

sgavmp 篇

觉着降生

也许在业界中 sgavmp 算是一个比较高的点了；它让很多人望而生畏，特别是对于我这种菜鸟来说，它简直是一个无法跨越的屏障。和猫一样人也对一些事物充满好奇，但往往催生你欲望的不是好奇而是外界事物。前不久有一个阿里安全部的面试邀请，记得最后一面的时候面试官几次问到了我他们 avmp 的实现原理，在当时我几乎除了听过其名字外对此一无所知，这便燃起了我的好奇心。

聊到 sgavmp 就不得不聊聊 liteVM，不过我并不想在这里过多的描述它，我会单独写一篇 liteVM 的文章。在逆 sgavmp 早期 litevm 并没有引起我的过多注意，从 sdk 中我知道有它的存在，但是它是怎样的存在就不得而知了；然而你想逆 sgavmp 你就不得不过 litevm 这一关，它出现的机率不亚于 sgavmp；早期的时候它经常会乱入，随着我们把坑慢慢填满，才终于发现原来 litevm 是这样的一个存在，它作为一个附属大礼包免费送给你了。

逆向 sgavmp 过程是极其煎熬和富有挑战的，因为它确实有难度；做逆向的人应该最喜欢流水式的程序，你不用思考，很多时候静态分析就解决了大半工作，剩下的部分只需看着伪代码在模糊点调试一下即可。困难和恐惧来自未知，当程序打破流水式的时候，你会发现即使有伪代码你仍看不懂，程序不在有连续性，它的走向是未知的；在未知中给你添加几个小黑盒，黑盒隐藏了大部分逻辑，当你兴高采烈的弄明白黑盒子时，你发现原来它只是个盒子，盒子里面的东西才是你想要的；当你再次高兴的把里面看清楚时，你发现原来你想要的东西还是在外边，你永远不会知道下一步等待你的是什么。代码风格好的程序也是逆向者的最爱，因为数据结构清晰，我们几乎不用动脑就能轻松的还原；如果编程者刻意隐藏结构之间的关系，让本来存在直接关系的结构变成间接关系，那想必会增加逆向难度。一个巨大的数据结构也会增加逆向的难度，想想如果你去逆向 linux 内核（假设没有源码），想还原 task_struct 结构你需要花特别多的力气，一个是它自身大，另外是和它存在关系的结构也非常巨大，想正确的还原每个结构的每个属性还是比较难的。

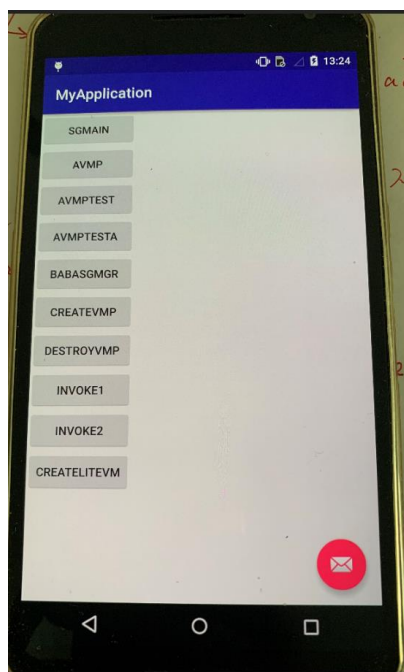
追寻正法

对于 sdk 类或者插件类程序，我并不喜欢直接调试或者逆向它的宿主程序，原因有三；一、sdk 或者插件本身就已经很复杂，和宿主融合到一起后会更加复杂，无疑增加逆向成本，二、需要找到 sdk 或者插件的调用入口点，三、需要绕过宿主的对抗。当一个宿主程序包罗万象大的夸张的时候，你千万别高估自己的定位能力，也千万别高估自己的耐心。

对抗 sdk 或者插件我的思路是从最小化集成到完全集成，通俗讲就是你来作为它的宿主，你负责它生命周期管理，需要什么集成什么，我也确是这么做的。

你的宿主可以很轻量化，并且能随时定制，仅提供你所需要的功能即可。

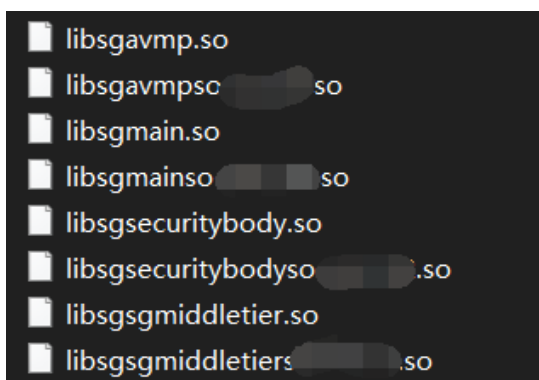
下图是我定制的宿主的 view，它提供了我所需要的功能。



多道

sgmain、sgavmp、sgsecuritybody 等前身是百川 sdk 下的无线保镖，最早由聚安全开发，同时对外对内都提供安全能力，对外提供低版本的 5.x，对内提供更具安全能力的 6.x 版本；5.x 版本不具备 avmp、litevm 功能，也不具备其他插件的能力。

如某内部 app 集成了无线保镖，它将拥有如下图所示部分（具体看实即集成）：



就 sgavmp 来说，libsgavmp.so 是一个压缩包，是一个完整的 apk，而带版本号的 so 才是真正的 so 文件。



这些插件中，sgmain 是主插件，其他插件强依赖这个插件，sgmain 对其他插件可能存在弱依赖关系，也就是说 sgmain 可以独立运行，但其他插件不可以，sgmain 插件的某些功能被

单独拿出实现成了其他插件，因此对其他插件可能存在弱依赖。

sgmain 插件：

```
{ "version": "6.4.176", "hasso": true, "pluginname": "main", "pluginclass": "com.alibaba.wireless.security.main.plugin.SecurityGuardMainPlugin", "dependencies": "", "weakdependencies": "securitybody:6.4.0;misc:6.4.0;", "reversedependencies": "securitybody:6.4.0;misc:6.4.0;avmp:6.4.0;nocaptcha:6.4.0", "thirdpartyso": false }root@shamu_t:~#
```

sgavmp 插件：

```
{ "version": "6.4.38", "hasso": true, "pluginname": "avmp", "pluginclass": "com.alibaba.wireless.security.avmp.plugin.SecurityGuardAVMPPlugin", "dependencies": "main:6.4.0;securitybody:6.4.0", "weakdependencies": "", "thirdpartyso": false }root@shamu_t:~#
```

我之所以这么了解，是因为在逆向过程中遇到了解决不了的问题，我还特意注册申请了该 sdk，详细的了解了文档中我想关注的地方；但很遗憾申请到的 sdk 版本是 5.x 的，上面也提到了它没有 avmp，最终它也没能解决我的问题；但通过了解文档相关信息我更加了解这个 sdk 了，它对我最终逆向是有益的；因此提醒大家有些时候收集信息真的很重要，千万不要陷入埋头的怪圈。

我用的版本是 sgmain6.4.176, sgavmp6.4.38，但为了保密性，我在文中隐藏了一些关键点，希望见谅。**该文档仅仅用来学习交流，用来提高安全门槛，不能用来做恶意事情，否则后果自付！**

初禅

当 sgmain 插件启动之后，我们就可以创建 avmp 了，来让我们通过 sdk 代码捋一下它的创建过程：

1、首先调用 createAVMPIntance 函数

```
@Override // com.alibaba.wireless.security.open.avmp.IAVMPGenericComponent
public IAVMPGenericInstance createAVMPInstance(String arg3, String arg4) {
    a v0 = new a(this, this.a);
    v0.a(arg3, arg4);
    return (IAVMPGenericInstance)v0;
}
```

2、内部类调用 doCommand(60901)

```
void a(String arg4, String arg5) {
    this.c = this.b.getRouter().doCommand(60901, new Object[]{arg4, arg5}).longValue();
}
```

3、so 层先准备 command，它的 JNI_OnLoad 函数大致如下图：

```
JNI_OnLoad() {
    init_global_objs((int)v2);

    env_ = getEnv(0);

    result = set_solt(env_); // SGPluginExtras long slot
    set_global_commandptr(v3); // JNI_OnLoad ^ 0x37 下次在调用JNI_OnLoad函数

    return create_avmp_command_prepare(); // 准备command
}
```

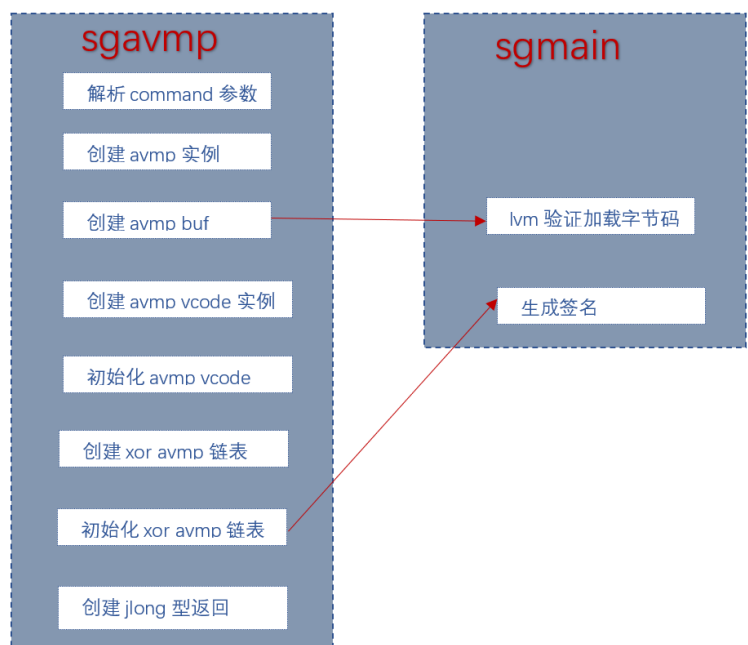
4、创建 avmp 实例

```
int __fastcall createAUMPInstance(command_arg *a1, const char *a2)
createAUMPInstance

var_30      = -0x30
var_2C      = -0x2C
var_28      = -0x28
var_24      = -0x24
var_20      = -0x20

PUSH        {R4-R7,LR}
ADD         R7, SP, #0xC
PUSH.W      {R8-R11}
SUB         SP, SP, #0x14
MOV         R4, R1
LDR         R1, =(off_A00BEDB8 - 0xA0093296)
ADD         R1, PC, #off_A00BEDB8
LDR         R1, [R1]
LDR         R1, [R1]
STR         R1, [SP, #0x30+var_20]
```

创建过程还是比较复杂的，对应结构也比较复杂，我以简单的图示来来展示一下重要的创建过程（图示中的名称都是我自己命名的，不代表真实情况）：



菩提证果

经过了漫长的逆向它慢慢的展现在我面前，我这里把一些结构的部分截图展示一下：
Avmplnst

```

struct AvmpInst {
    int fcodeLen; //
    int scodeLen;
    size_t maxMemSize; // 0
    int defaultMemSize; // 0
    int reservedSize; // 0
    void* getVbufSize; //
    void* vmpInitVcode; //
    void* releaseVcode; //
    void* vmpInterp; // 存

```

AvmpInstVcode

```

struct AvmpInstVcode {
    struct AvmpCodeReg reg; //
    int memSizethresholdBak; //
    int lastemSizethreshold; //
    char mbuf[64];
    void* symBase; /
    void* moduleBase; /

```

XorAvmpInst

```

struct XorAvmpInst { //
    void* xorAvmpinst; // avmpInst ^ time
    int timeRsr31; // 时间右移31位
    int time;
    int timersr31t; // 时间右移31位
    void* aVmpModule1; //
    void* aVmpModule2; //
    void* sign; //
    int signSz;
    void* vmpBuf; //
    struct XorAvmpInst* next;
};

```

还有很多复杂的结构，我就不一一展示了，感兴趣的同学可以自己逆向挑战一下。它有自己的内存管理，这个可能是大多 vmp 不具备的功能，外层数据需要下沉两层；一层是 vmp 内存数据对应的偏移，一层是真正的 vmp 内存。数据全程加密，当你使用数据时先进行解密，读取数据后在加密回去。

很多数据操作、交换、算法单单靠 vmp 指令实现是不现实的（代码膨胀的厉害、很多依赖关系无法解决），avmp 依赖两种外部调用实现；一种是封装 libc 库函数(叫封装可能不贴切，叫引用更合适)，我把此类函数称作 innerFunc，它可以完成内外部数据的交换等操作，如 memcpy，另一种调用也或多或少包含库函数，但除此之外它还可能调用 sgmain 中的相



```

MOV     R0, R0
BLX     R1
LDR.W   R2, [R9, #0x28]
MOV     R4, R0
MOV     R0, R9
MOV     R1, R8
BLX     R2
LDR     R2, [SP, #0x28+var_20]
MOV     R0, R5
MOV     R1, R4
BLX     __aeabi_memcpy
LDR     R0, =(off_A00BEDB8 - 0xA00B7112)
LDR     R1, [SP, #0x28+var_1C]
ADD     R0, PC, #off_A00BEDB8
LDR     R0, [R0]
LDR     R0, [R0]
SUBS    R0, R0, R1
ITTT EQ

```

关逻辑甚至是 lvm，我把此类函数称之为 handler，此类 handler 共 13 种，在初始化时会用 200xx 这样的编号进行注册。

```

struct AvmpHandler { // 共13种
    void* handler; // 
    // 0x20001, 0x20002, 0x20003, 0x20004, 0x2001
    // 0x20012, 0x20013, 0x20014, 0x20015, 0x2001
    // 0x20018
    int vtype;
    struct AvmpHandler* prev;
    struct AvmpHandler* next;
};

```

如某个 handler 逻辑中包含调用 command 的逻辑；这两种外部函数对应其 vmp 指令类型

```

break;
}
v7 = (int)getdataFromdocommand1203(v23); // 从docommand1203获取数据
v10 = 54;
v9 = 184;
v12 = 170;
}
if ( v9 != 184 )
{
    v7 = (int)getdataFromdocommand12014(v23);
    v10 = 54;
}
while ( 1 )

```

相同，关于创建我就说这么多把。

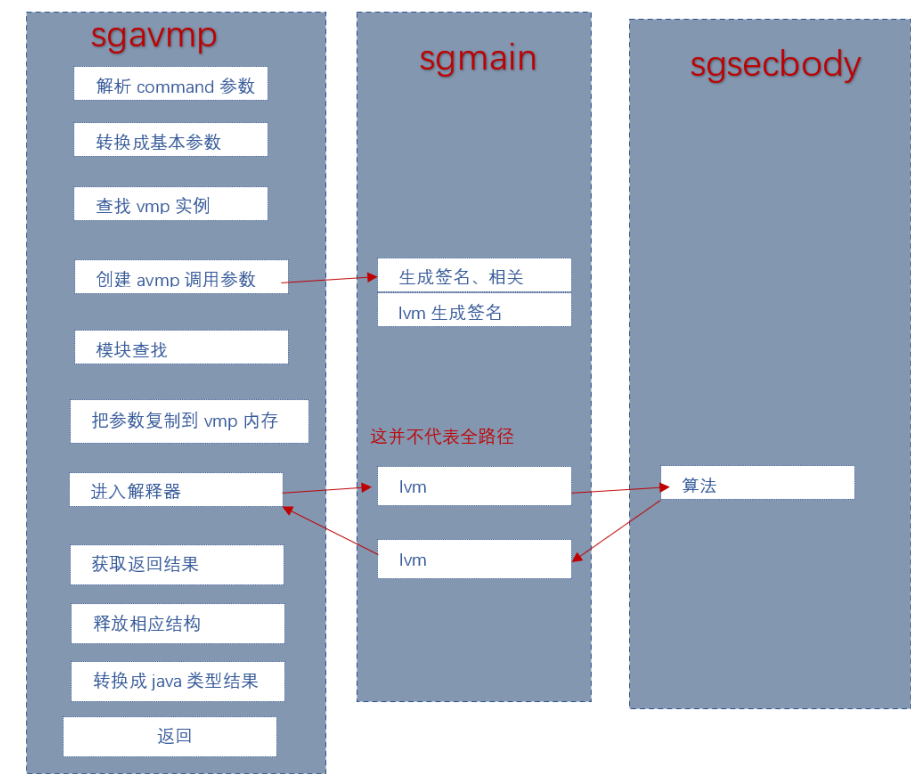
轮转法轮

波罗奈城

创建 avmp 实例后我们就可以进行 vmp 调用了，调用其大多是为了实现数据签名，我们来看一下 sdk 代码，调用 avmp 是通过 doCommand(60902)来实现的，long 型参数就是

```
@Override // com.alibaba.wireless.security.open.avmp.IAVMPGenericComponent$IAVMPGenericInstance
public Object invokeAVMP(String arg5, Class arg6, Object[] arg7) {
    return this.b.getRouter().doCommand(60902, new Object[]{Long.valueOf(this.c), arg5, arg6, arg7});
}
```

创建 avmp 实例时返回的，它大概调用序列如下图：



摩揭陀国

创建 avmp 实例时我们会得到字节码，字节码的前 8 个字节用来匹配解释器，目前 avmp 实现了三种解释器；调用哪个算法是通过匹配模块符号来确定的，字节码 9-16 字节用来确定模块符号，通过模块符号匹配定位模块索引，最终我们就可以找到字节码了。

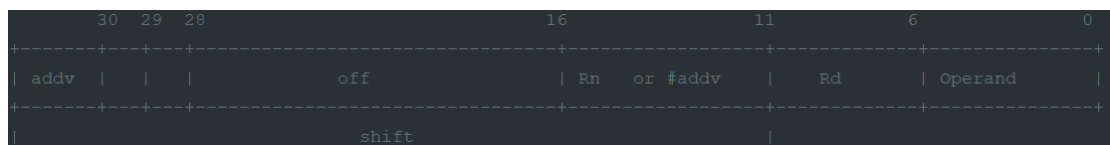
```

//      +-----+
//      0 |version      | // 前两个
//      4 |              |
//      8 +-----+ symBase //
//      |  sym 个数      | // addr
//      c |  symLen      | // 第三个
//      10+-----+ moduleBas
//      |  module1      |
//      |  |              | // 模块索
//      14|  modulen     |
//      +-----+ //模块名字
//      |  symbol section | // 异或后
//      |                  |
//      +-----+ module 节

```

释迦国

指令解析，就如同 arm 指令一样它定义了一套自己的指令集，指令长度同样为 4 字节，它大概如下图的样子（并不能十分准确）：



做逆向可以直接通过逆向工具看到汇编代码(它向我们隐藏了解码过程)，对于指令的解析，运行时 cpu 直接译码，静态时反编译器帮助我们解码；在实现自定义指令或者 vmp 时，解码工作是我们实现解释器的重要功能，解释器负责按照指令定义格式一个指令一个指令的一步一步解析，先加载指令；

```

ASRS      R5, R3, #0x10
STRB      R2, [R1, #0x1A]
STRH      R7, [R3, #0x3A]
STR       R0, [SP, #0xD4]
MOV       R8, R11 ; 更新指令索引
ADDS      R0, R2, #1
BEQ       loc_A14641FA
MOV       LR, R9
MOV       R1, R2
: ADD.W   R12, SP, #0xD8
0 LDR.W   R9, [SP, #0xB0]
3 B.W     loc_A14631EC ; module 部分解析

```

接着解析指令；


```

; LOAD:A2A181D0↓j
LDR      R0, [SP,#0x50] ; 指令解析
MOV      R11, R12
STR      R4, [SP,#0x04]
STR      R1, [SP,#0x08]
LDR.W    R6, [R0,R1,LSL#2] ;
LDR.W    R10, [SP,#0xC0]
SBFX.W   R0, R6, #0, #6 ; 取指令0-6位
STR.W    LR, [SP,#0x98]
CMP      R0, #0 ; 和0比
BLT      loc_A2A1726E
SBFX.W   R0, R6, #0, #6
CMP      R0, #0xF ; 和16比
BGE.W    loc_A2A172FC
SBFX.W   R0, R6, #0, #6
CMP      R0, #8 ; 和8比
BGE.W    loc_A2A173B0
SBFX.W   R0, R6, #0, #6
CMP      R0, #3 ; 和3比
BGE.W    loc_A2A17536
SBFX.W   R0, R6, #0, #6

```

我们以 avmp 某个指令为例（如指令 000C7BC2，指令类型为 2）：

1、取指令类型

```

BLT.W    loc_A2A17300
SBFX.W   R0, R6, #0, #6
CMP      R0, #2 ; 和2比
BLT.W    loc_A2A179F4

```

取0-6位

2、取源寄存器 Rn，取偏移

```

UBFX.W   R2, R6, #0x8, #5 ; 取11-15位
UBFX.W   R3, R6, #0x10, #0xC ; 取16-27位

```

3、取指令第 29、30 位

```

MOV.W    R1, #0x1000
MOV.W    R0, #0x2000
UBFX.W   R2, R6, #0x8, #5 ; 取11-15位
UBFX.W   R3, R6, #0x10, #0xC ; 取16-27位
AND.W    R1, R1, R6, LSR#17 ; 指令>>17 & 0x1000, 取30位
AND.W    R0, R0, R6, LSR#15 ; 指令>>15 & 0x2000, 取29位

```

4、F1 = off | CODE[30] | CODE[29]

5、load [context + 4*Rn]

6、OF = 指令>>16 & 0xc000, 取 31, 32 位

7、offset = OF | F1

8、addr = base + offset

```

AND.W    R1, R1, R6, LSR#17 ; 指令>>17 & 0x1000, 取30位
AND.W    R0, R0, R6, LSR#15 ; 指令>>15 & 0x2000, 取29位
ORRS     R1, R3 ; F1 = off | CODE[30]
LDR.W    R2, [R12,R2,LSL#2] ; load [context + 4*Rn]
ORRS     R0, R1 ; F2 = F1 | CODE[29]
MOV.W    R1, #0xC000
AND.W    R1, R1, R6, LSR#16 ; OF = 指令>>16 & 0xc000, 取31, 32位
ORRS     R0, R1 ; offset = OF | F1
ADD      R0, R2 ; 偏移 + addv

```

9、取指令 6-11 位为目标寄存器 Rd，保存数据到目标寄存器

11 UBFX.W	R1, R6, #6, #5	; 取6-11位
loc_A1463E18 00 STR.W	R0, [R12,R1,LSL#2]	; CODE XREF: LOAD:A146400A↓j ; 保存到目标内存
loc_A1463E1C LDR	R0, [SP,#0x74]	; CODE XREF: LOAD:A1463D88↑j

指令解析还是比较复杂的，解释器是由 n 多这样的块组成，最终完成各种指令的解析。

FFFFFFFE 类似是 pop 指令

FFFFFFEE 类似是 mov 指令

xxxxxxA 指令类似 ldr r0, [[base + off], adv], add r0, off + shift, str r0 [base + rd]

FFFFFFFA 类似是 mov 指令把内存值移动到寄存器, ldr r0, [[base + off], adv], str r0, rd

00000014 指令会调用 innerfunc 或者 handler

.....

我们需要特别关注一些类型为 14 的指令，因为它负责调用外部例程(上面提到的)，如调用 innerFunc:

```
40 F0 8E 84 BNE.W      LOC_A1404226
15 98      LDR          R2, [SP,#0x54]
53 46      MOV          R3, R10
10 90      LDR          R0, [SP,#0x74]
63 46      MOV          R11, R4
51 46      LDR          R1, [R2,#0x64]
80 80      LDR          R0, [R0]
C0 43      STR          R0, [R1,#0x3C]
C6 F3 84 10 UBFX.V     R0, R6, #6, #5
5C F8 20 00 LDR.W      R0, [R12,R0,LSL#2]
FF 20      CMP          R0, #0xFF
80 F2 91 04 BHI.W      loc_A146424E
81 E0 00 00 ADD.W      R0, R1, R0,LSL#2      ; r1是vampCode r0是相对vampCode的偏移
1C 46      MOV          R4, R3
76 46      MOV          R6, LR
D0 F8 A8 10 LDR.W      R1, [R0,#0xA8]      ; 调用innerfunc
18 46      MOV          R0, R2
80 47      BLX          R1                  ; 调用innerfunc
25 90      LDR          R5, [SP,#0x94]
80 F8 E0 0C B.V        loc_A1464316
; -----
```

在例如调用 handler，先根据注册类型查找 handler:

```
LDR          R5, [SP,#0x94]
ADD.W        R1, R1, #0xA8      ; 加载handler表头
LDRD.W       R8, R4, [SP,#0xC4] ; ???

loc_A1464258      ; CODE XREF: LOAD:A1464264↑j
LDR          R2, [R1,#4]        ; 循环找到对应类型的handler,如20001
CMP          R2, R0             ; 对比handler 类型
BEQ          loc_A146430C       ; 进入handler处理块
LDR          R1, [R1,#0xC]      ; 下一个handler
MOVS        R2, #0xC1 ;
CMP          R1, #0             ; 低7位不为0的类型的: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000
```

查找到 handler 后调用相应的 handler:

```
loc_A146430C      ; CODE XREF: LOAD:A146425C↑j
LDR          R1, [R1]          ; 加载handler的func
MOV          R4, R3
LDR          R0, [SP,#0x54]
MOV          R6, LR
BLX          R1                ; 进入对应的handler, 仅一个参数为vampinst

loc_A1464316      ; CODE XREF: LOAD:A146393E↑j
10 LDR.W      R8, [SP,#0xC4]
12 ADD.W      R2, R11, #1
MOV          LR, R6
```

舍卫城

avmp 主要用来实现签名算法的，其逻辑十分复杂，从上面的调用序列我们可以得到执行逻辑除 vmp 指令外，它还在 lvm 和 securitybody 之间穿梭；即使不存在 vmp 指令这套算法逻辑还是很复杂的，它的复杂不再于最后计算算法的本身，而是在于对抗逆向上，想彻底

搞明白它的算法，还原它的算法还是很难的。

对于签名算法我想说的是，如就 sign 算法：

首先它由三部分构成（三个部分拼接而成），算法外层分为两大类，算法内部每套都调用了三种密码学类算法（两套公用了四种密码学算法）。

其次你更多的需要关注 handler，有一个 handler 特别重要，每次它都是进入 lvm 的入口，也是最终调用算法的入口。

再次最终算法和 securitybody 有关，它会间接的进入 lvm。

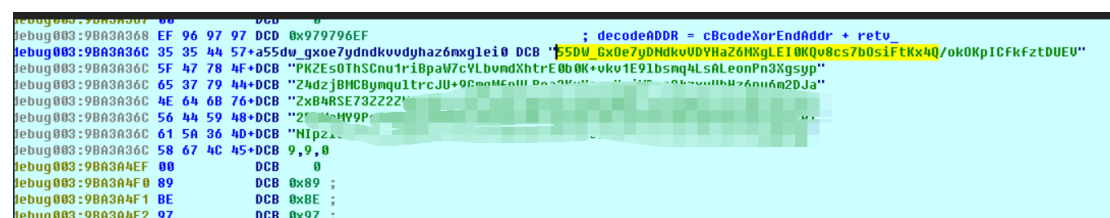
娑罗双树

金刚经

说了这么多，其实我觉得了解它的 vmp 指令或者还原它的指令、算法意义不大，我们应该更多的学习别人的优点，学习他们怎么做加固对抗的；另外补充说明他们真的很强大，做的东西真的很牛逼。

法华经

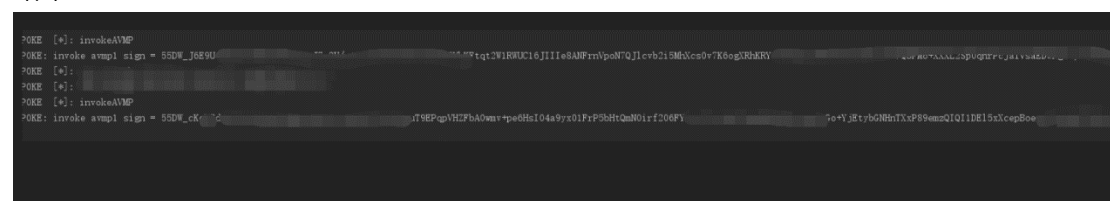
展示结果，1 调试获取



2、程序黑盒



结果



涅槃经

不过是个东西就有利有弊，我也肤浅的提出一些自己的看法，又不对的地方还请原谅：
优点： 全程加密；复杂、特别复杂，完全是指令级别的 vmp，有自己的指令集，设计过于复杂；拥有自己的内存管理，数据在内部保密性好；拥有多层数据管理，数据分散，保密性好。

缺点： 有独立内存也恰巧给了别人窥视的机会；很多复杂算法如加密算法(不是指 vmp 的内部数据加密)都是最终调用的外部实现这给了别人可乘之机；内部加解密也给了别人观看数据的机会；浪费空间浪费内存，内部内存太大，大多都用不到，一般小内存手机根本跑不起来，频繁的 vmp 内存操作，影响效率。

虽然这么说，但我自己可能连 vmp 都实现不了，哈哈，因为它确实已经非常优秀了，该有的都有，你能想到的它都有。

再次声明：**该文档仅仅用来学习交流，用来提高安全门槛，不能用来做恶意事情，否则后果自付！**

本人该项目 github 地址: https://github.com/ylcangel/crack_sgavmp
会不会提交某些数据结构看个人心情，你千万别指望我会这么做，谢谢！