VMProtect 2 - Detailed Analysis of the Virtual Machine Architecture

back.engineering/17/05/2021/

May 17, 2021

calendar May 17, 2021

clock 68 min read
tag VMProtect-2 Obfuscation
codepen Author(s): _xeroxz
Download link: VMProtect 2 Reverse Engineering

Table Of Contents

Credit - Links to Existing Work

Preamble - Intentions and Purpose

Before diving into this post I would like to state a few things in regards to existing VMProtect 2 work, the purpose of this article, and my intentions, as these seem to become misconstrued and distorted at times.

Purpose

Although there has been a lot of research already conducted on VMProtect 2, I feel that there is still information which has not been discussed publicly nor enough source code disclosed to the public. The information I am disclosing in this article aims to go beyond generic architectural analysis but much lower. The level in which one could encode their own virtual machine instructions given a VMProtect'ed binary as well as intercept and alter results of virtual instructions with ease. The dynamic analysis discussed in this article is based upon existing work by Samuel Chevet, my dynamic analysis research and vmtracer project is simply an expansion upon his work demonstrated in his presentation "Inside VMProtect".

Intentions

This post is not intending to cast any negative views upon VMProtect 2, the creator(s) of said software or anyone who uses it. I admire the creator(s) who clearly have impressive skills to create such a product.

This post has also been created under the impression that everything discussed here has most likely been discovered by private entities, and that I am not the first to find or document such things about the VMProtect 2 architecture. I am not intending to present this information as though it is ground breaking or something that no one else has already discovered, quite the opposite. This is simply a collection of existing information appended with my own research.

This being said, I humbly present to you, "VMProtect 2, Detailed Analysis of the Virtual Machine Architecture".

Terminology

`VIP` - Virtual Instruction Pointer, this equivalent to the x86-64 RIP register which contains the address of the next instruction to be executed. VMProtect 2 uses the native register RSI to hold the address of the next virtual instruction pointer. Thus RSI is equivalent to VIP.

`VSP` - Virtual Stack Pointer, this is equivalent to the x86-64 RSP register which contains the address of the stack. VMProtect 2 uses the native register RBP to hold the address of the virtual stack pointer. Thus RBP is equivalent to VSP.

`VM Handler` - A routine which contains the native code to execute a virtual instruction. For example, the VADD64 instruction adds two values on the stack together and stores the result as well as RFLAGS on the stack.

`Virtual Instruction` - Also known as “virtual bytecode” is the bytes interpreted by the virtual machine and subsequently executed. Each virtual instruction is composed of at least one or more operands. The first operand contains the opcode for the instruction.

`Virtual Opcode` - The first operand of every virtual instruction. This is the vm handler index. The size of a VMProtect 2 opcode is always one byte.

`IMM / Immediate Value` - A value encoded into a virtual instruction by which operations are to happen upon, such as loading said value onto the stack or into a virtual register. Virtual instructions such as LREG, SREG, and LCONST all have immediate values.

`Transformations` - The term “transform” used throughout this post refers specifically to operations done to decrypt operands of virtual instructions and vm handler table entries. These transformations consist of add, sub, inc, dec, not, neg, shl, shr, ror, rol, and lastly BSWAP. Transformations are done with sizes of 1, 2, 4, and 8 bytes. Transformations can also have immediate/constant values associated with them such as “xor rax, 0x123456”, or “add rax, 0x123456”.

Introduction

VMProtect 2 is a virtual machine based x86 obfuscator which converts x86 instructions to a RISC, stack machine, instruction set. Each protected binary has a unique set of encrypted virtual machine instructions with unique obfuscation. This project aims to disclose very significant signatures which are in every single VMProtect 2 binary with the intent to aid in further research. This article will also briefly discuss different types of VMProtect 2 obfuscation. All techniques to deobfuscate are tailor specifically to virtual machine routines and will not work on generally obfuscated routines, specifically routines which have real JCC’s in them.

Obfuscation - Deadstore, Opaque Branching

VMProtect 2 uses two types of obfuscation for the most part, the first being deadstore, and the second being opaque branching. Throughout obfuscated routines you can see a few instructions followed by a JCC, then another set of instructions followed by another JCC. Another contributing part of opaque branching is random instructions which affect the FLAGS register. You can see these little buggers everywhere. They are mostly bit test instructions, useless compares, as well as set/clear flags instructions.

Opaque Branching Obfuscation Example

In this opaque branching obfuscation example I will go over what VMProtect 2 opaque branching looks like, other factors such as the state of rflags, and most importantly how to determine if you are looking at an opaque branch or a legitimate JCC.

```
.vmp0:00000001400073B4 D0 C8                    ror     al, 1
.vmp0:00000001400073B6 0F CA                    bswap   edx
.vmp0:00000001400073B8 66 0F CA                 bswap   dx
.vmp0:00000001400073BB 66 0F BE D2              movsx   dx, dl
.vmp0:00000001400073BF 48 FF C6                 inc     rsi
.vmp0:00000001400073C2 48 0F BA FA 0F           btc     rdx, 0Fh
.vmp0:00000001400073C7 F6 D8                    neg     al
.vmp0:00000001400073C9 0F 81 6F D0 FF FF        jno     loc_14000443E
.vmp0:00000001400073CF 66 C1 FA 04              sar     dx, 4
.vmp0:00000001400073D3 81 EA EC 94 CD 47        sub     edx, 47CD94ECh
.vmp0:00000001400073D9 28 C3                    sub     bl, al
.vmp0:00000001400073DB D2 F6                    sal     dh, cl
.vmp0:00000001400073DD 66 0F BA F2 0E           btr     dx, 0Eh
.vmp0:00000001400073E2 8B 14 38                 mov     edx, [rax+rdi]
```

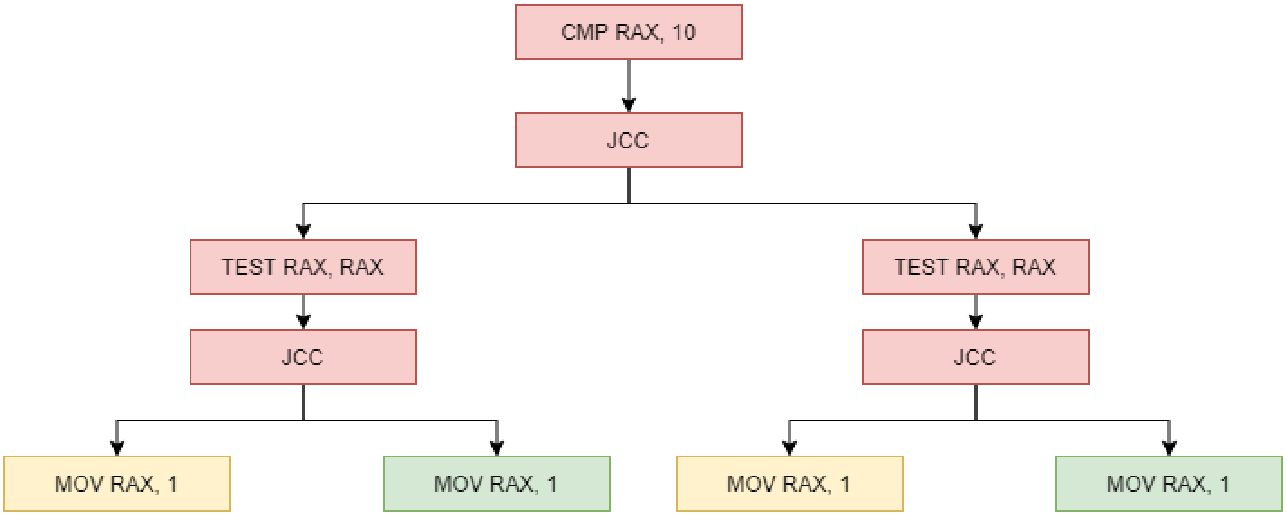
Consider the above obfuscated code. Notice the JNO branch. If you follow this branch in ida and compare the instructions against the instructions after the JNO you can see that the branch is useless as both paths execute the same meaningful instructions.

```
loc_14000443E:
.vmp0:000000014000443E F5                       cmc
.vmp0:000000014000443F 0F B3 CA                 btr     edx, ecx
.vmp0:0000000140004442 0F BE D3                 movsx   edx, bl
.vmp0:0000000140004445 66 21 F2                 and     dx, si
.vmp0:0000000140004448 28 C3                    sub     bl, al
.vmp0:000000014000444A 48 81 FA 38 04 AA 4E     cmp     rdx, 4EAA0438h
.vmp0:0000000140004451 48 8D 90 90 50 F5 BB     lea     rdx, [rax-440AAF70h]
.vmp0:0000000140004458 D2 F2                    sal     dl, cl
.vmp0:000000014000445A D2 C2                    rol     dl, cl
.vmp0:000000014000445C 8B 14 38                 mov     edx, [rax+rdi]
```

If you look close enough you can see that there are a few instructions which are in both branches. It can be difficult to determine what code is deadstore and what code is required, however if you select a register in ida and look at all the places it is written to prior to the instruction you are looking at, you can remove all of those other writing instructions up until there is a read of said register. Now, back to the example, In this case the following instructions are what matter:

```
.vmp0:0000000140004448 28 C3                    sub     bl, al
.vmp0:000000014000445C 8B 14 38                 mov     edx, [rax+rdi]
```

Generation of these opaque branches makes it so there are duplicate instructions. For each code path there is also more deadstore obfuscation as well as opaque conditions and other instructions that affect RFLAGS.



Deadstore Obfuscation Example

VMProtect 2 deadstore obfuscation adds the most junk to the instruction stream aside from opaque bit tests and comparisons. These instructions serve no purpose and can be spotted and removed by hand with ease. Consider the following:

```
.vmp0:0000000140004149 66 D3 D7                rcl     di, cl
.vmp0:000000014000414C 58                      pop     rax
.vmp0:000000014000414D 66 41 0F A4 DB 01       shld    r11w, bx, 1
.vmp0:0000000140004153 41 5B                   pop     r11
.vmp0:0000000140004155 80 E6 CA                and     dh, 0CAh
.vmp0:0000000140004158 66 F7 D7                not     di
.vmp0:000000014000415B 5F                      pop     rdi
.vmp0:000000014000415C 66 41 C1 C1 0C          rol     r9w, 0Ch
.vmp0:0000000140004161 F9                      stc
.vmp0:0000000140004162 41 58                   pop     r8
.vmp0:0000000140004164 F5                      cmc
.vmp0:0000000140004165 F8                      clc
.vmp0:0000000140004166 66 41 C1 E1 0B          shl     r9w, 0Bh
.vmp0:000000014000416B 5A                      pop     rdx
.vmp0:000000014000416C 66 81 F9 EB D2          cmp     cx, 0D2EBh
.vmp0:0000000140004171 48 0F A3 F1             bt      rcx, rsi
.vmp0:0000000140004175 41 59                   pop     r9
.vmp0:0000000140004177 66 41 21 E2             and     r10w, sp
.vmp0:000000014000417B 41 C1 D2 10             rcl     r10d, 10h
.vmp0:000000014000417F 41 5A                   pop     r10
.vmp0:0000000140004181 66 0F BA F9 0C          btc     cx, 0Ch
.vmp0:0000000140004186 49 0F CC                bswap   r12
.vmp0:0000000140004189 48 3D 97 74 7D C7       cmp     rax, 0FFFFFFFFC77D7497h
.vmp0:000000014000418F 41 5C                   pop     r12
.vmp0:0000000140004191 66 D3 C1                rol     cx, cl
.vmp0:0000000140004194 F5                      cmc
.vmp0:0000000140004195 66 0F BA F5 01          btr     bp, 1
.vmp0:000000014000419A 66 41 D3 FE             sar     r14w, cl
.vmp0:000000014000419E 5D                      pop     rbp
.vmp0:000000014000419F 66 41 29 F6             sub     r14w, si
.vmp0:00000001400041A3 66 09 F6                or      si, si
.vmp0:00000001400041A6 01 C6                   add     esi, eax
.vmp0:00000001400041A8 66 0F C1 CE             xadd    si, cx
.vmp0:00000001400041AC 9D                      popfq
.vmp0:00000001400041AD 0F 9F C1                setnle  cl
.vmp0:00000001400041B0 0F 9E C1                setle   cl
.vmp0:00000001400041B3 4C 0F BE F0             movsx   r14, al
.vmp0:00000001400041B7 59                      pop     rcx
.vmp0:00000001400041B8 F7 D1                   not     ecx
.vmp0:00000001400041BA 59                      pop     rcx
.vmp0:00000001400041BB 4C 8D A8 ED 19 28 C9    lea     r13, [rax-36D7E613h]
.vmp0:00000001400041C2 66 F7 D6                not     si
.vmp0:00000001400041CB 41 5E                   pop     r14
.vmp0:00000001400041CD 66 F7 D6                not     si
.vmp0:00000001400041D0 66 44 0F BE EA          movsx   r13w, dl
.vmp0:00000001400041D5 41 BD B2 6B 48 B7       mov     r13d, 0B7486BB2h
.vmp0:00000001400041DB 5E                      pop     rsi
.vmp0:00000001400041DC 66 41 BD CA 44          mov     r13w, 44CAh
.vmp0:0000000140007AEA 4C 8D AB 31 11 63 14    lea     r13, [rbx+14631131h]
.vmp0:0000000140007AF1 41 0F CD                bswap   r13d
.vmp0:0000000140007AF4 41 5D                   pop     r13
.vmp0:0000000140007AF6 C3                      retn
```

Let’s start from the top, one instruction at a time. The first instruction at 0x140004149 is “RCL - Rotate Left Carry”. This instruction affects the FLAGS register as well as DI. Lets see the next time DI is referenced. Is it a read or a write? The next reference to DI is the NOT instruction at 0x140004158. NOT reads and writes DI, so far both instructions are valid. The next instruction that references DI is the POP instructions. This is critical as all write’s to RDI prior to this POP can be removed from the instruction stream.

```
.vmp0:000000014000414C 58                    pop     rax
.vmp0:000000014000414D 66 41 0F A4 DB 01     shld    r11w, bx, 1
.vmp0:0000000140004153 41 5B                 pop     r11
.vmp0:0000000140004155 80 E6 CA              and     dh, 0CAh
.vmp0:000000014000415B 5F                    pop     rdi
```

The next instruction is POP RAX at `0x14000414C`. RAX is never written too throughout the entire instruction stream it is only read from. Since it has a read dependency this instruction cannot be removed. Moving onto the next instruction, `SHLD` - double precision shift left, a write dependency on R11, read dependency on BX. The next instruction that references R11 is the POP R11 at `0x140004153`. We can remove the SHLD instruction as its deadstore.

```
.vmp0:000000014000414C 58                    pop     rax
.vmp0:0000000140004153 41 5B                 pop     r11
.vmp0:0000000140004155 80 E6 CA              and     dh, 0CAh
.vmp0:000000014000415B 5F                    pop     rdi
```

Now just repeat the process for every single instruction. The end result should look something like this:

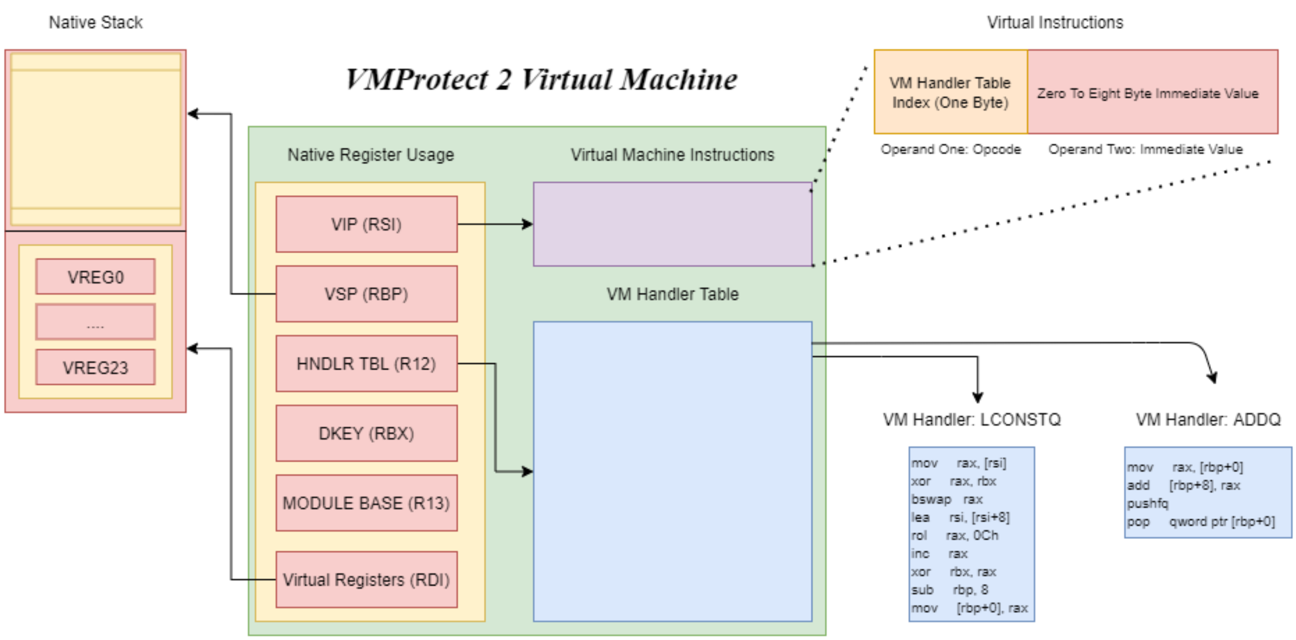
```
.vmp0:000000014000414C 58                    pop     rax
.vmp0:0000000140004153 41 5B                 pop     r11
.vmp0:000000014000415B 5F                    pop     rdi
.vmp0:0000000140004162 41 58                 pop     r8
.vmp0:000000014000416B 5A                    pop     rdx
.vmp0:0000000140004175 41 59                 pop     r9
.vmp0:000000014000417F 41 5A                 pop     r10
.vmp0:000000014000418F 41 5C                 pop     r12
.vmp0:000000014000419E 5D                    pop     rbp
.vmp0:00000001400041AC 9D                    popfq
.vmp0:00000001400041B7 59                    pop     rcx
.vmp0:00000001400041B7 59                    pop     rcx
.vmp0:00000001400041CB 41 5E                 pop     r14
.vmp0:00000001400041DB 5E                    pop     rsi
.vmp0:0000000140007AF4 41 5D                 pop     r13
.vmp0:0000000140007AF6 C3                    retn
```

This method is not perfect for removing deadstore obfuscation as there is a second POP RCX which is missing from this result above. POP and PUSH instructions are special cases which should not be emitted from the instruction stream as these instructions also change RSP. This method for removing deadstore is also only applied to vm_entry and vm handlers.

This cannot be applied to generically obfuscated routines as-is. Again, this method is NOT going to work on any obfuscated routine, it's specifically tailored for vm_entry and vm handlers as these routines have no legitimate JCC's in them.

Overview - VMProtect 2 Virtual Machine

Virtual instructions are decrypted and interpreted by virtual instruction handlers referred to as "vm handlers". The virtual machine is a RISC based stack machine with scratch registers. Prior to vm-entries an encrypted RVA (relative virtual address) to virtual instructions is pushed onto the stack and all general purpose registers as well as flags are pushed onto the stack. The VIP is decrypted, calculated, and loaded into RSI. A rolling decryption key is then started in RBX and is used to decrypt every single operand of every single virtual instruction. The rolling decryption key is updated by transforming it with the decrypted operand value.



Rolling Decryption

VMProtect 2 uses a rolling decryption key. This key is used to decrypt virtual instruction operands, which subsequently prevents any sort of hooking, as if any virtual instructions are executed out of order the rolling decryption key will become invalid causing further decryption of virtual operands to be invalid.

Native Register Usage

During execution inside of the virtual machine, some natiive registers are dedicated for the virtual machine mechanisms such as the virtual instruction pointer and virtual stack. In this section I will be discussing these native registers and their uses for the virtual machine.

Non-Volatile Registers - Registers With Specific Usage

To begin, RSI is always used for the virtual instruction pointer. Operands are fetched from the address stored in RSI. The initial value loaded into RSI is done by vm_entry.

RBP is used for the virtual stack pointer, the address stored in RBP is actually the native stack memory. RBP is loaded with RSP prior to allocation of scratch registers. This brings us to RDI which contains scratch registers. The address in RDI is initialized as well in vm_entry and is set to an address landing inside of the native stack.

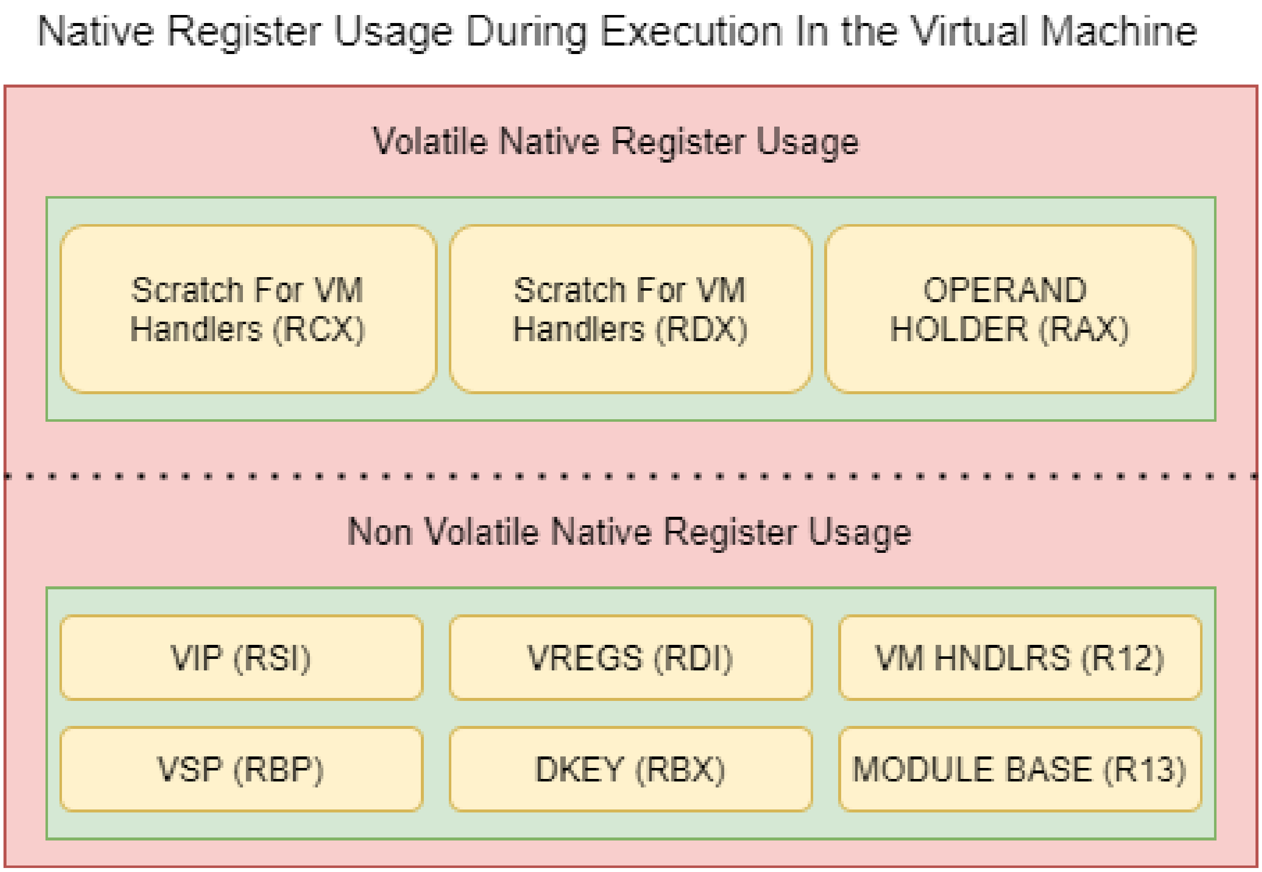
R12 is loaded with the linear virtual address of the vm handler table. This is done inside of vm_entry and throughout the entire duration of execution inside of the virtual machine R12 will contain this address.

R13 is loaded with the linear virtual address of the module base address inside of vm_entry and is not altered throughout execution inside of the virtual machine.

RBX is a very special register which contains the rolling decryption key. After every decryption of every operand of every virtual instruction RBX is updated by applying a transformation to it with the decrypted operand’s value.

Volatile Registers - Temp Registers

RAX, RCX, and RDX are used as temporary registers inside of the virtual machine, however RAX is used for very specific temporary operations over the other registers. RAX is used to decrypt operands of virtual instructions, AL specifically is used when decrypting the opcode of a virtual instruction.



vm_entry - Entering The Virtual Machine

vm_entry is a very significant component to the virtual machine architecture. Prior to entering the VM, an encrypted RVA to virtual instructions is pushed onto the stack. This RVA is a four byte value.

```
.vmp0:000000014000822C 68 FA 01 00 89          push     0FFFFFFFF890001FAh
```

After this value is pushed onto the stack, a jmp is then executed to start executing vm_entry. vm_entry is subjected to obfuscation which I explained in great detail above. By flattening and then removing deadstore code we can get a nice clean view of vm_entry.

```
> 0x822c :                                push 0xFFFFFFFF890001FA
> 0x7fc9 :                                push 0x45D3BF1F
> 0x48e4 :                                push r13
> 0x4690 :                                push rsi
> 0x4e53 :                                push r14
> 0x74fb :                                push rcx
> 0x607c :                                push rsp
> 0x4926 :                                pushfq
> 0x4dc2 :                                push rbp
> 0x5c8c :                                push r12
> 0x52ac :                                push r10
> 0x51a5 :                                push r9
> 0x5189 :                                push rdx
> 0x7d5f :                                push r8
> 0x4505 :                                push rdi
> 0x4745 :                                push r11
> 0x478b :                                push rax
> 0x7a53 :                                push rbx
> 0x500d :                                push r15
> 0x6030 :                                push [0x00000000000018E2]
> 0x593a :                                mov rax, 0x7FF634270000
> 0x5955 :                                mov r13, rax
> 0x5965 :                                push rax
> 0x596f :                                mov esi, [rsp+0xA0]
> 0x5979 :                                not esi
> 0x5985 :                                neg esi
> 0x598d :                                ror esi, 0x1A
> 0x599e :                                mov rbp, rsp
> 0x59a8 :                                sub rsp, 0x140
> 0x59b5 :                                and rsp, 0xFFFFFFFFFFFFFFF0
> 0x59c1 :                                mov rdi, rsp
> 0x59cb :                                lea r12, [0x0000000000000AA8]
> 0x59df :                                mov rax, 0x100000000
> 0x59ec :                                add rsi, rax
> 0x59f3 :                                mov rbx, rsi
> 0x59fa :                                add rsi, [rbp]
> 0x5a05 :                                mov al, [rsi]
> 0x5a0a :                                xor al, bl
> 0x5a11 :                                neg al
> 0x5a19 :                                rol al, 0x05
> 0x5a26 :                                inc al
> 0x5a2f :                                xor bl, al
> 0x5a34 :                                movzx rax, al
> 0x5a41 :                                mov rdx, [r12+rax*8]
> 0x5a49 :                                xor rdx, 0x7F3D2149
> 0x5507 :                                inc rsi
> 0x7951 :                                add rdx, r13
> 0x7954 :                                jmp rdx
```

As expected all registers as well as RFLAGS is pushed to the stack. The last push puts eight bytes of zeros on the stack, not a relocation which I first expected. The ordering in which these pushes happen are unique per-build, however the last push of eight zero's is always the same throughout all binaries. This is a very stable signature to determine when the end of general register pushes is done. Below are the exact sequences of instructions I am referring to in this paragraph.

```
> 0x48e4 :                                    push r13
> 0x4690 :                                    push rsi
> 0x4e53 :                                    push r14
> 0x74fb :                                    push rcx
> 0x607c :                                    push rsp
> 0x4926 :                                    pushfq
> 0x4dc2 :                                    push rbp
> 0x5c8c :                                    push r12
> 0x52ac :                                    push r10
> 0x51a5 :                                    push r9
> 0x5189 :                                    push rdx
> 0x7d5f :                                    push r8
> 0x4505 :                                    push rdi
> 0x4745 :                                    push r11
> 0x478b :                                    push rax
> 0x7a53 :                                    push rbx
> 0x500d :                                    push r15
> 0x6030 :                                    push [0x00000000000018E2] ; pushes 0's
```

After all registers and RFLAGS is pushed onto the stack the base address of the module is loaded into R13. This happens in every single binary, R13 always contains the base address of the module during execution of the VM. The base address of the module is also pushed onto the stack.

```
> 0x593a :                                    mov rax, 0x7FF634270000
> 0x5955 :                                    mov r13, rax
> 0x5965 :                                    push rax
```

Next, the relative virtual address of the desired virtual instructions to be executed is decrypted. This is done by loading the 32bit RVA into ESI from RSP+0xA0. This is a very significant signature and can be found trivially. Three transformations are then applied to ESI to get the decrypted RVA of the virtual instructions. The three transformations are unique per-binary. However, there are always three transformations.

```
> 0x596f :                                    mov esi, [rsp+0xA0]
> 0x5979 :                                    not esi
> 0x5985 :                                    neg esi
> 0x598d :                                    ror esi, 0x1A
```

Furthermore, the next notable operation that occurs is space allocated on the stack for scratch registers. RSP is always moved to RBP always, then RSP is subtracted by 0x140. Then aligned by 16 bytes. After this is done the address is moved into RDI. During the execution of the VM RDI always contains a pointer to scratch registers.

```
> 0x599e :                                              mov rbp, rsp
> 0x59a8 :                                              sub rsp, 0x140
> 0x59b5 :                                              and rsp, 0xFFFFFFFFFFFFFFF0
> 0x59c1 :                                              mov rdi, rsp
```

The next notable operation is loading the address of the vm handler table into R12. This is done on every single VMProtect 2 binary. R12 always contains the linear virtual address of the vm handler table. This is yet another significant signature which can be used to find the location of the vm handler table quite trivially.

```
> 0x59cb :                                              lea r12, [0x0000000000000AA8]
```

Another operation is then done on RSI to calculate VIP. Inside of the PE headers, there is a header called the “optional header”. This contains an assortment of information. One of the fields is called “ImageBase”. If there are any bits above 32 in this field those bits are then added to RSI. For example, vmptest.vmp.exe ImageBase field contains the value 0x140000000. Thus 0x100000000 is added to RSI as part of the calculation. If an ImageBase field contains less than a 32 bit value zero is added to RSI.

```
> 0x59df :                                              mov rax, 0x100000000
> 0x59ec :                                              add rsi, rax
```

After this addition is done to RSI, a small and somewhat insignificant instruction is executed. This instruction loads the linear virtual address of the virtual instructions into RBX. Now, RBX has a very special purpose, it contains the “rolling decryption” key. As you can see, the first value loaded into RBX is going to be the address of the virtual instructions themselves! Not the linear virtual address but just the RVA including the top 32bits of the ImageBase field.

```
> 0x59f3 :                                              mov rbx, rsi
```

Next, the base address of the vmp module is added to RSI computing the full, linear virtual address of the virtual instructions. Remember that RBP contains the address of RSP prior to the allocation of scratch space. The base address of the module is on the top of the stack at this point.

```
> 0x59fa :                                              add rsi, [rbp]
```

This concludes the details for vm_entry, the next part of this routine is actually referred to as “calc_vm_handler” and is executed after every single virtual instruction besides the vm_exit instruction.

calc_jmp - Decryption Of Vm Handler Index

calc_jmp is part of the vm_entry routine, however it’s referred to by more than just the vm_entry routine. Every single vm handler will eventually jump to calc_jmp (besides vm_exit). This snippet of code is responsible for decrypting the opcode of every virtual instruction as well as indexing into the vm handler table, decrypting the vm handler table entry and jumping to the resulting vm handler.

```
> 0x5a05 :                                    mov al, [rsi]
> 0x5a0a :                                    xor al, bl
> 0x5a11 :                                    neg al
> 0x5a19 :                                    rol al, 0x05
> 0x5a26 :                                    inc al
> 0x5a2f :                                    xor bl, al
> 0x5a34 :                                    movzx rax, al
> 0x5a41 :                                    mov rdx, [r12+rax*8]
> 0x5a49 :                                    xor rdx, 0x7F3D2149
> 0x5507 :                                    inc rsi
> 0x7951 :                                    add rdx, r13
> 0x7954 :                                    jmp rdx
```

The first instruction of this snippet of code reads a single byte out of RSI which as you know is VIP. This byte is an encrypted opcode. In other words it's an encrypted index into the vm handler table. There are 5 total transformations which are done. The first transformation is always applied to the encrypted opcode and the value in RBX as the source. This is the "rolling encryption" at play. It's important to note that the first value loaded into RBX is the RVA to the virtual instructions. Thus BL will contain the last byte of this RVA.

```
> 0x5a05 :                                    mov al, [rsi]
> 0x5a2f :                                    xor bl, al ; transformation is unique
to each build
```

Next, three transformations are applied to AL directly. These transformations can have immediate values, however there is never another register's value added into these transformations.

```
> 0x5a11 :                                    neg al
> 0x5a19 :                                    rol al, 0x05
> 0x5a26 :                                    inc al
```

The last transformation is applied to the rolling encryption key stored in RBX. This transformation is the same transformation as the first. However the registers swap places. The end result is the decrypted vm handler index. The value of AL is then zero extended to the rest of RAX.

```
> 0x5a2f :                                    xor bl, al
> 0x5a34 :                                    movzx rax, al
```

Now that the index into the vm handler table has been decrypted the vm handler entry itself must be fetched and decrypted. There is only a single transformation applied to these vm handler table entries. No register values are ever used in these transformations. The register in which the encrypted vm table entry value is loaded into is always RCX or RDX.

```
> 0x5a41 :                                    mov rdx, [r12+rax*8]
> 0x5a49 :                                    xor rdx, 0x7F3D2149
```

VIP is now advanced. VIP can be advanced either forward or backwards and the advancement operation itself can be an LEA, INC, DEC, ADD, or SUB instruction.

```
> 0x5507 :                                    inc rsi
```

Lastly, the base address of the module is added to the decrypted vm handler RVA and a JMP is then executed to start executing this vm handler routine. Again RDX or RCX is always used for this ADD and JMP. This is another significant signature in the virtual machine.

```
> 0x7951 :                                    add rdx, r13
> 0x7954 :                                    jmp rdx
```

This concludes the calc_jmp code snippet specifications. As you can see there are some very significant signatures which can be found trivially using Zydis. Especially the decryption done on vm handler table entries, and fetching these encrypted values.

vm_exit - Leaving The Virtual Machine

Unlike vm_entry, vm_exit is quite a straightforward routine. This routine simply POP's all registers back into place including RFLAGS. There are some redundant POP's which are used to clear the module base, padding, as well as RSP off of the stack since they are not needed. The order in which the pops occur are the inverse of the order in which they are pushed onto the stack by vm_entry. The return address is calculated and loaded onto the stack prior to the vm_exit routine.

```
.vmp0:000000014000635F 48 89 EC                mov     rsp, rbp
.vmp0:0000000140006371 58                      pop     rax ; pop module base of the
stack
.vmp0:000000014000637F 5B                      pop     rbx ; pop zero's off the stack
.vmp0:0000000140006387 41 5F                   pop     r15
.vmp0:0000000140006393 5B                      pop     rbx
.vmp0:000000014000414C 58                      pop     rax
.vmp0:0000000140004153 41 5B                   pop     r11
.vmp0:000000014000415B 5F                      pop     rdi
.vmp0:0000000140004162 41 58                   pop     r8
.vmp0:000000014000416B 5A                      pop     rdx
.vmp0:0000000140004175 41 59                   pop     r9
.vmp0:000000014000417F 41 5A                   pop     r10
.vmp0:000000014000418F 41 5C                   pop     r12
.vmp0:000000014000419E 5D                      pop     rbp
.vmp0:00000001400041AC 9D                      popfq
.vmp0:00000001400041B7 59                      pop     rcx ; pop RSP off the stack.
.vmp0:00000001400041BA 59                      pop     rcx
.vmp0:00000001400041CB 41 5E                   pop     r14
.vmp0:00000001400041DB 5E                      pop     rsi
.vmp0:0000000140007AF4 41 5D                   pop     r13
.vmp0:0000000140007AF6 C3                      retn
```

check_vsp - relocate scratch registers

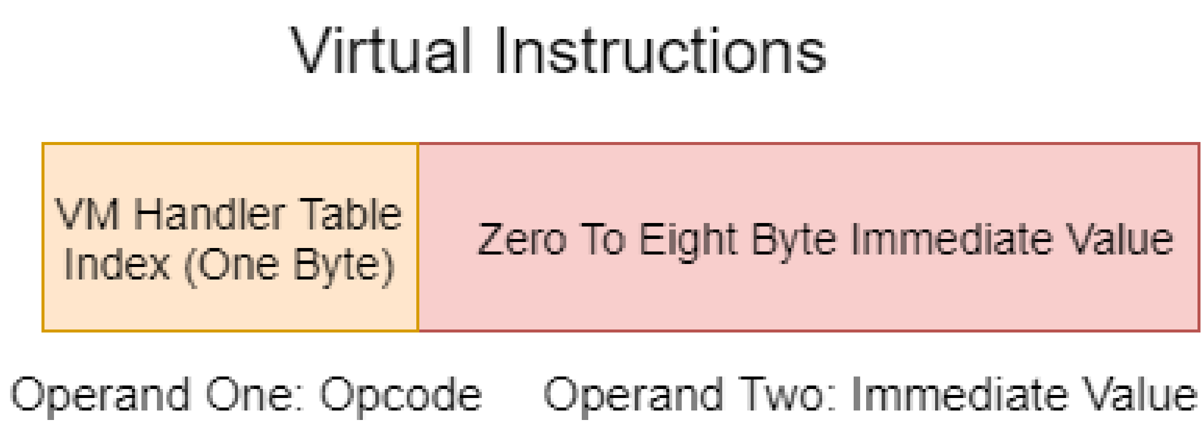
Vm handlers which put any new values on the stack will have a stack check after the vm handler executes. This routine checks to see if the stack is encroaching upon the scratch registers.

```
.vmp0:00000001400044AA 48 8D 87 E0 00 00 00       lea     rax, [rdi+0E0h]
.vmp0:00000001400044B2 48 39 C5                   cmp     rbp, rax
.vmp0:000000014000429D 0F 87 5B 17 00 00          ja      calc_jmp
.vmp0:00000001400042AC 48 89 E2                   mov     rdx, rsp
.vmp0:0000000140005E5F 48 8D 8F C0 00 00 00       lea     rcx, [rdi+0C0h]
.vmp0:0000000140005E75 48 29 D1                   sub     rcx, rdx
.vmp0:000000014000464C 48 8D 45 80                lea     rax, [rbp-80h]
.vmp0:0000000140004655 24 F0                      and     al, 0F0h
.vmp0:000000014000465F 48 29 C8                   sub     rax, rcx
.vmp0:000000014000466B 48 89 C4                   mov     rsp, rax
.vmp0:0000000140004672 9C                         pushfq
.vmp0:000000014000467C 56                         push    rsi
.vmp0:0000000140004685 48 89 D6                   mov     rsi, rdx
.vmp0:00000001400057D6 48 8D BC 01 40 FF FF FF    lea     rdi, [rcx+rax-0C0h]
.vmp0:00000001400051FC 57                         push    rdi
.vmp0:000000014000520C 48 89 C7                   mov     rdi, rax
.vmp0:0000000140004A34 F3 A4                      rep movsb
.vmp0:0000000140004A3E 5F                         pop     rdi
.vmp0:0000000140004A42 5E                         pop     rsi
.vmp0:0000000140004A48 9D                         popfq
.vmp0:0000000140004A49 E9 B0 0F 00 00             jmp     calc_jmp
```

Note the usage of "movsb" which is used to copy the contents of the scratch registers.

Virtual Instructions - Opcodes, Operands, Specifications

Virtual instructions consist of two or more operands. The first operand being the opcode of the virtual instruction. Opcodes are 8bit, unsigned values which when decrypted are the index into the vm handler table. There can be a second operand which is a one to eight byte immediate value.



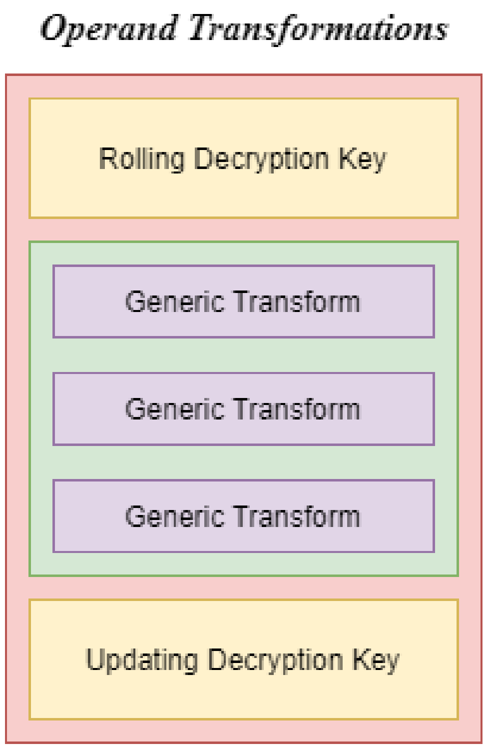
All operands are encrypted and must be decrypted with the rolling decrypt key. Decryption is done inside of calc_jmp as well as vm handlers themselves. Vm handlers that do decryption will be operating on immediate values only and not an opcode.

Operand Decryption - Transformations

VMProtect 2 encrypts its virtual instructions using a rolling decryption key. This key is located in RBX and is initially set to the address of the virtual instructions. The transformations done to decrypt operands consist of XOR, NEG, NOT, AND, ROR, ROL, SHL, SHR, ADD, SUB, INC, DEC, and BSWAP. When an operand is decrypted the first transformation applied to the operand includes the rolling decryption key. Thus only XOR, AND, ROR, ROL, ADD, and SUB are going to be the first transformation applied to the operand. Then, there are always three transformations directly applied to the operand. At this stage, the operand is completely decrypted and the value in RAX will hold the decrypted operand value. Lastly the rolling decryption key is updated by transforming the rolling decryption key with the fully decrypted operand value. An example looks like this:

```
.vmp0:0000000140005A0A 30 D8                  xor     al, bl ; decrypt using rolling
key...
.vmp0:0000000140005A11 F6 D8                  neg     al ; 1/3 transformations...
.vmp0:0000000140005A19 C0 C0 05               rol     al, 5 ; 2/3 transformations...
.vmp0:0000000140005A26 FE C0                  inc     al 3/3 transformations...
.vmp0:0000000140005A2F 30 C3                  xor     bl, al ; update rolling key...
```

This above snippet of code decrypts the first operand, which is always the instructions opcode. This code is part of the calc_jmp routine, however the transformation format is the same for any second operands.



VM Handlers - Specifications

VM handlers contain the native code to execute virtual instructions. Every VMProtect 2 binary has a vm handler table which is an array of 256 QWORD's. Each entry contains an encrypted relative virtual address to the corresponding VM handler. There are many variants of virtual instructions such as different sizes of immediate values as well as sign and zero extended values. This section will go over a few virtual instruction examples as well as some key information which must be noted when trying to parse VM handlers.

```
.vmp0:0000000140006473 vm_handler_table dq 13F3D67AFh, 13F3D7510h, 13F3D4C3Dh, 13F3D7B21h, 13F3D7510h
.vmp0:0000000140006473                                         ; DATA XREF: .vmp0:0000000140004CDF↑o
.vmp0:0000000140006473                                         ; start_0_0-25DF↑o
.vmp0:0000000140006473                 dq 13F3D578Fh, 13F3D58FBh, 13F3D4E35h, 13F3D58FBh, 13F3D6CCCh
.vmp0:0000000140006473                 dq 13F3D4E35h, 13F3D5C25h, 13F3D77EFh, 13F3D4065h, 13F3D5D96h
.vmp0:0000000140006473                 dq 13F3D6050h, 13F3D408Bh, 13F3D6049h, 2 dup(13F3D6B29h)
.vmp0:0000000140006473                 dq 13F3D4FF1h, 13F3D7976h, 13F3D5D75h, 13F3D7B21h, 13F3D578Fh
.vmp0:0000000140006473                 dq 13F3D6F80h, 13F3D6810h, 13F3D5CD1h, 13F3D7D6Eh, 13F3D6049h
.vmp0:0000000140006473                 dq 13F3D6227h, 13F3D67AFh, 13F3D7B21h, 13F3D4065h, 13F3D7976h
.vmp0:0000000140006473                 dq 13F3D6227h, 13F3D7B21h, 13F3D6F07h, 13F3D578Fh, 13F3D6049h
.vmp0:0000000140006473                 dq 13F3D4016h, 13F3D5D96h, 13F3D550Bh, 13F3D6B29h, 13F3D59BBh
.vmp0:0000000140006473                 dq 13F3D58FBh, 13F3D6050h, 13F3D6538h, 13F3D421Eh, 13F3D6804h
.vmp0:0000000140006473                 dq 2 dup(13F3D6538h), 13F3D5383h, 13F3D79D8h, 13F3D59F4h
.vmp0:0000000140006473                 dq 13F3D4E35h, 13F3D4065h, 13F3D7560h, 13F3D4065h, 13F3D436Ah
.vmp0:0000000140006473                 dq 13F3D5D75h, 13F3D432Dh, 13F3D51C2h, 13F3D4294h, 13F3D7DEFh
.vmp0:0000000140006473                 dq 13F3D7510h, 13F3D77C7h, 13F3D4294h, 13F3D6049h, 13F3D77EFh
.vmp0:0000000140006473                 dq 13F3D5853h, 13F3D550Bh, 13F3D6049h, 13F3D77C7h, 13F3D59F4h
.vmp0:0000000140006473                 dq 13F3D69B9h, 13F3D640Ah, 13F3D597Ah, 13F3D4C3Dh, 13F3D79D8h
.vmp0:0000000140006473                 dq 13F3D4CEEh, 13F3D4F50h, 13F3D6804h, 13F3D5C79h, 13F3D6F80h
.vmp0:0000000140006473                 dq 13F3D4F50h, 13F3D408Bh, 13F3D550Bh, 13F3D69B9h, 13F3D421Eh
.vmp0:0000000140006473                 dq 13F3D77EFh, 13F3D7DEFh, 13F3D5CD1h, 13F3D578Fh, 13F3D7B21h
.vmp0:0000000140006473                 dq 13F3D4016h, 13F3D5FEFh, 13F3D7D1Fh, 13F3D56A5h, 13F3D421Eh
.vmp0:0000000140006473                 dq 13F3D4016h, 13F3D7976h, 13F3D6050h, 13F3D51C2h, 13F3D7DEFh
.vmp0:0000000140006473                 dq 13F3D7B21h, 13F3D5853h, 13F3D550Bh, 13F3D4CF7h, 13F3D5B23h
.vmp0:0000000140006473                 dq 13F3D59BBh, 13F3D578Fh, 13F3D436Ah, 13F3D7B21h, 13F3D5FEFh
.vmp0:0000000140006473                 dq 13F3D5CD1h, 13F3D432Dh, 13F3D4E35h, 13F3D51C2h, 13F3D4CF7h
.vmp0:0000000140006473                 dq 13F3D7DEFh, 13F3D6F80h, 13F3D5853h, 13F3D432Dh, 13F3D69B9h
.vmp0:0000000140006473                 dq 13F3D4D91h, 13F3D597Ah, 13F3D640Ah, 13F3D4C3Dh, 13F3D5853h
.vmp0:0000000140006473                 dq 13F3D432Dh, 13F3D5853h, 13F3D5C25h, 13F3D578Fh, 13F3D5015h
.vmp0:0000000140006473                 dq 13F3D79D8h, 13F3D7560h, 13F3D7D1Fh, 13F3D69B9h, 13F3D4D91h
.vmp0:0000000140006473                 dq 13F3D77EFh, 13F3D432Dh, 13F3D4016h, 13F3D578Fh, 13F3D7B21h
.vmp0:0000000140006473                 dq 13F3D6F80h, 13F3D640Ah, 13F3D5B23h, 13F3D597Ah, 13F3D69B9h
.vmp0:0000000140006473                 dq 13F3D4065h, 13F3D7C18h, 13F3D5C25h, 13F3D677Ch, 13F3D7933h
.vmp0:0000000140006473                 dq 13F3D53FFh, 13F3D7E10h, 13F3D6810h, 13F3D7ED2h, 13F3D5383h
.vmp0:0000000140006473                 dq 13F3D4CEEh, 13F3D67AFh, 13F3D5D96h, 13F3D7B21h, 13F3D59F4h
.vmp0:0000000140006473                 dq 13F3D5D96h, 13F3D408Bh, 13F3D53FFh, 13F3D77EFh, 13F3D408Bh
.vmp0:0000000140006473                 dq 13F3D5CD1h, 13F3D7C18h, 13F3D7560h, 13F3D6F80h, 13F3D6227h
.vmp0:0000000140006473                 dq 13F3D4F50h, 13F3D7ED2h, 13F3D56A5h, 13F3D5D75h, 13F3D7D6Eh
.vmp0:0000000140006473                 dq 13F3D4294h, 13F3D6810h, 13F3D4294h, 13F3D58FBh, 13F3D5C79h
.vmp0:0000000140006473                 dq 13F3D436Ah, 13F3D578Fh, 13F3D7933h, 13F3D4C3Dh, 13F3D7CECh
.vmp0:0000000140006473                 dq 13F3D59F4h, 13F3D6227h, 13F3D67AFh, 13F3D5FEFh, 13F3D6F07h
.vmp0:0000000140006473                 dq 13F3D4CF7h, 13F3D56A5h, 13F3D77C7h, 13F3D7D1Fh, 13F3D640Ah
.vmp0:0000000140006473                 dq 13F3D69B9h, 13F3D7B21h, 13F3D6810h, 13F3D5C79h, 13F3D5CD1h
.vmp0:0000000140006473                 dq 13F3D58FBh, 13F3D6B29h, 13F3D7CECh, 13F3D432Dh, 13F3D4FF1h
.vmp0:0000000140006473                 dq 13F3D5A83h, 13F3D5B23h, 13F3D7C18h, 13F3D6810h, 13F3D58FBh
.vmp0:0000000140006473                 dq 13F3D5A83h, 13F3D408Bh, 13F3D4D91h, 13F3D56A5h, 13F3D4CF7h
.vmp0:0000000140006473                 dq 13F3D7D1Fh, 13F3D6F07h, 13F3D5C79h, 13F3D4016h, 13F3D7CECh
.vmp0:0000000140006473                 dq 13F3D6CCCh, 13F3D4CEEh, 13F3D5C79h, 13F3D5015h, 13F3D5FEFh
.vmp0:0000000140006473                 dq 13F3D59BBh, 13F3D6804h, 13F3D58FBh, 13F3D7510h, 13F3D5D75h
.vmp0:0000000140006473                 dq 13F3D432Dh, 13F3D4065h, 13F3D7560h, 13F3D7D1Fh, 13F3D6538h
.vmp0:0000000140006473                 dq 13F3D7ED2h, 13F3D5015h, 13F3D4294h, 13F3D59BBh, 13F3D7560h
.vmp0:0000000140006473                 dq 13F3D79D8h, 13F3D4D91h, 13F3D677Ch, 13F3D4065h, 13F3D56A5h
.vmp0:0000000140006473                 dq 13F3D51C2h, 13F3D5D96h, 13F3D5C79h, 13F3D7D6Eh, 13F3D6F80h
.vmp0:0000000140006473                 dq 13F3D6CCCh
```

VM handlers which handle immediate values fetch the encrypted immediate value from RSI. The traditional five transformations are then applied to this encrypted immediate value. The transformation format follows the same as the calc_jmp transformations. The first transformation is applied to the encrypted immediate value with the rolling decryption key being the source of the operation. Then three transformations are applied directly to the encrypted immediate value, this decrypts the value fully. Lastly the rolling decryption key is updated by doing the first transformation except with the destination and source operands swapped.

```
.vmp0:00000001400076D2 48 8B 06               mov     rax, [rsi] ; fetch immediate
value...
.vmp0:00000001400076D9 48 31 D8               xor     rax, rbx ; rolling key
transformation...
.vmp0:00000001400076DE 48 C1 C0 1D            rol     rax, 1Dh ; 1/3
transformations...
.vmp0:0000000140007700 48 0F C8               bswap   rax ; 2/3 transformations...
.vmp0:000000014000770F 48 C1 C0 30            rol     rax, 30h ; 3/3
transformations...
.vmp0:0000000140007714 48 31 C3               xor     rbx, rax ; update rolling
key...
```

Also note that vm handlers are subjected to opaque branching as well as deadstore obfuscation.

LCONST - Load Constant Value Onto Stack

One of the most iconic virtual machine instructions is LCONST. This virtual instruction loads a constant value from the second operand of a virtual instruction onto the stack.

LCONSTQ - Load Constant QWORD

This is the deobfuscated view of LCONSTQ VM handler. As you can see this VM handler reads the second operand of the virtual instruction out of VIP (RSI). It then decrypts this immediate value and advances VIP. The decrypted immediate value is then put onto the VSP.

```
mov     rax, [rsi]
xor     rax, rbx ; transformation
bswap   rax ; transformation
lea     rsi, [rsi+8] ; advance VIP…
rol     rax, 0Ch ; transformation
inc     rax ; transformation
xor     rbx, rax ; transformation (update rolling decrypt key)
sub     rbp, 8
mov     [rbp+0], rax
```

LCONSTCDQE - Load Constant DWORD Sign Extended to a QWORD

This virtual instruction loads a DWORD size operand from RSI, decrypts it, and extends it to a QWORD, finally putting it on the virtual stack.

```
mov     eax, [rsi]
xor     eax, ebx
xor     eax, 32B63802h
dec     eax
lea     rsi, [rsi+4] ; advance VIP
xor     eax, 7E4087EEh

; look below for details on this...
push    rbx
xor     [rsp], eax
pop     rbx

cdqe ; sign extend EAX to RAX…
sub     rbp, 8
mov     [rbp+0], rax
```

Note, this last vm handler updates the rolling decryption key by putting the value on the stack then applying the transformation. This is something that could cause significant problems when parsing these VM handlers. Luckily there is a very simple trick to handle this, always remember that the transformation applied to the rolling key is the same transformation as the first. In the above case it’s a simple XOR.

LCONSTCBW - Load Constant Byte Convert To Word

LCONSTCBW loads a constant byte value from RSI, decrypts it, and zero extends the result as a WORD value. This decrypted value is then placed upon the virtual stack.

```
movzx eax, byte ptr [rsi]
add al, bl
inc al
neg al
ror al, 0x06
add bl, al
mov ax, [rax+rdi*1]
sub rbp, 0x02
inc rsi
mov [rbp], ax
```

LCONSTCWDE - Load Constant Word Convert To DWORD

LCONSTCWDE loads a constant word from RSI, decrypts it, and sign extends it to a DWORD. Lastly the resulting value is placed upon the virtual stack.

```
mov ax, [rsi]
add rsi, 0x02
xor ax, bx
rol ax, 0x0E
xor ax, 0xA808
neg ax
xor bx, ax
cwde
sub rbp, 0x04
mov [rbp], eax
```

LCONSTDW - Load Constant DWORD

LCONSTDW loads a constant dword from RSI, decrypts it, and lastly places the result upon the virtual stack. Also note that VIP advances backwards in the example below. You can see this in the operand fetch as its subtracting from RSI prior to a dereference.

```
mov eax, [rsi-0x04]
bswap eax
add eax, ebx
dec eax
neg eax
xor eax, 0x2FFD187C
push rbx
add [rsp], eax
pop rbx
sub rbp, 0x04
mov [rbp], eax
add rsi, 0xFFFFFFFFFFFFFFFC
```

LREG - Load Scratch Register Value Onto Stack

Let’s look at another VM handler, this one by the name of LREG. Just like LCONST there are many variants of this instruction, especially for different sizes. LREG is also going to be in every single binary as it’s used inside of the VM to load register values into scratch registers. More on this later.

LREGQ - Load Scratch Register QWORD

LREGQ has a one byte immediate value. This is the scratch register index. A pointer to scratch registers is always loaded into RDI. As described above many times, there are five total transformations applied to the immediate value to decrypt it. The first transformation is applied from the rolling decryption key, followed by three transformations applied directly to the immediate value which fully decrypts it. Lastly the rolling decryption key is updated by applying the first transformation on it with the decrypted immediate value as the source.

```
mov     al, [rsi]
sub     al, bl
ror     al, 2
not     al
inc     al
sub     bl, al
mov     rdx, [rax+rdi]
sub     rbp, 8
mov     [rbp+0], rdx
inc     rsi
```

LREGDW - Load Scratch Register DWORD

LREGDW is a variant of LREG which loads a DWORD from a scratch register onto the stack. It has two operands, the second being a single byte representing the scratch register index. The snippet of code below is a deobfuscated view of LREGDW.

```
mov     al, [rsi]
sub     al, bl
add     al, 97h
ror     al, 1
neg     al
sub     bl, al
mov     edx, [rax+rdi]
sub     rbp, 4
mov     [rbp+0], edx
```

SREG - Set Scratch Register Value

Another iconic virtual instruction which is in every single binary is SREG. There are many variants to this instruction which set scratch registers to certain sizes values. This virtual instruction has two operands, the second being a single byte immediate value containing the scratch register index.

SREGQ - Set Scratch Register Value QWORD

SREGQ sets a virtual scratch register with a QWORD value from on top of the virtual stack. This virtual instruction consists of two operands, the second being a single byte representing the virtual scratch register.

```
movzx   eax, byte ptr [rsi]
sub     al, bl
ror     al, 2
not     al
inc     al
sub     bl, al
mov     rdx, [rbp+0]
add     rbp, 8
mov     [rax+rdi], rdx
```

SREGDW - Set Scratch Register Value DWORD

SREGDW sets a virtual scratch register with a DWORD value from on top of the virtual stack. This virtual instruction consists of two operands, the second being a single byte representing the virtual scratch register.

```
movzx eax, byte ptr [rsi-0x01]
xor al, bl
inc al
ror al, 0x02
add al, 0xDE
xor bl, al
lea rsi, [rsi-0x01]
mov dx, [rbp]
add rbp, 0x02
mov [rax+rdi*1], dx
```

SREGW - Set Scratch Register Value WORD

SREGW sets a virtual scratch register with a WORD value from on top of the virtual stack. This virtual instruction consists of two operands, the second being a single byte representing the virtual scratch register.

```
movzx eax, byte ptr [rsi-0x01]
sub al, bl
ror al, 0x06
neg al
rol al, 0x02
sub bl, al
mov edx, [rbp]
add rbp, 0x04
dec rsi
mov [rax+rdi*1], edx
```

SREGB - Set Scratch Register Value Byte

SREGB sets a virtual scratch register with a BYTE value from on top of the virtual stack. This virtual instruction consists of two operands, the second being a single byte representing the virtual scratch register.

```
mov al, [rsi-0x01]
xor al, bl
not al
xor al, 0x10
neg al
xor bl, al
sub rsi, 0x01
mov dx, [rbp]
add rbp, 0x02
mov [rax+rdi*1], dl
```

ADD - Add Two Values

The virtual ADD instruction adds two values on the stack together and stores the result in the second value position on the stack. RFLAGS is then pushed onto the stack as the ADD instruction alters RFLAGS.

ADDQ - Add Two QWORD Values

ADDQ adds two QWORD values stored on top of the virtual stack. RFLAGS is also pushed onto the stack as the native ADD instruction alters flags.

```
mov     rax, [rbp+0]
add     [rbp+8], rax
pushfq
pop     qword ptr [rbp+0]
```

ADDW - Add Two WORDS Values

ADDW adds two WORD values stored on top of the virtual stack. RFLAGS is also pushed onto the stack as the native ADD instruction alters flags.

```
mov ax, [rbp]
sub rbp, 0x06
add [rbp+0x08], ax
pushfq
pop [rbp]
```

ADDB - Add Two Bytes Values

ADDB adds two BYTE values stored on top of the virtual stack. RFLAGS is also pushed onto the stack as the native ADD instruction alters flags.

```
mov al, [rbp]
sub rbp, 0x06
add [rbp+0x08], al
pushfq
pop [rbp]
```

MUL - Unsigned Multiplication

The virtual MUL instruction multiples two values stored on the stack together. These vm handlers use the native MUL instruction, additionally RFLAGS is pushed onto the stack. Lastly, it is a single operand instruction which means there is no immediate value associated with this instruction.

MULQ - Unsigned Multiplication of QWORD’s

MULQ multiples two QWORD values together, the result is stored on the stack at VSP+24, additionally RFLAGS is pushed onto the stack.

```
mov rax, [rbp+0x08]
sub rbp, 0x08
mul rdx
mov [rbp+0x08], rdx
mov [rbp+0x10], rax
pushfq
pop [rbp]
```

DIV - Unsigned Division

The virtual DIV instruction uses the native DIV instruction, the top operands used in division are located on top of the virtual stack. This is a single operand virtual instruction thus there is no immediate value. RFLAGS is also pushed onto the stack as the native DIV instruction can also RFLAGS.

DIVQ - Unsigned Division Of QWORD’s

DIVQ divides two QWORD values located on the virtual stack. Push RFLAGS onto the stack.

```
mov rdx, [rbp]
mov rax, [rbp+0x08]
div [rbp+0x10]
mov [rbp+0x08], rdx
mov [rbp+0x10], rax
pushfq
pop [rbp]
```

READ - Read Memory

The READ instruction reads memory of different sizes. There is a variant of this instruction to read one, two, four, and eight bytes.

READQ - Read QWORD

READQ reads a QWORD value from the address stored on top of the stack. This virtual instruction seems to sometimes have a segment prepended to it. However not all READQ vm handlers have this `ss` associated with it. The QWORD value is now stored on top of the virtual stack.

```
mov rax, [rbp]
mov rax, ss:[rax]
mov [rbp], rax
```

READDW - Read DWORD

READDW reads a DWORD value from the address stored on top of the virtual stack. The DWORD value is then put on top of the virtual stack. Below are two examples of READDW, one which uses this segment index syntax and the other without it.

```
mov rax, [rbp]
add rbp, 0x04
mov eax, [rax]
mov [rbp], eax
```

Note the segment offset usage below with `ss` *…*

```
mov rax, [rbp]
add rbp, 0x04
mov eax, ss:[rax]
mov [rbp], eax
```

READW - Read Word

READW reads a WORD value from the address stored on top of the virtual stack. The WORD value is then put on top of the virtual stack. Below is an example of this vm handler using a segment index syntax however keep in mind there are vm handlers without this segment index.

```
mov rax, [rbp]
add rbp, 0x06
mov ax, ss:[rax]
mov [rbp], ax
```

WRITE - Write Memory

The WRITE virtual instruction writes up to eight bytes to an address. There are four variants of this virtual instruction, one for each power of two up to and including eight. There are also versions of each vm handler which use a segment offset type instruction encoding. However in longmode some segment base addresses are zero. The segment that seems to always be used is the SS segment which has the base of zero thus the segment base has no effect here, it simply makes it a little more difficult to parse these vm handlers.

WRITEQ - Write Memory QWORD

WRITEQ writes a QWORD value to the address located on top of the virtual stack. The stack is incremented by 16 bytes.

```
.vmp0:0000000140005A74 48 8B 45 00             mov     rax, [rbp+0]
.vmp0:0000000140005A82 48 8B 55 08             mov     rdx, [rbp+8]
.vmp0:0000000140005A8A 48 83 C5 10             add     rbp, 10h
.vmp0:00000001400075CF 48 89 10                mov     [rax], rdx
```

WRITEDW - Write DWORD

WRITEDW writes a DWORD value to the address located on top of the virtual stack. The stack is incremented by 12 bytes.

```
mov rax, [rbp]
mov edx, [rbp+0x08]
add rbp, 0x0C
mov [rax], edx
```

Note the segment offset `ss` usage below…

```
mov rax, [rbp]
mov edx, [rbp+0x08]
add rbp, 0x0C
mov ss:[rax], edx ; note the SS usage here...
```

WRITEW - Write WORD

The WRITEW virtual instruction writes a WORD value to the address located on top of the virtual stack. The stack is then incremented by ten bytes.

```
mov rax, [rbp]
mov dx, [rbp+0x08]
add rbp, 0x0A
mov ss:[rax], dx
```

WRITEB - Write Byte

The WRITEB virtual instruction writes a BYTE value to the address located on top of the virtual stack. The stack is then incremented by ten bytes.

```
mov rax, [rbp]
mov dl, [rbp+0x08]
add rbp, 0x0A
mov ss:[rax], dl
```

SHL - Shift Left

The SHL vm handler shifts a value located on top of the stack to the left by a number of bits. The number of bits to shift is stored above the value to be shifted on the stack. The result is then put onto the stack as well as RFLAGS.

SHLCBW - Shift Left Convert Result To WORD

SHLCBW shifts a byte value to the left and zero extends the result to a WORD. RFLAGS is pushed onto the stack.

```
mov     al, [rbp+0]
mov     cl, [rbp+2]
sub     rbp, 6
shl     al, cl
mov     [rbp+8], ax
pushfq
pop     qword ptr [rbp+0]
```

SHLW - Shift Left WORD

SHLW shifts a WORD value to the left. RFLAGS is pushed onto the virtual stack.

```
mov ax, [rbp]
mov cl, [rbp+0x02]
sub rbp, 0x06
shl ax, cl
mov [rbp+0x08], ax
pushfq
pop [rbp]
```

SHLDW - Shift Left DWORD

SHLDW shifts a DWORD to the left. RFLAGS is pushed onto the virtual stack.

```
mov eax, [rbp]
mov cl, [rbp+0x04]
sub rbp, 0x06
shl eax, cl
mov [rbp+0x08], eax
pushfq
pop [rbp]
```

SHLQ - Shift Left QWORD

SHLQ shifts a QWORD to the left. RFLAGS is pushed onto the virtual stack.

```
mov rax, [rbp]
mov cl, [rbp+0x08]
sub rbp, 0x06
shl rax, cl
mov [rbp+0x08], rax
pushfq
pop [rbp]
```

SHLD - Shift Left Double Precision

The SHLD virtual instruction shifts a value to the left using the native instruction SHLD. The result is then put onto the stack as well as RFLAGS. There is a variant of this instruction for one, two, four, and eight byte shifts.

SHLDQ - Shift Left Double Precision QWORD

SHLDQ shifts a QWORD to the left with double precision. The result is then put onto the virtual stack and RFLAGS is pushed onto the virtual stack.

```
mov rax, [rbp]
mov rdx, [rbp+0x08]
mov cl, [rbp+0x10]
add rbp, 0x02
shld rax, rdx, cl
mov [rbp+0x08], rax
pushfq
pop [rbp]
```

SHLDDW - Shift Left Double Precision DWORD

The SHLDDW virtual instruction shifts a DWORD value to the left with double precision. The result is pushed onto the virtual stack as well as RFLAGS.

```
mov eax, [rbp]
mov edx, [rbp+0x04]
mov cl, [rbp+0x08]
sub rbp, 0x02
shld eax, edx, cl
mov [rbp+0x08], eax
pushfq
pop [rbp]
```

SHR - Shift Right

The SHR instruction is the complement to SHL, this virtual instruction alters RFLAGS and thus the RFLAGS value will be on the top of the stack after executing this virtual instruction.

SHRQ - Shift Right QWORD

SHRQ shifts a QWORD value to the right. The result is put onto the virtual stack as well as RFLAGS.

```
mov rax, [rbp]
mov cl, [rbp+0x08]
sub rbp, 0x06
shr rax, cl
mov [rbp+0x08], rax
pushfq
pop [rbp]
```

SHRD - Double Precision Shift Right

The SHRD virtual instruction shifts a value to the right with double precision. There is a variant of this instruction for one, two, four, and eight byte shifts. The virtual instruction concludes with RFLAGS being pushed onto the virtual stack.

SHRDQ - Double Precision Shift Right QWORD

SHRDQ shifts a QWORD value to the right with double precision. The result is put onto the virtual stack. RFLAGS is then pushed onto the virtual stack.

```
mov rax, [rbp]
mov rdx, [rbp+0x08]
mov cl, [rbp+0x10]
add rbp, 0x02
shrd rax, rdx, cl
mov [rbp+0x08], rax
pushfq
pop [rbp]
```

SHRDDW - Double Precision Shift Right DWORD

SHRDDW shifts a DWORD value to the right with double precision. The result is put onto the virtual stack. RFLAGS is then pushed onto the virtual stack.

```
mov eax, [rbp]
mov edx, [rbp+0x04]
mov cl, [rbp+0x08]
sub rbp, 0x02
shrd eax, edx, cl
mov [rbp+0x08], eax
pushfq
pop [rbp]
```

NAND - Not Then And

The NAND instruction consists of a not being applied to the values on top of the stack, followed by the result of this not being bit wise and'ed to the next value on the stack. The and instruction alters RFLAGS thus, RFLAGS will be pushed onto the virtual stack.

NANDW - Not Then And WORD's

NANDW NOT's two WORD values then bitwise AND's them together. RFLAGs is then pushed onto the virtual stack.

```
not dword ptr [rbp]
mov ax, [rbp]
sub rbp, 0x06
and [rbp+0x08], ax
pushfq
pop [rbp]
```

READCR3 - Read Control Register Three

The READCR3 virtual instruction is a wrapper vm handler around the native `mov reg, cr3`. This instruction will put the value of CR3 onto the virtual stack.

```
mov rax, cr3
sub rbp, 0x08
mov [rbp], rax
```

WRITECR3 - Write Control Register Three

The WRITECR3 virtual instruction is a wrapper vm handler around the native `mov cr3, reg`. This instruction will put a value into CR3.

```
mov rax, [rbp]
add rbp, 0x08
mov cr3, rax
```

PUSHVSP - Push Virtual Stack Pointer

PUSHVSP virtual instruction pushes the value contained in native register RBP onto the virtual stack stack. There is a variant of this instruction for one, two, four, and eight bytes.

PUSHVSPQ - Push Virtual Stack Pointer QWORD

PUSHVSPQ pushes the entire value of the virtual stack pointer onto the virtual stack.

```
mov rax, rbp
sub rbp, 0x08
mov [rbp], rax
```

PUSHVSPDW - Push Virtual Stack Pointer DWORD

PUSHVSPDW pushes the bottom four bytes of the virtual stack pointer onto the virtual stack.

```
mov eax, ebp
sub rbp, 0x04
mov [rbp], eax
```

PUSVSPW - Push Virtual Stack Pointer WORD

PUSVSPW pushes the bottom WORD value of the virtual stack pointer onto the virtual stack.

```
mov eax, ebp
sub rbp, 0x02
mov [rbp], ax
```

LVSP - Load Virtual Stack Pointer

This virtual instruction loads the virtual stack pointer register with the value at the top of the stack.

```
mov rbp, [rbp]
```

LVSPW - Load Virtual Stack Pointer Word

This virtual instruction loads the virtual stack pointer register with the WORD value at the top of the stack.

```
mov bp, [rbp]
```

LVSPDW - Load Virtual Stack Pointer DWORD

This virtual instruction loads the virtual stack pointer register with the DWORD value at the top of the stack.

```
mov ebp, [rbp]
```

LRFLAGS - Load RFLAGS

This virtual instruction loads the native flags register with the QWORD value at the top of the stack.

```
push [rbp]
add rbp, 0x08
popfq
```

JMP - Virtual Jump Instruction

The virtual JMP instruction changes the RSI register to point to a new set of virtual instructions. The value at the top of the stack is the lower 32bits of the RVA from the module base to the virtual instructions. This value is then added to the top 32bits of the image base value found in the optional header of the PE file. The base address is then added to this value.

```
mov esi, [rbp]
add rbp, 0x08
lea r12, [0x0000000000048F29]
mov rax, 0x00 ; image base bytes above 32bits...
add rsi, rax
mov rbx, rsi ; update decrypt key
add rsi, [rbp] ; add module base address
```

CALL - Virtual Call Instruction

The virtual call instruction takes an address of the top of the virtual stack and then calls it. RDX is used to hold the address so you can only really call functions with a single parameter using this.

```
mov rdx, [rbp]
add rbp, 0x08
call rdx
```

Significant Virtual Machine Signatures - Static Analysis

Now that VMProtect 2's virtual machine architecture has been documented, we can reflect on the significant signatures. In addition, the obfuscation that VMProtect 2 generates can also be handled with quite simple techniques. This can make parsing the vm_entry routine trivial. vm_entry has no legit JCC's so everytime we encounter a JCC we can simply follow it, remove the JCC from the instruction stream, then stop once we hit a JMP RCX/RDX. We can remove most deadstore by following how an instruction is used with Zydis, specifically tracking read and write dependencies on the destination register of an instruction. Finally with the cleaned up vm_entry we can now iterate through all of the instructions and find vm handlers, transformations required to decrypt vm handler table entries, and lastly the transformations required to decrypt the relative virtual address to the virtual instructions pushed onto the stack prior to jumping to vm_entry.

Locating VM Handler Table

One of the best, and most well known signatures is LEA r12, vm_handlers. This instruction is located inside of the vm_entry snippet of code and loads the linear virtual address of the vm handler table into R12. Using Zydis we can easily locate and parse this LEA to locate the vm handler table ourselves.

```
std::uintptr_t* vm::handler::table::get(const zydis_routine_t& vm_entry)
{
    const auto result = std::find_if(
        vm_entry.begin(), vm_entry.end(),
        [](const zydis_instr_t& instr_data) -> bool
        {
            const auto instr = &instr_data.instr;
            // lea r12, vm_handlers... (always r12)...
            if (instr->mnemonic == ZYDIS_MNEMONIC_LEA &&
                instr->operands[0].type == ZYDIS_OPERAND_TYPE_REGISTER &&
                instr->operands[0].reg.value == ZYDIS_REGISTER_R12 &&
                !instr->raw.sib.base) // no register used for the sib base...
                return true;

            return false;
        }
    );

    if (result == vm_entry.end())
        return nullptr;

    std::uintptr_t ptr = 0u;
    ZydisCalcAbsoluteAddress(&result->instr,
        &result->instr.operands[1], result->addr, &ptr);

    return reinterpret_cast<std::uintptr_t*>(ptr);
}
```

The above Zydis routine will locate the address of the VM handler table statically. It only requires a vector of ZydisDecodedInstructions, one for each instruction in the vm_entry routine. My implementation of this (vmprofiler) will deobfuscate vm_entry first then pass around this vector.

Locating VM Handler Table Entry Decryption

You can easily, programmatically determine what transformation is applied to VM handler table entries by first locating the instruction which fetches entries from said table. This instruction is documented in the vm_entry section, it consists of a SIB instruction with RDX or RCX as the destination, R12 as the base, RAX as the index, and eight as the scale.

```
.vmp0:0000000140005A41 49 8B 14 C4            mov     rdx, [r12+rax*8]
```

This is easily located using Zydis. All that must be done is locate a SIB mov instruction with RCX, or RDX as the destination, R12 as the base, RAX as the index, and lastly eight as the index. Now, using Zydis we can find the next instruction with RDX or RCX as the destination, this instruction will be the transformation applied to VM handler table entries.

```
bool vm::handler::table::get_transform(
    const zydis_routine_t& vm_entry, ZydisDecodedInstruction* transform_instr)
{
    ZydisRegister rcx_or_rdx = ZYDIS_REGISTER_NONE;

    auto handler_fetch = std::find_if(
        vm_entry.begin(), vm_entry.end(),
        [&](const zydis_instr_t& instr_data) -> bool
        {
            const auto instr = &instr_data.instr;
            if (instr->mnemonic == ZYDIS_MNEMONIC_MOV &&
                instr->operand_count == 2 &&
                instr->operands[1].type == ZYDIS_OPERAND_TYPE_MEMORY &&
                instr->operands[1].mem.base == ZYDIS_REGISTER_R12 &&
                instr->operands[1].mem.index == ZYDIS_REGISTER_RAX &&
                instr->operands[1].mem.scale == 8 &&
                instr->operands[0].type == ZYDIS_OPERAND_TYPE_REGISTER &&
                (instr->operands[0].reg.value == ZYDIS_REGISTER_RDX ||
                    instr->operands[0].reg.value == ZYDIS_REGISTER_RCX))
            {
                rcx_or_rdx = instr->operands[0].reg.value;
                return true;
            }

            return false;
        }
    );

    // check to see if we found the fetch instruction and if the next instruction
    // is not the end of the vector...
    if (handler_fetch == vm_entry.end() || ++handler_fetch == vm_entry.end() ||
        // must be RCX or RDX... else something went wrong...
        (rcx_or_rdx != ZYDIS_REGISTER_RCX && rcx_or_rdx != ZYDIS_REGISTER_RDX))
        return false;

    // find the next instruction that writes to RCX or RDX...
    // the register is determined by the vm handler fetch above...
    auto handler_transform = std::find_if(
        handler_fetch, vm_entry.end(),
        [&](const zydis_instr_t& instr_data) -> bool
        {
            if (instr_data.instr.operands[0].reg.value == rcx_or_rdx &&
                instr_data.instr.operands[0].actions & ZYDIS_OPERAND_ACTION_WRITE)
                return true;
            return false;
        }
    );

    if (handler_transform == vm_entry.end())
        return false;

    *transform_instr = handler_transform->instr;
    return true;
}
```

This function will parse the vm_entry routine and return the transformation done to decrypt VM handler table entries. In C++ each transformation operation can be implemented in lambdas and a single function can be coded to return the corresponding lambda routine for the transformation that must be applied.

```
.vmp0:0000000140005A41 49 8B 14 C4            mov     rdx, [r12+rax*8]
.vmp0:0000000140005A49 48 81 F2 49 21 3D 7F   xor     rdx, 7F3D2149h
```

The above code is equivalent to the below C++ code. This will decrypt vm handler entries. To encrypt new values an inverse operation must be done. However for XOR that is simply XOR.

```
vm::decrypt_handler _decrypt_handler =
    [](std::uint8_t idx) -> std::uint64_t
{
    return vm_handlers[idx] ^ 0x7F3D2149;
};

// this is not the best example as the inverse of XOR is XOR...
vm::encrypt_handler _encrypt_handler =
    [](std::uint8_t idx) -> std::uint64_t
{
    return vm_handlers[idx] ^ 0x7F3D2149;
};
```

Handling Transformations - Templated Lambdas and Maps

The above decrypt and encrypt handlers can be dynamically generated by creating a map of each transformation type and a C++ lambda reimplementation of this instruction. Furthermore a routine to handle dynamic values such as byte sizes can be created. This prevents a switch case from being created every single time a transformation is required.

```
namespace transform
{
    // ...
    template <class T>
    inline std::map<ZydisMnemonic, transform_t<T>> transforms =
    {
        { ZYDIS_MNEMONIC_ADD, _add<T> },
        { ZYDIS_MNEMONIC_XOR, _xor<T> },
        { ZYDIS_MNEMONIC_BSWAP, _bswap<T> },
        // SUB, INC, DEC, OR, AND, ETC...
    };

    // max size of a and b is 64 bits, a and b is then converted to
    // the number of bits in bitsize, the transformation is applied,
    // finally the result is converted back to 64bits...
    inline auto apply(std::uint8_t bitsize, ZydisMnemonic op,
        std::uint64_t a, std::uint64_t b) -> std::uint64_t
    {
        switch (bitsize)
        {
        case 8:
            return transforms<std::uint8_t>[op](a, b);
        case 16:
            return transforms<std::uint16_t>[op](a, b);
        case 32:
            return transforms<std::uint32_t>[op](a, b);
        case 64:
            return transforms<std::uint64_t>[op](a, b);
        default:
            throw std::invalid_argument("invalid bit size...");
        }
    }
    // ...
}
```

This small snippet of code will allow for easy implementation of transformations in C++ with overflows in mind. It’s very important that sizes are respected during transformation as without correct size overflows as well as rolls and shifts will be incorrect. The below code is an example of how to decrypt operands of a virtual instruction by implementing the transformation in C++ dynamically.

```
// here for your eyes - better understanding of the code :^)
using map_t = std::map<transform::type, ZydisDecodedInstruction>;

auto decrypt_operand(transform::map_t& transforms,
    std::uint64_t operand, std::uint64_t rolling_key) -> std::pair<std::uint64_t,
std::uint64_t>
{
    const auto key_decrypt = &transforms[transform::type::rolling_key];
    const auto generic_decrypt_1 = &transforms[transform::type::generic1];
    const auto generic_decrypt_2 = &transforms[transform::type::generic2];
    const auto generic_decrypt_3 = &transforms[transform::type::generic3];
    const auto update_key = &transforms[transform::type::update_key];

    // apply transformation with rolling decrypt key...
    operand = transform::apply(key_decrypt->operands[0].size,
        key_decrypt->mnemonic, operand, rolling_key);

    // apply three generic transformations...
    {
        operand = transform::apply(
            generic_decrypt_1->operands[0].size,
            generic_decrypt_1->mnemonic, operand,
            // check to see if this instruction has an IMM...
            transform::has_imm(generic_decrypt_1) ?
                generic_decrypt_1->operands[1].imm.value.u : 0);

        operand = transform::apply(
            generic_decrypt_2->operands[0].size,
            generic_decrypt_2->mnemonic, operand,
            // check to see if this instruction has an IMM...
            transform::has_imm(generic_decrypt_2) ?
                generic_decrypt_2->operands[1].imm.value.u : 0);

        operand = transform::apply(
            generic_decrypt_3->operands[0].size,
            generic_decrypt_3->mnemonic, operand,
            // check to see if this instruction has an IMM...
            transform::has_imm(generic_decrypt_3) ?
                generic_decrypt_3->operands[1].imm.value.u : 0);
    }

    // update rolling key...
    rolling_key = transform::apply(key_decrypt->operands[0].size,
        key_decrypt->mnemonic, rolling_key, operand);

    return { operand, rolling_key };
}
```

Extracting Transformations - Static Analysis Continued

The ability to reimplement transformations is important, however, being able to parse the transformations out of vm handlers and calc_jmp is another problem to be solved by itself. In order to determine where transformations are we must first determine if there is a need for transformations. Transformations are only applied to operands of virtual instructions. The first operand of a virtual instruction is always transformed in the same place, this code is known as calc_jmp which I explained earlier. The second place which transforms will be found is inside of vm handlers which handle immediate values. In other words if a virtual instruction has an immediate value there will be a unique set of transformations for that operand. Immediate values are read out of VIP (RSI) so we can use this key detail to determine if there is an immediate value as well as the size of the immediate value. It's important to note that the immediate value read out of VIP does not always equal the size allocated for the decrypted value on the stack for instructions such as LCONST. This is because of sign extended and zero extended virtual instructions. Let's examine an example virtual instruction which has an immediate value. This virtual instruction is called LCONSTWSE which stands for "load constant value of size word but sign extended to a DWORD". The deobfuscated vm handler for this virtual instruction looks like so:

```
.vmp0:0000000140004478 66 0F B7 06             movzx   ax, word ptr [rsi]
.vmp0:0000000140004412 66 29 D8                sub     ax, bx
.vmp0:0000000140004416 66 D1 C0                rol     ax, 1
.vmp0:0000000140004605 66 F7 D8                neg     ax
.vmp0:000000014000460A 66 35 AC 21             xor     ax, 21ACh
.vmp0:000000014000460F 66 29 C3                sub     bx, ax
.vmp0:0000000140004613 98                      cwde
.vmp0:0000000140004618 48 83 ED 04             sub     rbp, 4
.vmp0:0000000140006E4F 89 45 00                mov     [rbp+0], eax
.vmp0:0000000140007E2D 48 8D 76 02             lea     rsi, [rsi+2]
```

As you can see there are two bytes read out of VIP. It's the first instruction. This is something we can look for in zydis. Any MOVZX, MOVSX, or MOV where RAX is the destination and RSI is the source shows that there is an immediate value and thus we know that five transformations are expected in the instruction stream. We can then search for an instruction where RAX is the destination and RBX is the source. This will be the first transformation. In the above example, the first subtraction instruction is what we are looking for.

```
.vmp0:0000000140004412 66 29 D8                sub     ax, bx
```

Next we can look for three instructions which have a write dependency on RAX. These three instructions will be the generic transformations applied to the operand.

```
.vmp0:0000000140004416 66 D1 C0                rol     ax, 1
.vmp0:0000000140004605 66 F7 D8                neg     ax
.vmp0:000000014000460A 66 35 AC 21             xor     ax, 21ACh
```

At this point the operand is completely decrypted. The only thing left is a single transformation done to the rolling decryption key (RBX). This last transformation updates the rolling decryption key.

```
.vmp0:000000014000460F 66 29 C3                sub     bx, ax
```

All of these transformation instructions can now be re-implemented by C++ lambdas on the fly. Using std::find_if is very useful for these types of searching algorithms as you can take it one step at a time. First locate the key transformations, then find the next three instructions which write to RAX.

```
bool vm::handler::get_transforms(const zydis_routine_t& vm_handler, transform::map_t&
transforms)
{
    auto imm_fetch = std::find_if(
        vm_handler.begin(), vm_handler.end(),
        [](const zydis_instr_t& instr_data) -> bool
        {
            // mov/movsx/movzx rax/eax/ax/al, [rsi]
            if (instr_data.instr.operand_count > 1 &&
                (instr_data.instr.mnemonic == ZYDIS_MNEMONIC_MOV ||
                    instr_data.instr.mnemonic == ZYDIS_MNEMONIC_MOVSX ||
                    instr_data.instr.mnemonic == ZYDIS_MNEMONIC_MOVZX) &&
                instr_data.instr.operands[0].type == ZYDIS_OPERAND_TYPE_REGISTER &&
                util::reg::compare(instr_data.instr.operands[0].reg.value,
ZYDIS_REGISTER_RAX) &&
                instr_data.instr.operands[1].type == ZYDIS_OPERAND_TYPE_MEMORY &&
                instr_data.instr.operands[1].mem.base == ZYDIS_REGISTER_RSI)
                return true;
            return false;
        }
    );

    if (imm_fetch == vm_handler.end())
        return false;

    // this finds the first transformation which looks like:
    // transform rax, rbx <--- note these registers can be smaller so we to64 them...
    auto key_transform = std::find_if(imm_fetch, vm_handler.end(),
        [](const zydis_instr_t& instr_data) -> bool
        {
            if (util::reg::compare(instr_data.instr.operands[0].reg.value,
ZYDIS_REGISTER_RAX) &&
                util::reg::compare(instr_data.instr.operands[1].reg.value,
ZYDIS_REGISTER_RBX))
                return true;
            return false;
        }
    );

    // last transformation is the same as the first except src and dest are
swapped...
    transforms[transform::type::rolling_key] = key_transform->instr;
    auto instr_copy = key_transform->instr;
    instr_copy.operands[0].reg.value = key_transform->instr.operands[1].reg.value;
    instr_copy.operands[1].reg.value = key_transform->instr.operands[0].reg.value;
    transforms[transform::type::update_key] = instr_copy;

    if (key_transform == vm_handler.end())
        return false;

    // three generic transformations...
    auto generic_transform = key_transform;

    for (auto idx = 0u; idx < 3; ++idx)
    {
```

```
        generic_transform = std::find_if(++generic_transform, vm_handler.end(),
            [](const zydis_instr_t& instr_data) -> bool
            {
                if (util::reg::compare(instr_data.instr.operands[0].reg.value,
ZYDIS_REGISTER_RAX))
                    return true;

                return false;
            }
        );

        if (generic_transform == vm_handler.end())
            return false;

        transforms[(transform::type)(idx + 1)] = generic_transform->instr;
    }

    return true;
}
```

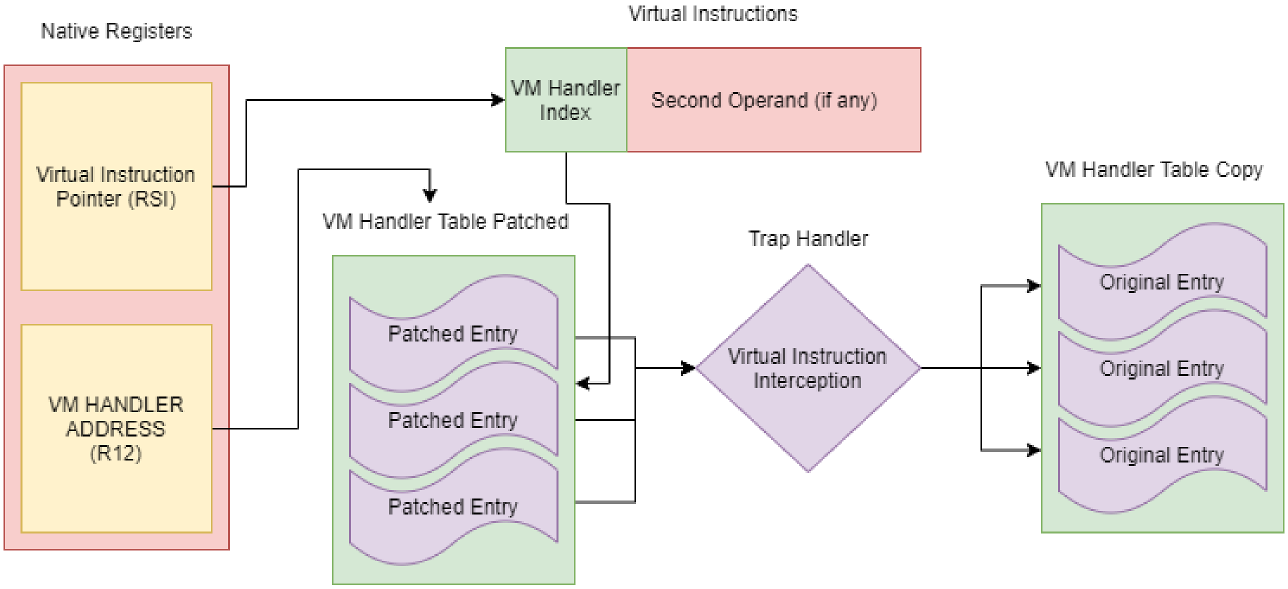
As you can see above, the first transformation is the same as the last transformation except the source and destination operands are swapped. VMProtect 2 takes some creative liberties when applying the last transformation and can sometimes push the rolling decryption key onto the stack, apply the transformation, then pop the result back into RBX. This small, but significant inconvenience can be handled by simply swapping the destination and source registers in the ZydisDecodedInstruction variable as demonstrated in the above code.

Static Analysis Dilemma - Static Analysis Conclusion

The dilemma with trying to statically analyze virtual instructions is that branching operations inside of the virtual machine are very difficult to handle. In order to calculate where a virtual JMP is jumping to, emulation is required. I will be pursuing this in the near future (unicorn).

vmtracer - Tracing Virtual Instructions

Virtual instruction tracing is trivially achievable by patching every single vm handler table entry to an encrypted value which when decrypted points to a trap handler. This will allow for inter-instruction inspection of registers as well as the possibility to alter the result of a vm handler. In order to make good usage of this feature it’s important to understand what registers contain what values. You can refer to the “Overview Section” of this post.



The first and foremost important piece of information to log when intercepting virtual instructions is the opcode value which is located in AL. Logging this will tell us all of the virtual instructions executed. The next value which must be logged is the rolling decryption key value which is located in BL. This will allow vmprofiler to decrypt operands statically.

Since we are able to, logging all scratch registers after every single virtual instruction is an important addition to the logged information as this will paint an even bigger picture of what values are being manipulated. Lastly, logging the top five QWORD values on the virtual stack is done to provide even more information as again, this virtual instruction set architecture is based off of a stack machine.

To conclude the dynamic analysis section of this post, I have created a small file format for this runtime data. The file format is called “vmp2” and contains all runtime log information. The structures for this file format are very simple, they are listed below.

```
namespace vmp2
{
    enum class exec_type_t
    {
        forward,
        backward
    };

    enum class version_t
    {
        invalid,
        v1 = 0x101
    };

    struct file_header
    {
        u32 magic; // VMP2
        u64 epoch_time;
        u64 module_base;
        exec_type_t advancement;
        version_t version;
        u32 entry_count;
        u32 entry_offset;
    };

    struct entry_t
    {
        u8 handler_idx;
        u64 decrypt_key;
        u64 vip;

        union
        {
            struct
            {
                u64 r15;
                u64 r14;
                u64 r13;
                u64 r12;
                u64 r11;
                u64 r10;
                u64 r9;
                u64 r8;
                u64 rbp;
                u64 rdi;
                u64 rsi;
                u64 rdx;
                u64 rcx;
                u64 rbx;
                u64 rax;
                u64 rflags;
            };
            u64 raw[16];
        } regs;
```

```
        union
        {
            u64 qword[0x28];
            u8 raw[0x140];
        } vregs;

        union
        {
            u64 qword[0x20];
            u8 raw[0x100];
        } vsp;
    };
}
```

vmprofile-cli - Static Analysis Using Runtime Traces

Provided a “vmp2” file, vmprofiler will produce pseudo virtual instructions including immediate values as well as affected scratch registers. This is not devirtualization by any means, nor does it provide a view of multiple code paths, however it does give a very useful trace of executed virtual instructions. Vmprofiler can also be used to statically locate the vm handler table and determine what transformation is used to decrypt these vm handler entries.

An example output of vmprofiler will produce all information about every vm handler including immediate value bit size, virtual instruction name, as well as the five transformations applied to the immediate value if there is an immediate value.

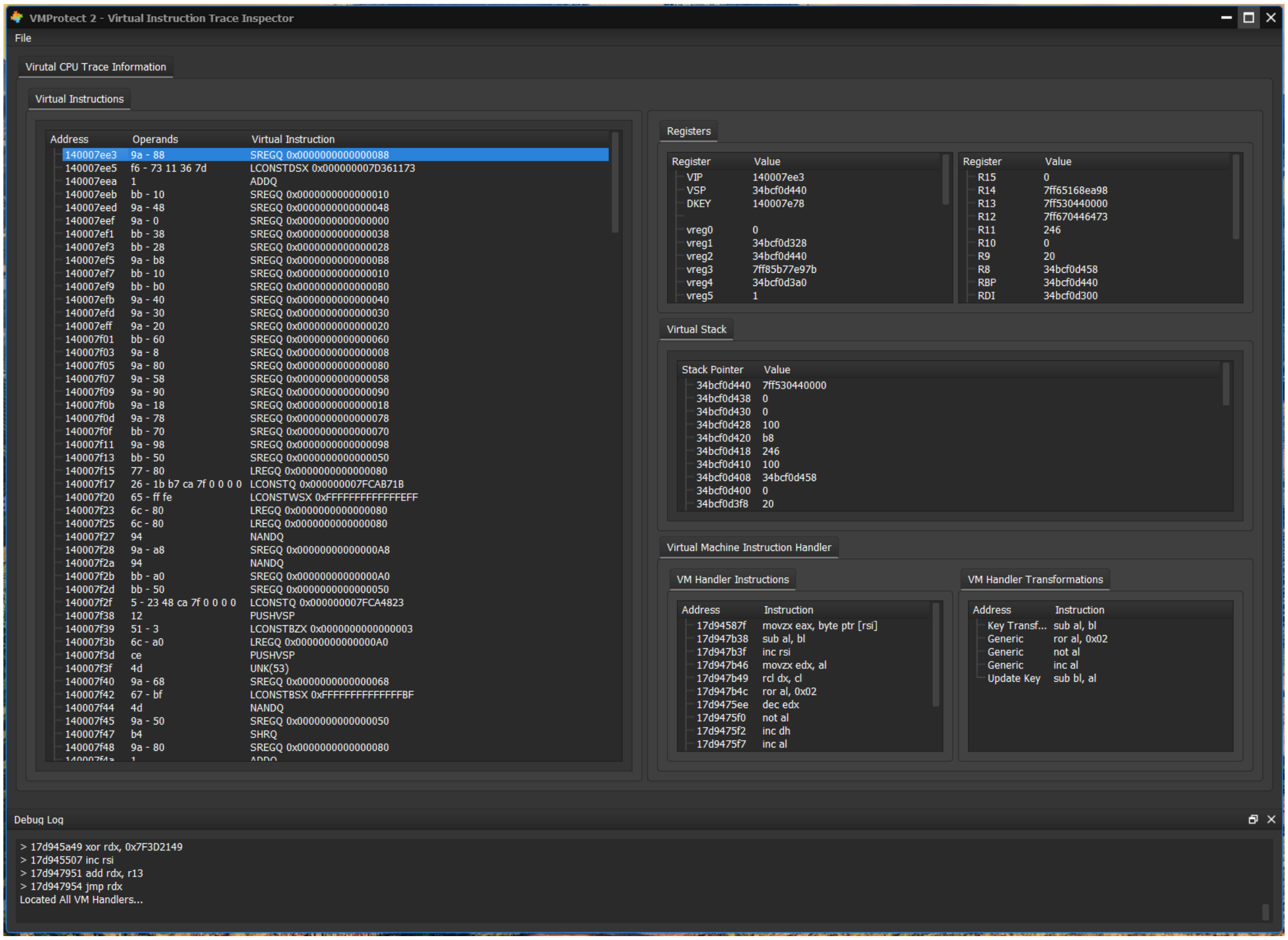
```
==========[vm handler LCONSTCBW, imm size = 8]=======
================[vm handler instructions]============
> 0x00007FF65BAE5C2E movzx eax, byte ptr [rsi]
> 0x00007FF65BAE5C82 add al, bl
> 0x00007FF65BAE5C85 add al, 0xD3
> 0x00007FF65BAE6FC7 not al
> 0x00007FF65BAE4D23 inc al
> 0x00007FF65BAE5633 add bl, al
> 0x00007FF65BAE53D5 sub rsi, 0xFFFFFFFFFFFFFFFF
> 0x00007FF65BAE5CD1 sub rbp, 0x02
> 0x00007FF65BAE62F8 mov [rbp], ax
=================[vm handler transforms]=============
add al, bl
add al, 0xD3
not al
inc al
add bl, al
=====================================================
```

The transformations, if any, are extracted as well from the vm handler and can be executed dynamically to decrypt operands.

```
> SREGQ 0x0000000000000088 (VSP[0] = 0x00007FF549600000) (VSP[1] =
0x0000000000000000)
> LCONSTDSX 0x000000007D361173 (VSP[0] = 0x0000000000000000) (VSP[1] =
0x0000000000000000)
> ADDQ (VSP[0] = 0x000000007D361173) (VSP[1] = 0x0000000000000000)
> SREGQ 0x0000000000000010 (VSP[0] = 0x0000000000000202) (VSP[1] =
0x000000007D361173)
> SREGQ 0x0000000000000048 (VSP[0] = 0x000000007D361173) (VSP[1] =
0x0000000000000000)
> SREGQ 0x0000000000000000 (VSP[0] = 0x0000000000000000) (VSP[1] =
0x0000000000000100)
> SREGQ 0x0000000000000038 (VSP[0] = 0x0000000000000100) (VSP[1] =
0x00000000000000B8)
> SREGQ 0x0000000000000028 (VSP[0] = 0x00000000000000B8) (VSP[1] =
0x0000000000000246)
> SREGQ 0x00000000000000B8 (VSP[0] = 0x0000000000000246) (VSP[1] =
0x0000000000000100)
> SREGQ 0x0000000000000010 (VSP[0] = 0x0000000000000100) (VSP[1] =
0x000000892D8FDA88)
> SREGQ 0x00000000000000B0 (VSP[0] = 0x000000892D8FDA88) (VSP[1] =
0x0000000000000000)
> SREGQ 0x0000000000000040 (VSP[0] = 0x0000000000000000) (VSP[1] =
0x0000000000000020)
> SREGQ 0x0000000000000030 (VSP[0] = 0x0000000000000020) (VSP[1] =
0x0000000000000000)
> SREGQ 0x0000000000000020 (VSP[0] = 0x0000000000000000) (VSP[1] =
0x2AAAAAAAAAAAAAAB)
// ...
```

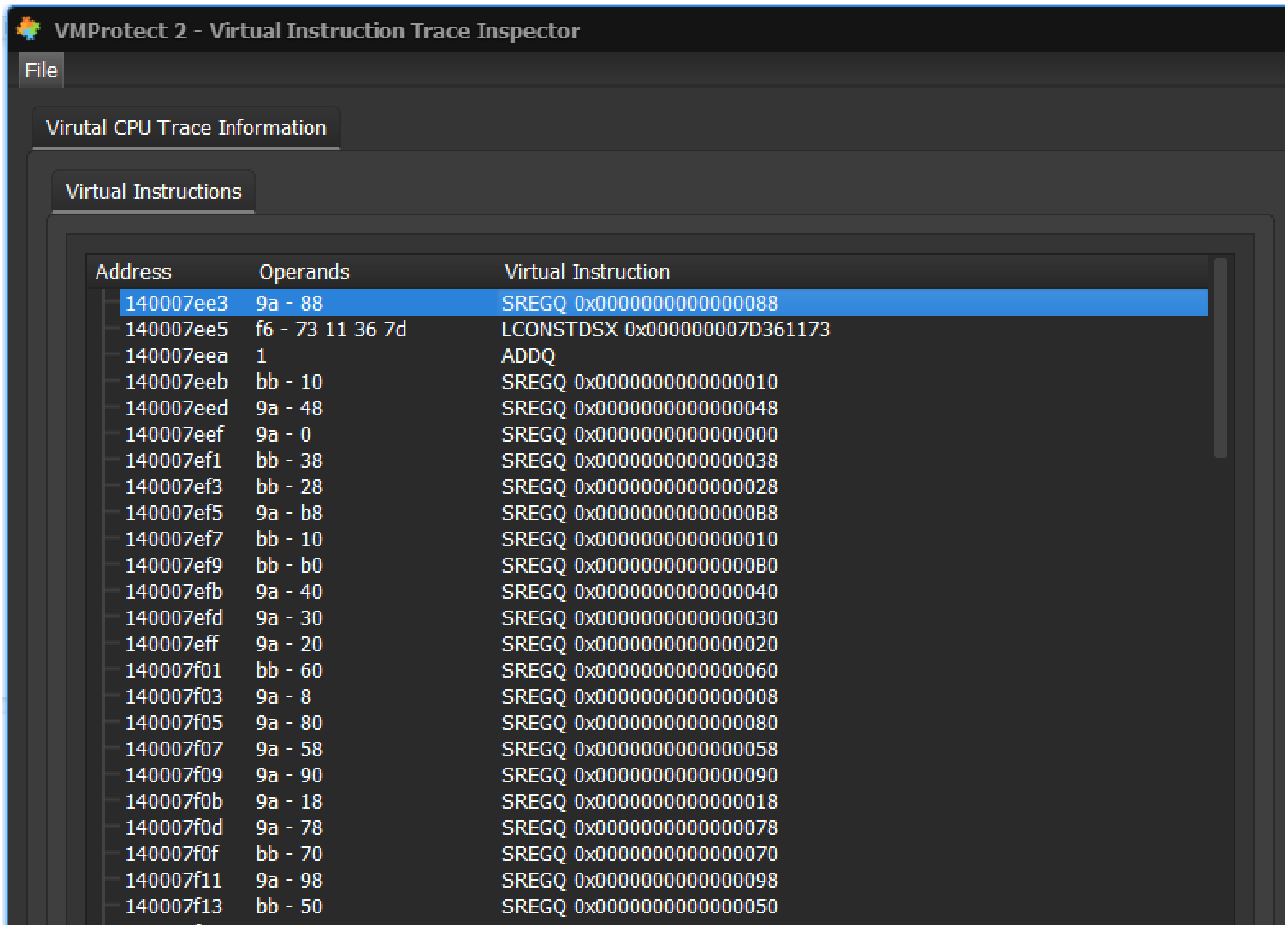
Displaying Trace Information - vmprofiler-qt

In order to display all traced information such as native register values, scratch register values and virtual stack values I have created a very small Qt project which will allow you to step through a trace. I felt that a console was way too restrictive and I also found it hard to prioritize what needs to be displayed on the console, thus the need for a GUI.



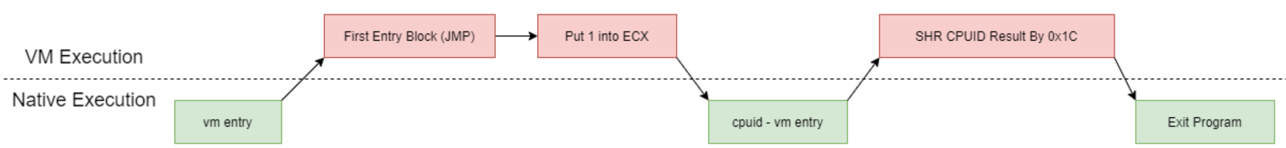
Virtual Machine Behavior

After the vm_entry routine executes, all registers that were pushed onto the stack are then loaded into virtual machine scratch registers. This also extends to the module base and RFLAGS which was also pushed onto the stack. The mapping of native registers to scratch registers is not respected.



Address	Operands	Virtual Instruction
140007ee3	9a - 88	SREGQ 0x0000000000000088
140007ee5	f6 - 73 11 36 7d	LCONSTDSX 0x000000007D361173
140007eea	1	ADDQ
140007eeb	bb - 10	SREGQ 0x0000000000000010
140007eed	9a - 48	SREGQ 0x0000000000000048
140007eef	9a - 0	SREGQ 0x0000000000000000
140007ef1	bb - 38	SREGQ 0x0000000000000038
140007ef3	bb - 28	SREGQ 0x0000000000000028
140007ef5	9a - b8	SREGQ 0x00000000000000B8
140007ef7	bb - 10	SREGQ 0x0000000000000010
140007ef9	bb - b0	SREGQ 0x00000000000000B0
140007efb	9a - 40	SREGQ 0x0000000000000040
140007efd	9a - 30	SREGQ 0x0000000000000030
140007eff	9a - 20	SREGQ 0x0000000000000020
140007f01	bb - 60	SREGQ 0x0000000000000060
140007f03	9a - 8	SREGQ 0x0000000000000008
140007f05	9a - 80	SREGQ 0x0000000000000080
140007f07	9a - 58	SREGQ 0x0000000000000058
140007f09	9a - 90	SREGQ 0x0000000000000090
140007f0b	9a - 18	SREGQ 0x0000000000000018
140007f0d	9a - 78	SREGQ 0x0000000000000078
140007f0f	bb - 70	SREGQ 0x0000000000000070
140007f11	9a - 98	SREGQ 0x0000000000000098
140007f13	bb - 50	SREGQ 0x0000000000000050

Another behavior which the virtual machine architecture exhibits is that if a native instruction is not implemented with vm handlers a vmexit will happen to execute the native instruction. In my version of VMProtect 2 CPUID is not implemented with vm handlers so an exit happens.



Prior to a vmexit, values from scratch registers are loaded onto the virtual stack. The vmexit virtual instruction will put these values back into native registers. You can see that the scratch registers are different from the ones directly after a vmentry. This is because like I said before scratch registers are not mapped to native registers.

```
140008206   6c - 28      LREGQ 0x0000000000000028
140008208   6c - 18      LREGQ 0x0000000000000018
14000820a   c3 - 10      LREGQ 0x0000000000000010
14000820c   77 - 38      LREGQ 0x0000000000000038
14000820e   6c - b0      LREGQ 0x00000000000000B0
140008210   77 - a0      LREGQ 0x00000000000000A0
140008212   6c - 88      LREGQ 0x0000000000000088
140008214   77 - 48      LREGQ 0x0000000000000048
140008216   6c - b8      LREGQ 0x00000000000000B8
140008218   6c - 8       LREGQ 0x0000000000000008
14000821a   6c - 58      LREGQ 0x0000000000000058
14000821c   6c - 20      LREGQ 0x0000000000000020
14000821e   6c - 98      LREGQ 0x0000000000000098
140008220   c3 - 68      LREGQ 0x0000000000000068
140008222   db - 80      LREGQ 0x0000000000000080
140008224   77 - 40      LREGQ 0x0000000000000040
140008226   6c - 78      LREGQ 0x0000000000000078
140008228   c3 - 28      LREGQ 0x0000000000000028
14000822a   6c - 10      LREGQ 0x0000000000000010
14000822c   30           VMEXIT
```

Demo - Creating and Inspecting A Virtual Trace

For this demo I will be virtualizing a very simple binary which just executes CPUID and returns true if AVX is supported, else it returns false. The assembly code for this is displayed below.

```
.text:00007FF776A01000 ; int __fastcall main()
.text:00007FF776A01000                  public main
.text:00007FF776A01000                  push    rbx
.text:00007FF776A01002                  sub     rsp, 10h
.text:00007FF776A01006                  xor     ecx, ecx
.text:00007FF776A01008                  mov     eax, 1
.text:00007FF776A0100D                  cpuid
.text:00007FF776A0100F                  shr     ecx, 1Ch
.text:00007FF776A01012                  and     ecx, 1
.text:00007FF776A01015                  mov     eax, ecx
.text:00007FF776A01017                  add     rsp, 10h
.text:00007FF776A0101B                  pop     rbx
.text:00007FF776A0101C                  retn
.text:00007FF776A0101C main             endp
```

When protecting this code I have opted out of using packing for simplicity of the demonstration. I have protected the binary with “Ultra" settings, which is just obfuscation + virtualization. Looking at the PE header of the output file, we can see that the entry point RVA is 0x1000, the image base is 0x140000000. We can now give this information to vmprofiler-cli and it should give us the vm handler table RVA as well as all of the vm handler information.

```
> vmprofiler-cli.exe --vmpbin vmptest.vmp.exe --vmentry 0x1000 --imagebase
0x140000000

> 0x00007FF670F2822C push 0xFFFFFFFF890001FA
> 0x00007FF670F27FC9 push 0x45D3BF1F
> 0x00007FF670F248E4 push r13
> 0x00007FF670F24690 push rsi
> 0x00007FF670F24E53 push r14
> 0x00007FF670F274FB push rcx
> 0x00007FF670F2607C push rsp
> 0x00007FF670F24926 pushfq
> 0x00007FF670F24DC2 push rbp
> 0x00007FF670F25C8C push r12
> 0x00007FF670F252AC push r10
> 0x00007FF670F251A5 push r9
> 0x00007FF670F25189 push rdx
> 0x00007FF670F27D5F push r8
> 0x00007FF670F24505 push rdi
> 0x00007FF670F24745 push r11
> 0x00007FF670F2478B push rax
> 0x00007FF670F27A53 push rbx
> 0x00007FF670F2500D push r15
> 0x00007FF670F26030 push [0x00007FF670F27912]
> 0x00007FF670F2593A mov rax, 0x7FF530F20000
> 0x00007FF670F25955 mov r13, rax
> 0x00007FF670F25965 push rax
> 0x00007FF670F2596F mov esi, [rsp+0xA0]
> 0x00007FF670F25979 not esi
> 0x00007FF670F25985 neg esi
> 0x00007FF670F2598D ror esi, 0x1A
> 0x00007FF670F2599E mov rbp, rsp
> 0x00007FF670F259A8 sub rsp, 0x140
> 0x00007FF670F259B5 and rsp, 0xFFFFFFFFFFFFFFF0
> 0x00007FF670F259C1 mov rdi, rsp
> 0x00007FF670F259CB lea r12, [0x00007FF670F26473]
> 0x00007FF670F259DF mov rax, 0x100000000
> 0x00007FF670F259EC add rsi, rax
> 0x00007FF670F259F3 mov rbx, rsi
> 0x00007FF670F259FA add rsi, [rbp]
> 0x00007FF670F25A05 mov al, [rsi]
> 0x00007FF670F25A0A xor al, bl
> 0x00007FF670F25A11 neg al
> 0x00007FF670F25A19 rol al, 0x05
> 0x00007FF670F25A26 inc al
> 0x00007FF670F25A2F xor bl, al
> 0x00007FF670F25A34 movzx rax, al
> 0x00007FF670F25A41 mov rdx, [r12+rax*8]
> 0x00007FF670F25A49 xor rdx, 0x7F3D2149
> 0x00007FF670F25507 inc rsi
> 0x00007FF670F27951 add rdx, r13
> 0x00007FF670F27954 jmp rdx
> located vm handler table... at = 0x00007FF670F26473, rva = 0x0000000140006473
```

We can see that vmprofiler-cli has flattened and deobfuscated the vm_entry code as well as located the vm handler table. We can also see the transformation done to decrypt vm handler entities, it's the XOR directly after mov rdx, [r12+rax*8].

```
> 0x00007FF670F25A41 mov rdx, [r12+rax*8]
> 0x00007FF670F25A49 xor rdx, 0x7F3D2149
```

We can also see that VIP advanced positively as RSI is incremented by the INC instruction.

```
> 0x00007FF670F25507 inc rsi
```

Armed with this information we can now compile a vmtracer program which will patch all vm handler table entries to our trap handler which will allow us to trace virtual instructions as well as alter virtual instruction results.

```
// lambdas to encrypt and decrypt vm handler entries
// you must extract this information from the flattened
// and deobfuscated view of vm_entry…

vm::decrypt_handler_t _decrypt_handler =
[](u64 val) -> u64
{

    return val ^ 0x7F3D2149;
};

vm::encrypt_handler_t _encrypt_handler =
[](u64 val) -> u64
{
    return val ^ 0x7F3D2149;
};

vm::handler::edit_entry_t _edit_entry =
[](u64* entry_ptr, u64 val) -> void
{
    DWORD old_prot;
    VirtualProtect(entry_ptr, sizeof val,
        PAGE_EXECUTE_READWRITE, &old_prot);

    *entry_ptr = val;
    VirtualProtect(entry_ptr, sizeof val,
        old_prot, &old_prot);
};

// create vm trace file header...
vmp2::file_header trace_header;
memcpy(&trace_header.magic, "VMP2", sizeof "VMP2" - 1);
trace_header.epoch_time = time(nullptr);
trace_header.entry_offset = sizeof trace_header;
trace_header.advancement = vmp2::exec_type_t::forward;
trace_header.version = vmp2::version_t::v1;
trace_header.module_base = module_base;
```

I have omitted some of the other code such as the ofstream code and vmtracer class instantiation, you can find that code here. The main purpose of displaying this information is to show you how to parse a vm_entry and extract the information which is required to create a trace.

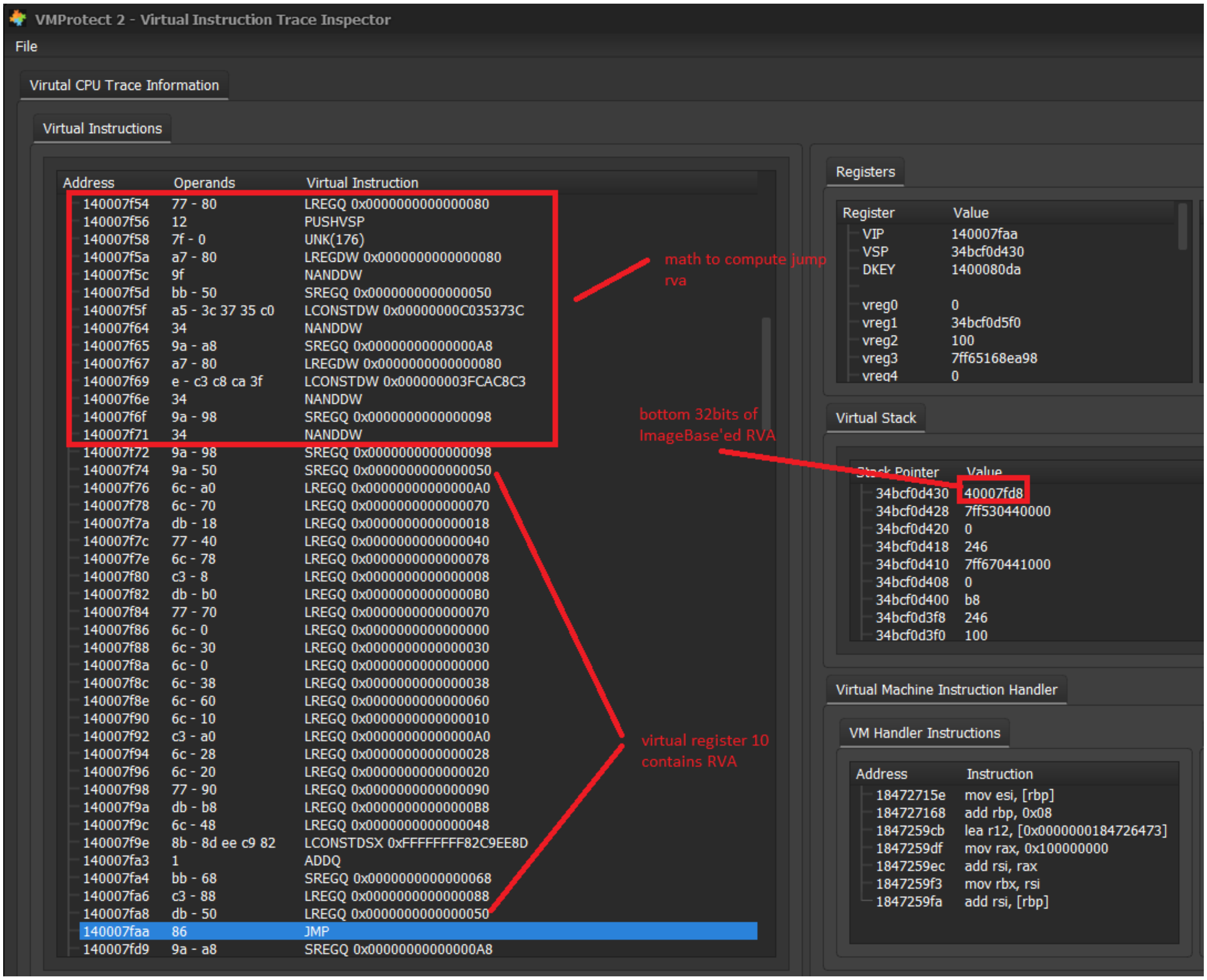
In my demo tracer I simply LoadLibraryExA the protected binary, initialize a vmtracer class, patch the vm handler table, then call the entry point of the module. This is far from ideal, however for demonstration purposes it will suffice.

```
// patch vm handler table...
tracer.start();

// call entry point...
auto result = reinterpret_cast<int (*)()>(
    NT_HEADER(module_base)->OptionalHeader.AddressOfEntryPoint + module_base)();

// unpatch vm handler table...
tracer.stop();
```

Now that a trace file has been created we can now inspect the trace via vmprofiler-cli or vmprofiler-qt. However I would suggest the latter as the program has been explicitly created to view trace files.

When loading a trace file into vmprofiler-qt, one must know the vm_entry RVA as well as the image base found in the optional header of the PE file. Given all of this information as well as the original protected binary, vmprofiler-qt will display all virtual instructions in a trace file and allow for you to “single step” through it.

Let’s look at the trace file and see if we can locate the original instructions which have now been converted to a RISC, stack machine based architecture. The first block of code that executes after vm_entry seems to contain no code pertaining to the original binary. It is here simply for obfuscation purposes and to prevent static analysis of virtual instructions as to understand where the virtual JMP instruction is going to land would require emulation of the virtual instruction set. This first jump block is located inside of every single protected binary.



The next block following the virtual JMP instruction does a handful of interesting math operations pertaining to the stack. If you look closely you can see that the math operation being executed is: `sub(x, y) = ~((~(x) & ~(x)) + y) & ~((~(x) & ~(x)) + y); sub(VSP, 10)`.

If we simplify this math operation we can see that the operation is a subtraction done to VSP. `sub(x, y) = ~((~x) + y)`. This is equivalent to the native operation `sub rsp, 0x10`. If we look at the original binary, the one that is not virtualized, we can see that there is in fact this instruction.

```
.text:00007FF776A01000
.text:00007FF776A01000 ; int __fastcall main()
.text:00007FF776A01000                 public main
.text:00007FF776A01000 main            proc near               ; DATA XREF: .pdata:ExceptionDir↓o
.text:00007FF776A01000                 push    rbx
.text:00007FF776A01002                 sub     rsp, 10h
.text:00007FF776A01006                 xor     ecx, ecx
.text:00007FF776A01008                 mov     eax, 1
.text:00007FF776A0100D                 cpuid
.text:00007FF776A0100F                 shr     ecx, 1Ch
.text:00007FF776A01012                 and     ecx, 1
.text:00007FF776A01015                 mov     eax, ecx
.text:00007FF776A01017                 add     rsp, 10h
.text:00007FF776A0101B                 pop     rbx
.text:00007FF776A0101C                 retn
.text:00007FF776A0101C main            endp
.text:00007FF776A0101C
```

The mov eax, 1 displayed above can be seen in the virtual instructions closely after the subtraction done on VSP. The MOV EAX, 1 is done via a LCONSTBSX and a SREGDW. The SREG bitsize matches the native register width of 32bits, as well as the constant value being loaded into it.

```
14000806f    0 - 1      LCONSTBSX 0x0000000000000001
140008071    4c - 30    SREGDW 0x0000000000000030
```

Next we see that a vmexit happens. We can see where code execution will continue outside of the virtual machine by going to the last ADDQ prior to the vmexit. The first two values on the stack should be the module base address and 32bit relative virtual address to the routine that will be returned to. In this trace the RVA is 0x140008236. If we inspect this address in IDA we can see that the instruction "CPUID" is here.

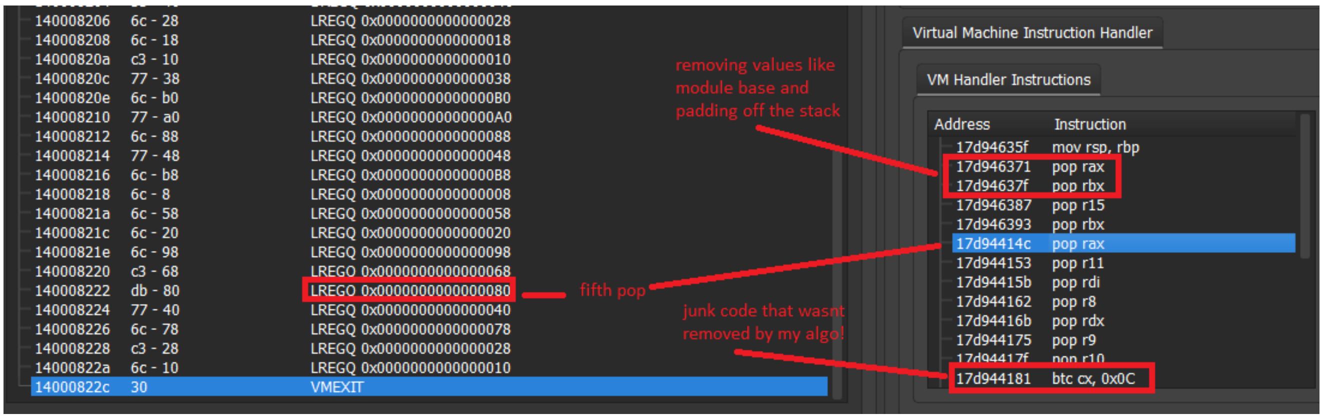
```
.vmp0:0000000140008236 0F A2                                     cpuid
.vmp0:0000000140008238 0F 81 88 FE FF FF                         jno
loc_1400080C6
.vmp0:000000014000823E 68 05 02 00 79                            push
79000205h
.vmp0:0000000140008243 E9 77 FD FF FF                            jmp
loc_140007FBF
```

As you can see, directly after the CPUID instruction, code execution enters back into the virtual machine. Directly after setting all virtual scratch registers with native register values located on the virtual stack a constant is loaded onto the stack with the value of 0x1C. The resulting value from CPUID is then shifted to the right by this constant value.

```
1400081d1    55 - 1c    LCONSTBZX 0x000000000000001C
1400081d3    a7 - 38    LREGDW 0x0000000000000038
1400081d5    60         SHRDW
1400081d6    9a - 30    SREGQ 0x0000000000000030
1400081d8    92 - 38    SREGDW 0x0000000000000038
1400081da    0 - 0      LCONSTBSX 0x0000000000000000
1400081dc    92 - 3c    SREGDW 0x000000000000003C
```

The AND operation is done with two NAND operations. The first NAND simply inverts the result from the SHR; `invert(x) = ~(x) & ~(x)`. This is done by loading the DWORD value twice onto the stack to make a single QWORD.

The result of this AND operation is then set into virtual scratch register seven (SREGDW 0x38). It is then moved into scratch register 16. If we look at the vmexit instruction and the order in which LREGQ's are executed we can see that this is indeed correct.



Lastly, we can also see the ADD instruction and LVSP instruction which adds a value to VSP. This is expected as there is an ADD RSP, 0x10 in the original binary.

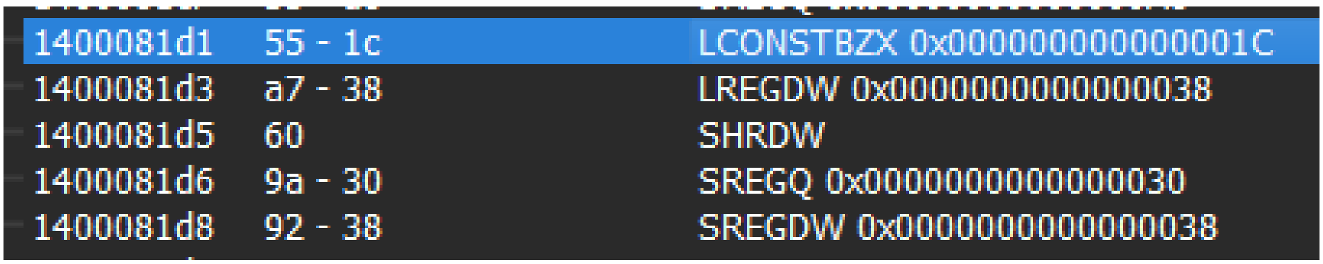
From the information above we can reconstruct the following native instructions:

```
sub rsp, 0x10
mov eax, 1
cpuid
shr ecx, 0x1C
and ecx, 1
mov eax, ecx ; from the LREGDW 0x38; SREGDW 0x80...
add rsp, 0x10
ret
```

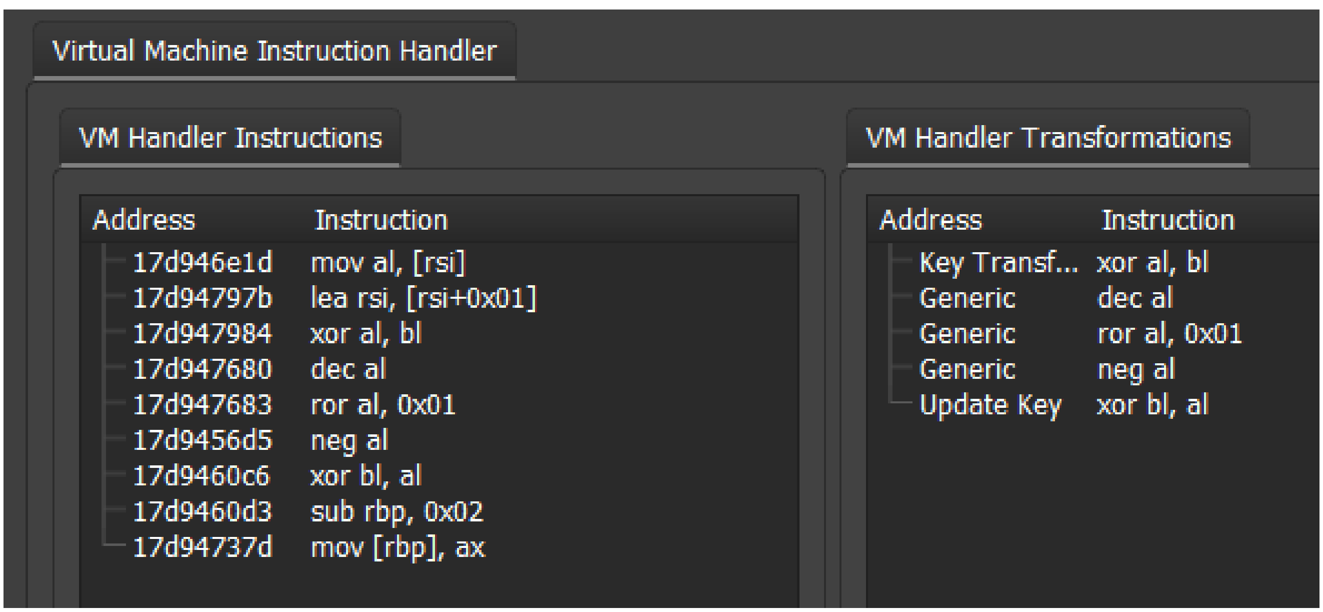
As you can see there are a few instructions which are missing, particularly the push's and pop's of RBX, as well as the XOR to zero the contents of ECX. I assume that these instructions are not converted to virtual instructions directly and are instead implemented in a roundabout way.

Altering Virtual Instruction Results

In order to alter virtual instructions one must reimplement the entire vm handler first. If the vm handler decrypts a second operand one must remember the importance of the decryption key validity. Thus the original immediate value must be computed and applied to the decryption key via the original transformation. However this value can be subsequently discarded after updating the decryption key. An example of this could be altering the constant value from the LCONST prior to the SHR in the above section.



This virtual instruction has two operands, the first being the vm handler index to execute and the second being the immediate value which in this case is a single byte. Since there are two operands there will be five transformations inside of the vm handler.



We can recode this vm handler and compare the decrypted immediate value with 0x1C, then branch to a subroutine to load a different value onto the stack. This will then result in the SHR computing a different result. Essentially we can spoof the CPUID results. An alternative to this would be recreating the SHR handler, however for simplicity sake i’m just going to shift to a bit that is set. In this case bit 5 in ECX after CPUID is set if VMX is supported and since my CPU supports virtualization this bit will be high. Below is the new vm handler.

```
.data
    __mbase dq 0h
    public __mbase

.code
__lconstbzx proc
    mov al, [rsi]
    lea rsi, [rsi+1]
    xor al, bl
    dec al
    ror al, 1
    neg al
    xor bl, al

    pushfq              ; save flags...
    cmp ax, 01Ch
    je swap_val

                    ; the constant is not 0x1C
    popfq            ; restore flags...
    sub rbp, 2
    mov [rbp], ax
    mov rax, __mbase
    add rax, 059FEh    ; calc jmp rva is 0x59FE...
    jmp rax

swap_val:            ; the constant is 0x1C
    popfq            ; restore flags...
    mov ax, 5        ; bit 5 is VMX in ECX after CPUID...
    sub rbp, 2
    mov [rbp], ax
    mov rax, __mbase
    add rax, 059FEh    ; calc jmp rva is 0x59FE...
    jmp rax
__lconstbzx endp
end
```

If we now run the vm tracer again with this new vm handler being set to index 0x55 we should be able to see a change in LCONSTBZX. In order to facilitate this hook, one must set the virtual address of the new vm handler into a `vm::handler::table_t` object.

```
// change vm handler 0x55 (LCONSTBZX) to our implimentation of it…
auto _meta_data = handler_table.get_meta_data(0x55);
_meta_data.virt = reinterpret_cast<u64>(&__lconstbzx);
handler_table.set_meta_data(0x55, _meta_data);
```

If we run the binary now it will return 1. You can see this below.

```
> TID = 11496, handler idx = 108, decryption key = 0x000000013D369098
> TID = 11496, handler idx = 108, decryption key = 0x000000013D369014
> TID = 11496, handler idx = 195, decryption key = 0x000000013D3690BF
> TID = 11496, handler idx = 219, decryption key = 0x000000013D36908C
> TID = 11496, handler idx = 119, decryption key = 0x000000013D36907B
> TID = 11496, handler idx = 108, decryption key = 0x000000013D369057
> TID = 11496, handler idx = 195, decryption key = 0x000000013D36901C
> TID = 11496, handler idx = 108, decryption key = 0x000000013D369098
> TID = 11496, handler idx = 48, decryption key = 0x000000013D3690B8
result = 1
> finished vm trace...
```

Encoding Virtual Instructions - Inverse Transformations

Since VMProtect 2 generates a virtual machine which executes virtual instructions encoded in its own bytecode one could run their own virtual instructions on the VM if they can encode them. The encoded virtual instructions must also be within a 4gb address space range though as the RVA to the virtual instructions is 32bits wide. In this section I will encode a very simple set of virtual instructions to add two QWORD values together and return the result.

To begin, encoding virtual instructions requires that the vm handlers for said virtual instructions are inside of the binary. Locating these vm handlers is done by ‘vmprofiler’. The vm handler index is the first opcode and the immediate value, if any, is the second. Combining these two sets of operands will yield an encoded virtual instruction. This is the first stage of assembling virtual instructions, the second is encrypting the operands.

Once we have our encoded virtual instructions we can now encrypt them using the inverse operations of vm handler transformations as well as the inverse operations for calc_jmp. It’s important to note that the way in which VIP advances must be taken into consideration when encrypting as the order of operands and virtual instructions depends on this advancement direction.

```
LCONSTBZX 0xA
LCONSTBZX 0xA
ADDQ
SREGQ 0x30
SREGQ 0x80
VMEXIT
```

```
> 0x00007FF7412D5A34 movzx rax, al
> 0x00007FF7412D5A41 mov rdx, [r12+rax*8]
> 0x00007FF7412D5A49 xor rdx, 0x7F3D2149
> 0x00007FF7412D5507 inc rsi
> 0x00007FF7412D7951 add rdx, r13
> 0x00007FF7412D7954 jmp rdx
> vinstr name = LCONSTBZX, has imm = 1, imm = 0x000000000000000A
> vinstr name = LCONSTBZX, has imm = 1, imm = 0x000000000000000A
> vinstr name = ADDQ, has imm = 0, imm = 0x0000000000000000
> vinstr name = SREGQ, has imm = 1, imm = 0x0000000000000030
> vinstr name = SREGQ, has imm = 1, imm = 0x0000000000000080
> vinstr name = VMEXIT, has imm = 0, imm = 0x0000000000000000
[+] finished encoding... encoded instructions below...
> 0x1c - 0xa
> 0x1c - 0xa
> 0x1
> 0x9a - 0x30
> 0x9a - 0x80
> 0x30
> decrypt key = 0x000000014000B000
> decrypt key = 0x000000014000B012
> decrypt key = 0x000000014000B004
> decrypt key = 0x000000014000B005
> decrypt key = 0x000000014000B0CF
> decrypt key = 0x000000014000B0D5
[+] finished encrypting... encrypted instructions below...
> virtual instructions must be allocated at = 0x000000014000B000
> 0x34 0x11 0x26 0x1f 0x5 0xab 0x12 0x61 0xd7 0x62
```

In order to execute these newly assembled virtual instructions, one must put the virtual instructions within a 32bit address range of the vm_entry routine, then put the encrypted rva to these virtual instructions onto the stack, and lastly call into vm_entry. I would suggest using VirtualAllocEx to allocate a RW page directly below the protected module. An example for running virtual instructions is displayed below.

```
SIZE_T bytes_copied;
STARTUPINFOA info = { sizeof info };
PROCESS_INFORMATION proc_info;

// start the protected binary suspended...
// keep in mind this binary is not packed...
CreateProcessA("vmptest.vmp.exe", nullptr, nullptr,
    nullptr, false,
    CREATE_SUSPENDED | CREATE_NEW_CONSOLE,
    nullptr, nullptr, &info, &proc_info);

// wait for the system to finish setting up...
WaitForInputIdle(proc_info.hProcess, INFINITE);
auto module_base = get_process_base(proc_info.hProcess);

// allocate space for the virtual instructions below the module...
auto virt_instrs = VirtualAllocEx(proc_info.hProcess,
    module_base + vmasm->header->offset,
    vmasm->header->size,
    MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

// write the virtual instructions...
WriteProcessMemory(proc_info.hProcess, virt_instrs,
    vmasm->data, vmasm->header->size, &bytes_copied);

// create a thread to run the virtual instructions...
auto thandle = CreateRemoteThread(proc_info.hProcess,
    nullptr, 0u,
    module_base + vm_entry_rva,
    nullptr, CREATE_SUSPENDED, &tid);

CONTEXT thread_ctx;
GetThreadContext(thandle, &thread_ctx);

// sub rsp, 8...
thread_ctx.Rsp -= 8;
thread_ctx.Rip = module_base + vm_entry_rva;

// write encrypted rva onto the stack...
WriteProcessMemory(proc_info.hProcess, thread_ctx.Rsp,
    &vmasm->header->encrypted_rva,
    sizeof vmasm->header->encrypted_rva, &bytes_copied);

// update thread context and resume execution...
SetThreadContext(thandle, &thread_ctx);
ResumeThread(thandle);
```

Conclusion - Static Analysis, Dynamic Analysis

To conclude, my dynamic analysis solution is not the most ideal solution, however It should allow for basic reverse engineering of protected binaries. With more time static analysis of virtual instructions will become possible, however for the time being dynamic analysis will have to do. In the future I will be using unicorn to emulate the virtual machine handlers.

Although I have documented a handful of virtual instructions there are many more that I have not documented. The goal of documenting the virtual instructions that I have is to allow the reader of this article to obtain a feel for how vm handlers should look as well as how one could alter the results of these vm handlers. The documented virtual instructions in this article are also the most common ones. These virtual instructions will most likely be inside of every virtual machine.

I have added a handful of reference builds inside of the repository for you to try your hand at making them return 1 by altering vm handlers. There is also a build which uses multiple virtual machines in a single binary.

Lastly, I would like to restate that this research has most definitely already been done by private entities, and I am not the first to document some of the virtual machine architecture discussed in this post. I have credited those whom I have studied the research of already, however there are probably many more people that have done research on VMProtect 2 that I have not listed simply because I have not come across their work.