

INSIDE CODE VIRTUALIZER

by scherzo - scherzocrk@gmail.com

February 16, 2007



Contents

1	Introduction	3
1.1	About Code Virtualizer	3
1.2	About this article	3
2	The Virtual Machine - <i>Light VM</i>	4
2.1	The Virtual Machine itself	4
2.2	Generating the Virtual Machine	6
3	The Virtual Opcodes	9
3.1	Disassembling and "Assembling" again	9
3.2	Generating and Writing the Virtual Opcodes	14
3.3	Completing the analysis: why does this really work?	18
4	How to Make an Unpacked version of Code Virtualizer Full	22
5	Hopes for the Future and Acknowledgments	24
5.1	Why write this article?	24
5.2	The general attack approach	25
5.3	Acknowledgments	27

DISCLAIMER

ALERT: THIS ARTICLE MUST BE USED ONLY FOR SCIENTIFIC/STUDY PURPOSES. THE AUTHOR OF THIS ARTICLE IS NOT RESPONSIBLE FOR ANY USE OF THE KNOWLEGDE DESCRIBED HERE FOR ILLEGAL PURPOSES. YOU DO ARE ONLY ALLOWED TO READ THIS ARTICLE IF YOU AGREE WITH THIS DISCLAIMER.

1 Introduction

1.1 About Code Virtualizer

Code Virtualizer is a powerful code obfuscation system that helps developers protect their sensitive code areas against Reverse Engineering. Code Virtualizer has been designed to enact high security for your sensitive code while requiring minimal system resources.

Code Virtualizer will convert your original code into Virtual Opcodes that will be only understood by an internal Virtual Machine. Those Virtual Opcodes and the Virtual Machine itself are different for every protected application, avoiding a general attack over Code Virtualizer.

Code Virtualizer can protect your code in any x32 and x64 native PE files, like executable files (EXEs), system services, DLLs, OCXs, ActiveX controls, screen savers and device drivers[1].

1.2 About this article

First of all, I need to say sorry. Probably you will see a lot of mistakes because of my english but I hope you will understand me.

This article aim to explain how Code Virtualizer works. During the last month, I spent all my free time analysing the Code Virtualizer Demo 1.0.1.0 unpacked by softworm[2]. Fortunately, I finished my analysis and I can say that this is the best software I have seen before. Not best in the meaning of protection, but in the meaning of organization. This was the most pleasing software I have analysed.

Three important things to notice are that the description and explanation of the code disassembled by OllyDbg[3] is done in the code execution order. Most things that I am going to say are applicable only for the 1.0.1.0 version of Code Virtualizer. For comments on new versions, see "Hopes for the Future and Acknowledgments". And I will not threat the 64-bit case.

This article is divided in three parts. Firstly I am going to talk about how the

Virtual Machine is generated and why Oreans[4] says that each Virtual Machine has its own characteristics. Secondly I use the concepts described before to explain how the Virtual Opcodes are generated, how they are executed and why they emulate the original code of an application. The last part is a bonus: you are going to learn how to make an unpacked version of Code Virtualizer full.

Enjoy this article and I hope you learn something reading it.

2 The Virtual Machine - *Light VM*

2.1 The Virtual Machine itself

I think you have noticed that I called this Virtual Machine as "Light VM". Actually, not me but Oreans developers did that probably referring to the Themida Virtual Machine.

Basically each Virtual Machine has 150 handlers and a main handler. By handler, I mean a kind of function that will deal with the Virtual Opcodes. In general, they are small (one to six lines of assembly code) and it is really important to understand each one.

Next I will show the first structure that I called `Handler_Information` and an example of it (figure 1):

- WORD `id` // *a number that represents the handler*
- DWORD `start` // *the address of the start of this handler in the Code Virtualizer file*
- DWORD `end` // *the address of the end of this handler in the Code Virtualizer file*
- DWORD `address` // *the address of the start of the handler in the protected file*

- WORD order // random number from (0Eh to A4h) that will indicate the place of the handler in the protected file

Figure 1: Handler_Information structure example

Address	Hex dump	ASCII
00603B72	00 00 F0 35 60 00 F8 35 60 00 DF 8B 41 00 88 00	...5'.°5'.■iA.è.

This structure is the principal one to generate the VM. I will not show you each of the 150 handlers. This is tedious but if you want to study Code Virtualizer deeper, you must read and understand one by one. I will show you just the handler I showed in the figure above (figure 1; id = 0000h; start = 006035F0h; end = 006035F8h) and the main handler (figure 3):

Figure 2: Handler 0000h

Address	Hex dump	Disassembly
006035F0	. AC	lods byte ptr ds:[esi]
006035F1	. 0FB6C0	movzx eax,al
006035F4	. 8D0487	lea eax,dword ptr ds:[edi+eax*4]
006035F7	. 50	push eax

Figure 3: Main Handler

Address	Hex dump	Disassembly
006044D4	90	nop
006044D5	90	nop
006044D6	60	pushad
006044D7	9C	pushfd
006044D8	FC	cld
006044D9	E8 00000000	call Code_Vir.006044DE
006044DE	5F	pop edi
006044DF	81EF 11111111	sub edi,11111111
006044E5	8BC7	mov eax,edi
006044E7	81C7 11111111	add edi,11111111
006044ED	3B47 2C	cmp eax,dword ptr ds:[edi+2C]
006044F0	75 02	jnz short Code_Vir.006044F4
006044F2	EB 13	jmp short Code_Vir.00604507
006044F4	8947 2C	mov dword ptr ds:[edi+2C],eax
006044F7	B9 11111111	mov ecx,11111111
006044FC	EB 05	jmp short Code_Vir.00604503
006044FE	01448F 34	add dword ptr ds:[edi+ecx*4+34],eax
00604502	49	dec ecx
00604503	0BC9	or ecx,ecx
00604505	75 F7	jnz short Code_Vir.006044FE
00604507	8B7424 24	mov esi,dword ptr ss:[esp+24]
0060450B	8BDE	mov ebx,esi
0060450D	03F0	add esi,eax
0060450F	8B47 30	mov eax,dword ptr ds:[edi+30]
00604512	B9 01000000	mov ecx,1
00604517	F0:0FB14F 30	lock cmpxchg dword ptr ds:[edi+30],ecx
0060451C	74 F9	je short Code_Vir.00604517
0060451E	90	nop
0060451F	90	nop
00604520	AC	lods byte ptr ds:[esi]
00604521	0FB6C0	movzx eax,al
00604524	FF2487	jmp dword ptr ds:[edi+eax*4]
00604527	61	popad
00604528	C3	retn

There is a particularity in the main handler: you can see three times the DWORD 11111111h. They are different depending on the protected application. The first DWORD is the address of the seventh line of the main handler in the protected file. The second one is the "image base" of the Virtual Machine. The last DWORD is the total number of handlers in that VM.

2.2 Generating the Virtual Machine

Here I will give arguments to prove after that the phrase "Those Virtual Opcodes and the **Virtual Machine** itself are different for every protected application, avoiding a general attack over Code Virtualizer." is not a very important feature.

The first step done by Code Virtualizer is to write the main handler. Next the other 150 handlers will be written following the Handler_Information.order sequence from 1Eh to A4h. As Handler_Information.order is randomly generated the result will be **a difference sequence of handlers for every protected application** (if you want an example, see [5]).

Now I am going to explain how the handler 0000h (see figure 2) is written. The same process occurs for every handler.

The next step is showed by the code below:

Figure 4: LODS special case

```

006081DA > 803E AC      cmp byte ptr ds:[esi],0AC      lods byte ptr ds:[esi]
006081DD . 74 2A      je short Code_Vir.00608209      lods word ptr ds:[esi]
006081DF . 66:813E 66AD  cmp word ptr ds:[esi],0AD66    lods dword ptr ds:[esi]
006081E4 . 74 23      je short Code_Vir.00608209      lods dword ptr ds:[esi]
006081E6 . 803E AD      cmp byte ptr ds:[esi],0AD      check for the handler 0154h
006081E9 . 0F85 A3020000  jne Code_Vir.00608492          check for the handler 0156h
006081EF . 817D 08 54010000  cmp [arg,1],154
006081F6 . 0F84 96020000    je Code_Vir.00608492
006081FC . 817D 08 56010000  cmp [arg,1],156
00608203 . 0F84 89020000    je Code_Vir.00608492

```

This piece of code looks for LODS instructions. This is not applicable for the handlers 0154h and 0156h. But why these checks? Well, the LODS instruction in a handler represents the reading of 1, 2 or 4 Virtual Opcodes. And to increase the security, Oreans developers insert random code after the LODS instruction. To do that, they use another structure that I have called Special_Handler. Here

you are:

- WORD Handler_Information.id // see *Handler_Information* structure
- BYTE instruction3 // number that says what kind of instruction will be written as the third instruction
- BYTE instruction2 // number that says what kind of instruction will be written as the second instruction
- BYTE instruction1 // number that says what kind of instruction will be written as the first instruction
- BYTE instruction4 // number that says what kind of instruction will be written as the fourth instruction
- DWORD Random1 // random number that will be part of the instruction
2
- DWORD Random2 // random number that will be part of the instruction
3

Table 1: Table of possible random instructions. Each of these instructions can be written in DWORD, WORD or BYTE format using the respective registers ax, bx, al, bl.

	instruction1	instruction2	instruction3	instruction4
0	<i>sub eax,ebx</i>	<i>sub eax,Random1</i>	<i>sub eax,Random2</i>	<i>sub ebx,eax</i>
1	<i>add eax,ebx</i>	<i>add eax,Random1</i>	<i>add eax,Random2</i>	<i>add ebx,eax</i>
2	<i>xor eax,ebx</i>	<i>xor eax,Random1</i>	<i>xor eax,Random2</i>	<i>xor ebx,eax</i>

Figure 5: Example of Special_Handler structure

Address	Hex dump
02280000	00 00 01 01 01 02 CF A6 DE 62 82 C4 07 26

So before those operations, the handler 0000h (figure 2) will be like this:

Figure 6: Handler 0000h before addition of 4 instructions

Address	Hex dump	Disassembly
03040000	AC	lods byte ptr ds:[esi]
03040001	2AC3	sub al,bl
03040003	2C 2B	sub al,2B
03040005	2C 38	sub al,38
03040007	02D8	add bl,al
03040009	0FB6C0	movzx eax,al
0304000C	8D0487	lea eax,dword ptr ds:[edi+eax*4]
0304000F	50	push eax

The next step is another security feature. Some kind of instructions are mutated by the Oreansf1.F4 function exported by Oreansf1.dll module. This means that the code of each handler will be obfuscated and more, this mutation engine is strictly related to the option Virtual Machine Obfuscation. Actually, this option **only** changes the complexity of the mutated opcode. This is really something strange because there is no difference if the complexity of the VM is low or highest in a general attack to Code Virtulizer (for more comments, see "Hopes for the Future and Acknowledgments").

Figure 7: Handler 0000h with mutated opcodes

Address	Hex dump	Disassembly
02FD079F	AC	lods byte ptr ds:[esi]
02FD07A0	2808	sub al,bl
02FD07A2	66:68 D109	push 901
02FD07A6	66:891C24	mov word ptr ss:[esp],bx
02FD07AA	68 FD060000	push 6FD
02FD07AF	890424	mov dword ptr ss:[esp],eax
02FD07B2	B0 90	mov al,90
02FD07B4	F608	neg al
02FD07B6	FEC8	dec al
02FD07B8	FEC8	dec al
02FD07BA	04 10	add al,10
02FD07BC	34 A0	xor al,0A0
02FD07BE	88C7	mov bh,al
02FD07C0	58	pop eax
02FD07C1	28F8	sub al,bh
02FD07C3	66:8B1C24	mov bx,word ptr ss:[esp]
02FD07C7	81C4 02000000	add esp,2
02FD07CD	52	push edx
02FD07CE	B2 38	mov dl,38
02FD07D0	2800	sub al,dl
02FD07D2	8B1424	mov edx,dword ptr ss:[esp]
02FD07D5	55	push ebp
02FD07D6	89E5	mov ebp,esp
02FD07D8	50	push eax
02FD07D9	B8 04000000	mov eax,4
02FD07DE	01C5	add ebp,eax
02FD07E0	58	pop eax
02FD07E1	53	push ebx
02FD07E2	B8 04000000	mov ebx,4
02FD07E7	0100	add ebp,ebx
02FD07E9	5B	pop ebx
02FD07EA	872C24	xchg dword ptr ss:[esp],ebp
02FD07ED	8B2424	mov esp,dword ptr ss:[esp]
02FD07F0	00C3	add bl,al
02FD07F2	0FB6C0	movzx eax,al
02FD07F5	8D0487	lea eax,dword ptr ds:[edi+eax*4]
02FD07F8	68 637F0000	push 7F63
02FD07FD	890424	mov dword ptr ss:[esp],eax

Before that, a JUMP to the main handler is written so the next handler will be called.

The next security feature is quite fun to see: all the 150 handlers are mixed randomly!!! For example, a piece of the handler 0161h is followed by a piece of the handler 0001h and the handler 0069h, etc...

So in the end there will be a complete obfuscated, unique and difficult code to be analysed. Really! I do not think so :).

3 The Virtual Opcodes

3.1 Disassembling and "Assembling" again

I know that the things are obscure. You probably still have no idea about how those handlers work but I promise that it will be clear in the section 3.3.

The figure below shows a macro not virtualized. The code that will be virtualized starts at 0040106Eh and ends at 0040107Dh.

Figure 8: Macro not virtualized

```
0040105C | 43 56 20 20 | short vc_examp.0040106E
0040105E | 00          | ascii "CU $",0
00401064 | 00          | ascii 0
00401065 | 00          | ascii 0
00401066 | 00          | ascii 0
00401067 | 00          | ascii 0
00401068 | 00          | ascii 0
00401069 | 00          | ascii 0
0040106A | 43 56 20 20 | ascii "CU "
0040106E | 33C9       | xor ecx,ecx
00401070 | 41         | inc ecx
00401071 | 8BD1       | mov edx,ecx
00401073 | 0FAFC2    | imul eax,edx
00401076 | 83F9 0A    | cmp ecx,0A
00401079 | 7C F5     | j! short vc_examp.00401070
0040107B | EB 10     | short vc_examp.0040108D
0040107D | 43 56 20 20 | ascii "CU $",0
00401083 | 00        | db 00
00401084 | 00        | db 00
00401085 | 00        | db 00
00401086 | 00        | db 00
00401087 | 00        | db 00
00401088 | 00        | db 00
00401089 | 43        | db 43
0040108A | 56        | db 56
0040108B | 20        | db 20
0040108C | 20        | db 20
0040108D | 6A 40     | push 40
```

Next a PUSH 0040108Dh and RET will be added to the original code so the program can continue its execution normally.

After that, the exported function Oreansf1.F1 disassembles the original code as you can see below. It was really a surprise to me when I saw that; I hoped that Code Virtualizer would threat the code through the bytes of the original code not through strings. It uses Delphi functions to threat strings and I think this is not the faster way but for sure it is easier.

Figure 9: Code disassembled

```

section .code base 0000000h code

    XOR    ECX, ECX

@label1:
    INC    ECX

    MOV    EDX, ECX

    IMUL   EAX, EDX

    CMP    ECX, 10

    JL     @label1
    PUSH   00040108dh

    RET

```

Now the function OreansX2dllR.F1 exported by OreansX2dllR.dll will do the principal and most complex work of assemble the assembly code in a Code Virtualizer syntax and generate the most important structure that I have called OreansX2.

OreansX2 structure:

- DWORD instruction // *type of instruction following the Code Virtualizer syntax*
- DWORD suffix // *suffix for the instruction*
- DWORD data1 // *data for the instruction*
- DWORD data2 // *data for the instruction*

- WORD unknown // *unknown use*

Table 2: Table of possible instructions for OreansX2 structure

OreansX2.instruction	instruction
00	LOAD
01	STORE
02	MOVE
03	IFJMP
04	EXTRN
05	UNDEF
06	IMULC
07	ADC
08	ADD
09	AND
0A	CMP
0B	OR
0C	SUB
0D	TEST
0E	XOR
0F	MOVZX
10	MOVZX_W
11	LEA
12	INC
13	RCL
14	RCR
15	ROL
16	ROR
17	SAL
18	SAR
19	SHL
1A	SHR
1B	DEC
1C	NOP
1D	MOVSX
1E	MOVSX_W
1F	CLC
20	CLD
21	CLI
22	CMC
23	STC
24	STD
25	STI
26	HLT

Table 3: Table of possible instructions for OreansX2 structure (cont.)

OreansX2.instruction	instruction
27	BT
28	BTC
29	BTR
2A	BTS
2B	SBB
2C	MUL
2D	IMUL
2E	DIV
2F	IDIV
30	BSWAP
31	NEG
32	NOT
33	RET

Table 4: Table of possible suffixes

OreansX2.suffix	suffix
00	
01	ADDR
02	%sADDR, %d
03	%sADDR, %.8x%h
04	BYTE PTR %s[ADDR]
05	WORD PTR %s[ADDR]
06	DWORD PTR %s[ADDR]
07	QWORD PTR %s[ADDR]
08	%sBYTE PTR [%.8x%h]
09	%sWORD PTR [%.8x%h]
0A	%sDWORD PTR [%.8x%h]
0B	%sQWORD PTR [%.8x%h]
0C	ADDR, BYTE PTR %s[%.8x%h]
0D	ADDR, WORD PTR %s[%.8x%h]
0E	ADDR, DWORD PTR %s[%.8x%h]
0F	ADDR, QWORD PTR %s[%.8x%h]
10	%s%d
11	%s%.8x%h
12	reserved
13	reserved
14	reserved
15	reserved

Table 5: Table of possible suffixes (cont.)

OreansX2.suffix	suffix
16	reserved
17	reserved
18	BYTE
19	WORD
1A	DWORD
1B	QWORD
1C	reserved
1D	reserved
1E	FLAGS
1F	%s[ADDR]
20	%sBYTE %d
21	%sWORD %d
22	%sDWORD %d
23	%sQWORD %d

As you can see, the syntax is quite logic. It uses XOR, ADD, etc. for well known instructions and obvious names like MOVE, STORE, LOAD for "special" instructions; the suffixes use a single variable ADDR and well known formats like DWORD PTR [ADDR].

I still do not understand completely how those instructions are generated from the original code disassembled but I think that this is not a problem if you do some tests to see the pattern. Next I show you one assembly instruction followed by the equivalent block of Code Virtualizer instructions with their respective OreansX2 structure (see the file [5] for more examples).

Figure 10: Example of Code Virtualizer syntax

```

XOR byte ptr [ESP+000000008h *EDI + 012345678h], 18
MOVE ADDR, DWORD PTR [F0000028h] - 02 00 00 80 0E 00 00 00 28 00 00 F0 00 00
SHL ADDR, 3 - 19 00 00 00 02 00 00 00 03 00 00 00 00 00
ADD ADDR, DWORD PTR [F0000038h] - 08 00 00 80 0E 00 00 00 38 00 00 F0 00 00
ADD ADDR, 12345678h - 08 00 00 00 03 00 00 00 78 56 34 12 00 00
LOAD BYTE PTR [ADDR] - 00 00 00 00 04 00 00 00 00 00 00 00 00 00
LOAD BYTE 18 - 00 00 00 00 20 00 00 00 12 00 00 00 00 00
XOR BYTE - 0E 00 00 00 18 00 00 00 00 00 00 00 00 00
STORE FLAGS - 01 00 00 00 1E 00 00 00 00 00 00 00 00 00
MOVE ADDR, DWORD PTR [F0000028h] - 02 00 00 80 0E 00 00 00 28 00 00 F0 00 00
SHL ADDR, 3 - 19 00 00 00 02 00 00 00 03 00 00 00 00 00
ADD ADDR, DWORD PTR [F0000038h] - 08 00 00 80 0E 00 00 00 38 00 00 F0 00 00
ADD ADDR, 12345678h - 08 00 00 00 03 00 00 00 78 56 34 12 00 00
ADD ADDR, 00000002h - 08 00 00 00 03 00 00 00 00 00 00 00 20 00 00
STORE BYTE PTR [ADDR] - 01 00 00 00 04 00 00 00 00 00 00 00 00 00

```

I do not know if you have noticed it, but the first parameter of the first OreansX2 structure above is 80000002h. 02 means MOVE as you can see in the Table 2, but this 80 means that this instruction has a relative address. That is, the address F0000028h is relative to the image base of the Virtual Machine.

3.2 Generating and Writing the Virtual Opcodes

Having a vector of the OreansX2 structure, now a sequence of operations will be done to reach the next structure that I have called Pre_Handler. The size of this structure is 28h bytes.

- DWORD counter // *counter that is incremented by 0Eh for each Pre_Handler structure*
- DWORD real_opcode_mark // *this DWORD is the address of the original opcode in an allocated memory. This is only applicable to the first Code Virtualizer instruction of the block of instructions that represent the original opcode*
- DWORD unknown1 // *unknown use*
- DWORD counter_0E // *this the Pre_handler.counter plus 0Eh (unknown use)*

- `BOOL is_special` // *True if the original opcode is any kind of call, jump, conditional jump and others. In this case, a special structure will be generated for those instructions*
- `BYTE instruction` // *Same as OreansX2.instruction*
- `DWORD suffix` // *Same as OreansX2.suffix*
- `DWORD data1` // *Same as OreansX2.data1*
- `DWORD data2` // *Same as OreansX2.data2*
- `WORD unknown2` // *Same as OreansX2.unknown*
- 7 bytes unknown
- `BOOL is_relative_address` // *TRUE if the instruction has a relative address*

Figure 11: Example of Pre_handler structure

Address	Hex dump	ASCII
02AE0000	00 00 00 00 00 00 F4 027E
02AE0008	00 00 00 00 0E 00 00 00A.
02AE0010	00 02 03 00 00 00 10 00>
02AE0018	00 F0 00 00 00 00 00 00
02AF0020	00 00 00 00 00 00 00 011

So now the principal structure that is directly related with the Virtual Op-codes generation can be studied. I have called this structure as Handler.

- `WORD handler` // *this is the principal parameter: it is the the one who will determine what handler must be called. It is equivalent to Handler_Information.id*
- `DWORD Pre_Handler_addr` // *address in memory of the correspondent Pre_Handler structure that generated this Handler structure*
- `DWORD memory_opcode` // *memory address of where the Virtual Op-code represented by this structure will be written*

- BYTE type_of_handler // 0 if the handler does not read Virtual Opcodes through LODS intrscution. 1, 2, 4, 8 if the handler reads 1, 2, 4, 8 Virtual Opcodes
- BYTE unknown2 // unknown use
- DWORD data1 // data for the Code Virtualizer instruction (like for example LOAD 18h, data1 will be 18h)
- DWORD data2 // data for the case of 64-bit Code Virtualizer instruction
- DWORD file_opcode // address in the protected file of where the Virtual Opcode represented by this structure will be written

Figure 12: Example of Handler structure

```

02430030 01 00 00 00 AC 02 31 00 0...%#1.
02430038 06 03 00 00 00 00 00 00 0...
02430040 00 00 00 00 3C C1 40 00 0...<@

```

Each Handler structure can generate 1, 2 or 4 Virtual Opcodes and that is a must to understand how the vector of Handler structures is generated.

This is not so complicated but if I put each case here, this article would be too big. So I will just comment how this works and if you want more details see [6].

Basically each vector of Handler structures starts with the handler 015Bh and ends with the handlers 0161h and 015Ch. The handlers 015Bh and 015Ch do not exist actually. They are there just to tell Code Virtualizer that special code must be inserted to handle when the execution of Virtual Opcodes is initiated and when it is finished. This special code will be showed shortly.

Between those handlers the Pre_handler structure is threated like this: if Pre_handler.is_special is TRUE, the handler 0161h will be added to the correspondent Handler structure. After that, a different sequence of Handlers structures is generated for each of the cases: MOVE, LOAD, STORE, SHL, ADD,

SUB, IFJMP, RET, UNDEF and default case (for the others Code Virtualizer instructions). You can see more details about those sequences in [6].

Having understood how the vector of Handler structures is generated, you can finally understand the brilliant part of Code Virtualizer: how the Virtual Opcodes are built.

The first thing to say is about when Code Virtualizer finds the handlers 015Bh and 015Ch. There is a pre-built virtualized code (this means that the Code Virtualizer instructions and the others structures are not there) that is responsible to initialize and unitalize the Virtual Machine for example, catching or returning the registers and flags before the protected application executes its Virtual Opcodes.

So now I am going to talk about the generation of Virtual Opcodes given the Handler structure. The first thing that Code Virtualizer does is quite surprising. Using a random number generator, it decides about the execution of a specific CALL. This CALL is responsible to generate "fake" Virtual Opcodes. That is, those Virtual Opcodes are going to be executed but they will not change anything in the program (like a sequence of NOPs) and so they are useful to obfuscate the real Virtual Opcodes. Besides, there are five different sequences of "fake" Virtual Opcodes difficulting even more the analysis of the program. And I say more, the option Virtual Opcode Obfuscation (low, normal, high, highest) is strictly related (I meant **only related**) with these "fake" Virtual Opcodes. Depending on that option, the chance of the random number generator allow the recursively execution of the specific CALL more than one time can be increased or decreased. So for example, in the middle of the emulation of a instruction, there can be a lot of "fake" Virtual Opcodes. They can increase the size of the Virtual Opcodes by a factor of 3!!!

Unless the "fake" Virtual Opcodes, you can say that the Virtual Opcodes would be identical if you protect an application twice and compare the Virtual Opcodes. What make them different, is a global variable in the Code Virtualizer that I have called *key*.

So if the handler 0010h must be called, given the `Handler_Information.order` and the `Special_Handler` structure (see section 2.1 and 2.2 for the explanation of these structures), the inverse operations of the ones described in Table 1 (that is ADD, SUB, XOR) will be executed to reach the correct Virtual Opcode. The things are a little confusing I think. So let's clear them.

3.3 Completing the analysis: why does this really work?

The aim of this section is to explain step-by-step the initialization of the Virtual Machine and the execution of the Virtual Opcodes. To do that, I will use a file that I prepare and that does not have fragmented handlers and mutation engine[7].

When the protected application reaches a macro, the code is redirected to a PUSH/JMP sequence in a section created by Code Virtualizer.

Figure 13: PUSH/JMP example

Address	Hex dump	Disassembly
0040AE57	> 68 E8AC4000	push handler.0040ACE8
0040AE5C	.^ E9 2FC4FFFF	jmp handler.00407290

The value pushed is the address of the first Virtual Opcode and the jump is to the main handler.

is stored in the EBX register. The ESI register has the current address of the Virtual Opcode read and the EDI register has the Image base of the Virtual Machine. The stack is used to store values and the EAX register is used for operation like XOR, ADD, etc.

So when the code reaches the address 004072D8h, the registers are like this:

Figure 16: Register in the Main Handler

```
Registers (FPU)
EAX 00000001
ECX 00000001
EDX 0014329C
EBX 0040ACE8 handler.0040ACE8
ESP 0012FB14
EBP 0012FB3C
ESI 0040ACE8 handler.0040ACE8
EDI 00407000 handler.00407000
EIP 004072D8 handler.004072D8
```

Now the byte 62h is read and after some operation with the key (those random operations explained in the section 2.2; see figure 15), when the code reaches the address 004072E4h, the registers are like this:

Figure 17: Jumping to handler 2Dh

```
Registers (FPU)
EAX 0000002D
ECX 00000001
EDX 0014329C
EBX 0040ACBB handler.0040ACBB
ESP 0012FB14
EBP 0012FB3C
ESI 0040ACE9 handler.0040ACE9
EDI 00407000 handler.00407000
EIP 004072E4 handler.004072E4
```

As you can see, the *key* was changed and the ESI register was updated. Now the code `jmp dword ptr ds:[edi+eax*4]` seems obvious: as EDI has the image base of the Virtual Machine, the EAX value obtained from the Virtual Opcode plus some operation is very important to call the handler if you notice that there is a table of pointers to handlers:

Figure 18: Piece of table of pointers to handlers

004070B0	28744000	dd handler.00407428
004070B4	33744000	dd handler.00407433
004070B8	48744000	dd handler.00407448
004070BC	54744000	dd handler.00407454
004070C0	61744000	dd handler.00407461
004070C4	6C744000	dd handler.0040746C
004070C8	77744000	dd handler.00407477
004070CC	8C744000	dd handler.0040748C
004070D0	A4744000	dd handler.004074A4
004070D4	AB744000	dd handler.004074AB
004070D8	BC744000	dd handler.004074BC

By now, you know how every handler is called and it is possible to explain why the Virtual Opcodes are unique for every protected application: because of the *key*. The *key* is changed a lot of times and it is address dependent. As the Virtual Opcodes depend on the *key* (see section 3.2 for explanation) and the size of the Virtual Machine is not constant, the Virtual Opcodes are unique.

The first two instructions of the Main Handler (PUSHAD and PUSHFD) push onto the stack the registers EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI and the Flags. After, the pre-built Virtual Opcodes that I have talked about are responsible to pop those registers in the first 38 bytes of the Virtual Machine. Now a instruction like XOR ECX, ECX will change the value in the address 00407014h. At the end of the execution of the virtualized code, the registers are restored in their correct position allowing the application to continue its execution.

Figure 19: Virtual Machine registers

00407000	.	00000000	dd 00000000	EDX
00407004	.	00000000	dd 00000000	EAX
00407008	.	00000000	dd 00000000	EDI
0040700C	.	00000000	dd 00000000	EBP
00407010	.	00000000	dd 00000000	ESI
00407014	.	00000000	dd 00000000	ECX
00407018	.	00000000	dd 00000000	EBX
0040701C	.	00000000	dd 00000000	EFlags
00407020	.	00000000	dd 00000000	
00407024	.	00000000	dd 00000000	
00407028	.	00000000	dd 00000000	
0040702C	.	00000000	dd 00000000	
00407030	.	01000000	dd 00000001	
00407034	.	00000000	dd 00000000	

Now that is your time. I will not comment every executed line. I gave you the basis and I hope that the things are more clear now. So trace the example

program [7] and understand how the others handlers are executed.

4 How to Make an Unpacked version of Code Virtualizer Full

This section is here because, I do not know if you remember that fly from unpack.cn released a version of Code Virtualizer Demo cracked, but his release only allow the use of more Virtual Machines per application and does not allow you to protect two or more macros with the same Virtual Machine. To clear the things I did this section.

The first thing you must know is where information about the protection options are stored and how they are changed to restrict the use of Code Virtualizer (yes, maybe you will not believe me but the custom options are there in the demo version; big mistake:)).

So here I show you the structure that keep information about the protection options:

Figure 20: Data structure of protection options

Address	Hex dump	Disassembly	Comment
0073A18C	. 03000000	dd 00000003	Virtual Opcodes Obfuscation
0073A190	. 03000000	dd 00000003	Virtual Machine Complexity
0073A194	. 05000000	dd 00000005	Number of Virtual Machines
0073A198	. 01000000	dd 00000001	Last Section Name
0073A19C	. 01000000	dd 00000001	Strip relocations

- Virtual Opcodes Obfuscation: 0, 1, 2, 3 are low, normal, high and highest respectively
- Virtual Machine Complexity: 0, 1, 2, 3 are low, normal, high and highest respectively
- Number of Virtual Machines: the number from 1 to 5 indicates the number os Virtual Machines that will be created
- Last Section Name: 1 if the name of the last section will not de changed

or 0 in the opposite

- Strip relocations: 1 if the relocations will be striped or 0 in the opposite

Below you can see the how the demo version works (by the way, a patch of NOPs solves the problem):

Figure 21: First patch

```
0060A10E . C705 90A17301 mov dword ptr ds:[<Virtual Machine Complexity>],1
0060A118 . C705 94A17301 mov dword ptr ds:[<Number of Virtual Machines>],1
0060A122 . C705 8CA17301 mov dword ptr ds:[<Virtual Opcodes Obfuscation>],1
```

So now I can say that the restriction of only normal Virtual Opcodes Obfuscation and Virtual Machine Complexity are broken. There is still some patches for the case of only one Macro per application.

Another important piece of code:

Figure 22: Virtual Opcodes generator function

```
0060A13C . FF35 92A87001 push dword ptr ds:[70A892]
0060A142 . 50          push eax
0060A143 . 68 BAA87000 push Code_Vir.0070A8BA   Pointer to number of macros
0060A148 . E8 9DFAFFFF call Code_Vir.00609BEA   Virtualize each Macro
```

And in the line 00609C82h, a NOP must be inserted:

Figure 23: Second patch

```
00609C82 . C745 10 0100 mov [arg.3],1           A NOP must be placed here
00609C89 . 8B45 10     mov eax,[arg.3]
00609C8C . 8945 00     mov [local.12],eax
```

Change from JE to JMP to protect more than one time your application:

Figure 24: Third patch

Address	Hex dump	Disassembly	Comment
006B63F0	74 10	je short Code_Vir.006B640F	Change it to JMP
006B63F2	6A 10	push 10	
006B63F4	B9 80796B00	mov ecx,Code_Vir.006B7980	ASCII "DEMO Limitation"
006B63F9	BA 90796B00	mov edx,Code_Vir.006B7990	ASCII "The application is already protected"

NOP this block of code to avoid the MessageBox Demo Limitation and the only one Macro limitation:

Figure 25: Fourth patch

```

006B65B0 6A 40      push 40
006B65B2 B9 DC7A6B00 mov ecx,Code_Vir.006B7ADC  ASCII "DEMO Version"
006B65B7 BA EC7A6B00 mov edx,Code_Vir.006B7AEC  ASCII "Limitations in DEMO version:"
006B65BC A1 88287400 mov eax,dword ptr ds:[742888]
006B65C1 8B00      mov eax,dword ptr ds:[eax]
006B65C3 E8 F40EE0FF call Code_Vir.004B74BC
006B65C8 . A1 EC257400 mov eax,dword ptr ds:[7425EC]
006B65CD . 8B00      mov eax,dword ptr ds:[eax]
006B65CF . C780 FC0F0000 mov dword ptr ds:[eax+FFC],1  avoid only one Macro per application

```

Change the JLE to JMP to allow you increase the number of Virtual Machines per application:

Figure 26: Fifth patch

Address	Hex dump	Disassembly	Comment
006F65A9	7E 2B	jle short Code_Vir.006F65D6	
006F65AB	6A 30	push 30	
006F65AD	B9 E4656F00	mov ecx,Code_Vir.006F65E4	ASCII "DEMO Restriction"

Congratulations, you have the Custom 1.0.1.0 version of Code Virtualizer.

5 Hopes for the Future and Acknowledgments

5.1 Why write this article?

As I said in the disclaimer, the main purpose of this article is to transmit the knowledge that I have learned to you. Besides that, I need to say that I intended to write a tool instead of this article. And I have also started it but as I am not a programmer I saw that with the amount of free time that I will have I would not be able to write this tool.

So what I hope is that someone gets interest in writing this tool (just e-mail me). I can help and even provide source code of what I have coded until now. But be aware that this is not an easy work.

An important thing to say is that this is a very resummed article. I mean there is a lot of details that I omitted (no time and so tired now to say them) and others details that I did not notice. If you have any questions or if you saw something wrong in my article or if you wants to improve this article just e-mail me.

And my main hope is to see a similar article about the Themida Virtual Machine. Let me say, this would not be too difficult now before this article mainly because Themida uses the same DLLs as Code Virtualizer and because Oreans developers themselves told us that Code Virtualizer is a version of Themida Virtual Machine a little simpler (remember the *Light VM*).

And a word for Oreans: this is really a great tool to protect sensitive code areas but as you said not 100% safe (there is not anyone 100% safe). I think there is not a similar one in the market too good as this one. Keep your good job improving this software!

5.2 The general attack approach

So here I will comment my ideas about a tool to deal with Code Virtualizer and how to threat new versions. The toll is divided in three parts:

- Preprocessing
 - Get all information about the file using the PEheader
 - Look for Virtual Machines
 - * Fill the VM class with information about the Virtual Machine
 - * Identify every handler
 - Look for Macros
 - * Get the total size of the Virtual Opcodes
 - * Find jumps to the macro. This is the place where the original code was
- Analysis

- Find the "fake" Virtual Opcodes and eliminate them
 - Retrieve the Code Virtualizer instructions
 - Analyse them and retrieve the original code
- Posprocessing
 - Save the original code in the correct place
 - Correct the PEheader
 - Save the file

The two most difficult things are to find and identify each handler in the Virtual Machine and to retrieve the original code from the block of Code Virtualizer instructions.

For the first thing I say, you have two options: study the mutation engine and do reverse engineering (very difficult); or as the mutation engine does not mutate all the opcodes I noticed that it is almost 100% possible to find each handler by their not mutated instructions.

For the second thing I say, you have two options: study how the Code Virtualizer instructions are generated from the original disassembled code and do reverse engineering (difficult); or do some tests with different kind of instructions and see the pattern. By the way, a hint is that a very well recognizable handler is used always for every original instruction: the *STORE FLAGS*. This makes the work of find the number of original instructions easier.

This tool must support different versions of Code Virtualizer. As the structure of it does not change, you need to adapt a few things for example new handlers, modified handlers, and other things.

A fun example: commands like ADD, XOR, SHL, etc. have in general three handlers; one for the byte operation, one for the word and one for the dword. But when I **first** saw the three handlers for the SHL instruction I saw something very strange:

Figure 27: Code Virtualizer bug

```
006037B1 . 66:59      pop     cx
006037B3 . 023C24     sar    byte ptr ss:[esp],cl
006037B6 . 9C        pushfd
006037B7 . 66:59      pop     cx
006037B9 . 66:033C24 sar    word ptr ss:[esp],cl
006037BD . 9C        pushfd
006037BE . 66:59      pop     cx
006037C0 . 033C24     sar    dword ptr ss:[esp],cl
006037C3 . 9C        pushfd
006037C4 . 66:59      pop     cx
006037C6 . 022424     shl   byte ptr ss:[esp],cl
006037C9 . 9C        pushfd
006037CA . 66:59      pop     cx
006037CC . 032424     shl   dword ptr ss:[esp],cl
006037CF . 9C        pushfd
006037D0 . 66:59      pop     cx
006037D2 . 032424     shl   dword ptr ss:[esp],cl
006037D5 . 9C        pushfd
```

Problem here

But **only** in the version 1.2.0.0 we saw: "[!] Fixed Virtualization of "SHL reg16, imm""[8]. Interesting, isn't it?

5.3 Acknowledgments

I must say a big thanks to people who helped me directly and indirectly to write this article. So here you are:

- Melvill, Portuogral, forgetoz and SpecOp (CRKTeam): people really important to me. They introduced me to the Reverse Engineering and helped me a lot. This article is especially dedicated to them.
- softworm: well... what can i say? Without his really good job, this article would not exist.
- Ricardo Narvaja and CrackSLatinoS: really good tutorials
- The Reverse Engineering Community (the ones where I am active): Crk-Portugal, ARTeam, Unpack.cn, Tuts4you, EXETOOLS

References

[1] Code Virtualizer Help File - *Code Virtualizer Help.chm*

- [2] <http://www.unpack.cn/viewthread.php?tid=5802&fpage=1&highlight=code%2Bvirtualizer>
- [3] OllyDbg v1.10 by Oleh Yuschuk - <http://www.ollydbg.de/>
- [4] <http://www.oreans.com/>
- [5] .. |*Annex\Example of Code Virtualizer instructions.rtf* - this file is included in the file *Inside Code Virtualizer.rar*
- [6] .. |*Annex\Analysis of Code Virtualizer instructions* - this folder is included in the file *Inside Code Virtualizer.rar*
- [7] .. |*Annex\handler.exe* - this file is included in the file *Inside Code Virtualizer.rar*
- [8] <http://www.oreans.com/CodeVirtualizerWhatsNew.php>