deroko [ARTeam], 2kAD [ARTeam]

Special Issue for SecuRom 7.30.0014 Take2 VM Analysis

Version 1.0
Last Rev.: December 2007

Into this Tutorial

1. SecuROM : Even caveman can do it by deroko
2. Recursive “VM” by 2kAD

Forewords

After the publication of our previous tutorial on SecuROM [1] we had a lot of discussions about its title, on several forums we had comments about if it was a really complete owning or not.

On woodmann forum (www.woodmann.com, search for “*A continued discussion on "ownage"*”, and “*ARTeam: Special Issue For SecuRom 7.30.0014 Complete Owning, AnonymouS, Human, deroko*”) we had an interesting discussion on what can be considered owning of a program and what cannot.

The discussion has been really long and I’ll not report it here, but would summarize the two main positions.

Owning is when you return the program to be virgin as it was when it was still to be packed. So cracks around which circumvent the protection without actually removing it are not real own. I would call this position the position of “Purists”.

On the other hand there’s the position of who tells (and I’m among them) that any method is legit to fool the protection. It the application fails its task (protecting) then it makes no sense to even exist. Protections are placed to protect and only for that reason, often even at the price of portability and efficiency of the code. If you, using any dirty method, successfully fool the application, letting it run in unforeseen contexts (e.g. a not licensed PC), you then owned the protection. Or better you might not have run the protector (not understood all its aspects) but surely you owned the protection. And a protector without a protection is nothing in my humble opinion.

“Purists” approach is much more hard because implies also understanding parts of the protection not really needed to break it. It’s something one can call professional reverse engineering. If you are e developer you are anyway interested in techniques useful to avoid *any* type of reverse engineering, professional or not, which owns your application or your protection or your protector.

You can think more or less the same when you try to distinguish cracking from reverse engineering..

This said we decided to show that owning on SecuROM can be done either ways you consider it. This second issue on the protector tries to cover some aspects that was not covered in previous one, specifically the most important part of the SecuROM protector, the Virtual Machine, it’s really well done: is recursive and changes from application to application ...

This time deroko again hits the ground, but it’s also the first time of 2kAD. I hope you will enjoy it, but I must warn you, this one is not going to be an easy one ☺

Have phun,
Shub

Editor: Shub-Nigurrath

Disclaimers

All code included with this tutorial is free to use and modify; we only ask that you mention where you found it. This tutorial is also free to distribute in its current unaltered form, with all the included supplements.

All the commercial programs used within this document have been used only for the purpose of demonstrating the theories and methods described. No distribution of patched applications has been done under any media or host. The applications used were most of the times already been patched, and cracked versions were available since a lot of time. ARTeam or the authors of the paper cannot be considered responsible for damages to the companies holding rights on those programs. The scope of this tutorial as well as any other ARTeam tutorial is of sharing knowledge and teaching how to patch applications, how to bypass protections and generally speaking how to improve the RCE art and generally the comprehension of what happens under the hood. We are not releasing any cracked application. We are what we know..

Verification

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: http://releases.accessroot.com

Table of Contents

SecuROM : Even caveman can do it by deroko

1. Forewords

This tut was done 2 months ago, but due to lack of interest to release it we didn't release it immediately. Whenever new document is written, I always try to think what you, as a reader, will learn from it. To be honest, this dilemma was the only thing which stoped me from publishing this tutorial 2 months ago, when it was done. I still don't know what you will learn from it, but still, I decided to publish it.

So, enjoy in it ☺

2. Tools and Target

Target used this time is game protected with SecuROM 7.34 demo version as I wasn't able to get full version at the time of writing this tutorial, but everything from this tutorial should be ***mutatis mutandis*** applied to full version without any problem.

Tools that we will need, are:

- SoftICE
- IDA

3. Few words about SecuROM VM

First of all SecuROM is composed of 256 handlers, each handler is responsible for performing simple operation which exist in IA32 (eg. no instructions such as mov [vm_reg], [vm_reg]). Opcodes on other hand don't have predefined format, and execution of opcodes is dependent on execution of previous opcode(s), as if one of opcodes is not executed you will break predefined flow of a VM.

Well you'll see all of this as we go along with VM analyze, and you will see how easy is to extract all needed info to write emulator or vm decompiler. Although easy doesn't mean short and fast coding ☺

4. VM analyse

VM_Enter is code (well you may name it different) responsible for setting VM_Context, and for dispatching execution to VM handlers.

Let's see one example of VM_Enter:

```
.text:10D9D436 loc_10D9D436:
.text:10D9D436                 push    eax             ; nShowCmd
.text:10D9D437                 push    esi             ; lpCmdLine
.text:10D9D438                 push    0               ; hPrevInstance
.text:10D9D43A                 push    offset __ImageBase ; hInstance
.text:10D9D43F                 call    _WinMain@16     ; WinMain(x,x,x,x)
```

…

```
.text:10912B10 ; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nShowCmd)
.text:10912B10 _WinMain@16     proc near
.text:10912B10
.text:10912B10 var_C           = dword ptr -0Ch
.text:10912B10
.text:10912B10
.text:10912B10                 jmp     ds:off_11E0515C
.text:10912B10 _WinMain@16     endp
```

And we come to code responsible for calling VM:

```
.bla:11E95FB0 win_main_ref:
.bla:11E95FB0                push    offset vm_argument
.bla:11E95FB5                push    401149h          ; dummy argument to simulate call
.bla:11E95FBA                push    (offset loc_11639F6D+3)
.bla:11E95FBF                pushf
.bla:11E95FC0                sub     dword ptr [esp+4], 1ABB0h
.bla:11E95FC8                popf
.bla:11E95FC9                retn                     ; goes to 1161F3C0

.bla:11E95FCA vm_argument    dd offset vm_opcodes_0
.bla:11E95FCE                dd 1C93h
.bla:11E95FD2                dd 6B21h
.bla:11E95FD6                dd 4BE4h
.bla:11E95FDA                dd 0
.bla:11E95FDE                db    0
.bla:11E95FDF                db    0
```

So far so good, and we are almost there, but before we enter into VM_Enter there are certain stuff that I wanna show at this point:

1. You may see my comments and that address 401149h is dummy argument used to simulate call to VM. This is logical, as this argument is later on used as x86 EIP.
2. Code between pushfd/popfd is used to calculate offset of VM_Enter.
3. VM_Argument is important as it is used on other hand to tell VM where are Opcodes and what set of vm handlers to use.

```
.bla:11E76756 vm_opcodes_0   dd 70B2F0C5h, 0D491B739h, 0D845BB32h,
                                13383248h, 120078DAh
.bla:11E76756                dd 995DD350h, 1137B10h, 630F31E6h, 98B64673h,
                                48BDCD2Fh
.bla:11E76756                dd 0B2E7CA02h, 4C86144Dh, 3A72F00h, 6290B218h,
                                0E287B2h
.bla:11E76756                dd 1A175D8Dh, 0D9C28D4Dh, 87614CA1h,
                                0DC1295ADh, 0BA815B98h
.bla:11E76756                dd 204D9623h, 8D76ED4Dh, 67C70000h, 7571002Dh,
                                0D63A685Ch
.bla:11E76756                dd 7B55CA91h, 972CFEB1h, 5E7FA624h, 0CF06AC15h,
                                13D62E2Ch
.bla:11E76756                dd 0C39CBB56h, 13392C20h, 29DA135Ah, 5F68ADF5h,
                                0CC40ED6Eh
.bla:11E76756                dd 2B22794h, 0D1B64D29h, 112298E0h, 9435A453h
.bla:11E767F2                dd 5296C311h
```

Oki now is time to show full VM_enter and to cover it line by line for better understanding:

```
valloc:018B0000 vm_enter_main_handler:
valloc:018B0000                 jmp     short __skip_space_for_rent ; esp-20
valloc:018B0000
valloc:018B0002                 db  3Ch ; <
valloc:018B0003                 db  20h
valloc:018B0004                 db  73h ; s
valloc:018B0005                 db  70h ; p
valloc:018B0006                 db  61h ; a
valloc:018B0007                 db  63h ; c
valloc:018B0008                 db  65h ; e
valloc:018B0009                 db  20h
valloc:018B000A                 db  66h ; f
valloc:018B000B                 db  6Fh ; o
valloc:018B000C                 db  72h ; r
valloc:018B000D unk_18B000D     db  20h
valloc:018B000E                 db  72h ; r
valloc:018B000F                 db  65h ; e
valloc:018B0010 unk_18B0010     db  6Eh ; n
valloc:018B0011                 db  74h ; t
valloc:018B0012                 db  20h
valloc:018B0013                 db  3Eh ; >
valloc:018B0014
valloc:018B0014
```

Funny, really funny, but as this code obviously has no any meaning we may skip it. Now we enter into VM_wait loop, which will wait until byte at 18B0020 isn't set to 1 again.

```
valloc:018B0014 __skip_space_for_rent:
valloc:018B0014                 pusha                           ; esp-20
valloc:018B0015                 pushf                           ; esp-24
valloc:018B0016                 call    $+5                     ; esp-28
valloc:018B001B                 call    sub_18B0022
valloc:018B001B
valloc:018B0020 byte_18B0020    db 1
valloc:018B0021                 db 1
valloc:018B0022
valloc:018B0022
valloc:018B0022
valloc:018B0022 sub_18B0022     proc near
valloc:018B0022
valloc:018B0022                 pop     edx
valloc:018B0023
valloc:018B0023 __vm_wait_loop:
valloc:018B0023                 lock dec byte ptr [edx]
valloc:018B0026                 jns     short __vm_ready
valloc:018B0028
valloc:018B0028 __vm_wait:
valloc:018B0028                 cmp     byte ptr [edx], 0
valloc:018B002B                 pause
valloc:018B002D                 jle     short __vm_wait
valloc:018B002F                 jmp     short __vm_wait_loop
valloc:018B002F
valloc:018B0031 aMassesAgainstT db '-[ Masses Against the Classes <¦>>< ]-',0
```

In this code you may see that SecuROM gives you 10 possible VM_context which means that there is possibility of executing 10 threads in VM at the same time. (mov ecx, 0Ah). Each context struct will be checked for busy flag, until one of them isn't released by VM_exit handlers (I'll cover a few latter on). Note how byte at 18B0020 is used as vm interpreter busy flag.

```
valloc:018B0058
valloc:018B0058 __vm_ready:
valloc:018B0058
valloc:018B0058                 mov     eax, 0FFFFFFFCh
valloc:018B005D                 mov     ecx, 0Ah
valloc:018B0062
valloc:018B0062 __find_free_vmcontext:
valloc:018B0062                 add     eax, 4
valloc:018B0067                 xchg    ebp, ebp
valloc:018B0069                 adc     ebp, eax
valloc:018B006B                 mov     edi, ds:vm_contexts[eax]
valloc:018B0071                 cmp     [edi+srom_vm_context.busy], 0DE859E9h
valloc:018B0078                 jnz     short __vm_context_free_found
valloc:018B007A                 loop    __find_free_vmcontext
valloc:018B007C                 pause
valloc:018B007E                 jmp     short __vm_ready
valloc:018B0080
```

Now SecuROM simply walks all 10 vm_cotexes and checks busy flag in all of them, once free context is found it will zero whole context, mark it as used, and simply will set byte as 18B0020 to 1, which means that vm_interpreter is ready to handle another thread. You may see that in following code:

```
valloc:018B0080
valloc:018B0080 __vm_context_free_found:
valloc:018B0080                 cld
valloc:018B0081                 mov     ebx, edi
valloc:018B0083                 mov     ecx, 100h
valloc:018B0088                 mov     eax, 0
valloc:018B008D                 rep stosd
valloc:018B008F                 mov     [ebx+srom_vm_context.busy], 0DE859E9h
valloc:018B0096                 mov     al, [edx+1]
valloc:018B0099                 xchg    al, [edx]
```

VM_context is marked as used, and now SecuROM will processed with filling important parts of vm_context with needed data.

```
valloc:018B009B                 sub     dword ptr [esp], 1Bh ; edx = 018B0000
valloc:018B009B                                              ;            |
valloc:018B00A2                 pop     edx                  ; <---------+
```

At this point SecuROM vm_interpreter is calculating it's offset in the memory which will be later used as delta to VM_Opcodes and delta to VM_Handlers.

```
valloc:018B00A3                 mov     eax, [esp+28h]
valloc:018B00A3
valloc:018B00A3
valloc:018B00A7                 mov     [ebx+srom_vm_context.argument], eax
valloc:018B00AA                 mov     eax, [eax] ; grab dword from argument
valloc:018B00AC                 sub     eax, edx
```

If you remember I showed earlier that argument has pointer to vm_opcodes, so this is the point when that dword is taken and SecuROM makes it relative to vm_enter using this simple formula:

VM_opcodes – offset vm_enter

```
valloc:018B00AE             push    eax
valloc:018B00AF             mov     eax, [esp+2Ch]   ; argument
valloc:018B00B3             cmp     dword ptr [eax+0Ch], 50h ; argument + 0C
valloc:018B00BA             jnz     short loc_18B00CA
valloc:018B00BC             xchg    ebp, ebp
valloc:018B00BE             adc     ebp, eax
```

```
valloc:018B00C0             mov     eax, offset vm_handlers2
valloc:018B00C5             jmp     loc_18B00D3
valloc:018B00CA loc_18B00CA:
valloc:018B00CA             xchg    ebp, ebp
valloc:018B00CC             adc     ebp, eax
valloc:018B00CE             mov     eax, offset vm_handlers1 valloc:018B00D3
valloc:018B00D3 loc_18B00D3:                          ; CODE XREF: sub_18B0022+A3j
valloc:018B00D3             sub     eax, edx
valloc:018B00D5             mov     [ebx+srom_vm_context.handlers_delta], eax
valloc:018B00D7             pop     eax
valloc:018B00D8             mov     [ebx+srom_vm_context.vm_kinda_eip], eax
```

Oki doki, game continues, now SecuROM checks which set of handlers to use? It all depends on value passed in argument+0xC. If that value is 0x50 then SecuROM will use set of 64 handlers, otherwise it will use set of 256 handlers. Note also how at 18B00D3 address of handlers is made relative to vm_enter. If you look at your debugger or IDA without nice struct that I'm using for analyze you will see what is what, but I will show struct later.

So far so good, SecuROM now sets parts of VM which are very important for vm. As you may see it takes eflags from the stack, sets modifier (this field I have named like this as it is very important for vm work, it is byte located at vm_context + 0x10):

```
valloc:018B00DB          mov   eax, [esp]
valloc:018B00DE          mov  [ebx+srom_vm_context.e_flags], eax
valloc:018B00E1          mov  [ebx+srom_vm_context.vm_enter_address], edx
valloc:018B00E4          mov  [ebx+srom_vm_context.modifier], 95h
valloc:018B00E8          mov  [ebx+srom_vm_context.self_context_pointer], ebx
valloc:018B00EB          mov  [ebx+srom_vm_context.x86_eflags_regs], esp
```

Well this code yet doesn't have clear meaning as it is only pointer to proper vm_context, so we leave it as unknown field. Honestly I didn't see it being used in any part of VM I have analyzed so I leave it as is.

```
valloc:018B00EE                 mov     eax, [esp+28h]  ; argument
valloc:018B00F2                 cmp     dword ptr [eax+0Ch], 50h
valloc:018B00F9                 jnz     short loc_18B0109
valloc:018B00FB                 adc     ebp, eax
valloc:018B00FD                 xchg    ebp, ebp
valloc:018B00FF                 mov     eax, offset off_11A11828
valloc:018B0104                 jmp     loc_18B0112
valloc:018B0109
valloc:018B0109 loc_18B0109:
valloc:018B0109                 adc     ebp, eax
valloc:018B010B                 xchg    ebp, ebp
valloc:018B010D                 mov     eax, offset off_11A11810
valloc:018B0112
valloc:018B0112 loc_18B0112:
valloc:018B0112                 mov     [ebx+srom_vm_context.field_20], eax ; depedning on
[argument + 0xC] == 0x50
```

Oki, now SecuROM will put into vm_context.obsfucated_apis address of APIs used as anti-dump. There you may find PID, cpuid value, GetCurrentProcessId, OpenEventA and CloseHandle. I will show also handlers responsible for their usage, but we will come to that part.

```
valloc:018B0115             xchg    ebp, ebp
valloc:018B0117             adc     ebp, eax
valloc:018B0119             mov     eax, 11A24A8h
valloc:018B011E             mov    [ebx+srom_vm_context.obsfucated_apis], eax
```

Once all fields are set SecuROM will approach to kinda generic way of processing VM_opcodes, at this point it is crucial to know layout of VM_Context struct so you may follow code much better:

```
00000000 srom_vm_context struc ; (sizeof=0x38)
00000000 handlers_delta  dd ?
00000004 vm_kinda_eip    dd ?
00000008 e_flags         dd ?
0000000C vm_enter_address dd ?
00000010 modifier        db ?
00000011                 db ? ; undefined
00000012                 db ? ; undefined
00000013                 db ? ; undefined
00000014 self_context_pointer dd ?
00000018 argument        dd ?
0000001C x86_eflags_regs dd ?
00000020 field_20        dd ?
00000024 busy            dd ?
00000028 obsfucated_apis dd ?
00000038 srom_vm_context ends
```

According to VM_Context it takes delta address of vm_opcodes (vm_kinda_eip):

```
valloc:018B0121                 mov     eax, 22h
valloc:018B0126                 xor     eax, 26h        ; eax = 4
valloc:018B012B                 add     eax, ebx
valloc:018B012D                 mov     eax, [eax]
```

Next thing to be done is to make delta offset valid offset by adding vm_enter address to it. If we take a look at vm_context struct, we may see that this address is located at offset vm_context+0xC:

```
valloc:018B012F                 mov     edx, 60h
valloc:018B0134                 shr     edx, 3          ; edx = 0xC
valloc:018B0137                 add     edx, ebx        ; edx = lea [ebx+c]
valloc:018B0139                 add     eax, [edx]
valloc:018B013B                 mov     edx, eax
```

Once we have valid address of vm_opcodes, SecuROM takes 1st dword from vm_opcodes, which in this paticilar analyzed vm_enter is : `70B2F0C5h .` Oki opcode is loaded into eax and edx. VM gets modifier into cl at line 18B014F (context+0x10), then extracts 1st byte from vm_opcode which in this case is 0C5. That byte (1st from opcode) is used to update modifier for next opcode:

```
valloc:018B013D                 mov     eax, [eax]
valloc:018B013F                 mov     edx, eax

valloc:018B0141                 push    edx
valloc:018B0142                 mov     edi, 28h
valloc:018B0147                 xor     edi, 38h
valloc:018B014D                 add     edi, ebx
valloc:018B014F                 mov     cl, [edi]
valloc:018B0151                 mov     edx, [esp]
valloc:018B0154                 push    ecx
valloc:018B0155                 mov     cl, 0EAh
valloc:018B0157                 xor     cl, 0F2h
valloc:018B015A                 shl     edx, cl
valloc:018B015C                 mov     cl, 1Dh
valloc:018B015E                 xor     cl, 5
valloc:018B0161                 shr     edx, cl
valloc:018B0163                 mov     ecx, 72h
valloc:018B0168                 xor     ecx, 62h
```

```
valloc:018B016E         add     ecx, ebx
valloc:018B0170         add     [ecx], dl
valloc:018B0172         pop     ecx
valloc:018B0173         pop     edx
```

Now SecuROM extracts 2nd byte from vm_opcode which is 0xF0 and updates it with current modifier which in vm_enter is 0x95. This is actually next handler index:

```
valloc:018B0174         shl     edx, 10h
valloc:018B0177         shr     edx, 18h
valloc:018B017A         add     dl, cl
```

Now it will extract vm_handlers_delta and update it with delta address stored at vm_context+0xC:

```
valloc:018B017C         mov     eax, [ebx]
valloc:018B017E         mov     edi, 2Fh
valloc:018B0183         xor     edi, 23h
valloc:018B0189         add     edi, ebx
valloc:018B018B         add     eax, [edi]
```

Next step is to update vm_kinda_eip as job of vm_enter is only to prepare modifier for next opcode, and to call next handler:

```
valloc:018B018D         mov     ecx, 19h
valloc:018B0192         xor     ecx, 1Dh
valloc:018B0198         add     ecx, ebx
valloc:018B019A         add     dword ptr [ecx], 4
```

Size of opcode can vary, but in this case it is 4, basically SecuROM has opcodes of 3 type: 4 bytes, 8 bytes and 10 bytes. 4 byters are opcodes which perform operations on vm_regs only, like mov [vm_reg], vm_reg or add vm_reg, vm_reg, etc… 8 byte opcodes on other hand are used when opcode has immediate value. Such as mov vm_reg, imm or add vm_reg, imm etc… 10 byte long opcodes are jcc. You can always guess size of vm_opcode very fast, obsfucation here doesn't help that much to stop attacker, as after a few minutes you get used to obsfucation and simply you don't even see it.

You should remember that edx has at this pointe next handler index and it is multiplied with 4 to get valid offset into handlers table located at vm_context+0.

```
valloc:018B01A0         shl     edx, 2
valloc:018B01A3         add     edx, eax
```

Above code simply goes to vm_handlers+EDX which will tell us what is next handler to be executed, after that, obsfucated handler with cpuid is pushed onto stack and deobsfucated using cpuid. By analyzing following code you may see simple formula used to calculate handler address:

```
mov eax, 1
cpuid
and eax, 0FFFFFFDFh
mov ecx, eax
ror obsfucated_handler, cl
mov eax, deobsfucated_handler
add eax, [ebx+0xC]   <---  vm_context+0xC (delta)
jmp  eax
```

```
valloc:018B01A5                 push    dword ptr [edx]
valloc:018B01A7                 mov     ecx, 9Ah
valloc:018B01AC                 xor     ecx, 96h
valloc:018B01B2                 add     ecx, ebx
valloc:018B01B4                 mov     eax, [ecx]
valloc:018B01B6                 push    eax
valloc:018B01B7                 mov     eax, 1
valloc:018B01BC                 push    ebx
valloc:018B01BD                 cpuid
valloc:018B01BF                 and     eax, 0FFFFFFDFh
valloc:018B01C4                 pop     ebx
valloc:018B01C5                 mov     cl, al
valloc:018B01C7                 ror     dword ptr [esp+4], cl
valloc:018B01CB                 pop     eax
valloc:018B01CC                 add     [esp], eax
valloc:018B01CF                 pop     eax
valloc:018B01D0                 jmp     eax             ; game continues :)
```

Well list of all handlers you may see if you follow dword stored into vm_context+0 before vm onterpreter makes delta from it.

```
valloc:01860000 vm_handlers1    dd 0FDCFA0FFh
valloc:01860000
valloc:01860004                 dd 0FC155FFFh
valloc:01860008                 dd 0FDE3D7FFh
valloc:0186000C                 dd 0FDC42CFFh
valloc:01860010                 dd 0FDAEDFFFh
valloc:01860014                 dd 0FC7CB0FFh
valloc:01860018                 dd 0FC2D6BFFh
valloc:0186001C                 dd 0FC1AC2FFh
valloc:01860020                 dd 0FC233FFFh
valloc:01860024                 dd 0FD1377FFh
valloc:01860028                 dd 0FD1CBEFFh
valloc:0186002C                 dd 0FC1F79FFh
valloc:01860030                 dd 0FC1494FFh
valloc:01860034                 dd 0FCCF4EFFh
valloc:01860038                 dd 0FD5F26FFh
valloc:0186003C                 dd 0FD5479FFh
valloc:01860040                 dd 0FC107DFFh
```

etc... Total 256 of them, to go fast trough handler addresses I wrote simple code to give me address based on handler index.

```
                        p586
                        .model  flat, stdcall
                        p586
                        .model  flat, stdcall
                        locals
                        jumps

include                 c:\tasm32\include\shitheap.inc
include                 c:\tasm32\include\extern.inc

public C start

                        .data
vm_handlers1     dd 0FDCFA0FFh
                 dd 0FC155FFFh
                 dd 0FDE3D7FFh
                 dd 0FDC42CFFh
                 dd 0FDAEDFFFh, 0FC7CB0FFh, 0FC2D6BFFh, 0FC1AC2FFh,
                   ... more more handlers...
```

```
start:          mov     eax, 1
                cpuid
                and     eax, 0FFFFFFDFh
                mov     cl, al
                mov     eax, 5

                mov     eax, vm_handlers1[eax*4]
                ror     eax, cl
                add     eax, 018B0000h
                nop
                nop
```

Now simply when I want to get in IDA to same handler index fast I simply change mov eax, 5 with proper handler index and can easily get to proper handler in IDA without a problem.

Now let's see how the next handler is called:

- We know what byte is used to update modifier
- We know what byte is used to calculate next handler index

So let's get back to our opcode once more : `70B2F0C5h`

So 1st byte is used to update modifier which is by vm_enter set to 0x95, 2nd byte is used to calculate next handler index:

Next_modifier = 0x95 + 0xC5 = 0x5A
Handler_index = 0xF0 + 0x95 = 0x85

Good, let's put 0x85 in above program and see what is next handler address : 1870000 is what I get on my machine, and handler that we are looking at is :

```
valloc:01870000 loc_1870000:
valloc:01870000                 mov     eax, 1
valloc:01870005                 shl     eax, 2
valloc:01870008 loc_1870008:
valloc:01870008                 add     eax, ebx
valloc:0187000A                 mov     eax, [eax]
valloc:0187000C                 mov     edx, 52h
valloc:01870011                 xor     edx, 5Eh
valloc:01870017                 add     edx, ebx
valloc:01870019                 add     eax, [edx]
valloc:0187001B                 mov     edx, eax
valloc:0187001D                 mov     eax, [eax]
valloc:0187001F                 mov     edx, eax
valloc:01870021                 push    edx
valloc:01870022                 mov     edi, 4
valloc:01870027                 shl     edi, 2
valloc:0187002A                 add     edi, ebx
valloc:0187002C                 mov     cl, [edi]
valloc:0187002E                 mov     edx, [esp]
valloc:01870031                 push    ecx
valloc:01870032                 mov     cl, 22h
valloc:01870034                 xor     cl, 3Ah
valloc:01870037                 shl     edx, cl
valloc:01870039                 mov     cl, 76h
valloc:0187003B                 xor     cl, 6Eh
valloc:0187003E                 shr     edx, cl
valloc:01870040                 pop     ecx
valloc:01870041                 mov     edi, 9Fh
valloc:01870046                 xor     edi, 8Fh
valloc:0187004C                 add     edi, ebx
valloc:0187004E                 add     [edi], dl
valloc:01870050                 mov     edx, [esp]
valloc:01870053                 push    ecx
valloc:01870054                 mov     cl, 1
valloc:01870056                 shl     cl, 3
```

```
valloc:01870059                 shl     eax, cl
valloc:0187005B                 mov     cl, 0C0h
valloc:0187005D                 shr     cl, 3
valloc:01870060                 shr     eax, cl
valloc:01870062                 pop     ecx
valloc:01870063                 ror     al, cl
valloc:01870065                 shl     eax, 2
valloc:01870068                 add     eax, ebx
valloc:0187006A                 mov     eax, [eax]
valloc:0187006C                 push    ecx
valloc:0187006D                 mov     cl, 60h
valloc:0187006F                 shr     cl, 2
valloc:01870072                 shr     edx, cl
valloc:01870074                 pop     ecx
valloc:01870075                 rol     dl, cl
valloc:01870077                 shl     edx, 2
valloc:0187007A                 add     edx, ebx
valloc:0187007C                 mov     [edx], eax
valloc:0187007E                 pop     edx
valloc:0187007F                 push    ecx
valloc:01870080                 mov     cl, 36h
valloc:01870082                 xor     cl, 26h
valloc:01870085                 shl     edx, cl
valloc:01870087                 mov     cl, 0FFh
valloc:01870089                 xor     cl, 0E7h
valloc:0187008C                 shr     edx, cl
valloc:0187008E                 pop     ecx
valloc:0187008F                 add     dl, cl
valloc:01870091                 mov     eax, [ebx]
valloc:01870093                 mov     ecx, 44h
valloc:01870098                 xor     ecx, 48h
valloc:0187009E                 add     ecx, ebx
valloc:018700A0                 add     eax, [ecx]
valloc:018700A2                 mov     ecx, 1
valloc:018700A7                 shl     ecx, 3
valloc:018700AA                 shr     ecx, 1
valloc:018700AC                 add     ecx, ebx
valloc:018700AE                 add     dword ptr [ecx], 4
valloc:018700B4                 shl     edx, 2
valloc:018700B7                 add     edx, eax
valloc:018700B9                 push    edx
valloc:018700BA                 mov     ecx, 72h
valloc:018700BF                 xor     ecx, 7Eh
valloc:018700C5                 add     ecx, ebx
valloc:018700C7                 mov     eax, [ecx]
valloc:018700C9                 mov     edx, [esp]
valloc:018700CC                 mov     edx, [edx]
valloc:018700CE                 mov     [esp], edx
valloc:018700D1                 push    eax
valloc:018700D2                 mov     eax, 1
valloc:018700D7                 push    ebx
valloc:018700D8                 cpuid
valloc:018700DA                 and     eax, 0FFFFFFDFh
valloc:018700DF                 pop     ebx
valloc:018700E0                 mov     cl, al
valloc:018700E2                 ror     dword ptr [esp+4], cl
valloc:018700E6                 pop     eax
valloc:018700E7                 add     [esp], eax
valloc:018700EA                 pop     eax
valloc:018700EB                 jmp     eax
```

Well certainly this looks like a mess, but don't worry it isn't that hard once you spend a few minutes with it ☺ What helps is knowing logic of how vm_handler works:

1. ALWAYS will extract opcode using vm_context.vm_kinda_eip + vm_context.delta (0xC)

2. ALWAYS will extract modifier as it is needed to deobsfucate regs, immediate, next handler index
3. ALWAYS will update modifier for next opcode using random bytes from this opcode. When I say random I mean depending on handler it can be 1st, 2nd, 3rd or 4th byte, and sometimes it is plain add to old modifier, sometimes value is xored and added to old modifier so to write decompiler you will have to analyse all 256 handlers ☺ That's why it is boring ☺
4. ALWAYS will extract next handler index and using current modifier, decide where to jmp.

Following this logic we may split above handler into 4 main parts:

```
valloc:0187002C                 mov     cl, [edi]
valloc:0187002E                 mov     edx, [esp]
valloc:01870031                 push    ecx
valloc:01870032                 mov     cl, 22h
valloc:01870034                 xor     cl, 3Ah
valloc:01870037                 shl     edx, cl
valloc:01870039                 mov     cl, 76h
valloc:0187003B                 xor     cl, 6Eh
valloc:0187003E                 shr     edx, cl
valloc:01870040                 pop     ecx
valloc:01870041                 mov     edi, 9Fh
valloc:01870046                 xor     edi, 8Fh
valloc:0187004C                 add     edi, ebx
valloc:0187004E                 add     [edi], dl
```

We already know that opcode is extracted now question is what byte is used to update modifier for the next opcode:

22 ^ 3A = 0x18 so it takes 1st byte to update modifier!!!

```
valloc:01870054                 mov     cl, 1
valloc:01870056                 shl     cl, 3
valloc:01870059                 shl     eax, cl
valloc:0187005B                 mov     cl, 0C0h
valloc:0187005D                 shr     cl, 3
valloc:01870060                 shr     eax, cl
valloc:01870062                 pop     ecx
valloc:01870063                 ror     al, cl
valloc:01870065                 shl     eax, 2
valloc:01870068                 add     eax, ebx
valloc:0187006A                 mov     eax, [eax]
```

This part is also simple it extracts 3rd byte as vm_reg_src:

1 * 8 = 8   so it extracts 3rd byte as vm_reg_src, but then this extracted byte is deobsfucated with modifier as ror al, cl. Now we know we have index into vm_reg, and to get offset to vm_reg we have to multiply it by 4 which you may see as shl eax, 2. Next step is to get vm_reg offset by adding to eax vm_context offset which is almost always held in ebx register trough all vm_handlers. As this is mov vm_reg, vm_reg. All it has to do now is to take value from vm_reg_src, which you may see as mov eax, [eax]. If we would have mov vm_reg, [vm_reg], you would see something like mov eax, [eax] followed by yet another mov eax, [eax] or similar code which could be obsfucated or not. But as I mentioned after a few minutes of looking at vm_handlers you don't even see obsfucation. You are able to focus on good parts almost instantly.

Now it has to extract vm_reg_dst:

```
valloc:0187006D                 mov     cl, 60h
valloc:0187006F                 shr     cl, 2
valloc:01870072                 shr     edx, cl
valloc:01870074                 pop     ecx
valloc:01870075                 rol     dl, cl
valloc:01870077                 shl     edx, 2
valloc:0187007A                 add     edx, ebx
valloc:0187007C                 mov     [edx], eax
```

60/4 = 18 so it extracts 1st byte as vm_reg_dst index, and deobsfucates it with current modifier as rol dl, cl, next step is to again get offset into vm_reg by multiplying index by 4 and adding to it vm_context. To this address is moved content of vm_reg_src, so this is nothing more then simple mov vm_reg_dst, vm_reg_src. In case of mov [vm_reg], vm_reg you would see that it takes address from vm_reg_dst, something like this:

mov edx, [edx]
mov [edx], eax

Next step is to extract next handler index :

```
valloc:01870080                 mov     cl, 36h
valloc:01870082                 xor     cl, 26h
valloc:01870085                 shl     edx, cl
valloc:01870087                 mov     cl, 0FFh
valloc:01870089                 xor     cl, 0E7h
valloc:0187008C                 shr     edx, cl
valloc:0187008E                 pop     ecx
valloc:0187008F                 add     dl, cl
```

36 ^ 26 = 10h so it takes 2nd byte as next_handler_index. And deobsfucates that byte by adding to it current modifier. At this point you may stop your analyze as you have enough info to decode this handler ☺ Also it is important to know size of opcode, or else your decompiler won't decompile next opcode properly:

```
valloc:01870091                 mov     eax, [ebx]
valloc:01870093                 mov     ecx, 44h
valloc:01870098                 xor     ecx, 48h
valloc:0187009E                 add     ecx, ebx
valloc:018700A0                 add     eax, [ecx]
valloc:018700A2                 mov     ecx, 1
valloc:018700A7                 shl     ecx, 3
valloc:018700AA                 shr     ecx, 1
valloc:018700AC                 add     ecx, ebx
valloc:018700AE                 add     dword ptr [ecx], 4
valloc:018700B4                 shl     edx, 2
```

[ecx] is actually vm_context+0x4 or vm_context.vm_kinda_eip, so vm_eip is incremented by 4 which means that size of this opcode is 4. Actually data handled by this handler is 4. We may not call vm_opcodes, opcodes as opcode is something which can be executed by CPU depending on it's format. Here, it is possible that same data has different meaning depending on order of it's execution, because next opcode to be executed depends on previous opcode and next handler index used in it, and also depending on modifier update in previous handler.

Oki, let's see some handler which is not obsfucated so you may see in clean example without obsfucation how it works:

```
valloc:01872BFC                 mov     eax, [ebx+4]
valloc:01872BFF                 add     eax, [ebx+0Ch]
valloc:01872C02                 mov     edx, eax
valloc:01872C04                 mov     eax, [eax]    <-- get opcode into eax
valloc:01872C06                 mov     edx, eax      <-- save into edx
valloc:01872C08                 push    edx
```

```
valloc:01872C09                 mov     cl, [ebx+10h]
valloc:01872C0C                 mov     edx, [esp]
valloc:01872C0F                 shl     edx, 18h
valloc:01872C12                 shr     edx, 18h
valloc:01872C15                 add     [ebx+10h], dl       <-- 1st to update mod
valloc:01872C18                 mov     edx, [esp]
valloc:01872C1B                 shr     edx, 18h            <-- 4th as vm_reg_dst
valloc:01872C1E                 rol     dl, cl              <-- deobsfucate it
valloc:01872C20                 lea     edi, [ebx+edx*4]    <-- offset of vm_reg_src
valloc:01872C23                 sub     ebx, 81723h
valloc:01872C29                 sub     ebx, 712h
valloc:01872C2F                 pop     edx
valloc:01872C30                 shl     edx, 10h            <-- next handler as 2nd byte
valloc:01872C33                 shr     edx, 18h
valloc:01872C36                 add     dl, cl
valloc:01872C38                 mov     eax, [ebx+81E35h]
valloc:01872C3E                 add     eax, [ebx+81E41h]

valloc:01872C44                 add     dword ptr [ebx+81E39h], 4 <-- size of opcode is 4

valloc:01872C4E                 add     ebx, 712h
valloc:01872C54                 add     ebx, 81723h
valloc:01872C5A                 pop     dword ptr [edi]             <-- pop vm_reg
valloc:01872C5C                 push    dword ptr [eax+edx*4]       <-- get obsfucated handler
valloc:01872C5F                 mov     eax, [ebx+0Ch]
valloc:01872C62                 push    eax
valloc:01872C63                 mov     eax, 1
valloc:01872C68                 push    ebx
valloc:01872C69                 cpuid
valloc:01872C6B                 and     eax, 0FFFFFFDFh
valloc:01872C70                 pop     ebx
valloc:01872C71                 mov     cl, al
valloc:01872C73                 ror     dword ptr [esp+4], cl
valloc:01872C77                 pop     eax
valloc:01872C78                 add     [esp], eax
valloc:01872C7B                 pop     eax
valloc:01872C7C                 jmp     eax                 <-- go to next handler
```

Next thing that I wanna show is one handler where is used immediate value and which has size of 8 bytes:

Yeah, you may already see that it is long code but it is very simply at bottom line:

```
valloc:0188CFA0                 sub     esp, 4
valloc:0188CFA6                 mov     [esp], esi
valloc:0188CFA9                 xor     esi, [ebx+4]
valloc:0188CFAC                 xor     esi, [esp]
valloc:0188CFAF                 add     esp, 4
valloc:0188CFB5                 sub     esp, 4
valloc:0188CFBB                 mov     [esp], edi
valloc:0188CFBE                 sub     esp, 4
valloc:0188CFC4                 mov     dword ptr [esp], 4131h
valloc:0188CFCB                 pop     edi
valloc:0188CFCC                 push    edx
valloc:0188CFCD                 mov     edx, 36D6h
valloc:0188CFD2                 xor     edx, 36DBh
valloc:0188CFD8                 add     edi, edx
valloc:0188CFDA                 pop     edx
valloc:0188CFDB                 xor     edi, 4132h
valloc:0188CFE1                 sub     esp, 4
valloc:0188CFE7                 mov     [esp], ebx
valloc:0188CFEA                 add     [esp], edi
valloc:0188CFED                 pop     ebx
valloc:0188CFEE                 pop     edi
valloc:0188CFEF                 add     esi, [ebx]
```

```
valloc:0188CFF1                 push    1207h
valloc:0188CFF6                 xor     dword ptr [esp], 34B3h
valloc:0188CFFD                 add     ebx, [esp]
valloc:0188D000                 add     esp, 4
valloc:0188D006                 add     ebx, 0FFFFD7D7h
valloc:0188D00C                 sub     ebx, 0D633h
valloc:0188D012                 sub     esp, 4
valloc:0188D018                 mov     [esp], ebp
valloc:0188D01B                 sub     esp, 4
valloc:0188D021                 mov     dword ptr [esp], 0D9D3h
valloc:0188D028                 pop     ebp
valloc:0188D029                 add     ebp, 2776h
valloc:0188D02F                 sub     ebp, 29ADh
valloc:0188D035                 push    ebp
valloc:0188D036                 add     ebx, [esp]
valloc:0188D039                 pop     ebp
valloc:0188D03A                 pop     ebp
valloc:0188D03B                 sub     esp, 4
valloc:0188D041                 mov     [esp], edi
valloc:0188D044                 xor     edi, [esi+4]
valloc:0188D047                 xor     edi, [esp]
valloc:0188D04A                 add     esp, 4
valloc:0188D050                 sub     esp, 4
valloc:0188D056                 mov     [esp], esi
valloc:0188D059                 xor     esi, [esi]
valloc:0188D05B                 xor     esi, [esp]
valloc:0188D05E                 add     esp, 4
valloc:0188D064                 mov     cl, [ebx+10h]
valloc:0188D067                 sub     esp, 4
valloc:0188D06D                 mov     [esp], esi
valloc:0188D070                 pop     eax
valloc:0188D071                 shl     eax, 8
valloc:0188D074                 shr     eax, 18h
valloc:0188D077                 xor     al, 3Eh
valloc:0188D079                 add     [ebx+10h], al
valloc:0188D07C                 sub     esp, 4
valloc:0188D082                 mov     [esp], eax
valloc:0188D085                 xor     eax, edi
valloc:0188D087                 xor     eax, [esp]
valloc:0188D08A                 add     esp, 4
valloc:0188D090                 xor     eax, 1DF08D0h
valloc:0188D095                 sub     esp, 4
valloc:0188D09B                 mov     [esp], eax
valloc:0188D09E                 sub     esp, 4
valloc:0188D0A4                 mov     [esp], esi
valloc:0188D0A7                 pop     eax
valloc:0188D0A8                 shl     eax, 10h
valloc:0188D0AB                 shr     eax, 18h
valloc:0188D0AE                 sub     al, cl
valloc:0188D0B0                 xor     al, 31h
valloc:0188D0B2                 shl     eax, 2
valloc:0188D0B5                 sub     esp, 4
valloc:0188D0BB                 mov     [esp], eax
valloc:0188D0BE                 add     [esp], ebx
valloc:0188D0C1                 pop     eax
valloc:0188D0C2                 sub     esp, 4
valloc:0188D0C8                 mov     [esp], eax
valloc:0188D0CB                 pop     edi
valloc:0188D0CC                 pop     eax
valloc:0188D0CD                 push    8F17h
valloc:0188D0D2                 xor     dword ptr [esp], 9671h
valloc:0188D0D9                 add     ebx, [esp]
valloc:0188D0DC                 add     esp, 4
valloc:0188D0E2                 sub     esp, 4
valloc:0188D0E8                 mov     [esp], eax
valloc:0188D0EB                 mov     eax, 0FA2Ah
valloc:0188D0F0                 add     eax, 5Dh
```

```
valloc:0188D0F5                 xor     eax, 0E3D9h
valloc:0188D0FA                 sub     ebx, eax
valloc:0188D0FC                 pop     eax
valloc:0188D0FD                 push    dword ptr [ebx]
valloc:0188D0FF                 pop     edx
valloc:0188D100                 sub     esp, 4
valloc:0188D106                 mov     [esp], esi
valloc:0188D109                 push    9FC8h
valloc:0188D10E                 pop     esi
valloc:0188D10F                 push    ebp
valloc:0188D110                 mov     ebp, 0F642h
valloc:0188D115                 sub     ebp, 0F579h
valloc:0188D11B                 sub     esi, ebp
valloc:0188D11D                 pop     ebp
valloc:0188D11E                 sub     esp, 4
valloc:0188D124                 mov     [esp], edi
valloc:0188D127                 mov     edi, 12F70h
valloc:0188D12C                 add     edi, 29h
valloc:0188D132                 push    esi
valloc:0188D133                 mov     esi, 0FFFFF638h
valloc:0188D138                 add     esi, 9A6Ah
valloc:0188D13E                 sub     edi, esi
valloc:0188D140                 pop     esi
valloc:0188D141                 sub     esi, edi
valloc:0188D143                 pop     edi
valloc:0188D144                 sub     ebx, esi
valloc:0188D146                 pop     esi
valloc:0188D147                 sub     esp, 4
valloc:0188D14D                 mov     [esp], edx
valloc:0188D150                 popf
valloc:0188D151                 sub     [edi], eax
valloc:0188D153                 pushf
valloc:0188D154                 pop     dword ptr [ebx+8]
valloc:0188D157                 sub     esp, 4
valloc:0188D15D                 mov     [esp], eax
valloc:0188D160                 xor     eax, esi
valloc:0188D162                 xor     eax, [esp]
valloc:0188D165                 add     esp, 4
valloc:0188D16B                 shl     eax, 0
valloc:0188D16E                 shr     eax, 18h
valloc:0188D171                 ror     al, cl
valloc:0188D173                 xor     al, 0C2h
valloc:0188D175                 shl     eax, 2
valloc:0188D178                 push    dword ptr [ebx]
valloc:0188D17A                 pop     edi
valloc:0188D17B                 push    0FB8Dh
valloc:0188D180                 xor     dword ptr [esp], 0FB81h
valloc:0188D187                 add     ebx, [esp]
valloc:0188D18A                 add     esp, 4
valloc:0188D190                 add     edi, [ebx]
valloc:0188D192                 sub     esp, 4
valloc:0188D198                 mov     [esp], ecx
valloc:0188D19B                 push    64Eh
valloc:0188D1A0                 pop     ecx
valloc:0188D1A1                 push    eax
valloc:0188D1A2                 mov     eax, 0E62Dh
valloc:0188D1A7                 sub     eax, 0E60Fh
valloc:0188D1AC                 sub     ecx, eax
valloc:0188D1AE                 pop     eax
valloc:0188D1AF                 xor     ecx, 63Ch
valloc:0188D1B5                 sub     ebx, ecx
valloc:0188D1B7                 pop     ecx
valloc:0188D1B8                 sub     esp, 4
valloc:0188D1BE                 mov     [esp], eax
valloc:0188D1C1                 add     edi, [esp]
valloc:0188D1C4                 pop     eax
valloc:0188D1C5                 sub     esp, 4
```

```
valloc:0188D1CB                 mov     [esp], edi
valloc:0188D1CE                 xor     edi, [edi]
valloc:0188D1D0                 xor     edi, [esp]
valloc:0188D1D3                 add     esp, 4
valloc:0188D1D9                 add     dword ptr [ebx+4], 8
valloc:0188D1E0                 sub     esp, 4
valloc:0188D1E6                 mov     dword ptr [esp], 110h
valloc:0188D1ED                 pop     eax
valloc:0188D1EE                 xor     eax, 22h
valloc:0188D1F3                 sub     esp, 4
valloc:0188D1F9                 mov     [esp], ecx
valloc:0188D1FC                 mov     ecx, 0FFFFFF60h
valloc:0188D201                 sub     ecx, 44h
valloc:0188D207                 sub     ecx, 0D334h
valloc:0188D20D                 add     ecx, 0D49Ah
valloc:0188D213                 sub     eax, ecx
valloc:0188D215                 pop     ecx
valloc:0188D216                 sub     esp, 4
valloc:0188D21C                 mov     [esp], edi
valloc:0188D21F                 sub     esp, 4
valloc:0188D225                 mov     dword ptr [esp], 125h
valloc:0188D22C                 pop     edi
valloc:0188D22D                 xor     edi, 5
valloc:0188D233                 push    esi
valloc:0188D234                 mov     esi, 0FFFFE9DEh
valloc:0188D239                 add     esi, 1693h
valloc:0188D23F                 sub     edi, esi
valloc:0188D241                 pop     esi
valloc:0188D242                 sub     eax, edi
valloc:0188D244                 pop     edi
valloc:0188D245                 sub     esp, 4
valloc:0188D24B                 mov     [esp], ebx
valloc:0188D24E                 cpuid
valloc:0188D250                 and     eax, 0FFFFFFDFh
valloc:0188D255                 pop     ebx
valloc:0188D256                 mov     cl, al
valloc:0188D258                 ror     edi, cl
valloc:0188D25A                 sub     esp, 4
valloc:0188D260                 mov     [esp], ebp
valloc:0188D263                 sub     esp, 4
valloc:0188D269                 mov     dword ptr [esp], 5E02h
valloc:0188D270                 pop     ebp
valloc:0188D271                 xor     ebp, 56h
valloc:0188D277                 push    esi
valloc:0188D278                 mov     esi, 0C1CFh
valloc:0188D27D                 sub     esi, 6387h
valloc:0188D283                 sub     ebp, esi
valloc:0188D285                 pop     esi
valloc:0188D286                 push    ebp
valloc:0188D287                 add     ebx, [esp]
valloc:0188D28A                 pop     ebp
valloc:0188D28B                 pop     ebp
valloc:0188D28C                 push    dword ptr [ebx]
valloc:0188D28E                 add     edi, [esp]
valloc:0188D291                 add     esp, 4
valloc:0188D297                 sub     esp, 4
valloc:0188D29D                 mov     [esp], eax
valloc:0188D2A0                 push    0FFFF261Fh
valloc:0188D2A5                 sub     dword ptr [esp], 0E1h
valloc:0188D2AC                 pop     eax
valloc:0188D2AD                 sub     eax, 1024h
valloc:0188D2B2                 push    edi
valloc:0188D2B3                 mov     edi, 12CDCh
valloc:0188D2B8                 sub     edi, 41EAh
valloc:0188D2BE                 add     eax, edi
valloc:0188D2C0                 pop     edi
valloc:0188D2C1                 sub     ebx, eax
```

```
valloc:0188D2C3                 pop     eax
valloc:0188D2C4                 jmp     edi
```

1st we locate size of opcode:

```
valloc:0188D1D9                 add     dword ptr [ebx+4], 8
```

Then we get byte used to update modifier and how it is done:

```
valloc:0188D064                 mov     cl, [ebx+10h]
valloc:0188D067                 sub     esp, 4
valloc:0188D06D                 mov     [esp], esi
valloc:0188D070                 pop     eax
valloc:0188D071                 shl     eax, 8
valloc:0188D074                 shr     eax, 18h
valloc:0188D077                 xor     al, 3Eh
valloc:0188D079                 add     [ebx+10h], al
```

Ok 3rd byte from opcode is used to extract and update modifier, note how byte is xored before modifier is updated.

Next thing is to deobsfucate immediate value which is actually 2nd dword of vm_opcode:

```
valloc:0188D090                 xor     eax, 1DF08D0h
```

Well this handler has simple deobsfucation of immediate, in some handlers deobsfucation would look like this:

```
mov     edx, [esi+4]
ror     dl, cl
ror     edx, 8
ror     dl, cl
ror     edx, 8
ror     dl, cl
ror     edx, 8
ror     dl, cl
ror     edx, 8

mov     edi, 87C5573Ah
shr     edi, 1
xor     edx, edi
mov     edi, edx
```

Ok next thing is to find targeted vm_reg, and see what operation we have here:

```
valloc:0188D0A8                 shl     eax, 10h
valloc:0188D0AB                 shr     eax, 18h
valloc:0188D0AE                 sub     al, cl
valloc:0188D0B0                 xor     al, 31h
valloc:0188D0B2                 shl     eax, 2
valloc:0188D0B5                 sub     esp, 4
valloc:0188D0BB                 mov     [esp], eax
valloc:0188D0BE                 add     [esp], ebx
valloc:0188D0C1                 pop     eax
valloc:0188D0C2                 sub     esp, 4
valloc:0188D0C8                 mov     [esp], eax
valloc:0188D0CB                 pop     edi
```

Also it is very common to see that offset of vm_reg_dst is moved to edi, remember that at this time eax has deobsfucated imm value:

If you analyze code which is between reg offset extraction and this code, you will see that handler will move [ebx+8] to edx, and from our listed struct that is eflags used to in VM:

```
valloc:0188D141                 sub     esi, edi
valloc:0188D143                 pop     edi
valloc:0188D144                 sub     ebx, esi
valloc:0188D146                 pop     esi
valloc:0188D147                 sub     esp, 4
valloc:0188D14D                 mov     [esp], edx     <-- load eflags
valloc:0188D150                 popf
valloc:0188D151                 sub     [edi], eax     <-- perform op
valloc:0188D153                 pushf
valloc:0188D154                 pop     dword ptr [ebx+8] <-- save eflags
```

Bingo, now we know that this is:

sub [vm_regXXX], immediate value

Our task is done, all we have to do now is to calculate next handler index:

```
valloc:0188D16B                 shl     eax, 0
valloc:0188D16E                 shr     eax, 18h
valloc:0188D171                 ror     al, cl
valloc:0188D173                 xor     al, 0C2h
valloc:0188D175                 shl     eax, 2
```

This is 4th byte extract from vm_opcode to calculate index of next handler. Now you have all you need to properly decode this opcode:

1. You know how immediate value is extracted
2. You know how modifier is updated for next opcode
3. You know what vm_reg is used
4. You know what operation is executed
5. You know what byte is used to get next handler index

So what's stopping you from writing own handler interpreter? Absolutely nothing ☺

You MUST keep track of modifier, as current handler will always update modifier for the next handler, which will be used to properly decode opcode.

Last example of vm_decoding is jcc emulation which is always 10 bytes. Dunno why, as jccs could be also stored as 8 bytes, you will see why in a second:

```
valloc:018725EE                 mov     eax, 400h
valloc:018725F3                 shr     eax, 8
valloc:018725F6                 add     eax, ebx
valloc:018725F8                 mov     eax, [eax]
valloc:018725FA                 mov     ecx, 0FDh
valloc:018725FF                 xor     ecx, 0F1h
valloc:01872605                 add     ecx, ebx
valloc:01872607                 add     eax, [ecx]
valloc:01872609                 mov     ecx, 0EDDEDDEDh
valloc:0187260E                 xor     ecx, 0EDDEDDE5h
valloc:01872614                 add     ecx, eax
valloc:01872616                 movzx   esi, word ptr [ecx]
valloc:01872619                 mov     ecx, 8
valloc:0187261E                 xor     ecx, 0Ch
```

```
valloc:01872624                 add     ecx, eax
valloc:01872626                 mov     edx, [ecx]
valloc:01872628                 mov     eax, [eax]
valloc:0187262A                 xor     ecx, ecx
valloc:0187262C                 mov     cl, 82h
valloc:0187262E                 xor     cl, 92h
valloc:01872631                 add     ecx, ebx
valloc:01872633                 mov     cl, [ecx]
valloc:01872635                 ror     dl, cl
valloc:01872637                 push    ecx
valloc:01872638                 mov     ecx, 1
valloc:0187263D                 shl     ecx, 3
valloc:01872640                 ror     edx, cl
valloc:01872642                 pop     ecx
valloc:01872643                 ror     dl, cl
valloc:01872645                 push    ecx
valloc:01872646                 mov     ecx, 55h
valloc:0187264B                 xor     ecx, 5Dh
valloc:01872651                 ror     edx, cl
valloc:01872653                 pop     ecx
valloc:01872654                 ror     dl, cl
valloc:01872656                 push    ecx
valloc:01872657                 mov     ecx, 11h
valloc:0187265C                 xor     ecx, 19h
valloc:01872662                 ror     edx, cl
valloc:01872664                 pop     ecx
valloc:01872665                 ror     dl, cl
valloc:01872667                 push    ecx
valloc:01872668                 mov     ecx, 20h
valloc:0187266D                 shr     ecx, 2
valloc:01872670                 ror     edx, cl
valloc:01872672                 pop     ecx
valloc:01872673                 mov     edi, 87C5573Ah
valloc:01872678                 shr     edi, 1
valloc:0187267A                 xor     edx, edi
valloc:0187267C                 btr     dword ptr [ebx+8], 6
valloc:01872681                 jnb     loc_18726A3
valloc:01872687                 add     [ebx+4], edx
valloc:0187268A                 mov     edx, esi
valloc:0187268C                 mov     eax, esi
valloc:0187268E                 shr     edx, 8
valloc:01872691                 rol     dl, cl
valloc:01872693                 shl     eax, 18h
valloc:01872696                 shr     eax, 18h
valloc:01872699                 ror     al, cl
valloc:0187269B                 add     [ebx+10h], al
valloc:0187269E                 jmp     loc_18726C0

valloc:018726A3
valloc:018726A3 loc_18726A3:
valloc:018726A3                 push    eax
valloc:018726A4                 shl     eax, 10h
valloc:018726A7                 shr     eax, 18h
valloc:018726AA                 add     al, cl
valloc:018726AC                 movzx   edx, al
valloc:018726AF                 pop     eax
valloc:018726B0                 shl     eax, 18h
valloc:018726B3                 shr     eax, 18h
valloc:018726B6                 add     [ebx+10h], al
valloc:018726B9                 add     dword ptr [ebx+4], 0Ah
valloc:018726C0
valloc:018726C0 loc_18726C0:
valloc:018726C0                 mov     eax, [ebx]
valloc:018726C2                 add     eax, [ebx+0Ch]
valloc:018726C5                 push    dword ptr [eax+edx*4]
valloc:018726C8                 mov     eax, [ebx+0Ch]
valloc:018726CB                 push    eax
```

```
valloc:018726CC                 mov     eax, 1
valloc:018726D1                 push    ebx
valloc:018726D2                 cpuid
valloc:018726D4                 and     eax, 0FFFFFFDFh
valloc:018726D9                 pop     ebx
valloc:018726DA                 mov     cl, al
valloc:018726DC                 ror     dword ptr [esp+4], cl
valloc:018726E0                 pop     eax
valloc:018726E1                 add     [esp], eax
valloc:018726E4                 pop     eax
valloc:018726E5                 jmp     eax
```

Oki let's split this jcc emulation fast:

1st we may see that vm_kinda_eip is update different depending on result of taken/not taken jcc.

Taken:

```
valloc:01872687                 add     [ebx+4], edx
```

Not Taken:

```
valloc:018726B9                 add     dword ptr [ebx+4], 0Ah
```

If jcc is taken, 2nd dword is deobsfucated and used as offset for new opcode:

```
valloc:01872635                 ror     dl, cl
valloc:01872637                 push    ecx
valloc:01872638                 mov     ecx, 1
valloc:0187263D                 shl     ecx, 3
valloc:01872640                 ror     edx, cl
valloc:01872642                 pop     ecx
valloc:01872643                 ror     dl, cl
valloc:01872645                 push    ecx
valloc:01872646                 mov     ecx, 55h
valloc:0187264B                 xor     ecx, 5Dh
valloc:01872651                 ror     edx, cl
valloc:01872653                 pop     ecx
valloc:01872654                 ror     dl, cl
valloc:01872656                 push    ecx
valloc:01872657                 mov     ecx, 11h
valloc:0187265C                 xor     ecx, 19h
valloc:01872662                 ror     edx, cl
valloc:01872664                 pop     ecx
valloc:01872665                 ror     dl, cl
valloc:01872667                 push    ecx
valloc:01872668                 mov     ecx, 20h
valloc:0187266D                 shr     ecx, 2
valloc:01872670                 ror     edx, cl
valloc:01872672                 pop     ecx
valloc:01872673                 mov     edi, 87C5573Ah
valloc:01872678                 shr     edi, 1
valloc:0187267A                 xor     edx, edi
```

Actually this is the code I showed earlier in more readable form with ror dl,cl/ror edx,8. And yes same code is used to deobsfucate next_eip for jcc, and to extract immediate value for some 8 bytes long opcodes.

If jcc is taken then 9th and 10th bytes are used to calculate next_modifier and next_handler_index, otherwise, if jcc is not taken then bytes from 1-4 can be used to calculate next_handler_index and next_modifier. Basically SecuROM developers could put everything here into 8 bytes as they only need 2 bytes in case that jcc is taken, and 2 if not taken, + 4 bytes as next_eip offset. So yes, this opcode can be "compressed" to 8 bytes.

So if jcc is taken this is hos modifier and new handler index are calculated:

```
valloc:0187268A                 mov     edx, esi
valloc:0187268C                 mov     eax, esi
valloc:0187268E                 shr     edx, 8
valloc:01872691                 rol     dl, cl
valloc:01872693                 shl     eax, 18h
valloc:01872696                 shr     eax, 18h
valloc:01872699                 ror     al, cl
valloc:0187269B                 add     [ebx+10h], al
```

10th byte as next_handler_index
9th byte as next_modifier

If jcc is not taken:

```
valloc:018726A3                 push    eax
valloc:018726A4                 shl     eax, 10h
valloc:018726A7                 shr     eax, 18h
valloc:018726AA                 add     al, cl
valloc:018726AC                 movzx   edx, al
valloc:018726AF                 pop     eax
valloc:018726B0                 shl     eax, 18h
valloc:018726B3                 shr     eax, 18h
valloc:018726B6                 add     [ebx+10h], al
```

2nd byte as next_handler_index
1st byte to update next_modifier

Simple? Heh, now only question left to go over is what jcc is emulated?

```
valloc:0187267C                 btr     dword ptr [ebx+8], 6
valloc:01872681                 jnb     loc_18726A3
```

6th bit is Z flag, so BTR will reset this bit and set CF with value 1 if 6th bit (ZF) was set. JNB is taken when CF is not set, so this is jz emulation. ZF = 1 -> BTR eflags, 6 -> CF = 1, jnb jumps if CF = 0, as CF = 1, jnb is not taken, but next eip is updated only then with deobsfucated value, so this is jz emulation. Well there are only 4 jcc emulations. You will find them if you really wann deal with this VM. Maybe it seems complicated, but it's not. Only takes 1min (or less) to analyze handler and to get all info you need from it.

As a bonus I will cover some anti-dump handlers too.

Oki first we have anti-dump cpuid check handler. (only important part of it is presented):

```
valloc:01871A50                 mov     ecx, 19h
valloc:01871A55                 xor     ecx, 1Dh
valloc:01871A5B                 add     ecx, ebx
valloc:01871A5D                 add     dword ptr [ecx], 4
valloc:01871A63                 push    eax
valloc:01871A64                 push    edx
valloc:01871A65                 push    ecx
valloc:01871A66                 mov     edx, [ebx+28h]
valloc:01871A69                 mov     edx, [edx]
valloc:01871A6B                 sub     edx, 38271Eh
valloc:01871A71                 mov     eax, 1
```

```
valloc:01871A76                 push    ebx
valloc:01871A77                 push    edx
valloc:01871A78                 cpuid
valloc:01871A7A                 and     eax, 0FFFFFFDFh
valloc:01871A7F                 pop     edx
valloc:01871A80                 pop     ebx
valloc:01871A81                 sub     edx, eax
valloc:01871A83                 pop     ecx
valloc:01871A84                 add     [ecx], edx
```

ecx = vm_context.vm_kinda_eip
ebx+28 = vm_context.obsfucated_apis

So it takes obsfucated value of cpuid, and subtracts from it current cpuid which should be 0, and result of substraction is added to **vm_context.vm_kinda_eip**

Next anti-dump is OpenEventA/CloseHandle:

```
valloc:01871B2E                 mov     ecx, 19h
valloc:01871B33                 xor     ecx, 1Dh
valloc:01871B39                 add     ecx, ebx
valloc:01871B3B                 add     dword ptr [ecx], 4
valloc:01871B41                 push    eax
valloc:01871B42                 push    edx
valloc:01871B43                 push    ecx
valloc:01871B44                 mov     edx, [ebx+28h]
valloc:01871B47                 add     edx, 8
valloc:01871B4D                 mov     edx, [edx]
valloc:01871B4F                 xor     edx, 4144DDB9h
valloc:01871B55                 call    loc_1871B7B
valloc:01871B55
valloc:01871B5A a0e11515d2a3624ea5783404 db '0E11515D2A3624EA57834044BE8B39EF',0
valloc:01871B7B
valloc:01871B7B
valloc:01871B7B loc_1871B7B:
valloc:01871B7B                 push    0
valloc:01871B80                 push    2
valloc:01871B85                 call    edx
valloc:01871B87                 push    eax
valloc:01871B88                 mov     edx, [ebx+28h]
valloc:01871B8B                 add     edx, 14h
valloc:01871B91                 mov     edx, [edx]
valloc:01871B93                 xor     edx, 0CDA3D1CBh
valloc:01871B99                 call    edx
valloc:01871B9B                 mov     edx, eax
valloc:01871B9D                 sub     edx, 1
valloc:01871BA3                 pop     ecx
valloc:01871BA4                 add     [ecx], edx
```

Pretty much same as above, it opens event with certain name which you may see after call, of course, OpenEventA is obsfucated, then it takes obsfucated CloseHandle, deobsfucates it, and on success CloseHandle will return 1, so it sub edx, 1 should be 0, result of this substraction is added to **vm_context.vm_kinda_eip**

Next anti-dump is GetCurrentProcessId:

```
valloc:01871967                 mov     ecx, 19h
valloc:0187196C                 xor     ecx, 1Dh
valloc:01871972                 add     ecx, ebx
valloc:01871974                 add     dword ptr [ecx], 4
valloc:0187197A                 push    eax
valloc:0187197B                 push    edx
valloc:0187197C                 push    ecx
```

```
valloc:0187197D                 mov     ecx, [ebx+28h]
valloc:01871980                 add     ecx, 10h
valloc:01871986                 mov     ecx, [ecx]
valloc:01871988                 xor     ecx, 3EA19F42h
valloc:0187198E                 mov     edx, [ebx+28h]
valloc:01871991                 add     edx, 0Ch
valloc:01871997                 mov     edx, [edx]
valloc:01871999                 xor     edx, 5DB070DAh
valloc:0187199F                 call    edx
valloc:018719A1                 mov     edx, eax
valloc:018719A3                 sub     edx, ecx
valloc:018719A5                 pop     ecx
valloc:018719A6                 add     [ecx], edx
```

Well pretty much same as above, it takes obsfucated pid from vm_context.obsfucated_apis and substracts from current PID, the one saved in vm_context, and then result of substraction, which in case that program isn't dumped, should be 0. Thre is also anti-trace checks with famous **push ss/pop ss/pushfd** but that isn't very important for us, as we are not singlesteping code.

Ok let's not focus on vm_decompiling and get clear menaning of some code executed in the target.

5. VM understanding

I will go back to example from which we have started at WinMain and analyze what is going on, for one jmp bridge, also we will cover some instruction emulation, and API redirection:

Oki, as reminder here we go again with WinMain code:

```
.text:10912B10 ; int __stdcall WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR
lpCmdLine,int nShowCmd)
.text:10912B10 _WinMain@16     proc near               .text:10912B10
.text:10912B10 var_C           = dword ptr -0Ch
.text:10912B10
.text:10912B10
.text:10912B10                 jmp     ds:off_11E0515C
.text:10912B10 _WinMain@16     endp
```

And when we decompile opcodes stored here:

```
(85) mov vm_reg14C, vm_reg190
(11) mov vm_reg14C, 11E0B648h
(88) mov vm_reg48, 11B0A5B4h
(A8) add vm_reg48, 00000060h
(97) mov vm_reg48, [vm_reg48]
(F9) mov vm_reg4C, 010FC5E4h
(A3) add vm_reg4C, vm_reg48
(AA) mov vm_reg48, 00000140h
(4D) cmp [vm_reg4C], vm_reg48
(BD) jb emulation
(6C) mov vm_reg15C, 960AA921h
(23) mov [vm_reg14C], vm_reg15C
(DA) mov vm_reg190, 11E0515C
(11) mov vm_reg15C, 11E1A820h
(EB) mov [vm_reg190], vm_reg15C
(1B) mov vm_reg14C, vm_reg1C
(0A) add vm_reg14C, 00000024h
(DB) mov vm_reg15C, 11E1A820
(EE) mov [vm_reg14C], vm_reg15C
vm_exit
vm_trace_done
```

```
vm_reg48 = 11B0A5B4h + 60h = 11B0A614
vm_reg48 = [vm_reg48] = 10900000 (imagebase)
vm_reg4C = 10FC5E4h + imagebase = 119FC5E4h
vm_reg48 = 140h
cmp [vm_reg4C], 140 => [vm_reg4C] = 106
jb is taken as 106 is below 140 ☺
```

You might wonder why the hack is here mov [vm_reg14C], 960AA921h? Well I'll come to that part very soon, as I exlain this small vm_code:

```
15C
vm_reg190 = 11E0515Ch <-- offset used in jmp dword ptr[]
vm_reg15C = 11E1A820h <-- new address to be stored there which is very common for securom jmp redirections

mov [vm_reg190], vm_reg15C = mov [11E0515Ch], 11E1A820h

mov vm_reg14C, vm_reg1C <-- vm_reg1C = vm_context.x86_eflags_registers
add vm_reg14C, 24h      <-- offset to x86 EIP
mov vm_reg15C, 11E1A820h <-- same address write to jmp bridge
mov [vm_reg14C], vm_reg15C <-- mov x86_eip, 11E1A820h
```

Done, vm exits and transfers execution to the 11E1A820h:

```
.bla:11E1A820                push    ebp
.bla:11E1A821                mov     ebp, esp
.bla:11E1A823                push    esi
.bla:11E1A824                push    offset loc_11E1A833
.bla:11E1A829                push    offset sub_10CC4660
.bla:11E1A82E                retn
.bla:11E1A82F                dd 0F959801Dh
.bla:11E1A833 loc_11E1A833:
.bla:11E1A833                mov     esi, eax
.bla:11E1A835                push    offset off_10FCDBA8
.bla:11E1A83A                push    esi
.bla:11E1A83B                push    offset loc_11E1A855
.bla:11E1A840                jmp     sub_10CA9740
.bla:11E1A845                dd 0C5EAF073h
.bla:11E1A849                dd 143932B3h
.bla:11E1A84D                dd 63886112h
.bla:11E1A851                dd 53A73E09h
.bla:11E1A855 loc_11E1A855:
.bla:11E1A855                add     esp, 8
.bla:11E1A858                test    eax, eax
.bla:11E1A85A                jz      short loc_11E1A8A4
.bla:11E1A85C                push    eax
.bla:11E1A85D                pushf
.bla:11E1A85E                mov     eax, 87497261h
.bla:11E1A863                nop
.bla:11E1A864                push    esp
.bla:11E1A865                nop
.bla:11E1A866                push    offset word_11E0495A
.bla:11E1A86B                xor     eax, ds:dword_11E0B648
.bla:11E1A871                nop
.bla:11E1A872                call    eax
.bla:11E1A874                popf
.bla:11E1A875                xchg    eax, [esp]
.bla:11E1A878                push    offset sub_11E1A887
.bla:11E1A87D                jmp     short sub_11E1A887
```

Now in this small code I want you to focus on lines between 11E1A85C and 11E1A878. This is call to so-called recursive VM. Also take a closer look how value of eax is calculated ☺ Now look in vm disassembly, and you will see that mov [vm_reg14C], 960AA921h is actually setting this dword with proper value which, of course, before jmp bridge is executed, was zero ☺ If jmp bridge wasn't executed you wouldn't know where is recursive vm located.

While I was playing with SecuROM 7.34, I've found one new anti-dump in it. Very neet, but not problem to remove, you will have to think how YOU would remove it. Ok here comes disassembly of an interesting code part:

```
.text:1090F70E                  mov     vm_data, offset vm_opcodes
.text:1090F718                  mov     dword_112ABB80, offset variable_to_be_set
.text:1090F722                  mov     dword_112ABB84, offset unk_11143364
.text:1090F72C                  mov     dword_112ABB8C, offset unk_111433E4
.text:1090F736                  push    offset vm_data
.text:1090F73B                  call    j_vm_enter_main_handler
.text:1090F740                  jmp     short loc_1090F750
.text:1090F742                  align 4
.text:1090F744                  dd 0
.text:1090F748                  db 3 dup(0)
.text:1090F74B byte_1090F74B    db 0
.text:1090F74C                  align 10h
.text:1090F750
.text:1090F750 loc_1090F750:
.text:1090F750                  cmp     variable_to_be_set, 0
.text:1090F757                  jnz     short loc_1090F763
.text:1090F759                  mov     dword_112ABBB0, 1
```

If we decode instructions (only a few) we may see what is going on here:

```
(2E) mov vm_reg0, vm_reg0
(24) mov vm_reg3C8, 0424448Bh
(9F) mov vm_reg3CC, 08244C8Bh
(DB) mov vm_reg3D0, 0004C969
(DA) mov vm_reg3D4, 30800000
(11) mov vm_reg3D8, FAE2409Fh
(11) mov vm_reg3DC, 505A5958h
(D8) mov vm_reg3E0, 9090C351h
(24) mov vm_regC8, 9F9F3276h
(F9) mov vm_regCC, FAEDF89Fh
(DA) mov vm_regD0, BFE5EBFA
(43) mov vm_regD4, CDBFF0EBh
(DB) mov vm_regD8, FEF0F3FA
(11) mov vm_regDC, B3FBFAFBh
(43) mov vm_regE0, EBFED7BFh
(F9) mov vm_regE4, B3FBFAEDh
(AA) mov vm_regE8, EBF6C9BFh
(43) mov vm_regEC, EBF6F3FEh
(11) mov vm_regF0, D6BFB3E6h
(F9) mov vm_regF4, EDFAF2F2h
(AA) mov vm_regF8, F1F0F6ECh
(6C) mov vm_regFC, EAF0E69Fh
(E2) mov vm_reg100, F3F3FEBFh
(24) mov vm_reg104, FAE8F0BFh
(DA) mov vm_reg108, BFFAF2BF
(E2) mov vm_reg10C, FAFDBFFEh
(F9) mov vm_reg110, BFB3EDFAh
(9F) mov vm_reg114, F2BFEDF0h
(6C) mov vm_reg118, FAFDE6FEh
(AA) mov vm_reg11C, FDFEFDBFh
(6C) mov vm_reg120, BFADBFE6h
(24) mov vm_reg124, 9FB6B2A5h
(AA) mov vm_reg128, FAFAEDF8h
(6C) mov vm_reg12C, EBBFE5EBh
(6C) mov vm_reg130, EABCBFF0h
(6C) mov vm_reg134, FCFEEFF1h
(43) mov vm_reg138, F8F1F6F4h
```

```
(F9) mov vm_reg13C, FCBCBFB3h
(D8) mov vm_reg140, F4FCFEEDh
(DB) mov vm_reg144, ABF8F1F6
(43) mov vm_reg148, FDE8FAF1h
(43) mov vm_reg14C, 9FECFAF6h
(43) mov vm_reg150, BFBFBFBFh
(AA) mov vm_reg154, BFBFBFBFh
(DA) mov vm_reg158, BFBFBFBF
(43) mov vm_reg15C, BFBFBFBFh
(D8) mov vm_reg160, BFBFBFBFh
(E2) mov vm_reg164, D3BFBFBFh
(E2) mov vm_reg168, BFFAE9F0h
(11) mov vm_reg16C, F2F0EDD9h
(DB) mov vm_reg170, F2BFBFB3
(6C) mov vm_reg174, B2A5BFFAh
(43) mov vm_reg178, 9E279FEFh
(9F) mov vm_reg17C, 769F9F9Fh
(D8) mov vm_reg180, 9F9F9F84h
(DB) mov vm_reg184, BFF7F8EA
(6C) mov vm_reg188, BFF7F8EAh
(F9) mov vm_reg18C, EDEBF3EAh
(DB) mov vm_reg190, FAF3BFFE
(E2) mov vm_reg194, EFBFEBFAh
(24) mov vm_reg198, F6FBFBFEh
(24) mov vm_reg19C, CC77F8F1h
(D8) mov vm_reg1A0, 9F9F9476h
(43) mov vm_reg1A4, F1F0FB9Fh
(DB) mov vm_reg1A8, FEEFBFEB
(6C) mov vm_reg1AC, 76FCF6F1h
(D8) mov vm_reg1B0, BF763D90h
(DA) mov vm_reg1B4, F79F9F9F
(9F) mov vm_reg1B8, F0E8BFF2h
(6C) mov vm_reg1BC, EDFAFBF1h
(E2) mov vm_reg1C0, FEF7E8BFh
(DB) mov vm_reg1C4, F9BFECEB
(DA) mov vm_reg1C8, EBBFEDF0
(F9) mov vm_reg1CC, EBBFFEFAh
(9F) mov vm_reg1D0, F8F6F1F0h
(11) mov vm_reg1D4, 7670EBF7h
(F9) mov vm_reg1D8, 9F9F9F91h
(DB) mov vm_reg1DC, BDD3A9B4
(D8) mov vm_reg1E0, 16C47093h
(AA) mov vm_reg1E4, 9F9DC71Ch
(D8) mov vm_reg1E8, A2765C9Fh
(E2) mov vm_reg1EC, AD9F9F9Fh
(24) mov vm_reg1F0, A9AFB2A7h
(24) mov vm_reg1F4, AFAFADB2h
(F9) mov vm_reg1F8, ABAEBFA8h
(43) mov vm_reg1FC, BFA6AEA5h
(AA) mov vm_reg200, F0FBBFB2h
(DB) mov vm_reg204, FCBFEBF1
(AA) mov vm_reg208, F4FCFEEDh
(F9) mov vm_reg20C, B3FAF2BFh
(DB) mov vm_reg210, E8BFF6BF
(43) mov vm_reg214, BFEBF1FEh
(AA) mov vm_reg218, ECBFF0EBh
(E2) mov vm_reg21C, EFFAFAF3h
(DB) mov vm_reg220, E8F0F1BF
(6C) mov vm_reg224, EDF7BFB3h
(DB) mov vm_reg228, 9FF9EFF2
(DA) mov vm_reg22C, 606040BA
(24) mov vm_reg230, 0F307460h
(F9) mov vm_reg238, 0000005Bh
(66) mov vm_reg23C, vm_reg14
(BE) add vm_reg23C, 000000C8h
(93) push vm_reg238
(93) push vm_reg23C
```

```
(48) mov vm_reg23C, vm_reg14
(F4) add vm_reg23C, 000003C8h
(2C) call vm_reg23C
(24) mov vm_reg320, 00000000h
(DA) mov vm_reg324, 00000001
(3F) mov vm_reg80, vm_reg18
(31) add vm_reg80, 00000004h
(56) mov vm_reg80, [vm_reg80]
(29) mov vm_regF0, vm_reg18
(C4) add vm_regF0, 00000008h
(19) mov vm_regF0, [vm_regF0]
(2E) mov vm_regF4, vm_reg18
(7A) add vm_regF4, 00000010h
(DE) mov vm_regF4, [vm_regF4]
(D6) mov vm_regF4, [vm_regF4]
(66) mov vm_reg7C, vm_reg28
(A6) mov vm_reg7C, [vm_reg7C]
(00) sub vm_reg7C, 0038271Eh
(FA) sub vm_reg7C, vm_reg258
(82) add vm_regF4, vm_reg7C
(9D) add vm_reg7C, 00000001h
(A1) mov [vm_reg80], vm_reg7C
(71) sub [vm_regF0], vm_regF4
vm_exit
vm_trace_done
```

Analysing this code statically we already can see what is going on:

```
(24) mov vm_reg3C8, 0424448Bh
(9F) mov vm_reg3CC, 08244C8Bh
(DB) mov vm_reg3D0, 0004C969
(DA) mov vm_reg3D4, 30800000
(11) mov vm_reg3D8, FAE2409Fh
(11) mov vm_reg3DC, 505A5958h
(D8) mov vm_reg3E0, 9090C351h
```

1st some data is moved to vm_registers from 3C8 to 3E0. We still don't know what this is. Could be a string, or code, or something else.

Then we see how more data is moved to vm_regs from C8 to 230. Next part which is interesting is this one:

```
(F9) mov vm_reg238, 0000005Bh
(66) mov vm_reg23C, vm_reg14
(BE) add vm_reg23C, 000000C8h
(93) push vm_reg238
(93) push vm_reg23C
(48) mov vm_reg23C, vm_reg14
(F4) add vm_reg23C, 000003C8h
(2C) call vm_reg23C
```

vm_reg14 is actually vm_context.self_context_ptr, so by looking at this code to vm_reg23C is stored address of vm_regC8, and then data is pushed onto stack, also value stored in vm_reg238 (5Bh) is also pushed onto stack. Then address of vm_reg3C8 is calculated, and SecuROM VM calls this code, remember it is that small portion of data moved to vm_rec3C8-3E0, so we are now sure that SecuROM at runtime generates code on it's vm_context which is executed. Right?

So when we dump that code, we may see what is SecuROM doing there:

```
seg000:018C03C8                 mov     eax, [esp+4]
seg000:018C03CC                 mov     ecx, [esp+8]
seg000:018C03D0                 imul    ecx, 4
seg000:018C03D6
seg000:018C03D6 __decrypt_loop:
seg000:018C03D6                 xor     byte ptr [eax], 9Fh
seg000:018C03D9                 inc     eax
seg000:018C03DA                 loop    __decrypt_loop
seg000:018C03DC                 pop     eax
seg000:018C03DD                 pop     ecx
seg000:018C03DE                 pop     edx
seg000:018C03DF                 push    eax
seg000:018C03E0                 push    ecx
seg000:018C03E1                 retn
```

Now you may see how code, which is passed in C8 to 320 descrypted, and then executed. When we take a look at this code we may see hiden cpuid check generated at runtime to stop easy vm dumping. In this code I will remove SecuROM's author comments (left there for some 0day groups, and that someone owns him a baby!? or a beer ☺)

```
seg000:018C00C8                 jmp     loc_18C017A
seg000:018C00C8
seg000:018C00CD aGreetzToReload db 'greetz to Reloaded, Hatred, Vitality, Immersion',0
seg000:018C00FD aYouAllOweMeABe db 'you all owe me a beer, or maybe baby 2 :-)',0
seg000:018C0128 aGreetzToUnpack db 'greetz to #unpacking, #cracking4newbies',0
seg000:018C0150 aLoveFromMeP    db '                              Love From,  me :-p',0
seg000:018C017A
seg000:018C017A loc_18C017A:
seg000:018C017A                 mov     eax, 1
seg000:018C017F                 jmp     loc_18C019F

seg000:018C019F loc_18C019F:
seg000:018C019F                 push    ebx
seg000:018C01A0                 jmp     loc_18C01B0
seg000:018C01A0
seg000:018C01A5 aDontPanics     db 'dont panicT'
seg000:018C01B0
seg000:018C01B0
seg000:018C01B0 loc_18C01B0:
seg000:018C01B0                 cpuid
seg000:018C01B2                 jmp     loc_18C01D7
seg000:018C01D7
seg000:018C01D7 loc_18C01D7:
seg000:018C01D7                 jmp     loc_18C01EA
seg000:018C01E2
seg000:018C01E2 loc_18C01E2:
seg000:018C01E2                 pop     ebx
seg000:018C01E3                 mov     [ebx+258h], eax
seg000:018C01E9                 retn
seg000:018C01EA
seg000:018C01EA
seg000:018C01EA loc_18C01EA:
seg000:018C01EA                 jmp     loc_18C022C
seg000:018C01EF a280620071419Do db '28-06-2007 14:19 - dont crack me, i want to sleep now,
hrmpf',0
seg000:018C022C
seg000:018C022C loc_18C022C:
seg000:018C022C                 and     eax, 0FFFFFFDFh
seg000:018C0231                 jmp     short loc_18C01E2
```

Well as you may see, I have deleted some comments from this code, but left some interesting ones ☺ Only prupose of this code is to mov cpuid value to vm_reg258h, and then execution of VM continues. How will you defeat this trick without vm restoring is up to you. I have a few ways, but as this is only tut about decoding VM, and how it is done, it will not cover this little problem (if it is a real problem of course). You will find bunch of this in VM, so think carefully how you would fix it.

Ok, let's see what's going on later on in this code:

```
(24) mov vm_reg320, 00000000h
(DA) mov vm_reg324, 00000001
```

Above 2 VM opcodes are junk, nothing special imho, so we analyse next ones:

```
(3F) mov vm_reg80, vm_reg18
(31) add vm_reg80, 00000004h
(56) mov vm_reg80, [vm_reg80]
```

Vm_reg18 is, if you take a look at VM_Context, pointer to argument passed to VM interpreter, but let's get back to the code which called this vm_interpreter:

```
.text:1090F70E                  mov     vm_data, offset vm_opcodes
.text:1090F718                  mov     argument_4, offset variable_to_be_set
.text:1090F722                  mov     argument_8, offset offset_32
.text:1090F72C                  mov     argument_10, offset offset_64
.text:1090F736                  push    offset vm_data
.text:1090F73B                  call    j_vm_enter_main_handler
.text:1090F740                  jmp     short loc_1090F750
...
.text:1090F750 loc_1090F750:
.text:1090F750                  cmp     variable_to_be_set, 0
.text:1090F757                  jnz     short loc_1090F763
```

So it takes address of variable_to_be_set into vm_reg80, not much to say, let's go further, to next code, again we see same logic used here, but this time it takes address passed in argument+8 which is offset to dword with value 32h

```
(29) mov vm_regF0, vm_reg18
(C4) add vm_regF0, 00000008h
(19) mov vm_regF0, [vm_regF0]
```

Then it takes, using same logic, value stored in [argument+10h], this is pointer to dword with value of 64h, so this code loads 64h into vm_regF4.

```
(2E) mov vm_regF4, vm_reg18
(7A) add vm_regF4, 00000010h
(DE) mov vm_regF4, [vm_regF4]
(D6) mov vm_regF4, [vm_regF4]
```

This few lines are part of vm anti-dump, it load address of obsfucated_apis (vm_reg28) into vm_reg7C, and tnx to our knowledge about obsfucated APIs field we know that 1st DWORD in this array is obsfucated cpuid value, we also saw in code executed in vm_context that it saves cpuid value to vm_reg258, so now it will simply subs value obtained at runtime, with the one saved in vm_context:

```
(66) mov vm_reg7C, vm_reg28
(A6) mov vm_reg7C, [vm_reg7C]
(00) sub vm_reg7C, 0038271Eh
(FA) sub vm_reg7C, vm_reg258
```

If we had wrong cpuid value here, then vm_reg7C would have god knows what, instead of 64h. Next code simply loads 1 to vm_reg7C, and moves it to [vm_reg80] which is address of variable_to_be_set, and substracts from [argument+8] (which is for the 1st time 32h), 64h which is on other hand passed in [argument+10h].

```
(82) add vm_regF4, vm_reg7C
(9D) add vm_reg7C, 00000001h
(A1) mov [vm_reg80], vm_reg7C
(71) sub [vm_regF0], vm_regF4
```

If we are about to reconstruct this code to something user friendly we could write this:

```
mov   variable_to_be_set, 1
push  eax
mov   eax, [offset_64h]
sub   [offset_32h], eax
pop   eax
```

Last but not least is the way of decoding API redirection, actually, imho, this can't be called api redirection *per se*, as it is actually call dword ptr[] emulation, so really it should be considered as part of VM code morphing instead as a API redirection (I also posted this example at my blog) but it is good enough to be here too:

```
(85) mov vm_regF0, vm_reg1C
(7A) add vm_regF0, 00000024h
(D0) mov vm_regF0, [vm_regF0]
(B4) mov vm_regF4, vm_reg1C
(C4) add vm_regF4, 00000028h
(F0) mov [vm_regF4], vm_regF0
(34) single-step check
(29) mov vm_regF4, vm_reg1C
(A8) add vm_regF4, 00000024h
(F9) mov vm_reg90, 10FC5180h
(19) mov vm_reg90, [vm_reg90]
(FB) mov [vm_regF4], vm_reg90
(8B) vm_exit (retn)
vm_trace_done
```

This is just emulation of certain opcode, not something that we should handle separately:

1st it takes address of EIP which in this case is retn address stored on stack, as we arrived here trough call.
2nd It moves that address onto place of argument, as argument will be used this time as retn address from API call, that's also why this time we exit from vm with retn instead of retn 4.
3rd it takes EIP to vm_regF4, and moves to vm_reg90 offset 10FC5180h
4th moves dword from 10FC5180h to vm_reg90
5th sets EIP with dword from 10FC5180h
6th it exits (goes to updated EIP)

Basically this is emulation of call dword ptr[10FC5180h], but if we look in IDA what is stored at 10FC5180h we may see that this is API redirection:

```
.idata:10FC5180          ; DWORD GetCurrentThreadId(void)
.idata:10FC5180          extrn GetCurrentThreadId:dword
```

So this is call dword ptr[GetCurrentThreadId]. As I said, I wouldn't consider this as API redirection, only as call emulation. Nothing more, nor less.

And so on, so on. I could spend agaes just showing what is what, and how decompiling process of VM looks like, but that would take ages to show all possible combinations of what this VM is capable of doing.

6. Conclusion

Well, VM as VM, boring to reverse as it doesn't teach you anything new, only how to spend your time on some stupid things that no one actually cares about. If this document was usefull to you, just reply at our forum (http://forums.accessroot.com). Well that's all folks ☺

7. Greetings

First of all, I want to thank to my mates in ARTeam for sharing their knowledge, 29a vx group for the one of the best ezines ever, great unpackers from unpack.cn (fly, shoooo, okododo, heXer, softworm), friendly ppl at http://woodman.cjb.net , and of course you, for reading this document.

С вером у Бога, deroko of ARTeam

Ђенерал Дража

Ми смо војска Ђенерала Драже, који правду и слободу траже
За слободу српства и Србије, погинути нама жао није
Ој Србијо, узданице стара, ти си много водила мегдана
Душмани ће памтити док живе, Ђенерала Дражу из СРБИЈЕ
Вјечна слава Ђенералу Дражи, што почива на вјечитој стражи
Вјековима што српством напаја, ТАКО ЧЕТНИК УМИРЕ ЗА КРАЉА
Равна Горо отвори нам врата и херојског прими команданта

Три хиљаде црне браде

Три хиљаде црне браде, а пред њима Корда Раде
Сива тица, љута злица, крвопија потурица
Шарац носи, њиме коси, свети Раде на хиљаде
Памте турци и катили, кад су с' Радом заратили

SecuROM : Recursive VM by 2kAD

When I joined up with deroko on Securom he was already way ahead of me. At that time he had already looked through 100 or so VM handlers. So we decided that I should take a closer look at the recursive procedures that were called from within the target. I tend to call this a "recursive VM" well knowing that I/we will probably be spanked to death by certain individuals, but I don't know how other to describe it really! Before going any further let's clear up what recursive is…

Wikipedia defines recursive as: *"In mathematics and computer science, recursion specifies (or constructs) a class of objects or methods (or an object from a certain class) by defining a few very simple base cases or methods (often just one), and defining rules to break down complex cases into simpler cases."* Now is to me who is not a native of the English language just a telling it to me in plain Chinese, so here it is (also from Wikipedia) in a more understandable way:

1. Are we done yet? If so, return the results. Without such a *termination condition* a recursion would go on forever.
2. If not, *simplify* the problem, solve the simpler problem(s), and assemble the results into a solution for the original problem. Then return that solution.

Now we know what recursive is… Let's check Wikipedia for Virtual Machine : *"In computer science, a **virtual machine** (VM) is a software implementation of a machine (computer) that executes programs like a real machine."* These calls are not really programs being executed, but since I don't have any other name to call this procedure I name it "VM". These calls have arguments being pushed onto stack and then deobfuscated by the recursive procedure, which returns a value in EAX and sometimes also a value on stack.

What the recursive procedure does is basically a binary three. It checks two bits, if first bit and/or second bit are not set it calls itself, checks two bits, calls itself, checks two bits, call itself etc. etc. Eventually it encounters two bits that are set and from there it returns doing a simple operation such as XOR.

Remember how I said that the recursive sometimes changes a value on stack along with EAX? Well, in order to show this I will split the search process in finding recursive calls up into two catagories. Simple and advanced. Actually it's not hard finding either… Let's start with the easier recursive calls:

```
11E18667    50              PUSH EAX
11E18668    9C              PUSHFD
11E18669    B8 40DB4311     MOV EAX,bioshock.1143DB40
11E1866E    54              PUSH ESP
11E1866F    C1EA 00         SHR EDX,0
11E18672    68 DD47E011     PUSH bioshock.11E047DD
11E18677    FFD0            CALL EAX
11E18679    9D              POPFD

-------------

11E19BD8    50              PUSH EAX
11E19BD9    9C              PUSHFD
11E19BDA    B8 39C89FAF     MOV EAX,AF9FC839
11E19BDF    54              PUSH ESP
11E19BE0    68 B148E011     PUSH bioshock.11E048B1
11E19BE5    C1E3 00         SHL EBX,0
11E19BE8    35 B92ADCBE     XOR EAX,BEDC2AB9
11E19BED    FFD0            CALL EAX
11E19BEF    9D              POPFD

-------------

11E1A137    50              PUSH EAX
11E1A138    9C              PUSHFD
11E1A139    B8 536C8CC5     MOV EAX,C58C6C53
```

```
11E1A13E   54              PUSH ESP
11E1A13F   68 DC48E011     PUSH bioshock.11E048DC
11E1A144   35 6398CFD      XOR EAX,D4CF9863
11E1A149   FFD0            CALL EAX
11E1A14B   87DB            XCHG EBX,EBX
11E1A14D   9D              POPFD
```

These 3 code snippets are just a small fraction of simple calls. Notice that they are easily found by the PUSH EAX – PUSHFD followed by a CALL EAX further down. There is another “simple” recursive call that looks like this:

```
11E3B18B   50              PUSH EAX
11E3B18C   90               NOP
11E3B18D   50               PUSH EAX
11E3B18E   C1FE 00         SAR ESI,0
11E3B191   B8 C4949CFD     MOV EAX,FD9C94C4
11E3B196   90              NOP
11E3B197   54              PUSH ESP
11E3B198   35 F474DFEC     XOR EAX,ECDF74F4
11E3B19D   68 EB85E011     PUSH bioshock.11E085EB
11E3B1A2   8BF6            MOV ESI,ESI
11E3B1A4   FFD0            CALL EAX
11E3B1A6   094424 04       OR DWORD PTR SS:[ESP+4],EAX
11E3B1AA   58              POP EAX
11E3B1AB   58              POP EAX

-------------

11E3B368   50              PUSH EAX
11E3B369   C1E1 00         SHL ECX,0
11E3B36C   50              PUSH EAX
11E3B36D   B8 B2C2DB20     MOV EAX,20DBC2B2
11E3B372   54              PUSH ESP
11E3B373   87ED            XCHG EBP,EBP
11E3B375   35 82229831     XOR EAX,31982282
11E3B37A   68 C386E011     PUSH bioshock.11E086C3
11E3B37F   FFD0            CALL EAX
11E3B381   394424 04       CMP DWORD PTR SS:[ESP+4],EAX
11E3B385   58              POP EAX
11E3B386   C1E8 00         SHR EAX,0
11E3B389   58              POP EAX
```

There is a couple of more tricks to call the recursive procedure, but they all CALL EAX with a PUSHED EAX. The trick here is of course to write a tool the can identify this and luckily for you deroko already coded such tool. It even finds the advanced recursive calls. Look at how they are found:

```
11E2191A   50              PUSH EAX
11E2191B   9C              PUSHFD
11E2191C   87ED            XCHG EBP,EBP
11E2191E   83EC 04         SUB ESP,4
11E21921   C70424 D0B01177 MOV DWORD PTR SS:[ESP],7711B0D0
11E21928   83EC 04         SUB ESP,4
11E2192B   87C9            XCHG ECX,ECX
11E2192D   C70424 CFA76B13 MOV DWORD PTR SS:[ESP],136BA7CF   <- THIS WILL CHANGE
11E21934   54              PUSH ESP
11E21935   68 DF4AE011     PUSH bioshock.11E04ADF
11E2193A   68 0344FC5C     PUSH 5CFC4403
11E2193F   813424 33A4BF4D XOR DWORD PTR SS:[ESP],4DBFA433
11E21946   58              POP EAX
11E21947   90              NOP
11E21948   FFD0            CALL EAX
11E2194A   C1FD 00         SAR EBP,0
11E2194D   56              PUSH ESI
11E2194E   8D00            LEA EAX,DWORD PTR DS:[EAX]
11E21950   8B7424 04       MOV ESI,DWORD PTR SS:[ESP+4]
```

```
11E21954   03C6          ADD EAX,ESI
11E21956   8BF0          MOV ESI,EAX
11E21958   57            PUSH EDI
11E21959   8B7C24 0C     MOV EDI,DWORD PTR SS:[ESP+C]
11E2195D   33F7          XOR ESI,EDI
11E2195F   8BEE          MOV EBP,ESI
11E21961   5F            POP EDI
11E21962   90            NOP
11E21963   5E            POP ESI
11E21964   8D6D 00       LEA EBP,DWORD PTR SS:[EBP]
11E21967   58            POP EAX
11E21968   58            POP EAX
11E21969   9D            POPFD
```

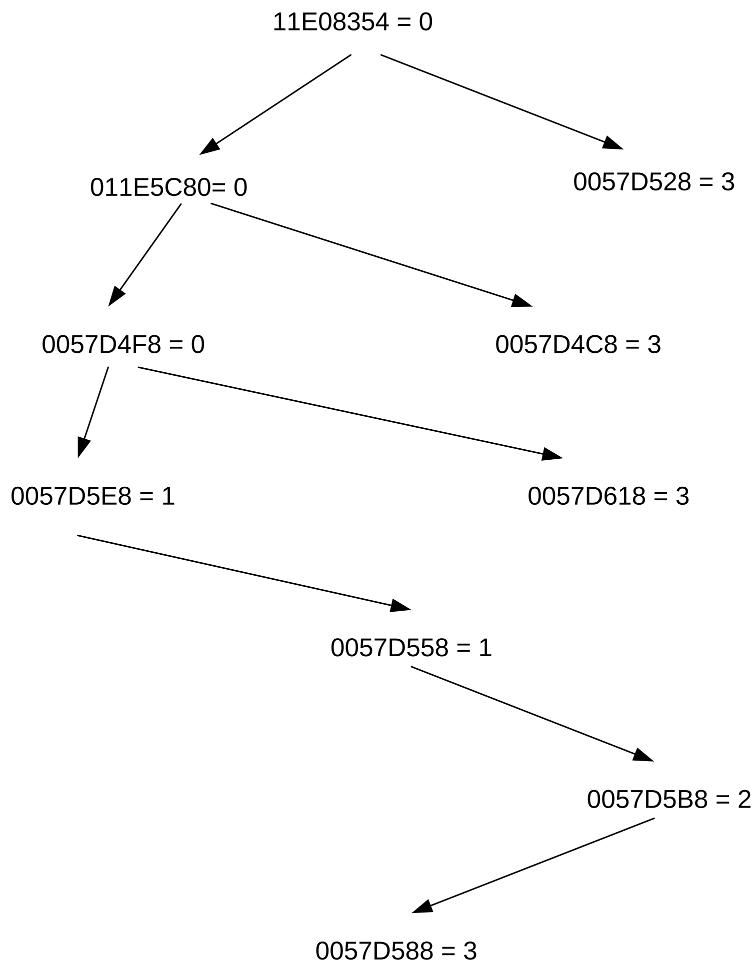
Again, CALL EAX is the call to the recursive procedure. This "advanced" one simple simply pushes one more argument onto stack and deobfuscate the value. This value is later used in another deobfuscation algorithm to retrieve the value for EAX. If we follow the CALL EAX we will end up in some small but code parts that simply redirects you into the *Recursive VM.* No reason to really show it, as it is quite obvious once you are there.

Now that we know how to search for the recursive calls, let's look at the recursive procedure itself. I don't want to spend much time on the procedure itself as it consists of nothing much that a bunch of byte comparing. These bytes can be considered opcodes, as they tend to tell the recursive procedure what command to perform. Anyway, every time the recursive procedure is called an argument containing a pointer to a *Recursive VM structure* is pushed onto stack.

This is how the structure looks like:

```
00 00 56 00  →  unknown
29 00 00 00  →  opcode
00 00 00 00  →  unknown
00 00 00 00  →  unknown
00 00 00 00  →  Test bits 1 and 2
07 00 00 00  →  unknown
28 D5 57 00  →  1st  jump if bit 1 not set
00 00 00 00  →  unknown
80 5C 1E 01  →  2nd  jump if bit 2 not set
00 00 00 00  →  unknown
00 00 00 00  →  unknown
```

Now, I explained earlier the *Rescursive VM* checks if the TEST BITS are set. If 1st bit is not set we branch until we return here, and then test 2nd bit and branches if not set. Once the 2nd branch returns we perform some kind of simple operation.



In the above I have tried to show how the recursive procedure runs through a chain. The offsets represent the pointers to recursive structures. When the recursive vm encounters a test bit not set it branches. When both are set it has reached a "dead end" and returns to the parent caller.

Recursive Structure	Opcode	Test bits	Branch/Value 1	Branch/Value 2
11E08354	29	00	57D528	11E5C80
0057D528	A2	11	42E4	57D0D0
011E5C80	CB	00	57D4C8	54D4F8
0057D4C8	23	11	584667	1785
0057D4F8	29	00	57D618	57D5E8
0057D618	74	11	040A0066	040A71E2
0057D5E8	08	01	01BE4C83	57D558
0057D558	FD	01	0	57D5B8
0057D5B8	4B	10	57D588	0C
0057D588	17	11	6B8B	0

Stop
Branch
Branch

The above table shows the branching of the recursive vm.

Each new call then performs test on test bit 1, then branches if necessary, then performs check on test bit 2, then branches if necessary, and last perform a simple command and return to parent call.

Well, I have done the hard part for you and traced through the chain. Once the recursive vm rolls back we can see each command. The first command showed below is of course the last in the chain, and the last command in the below is the first.

```
MOV EAX, [EDI+EAX*4]        <- get argument
SHL EAX, ECX                <- deobfuscate
MOV [EBX+EAX*4], EDI        <- put argument back

XOR EAX, ESI                <- deobfuscate EAX (entry value = 01BE4C83)
XOR EAX, EDI                <- deobfuscate EAX
XOR EAX, ESI                <- deobfuscate EAX
XOR EAX, ESI                <- deobfuscate EAX
XOR EAX, EDI                <- deobfuscate EAX (return value = 0308426E)
```

The registered used for deobfuscating EAX gets it's values from the branch/value boxes in the above table.

As one can see the recursive vm is not actually a virtual machine but rather a binary three. It's fairly simple to reverse once to set your mind up to follow the chain. But how would one go about fixing this in the executable? The easiest way is probably to just search for the simple and advanced calls to recursive vm. Then code a tracer that will run over the call to recursive vm. If 2 arguments are pushed we need to grab stack value too after the run over. Once we have obtained the value/values we simply write this/these values back into the code, overwriting the actual CALL EAX.

As a final remark I would like to add that this was written in a rush. Real life for me at the moment is quite busy. Hope it gave you some insight into the recursive vm or binary three if you will. Feel free to ask as much as you want on the ARTeam forums.

2kAD / ARTeam – Securom : *superbia mos cado*

8. References

1. "Special Issue For SecuRom 7.30.0014 Complete Owning", AnonymousS, Human, deroko, http://tutorials.accessroot.com