**VMProtect, Part 0: Basics**

**Author:**[RolfRolles](http://www.openrce.org/profile/view/RolfRolles) [http://www.openrce.org/img/email_icon.gif](http://www.openrce.org/messages/compose/RolfRolles)

Translator:千里之外

[VMProtect](http://www.vmprotect.ru/) is a virtualization protector. Like other protections in the genre, among others [ReWolf's x86 Virtualizer](http://www.openrce.org/blog/view/847/x86_Virtualizer_-_source_code) and [CodeVirtualizer](http://www.oreans.com/codevirtualizer.php), it works by disassembling the x86 bytecode of the target executable and compiling it into a proprietary, polymorphic bytecode which is executed in a custom interpreter at run-time. This is unlike the traditional notions of packing, in which the x86 bytecode is simply encrypted and/or compressed:  with virtualization, the original x86 bytecode in the protected areas is gone, never to be seen again.  Or so the idea goes.

译文:

[VMProtect](http://www.vmprotect.ru/) 是一个虚拟化的保护器.像其他这种类型的保护器[ReWolf's x86 Virtualizer](http://www.openrce.org/blog/view/847/x86_Virtualizer_-_source_code) 和 [CodeVirtualizer](http://www.oreans.com/codevirtualizer.php) 一样,它反汇编目标可执行程序的X86字节码并把它编译为一个可以在运行时通过一个通用的解释器执行的私有的多态的字节码.这种做法不同于传统的只是将X86字节码简单的加密and/or 压缩的打包观念:运用虚拟化技术,在保护区域的原始的X86字节码消失了,再也看不到了.或者这样的想法也没了.

If you've never looked at VMProtect before, I encourage you to take a five-minute look in IDA ([here's](http://www.tuts4you.com/download.php?view.1601) a sample packed binary).  As far as VMs go, it is particularly skeletal and easily comprehended.  The difficulty lies in recreating working x86 bytecode from the VM bytecode.  Here's a two-minute analysis of its dispatcher.  
译文:

如果你以前从没看过VMProtect,我鼓励你花5分钟看看IDA([这里](http://www.tuts4you.com/download.php?view.1601)是一个二进制打包的例子).直到VMs运行前,它是特别简单和容易理解的.难点在于从VM的字节码中重建X86字节码.这里是它的派遣函数的2分钟的分析.

  push  edi                        ; push all registers 压入所有的寄存器  
  push  ecx  
  push  edx  
  push  esi  
  push  ebp  
  push  ebx  
  push  eax  
  push  edx  
  pushf  
  push  0                          ; imagebase fixup 映像基址修正  
  mov   esi, [esp+8+arg\_0]         ; esi = pointer to VM bytecode esi = VM字节码指针  
  mov   ebp, esp                   ; ebp = VM's "stack" pointer ebp = VM 的 “栈”指针  
  sub   esp, 0C0h  
  mov   edi, esp                   ; edi = "scratch" data area edi = “刻痕”数据区  
  
VM\_\_FOLLOW\_\_Update:  
  add   esi, [ebp+0]  
  
VM\_\_FOLLOW\_\_Regular:  
  mov   al, [esi]                  ; read a byte from ESI 从ESI中读取一个字节  
  movzx eax, al  
  sub   esi, -1                    ; increment ESI 增加ESI  
  jmp   ds:VM\_\_HandlerTable[eax\*4] ; execute instruction handler 执行指令处理程序  
  
A feature worth discussing is the "scratch space", referenced by the register edi throughout the dispatch loop.  This is a 16-dword-sized area on the stack where VMProtect saves the registers upon entering the VM, modifies them throughout the course of a basic block, and from whence it restores the registers upon exit.  For each basic block protected by the VM, the layout of the registers in the scratch space can potentially be different.  
译文:

一个值得讨论的特征是被EDI寄存器引用并且贯穿派遣循环的”刻痕区”.这是一个栈上16个DOWRD大小的区域,在进入Vm后VMProtect在这里保存了寄存器,并且在贯穿基本块的过程中都会修改它们,在退出的时候从这里恢复这些寄存器.每一个被VM保护的基本块,在刻痕区域中的寄存器的布局可能不同.

Here's a disassembly of some instruction handlers.  Notice that A) VMProtect is a stack machine and that B) each handler -- though consisting of scant few instructions -- performs several tasks, e.g. popping several values, performing multiple operations, pushing one or more values.

译文:

这里是一些指令处理程序的反汇编.注意A) VMProtect是一个栈机器B)每一个指令处理程序—尽管只包含了不足几个指令—却执行各自的任务,例如弹出几个数,执行乘法操作,压入一个或更多的数.  
  
#00:  x = [ESI-1] & 0x3C; y = popd; [edi+x] = y  
  
.text:00427251   and   al, 3Ch        ; al = instruction number al = 指令数  
.text:00427254   mov   edx, [ebp+0]   ; grab a dword off the stack 从栈中抢一个dword  
.text:00427257   add   ebp, 4         ; pop the stack 弹出栈  
.text:0042725A   mov   [edi+eax], edx ; store the dword in the scratch space  
 在刻痕区域中存储一个dword  
#01:  x = [ESI-1] & 0x3C; y = [edi+x]; pushd y  
  
.vmp0:0046B0EB   and   al, 3Ch        ; al = instruction number al = 指令数  
.vmp0:0046B0EE   mov   edx, [edi+eax] ; grab a dword out of the scratch space

从刻痕区域中抢一个dword  
.vmp0:0046B0F1   sub   ebp, 4         ; subtract 4 from the stack pointer

从栈指针中减去4  
.vmp0:0046B0F4   mov   [ebp+0], edx   ; push the dword onto the stack  
 把一个dword压入栈中  
#02:  x = popw, y = popw, z = x + y, pushw z, pushf  
  
.text:004271FB   mov   ax, [ebp+0]       ; pop a word off the stack 从栈中弹出一个字  
.text:004271FF   sub   ebp, 2  
.text:00427202   add   [ebp+4], ax       ; add it to another word on the stack

把它加到栈上的另一个字  
.text:00427206   pushf                  
.text:00427207   pop   dword ptr [ebp+0] ; push the flags 压入标志位  
  
#03:  x = [ESI++]; w = popw; [edi+x] = Byte(w)  
  
.vmp0:0046B02A   movzx eax, byte ptr [esi] ; read a byte from ESI 从ESI中读一个字节  
.vmp0:0046B02D   mov   dx, [ebp+0]         ; pop a word off the stack从栈上弹出一个字  
.vmp0:0046B031   inc   esi                 ; ESI++  
.vmp0:0046B032   add   ebp, 2              ; adjust stack pointer 调整栈指针  
.vmp0:0046B035   mov   [edi+eax], dl       ; write a byte into the scratch area  
 在刻痕区域内写入一个字节  
#04:  x = popd, y = popb, z = x << y, pushd z, pushf  
  
.vmp0:0046B095   mov   eax, [ebp+0]      ; pop a dword off the stack

从栈上弹出一个DWORD  
.vmp0:0046B098   mov   cl, [ebp+4]       ; pop a byte off the stack

从栈上弹出一个字节  
.vmp0:0046B09B   sub   ebp, 2  
.vmp0:0046B09E   shr   eax, cl           ; shr the dword by the byte

dword算数右移 byte 位  
.vmp0:0046B0A0   mov   [ebp+4], eax      ; push the result 压入结果  
.vmp0:0046B0A3   pushf  
.vmp0:0046B0A4   pop   dword ptr [ebp+0] ; push the flags 压入标志位  
  
#05:  x = popd, pushd ss:[x]  
  
.vmp0:0046B5F7   mov   eax, [ebp+0]  ; pop a dword off the stack

从栈中弹出一个dword  
.vmp0:0046B5FA   mov   eax, ss:[eax] ; read a dword from ss

从SS中读一个dword  
.vmp0:0046B5FD   mov   [ebp+0], eax  ; push that dword 压入那个dword

**Part 1: Bytecode and IR**

The [approach I took with ReWolf's x86 Virtualizer](http://www.openrce.org/blog/view/1110/Compiler_1,_X86_Virtualizer_0) is also applicable here, although a more sophisticated compiler is required.  What follows is some preliminary notes on the design and implementation of such a component.  These are not complete details on breaking the protection; I confess to having only looked at a few samples, and I am not sure which protection options were enabled.  
译文:

尽管需要一个更复杂的编译器,但是我[处理ReWolf's x86虚拟器的方法](http://www.openrce.org/blog/view/1110/Compiler_1,_X86_Virtualizer_0)在这里也适用.

接下来是设计和实现这样一个组件的初步说明.这里没有打破这个保护的完整细节;我承认我只看过几个例子,我也不确定这些保护选项已启用.

As before, we begin by constructing a disassembler for the interpreter.  This is immediately problematic, since the bytecode language is polymorphic.  I have created an IDA plugin that automatically constructs OCaml source code for a bytecode disassembler.  In a production-quality implementation, this should be implemented as a standalone component that returns a closure.  
译文:

像从前一样,我们首先为这个解释器构建一个反汇编程序.由于字节码语言是多态的,所以立刻会有一个问题.我已经创建了一个IDA插件用来自动把字节码汇编程序构建为Ocaml源码.为了保证产品的实现质量,它应该作为一个单独的返回一个关闭的组件去实现.

The generated disassembler, then, looks like this:

产生的反汇编程序看起来像这样:  
  
let disassemble bytearray index =  
  match (bytearray.(index) land 0xff) with  
    0x0  -> (VM\_\_Handler0\_\_PopIntoRegister(0),[index+1])  
  | 0x1  -> (VM\_\_Handler1\_\_PushDwordFromRegister(0),[index+1])  
  | 0x2  -> (VM\_\_Handler2\_\_AddWords,[index+1])  
  | 0x3  -> (VM\_\_Handler3\_\_StoreByteIntoRegister(bytearray.(index+1)),[index+2])  
  | 0x4  -> (VM\_\_Handler0\_\_PopIntoRegister(4),[index+1])  
  | 0x5  -> (VM\_\_Handler1\_\_PushDwordFromRegister(4),[index+1])  
  | 0x6  -> (VM\_\_Handler4\_\_ShrDword,[index+1])  
  | 0x7  -> (VM\_\_Handler5\_\_ReadDword\_\_FromStackSegment,[index+1])  
  | ...  -> ...  
    
Were we to work with the instructions individually in their natural granularity, depicted above, the bookkeeping on the semantics of each would likely prove tedious.  For illustration, compare and contrast handlers #02 and #04.  Both have the same basic pattern:  pop two values (words vs. dwords), perform a binary operation (add vs. shr), push the result, then push the flags.  The current representation of instructions does not express these, or any, similarities.

译文:

我们处理的指令分别在它们的自然粒度上,从上面描述来看,在每个语义上的薄记很可能证明是繁琐的.为了方便说明.我们比较和对比下处理例程#02和#04.它们都有相同的基本模型:弹出两个值(words vs. dwords),执行一个二进制操作(add vs. shr),压入结果后加入标志位.当前指令的表示法没有表达这些或者任何相似的地方.

Handler #02:                 Handler #04:  
mov   ax, [ebp+0]           mov   eax, [ebp+0]       
sub   ebp, 2                  mov   cl, [ebp+4]        
add   [ebp+4], ax           sub   ebp, 2             
pushf                       shr   eax, cl            
pop   dword ptr [ebp+0]     mov   [ebp+4], eax       
                             pushf                    
                             pop   dword ptr [ebp+0]

Therefore, we pull a standard compiler-writer's trick and translate the VMProtect instructions into a simpler, "intermediate" language (hereinafter "IR") which resembles the pseudocode snippets atop the handlers in part zero.  Below is a fragment of that language's abstract syntax.

译文:

因此,我们耍一个标准编译器作者的花招,把VMProtect指令翻译为一种更简单的”中间”语言(下文中称"IR"),类似我们第一章中处理程序上面的伪代码片断.下面是那种语言语法摘要的片断.  
  
type size = B | W | D | Q  
type temp = int \* size  
type seg  = Scratch | SS | FS | Regular  
type irbinop  = Add | And | Shl | Shr | MakeQword  
type irunop  = Neg | MakeByte | TakeHighDword | Flags  
type irexpr = Reg of register   
            | Temp of int   
            | Const of const   
            | Deref of seg \* irexpr \* size   
            | Binop of irexpr \* irbinop \* irexpr   
            | Unop of irexpr \* irunop  
              
type ir =   
  DeclareTemps of temp list  
| Assign of irexpr \* irexpr  
| Push of irexpr  
| Pop of irexpr  
| Return  
              
A portion of the VMProtect -> IR translator follows; compare the translation for handlers #02 and #04.  
  
let make\_microcode = function  
  VM\_\_Handler0\_\_PopIntoRegister(b)       ->

[Pop(Deref(Scratch, Const(Dword(zero\_extend\_byte\_dword(b land 0x3C))), D))]  
| VM\_\_Handler2\_\_AddWords                 -> [DeclareTemps([(0, W);(1, W);(2, W)]);  
                                             Pop(Temp(0));  
                                             Pop(Temp(1));  
                                             Assign(Temp(2), Binop(Temp(0), Add, Temp(1)));  
                                             Push(Temp(2));  
                                             Push(Unop(Temp(2), Flags))]  
| VM\_\_Handler4\_\_ShrDword                 -> [DeclareTemps([(0, D);(1, W);(2, D)]);  
                                             Pop(Temp(0));  
                                             Pop(Temp(1));  
                                             Assign(Temp(2), Binop(Temp(0), Shr, Temp(1)));  
                                             Push(Temp(2));  
                                             Push(Unop(Temp(2), Flags))]  
| VM\_\_Handler7\_\_PushESP                  -> [Push(Reg(Esp))]  
| VM\_\_Handler23\_\_WriteDwordIntoFSSegment -> [DeclareTemps([(0, D);(1, D)]);  
                                             Pop(Temp(0));  
                                             Pop(Temp(1));  
                                             Assign(Deref(FS, Temp(0), D), Temp(1))]  
| ...                                    -> ...  
  
To summarize the process, below is a listing of VMProtect instructions, followed by the assembly code that is executed for each, and to the right is the IR translation.  
译文:

为了总结过程,下面是VMProtect指令的一个列表,跟随在每个汇编执行代码右边的是IR翻译.

VM\_\_Handler1\_\_PushDwordFromRegister 32  
  
and   al, 3Ch ; al = 32  
mov   edx, [edi+eax]  
sub   ebp, 4  
mov   [ebp+0], edx        Push (Deref (Scratch, Const (Dword 32l), D));  
  
VM\_\_Handler7\_\_PushESP  
  
mov   eax, ebp  
sub   ebp, 4  
mov   [ebp+0], eax        Push (Reg Esp);  
  
VM\_\_Handler0\_\_PopIntoRegister 40  
  
and   al, 3Ch  
mov   edx, [ebp+0]  
add   ebp, 4  
mov   [edi+eax], edx      Pop (Deref (Scratch, Const (Dword 40l), D));  
  
VM\_\_Handler19\_\_PushSignedByteAsDword (-1l)  
  
movzx eax, byte ptr [esi] ; \*esi = -1  
sub   esi, 0FFFFFFFFh  
cbw  
cwde  
sub   ebp, 4  
mov   [ebp+0], eax        Push (Const (Dword (-1l)));   
  
VM\_\_Handler9\_\_PushDword 4525664l  
  
mov   eax, [esi] ; \*esi = 4525664l  
add   esi, 4  
sub   ebp, 4  
mov   [ebp+0], eax        Push (Const (Dword 4525664l));  
  
VM\_\_Handler9\_\_PushDword 4362952l};  
  
mov   eax, [esi] ; \*esi = 4362952l  
add   esi, 4  
sub   ebp, 4  
mov   [ebp+0], eax        Push (Const (Dword 4362952l));  
  
VM\_\_Handler19\_\_PushSignedByteAsDword 0l};  
  
movzx eax, byte ptr [esi] ; \*esi = 0  
sub   esi, 0FFFFFFFFh  
cbw  
cwde  
sub   ebp, 4  
mov   [ebp+0], eax        Push (Const (Dword (0l)));   
  
VM\_\_Handler42\_\_ReadDwordFromFSSegment};  
  
mov   eax, [ebp+0]        DeclareTemps([(0,D)]);  Pop (Temp 0);  
mov   eax, fs:[eax]  
mov   [ebp+0], eax        Push (Deref (FS, Temp 0, D));

**Part 2:****Introduction to** **Optimization**

|  |  |
| --- | --- |
| **Author:**[RolfRolles](http://www.openrce.org/profile/view/RolfRolles) [http://www.openrce.org/img/email_icon.gif](http://www.openrce.org/messages/compose/RolfRolles) | **# Views:**5385 |

Basically, VMProtect bytecode and the IR differ from x86 assembly language in four ways:  
  
1)  It's a stack machine;  
2)  The IR contains "temporary variables";  
3)  It contains what I've called a "scratch" area, upon which computations are performed rather than in the registers;  
4)  In the case of VMProtect Ultra, it's obfuscated (or rather, "de-optimized") in certain ways.  
译文:

基本上, VMProtect字节码和IR相比X86汇编语言有4方面不同:

1. 它是一个栈机器;
2. IR包含”临时变量”;
3. 它包含一个我称为”刻痕”的区域来执行计算而不是在寄存器中;
4. 在VMProtect Ultra版本中,在某些方面它是混淆的(更精确的说是”未优化的”).

It turns out that removing these four aspects from the IR is sufficient preparation for compilation into sensible x86 code.  We accomplish this via standard compiler optimizations applied locally to each basic block.  In general, there are a few main compiler optimizations used in this process.  The first one is "constant propagation".  Consider the following C code.

译文:

事实证明从IR中移出这4个方面是把它编译为明显的X86代码的充分准备.我们通过在每个基本块应用标准编译器优化来实现它.通常,在这个过程中有几个主要的编译器优化被使用.第一个是”常量传播”.研究下面的C代码.

int x = 1;  
function(x);  
  
Clearly x will always be 1 when function is invoked:  it is defined on the first line, and is not re-defined before it is used in the following line (the definition in line one "reaches" the use in line two; alternatively, the path between the two is "definition-clear" with respect to x).  Thus, the code can be safely transformed into "function(1)".  If the first line is the only definition of the variable x, then we can replace all uses of x with the integer 1.  If the second line is the only use of the variable x, then we can eliminate the variable.  
译文:

显然在函数被调用的时候X总是1.它在第一行被定义,在接下来的行使用它前没有被重新定义(第一行的定义”到达”第二行的使用;另外,对于X来说两者之间的路径是”定义清晰的”).因此,代码可以被安全的转换为”function(1)”.如果第一行是变量X的唯一定义,那么我们可以把所有的X用数字1替换.如果第二行是唯一使用变量X的地方,

那么我们可以消除这个变量.

The next is "constant folding".  Consider the following C code.

下一个是”常量折叠”. 研究下面的C代码.

int x = 1024;  
function(x\*1024);  
  
By the above remarks, we know we can transform the second line into "function(1024\*1024);".  It would be silly to generate code that actually performed this multiplication at run-time:  the value is known at compile-time, and should be computed then.  We can replace the second line with "function(1048576);", and in general we can replace any binary operation performed upon constant values with the computed result of that operation.  
译文:

通过上述言论,我们知道我们可以把第二行转换为"function(1024\*1024);"生成这种在运行时执行乘法运算的代码是愚蠢的:这个值在编译时刻是已知的,然后应该被计算出来.我们可以用"function(1048576);"代替第二行,一般而言我们可以用运算结果代替任何根据常数的二进制运算.

Similar to constant propagation is "copy propagation", as in the below.  
和常量传播相似的是拷贝传播,如下面的.  
void f(int i)  
{  
  int j = i;  
  g(j);  
}  
  
The variable j is merely a copy of the variable i, and so the variable i can be substituted in for j until the point where either is redefined.  Lacking redefinitions, j can be eliminated entirely.  
译文:

变量j只是变量 i的一个拷贝,因此变量i可以 取代j直到任一个的指针被重新定义.缺乏重新定义的话,j可以完全的消除.  
The next optimization is "dead-store elimination".  Consider the following C code.  
下一个优化是”死-存储 消除”. 研究下面的C代码.

int y = 2;  
y = 1;  
  
The definition of the variable y on line one is immediately un-done by the one on line two.  Therefore, there is no reason to actually assign the value 2 to the variable; the first store to y is a "dead store", and can be eliminated by the standard liveness-based compiler optimization known as "dead-store elimination", or more generally "dead-code elimination".  Here's an example from the VMProtect IR.  
译文:

第一行变量y的定义立刻被第二行取代.因此,没有理由实际上分配2给这个变量;y的第一个存储是一个”死存储”,这个可以被标准的活性为基础的编译器优化当作” 死-存储 消除”,或者当作更一般的”死-代码 消除”来消除.下面是来自VMProtect IR 的一个例子.

ecx = DWORD Scratch:[Dword(44)]  
ecx = DWORD Scratch:[Dword(20)]  
  
After dead-store-eliminating the first instruction, it turns out no other instructions use Scratch:[Dword(44)], and so its previous definition can be eliminated as well.

第一个指令经过死-存储-消除后,事实证明没有任何指令利用Scratch:[Dword(44)],所以它先前的定义也可以被消除.

|  |  |  |
| --- | --- | --- |
| **Part 3:** **Optimizing and Compiling**   |  |  | | --- | --- | | **Author:**[RolfRolles](http://www.openrce.org/profile/view/RolfRolles) [http://www.openrce.org/img/email_icon.gif](http://www.openrce.org/messages/compose/RolfRolles) | **# Views:**7549 |   The reader should inspect [this unoptimized IR listing](https://www.openrce.org/repositories/users/RolfRolles/Unoptimized.txt) before continuing.  In an attempt to keep this entry from becoming unnecessarily long, the example snippets will be small, but for completeness a more thorough running example is linked throughout the text.  译文:  在继续进行之前,读者需要检查下[这个未优化的IR清单](https://www.openrce.org/repositories/users/RolfRolles/Unoptimized.txt).为了不使这个项目变成不必要的长度,例子的片断会是小的,但一个更为彻底的完整的例子是整个文本链接.  We begin by removing the stack machine features of the IR.  Since VMProtect operates on disassembled x86 code, and x86 itself is not a stack machine, this aspect of the protection is unnatural and easily removed.  Here is a 15-line fragment of VMProtect IR.  译文:  我们从删除IR的栈机器特征开始.由于VMProtect运行在X86代码反汇编上,而且X86自身不是一个栈机器,所以这方面的保护是不自然的而且很容易删除.这里是VMProtect IR 的15行片断.  push Dword(-88) push esp push Dword(4) pop t3 pop t4 t5 = t3 + t4 push t5 push flags t5 pop DWORD Scratch:[Dword(52)] pop t6 pop t7 t8 = t6 + t7 push t8 push flags t8 pop DWORD Scratch:[Dword(12)] pop esp  All but two instructions are pushes or pops, and the pushes can be easily matched up with the pops.  Tracking the stack pointer, we see that, for example, t3 = Dword(4).  A simple analysis allows us to "optimize away" the push/pop pairs into assignment statements.  Simply iterate through each instruction in a basic block and keep a stack describing the source of each push.  For every pop, ensure that the sizes match and record the location of the corresponding push.  We wish to replace the pop with an assignment to the popped expression from the pushed expression, as in 译文:  差不多都是pushes 或者pops 两条指令,而且pushes很容易和pops匹配.跟踪栈指针,我们注意到,例如, t3 = Dword(4).一个简单的分析允许我们用赋值语句”优化掉” push/pop对.简单的在一个基本块中遍历每个指令,并保持一个堆栈描述每个push源.每一个pop,确保大小匹配和记录相应push的位置.我们希望用pushed表达式赋值poped表达式的方法替换pop.例如  t3 = Dword(4) t4 = esp  t7 = Dword(-88)  With the stack aspects removed, we are left with a more conventional listing containing many assignment statements.  This optimization substantially reduces the number of instructions in a given basic block (~40% for the linked example) and opens the door for other optimizations.  The newly optimized code is eight lines, roughly half of the original: 译文:  随着栈问题的解决,我们留下了一个更普通的包含大量赋值语句的列表.这种优化大幅减少了在一个给定的基本块中指令的数量(链接例子用大约是40%)并且打开了其他优化的大门.新代码是8行,大约是源码的一半. t3 = Dword(4) t4 = esp t5 = t3 + t4 DWORD Scratch:[Dword(52)] = flags t5 t6 = t5 t7 = Dword(-88) t8 = t6 + t7 DWORD Scratch:[Dword(12)] = flags t8 esp = t8  A complete listing of the unoptimized IR versus the one with the stack machine features removed is [here](https://www.openrce.org/repositories/users/RolfRolles/Unoptimized-vs-StackRemoved.txt), which should be perused before proceeding.  译文:  一个删除了栈机器特征的未优化IR列表在[这里](https://www.openrce.org/repositories/users/RolfRolles/Unoptimized-vs-StackRemoved.txt),在继续进行前需要细读下.  Now we turn our attention to the temporary variables and the scratch area.  Recall that the former were not part of the pre-protected x86 code, nor the VMProtect bytecode -- they were introduced in order to ease the IR translation.  The latter is part of the VMProtect bytecode, but was not part of the original pre-protected x86 code.  Since these are not part of the languages we are modelling, we shall eliminate them wholesale.  On a high level, we treat each temporary variable, each byte of the scratch space, and each register as being a variable defined within a basic block, and then eliminate the former two via the compiler optimizations previously discussed.  译文:  现在让我们来关注临时变量和刻痕区域,记得前者不是保护前X86代码的一部分,也不是VMProtect字节码—他们被引入用来缓解IR的翻译.后者是VMProtect字节码的一部分,当不俗保护前X86代码的一部分.由于他们不是我们正在建模的语言的一部分,所以我们应该整个消除它们.在高层,我们可以把每一个临时变量,刻痕区的每个字节,每个寄存器当作一个基本块中的变量定义,然后用我们前面讨论的编译器优化来消除前两者.  Looking again at the last snippet of IR, we can see several areas for improvement.  First, consider the variable t6.  It is clearly just a copy of t5, neither of which are redefined before the next use in the assignment to t8.  Copy propagation will replace variable t6 with t5 and eliminate the former.  More generally, t3, t4, and t7 contain either constants or values that are not modified between their uses.  Constant and copy propagation will substitute the assignments to these variables in for their uses and eliminate them. 译文:  再次看最后的IR 代码片断,我们可以看到几处可以改进的地方.首先研究下变量t6.它显然是t5的一个拷贝,在下个使用它们赋值到t8前,它们都没有重新定义.拷贝传播会用t5取代变量t6并且消除前者.更一般的,t3,t4和t7包含常量或者数值在它们使用之间没有被修改.常量和拷贝优化会在他们使用的时候做替换工作并消除它们.  The newly optimized code is a slender three lines compared to the original 15; we have removed 80% of the IR for the running example.  译文:  最新的优化代码相比原来的15行只有瘦小的三行;我们已经为这个运行中的例子移出了80%的IR.  DWORD Scratch:[Dword(52)] = flags Dword(4) + esp esp = Dword(4) + esp + Dword(-88) DWORD Scratch:[Dword(12)] = flags Dword(4) + esp + Dword(-88)  The side-by-side comparison can be found [here](https://www.openrce.org/repositories/users/RolfRolles/StackRemoved-vs-TemporariesRemoved.txt). 并存的比较可以在这里[找到](https://www.openrce.org/repositories/users/RolfRolles/StackRemoved-vs-TemporariesRemoved.txt). The IR now looks closer to x86, with the exception that the results of computations are being stored in the scratch area, not into registers.  As before, we apply dead-store elimination, copy and constant propagation to the scratch area, removing dependence upon it entirely in the process.  See [here](https://www.openrce.org/repositories/users/RolfRolles/TemporariesRemoved-vs-ScratchRemoved.txt)for a comparison with the last phase. 译文:  现在IR看起来很接近x86,一个例外是运算的结果保存在刻痕区域而不是寄存器中.和从前一样,我们在刻痕区域应用死-存储消除,拷贝和常量传播,移除在整个过程中对它的依赖.[这里](https://www.openrce.org/repositories/users/RolfRolles/TemporariesRemoved-vs-ScratchRemoved.txt)是最终结果的比较.  Here is a comparison of the final, optimized code against the original x86:  这里是优化代码和原始X86的最终比较:  push ebp                               push    ebp                     ebp = esp                               mov     ebp, esp                push Dword(-1)                         push    0FFFFFFFFh              push Dword(4525664)                     push    450E60h                 push Dword(4362952)                     push    offset sub\_4292C8       eax = DWORD FS:[Dword(0)]               mov     eax, large fs:0 push eax                               push    eax                     DWORD FS:[Dword(0)] = esp               mov     large fs:0, esp         eflags = flags esp + Dword(-88)                             esp = esp + Dword(-88)                 add     esp, 0FFFFFFA8h         push ebx                               push    ebx                     push esi                               push    esi                     push edi                               push    edi                     DWORD SS:[Dword(-24) + ebp] = esp      mov     [ebp-18h], esp          call DWORD [Dword(4590300)]            call    dword ptr ds:unk\_460ADC vmreturn Dword(0) + Dword(4638392)  Code generation is an afterthought.  代码生成时一种事后的想法. |
|  |

Blog Comments

|  |  |
| --- | --- |
| **http://www.openrce.org/img/flags/il.gif**[**Orr**](http://www.openrce.org/profile/view/Orr)**[http://www.openrce.org/img/email_icon.gif](http://www.openrce.org/messages/compose/Orr)** | **Posted: Sunday, August 10 2008 06:43.41 CDT** |
| Just got the time to read the entire thing. Amazing ;) | |

|  |  |
| --- | --- |
| **http://www.openrce.org/img/flags/us.gif**[**RolfRolles**](http://www.openrce.org/profile/view/RolfRolles)**[http://www.openrce.org/img/email_icon.gif](http://www.openrce.org/messages/compose/RolfRolles)** | **Posted: Wednesday,** **January 28 2009 20:42.33 CST** |
| I submitted a paper to the 2009 IEEE Symposium on Security and Privacy on this topic.  I didn't have time to finish it, so I submitted an incomplete draft and wasn't surprised when it was rejected.  However, I liked this bit of feedback from the anonymous reviewers:    "Looking around the Internet, the paper's main contribution is mostly similar to blog post of Rolf Rolles' blog that talk about manually reverse engineering VMProtect. The authors should include novel work [...]"  :-( | |

|  |  |
| --- | --- |
| **http://www.openrce.org/img/flags/de.gif**[**sp**](http://www.openrce.org/profile/view/sp)**[http://www.openrce.org/img/email_icon.gif](http://www.openrce.org/messages/compose/sp)** | **Posted: Thursday, January 29 2009 10:27.01 CST** |
| That's funny, mate. At least for the uninvolved reader. :) | |