逆向一个简单的虚拟机-教程

Reverser：Maximus

获得指令和寄存器

好，今晚太累了，我下载了很多我喜欢的好歌，该是逆向的时间了。

我听人谈起HyperUnpackMe2这个crackme很多次，于是我找到了它。我打开了我的IDA 4.3-是的，我不用破解的那个…工具而已，只是给那些如果没有就什么都做不了的…

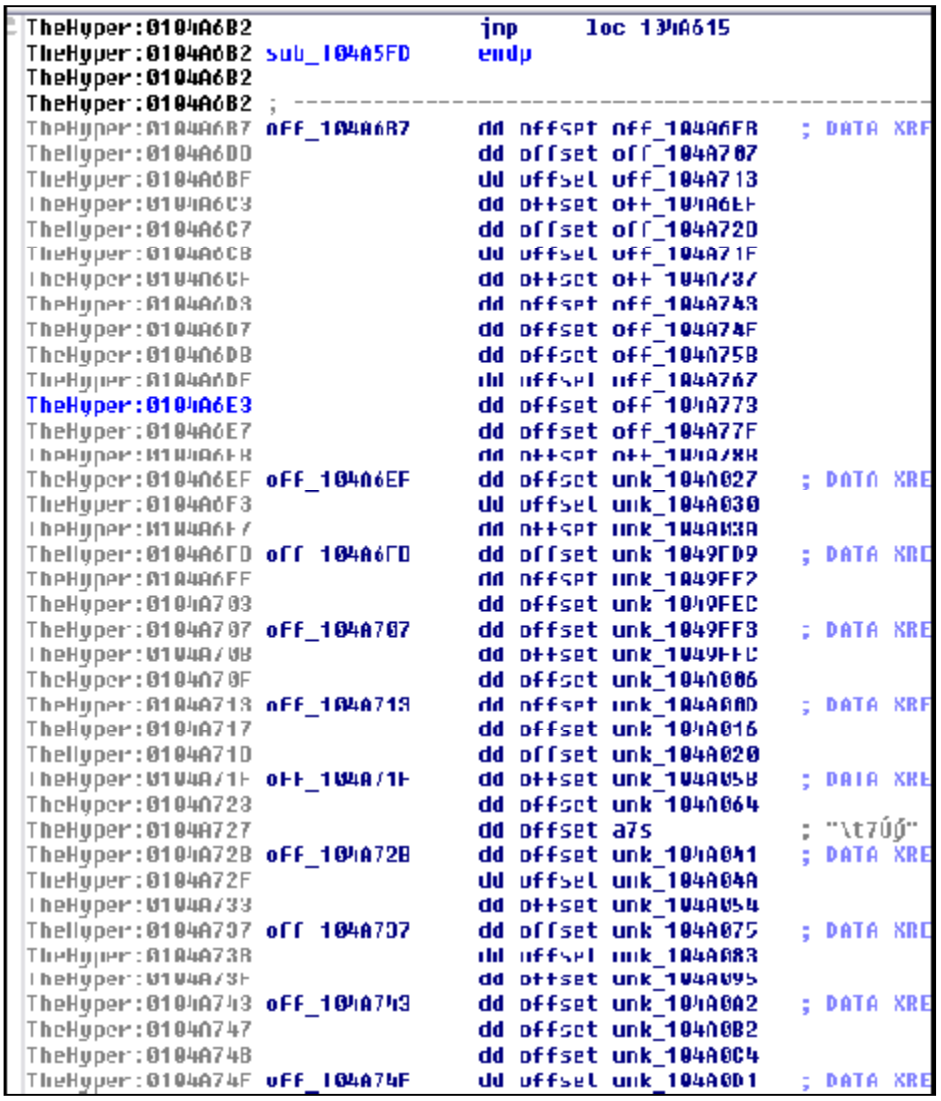
我打开了这个crakeme。它使用了很多丑陋的反IDA的技巧，这需要很多时间使用U快捷键去取消“跳转/函数调用”的指针，然后使用C快捷键，重新定义代码。它隐藏了LoadLibrary和字符串的指针如“VirtualAlloc”。好了，可笑但并不有趣，我要开始看这个虚拟机了。希望它没有加密，否则我不得不使用OLLY解压脱壳直到虚拟机在代码中呈现出来。

那么，我们如何使用IDA4.3在代码中找到一个虚拟机呢？很简单：使用你的滚动条和最古老有效的逆向工具：第六感。

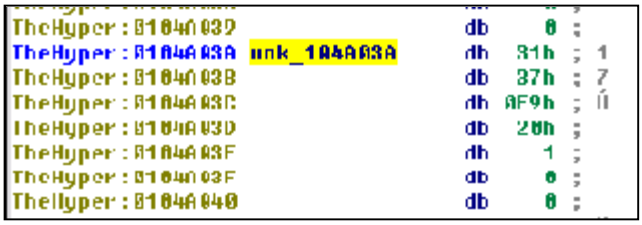
我们到底该找些什么呢？我们的“灵感”点在哪里呢？当我浏览aspr1.2dll的时候，我发现在Push序列之后的RET就是关键所在-它确实是，那么虚拟机呢？一个虚拟机由一个指令仿真器构成，它通常是一个公共的代码循环跳转到一系列函数和地址。在本例中，我们查找指针/函数列表。是的，这种列表可能是别的东西，例如，对象（objects）也是这样放置的。我们如何把他们和虚拟机区别开来，或是如果是一个使用高级语言带有对象的虚拟机呢？

答案很简单，查看这些过程，需找重复的代码格式。例如，如果它们使用相同的参数，并且这些相同的参数在多个使用它的函数中使用同样的格式，那它可能就是一个虚拟机。就我个人而言，我总是试着找共有的攻击点，如程序计数器（等效EIP）。这并不总是简单的事。如像*F（译者加：可能是一种虚拟机）的binded flow虚拟机通常就非常复杂（顺便说一下，你可以使用很多方法来记录）。

让我们回到这个crackme，滚动屏幕，寻找并跟随那些随机跳转和过程，我们发现一个有趣的列表，就像下面这样。

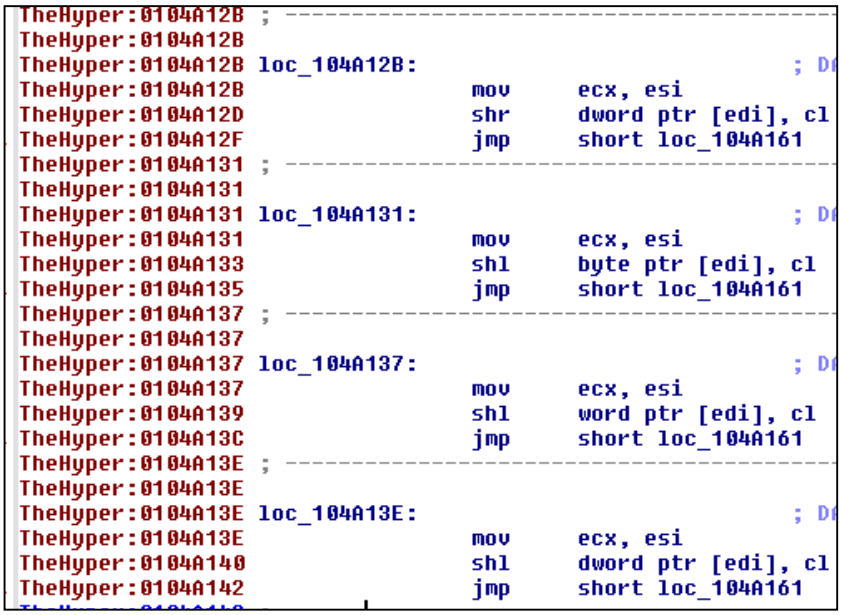


它看上去是不是有点意思，一个堆满指针的大表。我们查看一下这些多级指针（第一张表的链接指向第二张表的头部）的其中一个。



IDA给出了如上一堆数据，当我们使用C键把它们标记为代码后，就会变成：

有趣吧！在一个异或操作紧跟着一个跳转，我们在所有的这些地方使用快捷键，然后再看看发生了什么。。。



这些是我最初使用C的位置。查看一下代码，所有的片段都跳到相同的地址，这似乎意味着所有的片段都有一个相同的收场。

注意它们的第一条指令：在所有的入口均使用了mov ecx, esi这个指令！这不正是相同的格式片段吗-或许相同的逻辑参数被送入esi中？很明显，这是在下一条移位指令中是使用的移位计数，一个SHL。在这些的代码片段中，它们使用[edi]寄存器来保存结果。这三个代码片段都是相同的结构，只是改变了核心指令存储参照：字节指针，字指针，双字指针。这就是虚拟移位指令的三种方式。太好了！

我们已经知道了在这里SHL的源操作数被传递给esi，目的地址是edi，并且我们找出了字节移位、字移位、双字移位这几个移位指令。

我们很幸运，通常虚拟机的指令结构要比这复杂。这个虚拟机没有应用那些复杂的技术如指令内不同种类的寄存器/存储器/偏移参考等，它似乎在指令中使用了一个固定的源/目的地址：ESI是常规的源头指针，EDI是常规的目的结果指针（我们逆向更多后我们可以看出，通用寄存器通过内存参考传递给虚拟指令-例如，如果SHL目标是一个通用寄存器R1，edi将包含R1的指针）。

虚拟机的一个通常的非常标准的攻击点是等效的NOP指令，你如何能发现它们？简单。它们除了更新虚拟机的内部状态之外，什么都不做。因此，一个只是更新似乎是被用来作为程序计数器的寄存器极有可能就是我们这个虚拟机的NOP指令。然而这个crackme的虚拟机非常简洁，因此我们直接去识别那些复杂指令。

现在，是我们逆向所有指令并重命名它们的时候了。结果就像下面这样：

```
BINARY_TABLE    dd offset MOV           ; DAT
                dd offset ADD
                dd offset SUB
                dd offset XOR
                dd offset AND
                dd offset OR
                dd offset IMUL
                dd offset IDIV
                dd offset IDIV_REST
                dd offset ROR
                dd offset ROL
                dd offset SHR
                dd offset SHL
                dd offset CMP
XOR             dd offset XOR_BYTEPTR   ; DAT
                dd offset XOR_WORDPTR
                dd offset XOR_DWORDPTR
MOV             dd offset MOV_BYTEPTR   ; DAT
                dd offset MOV_WORDPTR
                dd offset MOV_DWORDPTR
ADD             dd offset ADD_BYTEPTR   ; DAT
                dd offset ADD_WORDPTR
                dd offset ADD_DWORDPTR
SUB             dd offset SUB_BYTEPTR   ; DAT
                dd offset SUB_WORDPTR
                dd offset SUB_DWORDPTR
OR              dd offset OR_BYTEPTR    ; DAT
                dd offset OR_WORDPTR
                dd offset OR_DWORDPTR
AND             dd offset AND_BYTEPTR   ; DAT
                dd offset AND_WORDPTR
                dd offset AND_DWORDPTR
IMUL            dd offset IMUL_BYTEPTR  ; DAT
                dd offset IMUL_WORDPTR
                dd offset IMUL_DWORDPTR
IDIV            dd offset IDIV_BYTEPTR  ; DAT
                dd offset IDIV_WORDPTR
                dd offset IDIV_DWORDPTR
IDIV_REST       dd offset IDIV_REST_BYTEPTR
```

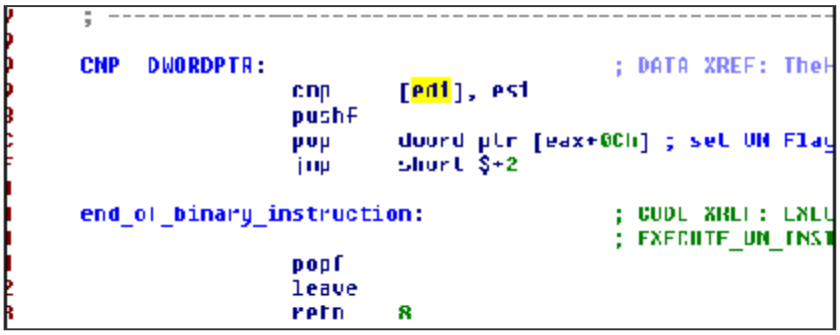
所有这些指令都与SHL指令非常相似（或多或少）。你会注意到，IDIV指令有点特别，它被分为IDIV和IDIV_REST两条指令。不知你是否记得，IDIV还要返回余数。如果你仔细看一下这两条指令，你会发现：

```
IDIV_DWORDPTR:                          ; DATA XREF
                xor     edx, edx
                mov     eax, [edi]
                idiv    esi
                mov     [edi], eax
                jmp     end_of_binary_instruction
```

```
IDIV_REST_DWORDPTR:                     ; DATA XREF: The
                mov     eax, [edi]
                xor     edx, edx
                idiv    esi
                mov     [edi], edx
                jmp     short end_of_binary_instruction
```

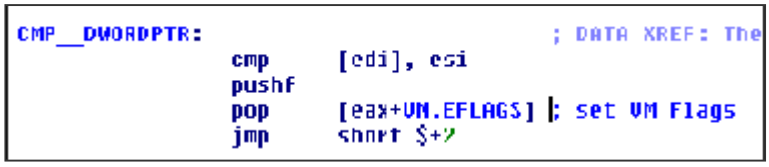
IDIV在EDI中返回不同的寄存器。你会想-为什么？简单，一个是结果，一个是余数。由于虚拟机指令是二元设置（源/目的）结构，所以对三元指令，作者需要重复一次操作。

注意，在我们重构一个虚拟机之前，我通常翻看所有的指令，以查找是不是还有我们没有谈过的的重要的东西。我总是寻找关于虚拟机寄存器结构的线索。例如，当我发现下面的指令，我首先想到：



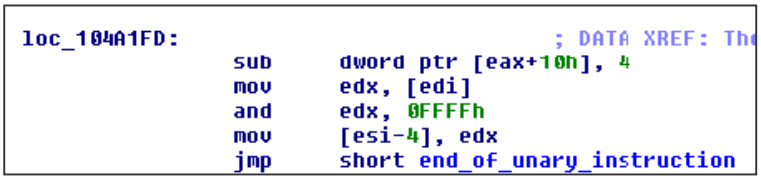
“PHUSF”??为什么它需要一个PHSHF指令？它在一个比较操作后保存标志位。呵呵…之后把他们弹送到一个EAX寄存器相关的结构，EAX被用来和其它虚拟机代码段交互吗？是的，当然。

[EAX+0Ch]很明显指向了一个虚拟EFLAGS结构。于是我们打开IDA的结构页，创建VM_EFLAGS结构并增加双字参数知道我们创建“Field_0Ch”，我们把它重命名为VM_EFLAGS。



结果如上。

现在我们识别了我们第一个虚拟机寄存器！我们再继续，当我们在指令中逆向操作代码时，我们发现了下面这个：



当我看到这个，我注意到：它取了一个固定寄存器（因为它相对于一个VM结构中的基址,EAX,的偏移10h是固定的）并把它减去4。从edi中取出操作数，并截取其低两个字节，再保存它们。你了解的什么汇编操作是当写入的时候减少寄存器数值？

或许这样会更清晰一些：

```
loc_104A20E:                            ; DATA XREF: The
                sub     [eax+VM.ESP], 4
                mov     edx, [edi]
                mov     [esi-4], edx
                jmp     short end_of_unary_instruction
```

我想我不需要再解释什么了，这是一个PHSH双字的操作。

又搞定一个虚拟机寄存器。我们继续，我们还没有找到EIP这个通用寄存器…让我们找到它们。浏览这个指令，我们会看到：

```
JZ:                                     ; DATA XREF: T
                push    dword ptr [eax+0Ch]
                popf
                jz      short loc_104A32A
                mov     [eax+8], edi

loc_104A32A:                            ; CODE XREF: T
                jmp     short end_of_flow_instruction
```

现在这个指令，与CMP指令类似，但它是一个JZ指令。它是个跳转，好，EIP肯定会被使用，如果我们跳到某处，我们必定要改变EIP寄存器。我们已经知道EAX+0cH是我们的VM_EFLAGS。因此，这里虚拟标志位送到CPU标志eflags里，JZ被执行。如果跳转没有发生，EDI参数被移到EAX+8。我们知道EAX包含VM_CONTEXT，因此我们猜想这个被拷贝的指令参数就是-跳转之后的新的EIP（这说明这个指令是JNZ而不是JZ），

于是：

```
JZ:                                     ; DATA XREF: T
                push    [eax+VM.EFLAGS]
                popf
                jz      short loc_104A32A
                mov     [eax+VM.EIP], edi

loc_104A32A:                            ; CODE XREF: T
                jmp     short end_of_flow_instruction
```

我们也找到了EIP寄存器。现在，试着识别一下下面这条指令吧：

```
                                        ; DATA XREF: T
        mov     edx, [eax+VM.EIP]
        mov     ecx, [eax+VM.ESP]
        sub     [eax+VM.ESP], 4
        mov     [ecx-4], edx
        mov     [eax+VM.EIP], edi
        jmp     short end_of_flow_instruction
```

除了它使用ESP和EIP外，我不会再给任何提示，想想吧。

另一个需要注意的是，你应该始终记住，虚拟机的作者在编制虚拟机时，不会严格遵守

“常规”。因此，指令不需要“标准”，他们可以依照创造者的意愿做任何事情。如，有条指令是这样的：

```
mov     esi, esp
mov     edx, [eax+VM.ESP]
mov     esp, edx
call    eui
mov     edx, [ebp+VM.EIP]
mov     [edx+38h], eax
mov     esp, esi
jmp     short $+2
```

你应该注意到:它使用了真实ESP寄存器！为什么？它保存真实的ESP，并把用虚拟堆栈设定为真实堆栈。接着通过EDX调用一个函数。这意味着这个虚拟机能在真实CPU空间中进行函数调用，它把虚拟参数压入虚拟堆栈再调用这条“交换堆栈”的指令（如果你了解处理器，它让我想起了一点在内部权限门之间通过参数拷贝的进行堆栈交换），还要注意执行函数的真实返回值被保存到我们的虚拟机环境(VM_context)中的某处…

我在半个小时内几乎逆向了全部指令和寄存器，只要付出点努力，你也可以做到。只有少数指令比较复杂，但对虚拟机逆向并不重要（我的意思是，对理解整个结构来说）

好了，该去睡觉了，非常非常晚了！希望你有所收获。----------Maximus

2 通用虚拟机结构

…又来了;-)

…好了，又该是我们打开mp3播放器的时候了;-)

如果我们检查虚拟机的结构，我们通常会发现一个大的循环来控制虚拟机的运转，来模拟一个处理器的运行，进行获取数据，解码和执行指令。HyperCrackme2使用的就是虚拟机的常规结构：

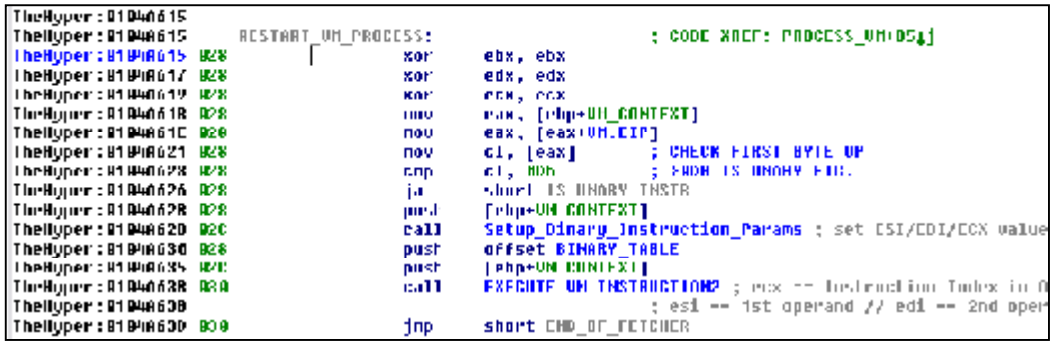
1、 设置虚拟机运行环境（VM_Context）。
2、 进入虚拟机循环。
3、 从VM.EIP地址读取指令字节，检查指令类型以支持不同的指令类型：
   i. 二进制指令
   ii. 一元指令
   iii. 流程控制指令
   iv. 特殊指令
   v. 调试指令
   vi. 空指令和挂起指令（“退出虚拟机”的别名）-虚拟机循环的结束
4、 跳到虚拟机循环的开始。

这个结构很容易记住。总的来看，每个虚拟机都包括以下要素：

- 虚拟机的初始化块/函数。

- 一个循环的函数块/函数来扫描和执行虚拟机程序的指令。
- 一个通用块或函数用来解释虚拟机指令的操作码和它的参数，寄存器，索引方式和其它任何虚拟机作者想放入的东东。
- 一张虚拟机指令地址表，每个代表一条指令。它们与用来分解和执行的普通汇编指令的微代码CPU的基本等效。
- 一套宏函数，是虚拟机特有的，不太容易与汇编代码对应起来。这些指令可能很难理解。

HyperCrackme2这个例子的初始结构要素可以通过查看下面的IDA注释片段来了解：



如你所见，RESTART_VM_PROCESS是上面描述的第（2）项。反之在ja short IS_UNARY_INSTR之下的部分等效于第（3.i）项。在本段中的代码，清理寄存器，取第一条操作码（VM.EIP指向的字节）并分析它，以选择虚拟机相应“执行单元”。

现在我们来看一个虚拟机构建部分，Setup_Binary_Instruction_Params函数，它负责处理二进制虚拟机操作码。我们已经知道EAX包换VM_CONTEXT，我们也知道eax+8中是我们的VM.IEP。

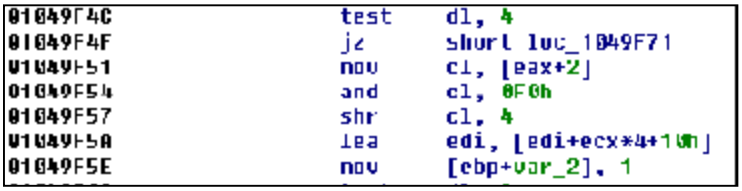
我想我们必须知道我们想要什么，否则一切分析都是无用功。我们在试图去恢复虚拟机指令结构，并弄清虚拟机结构的更多细节。用于为二进制指令填充参数的过程必须知道如何解码二进制指令，因此通过检查操作代码的字节如何生成，我们可以重建虚拟机指令格式。我们希望找到什么呢？它主要取决于指令集的复杂度，因此，我们必须始终细心的查看指令字节是如何被应用的。请记住虚拟机指令并不总是相同大小，如x86指令的大小就各不相同…

你无法将下面的方法应用到其它虚拟机上。每个虚拟机使用它自己的操作码和虚拟机结构，因此，你应试图去理解那些会给重构虚拟机一些线索的代码片段。

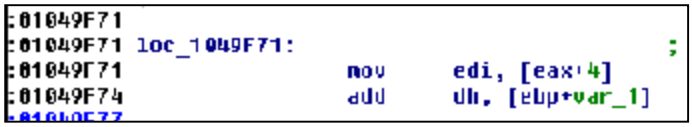
我们来看这个代码：

这个片段应该清楚了：我们装载由我们的虚拟EIP指向的第二个字节，[eax+1]，接着它

被送到dl寄存器。在我们做详细注释之前，我们应该注意我们只用了一个字节产生一条指令。我们继续吧：



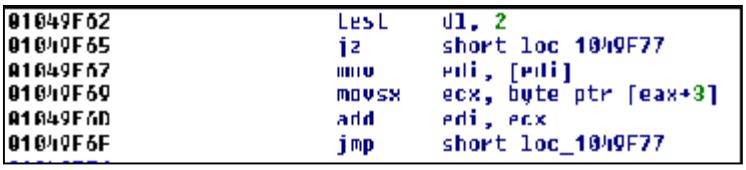
这个代码段与前一个非常相似（概念上）。EAX仍然包含我们的VM.EIP地址，现在构成操作码的第三个字节在内存中被载入和测试(只有它的高位被测试，你可以通过and/shr这对指令可以看出)。请注意接下来的指令，EDI包含我们的VM_CONTEXT指针。ECX寄存器包好一个双字索引，它被用在VM_CONTEXT结构中，来重新获得一个双字指针，接着它被偏移了10h，但是你是否还记得，VM_CONTEXT+10h= VM_ESP，这意味着，当ECX=0h时，在这里我们将会得到ESP寄存器的地址。当ECX=1h直到15h（1个半位元组范围是0-15）对应的是ESP之后的1-15个双字。于是我们立刻检测到这个二进制指令的第三个字节的可能用法-至少是用了它的高4位。下面这个片段是如果我们在上面的JZ指令实现跳转到的区域。



如你所见，它EAX（是我们的VM.EIP）中第一个双字之后的数据取走，并把它放入EDI中。我们知道EDI应该包含虚拟机操作指令的目的参数！这使我们知道，第一个双字被用作操作码，第二个双字是操作码参数。

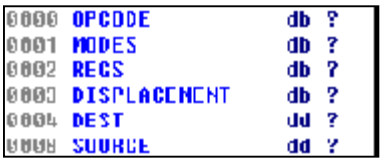
下面就是我们知道的VM_CONTEXT：

让我们继续我们的二进制操作码的分析，并试图确定VM_INSTRUCTION格式，我们已经遇到虚拟指令+1,+2偏移的情况，我们来看最后一个，偏移+3的情况：



这个字节通过MOVSX指令被直接装入ecx，你应该已经明白我要说什么了：为什么是MOVSX?这个字节接着被加到EDI参数上，那是我们的目的参数：我们为什么要给我们的EDI参数（它包含我们的目的操作数）加上些东西？当然，这是偏移…于是我们现在重建了二进

制指令的指令结构：



这部分，我没有评论太多，因为它是“虚拟机相关的”。

3 逆向指导

（略，一堆文字，有空再补吧）

风暴译 2007-10-6 QQ:719110750