litevm 篇

乾为天

要不要写这篇文章我是想了很久的，本来不想再写；写文章看似简单，实际上合理的组织语言写一篇即有点深度又易懂的文章是很难的；但自己许下的愿就这样半途而废有悖天地法则，所谓天圆地方，圆圆满满，做事需有头有尾，乃为规律。

我同样秉着和上篇文章一样的态度，尽量用浅显的语言让更多的人了解它。

地水师

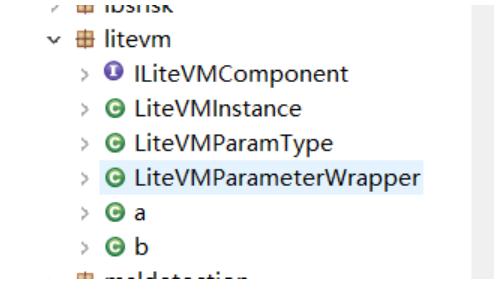
想研究无线保镖中的安全特性，你就不得不弄懂 litevm；如 avmp、安全对抗、数据收集等，很多核心安全的实现被内置在这个 litevm 中，sgmain6.3.80 对应 sgavmp6.3.29 中还没有 litevm，由此可以看出 litevm 和 avmp 是独立的，两者并不存在直接关系。avmp 的时间早于 litevm，litevm 是一个较新版本的安全组件。

其实义如其名 litevm 就是一个小的轻量的 vm，下面是我给它的一个定义：

```
// LiteVM
// lvm更像一个vm，或者沙盒
// 可能想把它打造成集vmp化elf的vm 、 dex vmp 、jni vmp 最终成为一个集所有于一体的vm
```

火天大有

说什么都没有直接拨开它更具说服力，我们先看看它的 sdk 中暴露的组件包结构形式：



在看看 sdk 中接口形式：

```
@InterfacePluginInfo(pluginName="main")
public interface ILiteVMComponent extends IComponent {
    byte[] callLiteVMByteMethod(LiteVMInstance arg1, int arg2, LiteVMParameterWrapper[] arg3) throws SecException;

    long callLiteVMLongMethod(LiteVMInstance arg1, int arg2, LiteVMParameterWrapper[] arg3) throws SecException;

    String callLiteVMStringMethod(LiteVMInstance arg1, int arg2, LiteVMParameterWrapper[] arg3) throws SecException;

    void callLiteVMVoidMethod(LiteVMInstance arg1, int arg2, LiteVMParameterWrapper[] arg3) throws SecException;

    LiteVMInstance createLiteVMInstance(String arg1, String arg2, byte[] arg3, Object[] arg4) throws SecException;

    void destroyLiteVMInstance(LiteVMInstance arg1) throws SecException;

    void reloadLiteVMInstance(LiteVMInstance arg1, byte[] arg2) throws SecException;
}
```

看看 sdk 中 litevm 包装类：

```
        }
        case -1376216685: {
            if(!v7.equals("reloadLiteVMInstance")) {
                v7_1 = -1;
                break;
            }

            v7_1 = 1;
            break;
        }
        case -1295482945: {
            if(!v7.equals("equals")) {
                v7_1 = -1;
                break;
            }

            v7_1 = 8;
            break;
        }
        case -611610698: {
            if(!v7.equals("createLiteVMInstance")) {
                v7_1 = -1;
                break;
            }

            v7_1 = 0;
            break;
        }
        case -514295244: {
            if(!v7.equals("destroyLiteVMInstance")) {
                v7_1 = -1;
                break;
            }

            v7_1 = 2;
            break;
        }
        case 66098133: {
            if(!v7.equals("callLiteVMStringMethod")) {
                v7_1 = -1;
                break;
```

我们在看看最接近 sgmain 的部分：

```
static b a(ISecurityGuardPlugin arg6, String arg7, String arg8, byte[] arg9, Object[] arg10) {
    return new b(arg6, ((Long)arg6.getRouter().doCommand(0x30D5, new Object[]{((Object)arg7), ((Object)arg8), ((Obje
}
```

看了这么多可以反过来证实一下我上面说的话了吧，他们预期将来对外提供一系列的安全方法，这些暴露给上层调用的方法是基于 dex 的方法（至于它将来内部想通过什么方式提供，如 dex vmp、 jni 反射、或者 elf vmp、直接字节码，这都不重要）；不过它可不仅仅提供 dex 方式的方法，它内部是个虚拟机，它可以提供任何方式的方法，并且保镖内部调用 lvm 完全和上层没有关系，vm 执行的是字节码，vm 还可以 vmp 化。sgavmp 的字节码加载、签名计算都调用了 lvm。或许在不久的将来你将无法在无线保镖中找到 sgavmp 的踪影。

山天大畜

就我研究的版面目前还未能提供它上层暴露的所有功能，上层也没有显著调用 lvm 的地方，应该是目前版本还未实现那些功能。目前只提供内部调用，在无线保镖内部其他组件通过 lvm command 方式调用 lvm 相关逻辑，这个后面再说。

想研究 lvm 首先需要了解它的创建过程，我们大致看一下它的创建过程。

初九：有厉、利己

我们就以从上到下的次序来看它的创建过程，首先通过代理调用创建 lvm 实例：

```
case -611610698: {
    if(!v7.equals("createLiteVMInstance")) {
        v7_1 = -1;
        break;
    }

    v7_1 = 0;
    break;
}
```

```
case 0: {
        return a.a(a.a(),
        ((String)((Object)arg8)[0]),
        ((String)((Object)arg8)[1]),
        ((byte[])((Object)arg8)[2]),
        ((Object[])((Object)arg8)[3]));
    }
```

```
private LiteVMInstance a(String arg2, String arg3, byte[] arg4, Object[] arg5) throws SecException {
    return new LiteVMInstance(b.a(a.a, arg2, arg3, arg4, arg5), arg2, arg3);
}
```

九二：舆说輹

在通过 doCommand 调用底层 sgmain 的 command 方法：

```
static b a(ISecurityGuardPlugin arg6, String arg7, String arg8, byte[] arg9, Object[] arg10) {
    return new b(arg6, ((Long)arg6.getRouter().doCommand(0x30D5, new Object[]{((Object)arg7), ((Object)arg8), (
}
```

而 0x30d5=12501，对应 native 层的 command 是 1，0x19，1，这也恰好是创建 lvm 的地方。

不过从下层看你会发现更多，你会发现 1,0x19,x 是一系列和 lvm 相关的命令：

```
if ( do_command )
{                                       // 创建lvm相关command
  v2 = 0;
  if ( !(*do_command)(1, 25, 1, 0) )    // // goto litevm_entry
  {
    v2 = 0;
    if ( !(*do_command_)(1, 25, 5, 0) )
    {
      v2 = 0;
      if ( !(*do_command_)(1, 25, 2, 0) )     // call lite vm
      {
        v2 = 0;
        if ( !(*do_command_)(1, 25, 3, 0) )
        {
          v2 = 0;
          if ( !(*do_command_)(1, 25, 4, 0) && !(*do_command_)(1, 25, 6, 0) )// destroy lvm
            v2 = 1;
        }
```

九三：良马遂，利艰贞。日闲舆卫，利有攸往

创建过程比较复杂，会生成很多的数据结构，如外层 litevm，实例 lvm_inst、lvm context 等、同时初始化这些结构，如计数、索引、实例方法等。

```
litevm_inst_ = (lvm_inst *)j_CreateLitevm(0, v17);
litevm_inst = (LiteVmInst *)litevm_inst_;
if ( litevm_inst_ )
{
  if ( v19 )
    ((void (*)(void))litevm_inst_->lvm_func22)();
  if ( (_DWORD)v6
    && SHIDWORD(v6) >= 1
    && ((int (__fastcall *)(LiteVmInst *, _DWORD))litevm_inst->lvm_func9)(litevm_inst, HIDWORD(v6)) )
  {
    v13 = 0;
    a4 = 21;
    goto LABEL_17;
  }
  v13 = (litevm *)create_litevm_struct((int)litevm_inst, v5, v4);
  init_litevm_idx(v13, (_QWORD *)v10, a3, &a4);
  if ( a4 )
    goto LABEL_17;
  if ( (*(int (**)(void))(g_destroy_litevm_vdata + 16))() != 1 )
  {
```

注册 lvm 方法，这些方法包含系统方法，jni 方法，和 sgmain 内部方法。

```
if ( !is_init_llc )                          // is_init_llc
{
  pthread_mutex_lock((pthread_mutex_t *)&ptread_mutex_lock_1);
  if ( !is_init_llc && !create_lvm_global_struct() && !do_register_llc_method() && !j_RegisterExtForAndroid() )
    is_init_llc = 1;
  pthread_mutex_unlock((pthread_mutex_t *)&ptread_mutex_lock_1);
}
```

总之为 lvm 准备先决条件。

风火家人

初九：闲在家，悔亡

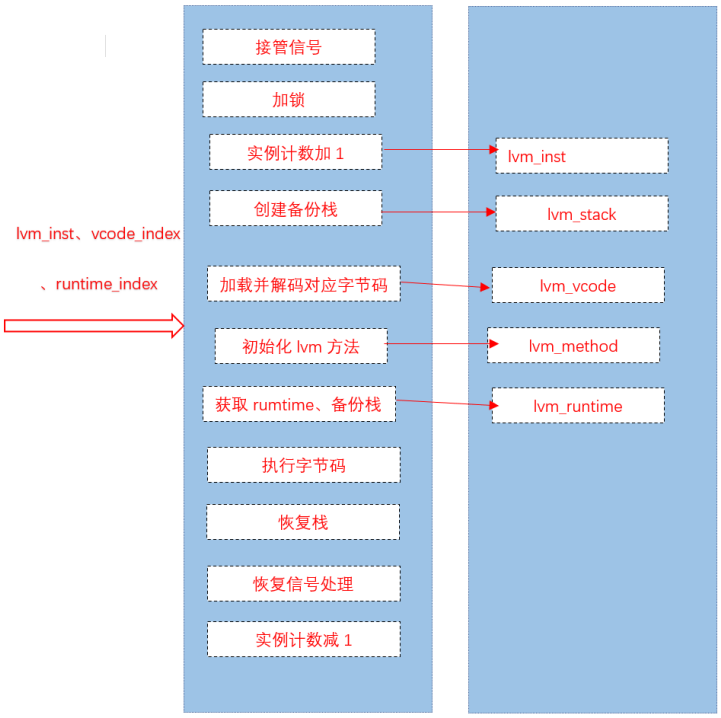
在执行 command 时，sgmain 优先在 lvm_command 中查找 command，找不到在去普通的 command 中查找。lvm command 查找主要存在两个主链；分别由 0x2261、0x2622 作为表头，没找到返回 0，外层返回 0x52。找到后即跳转到指定的 command 处。

但 lvm 字节码最终的执行入口都是 command1,0x19,2； 执行过程是调用 lvm 获取 lvm_runtime、打开 runtime，拷贝外部参数到 lvm 栈，进入 lvm 执行字节码，关闭 lvm_runtime 开关，得到返回结果。

```
v15 = v2->vcode_idx;
lvm_runtime_index = (*(int (__fastcall **)(_DWORD))(*(_QWORD *)&lvminst->lvm_inst + 0x28))(*(_QWORD *)&lvminst->lvm_inst);// 构造lvm_contxt->v3 v4, 获取lvm_runtime 索引
if ( lvm_runtime_index )
{
  v17 = lvm_runtime_index;
  (*(void (__fastcall **)(_DWORD, _DWORD, int, int))(lvm_inst + 0x38))(lvm_inst, v14, v22, lvm_runtime_index);// 拷贝vm参数
  v5 = (*(int (__fastcall **)(_DWORD, int, int))(lvm_inst + 0x30))(lvm_inst, v15, v17);// 进入vm
  if ( v5 )
  {
                                        // 这个返回结果影响后面sprintf
    (*(void (__fastcall **)(_DWORD, int))(lvm_inst + 0x2C))(lvm_inst, v17);// 销毁lvm_contxt->v3 v4,
    v9 = 0;
    v18 = 6;
  }
  else
  {
    v9 = (*(int (__fastcall **)(_DWORD, int))(lvm_inst + 0x40))(lvm_inst, v17);// 获取vm返回的返回结果
    (*(void (__fastcall **)(_DWORD, int))(lvm_inst + 0x2C))(lvm_inst, v17);
    v5 = 0;
    v18 = 0;
  }
  v21 = v18;
```

六四：富家，大吉

真正的程序逻辑影射的是 lvm 对应的字节码，字节码的解释执行就对应着程序逻辑的前进。lvm 执行字节码工作大致如下：



九五：王假有家，勿恤，吉

这里有一个比较有意思的地方，在执行字节码前先解码，这个解码不是解密，而是把外部的字节码转换为 lvm 内部能够解释执行的 code，我把这种 code 称之为 lvm_vcode；每条 vcode 都对应一个 handler，vcode 的 handler 可以相同，也可以不同；它还可以是 lvm_method。lvm 是基于栈的。

如这里列举一条字节码的解密过程（字节码好像也是 32 位）：

```
                        ; CODE XREF: bb2i34u32clsb+182↑j
UBFX.W          R1, R9, #9, #5 ; jumptable 9FFC4F8A case 2
STR.W           R1, [LR,#0xC]
UBFX.W          R1, R9, #5, #4
STR.W           R1, [LR,#8]
UBFX.W          R2, R9, #0xE, #5
CMP             R1, #5
STR.W           R2, [LR,#0x10]
UBFX.W          R2, R9, #0x13, #5
STR.W           R2, [LR,#0x14]
BHI.W           loc_9C3AC
LDR.W           R2, =(lvm_vmethod_list - 0x9C010)
ADD             R2, PC ; lvm_vmethod_list
ADD.W           R1, R2, R1,LSL#2
LDR.W           R1, [R1,#0x90]
B               loc_9C454
```

上九：有孚威如。终吉

每个 vcode 对应的 handler 都很短，handler 的主要责任就是执行 vcode，vcode 中标明了该执行的操作，如完成类似寄存器移动呀、两个数的累加呀、调用外部函数等。vcode 运算都是基于 lvm 栈进行的，你认为它有寄存器也可以，认为没有也可以。下面是一个短小的 handler 的图示：

```
      LDR             R0, [R10,#8]
      ADD             R2, R11, R0,LSL#3
      LDRD            R4, [R2]
      STRD            R4, [SP,#0x10]
      LDR             R0, [R10,#0xC]
      ADD             R2, R11, R0,LSL#3
      LDRD            R4, [R2]
      STRD            R4, [SP,#8]
      B               next_vcode ; glvm_code +
unction sub_9E28C
```

坎为水

我们以第一条vcode的执行过程为例，字节码执行的第一步就是找到第一个vcode的handler（第一个 handler 一般都是保存栈信息）：

```
  STMFD           SP!, {R4-R7,LR}
  ADD             R7, SP, #0xC
  STMFD           SP!, {R8-R12}
  SUB             SP, SP, #0x28
  LDR             R11, [R7,#8] ; r7指向进入函数时的sp，指向new_lvm_stack
  STR             R0, [SP,#0x20] ; lvm_inst
  STR             R1, [SP,#0x1C] ; lvm_runtime
  STR             R2, [SP,#0x18] ; glvm_lvm_code_list
  MOV             R10, R2
  STR             R10, [R11,#0x78]
  ADD             R10, R10, R3,LSL#5 ; 进入函数r3 = 0x000044DB, r3 lsl#5 = 0x89b60
                                     ; 在r2基础上加累加值，r2可能是个handler的base
                                     ; base + off 等于对应的handler
                                     ; 计算next call
  B               vcode_start ; 调试时，开始进入这里
ion lvm_entry
```

然后更新 vm 的 pc：

```
vcode_start                             ; CODE XREF: lv
                                        ;
                LDR             R9, [R10] ; 调试时，开始进入这里
                LDR             R0, [SP,#0x1C] ; lvm_runtime
                STR             R10, [R0,#8] ; 更新runtime->pc
                MOV             PC, R9  ; 调试时第一次跳转到loc_9DF6C
; END OF FUNCTION CHUNK FOR lvm_entry
;
```

保存栈信息（第一 vcode 的 handler）：

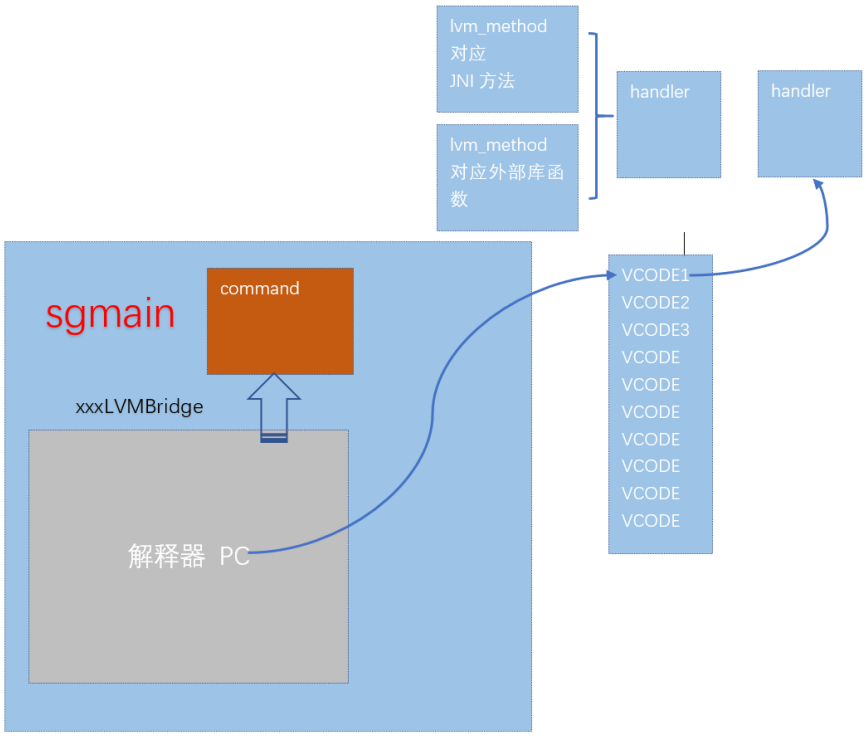
```
                STRD            R4, [R11,#0xC0]
                LDRD            R4, [R11,#0x48]
                STRD            R4, [R11,#0xC8]
                LDRD            R4, [R11,#0x50]
                STRD            R4, [R11,#0xD0]
                LDRD            R4, [R11,#0x58]
                STRD            R4, [R11,#0xD8]
                LDRD            R4, [R11,#0x60]
                STRD            R4, [R11,#0xE0]
                LDRD            R4, [R11,#0x68]
                STRD            R4, [R11,#0xE8]
                STR             R12, [R11,#0xF0]
                LDRD            R4, [R11,#0x78]
                STRD            R4, [R11,#0xF8]
                B               next_vcode ; glvm_code + 0x20 即32字节间隔
; End of function sub_9DF6C
```

进入下一个 vcode：

```
next_vcode                              ; CODE XREF:
                                        ; vm_call_jni_method+AC↓j ...
                LDR             R9, [R10,#0x20]! ; glvm_code + 0x20 即32字节间隔
                LDR             R0, [SP,#0x1C] ; lvm_runtime
                STR             R10, [R0,#8] ; 更新runtime->pc
                MOV             PC, R9  ; 调试时第一次走到
; END OF FUNCTION CHUNK FOR sub_9E738
```

雷泽归妹

就像 jvm 虚拟机一样，java 字节码中可以调用 java 方法（自己实现的）、java 库方法、native 方法，lvm 也提供了类似的方法调用，调用 lvm 自实现方法，调用外部方法（库函数如 libc 的方法、JNI 方法）、调用 sgmain 方法。大致如下图：



bridge 列表，有我单独标识的：

getVmBridge
get_lifeCycleFuncInfo_LVMBridge
dataCollectioninLVMBridge
encryptDataLVMBridge
decryptDataLVMBridge
safetokenWriteLVMBridge
safetokenReadLVMBridge
safetokenClearLVMBridge
safetokenRemoveFileLVMBridge
LiteVMCreateLVMBridge
LiteVMInvokeLVMBridge
LiteVMDestroyLVMBridge
AlgorithmHelperLVMBridge
downloadFileLVMBridge
setLifeCycleFuncIndexLVMBridge
getLifeCycleFuncInfoLVMBridge
freeLifeCycleFuncInfoLVMBridge
getTaskInfoLVMBridge
freeTaskInfoLVMBridge
invokeFuncBridgeLVMBridge
getCommonFunctionPtrLVMBridge
getGlobalContextLVMBridge
getJavaVMLVMBridge

如调用 lvm 方法入口：

```
LDRD            R4, [R11,#0x78]
STRD            R4, [R6,#0x78]
LDRD            R4, [R11,#0xF0]
STRD            R4, [R6,#0xF0]
LDR             R0, [SP,#0x20] ; a1
LDR             R1, [SP,#0x1C] ; a2
LDR             R2, [R10,#0x18] ; pc_addv
BLX             do_lvm_method_entry
LDR             R10, [R11,#0x78]
LDRD            R4, [R6,#0x40]
STRD            R4, [R11,#0x40]
B               next_vcode ; glvm_code + 0x20 即32字节间
```

根据解析 vcode 对应调用外部方法的类型调用真正的外部方法：

```
if ( vmethod->mtype > 16 || vmethod->is_litevm_method )
{
  v3 = vmethod->func;
  if ( vmethod->is_has_retv )
    a2->curr_stack.curr_stackp.RetV = ((int (__fastcall *)(_DWORD, _DWORD))v3)(a1, a2);// 带返回值
  else
    ((void (__fastcall *)(_DWORD, _DWORD))v3)(a1, a2);
}
else
{
  call_not_litevm_method(a1, a2, (int (__fastcall *)(_DWORD, _DWORD, _DWORD, _DWORD))vmethod->func);// 调用外部方法
}
```

外部方法部分截图：

cJSON_AddItemToObject
cJSON_CreateNull
cJSON_AddItemToObjectCS
cJSON_AddItemReferenceToArray
cJSON_AddItemReferenceToObject
cJSON_DetachItemFromArray
cJSON_DeleteItemFromArray
cJSON_DetachItemFromObject
cJSON_DeleteItemFromObject
cJSON_InsertItemInArray
cJSON_ReplaceItemInArray
cJSON_ReplaceItemInObject
cJSON_CreateTrue
cJSON_CreateFalse
cJSON_CreateBool
cJSON_CreateNumber
cJSON_CreateString
cJSON_CreateArray
cJSON_CreateObject
cJSON_CreateIntArray
cJSON_CreateFloatArray
cJSON_CreateDoubleArray
cJSON_CreateStringArray
cJSON_Duplicate
cJSON_Minify
cJSON_ToByte

AlgorithmHelperMd5
AlgorithmHelperMd5Hex
AlgorithmHelperSha1
AlgorithmHelperSha1Hex
AlgorithmHelperHmacSha1
AlgorithmHelperHmacSha1Hex
AlgorithmHelperHmacSha1Base64
AlgorithmHelperBase64Encode
AlgorithmHelperBase64Decode
AlgorithmHelperAesEncrypt
AlgorithmHelperAesDecrypt
AlgorithmHelperAes128EncryptBase64
AlgorithmHelperAes128DecryptBase64
sub_89A8C
AlgorithmHelperAes256EncryptBase64
AlgorithmHelperAes256DecryptBase64
AlgorithmHelperCompressData
AlgorithmHelperDecompressData
AlgorithmHelperGetRandom
AlgorithmHelperGetRandomBuffer
AlgorithmHelperGetRandomByte
AlgorithmHelperGetRandomString
AlgorithmHelperRcEncryptV1Base64
AlgorithmHelperRcDecryptV1Base64
AlgorithmHelperRcEncrypt
AlgorithmHelperRcDecrypt

VM_JNICallStaticCharMethod
VM_JNICallStaticCharMethodA
VM_JNICallStaticShortMethod
VM_JNICallStaticShortMethodA
VM_JNICallStaticIntMethod
VM_JNICallStaticIntMethodA
VM_JNICallStaticLongMethod
VM_JNICallStaticLongMethodA
VM_JNICallStaticVoidMethod
VM_JNICallStaticVoidMethodA
VM_JNIGetStaticFieldID
VM_JNIGetStaticObjectField
VM_JNIGetStaticBooleanField
VM_JNIGetStaticByteField
VM_JNIGetStaticCharField
VM_JNIGetStaticShortField
VM_JNIGetStaticIntField
VM_JNIGetStaticLongField
VM_JNISetStaticObjectField
VM_JNISetStaticBooleanField
VM_JNISetStaticByteField
VM_JNISetStaticCharField
VM_JNISetStaticShortField
VM_JNISetStaticIntField
VM_JNISetStaticLongField
VM_JNINewString

火水未济

好，虽然写的有些急促和不再状态，但我感觉我应该描述清楚了，最后赠上几个数据结构(不保证完全准确性):

lvm_method:

```
struct lvm_method { // 沙盒方法的抽象，多种类的，外部通过一个入口调用这些函数
    char* method_name; // 对应JNI method或其他方法名
    void* func; // 函数地址
    int mtype; // 暂时不知道这些数字代表啥， 0, 1， 2, 3， 4, 7 , 可以大于16
    int is_has_retv; // 是否存在返回值，1有0没有，仅适用于litevm 和 jni
    int is_litevm_method; // 0不是，1是， 它转接的jni方法此标记也为1
};
```

lvm_vcode:

```
struct lvm_vcode { // 解释型或自定义的code，目前实现了21个虚拟指令
    // 由两大类组成，一类是直接编译进去的默认的，一种是lvm_method实现的
    // vm先拿到vcode，然后更新runtime->vcode_pc，接着进入vcode-> handler
    void* handler; // 指令对应的解析方法，其余变量用于解析指令
    int v2; // 暂时都为0
    // 暂时这样命名
    int cond; // 某值(_R9 >> 5) & 0xF 、 (_R9 >> 6) & 0xF、 (_R9 >> 5) & 0x1F
    // (_R9 >> 9) & 0x1F 很多情况略，用于栈计算(保存栈起始地址 + retv_addv*8)用于保存pc_addv
    int op1; // 用于计算当前栈保存原操作1的位置
    int op2; // 用于计算当前栈保存原操作2，的位置(_R9 >> 14) & 0x1F 很多情况略
    int dst; // 用于计算当前栈保存目的操作数的位置，(_R9 >> 19) & 0x1F 很多情况略
    // 一个加值,多种用途 栈计算，vmethod id查找 vcode查找，返回结果地址计算
    // 取vmthod时pc_addv - 0x17FFFE， 但需小于0x17FFFD
    int addv; // 复合累计值，类似于arm指令中的shift_op
    int opn; // 可能相当于arm的opcode2
};
```

lvm_runtime:

```
struct lvm_runtime { // 最多支持62个，63个
    size_t inited; // 初始化后为0xB8A3
    size_t thread_id; // 当前线程id
    // 可以理解为pc，当前指令的位置，当前vcode对应的handler的起始位置
    void* vcode_pc; // 马上准备调用的函数；刚刚进入函数时，它保存的上一次调用的函数地址
    struct lvm_stack* caller_stack; // 调用者栈，从上次执行的curr_stack拷贝得来
    void* stack_top; // ((char*)new_stack + stack_size) & 0xFFFFFFF8，对齐8
    void* stack_base; // 1m的大小，指向其起始地址
    size_t stack_size; // 栈大小，默认0x100000=1m字节，它先从context->default_stack_size中获取大小
    void* v8;
    // 作为进入vmethod的当前栈使用
    struct lvm_stack curr_stack; // 创建新lvm_runtime时创建这个栈
    void* vm_runtime_end;
};
```

好，手工，累坏我了。
本人该项目 github 地址: https://github.com/ylcangel/crack_litevm

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！转载需标明出处，否则发现必究!!!