Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心


# Android APP共享库加固

硕士研究生 侯留洋

2017年12月10日

# 内容提要

- 背景
- 基础知识
  - 共享库文件结构
- 加固方案
  - 简单的加解密
  - 基于逆向工具的混淆
  - 一些反调试
  - Vmp介绍
- 加固利弊
  - 正面和负面

# 预备知识

# 背景介绍

- 为什么要加固?
  - 开发者若不对自身应用进行安全防护，极易被病毒植入、广告替换、支付渠道篡改、钓鱼、信息劫持等，严重侵害开发者的利益
- 为什么是加固共享库
  - 加固APP时，通过将需要加固的函数，写成库函数，然后把库加固，从而避免函数被逆向、破解

# 共享库的结构

# 共享库的结构

- 库文件结构
  - 库文件开始是一个描述整个文件组织的头部
  - 节区部分包含大量信息
    - 字符串表
    - 符号表(dynsym)
      - 在so文件中，每个函数的结构描述在.dynsym中
    - PT_DYNAMIC段
      - 该段用于查找各节区在内存中的位置

| 库文件格式 |
| --- |
| ELF头部 |
| 程序头部表 |
| 节区1/段1 |
| 节区2/段1 |
| … |
| 节区n/段n |
| 节区头部表 |

# 共享库的结构

- .hash节区
  - 哈希表支持符号表访问，可用于函数的快速查找
- PT_DYNAMIC段
  - 该结构数组包含各节区的位置信息，可对各节区抽取

```
Typedef struct{
        Elf32_Sword d_tag;
        union{
        Elf32_Word d_val;
        Elf32_Addr d_ptr;
        }
}
```

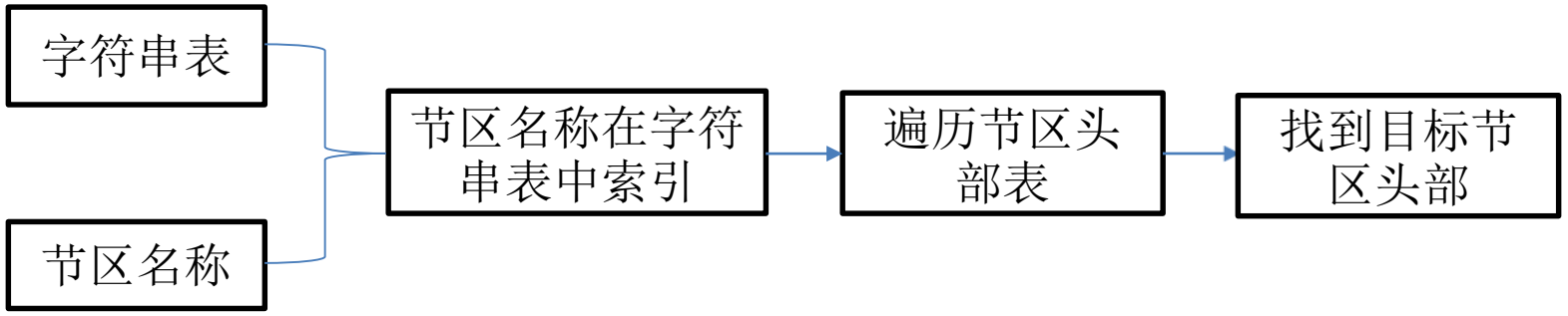
# 加固方案

# 简单加解密

- So库文件在使用时首先需要被加载器加载到内存中
  - 加载完后会自动执行init节区的函数
  - 由于init节区的函数在动态库加载完成后会被自动执行，因此可以通过init节区实现动态库的解密
- 基于init节区的加固流程
  - 函数或关键区段加密
  - 共享库加载到内存
  - 执行init节区中的解密函数后使用

# 获取目标节区地址

- 节区地址
  - 由共享库文件头中字符串表偏移找到字符串表然后找出要加密节区名称在字符串表中的偏移值X
  - 由共享库文件头中节区头部表偏移获取所有节区头部信息
  - 将节区头部中name值与X比较找到目标节区头部获得其位置和大小



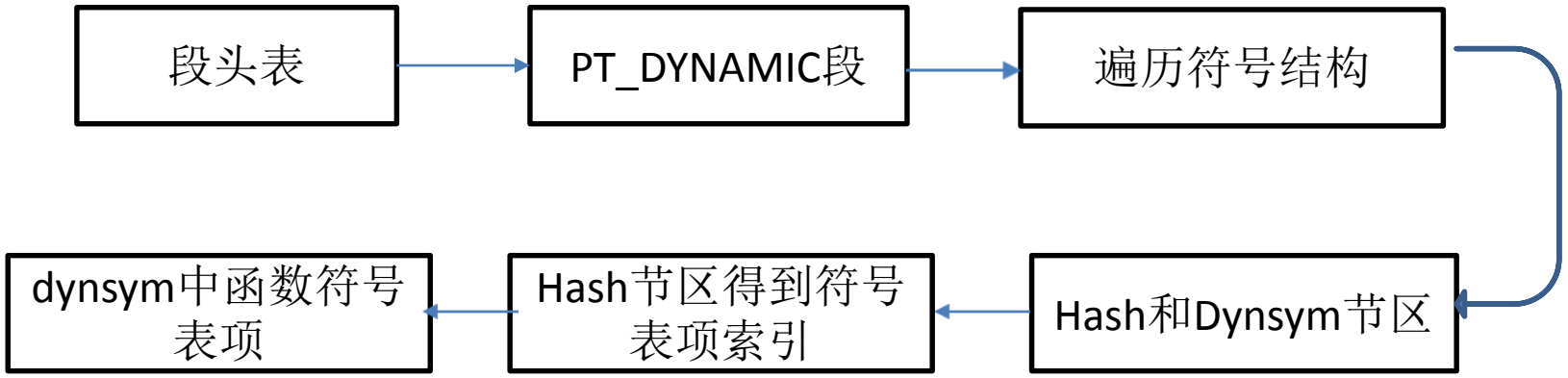
# 哈希表

- 哈希表通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。
  - 由函数名k（关键字）得到hash值f(k)，bucket[f(k)]）得到索引Y
  - 对不同的关键字可能得到同一散列地址，即k1≠k2，而f(k1)=f(k2)，这种现象称为碰撞（Collision）。
  - 拉链法处理冲突，即hash表中chain数组的作用：chain[Y]=Y1 chain[Y1]=Y2 chain[Y2]=Y3 …

| Hash节区结构 |
| --- |
| nbucket |
| nchain |
| Bucket[0] |
| … |
| Bucket[nbucket-1] |
| chain[0] |
| … |
| Chain[nchain-1] |

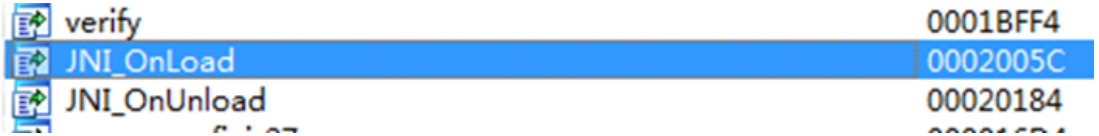
# 获取目标函数地址

- 函数地址
  - 获取.hash和.dynsym节区地址，然后得到函数符号表项



# 加密后效果

- IDA观察
  - 导出函数




  - 及函数内容

```
.lfxtext.1:0002005C                 EXPORT JNI_OnLoad
.lfxtext.1:0002005C JNI_OnLoad
.lfxtext.1:0002005C                 STCCC   p6, c6, [PC,#0x2DC]!
.lfxtext.1:0002005C ; ---------------------------------------------------------------------------
.lfxtext.1:00020060                 DCD 0x5694D310, 0x882BC9A5, 0x4D6CF295, 0x2B50584E, 0xB88380B
.lfxtext.1:00020060                 DCD 0x1D31404F, 0x2E0C411B, 0x90A8B2, 0x5D4C8E1F, 0x526642C7
.lfxtext.1:00020060                 DCD 0x722178E7, 0x2E0F455B, 0x1C223528, 0x50343BE8, 0x598978A
.lfxtext.1:00020060                 DCD 0x9A1A795F, 0x9FD6A2BD, 0x704C04AA, 0x91BA3CEC, 0xB90FBF5
```

- 对抗方式
  - 逆向出init区加密算法，解密加密数据

# 节区/函数 解密（2）

- 节区/函数 解密
  - 通过init节区函数
  - 共享库加载完成后自动执行
- 新的共享库解密目标库
  - 解密过程不一定需要在被加密的库文件自身的init节区实现，可以在其他后续被加载的库文件中的init节区实现
  - 不用修改目标目标库的代码
  - 保证加密区域使用前被新的共享库解密

# 一种解密方式

5 4b e1 9f 78 eb 90 98 a5 a8 2c dd af a0 46 1 43 49 c1 79
d4 70 4d 3b 28 d7 19 37 da 30 15 e5 9b 10 c8 2f 59 bb 3d 3
e4 45 ee fc be b3 6b 31 2d a2 b7 93 96 ae 17 a9 a4 a c0 e2
53 82 1f e0 61 5e c2 68 d6 67 e9 86 94 ce 47 8 14 b de 6f
55 4a 58 52 c 3c 32 27 74 e8 62 95 c3 f5 c4 97 7e 1c 64 38
bd 22 57 84 2b ad 80 b5 e7 8c 36 cf 89 65 c9 6a f9 d2 25 0
2a 4 ca 1a 8d ff c5 b6 4c b8 f3 df 2e ec 5f fd 6d 7d 7 5d
9e 35 8b 8a b9 d1 56 69 29 cd 60 d9 ed 54 81 40 d5 8e d3 e6
cb a3 b1 9c dc 9d 24 f0 9 1b 77 b2 26 db 71 2 83 a1 39 b4
72 fe 4e d8 3f f6 fb 50 21 18 6 23 bf 1e 85 87 6c fa d0 16
bc 42 9a 76 51 13 92 34 44 7f a7 ea b0 ba 5b f7 cc 63 75 41
e 7c 73 f4 12 48 11 3e 5c 8f 99 66 c6 3a f a6 5a 6e 7b f1
7a f2 aa 91 f8 ac 33 88 20 e3 1d ab ef c7 d 4f 0 0

c1= tranformTable[tranformTable[i+1]
+
tranformTable[tranformTable[i]+ tranformTable[

蓝色两项值相加得到索引X
红色两项值交换

i3=1

i3=i3&ff=1

c1=tranformtable[1]=4b

i2=i2+c1=4b

i2=i2&ff=4b

c12=tranform[4b]=8

tranformtable[1]=8

tranformTable[4b]=4b

c6 = 8

c12=info[0]

i1 =4b + 8

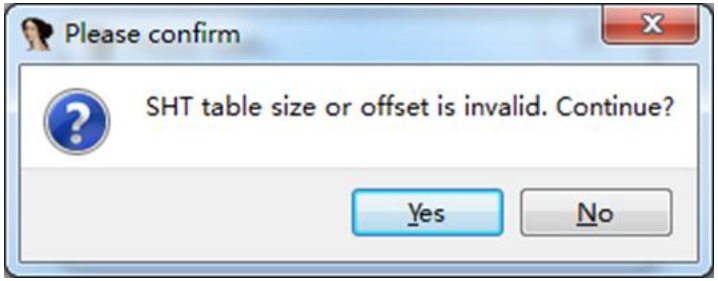
i1 &= 0xff=4b + 8

c1 = tranformTable[4b + 8]=3c

c1=52^info[0]

# 基于逆向工具的混淆

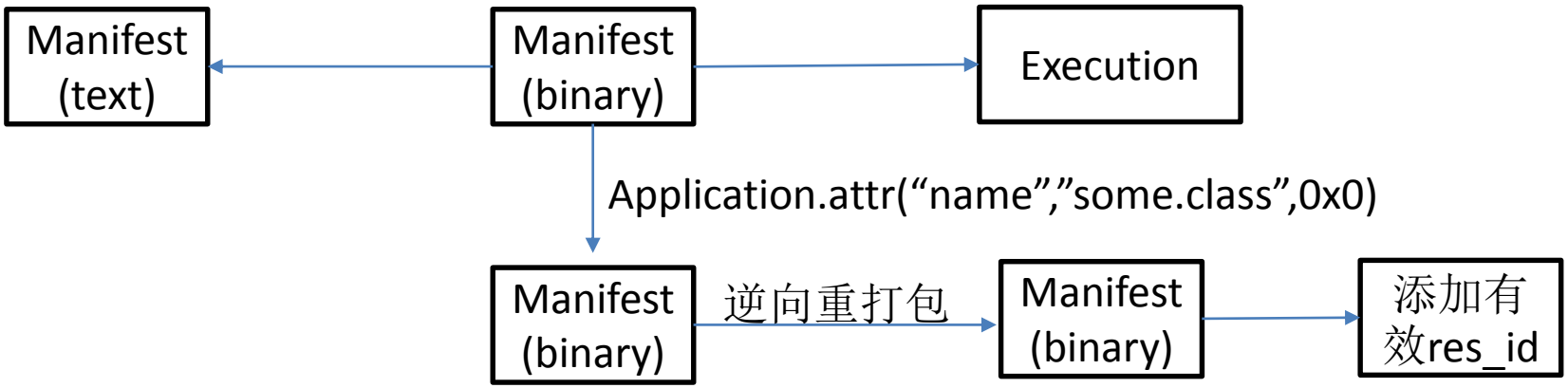
- ELF文件混淆
  - Android系统不需要section节区信息，
  - 但逆向工具会根据这些信息显示共享库内容，可以混淆文件相关信息起到一定对抗作用




- AXML文件介绍
  - AXML就是AndroidManifest.xml对应的二进制文件,apk包中存储的就是AXML文件。
  - Android系统解析AXML文件只是通过属性的res id号，而反编译工具会根据属性名解析并添加res id。

# 基于逆向工具的混淆

- Axml文件混淆
  - 可以通过添加特定的属性，让经过反编译后的apk一开始就运行在错误的环境




结果：重打包中逆向工具解析属性名并添加有效id，但APK中没有实现该类，所以启动时会报错“there is no trap.class”。

# 一些反调试

- 常见反调试方法
  - 当程序被调试时，程序的/proc/{PID}/status文件会有特殊变化

```
Tgid:   29139 /*线程组号*/
Pid:    29139 /*进程pid process id*/
PPid:   281 /*父进程的pid parent processid*/
TracerPid:      0 /*跟踪进程的pid*/
Uid:    10123   10123   10123   10123 /*uid euid suid fsuid*/
```

  - Fork返回值：子进程返回0，父进程返回子进程标记

```
if (fork() == 0) {//进入子进程
    int pt = ptrace(PTRACE_TRACEME, 0, 0, 0); //PTRACE_TRACEME，表示让父进程跟踪自己
    if (pt == -1)  exit(0);//失败说明在被调试，就退出
    while (1) {//子进程检查父进程的“/proc/{PID}/status”保护父进程不被调试
        fd = fopen(filename, "r");
        while (fgets(line, 128, fd)) {
            if (strncmp(line, "TracerPid", 9) == 0) {
                int status = atoi(&line[10]);//获取父进程tracerid值
```

# 一些反调试

- 23946端口检测反调试
  - IDA调试安卓应用时，android_server会注入应用程序，并监听23946端口与pc端的调试器通信
  - 调试器可以通过修改Android_server（ELF）文件使用其他端口来绕过反调试的检测

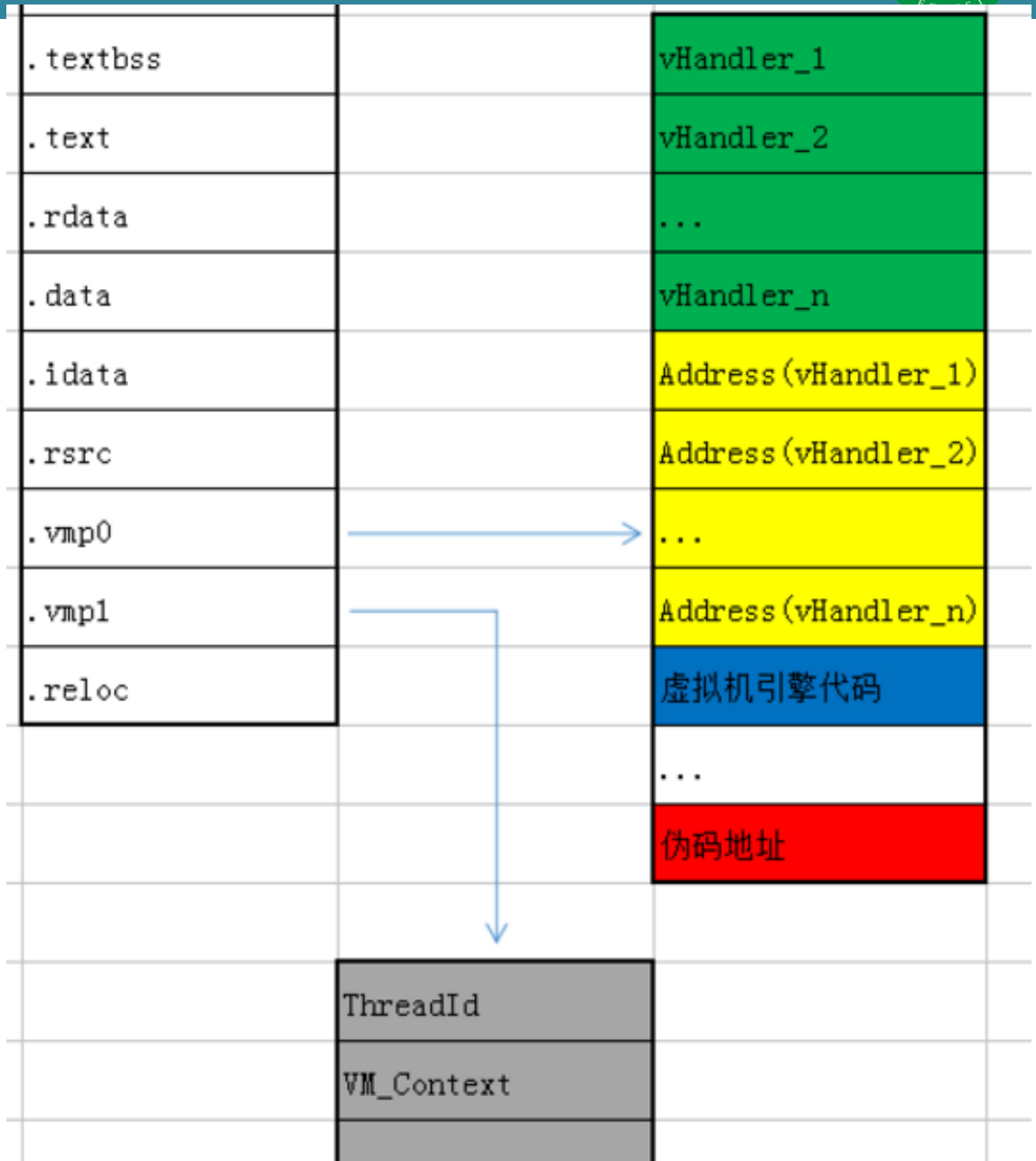
```
shell@pisces:/ $ su
root@pisces:/ # cd /data
root@pisces:/data # ./android_server
IDA Android 32-bit remote debug server(ST) v1.19. Hex-Rays (c) 2004-2015
Listening on port #23946...
```

# Vmp介绍

- 虚拟机保护
  - 将由编译器生成的本机代码(Native Code)转换成字节码(Bytecode)
  - 将控制权交由虚拟机，由虚拟机来控制执行
  - 转换后的字节码非常难以阅读，增加了逆向的复杂性
- 组成部分
  - Dispatcher(分发器)，负责读取Byte-code，并指派对应的handle进行解释执行。
  - Handler，虚拟机指令通过平台nativecode（如PC平台即为x86指令）的实现。
  - 虚拟机堆栈和存储(虚拟机寄存器)

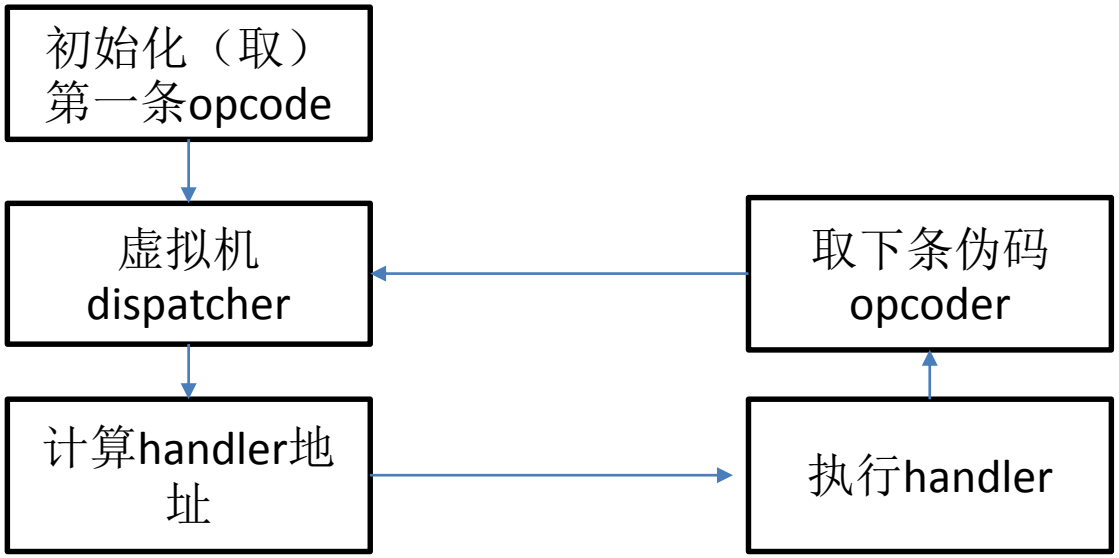
# 一种内存视图

- 一个内存视图
  - 字节码与handler对应，循环取出伪码处的字节码并到对应Handler处解释执行



# 一般流程

- 解释执行流程
  - 某个内存地址开始作为虚拟机的私有堆栈和变量存放区域
  - 引擎进入一个循环，该循环从计算出来的esi的位置开始，不断取出数据，执行虚拟机指令
  - 执行完所有的esi指向的伪码，退出虚拟机



# 虚拟机指令

- VM_MOVw_EDISTACKw_EBPSTACKw
  - 代码：

```
0043E7EC   0FB646 FF      MOVZX EAX,BYTE PTR DS:[ESI-1]          ; *
0043E7F6   28D8           SUB AL,BL                               ; *
0043E7FE   C0C8 05        ROR AL,5                                ; *
0043E80C   F6D8           NEG AL                                  ; *
0043E816   34 0E          XOR AL,0E                               ; *
0043E822   28C3           SUB BL,AL                               ; *
0043E82E   66:8B55 00     MOV DX,WORD PTR SS:[EBP]                ; *
0043E835   83C5 02        ADD EBP,2                               ; *
0043F03F   4E             DEC ESI                                 ; *
0043F045   66:891438      MOV WORD PTR DS:[EDI+EAX],DX            ; *
```

  - 功能：把EBPSTACK栈顶1个word的数据存储到EDISTACK。
  - 把原来的指令复杂化增加逆向分析的工作量

# 上下文轮循

- 上下文指当前的寄存器状态

真实指令：

Xchg ebx，ecx

Add eax, ecx

虚拟指令

```
GetREG R2 ; R2 = ECX
GetREG R1 ; R1 = EBX
SetREG R2 ; ECX = value of EBX
SetREG R1 ; EBX = value of ECX
GetREG R2
GetREG R0 ; R0 = EAX
Add
SetREG R0
```

有上下文轮循的虚拟指令

```
[Map R1 = ECX, R2 = EBX] ; exchange
GetREG R1          ; R1 = ECX
GetREG R0          ; R0 = EAX
Add
SetREG R0          ; R0 = EAX
```

# 加固利弊

# 加固利弊

- 正面:
  - 保护自己核心代码算法,提高破解/盗版/二次打包的难度
  - 缓解代码注入/动态调试/内存注入攻击
- 负面
  - 会降低低软件的执行效率
  - 部分应用市场会拒绝加固后的应用上架
- VMP这种使用虚拟化技术的加固方式，能增加逻辑的复杂性，是优秀的加固方法。

# 谢谢！

大成若缺，其用不弊。大盈若冲，其用不穷。大直若屈。大巧若拙。大辩若讷。静胜躁，寒胜热。清静为天下正。