

动手实现代码虚拟机

厕所打灯笼 (/author/厕所打灯笼) · 2015/11/19 14:16

Author:chong_xx@sina.com



(/author/厕所打灯笼)

厕所打灯笼 (/author/厕所打灯笼)

0x00 什么是代码虚拟化

虚拟化实际上我认为就是使用一套自定义的字节码来替换掉程序中原有的native指令，而字节码在运行的时候又由程序中的解释器来解释执行。自定义的字节码是只有解释器才能识别的，所以一般的工具是无法识别我们自定义的字节码，也是因为这一点，基于虚拟机的保护相对其他保护而言要更加难破解。但是解释器一般都是native代码，这样才能使解释器运行起来解释执行字节码。其中的关系就像很多的解释型语言一样，不是系统的可执行文件，不能直接在系统中运行，需要相应的解释器才能运行，如python。

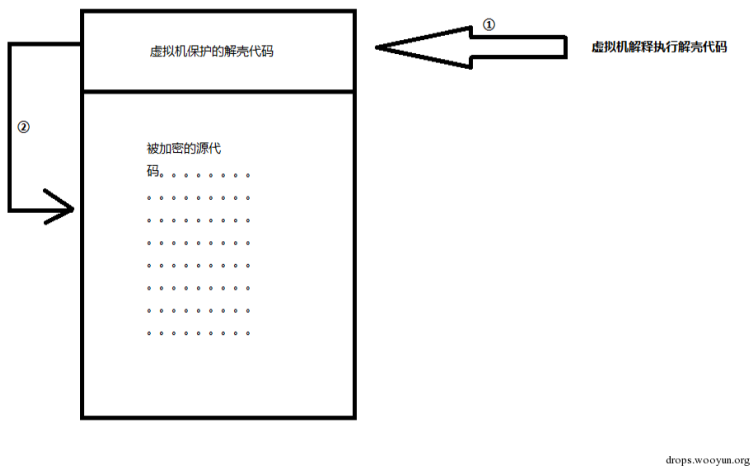
0x01 为什么研究代码虚拟化

目前很多地方都会用到虚拟化技术，比如sandbox、程序保护壳等。很多时候为了防止恶意代码对我们的系统造成破坏，我们需要一个sandbox，使程序运行在sandbox中，即使恶意代码破坏系统也只是破坏了sandbox而不会对我们的系统造成影响。还有如vmp,shielden这些加密壳就是内置了一个虚拟机来实现对程序代码的保护，基于虚拟机的保护相对其他保护而言破解起来会更加困难，因为使用现有的工具也是不能识别虚拟机的字节码。在见识过这类保护壳的威力之后，也萌生出了自己动手写一个的冲动，所以才有了本文。

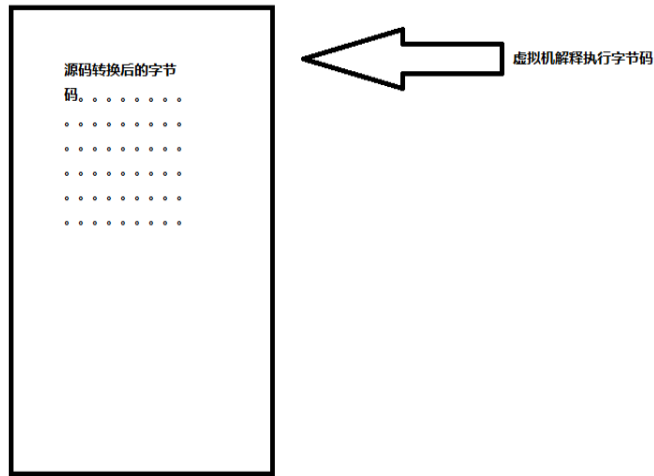
0x02 基于虚拟机的代码混淆

基于虚拟机的代码保护也可以算是代码混淆技术的一种。代码混淆的目的就是防止代码被逆向分析，但是所有的混淆技术都不是完全不能被分析出来，只是增加了分析的难度或者加长了分析的时间，虽然这些技术对保护代码很有效果，但是也存在着副作用，比如会或多或少的降低程序效率，这一点在基于虚拟机的保护中格外突出，所以大多基于虚拟机的保护都只是保护了其中比较重要的部分。在基于虚拟机的代码保护中可以大致分为两种：

- 1. 使用虚拟机解释执行解壳代码。这种混淆是为了隐藏原代码是如何被加密的，又是如何被解壳代码解密的。这种方式对于静态分析来说比较有效，但是对于动态调试效果不大。因为动态调试的时候完全可以等到解壳代码解密初源代码之后进行脱壳。只有配合其他保护技术才会有比较强的保护效果。



- 2. 把需要保护的程序源码转换为自定义字节码，再使用虚拟机解释执行被转换后的程序字节码，而程序的源码是不会出现在程序中的。这种方式不管静态还是动态都可以有效的保护。



drops.wooyun.org

可以看出两种保护的区别就是，第一种只保护解壳代码，没有保护源码。第二种直接保护了所有源码。所以第一种的强度也小于第二种。本文则是以第二种方式来实现保护，也就是保护所有源码。

在基于虚拟机的保护技术中，通常自定义的字节码与native指令都存在着映射关系，也就是说一条或多条字节码对应于一条native指令。至于为什么需要多条字节码对应同一条native指令，其实是为了增加虚拟机保护被破解的难度，这样在对被保护的代码进行转换的时候就可以随机生成出多套字节码不同，但执行效果相同的程序，导致逆向分析时的难度增加。

0x03 需要实现什么？

- 首先了解过代码虚拟化的原理之后，知道了其中的原理就是自定义一套字节码，然后使用一个解释器解释运行字节码。所以，目标分为两部分：

1. 定义字节码

字节码只是一个标识，可以随意定义，以下是我定义的字节码，其中每条指令标识都对应于一个字节

```
/*
 * opcode enum
 */
enum OPCODES
{
    MOV = 0xa0, // mov 指令字节码对应 0xa0
    XOR = 0xa1, // xor 指令字节码对应 0xa1
    CMP = 0xa2, // cmp 指令字节码对应 0xa2
    RET = 0xa3, // ret 指令字节码对应 0xa3
    SYS_READ = 0xa4, // read 系统调用字节码对应 0xa4
    SYS_WRITE = 0xa5, // write 系统调用字节码对应 0xa5
    JNZ = 0xa6 // jnz 指令字节码对应 0xa6
};
3
```

因为我的demo只是一个简单的crackme，所以只定义了几个常用的指令。如果有需要，可以在这个基础上继续定义出更多的字节码来丰富虚拟机功能。

2. 实现解释器

在定义好指令对应的字节码之后，就可以实现一个解释器用来解释上面定义的指令字节码了。在实现虚拟机解释器之前需要先搞清楚我们都要虚拟出一些什么。一个虚拟机其实就是虚拟出一个程序（自定义的字节码）运行的环境，其实这里的虚拟机在解释执行字节码时与我们真实处理器执行很相似。在物理机中的程序都需要一个执行指令的处理器、栈、堆等环境才可以运行起来，所以首当其冲需要虚拟出一个处理器，处理器中需要有一些寄存器来辅助计算，以下是我定义的虚拟处理器

```
/*
 * virtual processor
 */
typedef struct processor_t
{
    int r1; // 虚拟寄存器r1
    int r2; // 虚拟寄存器r2
    int r3; // 虚拟寄存器r3
    int r4; // 虚拟寄存器r4
    int flag; // 虚拟标志寄存器flag，作用类似于eflags
    unsigned char *eip; // 虚拟机寄存器eip，指向正在解释的字节码地址
    vm_opcode op_table[OPCODE_NUM]; // 字节码列表，存放了所有字节码与对应的处理函数
} vm_processor;
/*
 * opcode struct
```

```

*/
typedef struct opcode_t
{
    unsigned char opcode; // 字节码
    void (*func)(void *); // 与字节码对应的处理函数
} vm_opcode;
1

```

- 上面结构中r1~r4是4个通用寄存器，用来传参数和返回值。eip则指向当前正在执行的字节码地址。op_table中存放了所有字节码指令的处理函数。上面的两个虚拟出的结构就是虚拟机的核心，之后解释器在解释字节码的时候都是围绕着以上两个结构的。因为程序逻辑简单，所以只需要虚拟出一个处理器就可以了，堆和栈都不是必须的。程序中的数据我用了一个buffer来存储，也可以把整个buffer理解成堆或者是栈。

- 有了上面两个结构之后，就可以来动手写解释器了。解释器的工作其实就是判断当前解释的字节码是否可以解析，如果可以就把相应参数传递给相应的处理函数，让处理函数来解释执行这一条指令。以下是解释器代码

```

void vm_interp(vm_processor *proc)
{
    /* eip指向被保护代码的第一个字节
    * target_func + 4是为了跳过编译器生成的函数入口的代码
    */
    proc->eip = (unsigned char *) target_func + 4;
    // 循环判断eip指向的字节码是否为返回指令，如果不是就调用exec_opcode来解
    释执行
    while (*proc->eip != RET) {
        exec_opcode(proc);
    }
}
1

```

- 其中target_func是自定义字节码编写的目标函数，是eip指向目标函数的第一个字节，准备解释执行。当碰到RET指令就结束，否则调用exec_opcode执行字节码。以下是exec_opcode代码

```

void exec_opcode(vm_processor *proc)
{
    int flag = 0;
    int i = 0;
    // 查找eip指向的正在解释的字节码对应的处理函数
    while (!flag && i < OPCODE_NUM) {
        if (*proc->eip == proc->op_table[i].opcode) {
            flag = 1;
            // 查找到之后，调用本条指令的处理函数，由处理函数来解释
            proc->op_table[i].func((void *) proc);
        } else {
            i++;
        }
    }
}
5

```

- 解释字节码时首先判断是哪一个指令需要执行，接着调用它的处理函数。以下是target_func的伪代码。伪代码的逻辑就是首先从标准输入中读取0x12个字节，然后前8位与0x29异或，最后逐位与内存中8个字节比较，全部相同则输出success,失败输出error。以下的代码完全可以改成循环结构来实现，但是这里我偷懒了，全部是复制粘贴。

```

/*
mov r1, 0x00000000
mov r2, 0x12
call vm_read ; 输入
mov r1, input[0]
mov r2, 0x29
xor r1, r2 ; 异或
cmp r1, flag[0] ; 比较
jnz ERROR ; 如果不相同就跳转到输出错误的代码
; 同上
mov r1, input[1]
xor r1, r2
cmp r1, flag[1]
jnz ERROR
mov r1, input[2]
xor r1, r2
cmp r1, flag[2]
jnz ERROR
mov r1, input[3]
xor r1, r2
cmp r1, flag[3]
jnz ERROR
mov r1, input[4]
xor r1, r2
cmp r1, flag[4]
jnz ERROR
mov r1, input[5]
xor r1, r2
cmp r1, flag[5]
jnz ERROR
mov r1, input[6]
xor r1, r2
cmp r1, flag[6]
jnz ERROR
mov r1, input[7]
xor r1, r2
cmp r1, flag[7]
jnz ERROR
*/

```

- 相应处理函数代码在后文的完整代码中。有了以上关键函数，一个简单的虚拟机就可以运行了。在虚拟机中，还可以创建虚拟机堆栈以及更完整的寄存器来丰富虚拟机支持的指令。因为本程序相对简单所以没有用到堆栈，所有参数都通过寄存器传递，或者隐含在字节码中。有兴趣可以自己修改。

0x04 解释器解释执行过程

这里就用demo中的第一条字节码做演示来说明虚拟机中解释器解释执行时的过程，首先可以从上面看到解释器vm_interp执行时eip会指向target_func + 4，也就是target_func中内联汇编中定义的第一个字节0xa0，之后会判断eip指向的字节码是否为ret指令，ret指令是0xa3，所以不是eip指向的不是ret，进入exec_opcode函数进行字节码解释。

```
void vm_interp(vm_processor *proc)
{
    /* eip指向被保护代码的第一个字节
     * target_func + 4是为了跳过编译器生成的函数入口的代码
     */
    proc->eip = (unsigned char *) target_func + 4;

    // 循环判断eip指向的字节码是否为返回指令，如果不是就调用exec_opcode来解释执行
    while (*proc->eip != RET) {
        exec_opcode(proc);
    }
}
```

进入exec_opcode后开始在虚拟处理器的op_table中查找eip指向的字节码，当前就是0xa0，找到之后就调用它的解释函数。

```
void exec_opcode(vm_processor *proc)
{
    int flag = 0;
    int i = 0;

    // 查找eip指向的正在解释的字节码对应的处理函数
    while (!flag && i < OPCODE_NUM) {
        if (*proc->eip == proc->op_table[i].opcode) {
            flag = 1;
            // 查找到之后，调用本条指令的处理函数，由处理函数来解释
            proc->op_table[i].func((void *) proc);
        } else {
            i++;
        }
    }
}
```

字节码与解释函数的初始化在init_vm_proc中

```
proc->op_table[0].opcode = MOV;
proc->op_table[0].func = (void (*)(void *)) vm_mov;
```

```
MOV = 0xa0, // mov 指令字节码对应 0xa0
```

可以看出0xa0就对应着mov指令，所以当解释器遇到0xa0就会调用vm_mov函数来解释mov指令。

在vm_mov函数中首先把eip + 1处的一个字节和eip + 2处4个字节分别保存在dest和src中，dest是寄存器标识，在后面的switch中判断dest是哪个寄存器，在这个例子中dest是0x10，也就是r1寄存器，在case 0x10分支中就把*src赋值给r1。总体来看，前6个字节就是第一条mov指令，对应着mov r1, xxxx，xxxx就是这6个字节中的后4个，在这个例子中就是0x00000000。

```
unsigned char *dest = proc->eip + 1;
int *src = (int *) (proc->eip + 2);

// 前4个case分别对应r1~r4, 最后一个case中, *src
// 个字节赋值给r1
switch (*dest) {
    case 0x10:
        proc->r1 = *src;
        break;

    case 0x11:
        proc->r2 = *src;
        break;

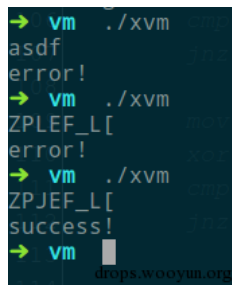
    case 0x12:
        proc->r3 = *src;
        break;

    case 0x13:
        proc->r4 = *src;
        break;

    case 0x14:
        proc->r1 = *(heap_buf + *src);
        break;
}
```

从这个例子中就可以大致了解一个解释器在解释执行字节码时的过程，其实很简单，就是通过一个字节码和解释函数的关系来调用相应的函数，或者通过一个很长的switch来判断每个字节码，并调用相应函数。而解释函数则通过执行相应的操作来模拟出一个指令。最后，把这些指令串联在一起就可以执行完一个完整的逻辑。

0x05 代码运行效果



```
→ vm ./xvm
asdf
error!
→ vm ./xvm
ZPLEF_L[
error!
→ vm ./xvm
ZPJEF_L[
success!
→ vm
```

0x06 虚拟机保护效果

静态分析

在静态分析基于虚拟机保护的代码时，一般的工具都是没有效果的，因为字节码都是我们自己定义的，只有解释器能够识别。所以在用ida分析时字节码只是一段不能识别的数据。

```

6 ; -----
6
6 public target_func
6 target_func:
6     push    rbp
7     mov     rbp, rsp
A
A loc_40069A:
A     mov     al, ds:1211A00000000010h ; DATA XREF: vm_interp+1010
A
A ; -----
3     db 3 dup(0), 0A4h, 0A0h
8 ; -----
8     adc     al, 0
8 ; -----
A     db 0
B     db 0
C     db 0
D     db 0A0h ; ?
E     db 11h
F     db 29h ; )
0     db 0
1     db 0
2     db 0
3     db 0A1h ; ?
4     db 0A2h ; ?
5     db 20h
6     db 0A6h ; ?
7     db 5Bh ; [
8     db 0A0h ; ?
9     db 14h
A     db 1
B     db 0
C     db 0
D     db 0
E     db 0A1h ; ?
F     db 0A2h ; ?
0     db 21h ; !
1     db 0A6h ; ?
2     db 50h ; P
3     db 0A0h ; ?
4     db 14h
5     db 2
6     db 0
7     db 0
8     db 0
9     db 0A1h ; ?

```

drops.wooyun.org

这就是ida识别到的target_func的代码，已经做到了对抗静态分析。不过还是可以静态分析我们的解释器,在分析解释器的时候，解释器中的控制流会比源程序的控制流复杂很多，这样也是会增加分析难度。

动态调试

在动态调试的时候字节码依然不能被识别，而且处理器也不会真正的去执行这些不被识别的东西。因为这些字节码都是被我们的虚拟处理器通过解释器执行的，而我们的解释器都是native指令，所以可以静态分析，也可以动态调试。但是动态调试的时候只是在调试解释器，在调试过程中只能看到在不断的调用各个指令的解释函数。所以想要真正还原出源码就需要在调试过程中找到所有字节码对应的native指令的映射关系，最后，通过这个映射关系来把字节码转换成native指令，当然也可以修复出一个完全脱壳并且可以执行的native程序，只是过程会比较繁琐。

0x07 完整代码

以下是demo的完整代码，已经在linux中测试通过。

xvm.h

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define OPCODE_NUM 7 // opcode number
#define HEAP_SIZE_MAX 1024

char *heap_buf; // vm heap

/*
 * opcode enum
 */
enum OPCODES
{
    MOV = 0xa0, // mov 指令字节码对应 0xa0
    XOR = 0xa1, // xor 指令字节码对应 0xa1
    CMP = 0xa2, // cmp 指令字节码对应 0xa2
    RET = 0xa3, // ret 指令字节码对应 0xa3
    SYS_READ = 0xa4, // read 系统调用字节码对应 0xa4
    SYS_WRITE = 0xa5, // write 系统调用字节码对应 0xa5
    JNZ = 0xa6 // jnz 指令字节码对应 0xa6
};

enum REGISTERS
{
    R1 = 0x10,
    R2 = 0x11,
    R3 = 0x12,

```

```

R4 = 0x13,
EIP = 0x14,
FLAG = 0x15
};

/*
 * opcode struct
 */
typedef struct opcode_t
{
    unsigned char opcode; // 字节码
    void (*func)(void *); // 与字节码对应的处理函数
} vm_opcode;

/*
 * virtual processor
 */
typedef struct processor_t
{
    int r1; // 虚拟寄存器r1
    int r2; // 虚拟寄存器r2
    int r3; // 虚拟寄存器r3
    int r4; // 虚拟寄存器r4

    int flag; // 虚拟标志寄存器flag，作用类似于eflags

    unsigned char *eip; // 虚拟机寄存器eip，指向正在解释的字节码地址

    vm_opcode op_table[OPCODE_NUM]; // 字节码列表，存放了所有字节码与对应的处理函数
} vm_processor;
0

xvm.c

#include "xvm.h"

void target_func()
{
    __asm__ __volatile__ (".byte 0xa0, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0xa0,
0x11, 0x12, 0x00, 0x00, 0x00, 0xa4, 0xa0, 0x14, 0x00, 0x00, 0x00, 0x00,
0xa0, 0x11, 0x29, 0x00, 0x00, 0x00, 0xa1, 0xa2, 0x20, 0xa6, 0x5b, 0xa0,
0x14, 0x01, 0x00, 0x00, 0x00, 0xa1, 0xa2, 0x21, 0xa6, 0x50, 0xa0, 0x14,
0x02, 0x00, 0x00, 0x00, 0xa1, 0xa2, 0x22, 0xa6, 0x45, 0xa0, 0x14, 0x03,
0x00, 0x00, 0x00, 0xa1, 0xa2, 0x23, 0xa6, 0x3a, 0xa0, 0x14, 0x04, 0x00,
0x00, 0x00, 0xa1, 0xa2, 0x24, 0xa6, 0x2f, 0xa0, 0x14, 0x05, 0x00, 0x00,
0x00, 0xa1, 0xa2, 0x25, 0xa6, 0x24, 0xa0, 0x14, 0x06, 0x00, 0x00, 0x00,
0xa1, 0xa2, 0x26, 0xa6, 0x19, 0xa0, 0x14, 0x07, 0x00, 0x00, 0x00, 0xa1,
0xa2, 0x27, 0xa6, 0x0f, 0xa0, 0x10, 0x30, 0x00, 0x00, 0x00, 0xa0, 0x11,
0x09, 0x00, 0x00, 0x00, 0xa5, 0xa3, 0xa0, 0x10, 0x40, 0x00, 0x00, 0x00,
0xa0, 0x11, 0x07, 0x00, 0x00, 0x00, 0xa5, 0xa3");

    /*
     mov r1, 0x00000000
     mov r2, 0x12
     call vm_read    ; 输入

     mov r1, input[0]
     mov r2, 0x29
     xor r1, r2      ; 异或
     cmp r1, flag[0] ; 比较
     jnz ERROR       ; 如果不相同就跳转到输出错误的代码

     ; 同上
     mov r1, input[1]
     xor r1, r2
     cmp r1, flag[1]
     jnz ERROR

     mov r1, input[2]
     xor r1, r2
     cmp r1, flag[2]
     jnz ERROR

     mov r1, input[3]
     xor r1, r2
     cmp r1, flag[3]
     jnz ERROR

     mov r1, input[4]
     xor r1, r2
     cmp r1, flag[4]
     jnz ERROR

     mov r1, input[5]
     xor r1, r2
     cmp r1, flag[5]
     jnz ERROR

     mov r1, input[6]
     xor r1, r2
     cmp r1, flag[6]
     jnz ERROR

     mov r1, input[7]
     xor r1, r2
     cmp r1, flag[7]
     jnz ERROR
    */
}

```

```

/*
 * xor 指令解释函数
 */
void vm_xor(vm_processor *proc)
{
    // 异或的两个数据分别存放在r1,r2寄存器中
    int arg1 = proc->r1;
    int arg2 = proc->r2;

    // 异或结果存在r1中
    proc->r1 = arg1 ^ arg2;

    // xor指令只占一个字节，所以解释后，eip向后移动一个字节
    proc->eip += 1;
}

/*
 * cmp 指令解释函数
 */
void vm_cmp(vm_processor *proc)
{
    // 比较的两个数据分别存放在r1和buffer中
    int arg1 = proc->r1; \
    // 字节码中包含了buffer的偏移
    char *arg2 = *(proc->eip + 1) + heap_buf;

    // 比较并对flag寄存器置位，1为相等，0为不等
    if (arg1 == *arg2) {
        proc->flag = 1;
    } else {
        proc->flag = 0;
    }

    // cmp指令占两个字节，eip向后移动2个字节
    proc->eip += 2;
}

/*
 * jnz 指令解释函数
 */
void vm_jnz(vm_processor *proc)
{
    // 获取字节码中需要的地址相距eip当前地址的偏移
    unsigned char arg1 = *(proc->eip + 1);

    // 通过比较flag的值来判断之前指令的结果，如果flag为零说明之前指令不想等，
    // jnz跳转实现
    if (proc->flag == 0) {
        // 跳转可以直接修改eip，偏移就是上面获取到的偏移
        proc->eip += arg1;
    } else {
        proc->flag = 0;
    }

    // jnz 指令占2个字节，所以eip向后移动两个字节
    proc->eip += 2;
}

/*
 * ret 指令解释函数
 */
void vm_ret(vm_processor *proc)
{
}

/*
 * read 系统调用解释函数
 */
void vm_read(vm_processor *proc)
{
    // read系统调用有两个参数，分别存放在r1,r2寄存器中，r1中是保存读入数据的buf
    // 的偏移，r2为希望读入的长度
    char *arg2 = heap_buf + proc->r1;
    int arg3 = proc->r2;

    // 直接调用read
    read(0, arg2, arg3);

    // read系统调用占1个字节，所以eip向后移动1个字节
    proc->eip += 1;
}

/*
 * write 系统调用解释函数
 */
void vm_write(vm_processor *proc)
{
    // 与read系统调用相同，r1中是保存写出数据的buf的偏移，r2为希望写出的长度
    char *arg2 = heap_buf + proc->r1;
    int arg3 = proc->r2;

    // 直接调用write
    write(1, arg2, arg3);

    // write系统调用占1个字节，所以eip向后移动1个字节
    proc->eip += 1;
}

/*
 * mov 指令解释函数
 */
void vm_mov(vm_processor *proc)
{
    // 与read系统调用相同，r1中是保存写出数据的buf的偏移，r2为希望写出的长度
    char *arg2 = heap_buf + proc->r1;
    int arg3 = proc->r2;

    // 直接调用write
    write(1, arg2, arg3);

    // write系统调用占1个字节，所以eip向后移动1个字节
    proc->eip += 1;
}

```



```

// mov 指令两个参数都隐含在字节码中了，指令标识后的第一个字节是寄存器的标识，指令标识后的第二到第五个字节是要mov的立即数，目前只实现了mov一个立即数到一个寄存器中和mov一个buffer中的内容到一个r1寄存器
unsigned char *dest = proc->eip + 1;
int *src = (int *) (proc->eip + 2);

// 前4个case分别对应r1~r4，最后一个case中，*src保存的是buffer的一个偏移，实现了把buffer中的一个字节赋值给r1
switch (*dest) {
    case 0x10:
        proc->r1 = *src;
        break;

    case 0x11:
        proc->r2 = *src;
        break;

    case 0x12:
        proc->r3 = *src;
        break;

    case 0x13:
        proc->r4 = *src;
        break;

    case 0x14:
        proc->r1 = *(heap_buf + *src);
        break;
}

// mov指令占6个字节，所以eip向后移动6个字节
proc->eip += 6;
}

/*
 * 执行字节码
 */
void exec_opcode(vm_processor *proc)
{
    int flag = 0;
    int i = 0;

    // 查找eip指向的正在解释的字节码对应的处理函数
    while (!flag && i < OPCODE_NUM) {
        if (*proc->eip == proc->op_table[i].opcode) {
            flag = 1;
            // 查找到之后，调用本条指令的处理函数，由处理函数来解释
            proc->op_table[i].func((void *) proc);
        } else {
            i++;
        }
    }
}

/*
 * 虚拟机的解释器
 */
void vm_interp(vm_processor *proc)
{
    // eip指向被保护代码的第一个字节
    // * target_func + 4是为了跳过编译器生成的函数入口的代码
    proc->eip = (unsigned char *) target_func + 4;

    // 循环判断eip指向的字节码是否为返回指令，如果不是就调用exec_opcode来解释执行
    while (*proc->eip != RET) {
        exec_opcode(proc);
    }
}

/*
 * 初始化虚拟机处理器
 */
void init_vm_proc(vm_processor *proc)
{
    proc->r1 = 0;
    proc->r2 = 0;
    proc->r3 = 0;
    proc->r4 = 0;
    proc->flag = 0;

    // 把指令字节码与解释函数关联起来
    proc->op_table[0].opcode = MOV;
    proc->op_table[0].func = (void (*)(void *)) vm_mov;

    proc->op_table[1].opcode = XOR;
    proc->op_table[1].func = (void (*)(void *)) vm_xor;

    proc->op_table[2].opcode = CMP;
    proc->op_table[2].func = (void (*)(void *)) vm_cmp;

    proc->op_table[3].opcode = SYS_READ;
    proc->op_table[3].func = (void (*)(void *)) vm_read;

    proc->op_table[4].opcode = SYS_WRITE;
    proc->op_table[4].func = (void (*)(void *)) vm_write;

    proc->op_table[5].opcode = RET;
    proc->op_table[5].func = (void (*)(void *)) vm_ret;

    proc->op_table[6].opcode = JNZ;
    proc->op_table[6].func = (void (*)(void *)) vm_jnz;
}

```



ew
se
nd)



/w
p-
log
in.
ph
p?

act
ion
=lo
go
ut&
red
ire
ct_
to=
htt
p
%3
A
%2
F
%2
Fdr
op
s.w
oo
yu
n.o
rg)

```
// 创建buffer
heap_buf = (char *) malloc(HEAP_SIZE_MAX);

// 初始化buffer
memcpy(heap_buf + 0x20, "syclover", 8);
memcpy(heap_buf + 0x30, "success!\n", 9);
memcpy(heap_buf + 0x40, "error!\n", 7);
}

// flag: ZPJEF_L[
int main()
{
    vm_processor proc = {0};

    // initial vm processor
    init_vm_proc(&proc);

    // execute target func
    vm_interp(&proc);
    return 0;
}
8
```

0x08 总结

以上程序为学习代码虚拟化之后的总结，其中有很多理解不正确的地方希望大牛指正。这只是最简单的实现，仅用于学习使用，想要深入学习虚拟化技术还是非常复杂，需要积累更多知识才能理解到位，这篇文章就当是抛砖引玉。在学习的过程也有很多问题没有解决，比如：如果想实现一个基于虚拟机的保护壳，必定需要把源程序中的native指令首先转换为自定义字节码，但是不知道用什么方法来转换比较好。

在很多国外文章里也看到另一种虚拟机保护，是基于LLVM-IR的虚拟机保护，有兴趣也可以继续深入研究一下。

0x09 参考

<http://www.cs.rhul.ac.uk/home/kinder/papers/wcre12.pdf>
(<http://www.cs.rhul.ac.uk/home/kinder/papers/wcre12.pdf>)

☆收藏 分享



写下你的评论...

发表

-  Wooooo 2015-11-20 00:27:21
膜拜。

回复
-  厕所打灯笼 2015-11-19 23:46:46
@cccbbbg 是可以的，所以解释器通常也会配合其他保护技术来增加分析难度

回复
-  cccbbbg 2015-11-19 22:48:46
感谢分享。 问个问题，解释器本身可能被分析出来？

回复
-  Ly1 2015-11-19 21:52:33
很详细，不错

回复
-  Jioun_dai 2015-11-19 18:20:41



厉害

回复



切尸 2015-11-19 18:05:45
你真棒你真棒你真棒你真棒

回复



Muhe 2015-11-19 18:00:53
膜拜

回复



牛肉包子 2015-11-19 17:24:43
前排膜拜

回复

感谢知乎授权页面模版