodV(class android/app/ActivityThread, currentActivityThread())
4 | JNIEnv->GetMethodID(android/app/ActivityThread.getSystemService()Landroid/app/ContextImpl;)
5 | JNIEnv->CallObjectMethodV(android.app.ActivityThread, getSystemService())
6 | JNIEnv->FindClass(android/app/ContextImpl)
7 | JNIEnv->GetMethodID(android/app/ContextImpl.getPackageManager()Landroid/content/pm/PackageManager;
8 | JNIEnv->CallObjectMethodV(android.app.ContextImpl, getPackageManager())
9 | JNIEnv->GetMethodID(android/content/pm/PackageManager.getPackageInfo(Ljava/lang/String;I)Landroid/
10 | JNIEnv->NewStringUTF("cn.missfresh.application")
11 | JNIEnv->CallObjectMethodV(android.content.pm.PackageManager, getPackageInfo("cn.missfresh.applicat
12 | JNIEnv->GetFieldID(android/content/pm/PackageInfo.applicationInfo Landroid/content/pm/ApplicationI
13 | JNIEnv->GetObjectField(android.content.pm.PackageInfo, applicationInfo Landroid/content/pm/Applica
14 | JNIEnv->GetFieldID(android/content/pm/ApplicationInfo.sourceDir Ljava/lang/String;)
15 | JNIEnv->GetObjectField(android.content.pm.ApplicationInfo, sourceDir Ljava/lang/String; => "/data/
16 | JNIEnv->GetStringUtfChars("/data/app/cn.missfresh.application-1")
17 | JNIEnv->GetFieldID(android/content/pm/ApplicationInfo.dataDir Ljava/lang/String;)
18 | JNIEnv->GetObjectField(android.content.pm.ApplicationInfo, dataDir Ljava/lang/String; => "/data/da
19 | JNIEnv->GetStringUtfChars("/data/data/cn.missfresh.application")
```

8.调用0x218A8

9.

```
1 | JNIEnv->GetFieldID(android/content/pm/ApplicationInfo.nativeLibraryDir Ljava/lang/String;)
2 | JNIEnv->GetObjectField(android.content.pm.ApplicationInfo@36d64342, nativeLibraryDir Ljava/lang/Str
3 | 1/lib/arm64")
   JNIEnv->GetStringUtfChars("/data/app/cn.missfresh.application-1/lib/arm64")
```

10.读/proc/pid/fd，匹配包名+base.apk，0x12EA38存放指向base.apk完整路径的指针的指针

11.

```
1 | JNIEnv->FindClass(com/secneo/apkwrapper/H)
2 | JNIEnv->GetStaticFieldID(com/secneo/apkwrapper/H.ISMPAASLjava/lang/String;)
3 | JNIEnv->GetStaticObjectField(class com/secneo/apkwrapper/H, ISMPAAS Ljava/lang/String; => "###MPAAS
4 | JNIEnv->GetStringUtfChars("###MPAAS###")
```

12.将得到的结果和###MPAAS###比较，0x12E7F8指向0x137D9C，0x137D9C存放比较结果

13.调用0x22068

14.调用0x23568

15.调用0x1F250

16.将字符串lib/libart.so存放于0x1378A8

17.读/proc/self/maps，找权限为"r-xp"的lib/libart.so

18.初始化下列字符串:

```
1 | /data/user/0/cn.missfresh.application/.cache
2 | /data/user/0/cn.missfresh.application/.cache/oat/arm64
3 | /data/user/0/cn.missfresh.application/.cache/classes.dve
4 | /data/app/cn.missfresh.application-xxx/oat/arm64/base.odex
```

19.fstat /data/app/cn.missfresh.application-xxx/oat/arm64/base.odex

20.计算md5(不太清楚具体算的什么)，0x12EC98指向0x130080，0x130080存放计算结果

21.access /data/user/0/cn.missfresh.application/.cache/classes.dve，不存在则把之前算的md5写入该文件；存在则读取其中的值和之前算的比较，不相等则写入新计算的值

22.调用0x3371C(根据标记位决定是否调用)

23.调用0x1FDC8

24.初始化下列字符串:

```
1 | /data/user/0/cn.missfresh.application/.cache/libDexHelper32
2 | /lib/armeabi-v7a/libDexHelper.so
3 | /lib/armeabi/libDexHelper.so
4 | assets/libDexHelper32
```

25.调用0x3766C

26.调用0xB4B94

27.调用0x9C0BC

28.调用0xD5CDC

29.调用0x24C08

30.调用0x1C40C

31.调用0xA7D3C

32.结束

总结

样本混淆强度还是比较大的，比前两篇文章中的样本要复杂很多。不过分析过程中也是有一些技巧：比如位置相邻的函数之前其实是有联系的，和另外某些壳的代码有类似的地方(估计也是抄来抄去)，可以网上搜一下旧版本的分析博客，有一些函数名和字符串没有被抹去，等等。

[看雪招聘平台创建简历并且简历完整度达到90%及以上可获得500看雪币~](#)

最后于 17小时前 被houjingyi编辑，原因：

上传的附件:

- [fix-cfg.py](#) (24.13kb, 12次下载)
- [libDexHelper.so](#) (1.16MB, 9次下载)

☆

12

👍

4

¥

☆

收藏 · 12

👍

点赞 · 4

¥

打赏

➡

分享

📖

课程

🏠
首页

💬
论坛

📄
招聘

☰
发现

☆

12

👍

4

¥

最新回复 (3)



huangjw 15小时前

2 楼 0 ...

谢谢分享，学习中。 目测是精华贴

极客



huaerxiela 3小时前

3 楼 0 ...

666

极客



王麻子本人 2小时前

4 楼 0 ...

硬

极客



sun-shine

内容

回帖

表情

高级回复

返回