Reversing a Simple Virtual Machine - Tutorial

1. Retrieving instructions and registers

Well, tonight I'm tired, I've downloaded a bunch of nice music songs that I like a lot, and it's time to reverse. Having received requests about this tutorial, contrary to my attitudes I'll write a small tutorial.

I've heard talking over and over of the HyperUnpackMe2, so at end, I opened it. I fired my IDA 4.3 -yeah, I don't use the cracked one... toolz are, after all, for those that can't do things without...

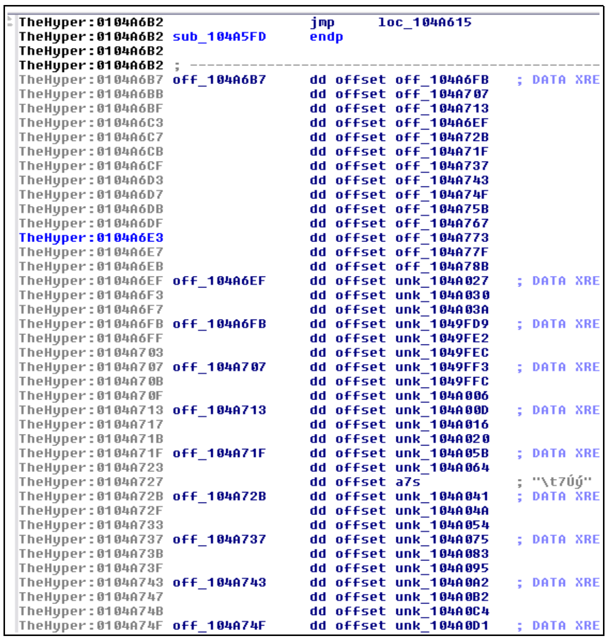
So, I opened the crackme. It starts with a lot of ugly anti-IDA tricks, which requires to un-define (U key) the jump/call pointers, and then redefine the pointed area as 'code' (C key). It hides the pointer to LoadLibrary and strings like i.e. “VirtualAlloc” this way. Ok, funny but not interesting, we want to see the virtual machine. Hoping it is not encrypted, otherwise we have to fire Olly and unpack the packer until the VM is in clear...

So, how do we search a VM in the code, using IDA 4.3? Simple: use the scrollbar and the most ancient of reversing tools: Zen.

What are we looking for, what could be a 'Zen' point? Well, When I browsed aspr 1.2 dll I found the *push* sequence followed by a *ret* to be 'Zen' point -indeed it was the to-do list of that packer. And for a VM? Well, a VM is formed by instruction emulation, which are usually function or addresses to which a common loop of code jumps to. In this case, we look for pointers/functions list. Yes, such lists can be many things. They could be objects, for example, which are stored this way. How can we distinguish them from a VM -or, what if the VM is coded with an object in HL?

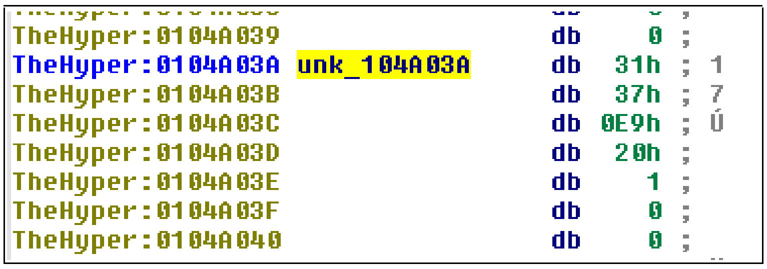
The answer is rather simple. Start examining these procedures, and look for recurrent patterns. For example, if they refer to the same parameters, and the same parameter seems to contain/be used in a pattern among more than one of these functions, you might be in presence of a VM. Personally, I always try to find references to common attack points, as the program counter (the EIP equivalent). This might not be always simple -i.e. binded flow VMs like *F are fairly complex (btw you can log it with various techniques).

But let's get back to the crackme. Let's say that scrolling, looking around and following randomly jumps and procs we found an interesting list, such the next one:



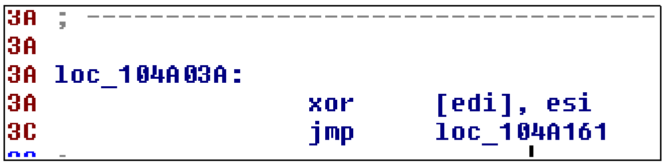
```
TheHyper:0104A6B2                 jmp     loc_104A615
TheHyper:0104A6B2 sub_104A5FD     endp
TheHyper:0104A6B2
TheHyper:0104A6B2 ; ---------------------------------------------------------------
TheHyper:0104A6B7 off_104A6B7     dd offset off_104A6FB    ; DATA XRE
TheHyper:0104A6BB                 dd offset off_104A707
TheHyper:0104A6BF                 dd offset off_104A713
TheHyper:0104A6C3                 dd offset off_104A6EF
TheHyper:0104A6C7                 dd offset off_104A72B
TheHyper:0104A6CB                 dd offset off_104A71F
TheHyper:0104A6CF                 dd offset off_104A737
TheHyper:0104A6D3                 dd offset off_104A743
TheHyper:0104A6D7                 dd offset off_104A74F
TheHyper:0104A6DB                 dd offset off_104A75B
TheHyper:0104A6DF                 dd offset off_104A767
TheHyper:0104A6E3                 dd offset off_104A773
TheHyper:0104A6E7                 dd offset off_104A77F
TheHyper:0104A6EB                 dd offset off_104A78B
TheHyper:0104A6EF off_104A6EF     dd offset unk_104A027    ; DATA XRE
TheHyper:0104A6F3                 dd offset unk_104A030
TheHyper:0104A6F7                 dd offset unk_104A03A
TheHyper:0104A6FB off_104A6FB     dd offset unk_1049FD9    ; DATA XRE
TheHyper:0104A6FF                 dd offset unk_1049FE2
TheHyper:0104A703                 dd offset unk_1049FEC
TheHyper:0104A707 off_104A707     dd offset unk_1049FF3    ; DATA XRE
TheHyper:0104A70B                 dd offset unk_1049FFC
TheHyper:0104A70F                 dd offset unk_104A006
TheHyper:0104A713 off_104A713     dd offset unk_104A00D    ; DATA XRE
TheHyper:0104A717                 dd offset unk_104A016
TheHyper:0104A71B                 dd offset unk_104A020
TheHyper:0104A71F off_104A71F     dd offset unk_104A05B    ; DATA XRE
TheHyper:0104A723                 dd offset unk_104A064
TheHyper:0104A727                 dd offset a7s            ; "\t7Úý"
TheHyper:0104A72B off_104A72B     dd offset unk_104A041    ; DATA XRE
TheHyper:0104A72F                 dd offset unk_104A04A
TheHyper:0104A733                 dd offset unk_104A054
TheHyper:0104A737 off_104A737     dd offset unk_104A075    ; DATA XRE
TheHyper:0104A73B                 dd offset unk_104A083
TheHyper:0104A73F                 dd offset unk_104A095
TheHyper:0104A743 off_104A743     dd offset unk_104A0A2    ; DATA XRE
TheHyper:0104A747                 dd offset unk_104A0B2
TheHyper:0104A74B                 dd offset unk_104A0C4
TheHyper:0104A74F off_104A74F     dd offset unk_104A0D1    ; DATA XRE
```

Does not it seem interesting? A long table of pointers. Let's then explore one of those secondary links (the first table of links just point to the head of seconds -mmh!)



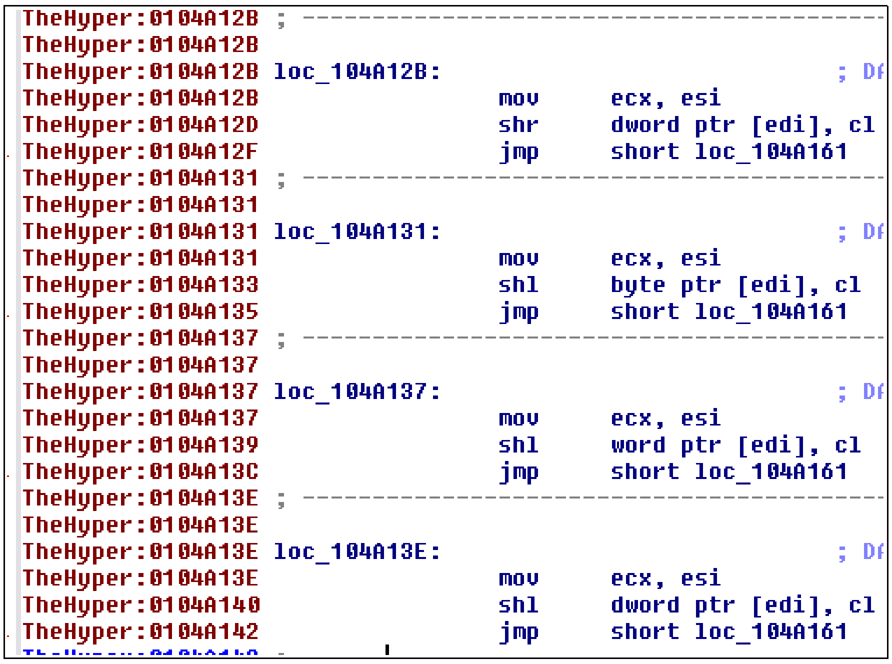
```
TheHyper:0104A039                 db    0 ;
TheHyper:0104A03A unk_104A03A     db  31h ; 1
TheHyper:0104A03B                 db  37h ; 7
TheHyper:0104A03C                 db 0E9h ; Ú
TheHyper:0104A03D                 db  20h ;
TheHyper:0104A03E                 db    1 ;
TheHyper:0104A03F                 db    0 ;
TheHyper:0104A040                 db    0 ;
```

IDA gives us this stuff as data, but after pressing C for marking it as code it becomes...



```
3A ; ------------------------------------
3A
3A loc_104A03A:
3A                    xor     [edi], esi
3C                    jmp     loc_104A161
```

Interesting, no? An *XOR* operation followed by a jump. Let's press 'C' on all the chunks, to see what's happen:



```
TheHyper:0104A12B ; ----------------------------------------
TheHyper:0104A12B
TheHyper:0104A12B loc_104A12B:                            ; DA
TheHyper:0104A12B                 mov     ecx, esi
TheHyper:0104A12D                 shr     dword ptr [edi], cl
TheHyper:0104A12F                 jmp     short loc_104A161
TheHyper:0104A131 ; ----------------------------------------
TheHyper:0104A131
TheHyper:0104A131 loc_104A131:                            ; DA
TheHyper:0104A131                 mov     ecx, esi
TheHyper:0104A133                 shl     byte ptr [edi], cl
TheHyper:0104A135                 jmp     short loc_104A161
TheHyper:0104A137 ; ----------------------------------------
TheHyper:0104A137
TheHyper:0104A137 loc_104A137:                            ; DA
TheHyper:0104A137                 mov     ecx, esi
TheHyper:0104A139                 shl     word ptr [edi], cl
TheHyper:0104A13C                 jmp     short loc_104A161
TheHyper:0104A13E ; ----------------------------------------
TheHyper:0104A13E
TheHyper:0104A13E loc_104A13E:                            ; DA
TheHyper:0104A13E                 mov     ecx, esi
TheHyper:0104A140                 shl     dword ptr [edi], cl
TheHyper:0104A142                 jmp     short loc_104A161
```

These are the first place were I originally pressed 'C'. Examine the code. All these snippets jump to the same address, which means they have a common epilogue.

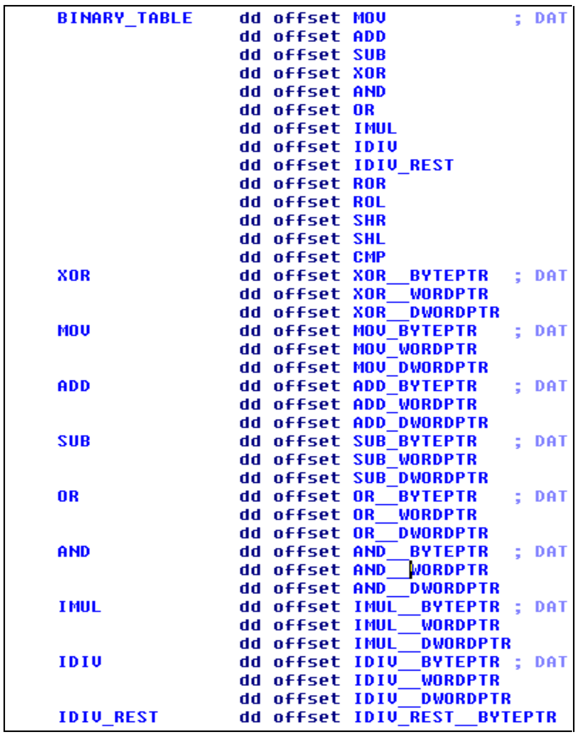
Notice the first instruction: a repeating *mov ecx, esi* in all the entries! Does it not sound as a pattern to you -maybe the same logical parameter is passed in *esi*? Clearly, it is the shift count used in the next instruction, a *shl*. They also uses the *[edi]* register as target area of the *shl* instruction in all the snippets. And all the three code blocks present the same structure, changing only the memory reference of the core (the 'acting') instruction: *byte ptr*, *word ptr*, *dword ptr*. Does this might be a virtual *shl* instruction in the three referencing possibilities? Yeah!

So we have understood that here the source parameter for *SHL* is passed in *esi*, the destination clearly in *edi*, and we have a sequence of *shl* on byte /*shl* on word/*shl* on dword.

We have been lucky, however. VMs are often more complex from the structural point of the instruction set. This VM does not implements many of the complexities related to the different kind of register/memory/displacement references within the instructions, as it seems to use a fixed source/destination mark for the instruction: *esi* is a generic pointer to source, and *edi* is a generic pointer to the destination result (as we can see by reversing more, generic VM registers are passed to the VM instructions by memory reference -i.e. If the destination of a *SHL* is the generic VM register *R1*, *edi* would contain the pointer to *R1)*.

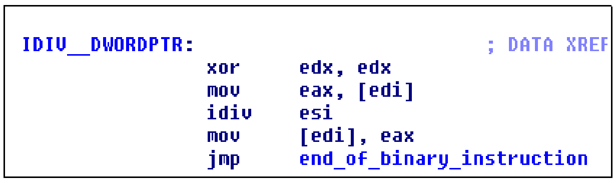
An usual and pretty standard attack point in VMs are the *NOP* instruction equivalents. How can you discover them? Simple. They do nothing but update the internal status of the VM. So, an instruction that just update a register which seems to be used as program counter can be very probably our *NOP* in such VM. This crackme's virtual machine is pretty straightforward, however, so we just attacked it recognizing complex instructions directly.

Now, it is time to reverse all these instruction blocks and name them. The result will lead to something like this:

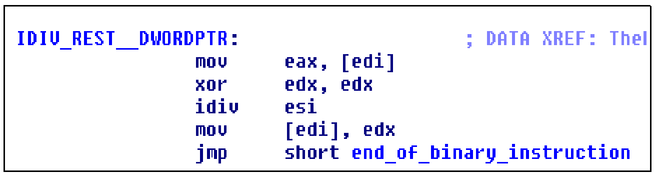


```
BINARY_TABLE    dd offset MOV           ; DAT
                dd offset ADD
                dd offset SUB
                dd offset XOR
                dd offset AND
                dd offset OR
                dd offset IMUL
                dd offset IDIV
                dd offset IDIV_REST
                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
```

All these instructions are structured exactly (more or less) like the *shl* one. One interesting point to observe is the *idiv* instruction. As you may notice, it has divided in IDIV and IDIV_REST. As you remember, IDIV return also the remainder of the division. If you examine how the the 2 virtual opcode are implemented, you'll notice:



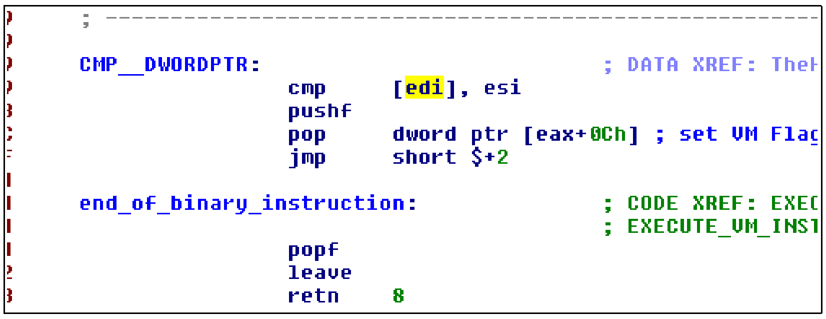
```
IDIV__DWORDPTR:                         ; DATA XREF
                xor     edx, edx
                mov     eax, [edi]
                idiv    esi
                mov     [edi], eax
                jmp     end_of_binary_instruction
```



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

the *idiv* return in EDI a different register. This should make you think -why? Simple. One is the result, the other the remainder. Being the VM instruction structured to work on binary set (source/destination), the author needed to duplicate the work of ternary instructions.

Notice that, before rebuilding a VM, I usually look to all the instruction set, trying to figure out something important we haven't talked yet about. I always look for hints about the VM register's structure. For example, when I found the following instructions, I first thought:

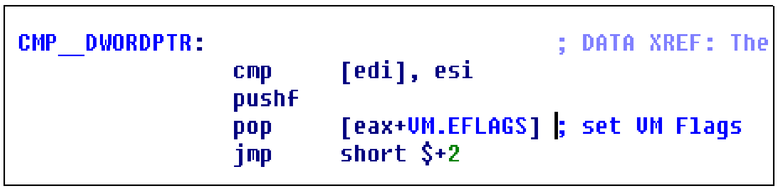


```
    ; ---------------------------------------------------
    CMP__DWORDPTR:                            ; DATA XREF: TheH
                    cmp     [edi], esi
                    pushf
                    pop     dword ptr [eax+0Ch] ; set VM Flag
                    jmp     short $+2

    end_of_binary_instruction:                ; CODE XREF: EXEC
                                              ; EXECUTE_VM_INST
                    popf
                    leave
                    retn    8
```

"*PUSHF*"??? Why do he need a PUSHF instruction here? He is saving the flags after a comparison. Mmh... and then pops them on a structure related to the EAX register. Is EAX register's used with displacements in other VM code snippets? Yes, of course.

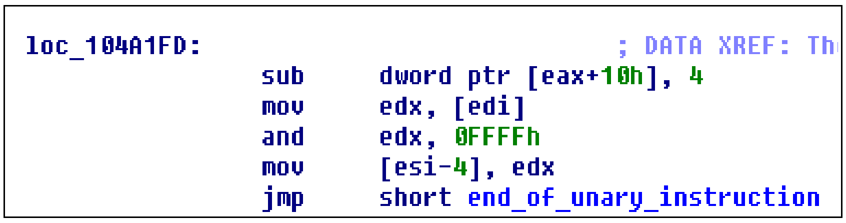
At this point ask yourself: why one should save the flags after a comparison within a relative structure? In case you did not understand this yet, the *[EAX+0Ch]* clearly points to the virtual EFLAGS register. So we can open the IDA structure page, create a structure and add doublewords until we create the "field_0Ch". Which we'll rename in VM_EFLAGS or such.



```
CMP__DWORDPTR:                            ; DATA XREF: The
                cmp     [edi], esi
                pushf
                pop     [eax+VM.EFLAGS] ; set VM Flags
                jmp     short $+2
```

As in the sample above.

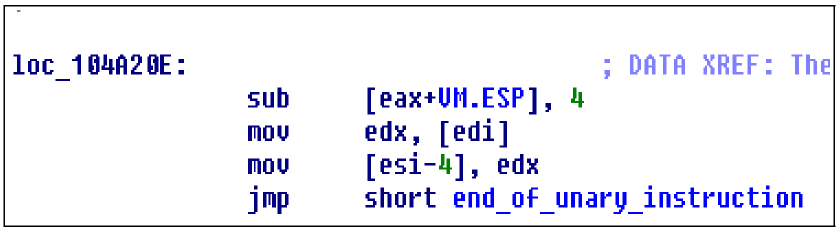
Now we have identified our first VM register! Let's hunt the other, while reversing opcodes. Among instructions, we find also the next one:



```
loc_104A1FD:                              ; DATA XREF: Th
                sub     dword ptr [eax+10h], 4
                mov     edx, [edi]
                and     edx, 0FFFFh
                mov     [esi-4], edx
                jmp     short end_of_unary_instruction
```

When I saw it I noted: it takes a fixed VM register (fixed because the offset from the VM structure base, *eax*, is fixed, 10h) and subtract 4. Take the operand from *edi* mask out the last 2 bytes and then store them. What asm operation do you know that decreases a register when writing?

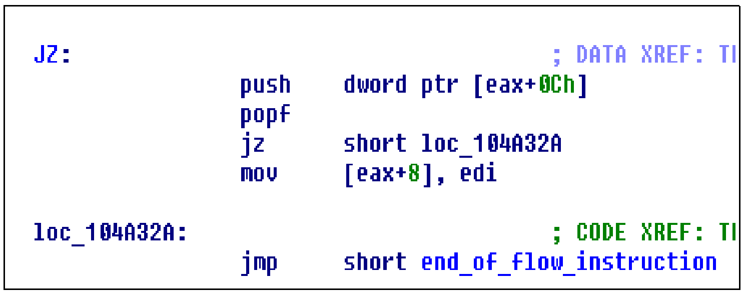
C'mon... maybe it is more clear now...



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

...I hope I needed not to comment it. This is PUSH DWORD.

And another VM register is uncovered. Let's go on, we still miss the EIP, the generic registers... Let's find them. Browsing the instructions, we can find:

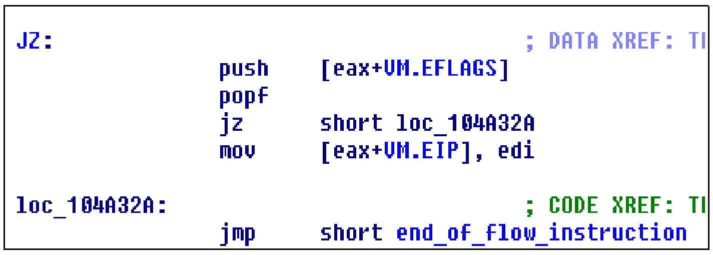


```
JZ:                                          ; DATA XREF: TI
                push    dword ptr [eax+0Ch]
                popf
                jz      short loc_104A32A
                mov     [eax+8], edi

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

Now, this instruction has the same layout of the *CMP*, but it features a *JZ* instruction. It is a jump, good. *EIP* must be used here, as we jump somewhere, so we must alter the *EIP* register somehow. We already know what EAX+0Ch is, it is our VM_EFLAGS. So, here the virtual eflags gets moved in CPU eflags, and JZ is executed. If the jump is NOT taken, the edi parameter is moved within *eax+8*. We know that *eax* contains our VM context, so we can bet that the instruction parameters that gets copied there is... our new *EIP* after the jump (technically, this means that the instruction is *JNZ*, not *JZ*!).

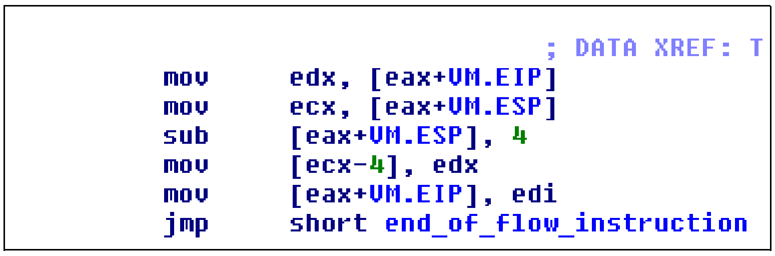
So...



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

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

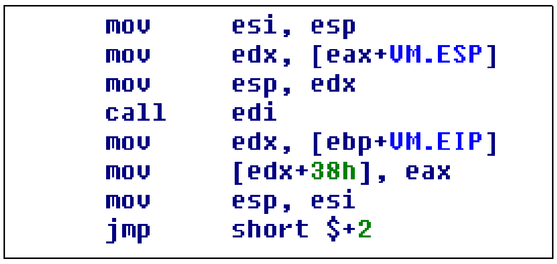
We found the VM EIP register. Now, try yourself to identify the next instruction:



```
                                 ; 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
```

I won't give you any hint, except that is clearly an instruction that uses ESP and EIP. Think please.

Another last interesting point. You should always keep in mind that the VM author is not ties to follow an 'rule' when coding a VM. So, instruction are not needed to be 'standard'. They can do anything their creator wishes. For example, one instruction does this:



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

You should notice this: it uses the real ESP register! Why? It saves the real ESP, than take the virtual stack and set it as the REAL stack. And call a function via EDX. This means that this virtual machine is capable of making calls in real CPU space, by pushing virtual parameters in the virtual stack and then calling this instruction, which swaps the stacks (it reminds a bit the stack switching with parameters copying between inter-privilege gates, if you know well processors). Also note that the return value of the real-cpu executed function is saved within our VM context, somewhere...

I reversed almost all the VM set and registers in half an hour, and you can do the same, with little effort. There are only a bunch of instructions that are more complex, but they are not important for VM reversing (I mean, for understanding the general structure).

Well, it is time for me to go to sleep, very very late! Hope you appreciated the small tute.

Maximus

2. General VM Structure

...next time ;-)

...Well, next time has come, let's fire the mp3 player with 'Liga' :-)

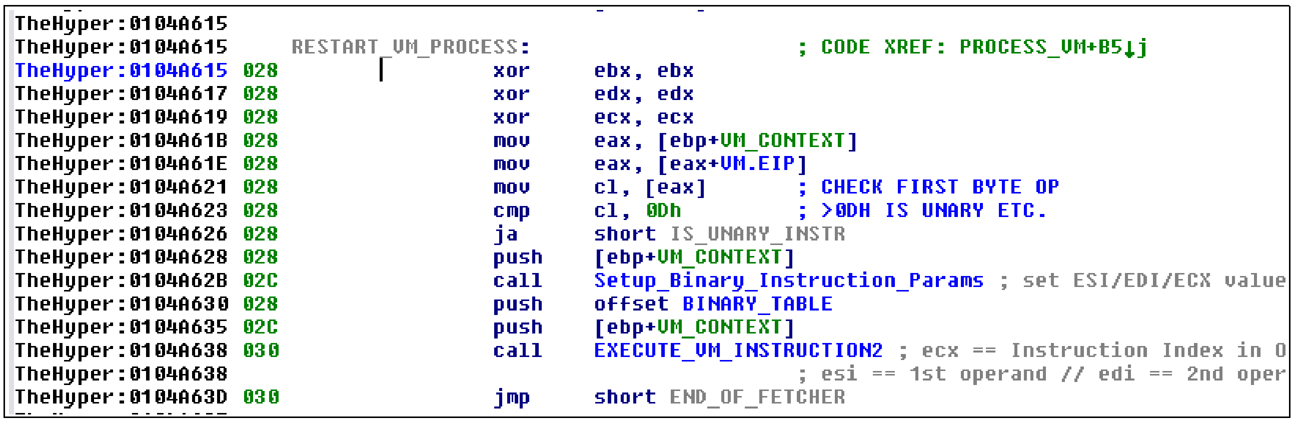
If we examine the general structure of a VM, we usually find a big cycle that takes care of running the VM across the virtual assembler, emulating this way the complex stages the processors execute when fetching, decoding and executing instructions. The HyperCrackme2 uses this generic VM structure:

1. Setup the VM Context.
2. Enter the VM loop.
3. Read byte at VM.EIP address and check the instruction type, supporting various instruction types:
    1. Binary Instructions.
    2. Unary Instructions.
    3. Flow Control Instructions.
    4. Special Instructions.
    5. Debug Instructions.
    6. NOP and HLT (alias “Quit VM”) instruction -the latter ending the VM loop.
4. Jump at start of the VM Loop.

This structure is general enough to be kept in mind. From a generic point of view, each VM contains the following elements:

- The initialization block/function of the Virtual Machine
- A loop block/function that scan and executes the instructions of the VM program.
- A generic block/function that decodes the VM instruction's opcode, with its parameters, registers, indexing modes and anything the VM creator wanted to place on.
- A list of VM instruction code blocks, which perform each an instruction duty. They are roughly the equivalent of the micro-code modern CPU's uses for decomposing and executing common ASM instructions.
- A set of macro-instructions, specific to the VM and not easily mappable to ASM opcodes. These instructions might be harder to understand.

An example of the HyperCrackme2 initial structure elements can be seen by examining the following commented IDA snip:



```
TheHyper:0104A615
TheHyper:0104A615     RESTART_VM_PROCESS:                        ; CODE XREF: PROCESS_VM+B5↓j
TheHyper:0104A615 028                   xor     ebx, ebx
TheHyper:0104A617 028                   xor     edx, edx
TheHyper:0104A619 028                   xor     ecx, ecx
TheHyper:0104A61B 028                   mov     eax, [ebp+VM_CONTEXT]
TheHyper:0104A61E 028                   mov     eax, [eax+VM.EIP]
TheHyper:0104A621 028                   mov     cl, [eax]        ; CHECK FIRST BYTE OP
TheHyper:0104A623 028                   cmp     cl, 0Dh          ; >0DH IS UNARY ETC.
TheHyper:0104A626 028                   ja      short IS_UNARY_INSTR
TheHyper:0104A628 028                   push    [ebp+VM_CONTEXT]
TheHyper:0104A62B 02C                   call    Setup_Binary_Instruction_Params ; set ESI/EDI/ECX value
TheHyper:0104A630 028                   push    offset BINARY_TABLE
TheHyper:0104A635 02C                   push    [ebp+VM_CONTEXT]
TheHyper:0104A638 030                   call    EXECUTE_VM_INSTRUCTION2 ; ecx == Instruction Index in O
TheHyper:0104A638                                                ; esi == 1st operand // edi == 2nd oper
TheHyper:0104A63D 030                   jmp     short END_OF_FETCHER
```

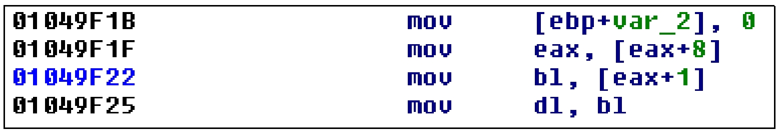
As you can see, the *RESTART_VM_PROCESS* is the point (2) of the above description, whereas the part under the *ja short IS_UNARY_INSTR* is equivalent to the (3.1) point. The code in this snippet, apart cleansing the registers, prefetch the first instruction Opcode (the byte pointed by *VM.EIP*) and analyse it for choosing which 'execution unit' of the VM should be utilised for the instruction type being fetched.

Let's now examine one of the 'building block' of this VM, the *Setup_Binary_Instruction_Params* function, which takes care of processing the binary VM opcodes. For examining the next fragment, remember that EAX contains our VM_CONTEXT. So, we already know that *eax+8* refers to our *VM.IEP*.

I think it is important now to understand what we are looking for, or analysis will be useless. We are trying to recover the VM Instruction structure, together with a more detailed description of the Virtual Machine structure. The procedure that fills up the parameters for the binary instructions must know how to decode the binary instructions, so by examining how the bytes that makes an opcode we can rebuild the VM instruction format. What should we expect to find? It depends heavily on the complexity of the instruction set, as it depends entirely by the author choices. Which we must reverse. So, we must always examine carefully how the instruction's byte are utilised, as they can change from instruction type to instruction type. And please remember that VM instruction are not compelled to be always of the same size, as x86 instruction's are not all of the same size...

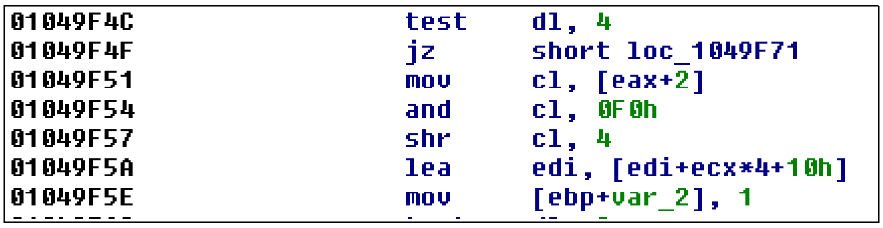
You won't be able to apply the method used below to other VMs. Each VM uses its own opcode and VM structure, so you should try to understand what fragments are used to hint its reconstruction.

Let's start by examining this code:



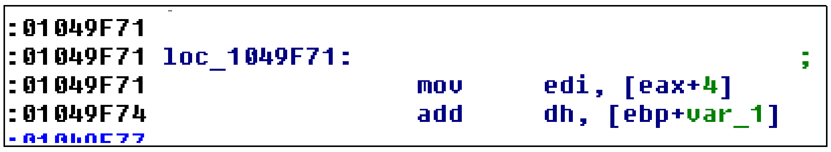
```
01049F1B                mov     [ebp+var_2], 0
01049F1F                mov     eax, [eax+8]
01049F22                mov     bl, [eax+1]
01049F25                mov     dl, bl
```

This snippet should be clear: we load the *second* byte pointed by our virtual *EIP*, *[eax+1]*, then we move it on the *dl* register. Before commenting in detail this point, we should keep notice we've just used one of the byes that makes an instruction. Let's move over.



```
01049F4C                  test    dl, 4
01049F4F                  jz      short loc_1049F71
01049F51                  mov     cl, [eax+2]
01049F54                  and     cl, 0F0h
01049F57                  shr     cl, 4
01049F5A                  lea     edi, [edi+ecx*4+10h]
01049F5E                  mov     [ebp+var_2], 1
```

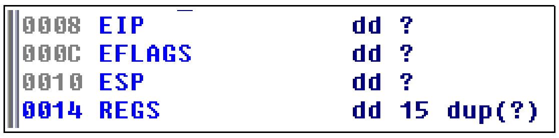
This snippet is pretty similar (conceptually) to the prior one. *EAX* still contains our *VM.EIP* address, and now the third byte forming the opcode is loaded in memory and tested (technically, only the high nibble of it is tested, as you can notice by the *and/shr* pair). And notice the instruction that follows. *EDI* contains our VM_CONTEXT pointer here. So, the *ECX* register contains a dword index, which is applied to the VM_CONTEXT for retrieving a dword pointer, which is then offseted by 10h. But do you remember? *VM_CONTEXT+10h == VM_ESP*. This means that when ECX is 0h here, we got the ESP register addressed. And when it is 1h, the dword after it is addressed, until the 15th DWORD after ESP (a nibble ranges 0-15, you'd know...). So, we detected right now a possible usage of the third byte of the binary opcodes -at least of its upper nibble. The snippet below is the area where we jump if we are successful in the *jz* instruction used in the code above.



```
:01049F71
:01049F71 loc_1049F71:                                  ;
:01049F71                 mov     edi, [eax+4]
:01049F74                 add     dh, [ebp+var_1]
```

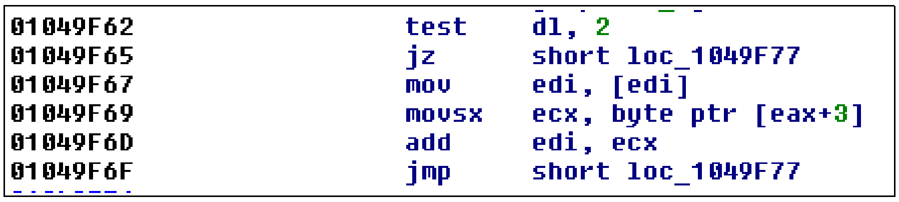
As you can notice, it takes the value that follows the first dword from *EAX* (which is our *VM.EIP*) and places it in *EDI*. And we know that *EDI* will contain at end the destination parameter of VM opcode! This help us understanding that the first dword is used only for the opcode purposes, and after it we have opcode parameters.

This is what we know of our VM_CONTEXT right now:



```
0008 EIP                dd ?
000C EFLAGS             dd ?
0010 ESP                dd ?
0014 REGS               dd 15 dup(?)
```

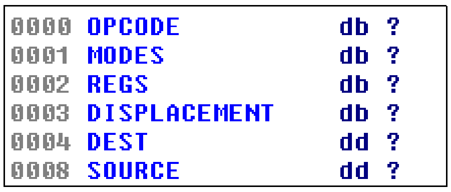
Let's continue our analysis of binary opcodes, and try to map the VM_INSTRUCTION format. We have already encountered the offsets +1,+2 of our VM instruction, so lets examine the last one, the +3:



```
01049F62                  test    dl, 2
01049F65                  jz      short loc_1049F77
01049F67                  mov     edi, [edi]
01049F69                  movsx   ecx, byte ptr [eax+3]
01049F6D                  add     edi, ecx
01049F6F                  jmp     short loc_1049F77
```

This byte is directly loaded in *ecx* using *MOVSX*. You should already understand what I'm about to say: why *MOVSX*?? this byte is then added to the EDI parameter, which contain our destination parameter. Why should we need to add something to our parameter? Displacement, of course...

So, we now can rebuild the instruction's structure for Binary instruction's:



```
0000 OPCODE          db ?
0001 MODES           db ?
0002 REGS            db ?
0003 DISPLACEMENT    db ?
0004 DEST            dd ?
0008 SOURCE          dd ?
```

I agree I haven't commented much this part. But the reason is that it is very 'VM-dependent'.

3. Reversing VMs Guidelines

The steps shown in prior chapters are an important step toward the comprehension of a VM.

- You can initially skip the structure of a VM instruction, as long as it is not decrypted/decoded within each instruction.
- At this point, we must examine deeply the instruction set trying to find something recognizable, as the NOP instruction -which might not be included at all.
- Once the instruction set is starting to result clear, at least in minimal part, a special care must be set by looking for possible VM register's usage. Eventually their usage won't be clear, as they can be 'shifty', remapped upon each VM entry etc. but we don't care. Knowledge is incremental, and making errors is human -especially if you abuse of Zen for quickening your analysis by intuition ;-)
- At this point, we must attack the 'living heart' of the VM, its decoder. It contains all the important information's of the VM and the structure of the VM instructions, as it is usually responsible for the scheduling and performing the instruction (pre-)processing. You must remember that often the decoder have to analyse the VM instruction for discovering things like the opcode length, parameters and so on. But it is also possible that part of the management is performed in the instructions itself -i.e. making instructions of fixed size (i.e. 16 bytes).
- And then? Then we must get back to the instruction set, trying to understand specific, non-standard opcodes that perform creative duties that are usually not part of a processor (i.e. Calls to 'real' functions, API functions, calculation blocks etc. etc.).
- At this point we have decoded most of the VM, and we might try to debug an instruction or two to se if things are as we expected, and if VM registers follows up our scheme.
- But before or later you have to get coding for dumping the VM Program in comprehensible shape. You might wish to write an IDA plugin (if you don't use 4.3 like me) or a script for decoding the VM program. Or much simpler but slightly less effective, you can code a logger, which is simply an hook in the VM instruction table, for each instruction (simply make your debugger-loader and use breakpoints which you defer in the breakpoint event, or inject a dll which hooks the table). Whenever an instruction is called, your hook dumps the opcode name, and the parameters. So, you can rebuild the flow of the program. An useful add-on to the logger is a VM.EIP dumper, which allows you to assign the right key to each VM instruction, and eventually the possibility to 'alter' the result of conditional jumps, so to allow the logger to examine the major part of the VM program and eventually 'skip' long cycles. Later, you can reassemble most of the VM program it using the VM.EIP logged for each instruction.

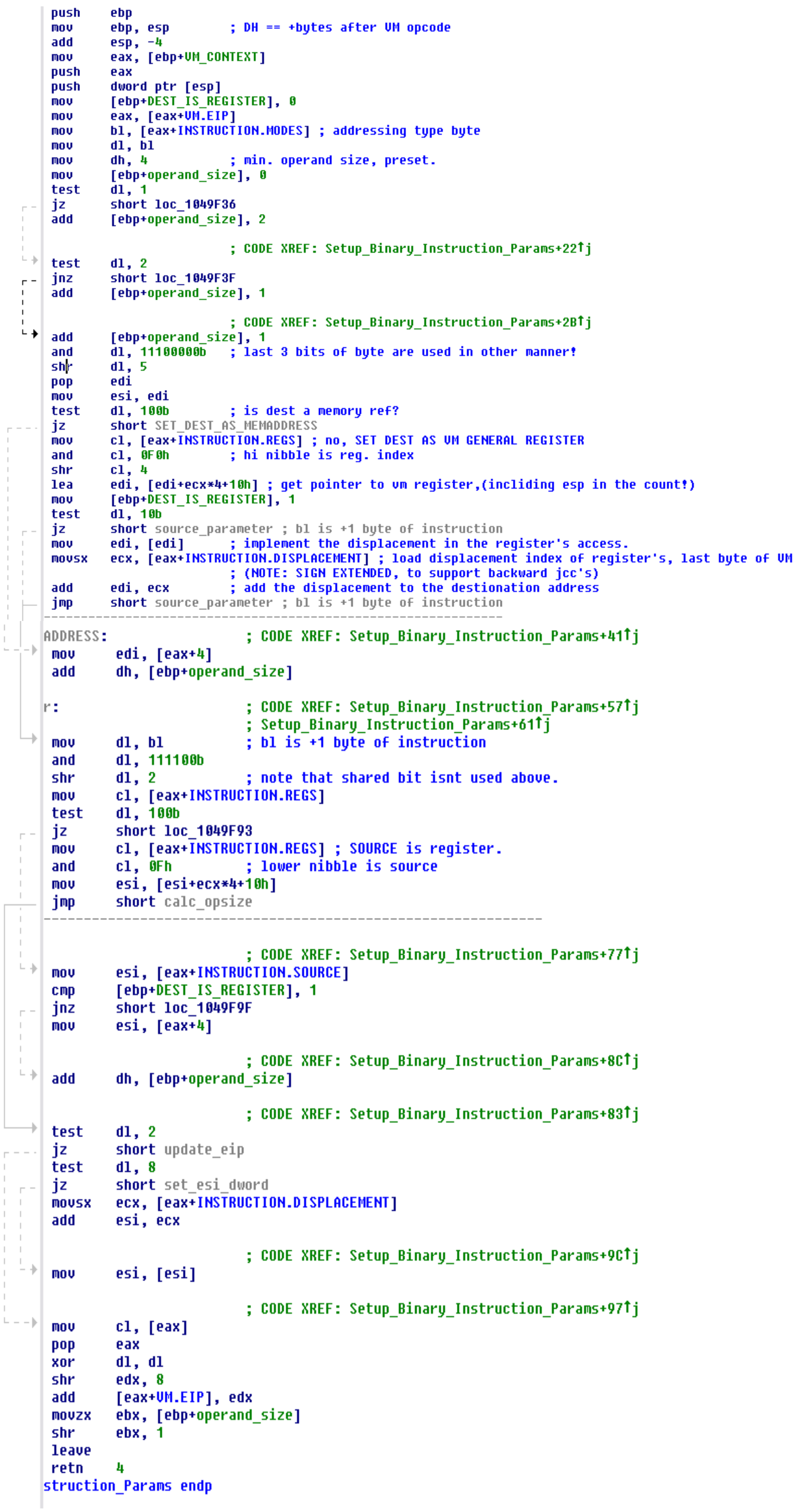
Well, I hope this can help you all to understand VMs better. I saw is common style in tutorials to place credits, so my thanks to the Community and my friends Zero and HAVOK.

Regards,

Maximus

15-16/7/2006

For the curious, this is my IDA analysis of the binary parameter's setup decoder of the crackme:



```
push    ebp
mov     ebp, esp        ; DH == +bytes after VM opcode
add     esp, -4
mov     eax, [ebp+VM_CONTEXT]
push    eax
push    dword ptr [esp]
mov     [ebp+DEST_IS_REGISTER], 0
mov     eax, [eax+VM.EIP]
mov     bl, [eax+INSTRUCTION.MODES] ; addressing type byte
mov     dl, bl
mov     dh, 4           ; min. operand size, preset.
mov     [ebp+operand_size], 0
test    dl, 1
jz      short loc_1049F36
add     [ebp+operand_size], 2

                        ; CODE XREF: Setup_Binary_Instruction_Params+22↑j
test    dl, 2
jnz     short loc_1049F3F
add     [ebp+operand_size], 1

                        ; CODE XREF: Setup_Binary_Instruction_Params+2B↑j
add     [ebp+operand_size], 1
and     dl, 11100000b   ; last 3 bits of byte are used in other manner!
shr     dl, 5
pop     edi
mov     esi, edi
test    dl, 100b        ; is dest a memory ref?
jz      short SET_DEST_AS_MEMADDRESS
mov     cl, [eax+INSTRUCTION.REGS] ; no, SET DEST AS VM GENERAL REGISTER
and     cl, 0F0h        ; hi nibble is reg. index
shr     cl, 4
lea     edi, [edi+ecx*4+10h] ; get pointer to vm register,(incliding esp in the count!)
mov     [ebp+DEST_IS_REGISTER], 1
test    dl, 10b
jz      short source_parameter ; bl is +1 byte of instruction
mov     edi, [edi]      ; implement the displacement in the register's access.
movsx   ecx, [eax+INSTRUCTION.DISPLACEMENT] ; load displacement index of register's, last byte of VM
                        ; (NOTE: SIGN EXTENDED, to support backward jcc's)
add     edi, ecx        ; add the displacement to the destionation address
jmp     short source_parameter ; bl is +1 byte of instruction
------------------------------------------------------------
ADDRESS:                 ; CODE XREF: Setup_Binary_Instruction_Params+41↑j
 mov     edi, [eax+4]
 add     dh, [ebp+operand_size]

r:                       ; CODE XREF: Setup_Binary_Instruction_Params+57↑j
                         ; Setup_Binary_Instruction_Params+61↑j
 mov     dl, bl          ; bl is +1 byte of instruction
 and     dl, 111100b
 shr     dl, 2           ; note that shared bit isnt used above.
 mov     cl, [eax+INSTRUCTION.REGS]
 test    dl, 100b
 jz      short loc_1049F93
 mov     cl, [eax+INSTRUCTION.REGS] ; SOURCE is register.
 and     cl, 0Fh         ; lower nibble is source
 mov     esi, [esi+ecx*4+10h]
 jmp     short calc_opsize
------------------------------------------------------------

                         ; CODE XREF: Setup_Binary_Instruction_Params+77↑j
 mov     esi, [eax+INSTRUCTION.SOURCE]
 cmp     [ebp+DEST_IS_REGISTER], 1
 jnz     short loc_1049F9F
 mov     esi, [eax+4]

                         ; CODE XREF: Setup_Binary_Instruction_Params+8C↑j
 add     dh, [ebp+operand_size]

                         ; CODE XREF: Setup_Binary_Instruction_Params+83↑j
 test    dl, 2
 jz      short update_eip
 test    dl, 8
 jz      short set_esi_dword
 movsx   ecx, [eax+INSTRUCTION.DISPLACEMENT]
 add     esi, ecx

                         ; CODE XREF: Setup_Binary_Instruction_Params+9C↑j
 mov     esi, [esi]

                         ; CODE XREF: Setup_Binary_Instruction_Params+97↑j
 mov     cl, [eax]
 pop     eax
 xor     dl, dl
 shr     edx, 8
 add     [eax+VM.EIP], edx
 movzx   ebx, [ebp+operand_size]
 shr     ebx, 1
 leave
 retn    4
struction_Params endp
```