Genaytyk-VM by Fare9

COSEC

UC3M COMPUTER SECURITY LAB

Introduction

This is a document explaining the devirtualized of Genaytyk-VM, a reversing challenge which is aimed to calculated a correct serial given a name, this will be a long technical document in order to remember how a virtual machine is resolved, and how python can be used to extract the code from a specific bytecode.

CrackMeStartMountGUI method (starting the code)

The start of the binary is just an exported function which starts the *graphic user interface* of the crackme, using common **user32.dll** functions as: *LoadIcon, LoadCursor, LoadBitmap, CreateWindowExA, UpdateWindow and GetMessage.*

```
void __noreturn CrackMeStartMountGUI()
{
  hInstance_modulehandle = GetModuleHandleA(0);
  window_class.cbSize = 48;
  window_class.style = 11;
  window_class.lpfnWndProc = (WNDPROC)virtualMachine;
  window_class.cbClsExtra = 0;
  window_class.cbWndExtra = 0;
  window_class.hInstance = hInstance_modulehandle;
  window_class.hbrBackground = (HBRUSH)1;
  window_class.lpszMenuName = 0;
  window_class.lpszClassName = "VM-kgme";
  icon_handler = LoadIconA(hInstance_modulehandle, (LPCSTR)0x64);
  window_class.hIcon = (HICON)icon_handler;
  window_class.hIconSm = (HICON)icon_handler;
  hcursor = LoadCursorA(0, (LPCSTR)0x7F00);
  window_class.hCursor = (HCURSOR)hcursor;
  bitmap = LoadBitmapA(hInstance_modulehandle, (LPCSTR)0x65);
  window_class.hbrBackground = CreatePatternBrush(bitmap);
  RegisterClassExA(&window_class);
  cxscreen_adjusted = (unsigned int)(GetSystemMetrics(SM_CXSCREEN) - 204) >> 1;
  cyscreen_adjusted = (unsigned int)(GetSystemMetrics(SM_CYSCREEN) - 300) >> 1;
  window_hwnd = CreateWindowExA(
                  0,
                  "VM-kgme",
                  "VM keygenme by Genaytyk",
                  0x800A0000,
                  cxscreen_adjusted,
                  cyscreen_adjusted,
                  204,
                  300,
                  0,
                  0,
                  hInstance_modulehandle,
                  (LPVOID)1);
  UpdateWindow(window_hwnd);
  while ( GetMessageA(&msg_structure, 0, 0, 0) )
  {
    TranslateMessage(&msg_structure);
    DispatchMessageA(&msg_structure);
  }
  ExitProcess(msg_structure.wParam);
}
```

As the *pointer of WndProc* used in the **WNDCLASSEXA** structure, we have the manager of the *user interface*.

virtualMachine method (User interface manager)

This code just manages the starting of all the GUI components, one of the messages managed is **WM_CREATE**

```
if ( Msg == WM_CREATE )
{
  font = CreateFontA(10, 5, 0, 0, 400, 0, 0, 0, 1u, 0, 0, 2u, 0, 0);
  window_name = CreateWindowExA(0, "EDIT", 0, 0x50010080u, 32, 211, 123, 9, hWndParent, 0, hInstance_modulehandle, 0);
  SendMessageA(window_name, 0x30u, (WPARAM)font, 0);
  window_serial = CreateWindowExA(
                    0,
                    "EDIT",
                    0,
                    0x50010080u,
                    32,
                    226,
                    123,
                    9,
                    hWndParent,
                    0,
                    hInstance_modulehandle,
                    0);
  SendMessageA(window_serial, 0x30u, (WPARAM)font, 0);
  find_resource = FindResourceA(hInstance_modulehandle, (LPCSTR)0x66, "REGION");
  loaded_resource = LoadResource(hInstance_modulehandle, find_resource);
  locked_resource = (RGNDATA *)LockResource(loaded_resource);
  size_of_resource = SizeofResource(hInstance_modulehandle, find_resource);
  window_region = ExtCreateRegion(0, size_of_resource, locked_resource);
  SetWindowRgn(hWndParent, window_region, 1);
  SetWindowTextA(hWndParent, "VM keygenme by Genaytyk");
  SetWindowTextA(window_name, "NAME");
  SetWindowTextA(window_serial, "SERIAL");
  ShowWindow(hWndParent, 1);
  logical_brush = (int)CreateSolidBrush(0xFFFFFFu);
  return DefWindowProcA(hWndParent, Msg, (WPARAM)hdc, lParam);
```

We have code to show a help message in case one of the button is pressed (**WM_LBUTTONDOWN**):

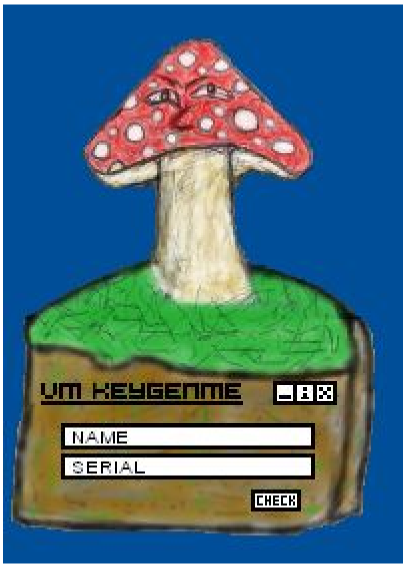
```
else
{
  MessageBoxA(
    hWndParent,
    "VM keygenme (VM Crackme n¦1)\r\n"
    "by Genaytyk\r\n"
    "\r\n"
    "The VM series crackmes are based on a Virtual Machine engine\r\n"
    "coded in pure assembly language. These crackmes are not suited\r\n"
    "for beginners at all.\r\n"
    "\r\n"
    "The first crackme of VM series, called VM keygenme uses\r\n"
    "VM engine only for a serial check. You will have to understand\r\n"
    "p-code, make analysis of algorithm and code a keygen.\r\n"
    "Of course patching isn't allowed...\r\n"
    "\r\n"
    "Good luck and have fun!!\r\n"
    "Please send your keygens (and tuts :) to :\r\n"
    "\r\n"
    "genaytyk@hotmail.com",
    "VM keygenme by Genaytyk",
    0x40u);
}
return DefWindowProcA(hWndParent, Msg, (WPARAM)hdc, lParam);
```

Finally the code that we are interested in:

```
if ( name_length )
{
  if ( name_length >= 0x23 )
  {
    error_message = "The name is too long";
  }
  else
  {
    if ( name_length >= 3 )
    {
      nameLength = name_length;
      serial_length = GetWindowTextA(window_serial, "", 36);
      if ( serial_length )
      {
        serialLength = serial_length;
        CrackMeStartTheVirtualMachineConfigs((int)&VM_CONFIGURATION_VALUES);
        if ( (_BYTE)finalComparation == 1 )
        {
          MessageBoxA(
            hWndParent,
            "Congratulations, you did it!!\n\nTry to code a keygen and send your work at\r\ngenaytyk@hotmail.com",
            "VM keygenme by Genaytyk",
            0x40u);
          return DefWindowProcA(hWndParent, Msg, (WPARAM)hdc, lParam);
        }
        error_message = "Nope, try again ;)";
        goto _showMessageBox;
      }
      goto _textNotGiven;
    }
    error_message = "The name is too short";
  }
owMessageBox:
  MessageBoxA(hWndParent, error_message, "VM keygenme by Genaytyk", 0x30u);
  return DefWindowProcA(hWndParent, Msg, (WPARAM)hdc, lParam);
}
tNotGiven:
error_message = "Please enter a name and a serial";
goto _showMessageBox;
```

Some of the constraints for the name is that must be greater than 2 bytes, and lower than 35. If a serial is given to the GUI, the real VM is started given a configuration.

This is the GUI:



CrackMeStartTheVirtualMachineConfigs (Start of the Virtual Machine)

As we saw, the virtual machine receives a configuration as parameter, the configuration is the next one:

```
VM_CONFIGURATION_VALUES dd offset VM_CODE
                                              ;
                        dd 7ACh
                        dd 403C6Ah
                        dd 113h
                        dd 0
                        db    0
                        db    0
```

For the virtual machine, I've created different structures and different enumerations, for the configuration values the next structure is used:

```
VM_CONFIG          struc ; (siz
VM_OEP             dd ?

VM_SIZE_OF_CODE dd ?
VM_HARDCODEDSTRING dd ?
VM_MAX_SIZE_OF_SERIAL dd ?
VM_STACK_SIZE      dd ?
VM_BYTE1           db ?
VM_BYTE2           db ?

VM_CONFIG          ends
```

Also I've created a structure to follow the state of the virtual machine with registers and a stack:

```
VM_LOGIC        struc ; (sizeof=0x13C,

VM_EIP          dd ?

REG0x4          dd ?
REG0x8          dd ?
REG0xC          dd ?
REG0x10         dd ?
VM_ESP          dd ?

VM_EBP          dd ?
REG0x1C         dd ?
REG0x20         dd ?
REG0x24         dd ?

REG0x28         dd ?
REG0x2C         dd ?
REG0x30         dd ?
REG0x34         dd ?
REG0x38         dd ?
VM_STACK        db 256 dup(?)
VM_LOGIC        ends
```

Let’s start digging in the code, with the first instructions, the VM starts with a setup code to get the fields from the configuration and writing them into different variables:

We can follow it in the disassembler, for having a pseudo-C code:

```
VM_CONFIG = (VM_CONFIG *)arg[2];
pointerToVM_Code = (void *)VM_CONFIG->VM_OEP;
if ( !pointerToVM_Code )
  pointerToVM_Code = arg[1];                   off=0; int
sizeOfCode = VM_CONFIG->VM_SIZE_OF_CODE;
pointerToHardcodedString = (void *)VM_CONFIG->VM_HARDCODEDSTRING;
sizeOfSerial = VM_CONFIG->VM_MAX_SIZE_OF_SERIAL;
stackSize = VM_CONFIG->VM_STACK_SIZE;
if ( !stackSize )
  stackSize = 256;
maybeDebugByte = VM_CONFIG->VM_BYTE1;
byte2 = VM_CONFIG->VM_BYTE2;
```

After this, the memory for the virtual machine is allocated, this will be used for almost everything like in a real architecture, the program that executes has a memory that can use to store data, this memory is later separated into different segments as stack, data, and so on. The next code allocates enough memory, points to the top of the stack (stack goes from higher memory to lower memory), set VM_ESP and VM_EBP, allocates space for one variable, and stores the real return address in that stack:

```
0040184E        mov     edx, 3Ch ; 3Ch = size of registers
00401853        add     edx, stackSize
00401859        push    40h ; flProtect
0040185B        push    1000h ; flAllocationType
00401860        push    edx ; dwSize
00401861        push    0 ; lpAddress
00401863        call    VirtualAlloc ; reserve memory of stack + size of registers
00401868        mov     vm_memory, eax ; store virtual machine memory in the variable
0040186D        mov     edi, vm_memory ; load memory in edi
00401873        mov     eax, pointerToVM_Code ; load pointer to code in eax
00401878        mov     [edi+VM_LOGIC.VM_EIP], eax ; VM_EIP now point to code of vm
0040187A        mov     eax, stackSize ; load top of the stack in eax
0040187F        sub     eax, 4 ; increment stack in 4
00401882        mov     [edi+VM_LOGIC.VM_ESP], eax ; at the beginning ESP and EBP
00401882                ; points to same place on stack
00401885        mov     [edi+VM_LOGIC.VM_EBP], eax
00401888        sub     [edi+VM_LOGIC.VM_ESP], 4 ; allocate space on stack for a value
0040188C        mov     eax, [edi+VM_LOGIC.VM_ESP] ; load vm_esp in eax
0040188F        add     eax, vm_memory ; points to vm_esp in virtual memory stack
00401895        mov     edx, ds:[ebp+4] ; Save the real return address inside stack of VM
00401899        mov     [eax], edx
```

As we can see, the lower part of the allocated memory is used to store the structure of the virtual machine (the registers), so *edi* points to that memory, and after that is used to access that memory in different offsets depending on accessing a register, or accessing the stack, we can check it in the decompiler:

```
vm_memory = (VM_LOGIC *)VirtualAlloc(0, stackSize + 60, 0x1000u, 0x40u);// allocates memory for the virtual machine
vm_memory->VM_EIP = (int)pointerToVM_Code;     // makes VM_EIP point to the code
vm_memory->VM_ESP = stackSize - 4;             // set ESP and EBP as pointers to end of memory
                                               // (where stack will be located)
vm_memory->VM_EBP = stackSize - 4;
vm_memory->VM_ESP -= 4;                        // allocate space for new variable on stack
*(int *)((char *)&vm_memory->VM_EIP + vm_memory->VM_ESP) = (int)arg[1];// save real return address on top of stack
```

The last sentence should be something like:

vm_memory[vm_memory->VM_ESP] = return_address;

But as VM_EIP is the first offset, IDA takes it as the address of that offset.

Let's continue, now we will have the loop of the virtual machine, the loop just goes over the bytecodes of the instruction set recognizing them while parsing and executing things. To improve visualization I created an enum with all the opcodes:

```
; enum VM_OPCODES, mappedto_30
MOVE_OPCODE        = 1
ADD_OPCODE         = 2
SUB_OPCODE         = 3
IMUL_OPCODE        = 4
IDIV_OPCODE        = 5
OR_OPCODE          = 6
XOR_OPCODE         = 7
AND_OPCODE         = 8
INC_OPCODE         = 9
DEC_OPCODE         = 0Ah
NOT_OPCODE         = 0Bh
SHR_OPCODE         = 0Ch
SHL_OPCODE         = 0Dh
ROR_OPCODE         = 0Eh
ROL_OPCODE         = 0Fh
JMP_OPCODE         = 10h
JZ_OPCODE          = 11h
JNZ_OPCODE         = 12h
JA_OPCODE          = 13h
JB_OPCODE          = 14h
JNB_OPCODE         = 15h
CALL_OPCODE        = 17h
PUSH_OPCODE        = 18h
POP_OPCODE         = 19h
RET_OPCODE         = 1Ah
NOP_OPCODE         = 1Bh
ERROR_OPCODE       = 1Eh
IMMEDIATE_OPCODE    = 49h
SERIAL_HASH_OPCODE   = 4Fh
ADDRESS_OPCODE     = 51h
REGISTER_OPCODE    = 52h
```

The first thing to do, is check that the VM_EIP is inside of the code bound, in other case, there would be an error and the VM should leave:

```
0040189B Jump to xref
0040189B _getAddressOfInstructionAndCheckBounds: ;
0040189B                     mov     edi, vm_memory ; edi points to virtual machine memory
004018A1                     mov     esi, [edi+VM_LOGIC.VM_EIP] ; retrieve EIP value
004018A3                     mov     eax, pointerToVM_Code ; eax point to start of vm code
004018A8                     cmp     esi, eax
004018AA                     jb      short _eipNotCorrect ; if EIP is below of virtual machine code fuck off

004018AC                     add     eax, sizeOfCode ; eax points to end of the code
004018B2                     cmp     esi, eax ; check if pointer to code is out of bound
004018B4                     ja      short _eipNotCorrect ; in that case, fuck off...
```

We can go again to the disassembler to see this in case we don't see it in the assembly:

```
while ( 1 )
{
  eip_value = (char *)vm_memory->VM_EIP;
  if ( vm_memory->VM_EIP < (unsigned int)vm_code )
    break;
  end_of_vm_code = (char *)vm_code + sizeOfCode;
  if ( eip_value > (char *)vm_code + sizeOfCode )
    break;
```

If the EIP value is not okay, the next is executed:

```
  end_of_vm_code = (byte *)-1;
_checkOEPIsCorrect:
  v6 = (VM_CONFIG *)arg[2];
  if ( !v6->VM_OEP )
    arg[1] = (DWORD *)((char *)vm_code + sizeOfCode);// load return address from vm stack
  if ( end_of_vm_code == (byte *)-1 )
    end_of_vm_code = (byte *)MessageBoxA(
                               0,
                               "An exception has stopped execution of virtual machine",
                               "VM by Genaytyk",
                               0x10u);
```

In order to improve readability of this technical text, I will try to categorize also this part, giving a title to each part.

Search of Instruction of NOP and RET

The first opcode that the VM checks, is the ERROR_OPCODE, as this is not really important (more than it means that an error occurred), I just show the decompiled code:

```
if ( *eip_value == ERROR_OPCODE )
  goto _checkOEPIsCorrect;
```

Execution of NOP

The next opcode it is the NOP_OPCODE, in the architectures this opcode represents an instruction that literally does nothing (commonly could be something like “mov eax, eax”), it only make the processor to execute 1 cycle:

```
004018BC                cmp     al, NOP_OPCODE ; check if opcode is equals to 1Bh, if it is the same
004018BC                        ; increment VM_EIP in 1, it looks like NOP
004018BE                jnz     short _checkRETOpcode ; looking the left tree, we could image
004018BE                        ; this is a RET
```

```
004018C0
004018C0 _executes_nop:
004018C0                 add     [edi+VM_LOGIC.VM_EIP], 1
004018C3                 jmp     short _getAddressOfInstructionAndCheckBounds ;
004018C3                         ; edi points to virtual machine memory
```

We can see it also in the decompiler:

```
if ( (_BYTE)opcode_from_code == NOP_OPCODE )
{
  ++vm_memory->VM_EIP;
}
```

The only thing it does is advance VM_EIP by one, so a NOP operation xD.

Execution of RET

Next one, is an important one for control flow, it is the RET_OPCODE, this one extracts from the stack a stored address (the address of the first instruction after a "*call*" instruction) and later the address is stored in VM_EIP, once that is done, the stack is unwind of that address:

```
004018C5
004018C5 _checkRETOpcode:        ; looking the left tree, we could image
004018C5                 cmp     al, RET_OPCODE ; this is a RET
004018C7                 jnz     short _checkOpcode
```

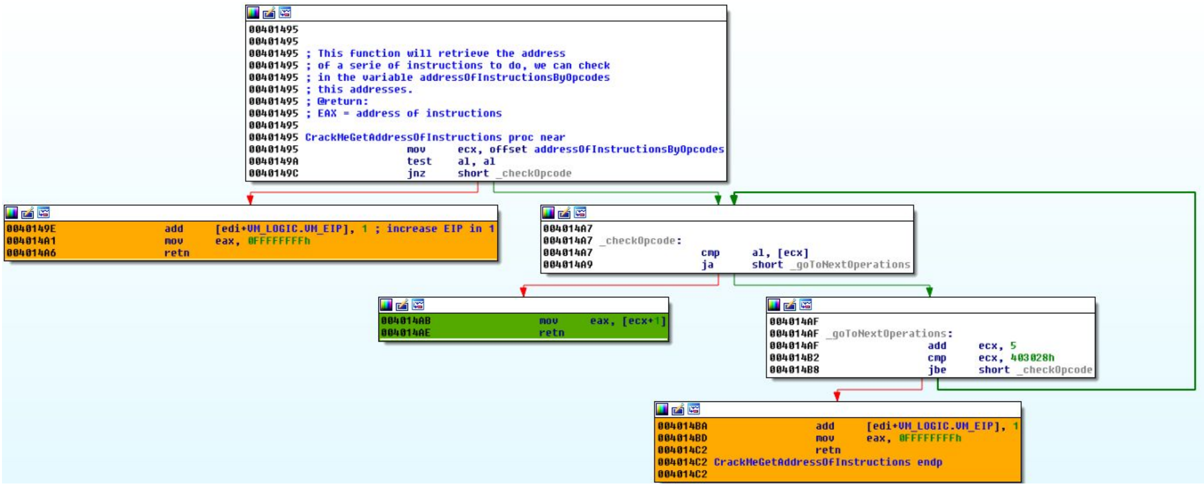
```
004018C9                 mov     eax, [edi+VM_LOGIC.VM_ESP]
004018CC                 add     eax, 3Ch
004018CF                 add     eax, vm_memory ; point to top of the stack
004018D5                 mov     eax, [eax] ; get the value in EAX
004018D7                 mov     [edi+VM_LOGIC.VM_EIP], eax ; set the value in VM_EIP
004018D9                 add     [edi+VM_LOGIC.VM_ESP], 4 ; Now increment the stack
004018DD                 jmp     short _getAddressOfInstructionAndCheckBounds ;
004018DD                         ; edi points to virtual machine memory
```

So, as we can see that's how it works in this VM the RET instruction. We can check it again in the decompiler to see it better :D

```
else if ( (_BYTE)opcode_from_code == RET_OPCODE )
{
  vm_memory->VM_EIP = *(_DWORD *)&vm_memory->VM_STACK[vm_memory->VM_ESP];
  vm_memory->VM_ESP += 4;
}
```

Search of Arithmetic, Logic, Moves, Jumps, Call, Push, Pop, Pushad and Popad

The next checks will be for arithmetic, logic operations, moves, jumps and so on. The check is done joining different operations, and selected with the next method.



It points to an address with an specific structure ("addressOfInstructionByOpcodes") in ecx, and then it goes checking the opcode (in "al"), with the byte value stored in the address, if the value in "al" is greater than the stored in "ecx" this register moves 5 bytes forward and checks again, it will go with this checks until it finds a proper value or until there's no value so the search is incorrect (and "ecx" get the address 0x403028). We can see the decompiled version:

```
signed int __usercall CrackMeGetAddressOfInstructions@<eax>(BYTE instruction_opcode@<al>, VM_LOGIC *vm_logic@<edi>)
{
  BYTE *pointer_to_instructions; // ecx@1
  signed int result; // eax@2

  pointer_to_instructions = &addressOfInstructionsByOpcodes;
  if ( instruction_opcode )
  {
    do
    {
      if ( instruction_opcode <= *pointer_to_instructions )
        return *(_DWORD *)(pointer_to_instructions + 1);
      pointer_to_instructions += 5;
    }
    while ( (unsigned int)pointer_to_instructions <= 0x403028 );
    ++vm_logic->VM_EIP;
    result = -1;
  }
  else
  {
    ++vm_logic->VM_EIP;
    result = -1;
  }
  return result;
}
```

And here we can see the structure with the "limit" opcode, and the addresses to the operations:

```
00403000 addressOfInstructionsByOpcodes db 8      ; DATA XREF: CrackMeGetAddressOfInstructions↑o
00403001                         dd offset MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND_OPERATIONS
00403005                         db  0Bh
00403006                         dd offset INC_DEC_NOT_OPERATIONS
0040300A                         db  0Fh
0040300B                         dd offset SHIFT_ROR_ROL_OPERATIONS
0040300F                         db  10h
00403010                         dd offset JMP_OPERATION
00403014                         db  16h
00403015                         dd offset JZ_JNZ_JA_JB_JNB_JBE_OPERATIONS
00403019                         db  17h
0040301A                         dd offset CALL_RET_OPERATION
0040301E                         db  19h
0040301F                         dd offset PUSH_POP_OPERATION
00403023                         db  1Dh
00403024                         dd offset PUSAD_OPERATION
00403028                         db 0
```

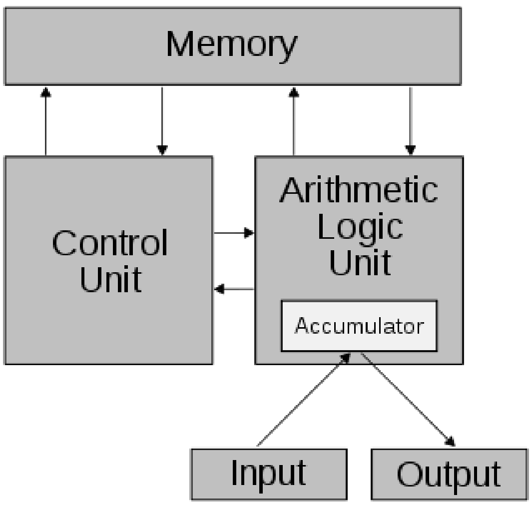
If we go to the enumeration of the OPCODE from the VM, we can get for example "ADD" that is 2, the loop will check: "is 2 lower than 8?" in affirmative keys it will retrieve the *"MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND_OPERATIONS"* address and it will return it in order to execute it.

As some instructions are pretty similar are joined in the address, because they commonly use the same set of operands (for example 2 registers, 1 register - 1 immediate value, etc).

Once an address is chosen it's executed, to do that it will decide which operands to use, we will see how that is done.

```
004018E9                 call    eax ; execution of the instruction
004018EB                 cmp     eax, 0FFFFFFFFh
004018EE                 jz      short _errorSearching
```

The code just executes a call to the instruction, this would be the signal to a real process for the execution of an instruction, and the previous part would be the unit control, together with the next sections:



Execution of MOV, ADD, SUB, IMUL, IDIV, OR, XOR and AND (Part 1)

So as we said, once the function return an address, this is executed with a simple “call eax”:

```
call    CrackMeGetAddressOfInstructions ; of a serie of instructions to do, we can check
        ; in the variable addressOfInstructionsByOpcodes
        ; this addresses.
        ; @return:
        ; EAX = address of instructions
cmp     eax, 0FFFFFFFFh
jz      short _errorSearching

004018E9                call    eax
004018EB                cmp     eax, 0FFFFFFFFh
004018EE                jz      short _errorSearching
```

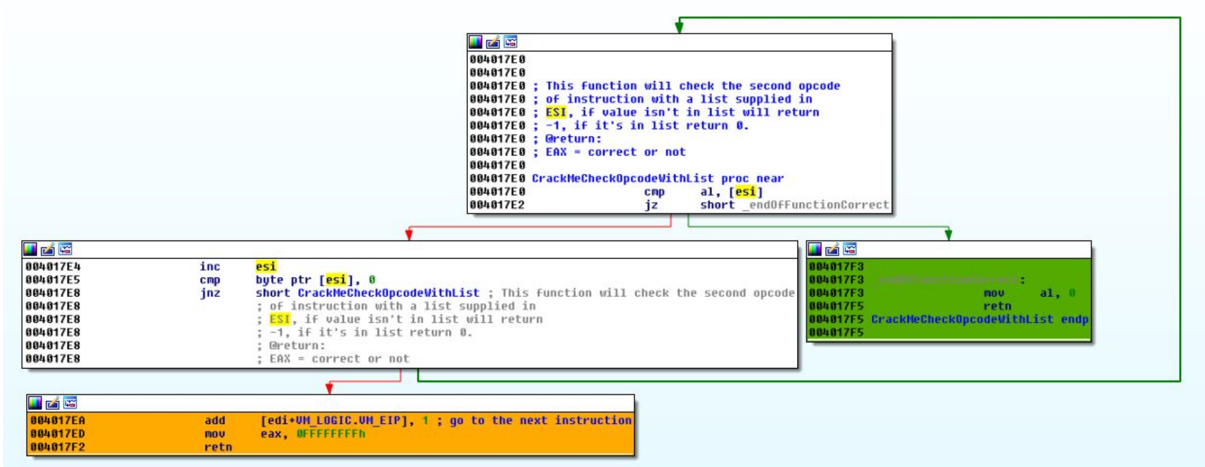
And we would jump to the next code:

```
:00401000 MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND_OPERAtIONS: ; DATA XREF: .rsrc:00403001↓o
:00401000                 mov     esi, [edi+VM_LOGIC.VM_EIP]
:00401002                 mov     al, [esi+1]
:00401005                 mov     esi, offset MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND_SECOND_OPCODE
:0040100A                 call    CrackMeCheckOpcodeWithList ; This function will check the second opcode
:0040100A                                         ; of instruction with a list supplied in
:0040100A                                         ; ESI, if value isn't in list will return
:0040100A                                         ; -1, if it's in list return 0.
:0040100A                                         ; @return:
:0040100A                                         ; EAX = correct or not
:0040100F                 cmp     eax, 0FFFFFFFFh
:00401012                 jnz     short PREPARING_MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND
:00401014                 retn
```

“esi” will point to the offset where the instruction opcode is, after that the next opcode (*[esi + 1])* is loaded in “al”, so it contains the index of the structure type for the operands, for the instructions MOV, ADD, SUB, IMUL, IDIV, OR, XOR and AND the index for the structures are:

```
MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND_SECOND_OPCODE db 6
                                        ; DATA XREF: .rsrc:00401005↑o
                db    7
                db    8
                db    9
                db  0Ah
                db  0Bh
                db  0Ch
                db  0Dh
                db  0Eh
                db  0Fh
                db  10h
                db  11h
                db  12h
                db  13h
                db    0
```

The next called function, the only thing it does is just to check that the obtained opcode is inside of the previous list:



Or the decompiled version:

```
// EAX = correct or not
char __usercall CrackMeCheckOpcodeWithList@<al>(char operand_value@<
{
  while ( operand_value != *operands_index_values )
  {
    if ( !*++operands_index_values )
    {
      ++vm_logic->VM_EIP;
      return -1;
    }
  }
  return 0;
}
```

So now we have the instruction opcode, and we have an index to a field in a structure that tells us the operands of the instruction, let's continue:

```
00401015
00401015 PREPARING_MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND: ; CODE XREF: .rsrc:00401012↑j
00401015                 call    CrackMeGetValuesForInstruction
0040101A                 cmp     edi, 0FFFFFFFFh
0040101D                 jnz     short MOVE_DWORD
0040101F                 mov     eax, 0FFFFFFFFh
00401024                 mov     edi, vm_memory
0040102A                 inc     [edi+VM_LOGIC.VM_EIP]
0040102C                 retn
```

This part is really, really important, because it takes the operands of the instruction and this process is a little bit confusing because involves various structures, depending on the kind of operands, so I will try to explain it in assembly step by step:

Finding operands to the instructions

```
004014C3 CrackMeGetValuesForInstruction proc near
004014C3                 mov     dword ptr unk_403DAA, 0
004014CD                 mov     dword ptr numberOfOperands, 0
004014D7                 mov     esi, [edi+VM_LOGIC.VM_EIP]
004014D9                 call    CrackMeGetOperandStructAndSizeOfInstruction ; This function will do the algorithm
004014D9                         ; to get the struct from the operand (where
004014D9                         ; VM will get type of operand and other values)
004014D9                         ; and the size of instruction.
004014D9                         ; @return:
004014D9                         ; EAX = address of the operand struct
004014D9                         ; EBX = size of instruction
```

The first thing it does *CrackMeGetValuesForInstruction* is to call *CrackMeGetOperandStructAndSizeOfInstruction* this function is really important as it calculates the address and the size of that structure, in *esi* again we have the address to the instruction opcode:

```
0040179B CrackMeGetOperandStructAndSizeOfInstruction proc near
0040179B                 mov     bl, [esi+1]
0040179E                 mov     ecx, offset operandStruct ; point to struct that has 3 values
0040179E                         ; 1 = number of operand struct of same instruction
0040179E                         ; 2 = size of the operand in bytes (example a register 1 byte)
0040179E                         ; 3 = type of operand
004017A3                 xor     eax, eax ; here bl = index of the structure
004017A3                         ; ecx = pointer to operandStruct
```

The first thing we have here, is that, from *esi + 1* is extracted the index to the structure field *operandStruct* (this byte was checked previously, if it was inside of certain bound of numbers). The operandStruct is extracted also in the disassembler: https://github.com/Fare9/Genaytyk-VM/blob/master/vm_disassembler.py#L96

For example *[1,3,0x49]* this structure is based in 3 components, the first one says how many structures follow it, the second one is the number of bytes to read from the code, and the third one tells about the meaning of those bytes. In this case:

1 → Only one structure
3 → Read 3 bytes from code
0x49 → The read 3 bytes are an immediate value.

Another more complex example: *[2,1,0x51,4,0x49]*.

2 → Read two structures.
1 → Read one byte from code.
0x51 → The read byte, is an index to a structure which represents the registers (The structure is represented in special way in the x86 disassembler: https://github.com/Fare9/Genaytyk-VM/blob/master/vm_disassembler_x86.py#L125)
4 → Read 4 bytes from code
0x49 → Those 4 bytes are an immediate value.

Okay, so this is the way that the operands for the instructions are decided, using the structure, and the index taken from the code (read as *[esi + 1]*) is the index for that structure.

An what it does this function (*CrackMeGetOperandStructAndSizeOfInstruction*) is using that index, go through *operandStruct* to find the correct field:

```
004017A5
004017A5 _getTheAddressOfTheOpcodeStruct: ;
004017A5                 dec     bl ; decrement second operand of instruction
004017A7                 test    bl, bl
004017A9                 jz      short _endOfLoop
```

The first part of the loop is decreasing bl one by one (that's the way the index works), the left part (if bl is not zero) it does the next:

```
004017A9                 jz      short _e

004017AB                 mov     al, [ecx+SIZE_OPERAND_STRUCT.number_of_operand_struct] ; get number of structs that follow this value
004017AD                 inc     ecx ; point to the structure
004017AE                 lea     ecx, [ecx+eax*2] ; go to next structure
004017B1                 cmp     ecx, offset endOfOperandStruct
004017B7                 jnz     short _getTheAddressOfTheOpcodeStruct ;
004017B7                 ; decrement second operand of instruction
```

this left part what it does is to take the first field (number of structures) and jumps over that number of structures with the *lea* instruction (because each field of the structure is 2 bytes:

```
index_in_operand_struct = a1[1];
operand_struct = (SIZE_OPERAND_STRUCT *)&operandStruct;
number_of_operand_struct = 0;
while ( --index_in_operand_struct )
{
  LOBYTE(number_of_operand_struct) = operand_struct->number_of_operand_struct;
  operand_struct = (SIZE_OPERAND_STRUCT *)((char *)operand_struct + 2 * number_of_operand_struct + 1);
  if ( (char *)operand_struct == &endOfOperandStruct )
    return (_BYTE *)-1;
}
```

If the program goes to the right part of the conditional, it means that it has found the correct structure (as the index is 0 now):

```
004017BF
004017BF _endOfLoop:                     ;
004017BF                 mov     eax, ecx ; move to eax the correct operandStruct
004017C1                 mov     operand_struct_selected, eax ; store it inside of a global variable
004017C6                 push    eax
004017C7                 mov     cl, [eax+SIZE_OPERAND_STRUCT.number_of_operand_struct] ; get number of structs of operandStructs
004017C9                 inc     eax ; points to the structures (bytes_to_read, type_of_field)
004017CA                 xor     ebx, ebx
```

So it stores in a global variable the operandStruct structure selected, and it takes its size, finally points with eax to the real structure (where are the bytes to read and the type of field as we saw). After that a loop is executed to get the total size of their fields:

```
004017CC
004017CC _getTheSizeOfOperands:  ;
004017CC                add     bl, [eax+OPERAND_STRUCT.bytes_to_read] ; add size of operands to bl
004017CE                add     eax, 2 ; go to the next operand
004017D1                dec     cl ; check there's no more operands
004017D3                jnz     short _getTheSizeOfOperands ;
004017D3                        ; add size of operands to bl
```

```
004017D5                add     ebx, 2 ; get size of instruction adding 2
004017D8                mov     size_of_instruction, ebx ; store size of instruction
004017DE                pop     eax ; recover pointer to the structure
004017DF                retn
004017DF CrackMeGetOperandStructAndSizeOfInstruction endp
004017DF
```

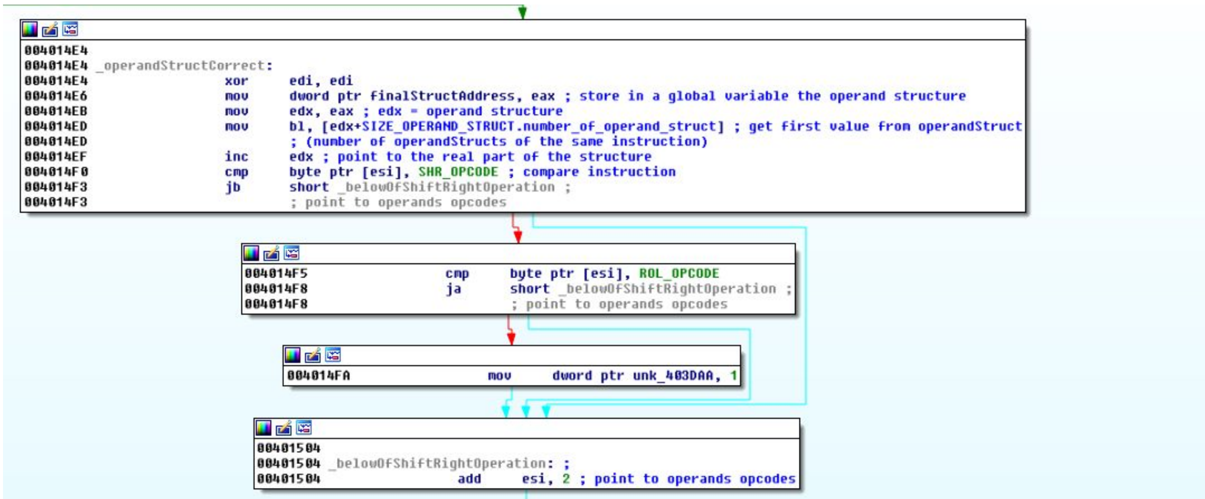
So we have in EAX the pointer to the structure (recovered from the stack), and in EBX the size of the instruction in total. The complete function can be seen in the decompiled code:

```
SIZE_OPERAND_STRUCT *__usercall CrackMeGetOperandStructAndSizeOfInstruction@<eax>(BYTE *a1@<esi>)
{
  BYTE index_in_operand_struct; // bl@1
  SIZE_OPERAND_STRUCT *operand_struct; // ecx@1
  int number_of_operand_struct; // eax@1
  char number_of_operand_struct_; // cl@5
  OPERAND_STRUCT *operand_struct_; // eax@5
  int bytes_to_read_total; // ebx@5

  index_in_operand_struct = a1[1];
  operand_struct = (SIZE_OPERAND_STRUCT *)&operandStruct;
  number_of_operand_struct = 0;
  while ( --index_in_operand_struct )
  {
    LOBYTE(number_of_operand_struct) = operand_struct->number_of_operand_struct;
    operand_struct = (SIZE_OPERAND_STRUCT *)((char *)operand_struct + 2 * number_of_operand_struct + 1);
    if ( (char *)operand_struct == &endOfOperandStruct )
      return (SIZE_OPERAND_STRUCT *)-1;
  }
  operand_struct_selected = operand_struct;
  number_of_operand_struct_ = operand_struct_selected->number_of_operand_struct;
  operand_struct_ = &operand_struct_selected->operand_struct;
  bytes_to_read_total = 0;
  do
  {
    LOBYTE(bytes_to_read_total) = operand_struct_->bytes_to_read + bytes_to_read_total;
    ++operand_struct_;
    --number_of_operand_struct_;
  }
  while ( number_of_operand_struct_ );
  size_of_instruction = bytes_to_read_total + 2;
  return operand_struct_selected;
}
```

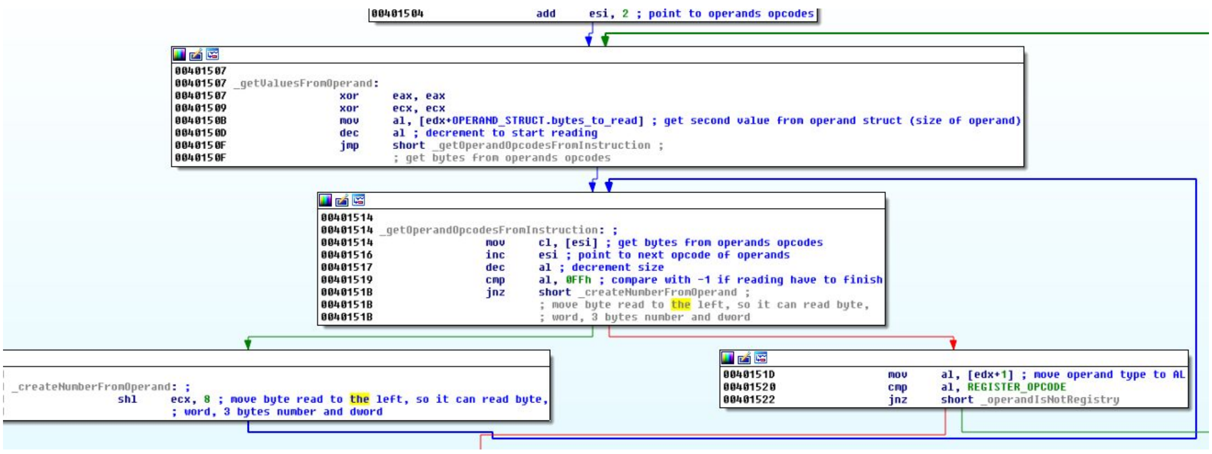
Once we return to *CrackMeGetValuesForInstruction* the code checks if there was an error and the pointer to the structure is not correct:

```
004014D9                        ; EAX = address of the operand struct
004014D9                        ; EBX = size of instruction
004014DE                cmp     eax, 0FFFFFFFFh
004014E1                jnz     short _operandStructCorrect
```

Going to the right path (if previous function was success), it does different checks on the instruction opcode, because different instructions have different operands and behaviours:



If we follow the down path, what it does is, now that esi points in the code to the opcodes of the operands, it will read them one by one (having the size as limit):

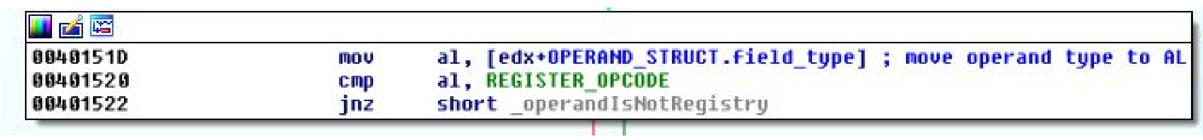


So for example if we had that the size of the operand it was 4, we could have 4 bytes like: 00 11 22 33, those bytes are going to be read in ecx, and ecx finally will be:

ecx = 00112233

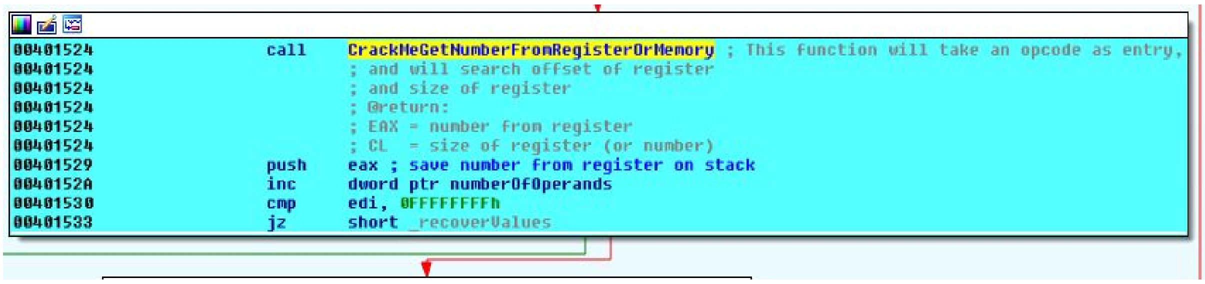
And probably that will be an immediate value.

After that in the down right part, we have that the program reads the operand type, and is compared with a type of operand:



Reading register operand

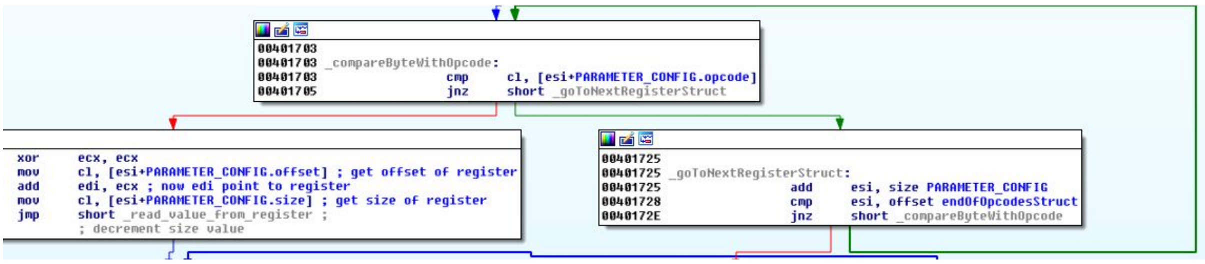
The first check is done against the REGISTER_OPCODE (0x52), so let’s gonna see how the registers are chosen.



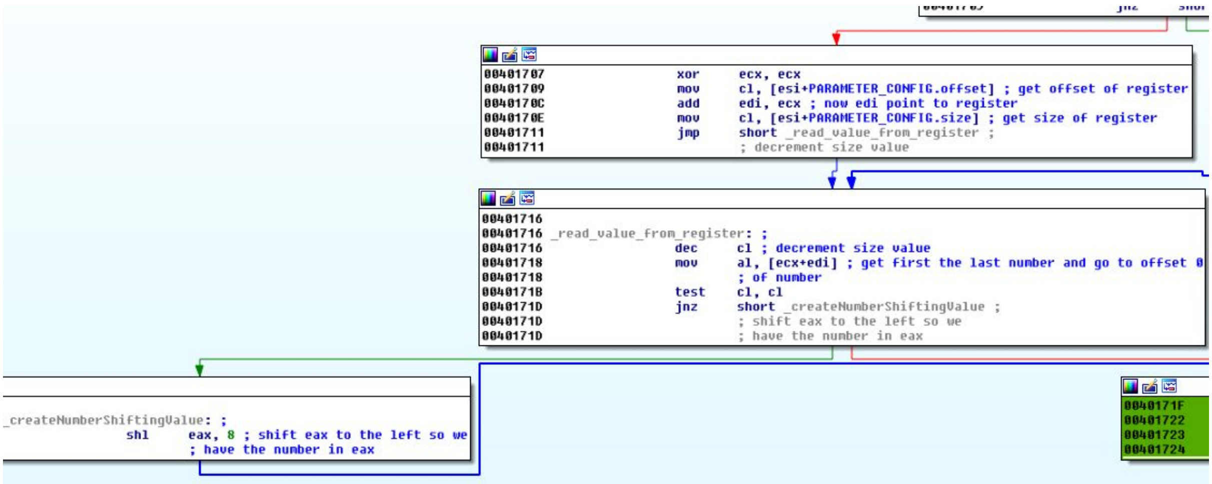
First function it calls is *CrackMeGetNumberFromRegisterOrMemory* , here another structure will be presented so let’s dig into it:

This first part of the code, just makes edi point to the memory of the vm, and esi point to the structure, the structure as I left before can be seen in here: https://github.com/Fare9/Genaytyk-VM/blob/master/vm_disassembler_x86.py#L125 The structure is a 3 field structure in the code: [opcode of the register, offset of the register in vm memory, size of register (1, 2 or 4 bytes)]

The next code it will traverse all the fields in the structure to get the good one:

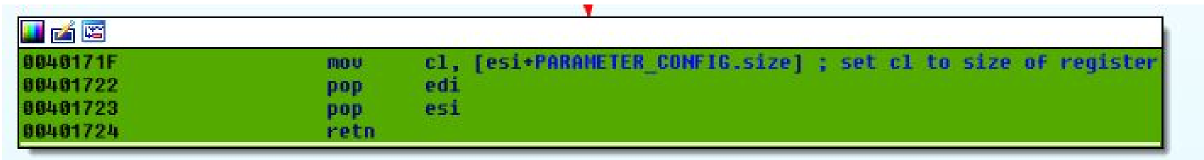


It goes checking the opcode with the one read from the code, and if it’s not the correct one, it adds 3 to esi, as the structure size is 3 bytes. Now we will go through the left size, that is the correct structure:



This code points to the register in vm_memory, and then moves to the end of that registers, for reading from back to front the value of the register in *eax*.

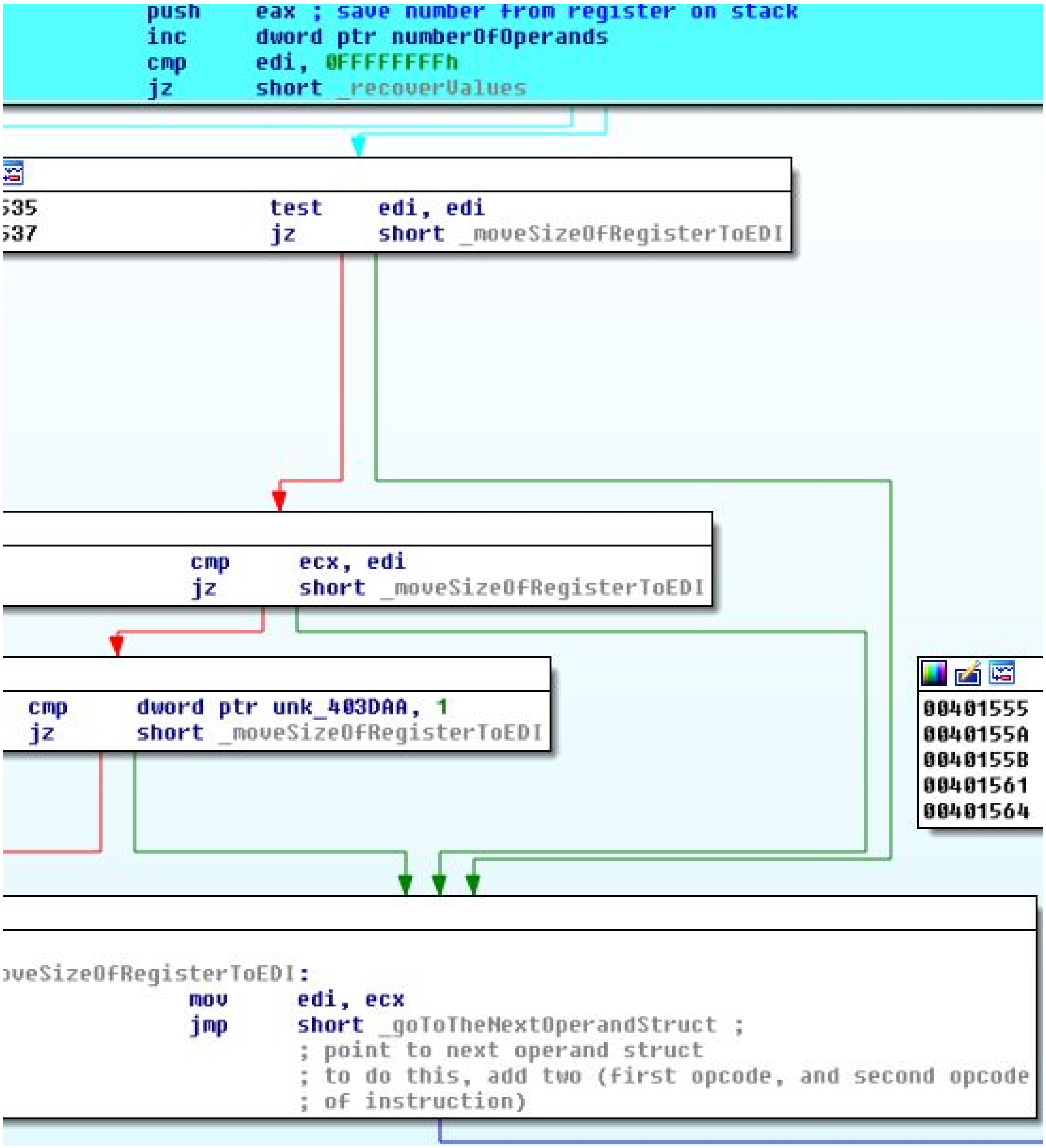
Last thing to do is just moving the size to *cl*, and we will have in *eax* already the register value.



We can see the decompiled version, of the function to read a register value:

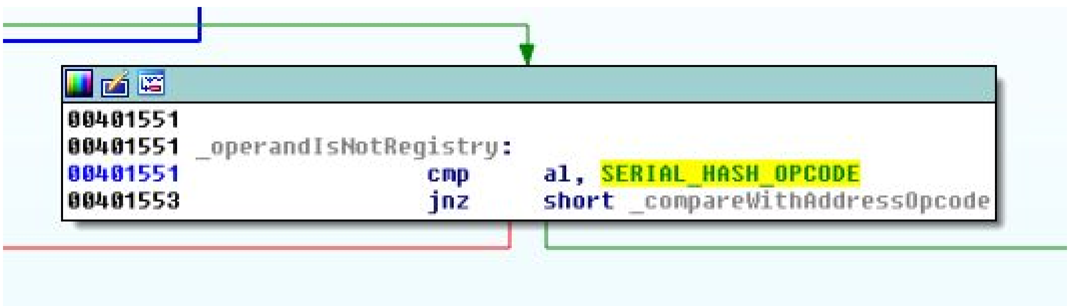
```
// This function will take an opcode as entry,
// and will search offset of register
// and size of register
// @return:
// EAX = number from register
// CL  = size of register (or number)
char __fastcall CrackMeGetNumberFromRegisterOrMemory(BYTE opcode)
{
  PARAMETER_CONFIG *opcodesIndexOffsetSize; // esi@1
  char value_of_register; // al@1
  char *pointer_to_register; // edi@3
  int register_offset; // ecx@3
  char register_size; // cl@5

  opcodesIndexOffsetSize = (PARAMETER_CONFIG *)&OpcodeRegOffsetAndRegisterSize;
  value_of_register = 0;
  while ( opcode != opcodesIndexOffsetSize->opcode )
  {
    ++opcodesIndexOffsetSize;
    if ( (char *)opcodesIndexOffsetSize == &endOfOpcodesStruct )
      return value_of_register;
  }
  register_offset = opcodesIndexOffsetSize->offset;
  pointer_to_register = (char *)vm_memory + register_offset;
  LOBYTE(register_offset) = opcodesIndexOffsetSize->size;
  do
  {
    LOBYTE(register_offset) = register_offset - 1;
    value_of_register = pointer_to_register[register_offset];
  }
  while ( (_BYTE)register_offset );
  register_size = opcodesIndexOffsetSize->size;
  return value_of_register;
}
```

One that is done it goes to get the next structure of the operando:



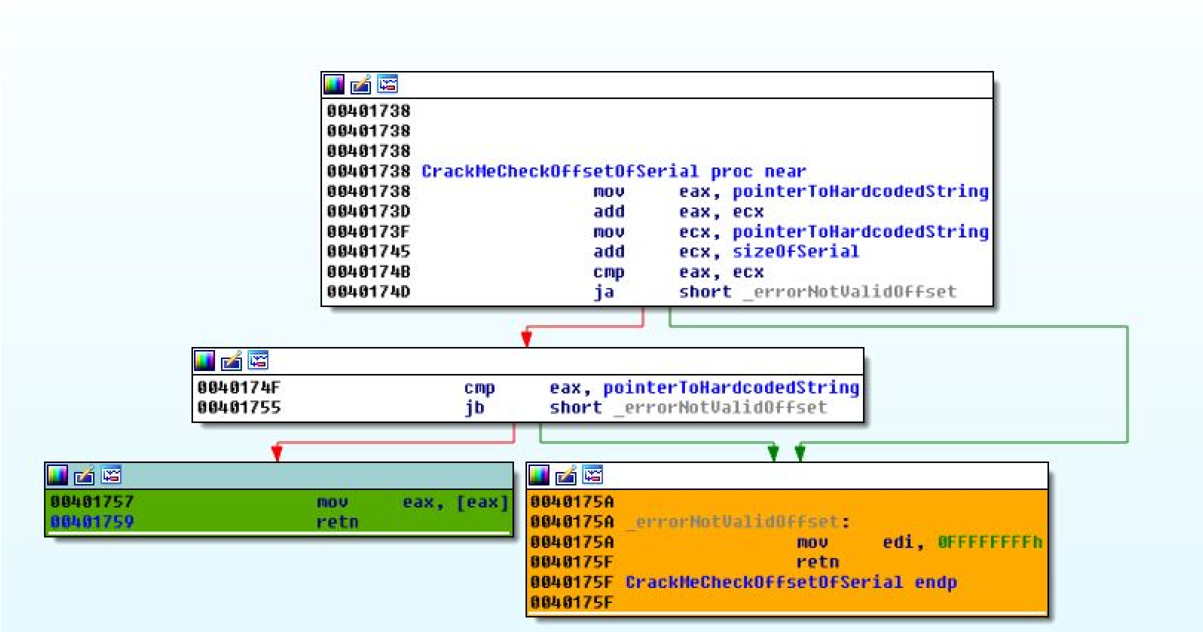
The next operand checked is the SERIAL_HASH_OPCODE (0x4F):

Reading SERIAL_HASH



```
00401555                call    CrackMeCheckOffsetOfSerial
0040155A                push    eax
0040155B                inc     dword ptr numberOfOperands
00401561                cmp     edi, 0FFFFFFFFh
00401564                jz      short _recoverValues
```

It calls a short function to read a value from the hardcoded serial given the read bytes from the code:



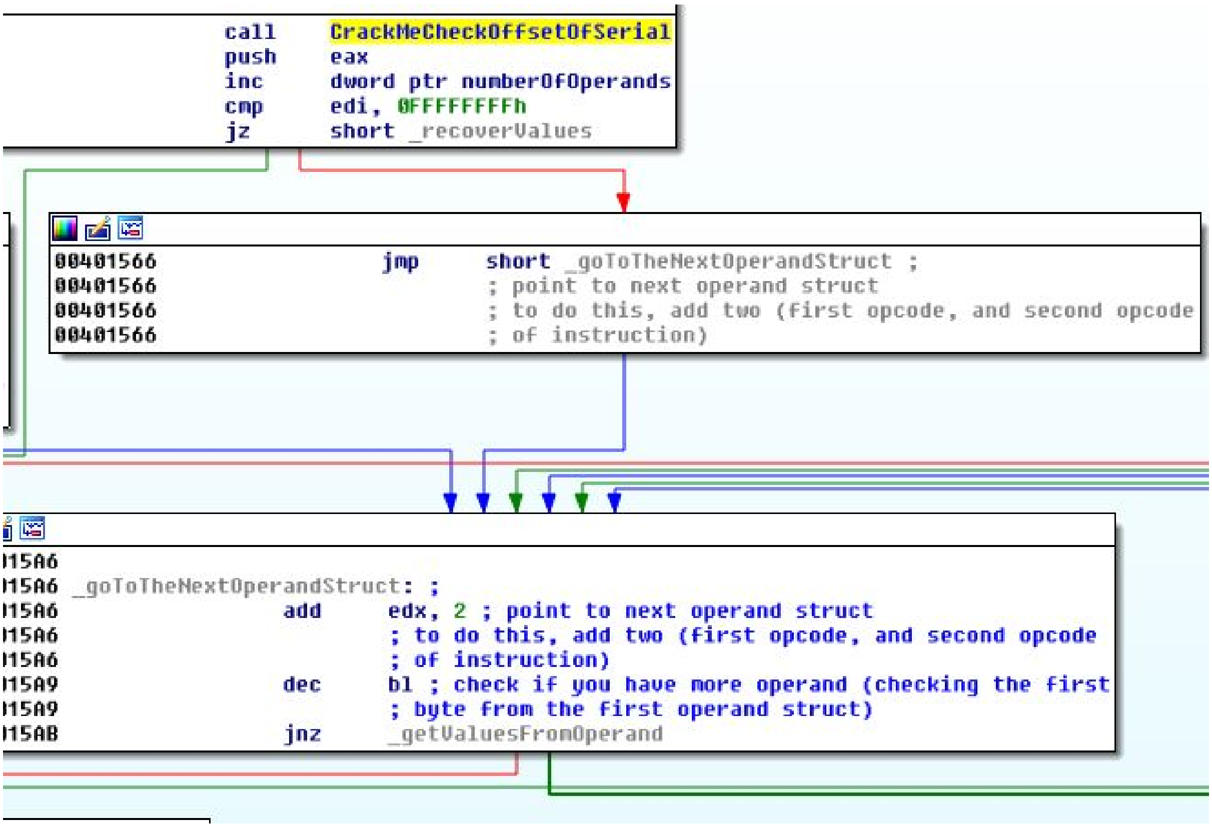
And the decompiled version:

```
char *__fastcall CrackMeCheckOffsetOfSerial(int a1)
{
  char *result; // eax@1

  result = (char *)pointerToHardcodedString + a1;
  if ( (char *)pointerToHardcodedString + a1 <= (char *)pointerToHardcodedString + sizeOfSerial
    && result >= pointerToHardcodedString )
  {
    result = *(char **)result;
  }
  return result;
}
```

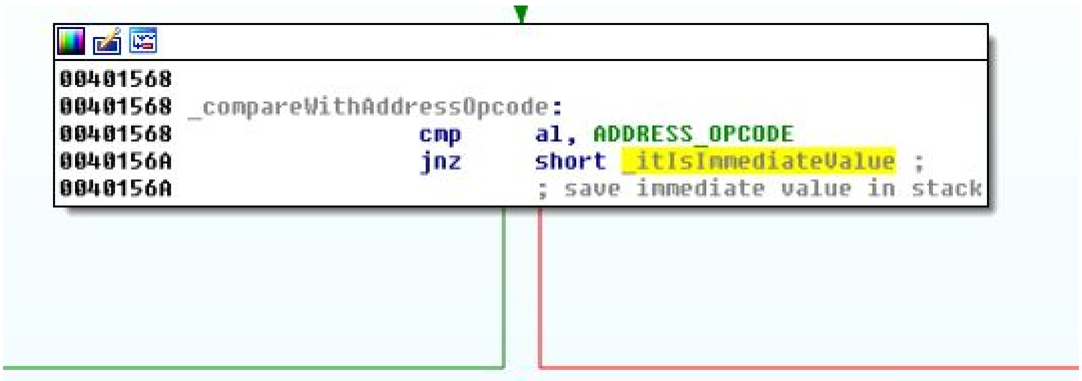
This code would be reading a DWORD from:

HardcodedString db 'aAb0cBd1eCf2gDh3jEk4lFm5nGp6qHr7sJt8uKv9w',0

The hardcoded string is taken from the beginning of the code, where a pointer to a string was set, finally goes to the next operandStruct.

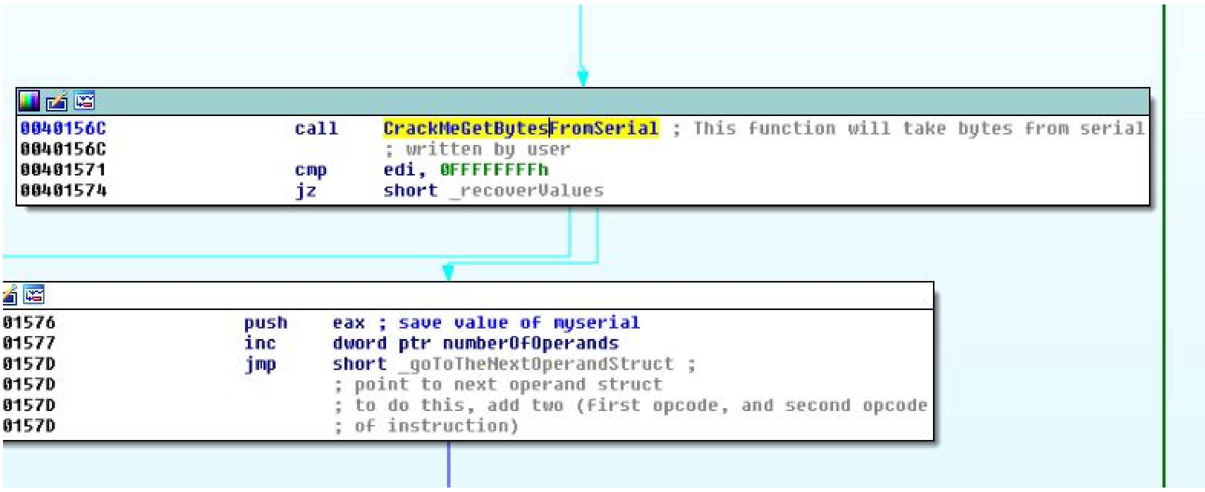


Next one will be to check with the ADDRESS_OPCODE (0x51):

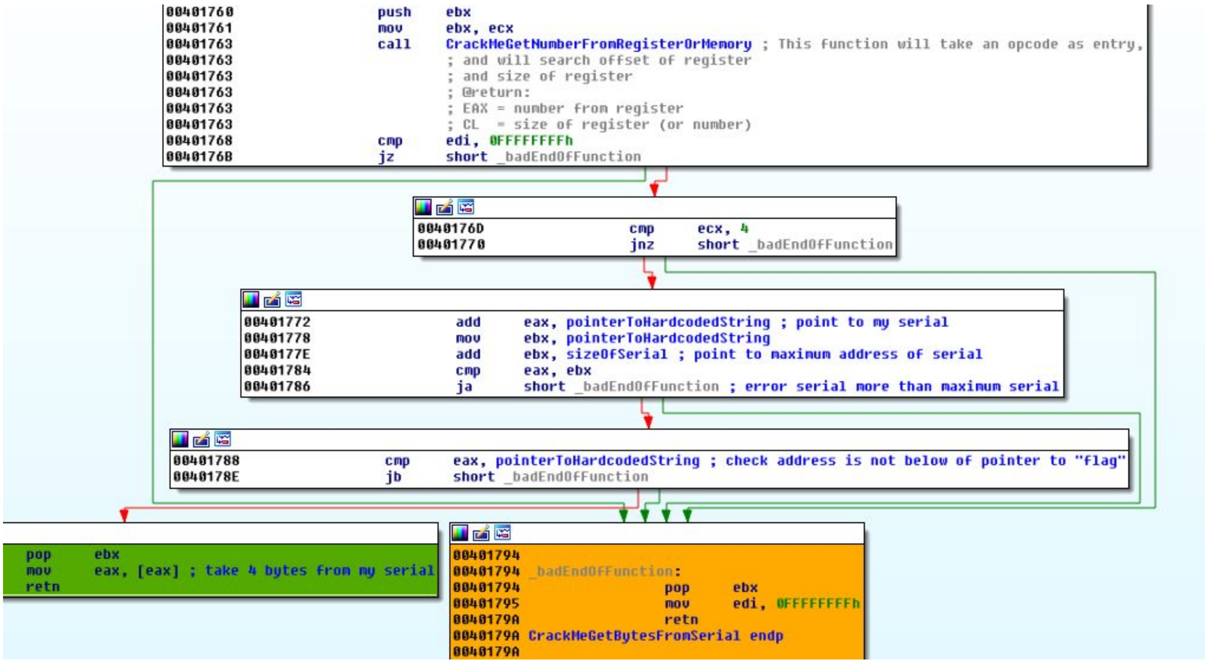


Reading my serial by Register value

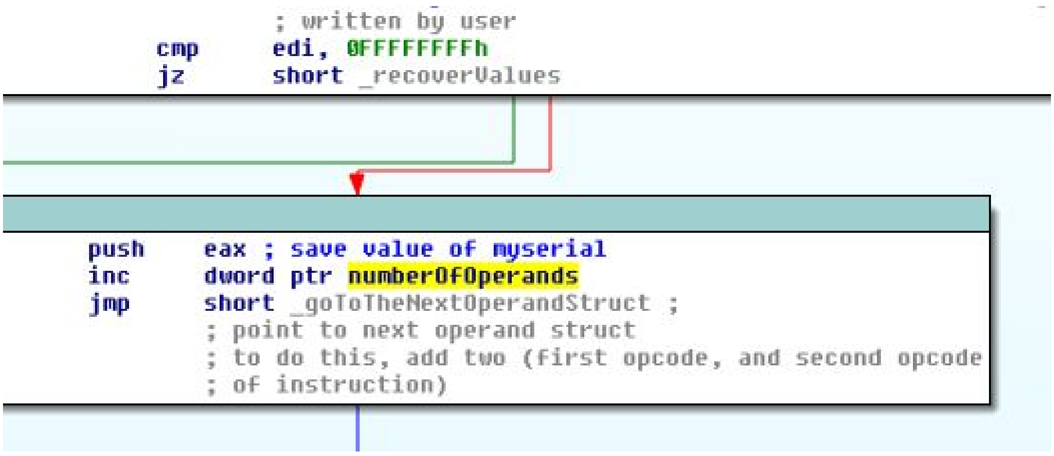
This is what it does the code with the ADDRESS_OPCODE, reading from *pointerToHardcodedString* which also points to the serial we wrote before:



And the function that it calls:

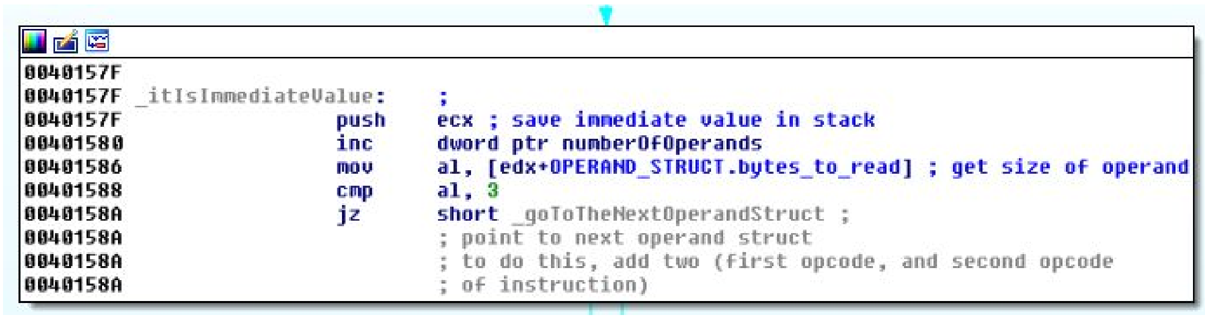


So it reads a register, and the value from this register is used as an offset to a dword from the serial we wrote, only those registers of 4 bytes are used, because the addresses are taken as 4 byte addresses. After that, access the address and return the dword.



After that just go to the next operand.

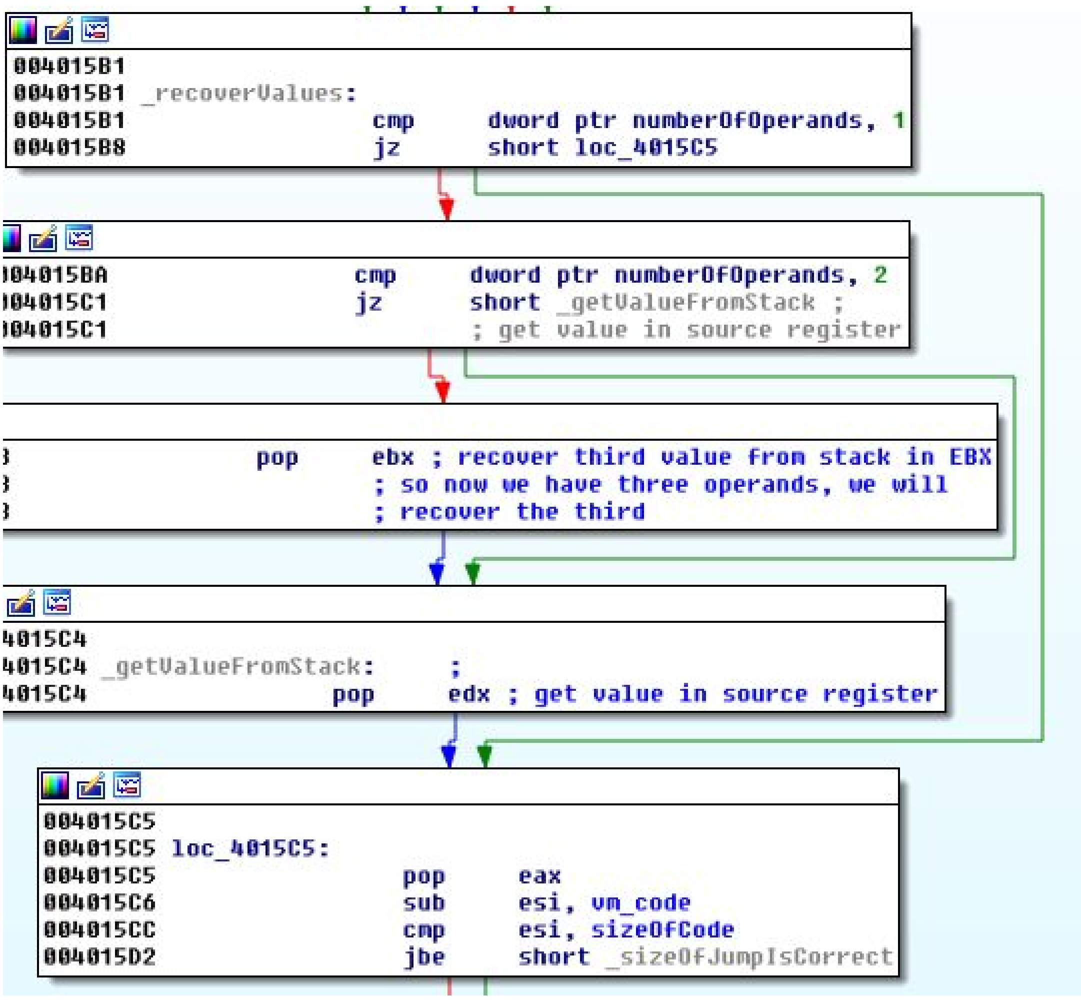
Reading IMMEDIATE value



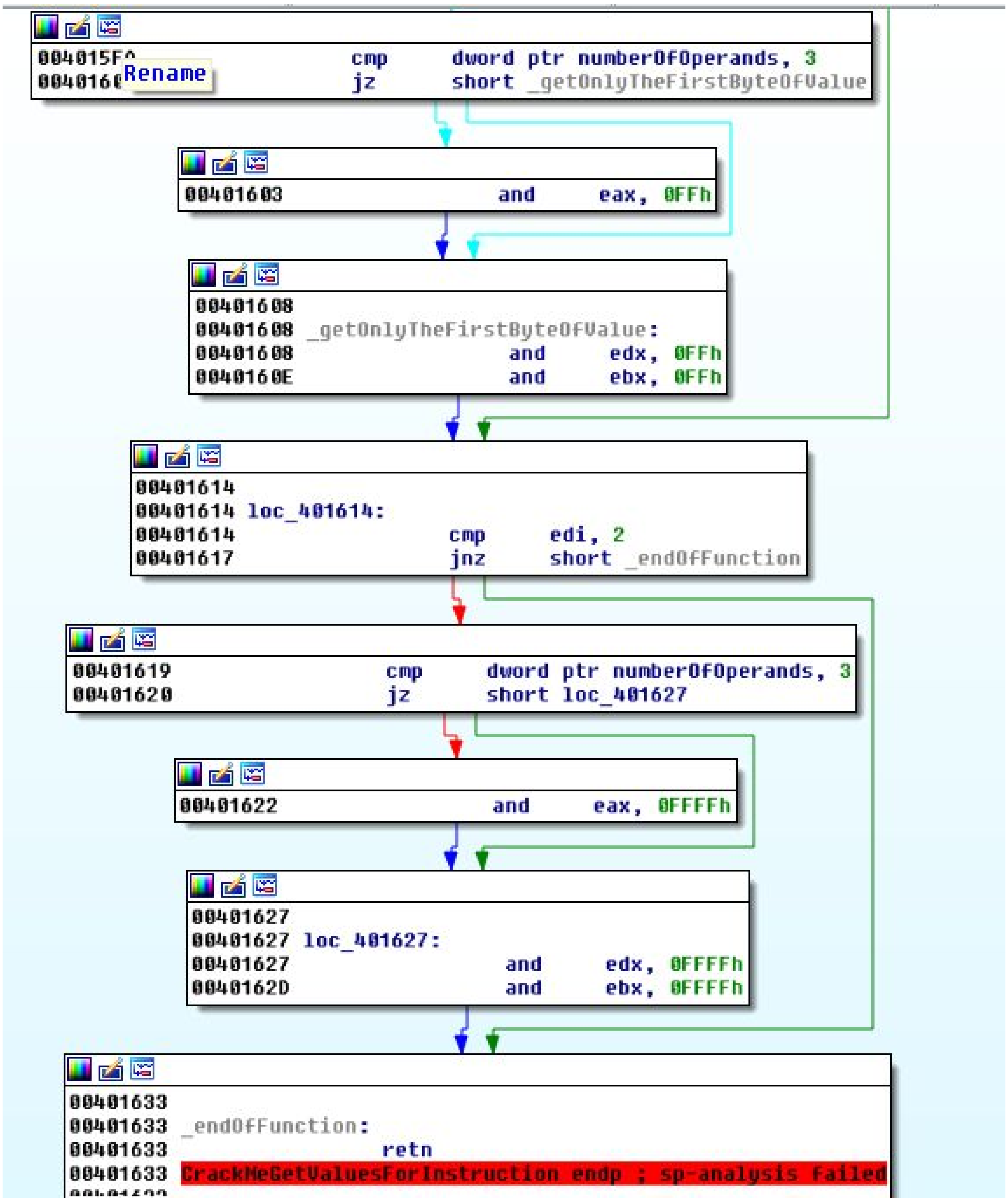
Just take the value and push it on the stack

Recovering values and finishing getting operands

Once we finish of pushing values into the stack taken from registers, serial_hash, my serial, or immediate, those are popped into specific registers:



Depending on number of operands, it will be popped out in different registers, and finally, AND operation is used to set the real value in the register avoiding high bytes values:



Execution of MOV, ADD, SUB, IMUL, IDIV, OR, XOR and AND (Part 2)

So now that we have the operands, we just have to apply the different instructions, let's go one by one:

```
PREPARING_MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND: ; CODE XREF: .
                call    CrackMeGetValuesForInstruction
                cmp     edi, 0FFFFFFFFh
                jnz     short MOVE_DWORD
                mov     eax, 0FFFFFFFFh
                mov     edi, vm_memory
                inc     [edi+VM_LOGIC.VM_EIP]
                retn
; ---------------------------------------------------------------------------
```

Returning from the call, just jump to MOV instruction.

MOV Instruction

```
MOVE_DWORD:                             ; CODE XREF: .rsrc:0040101D↑j
                mov     dword ptr unk_403DA2, edi
                mov     size_of_instruction, esi
                mov     edi, vm_memory
                mov     esi, [edi+VM_LOGIC.VM_EIP]
                mov     cl, [esi]
                cmp     cl, MOVE_OPCODE
                jnz     short ADD_DWORD
                mov     eax, edx
```

What it does is to check if the opcode corresponds to the MOVE_OPCODE, in case this is not true, it jumps to check if it’s add. The operation is the last instruction, so move the value from edx to eax.
In all these two operand operations, eax is used as the destination for the operation, and edx as one of the sources.

ADD Instruction

```
ADD_DWORD:                              ; CODE XREF: .rsrc:00401046↑j
                cmp     cl, ADD_OPCODE
                jnz     short SUB_DWORD
                add     eax, edx
```

Again, last instruction is an add operation.

SUB Instruction

```
SUB_DWORD:                              ; CODE XREF: .rsrc:0040104D↑j
                cmp     cl, SUB_OPCODE
                jnz     short IMUL_DWORD
                sub     eax, edx
```

IMUL Instruction

```
IMUL_DWORD:                                     ; CODE XREF: .rsrc:00401054↑j
                cmp     cl, IMUL_OPCODE
                jnz     short IDIV_DWORD
                imul    eax, edx
```

IDIV Instruction

```
IDIV_DWORD:                                     ; CODE XREF: .rsrc:0040105B↑j
                cmp     cl, IDIV_OPCODE
                jnz     short OR_DWORD
                test    edx, edx        ; check divisor is not 0 (so idiv could crash)
                jz      short ERROR_DIVIDED_BY_ZERO
                push    ebx
                mov     ebx, edx
                cdq
                idiv    ebx
                mov     [edi+VM_LOGIC.REG0x24], edx ; save the remainder
                pop     ebx
```

This is a little bit different, as it checks if the divisor is 0 to avoid crashes on the virtual machine (so an ERROR can be thrown and avoid real processor exceptions), finally idiv is done and the remainder stored in a specific register of the virtual machine. In case of division by zero, just skip the instruction:

```
ERROR_DIVIDED_BY_ZERO:                          ; CODE XREF: .rsrc:00401067↑j
                mov     eax, size_of_instruction
                add     [edi+VM_LOGIC.VM_EIP], eax
                mov     eax, 0FFFFFFFFh
                retn
```

OR Instruction

```
OR_DWORD:                                       ; C
                cmp     cl, OR_OPCODE
                jnz     short XOR_DWORD
                or      eax, edx
```

XOR Instruction

```
XOR_DWORD:                                      ; CODE XREF: .rsrc:00401076↑j
                cmp     cl, XOR_OPCODE
                jnz     short AND_DWORD
                xor     eax, edx
```

AND Instruction

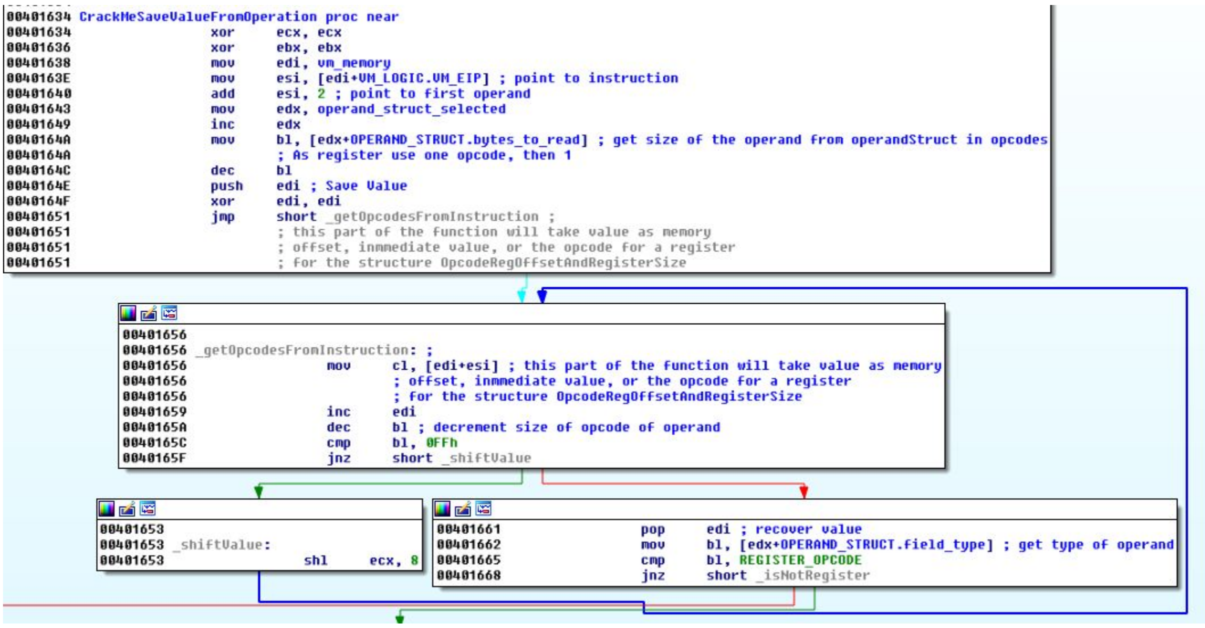
```
AND_DWORD:                                  ; CODE XREF: .rsrc:0040107D↑j
                cmp     cl, AND_OPCODE
                jnz     short MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND_INCREMENT_EIP
                and     eax, edx
```

Storing the values from the operation

This is another part of the VM, as we are able to read the operands, we have to be able to save the values in the VM registers or memory:

```
MOVE_ADD_SUB_IMUL_IDIV_OR_XOR_AND_INCREMENT_EIP: ; CODE XREF: .rsrc:00401084↑j
                call    CrackMeSaveValueFromOperation ; For the moment this function saves the result
                                        ; of an operation in a register, or destiny
                                        ; operand
                                        ; eax = result of operacion
                mov     eax, size_of_instruction
                add     [edi+VM_LOGIC.VM_EIP], eax
                xor     eax, eax
                retn
; ---------------------------------------------------------------------------
```

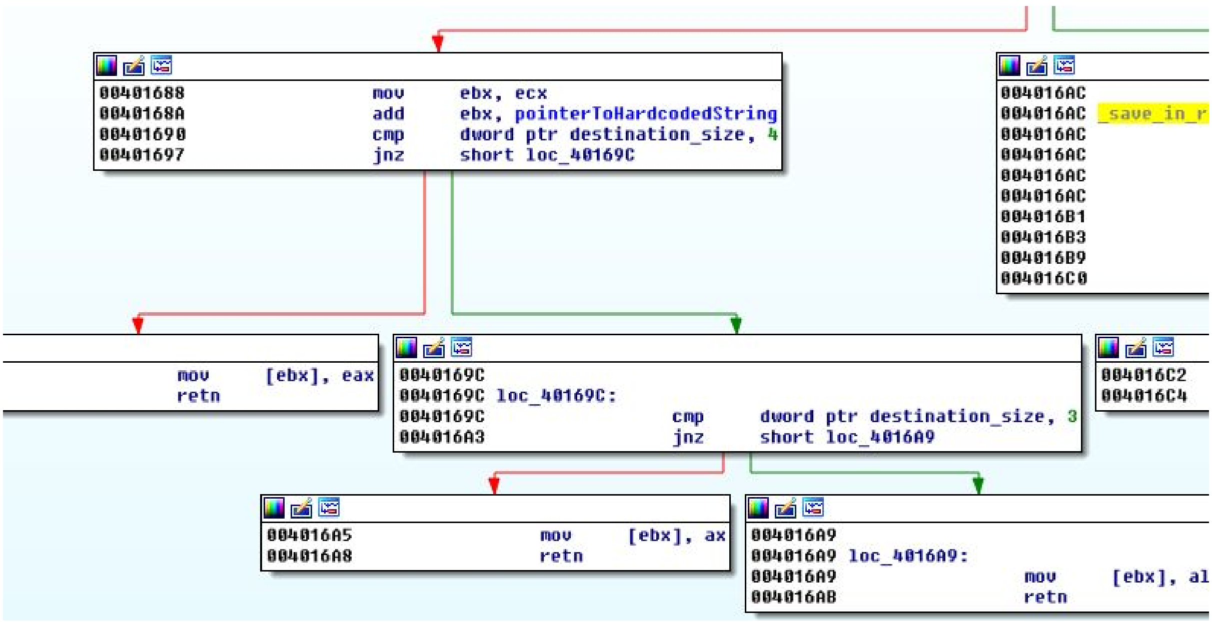
To do this, we use the function *CrackMeSaveValueFromOperation*:



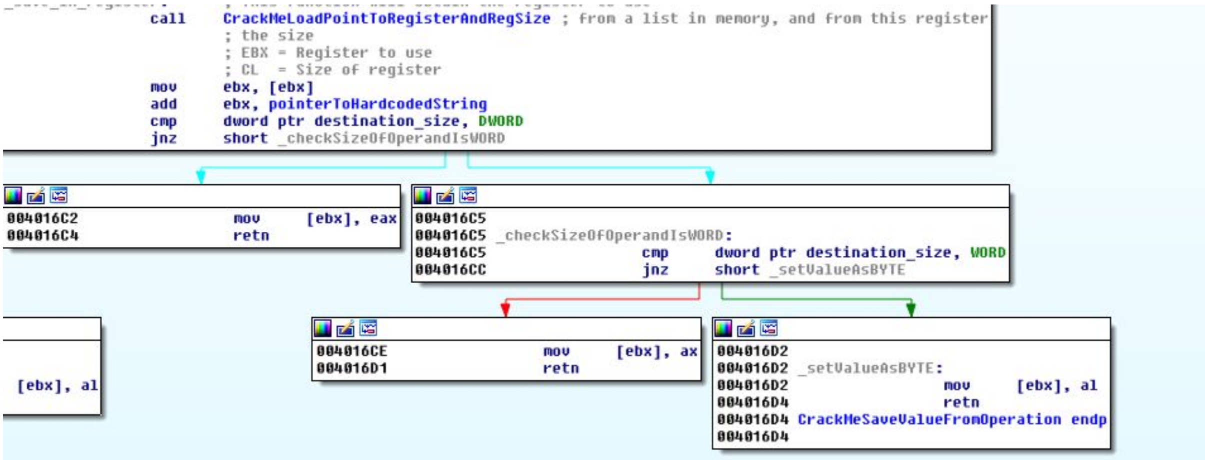
Again we get the operands, and we read the value for the first operand (as all the operations had 2 operands, take the first) that is where the result from the operation will be stored. In down right part we have that checks if the operand is a register opcode, if that is the case the result will be stored in a register with the next code:

So it takes in EBX the address to the register, and depending on the size, it’s stored eax, ax or al.

The second check if with SIGNATURE_OPCODE, so it can be stored there too:



Finally in the buffer of hardcoded string but with a register as the index:



Execution of INC, DEC and NOT

If we remember, the next set of instructions that are followed in memory in the vm are INC, DEC and NOT, these instructions just have one operand because increase, decrease or negate the bits from that operand, storing the result on it.

```
INC_DEC_NOT_OPERATIONS:                 ; DATA XREF: .rsrc:00403006↓o
                mov     esi, [edi]
                mov     al, [esi+1]
                mov     esi, offset INC_DEC_NOT_Second_Opcodes
                call    CrackMeCheckOpcodeWithList ; This function will check the second opcode
                                        ; of instruction with a list supplied in
                                        ; ESI, if value isn't in list will return
                                        ; -1, if it's in list return 0.
                                        ; @return:
                                        ; EAX = correct or not
                cmp     eax, 0FFFFFFFFh
                jnz     short PREPARING_INC_DEC_NOT
                retn
; ---------------------------------------------------------------------------

PREPARING_INC_DEC_NOT:                  ; CODE XREF: .rsrc:004010B6↑j
                call    CrackMeGetValuesForInstruction
                cmp     edi, 0FFFFFFFFh
                jnz     short INC_DWORD
                mov     eax, 0FFFFFFFFh
                mov     edi, vm_memory
                inc     [edi+VM_LOGIC.VM_EIP]
                retn
```

Same than before, we have a code to check if the opcode for the operand is correct and finally gets the operand value for the instruction.

INC Instruction

```
INC_DWORD:                              ; CODE XREF: .rsrc:004010C1↑j
                mov     dword ptr destination_size, edi
                mov     size_of_instruction, esi
                mov     edi, vm_memory
                mov     esi, [edi+VM_LOGIC.VM_EIP]
                mov     cl, [esi]
                cmp     cl, INC_OPCODE
                jnz     short DEC_DWORD
                inc     eax
```

Just check if it’s the correct instruction, if not jumps to the DEC operation, and if it’s correct increment the value in EAX.

DEC Instruction

```
DEC_DWORD:                              ; CODE XREF: .rsrc:004010EA↑j
                cmp     cl, DEC_OPCODE
                jnz     short NOT_DWORD
                dec     eax
```

NOT Instruction

```
NOT_DWORD:                              ; CODE XREF: .rsrc:004010F0↑j
                cmp     cl, NOT_OPCODE
                jnz     short INC_DEC_NOT_INCREMENT_EIP
                not     eax
```

Storing the values from the operation

This is the same than the previous one again:

```
INC_DEC_NOT_INCREMENT_EIP:              ; CODE XREF: .rsrc:004010F6↑j
                call    CrackMeSaveValueFromOperation ; For the moment this function saves the result
                                        ; of an operation in a register, or destiny
                                        ; operand
                                        ; eax = result of operacion
                mov     eax, size_of_instruction
                add     [edi+VM_LOGIC.VM_EIP], eax
                xor     eax, eax
                retn
```

Execution of SHIFT, ROR and ROL

```
SHIFT_ROR_ROL_OPERATIONS:               ; DATA XREF: .rsrc:0040300B↓o
                mov     esi, [edi]
                mov     al, [esi+1]
                mov     esi, offset SHIFT_ROR_ROL_Operands
                call    CrackMeCheckOpcodeWithList ; This function will check the second opcode
                                        ; of instruction with a list supplied in
                                        ; ESI, if value isn't in list will return
                                        ; -1, if it's in list return 0.
                                        ; @return:
                                        ; EAX = correct or not
                cmp     eax, 0FFFFFFFFh
                jnz     short PREPARING_SHIFT_ROR_ROL ; this time uses dl instead of cl because cl will be use it
                                        ; for shift, ror and rol
                retn
; ---------------------------------------------------------------------------

PREPARING_SHIFT_ROR_ROL:                ; CODE XREF: .rsrc:0040111B↑j
                call    CrackMeGetValuesForInstruction ; this time uses dl instead of cl because cl will be use it
                                        ; for shift, ror and rol
                cmp     edi, 0FFFFFFFFh
                jnz     short SHIFT_RIGHT_DWORD
                mov     eax, 0FFFFFFFFh
                mov     edi, vm_memory
                inc     [edi+VM_LOGIC.VM_EIP]
                retn
```

As the comment says, this time instead of using EDX register as the second operand for the instruction, ECX or more exactly CL will be used to shift, ror or rol the values.

SHIFT Instruction

We can have Shift to the right and shift to the left:

```
SHIFT_RIGHT_DWORD:                      ; CODE XREF: .rsrc:00401126↑j
                mov     dword ptr destination_size, edi
                mov     size_of_instruction, esi
                mov     edi, vm_memory
                mov     esi, [edi]
                mov     cl, [esi]
                xchg    edx, ecx
                cmp     dl, SHR_OPCODE
                jnz     short SHIFT_LEFT_DWORD
                shr     eax, cl

SHIFT_LEFT_DWORD:                       ; CODE XREF: .rsrc:00401151↑j
                cmp     dl, SHL_OPCODE
                jnz     short ROR_BYTE
                shl     eax, cl
```

ROR Instruction

This instruction will be three one for byte, other for word and finally for dword:

```
ROR_BYTE:                               ; CODE XREF: .rsrc:00401158↑j
                cmp     dl, ROR_OPCODE
                jnz     short ROL_BYTE
                cmp     dword ptr destination_size, 1
                jnz     short ROR_WORD
                ror     al, cl
                jmp     short ROL_BYTE
; ---------------------------------------------------------------------------

ROR_WORD:                               ; CODE XREF: .rsrc:00401168↑j
                cmp     dword ptr destination_size, 2
                jnz     short ROR_DWORD
                ror     ax, cl
                jmp     short ROL_BYTE
; ---------------------------------------------------------------------------

ROR_DWORD:                              ; CODE XREF: .rsrc:00401175↑j
                ror     eax, cl
```

ROL Instruction

As the previous one, this instruction can be applied to byte, word or dword:

```
ROL_BYTE:                               ; CODE XREF: .rsrc:0040115F↑j
                                        ; .rsrc:0040116C↑j ...
                cmp     dl, ROL_OPCODE
                jnz     short SHIFT_ROR_ROL_INCREMENT_EIP
                cmp     dword ptr destination_size, 1
                jnz     short ROL_WORD
                rol     al, cl
                jmp     short SHIFT_ROR_ROL_INCREMENT_EIP
; ---------------------------------------------------------------------------

ROL_WORD:                               ; CODE XREF: .rsrc:0040118A↑j
                cmp     dword ptr destination_size, 2
                jnz     short ROL_DWORD
                rol     ax, cl
                jmp     short SHIFT_ROR_ROL_INCREMENT_EIP
; ---------------------------------------------------------------------------

ROL_DWORD:                              ; CODE XREF: .rsrc:00401197↑j
                rol     eax, cl
```

Storing the values from the operation

```
SHIFT_ROR_ROL_INCREMENT_EIP:            ; CODE XREF: .rsrc:00401181↑j
                                        ; .rsrc:0040118E↑j ...
                xchg    edx, ecx
                call    CrackMeSaveValueFromOperation ; For the moment this function saves the result
                                        ; of an operation in a register, or destiny
                                        ; operand
                                        ; eax = result of operacion
                mov     eax, size_of_instruction
                add     [edi+VM_LOGIC.VM_EIP], eax
                xor     eax, eax
                retn
```

Execution of JUMP

This is an inconditional jump, the implementation it's straightforward just setting new value in VM_EIP, so next instruction will start from there:

```
JMP_OPERATION:                          ; DATA XREF: .rsrc:00403010↓o
                mov     esi, [edi]
                mov     al, [esi+1]
                mov     esi, offset JMP_CALL_SECOND_OPCODES
                call    CrackMeCheckOpcodeWithList ; This function will check the second opcode
                                        ; of instruction with a list supplied in
                                        ; ESI, if value isn't in list will return
                                        ; -1, if it's in list return 0.
                                        ; @return:
                                        ; EAX = correct or not
                cmp     eax, 0FFFFFFFFh
                jnz     short PREPARING_JMP
                retn
; ---------------------------------------------------------------------------

PREPARING_JMP:                          ; CODE XREF: .rsrc:004011C3↑j
                call    CrackMeGetValuesForInstruction
                cmp     edi, 0FFFFFFFFh
                jnz     short JMP       ; it looks like a JMP or GoTo
                mov     eax, 0FFFFFFFFh
                mov     edi, vm_memory
                inc     [edi+VM_LOGIC.VM_EIP]
                retn
; ---------------------------------------------------------------------------

JMP:                                    ; CODE XREF: .rsrc:004011CE↑j
                mov     edi, vm_memory  ; it looks like a JMP or GoTo
                add     eax, vm_code
                mov     [edi+VM_LOGIC.VM_EIP], eax
                xor     eax, eax
                retn
```

Execution of JZ, JNZ, JA, JB, JNB and JBE

```
004011EF
004011EF JZ_JNZ_JA_JB_JNB_JBE_OPERATIONS:        ; DATA XREF: .rsrc:00403015↓o
004011EF                 mov     edi, vm_memory
004011F5                 mov     esi, [edi+VM_LOGIC.VM_EIP]
004011F7                 mov     al, [esi+1]
004011FA                 mov     esi, offset JZ_JNZ_JA_JB_JNB_JBE_opcodes
004011FF                 call    CrackMeCheckOpcodeWithList ; This function will check the second opcode
004011FF                                         ; of instruction with a list supplied in
004011FF                                         ; ESI, if value isn't in list will return
004011FF                                         ; -1, if it's in list return 0.
004011FF                                         ; @return:
004011FF                                         ; EAX = correct or not
00401204                 cmp     eax, 0FFFFFFFFh
00401207                 jnz     short PREPARING_JZ_JNZ_JA_JB_JNB_JBE
00401209                 retn
0040120A ; ---------------------------------------------------------------------------
0040120A
0040120A PREPARING_JZ_JNZ_JA_JB_JNB_JBE:         ; CODE XREF: .rsrc:00401207↑j
0040120A                 call    CrackMeGetValuesForInstruction
0040120F                 cmp     edi, 0FFFFFFFFh
00401212                 jnz     short JZ
00401214                 mov     eax, 0FFFFFFFFh
00401219                 mov     edi, vm_memory
0040121F                 inc     [edi+VM_LOGIC.VM_EIP]
00401221                 retn
```

The beginning of the code is mostly the same than previous, now the instructions are two steps instruction, because as these are conditional jumps a comparison has to be done previous to the jump, after that the correct jump is applied.
There will be 3 operands for the instruction, one with the new address, and others for comparisons.

JZ Instruction

```
JZ:                                      ; CODE XREF: .rsrc:00401212↑j
                mov     dword ptr destination_size, edi
                mov     size_of_instruction, esi
                mov     edi, vm_memory
                mov     esi, [edi+VM_LOGIC.VM_EIP]
                mov     cl, [esi]
                cmp     cl, JZ_OPCODE
                jnz     short JNZ
                cmp     edx, ebx
                jz      short SET_VALUE_IN_EIP
                jmp     short JZ_JNZ_JA_JB_JNB_JBE_INCREMENT_EIP
```

A comparison is done using edx and ebx, and the jump if zero instruction is set to jump to an address that set EIP (this will be the case in all the jumps).

JNZ Instruction

```
00401243
00401243 JNZ:                                    ; CODE XREF: .rsrc:0040123B↑j
00401243                 cmp     cl, JNZ_OPCODE
00401246                 jnz     short JA
00401248                 cmp     edx, ebx
0040124A                 jnz     short SET_VALUE_IN_EIP
0040124C                 jmp     short JZ_JNZ_JA_JB_JNB_JBE_INCREMENT_EIP
```

JA Instruction

```
0040124E
0040124E JA:                                  ; CODE XREF: .rsrc:00401246↑j
0040124E            cmp     cl, JA_OPCODE
00401251            jnz     short JB
00401253            cmp     edx, ebx
00401255            ja      short SET_VALUE_IN_EIP
00401257            jmp     short JZ_JNZ_JA_JB_JNB_JBE_INCREMENT_EIP
```

JB Instruction

```
00401259
00401259 JB:                                  ; CODE XREF: .rsrc:00401251↑j
00401259            cmp     cl, JB_OPCODE
0040125C            jnz     short JNB
0040125E            cmp     edx, ebx
00401260            jb      short SET_VALUE_IN_EIP
00401262            jmp     short JZ_JNZ_JA_JB_JNB_JBE_INCREMENT_EIP
```

JNB Instruction

```
00401264
00401264 JNB:                                 ; CODE XREF: .rsrc:0040125C↑j
00401264            cmp     cl, JNB_OPCODE
00401267            jnz     short JBE
00401269            cmp     edx, ebx
0040126B            jnb     short SET_VALUE_IN_EIP
0040126D            jmp     short JZ_JNZ_JA_JB_JNB_JBE_INCREMENT_EIP
```

JBE Instruction

```
:0040126F
:0040126F JBE:                                ; CODE XREF: .rsrc:00401267↑j
:0040126F           cmp     edx, ebx
:00401271           jbe     short SET_VALUE_IN_EIP
:00401273           jmp     short JZ_JNZ_JA_JB_JNB_JBE_INCREMENT_EIP
```

Execution of CALL

This is a very important instruction in any processor because allows the programmers to avoid writing useful code once and again and again, writing that code inside of a function and finally calling that function.
When a call is done, the address after the call is stored on the stack so when ret is executed the address is recovered from the stack and return there, the program has to check if there’s enough space on the stack, that will be done too:

```
PREPARING_CALL:                         ; CODE XREF: .rsrc:0040129C↑j
                call    CrackMeGetValuesForInstruction
                cmp     edi, 0FFFFFFFFh ; now in EAX offset to jump (or call)
                jnz     short CHECK_CALL_IS_POSSIBLE
                mov     eax, 0FFFFFFFFh
                mov     edi, vm_memory
                inc     [edi+VM_LOGIC.VM_EIP]
                retn
; ---------------------------------------------------------------------------

CHECK_CALL_IS_POSSIBLE:                 ; CODE XREF: .rsrc:004012A7↑j
                mov     edi, vm_memory
                mov     edx, [edi+VM_LOGIC.VM_EIP]
                add     edx, esi        ; EDX = Actual Instruction + size of instruction = Point to the next instruction
                mov     ecx, [edi+VM_LOGIC.VM_ESP]
                cmp     ecx, 4          ; check if can push in the stack
                jnb     short CALL      ; substrate 4 to the VM_ESP
                add     [edi+VM_LOGIC.VM_EIP], esi ; point to the next instruction directly
                mov     eax, 0FFFFFFFFh
                retn
; ---------------------------------------------------------------------------

CALL:                                   ; CODE XREF: .rsrc:004012C7↑j
                sub     [edi+VM_LOGIC.VM_ESP], 4 ; substrate 4 to the VM_ESP
                sub     ecx, 4          ; Substrate 4 from value of VM_ESP saved in ECX
                add     ecx, vm_memory  ; Make ECX Point to the virtual stack from vm
                add     ecx, 3Ch
                mov     [ecx], edx      ; save the next instruction in the stack
                add     eax, vm_code    ; Point to the address of function
                mov     [edi+VM_LOGIC.VM_EIP], eax ; save function address in the EIP (make the call)
                xor     eax, eax
                retn
```

In the second block, the current address is obtained from the VM_EIP, and the size of the instruction is added, so in EDX now we have the pointer to the instruction after the call, after that the offset of VM_ESP is obtained in ECX (remember that on a stack we go from high addresses or offsets to low addresses or offsets), and checks if the value is below from 4 (in that case it would be 0), if it's above or equal jumps to make the call, the address from next instruction is stored on the stack, and to make the call is simply just storing in VM_EIP the next address.

Execution of PUSH and POP

Here two different preparation codes are performed, why? Because these operations are one the opposite of the other, but PUSH allows one operand that POP doesn't: an IMMEDIATE value (we cannot pop to an immediate value of course).

So here we would have the preparation of the push:

```
PUSH_POP_OPERATION:                     ; DATA XREF: .rsrc:0040301F↓o
                mov     esi, [edi+VM_LOGIC.VM_EIP]
                mov     al, [esi]
                cmp     al, PUSH_OPCODE ; check if instruction is push, if it's pop, another
                                        ; code has to be executed
                jnz     short PREPARING_POP
                mov     al, [esi+1]     ; get opcode
                mov     esi, offset PUSH_SECOND_OPCODES ; values can be register, address or immediate value
                call    CrackMeCheckOpcodeWithList ; This function will check the second opcode
                                        ; of instruction with a list supplied in
                                        ; ESI, if value isn't in list will return
                                        ; -1, if it's in list return 0.
                                        ; @return:
                                        ; EAX = correct or not
                cmp     eax, 0FFFFFFFFh
                jnz     short GET_VALUES_FOR_PUSH ; get values for push
                retn
```

```
instruction_size_correct:                    ; CODE XREF: .rsrc:00401311↑j
                cmp     edi, 4              ; check source size is 4
                jz      short PREPARING_PUSH
                mov     eax, 0FFFFFFFFh
                mov     edi, vm_memory
                add     [edi+VM_LOGIC.VM_EIP], esi
                retn
; ---------------------------------------------------------------------------

PREPARING_PUSH:                              ; CODE XREF: .rsrc:00401324↑j
                mov     dword ptr source_size, edi
                mov     size_of_instruction, esi
                mov     edi, vm_memory
                cmp     [edi+VM_LOGIC.VM_ESP], 4
                jnb     short PUSH_DWORD
                mov     eax, size_of_instruction
                add     [edi+VM_LOGIC.VM_EIP], eax
                mov     eax, 0FFFFFFFFh
                retn
```

And here the preparation of pop:

```
00401375 PREPARING_POP:                          ; CODE XREF: .rsrc:004012F4↑j
00401375                 mov     esi, offset POP_SECOND_OPCODES ; check if address or register
0040137A                 mov     al, [esi+1]
0040137D                 call    CrackMeCheckOpcodeWithList ; This function will check the second opcode
0040137D                                         ; of instruction with a list supplied in
0040137D                                         ; ESI, if value isn't in list will return
0040137D                                         ; -1, if it's in list return 0.
0040137D                                         ; @return:
0040137D                                         ; EAX = correct or not
00401382                 cmp     eax, 0FFFFFFFFh
00401385                 jnz     short POP_DWORD
00401387                 retn
00401388 ; ---------------------------------------------------------------------------
00401388
00401388 POP_DWORD:                              ; CODE XREF: .rsrc:00401385↑j
00401388                 mov     edi, vm_memory
0040138E                 mov     esi, [edi+VM_LOGIC.VM_EIP]
00401390                 call    CrackMeGetOperandStructAndSizeOfInstruction ; This function will do the algorithm
00401390                                         ; to get the struct from the operand (where
00401390                                         ; VM will get type of operand and other values)
00401390                                         ; and the size of instruction.
00401390                                         ; @return:
00401390                                         ; EAX = address of the operand struct
00401390                                         ; EBX = size of instruction
```

PUSH Instruction

Push and Pop are a game with ESP, a push instruction can be translated to something like:

sub esp, 4
mov [esp], <reg,addr,imm>

```
00401359 PUSH_DWORD:                             ; CODE XREF: .rsrc:0040134A↑j
00401359                 sub     [edi+VM_LOGIC.VM_ESP], 4 ; substrate VM_ESP by 4
0040135D                 mov     esi, [edi+VM_LOGIC.VM_ESP]
00401360                 add     esi, vm_memory
00401366                 add     esi, 3Ch
00401369                 mov     [esi], eax      ; store the value on the stack
0040136B                 mov     eax, size_of_instruction
00401370                 add     [edi+VM_LOGIC.VM_EIP], eax ; go to next instruction
00401372                 xor     eax, eax
00401374                 retn
```

POP Instruction

The pop instruction can be translated to:

```
mov <reg,addr>, [esp]
add esp, 4
```

This can be seen in the VM code:

```
00401388
00401388 POP_DWORD:                            ; CODE XREF: .rsrc:00401385↑j
00401388                 mov     edi, vm_memory
0040138E                 mov     esi, [edi+VM_LOGIC.VM_EIP]
00401390                 call    CrackMeGetOperandStructAndSizeOfInstruction ; This function will do the algorithm
00401390                                       ; to get the struct from the operand (where
00401390                                       ; VM will get type of operand and other values)
00401390                                       ; and the size of instruction.
00401390                                       ; @return:
00401390                                       ; EAX = address of the operand struct
00401390                                       ; EBX = size of instruction
00401395                 mov     esi, [edi+VM_LOGIC.VM_ESP] ; get VM_ESP in esi
00401398                 add     esi, vm_memory
0040139E                 add     esi, 3Ch
004013A1                 mov     eax, [esi]      ; retrieve the value from the stack
004013A3                 add     [edi+VM_LOGIC.VM_ESP], 4 ; increment the stack in 4
004013A7                 call    CrackMeSaveValueFromOperation ; For the moment this function saves the result
004013A7                                       ; of an operation in a register, or destiny
004013A7                                       ; operand
004013A7                                       ; eax = result of operacion
004013AC                 mov     eax, size_of_instruction
004013B1                 add     [edi+VM_LOGIC.VM_EIP], eax
004013B3                 xor     eax, eax
004013B5                 retn
```

Only pops needs to store the value in a specific vm register or address.

Execution of PUSHAD and POPAD

These instructions store the registers on the stack, and pop the registers from the stack.

PUSHAD Instruction

```
004013B6 PUSAD_OPERATION:                        ; DATA XREF: .rsrc:00403024↓o
004013B6                 mov     esi, [edi+VM_LOGIC.VM_EIP] ; set VM_EIP address in ESI
004013B8                 mov     al, [esi]       ; get the opcode from EIP
004013BA                 cmp     al, PUSHAD_OPCODE
004013BC                 jnz     short preparingPOPAD
004013BE                 cmp     [edi+VM_LOGIC.VM_ESP], 18h ; check if there's enough size on the stack
004013C2                 jnb     short PUSHAD    ; substrate space enough for registers
004013C4                 mov     eax, 0FFFFFFFFh
004013C9                 retn
004013CA ; ---------------------------------------------------------------------------
004013CA
004013CA PUSHAD:                                 ; CODE XREF: .rsrc:004013C2↑j
004013CA                 sub     [edi+VM_LOGIC.VM_ESP], 30h ; substrate space enough for registers
004013CE                 mov     edx, [edi+VM_LOGIC.VM_ESP] ; use edx as VM_ESP
004013D1                 add     edx, vm_memory
004013D7                 add     edx, 3Ch
004013DA                 mov     eax, [edi+VM_LOGIC.REG0x4] ; Now goes register by register saving their values on the stack
004013DD                 mov     [edx], eax
004013DF                 mov     eax, [edi+VM_LOGIC.REG0x8]
004013E2                 mov     [edx+4], eax
004013E5                 mov     eax, [edi+VM_LOGIC.REG0xC]
004013E8                 mov     [edx+8], eax
004013EB                 mov     eax, [edi+VM_LOGIC.REG0x10]
004013EE                 mov     [edx+0Ch], eax
004013F1                 mov     eax, [edi+VM_LOGIC.REG0x1C]
004013F4                 mov     [edx+10h], eax
004013F7                 mov     eax, [edi+VM_LOGIC.REG0x20]
004013FA                 mov     [edx+14h], eax
004013FD                 mov     eax, [edi+VM_LOGIC.REG0x24]
00401400                 mov     [edx+18h], eax
00401403                 mov     eax, [edi+VM_LOGIC.REG0x28]
00401406                 mov     [edx+1Ch], eax
00401409                 mov     eax, [edi+VM_LOGIC.REG0x2C]
0040140C                 mov     [edx+20h], eax
0040140F                 mov     eax, [edi+VM_LOGIC.REG0x30]
00401412                 mov     [edx+24h], eax
00401415                 mov     eax, [edi+VM_LOGIC.REG0x34]
00401418                 mov     [edx+28h], eax
0040141B                 mov     eax, [edi+VM_LOGIC.REG0x38]
0040141E                 mov     [edx+2Ch], eax
00401421                 inc     [edi+VM_LOGIC.VM_EIP]
00401423                 xor     eax, eax
00401425                 retn
```

POPAD Instruction

This is exactly the opposite of the other:

```
6
6 preparingPOPAD:                          ; CODE XREF: .rsrc:004013BC↑j
6                 mov     eax, stackSize
B                 sub     eax, 18h
E                 cmp     [edi+VM_LOGIC.VM_ESP], eax
1                 jb      short POPAD
3                 mov     eax, 0FFFFFFFFh
8                 retn
9 ; ---------------------------------------------------------------------------
9
9 POPAD:                                   ; CODE XREF: .rsrc:00401431↑j
9                 mov     edx, [edi+VM_LOGIC.VM_ESP]
C                 add     edx, vm_memory
2                 add     edx, 3Ch
5                 mov     eax, [edx]
7                 mov     [edi+VM_LOGIC.REG0x4], eax
A                 mov     eax, [edx+4]
D                 mov     [edi+VM_LOGIC.REG0x8], eax
0                 mov     eax, [edx+8]
3                 mov     [edi+VM_LOGIC.REG0xC], eax
6                 mov     eax, [edx+0Ch]
9                 mov     [edi+VM_LOGIC.REG0x10], eax
C                 mov     eax, [edx+10h]
F                 mov     [edi+VM_LOGIC.REG0x1C], eax
2                 mov     eax, [edx+14h]
5                 mov     [edi+VM_LOGIC.REG0x20], eax
8                 mov     eax, [edx+18h]
B                 mov     [edi+VM_LOGIC.REG0x24], eax
E                 mov     eax, [edx+1Ch]
1                 mov     [edi+VM_LOGIC.REG0x28], eax
4                 mov     eax, [edx+20h]
7                 mov     [edi+VM_LOGIC.REG0x2C], eax
A                 mov     eax, [edx+24h]
D                 mov     [edi+VM_LOGIC.REG0x30], eax
0                 mov     eax, [edx+28h]
3                 mov     [edi+VM_LOGIC.REG0x34], eax
6                 mov     eax, [edx+2Ch]
9                 mov     [edi+VM_LOGIC.REG0x38], eax
C                 add     [edi+VM_LOGIC.VM_ESP], 30h
0                 inc     [edi+VM_LOGIC.VM_EIP]
2                 xor     eax, eax
4                 retn
```

End of the VM

This is the end of how it works the virtual machine of Genaytyk, it was pretty interesting and really good to learn about how virtualization is done.

After understanding how the structures work, and how the VM executes the instructions, it's possible to write a disassembler in a pseudo-assembly using the VM registers, even with the problem of having more registers than a x86 structure, I also wrote a disassembler that finally writes an assembly output proper for a x86 assembler. You can find them in the next links:

- https://github.com/Fare9/Genaytyk-VM/blob/master/vm_disassembler.py
- https://github.com/Fare9/Genaytyk-VM/blob/master/vm_disassembler_x86.py

And finally the assembly output for both:

- https://github.com/Fare9/Genaytyk-VM/blob/master/vm_instructions.txt
- https://github.com/Fare9/Genaytyk-VM/blob/master/vm_instructions.asm

My next idea is to learn about LLVM translation from binary to LLVM IR with this challenge. For what I've seen generation of LLVM IR is not as easy so some hours of coding it are necessary, I've found that this book teaches really well about that topic for people interested on it: https://www.amazon.es/LLVM-Essentials-Suyog-Sarda/dp/1785280805

Last words

I have to give thanks to my friend Valthek who sent me this challenge two years ago to learn about reversing of VMs, my friend Arrizen who once or twice had a skype with me to reversing it at night, my girlfriend that even when it was easter week allowed me to work on this.

This analysis was written and improved on 2020, but the complete analysis was done in 2018.

References

- Intruction set of x86-64, x86 by Intel: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf
- Reversing a Simple Virtual Machine by Maximus: http://index-of.co.uk/Reversing-Exploiting/Reversing%20a%20Simple%20Virtual%20Machine.pdf
- Practical Reverse Engineering by Bruce Dang and Elias Bachaalany (specially chapter 5 about obfuscation with Rolf Rolles): https://www.amazon.es/Practical-Reverse-Engineering-Reversing-Obfuscation/dp/1118787315/ref=sr_1_1?__mk_es_ES=%C3%85M%C3%85%C5%BD%C3%95%C3%91&crid=2BP8VU2VOQFAG&dchild=1&keywords=practical+reverse+engineering&qid=1590947179&sprefix=Practical+Revers%2Caps%2C166&sr=8-1