by comae technologies

REVERSE ENGINEERING

Crackme Solution

Mohamed Saher (halsten)
iamhalsten@gmail.com

February 9, 2017

Abstract

This paper discusses an in-depth analysis and a solution guide for the OPCDE_ Challenge. The crackme is rather straight-forward and simple for the experienced reverser. However, for a beginner it could be problematic due to the nature of the protection and calculation method. I am going to show a couple of ways to explore how to attack it. Ranging from a pure abstract static analysis, dynamic analysis using a debugger, Concolic and Symbolic execution to Constraint solvers.

"with reverse engineering, everything is open-source"

Contents

List of Figures

List of Listings

List of Algorithms

List of Equations

List of Tables

1 Introduction

Reverse Engineering is an art involving the extraction of knowledge or design information from anything man-made and re-producing it or re-producing anything based on the extracted information. This process involves disassembling binary files to their equivalent machine code in order to analyze its components and workings in detail.

2 Prerequisites

Before we jump into the analysis and solution, the reader is expected to at least have a basic understanding of the following items:

- x86 Assembly Language.
  - General purpose registers.
  - General purpose instructions.
- x86 Architecture.

3 Outer Layer Analysis

In this section, we will examine the binary file from the outer layer only. The more we understand about the binary file and the more information we can gather, will eventually allow us to be able to solve it.

3.1 File Format Information

We have to examine the binary file format first, to look for some hints or extra information that could be useful in the analysis later on.

In figure 1, we can see the most common information regarding the binary file. Information like the linker and compiler, entry-point, number of sections and so forth could be very useful in the analysis.

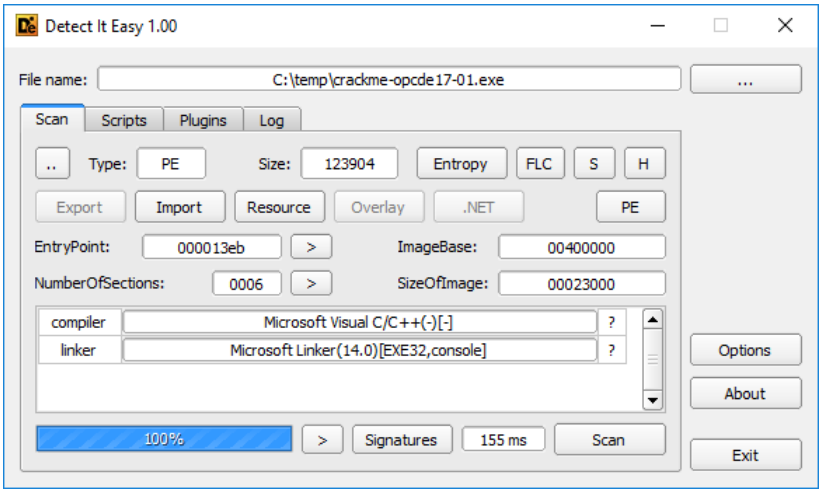

Figure 1: Main PE Structure

In figure 2, we can see the extra information about the binary file. It shows that it was compiled with a Microsoft Visual C compiler and linked with Microsoft Linker 14.0. This means that the binary file was definitely written using Visual Studio 2015. Sometimes its easy to fake this information, but so far we can trust it and look for more evidence to support such assumption.

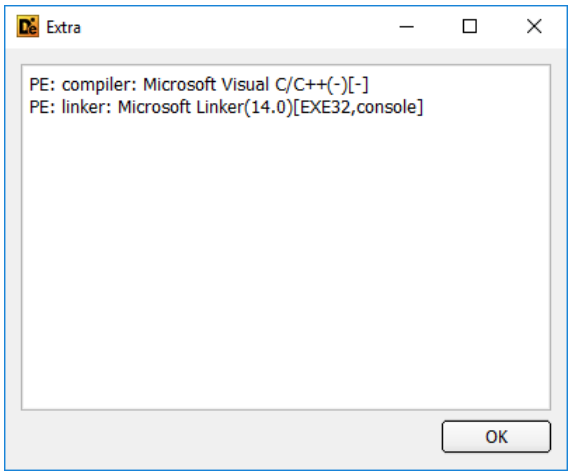

Figure 2: Linker/Compiler Signature

In figure 3, we can see the PE basic information. The `ImageBase` looks normal. There is no checksum, which is also fine. The `SizeOfImage` is rather

big, which is normal given the fact that this is a C/C++ application written under Visual Studio. It always adds a lot of extra code for the C Run-Time library and other security protections. We notice that the `Subsystem` value is `3` which means that this is a console application.

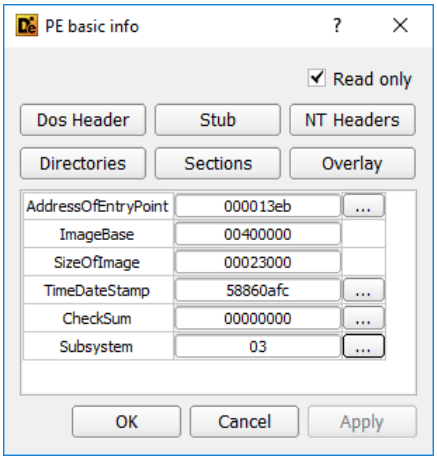

Figure 3: Basic PE Info

In figure 4, we can see the sections of the PE binary file. The values listed shows the location, size and flags of the PE binary file on disk. We can also notice the entropy value for each section. This is very important information, since it can conclude whether a section is packed or not based on how high the entropy is. It seems that there is no packing in those sections especially the `.text` section which contains the main code we are interested in analyzing.

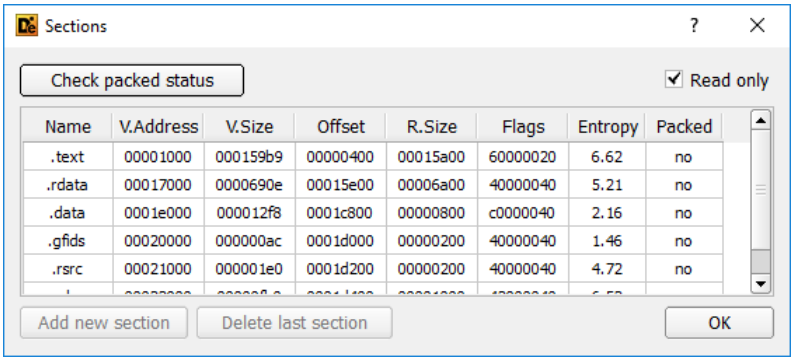

Figure 4: PE Sections

In figure 5, we can see the graph for the entropy of the whole binary file, the highlighted part is the entropy for the `.text` section only. As I have mentioned earlier, the entropy is rather high, but this is not an abnormal indicator. Overall, the entropy across the binary file is normally distributed and there are no abnormal peaks in it.

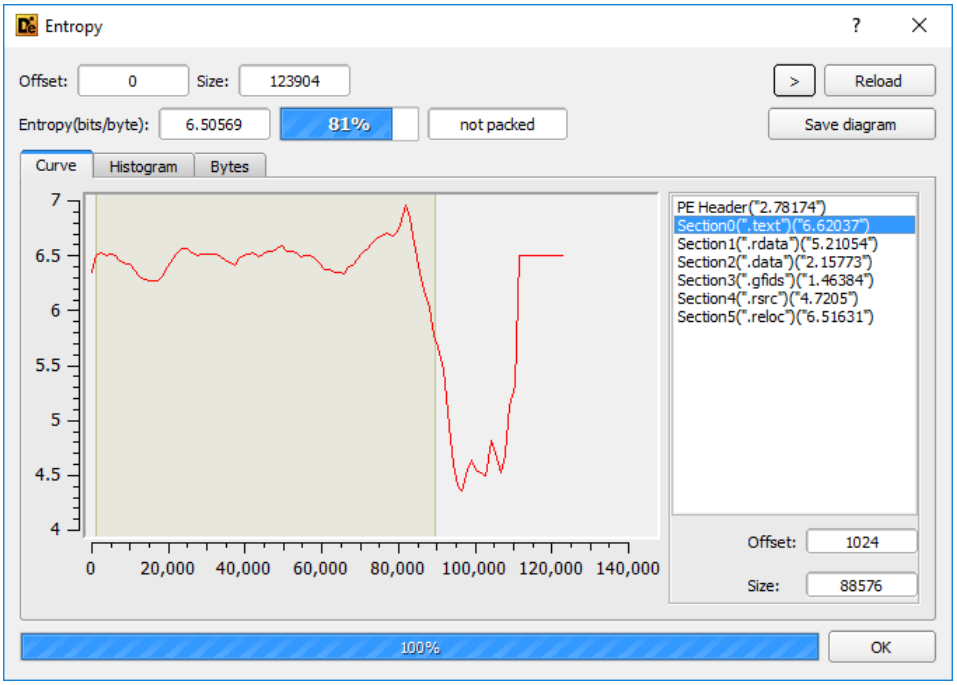

Figure 5: PE Section(s) Entropy

In figure 6, we can see the information for the file header of the PE NT headers of the binary file. We can see that the `Machine` is `014C` which means it is meant to run on `i386` or 32-bit architecture platform. There are 5 sections as we also saw before in listing 4. Pretty normal properties and values as well.

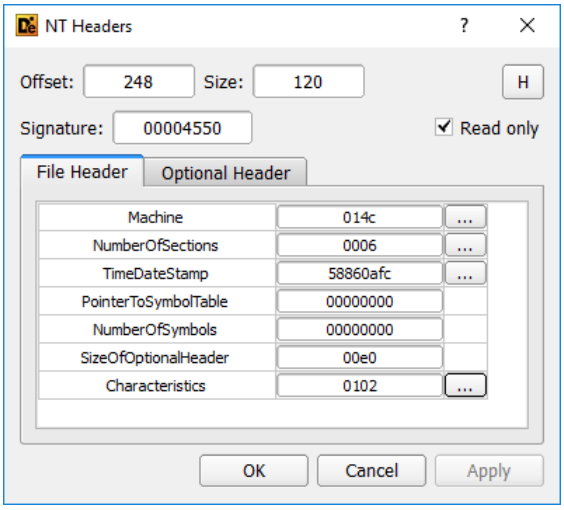

Figure 6: PE NT Header

In figure 7, we can see the information for the optional header of the PE NT headers of the binary file. We can see that the `Magic` is `010B` which means it is a 32-bit PE EXE file. Most of the other information won't be of much importance to use, since they are all virtual addresses that is relative to the code and data that resides in their appropriate sections. Things look good in this part as well.

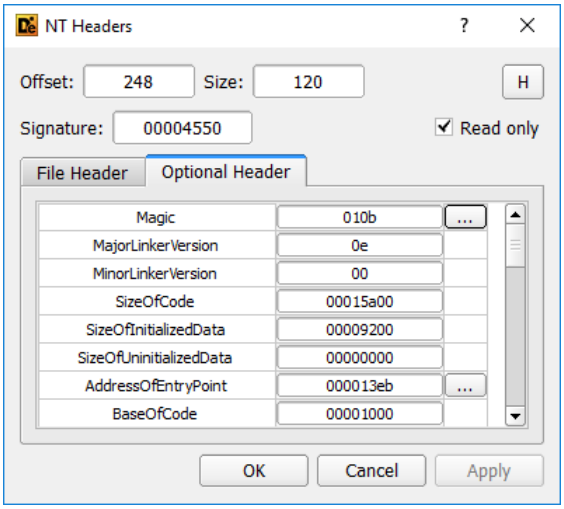

Figure 7: PE Optional Header

In figure 8, we can see the information for the imports of the PE EXE file. We can see that the crackme only imports `Kernel32` DLL. So it seems that there won't be much interaction between the crackme and other external resources. Most of the imported APIs are essentially used by the C Run-time library of Visual Studio, so we should not be bothered by that for now.

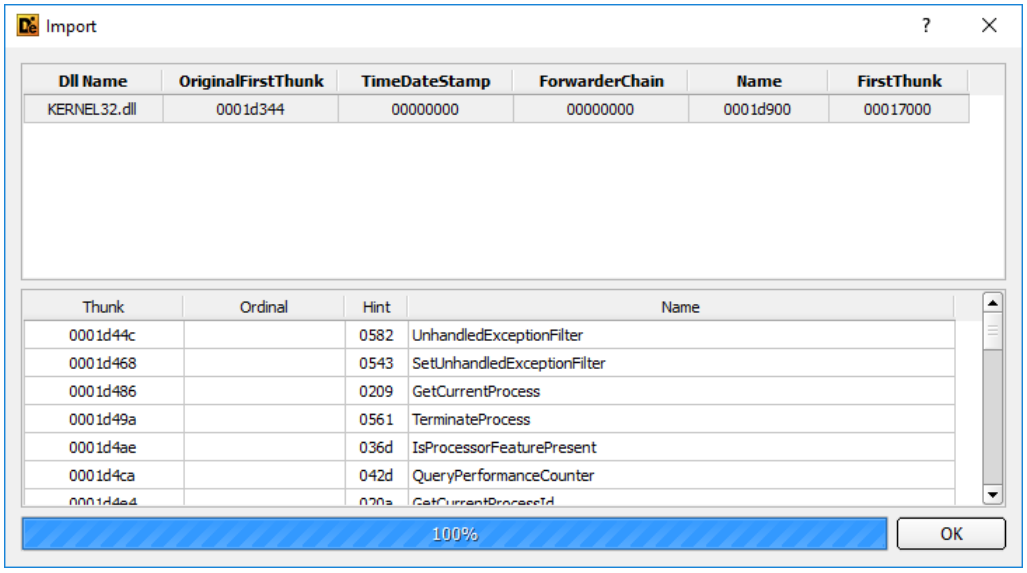

Figure 8: PE Imports

In figure 9, we can see the information for all the directories of the PE EXE file. We can see that the crackme makes use of some of the directories available in the PE EXE structure. However, we are only interested in some of them. Those interesting directories are `IMPORT`, `BASERELOC`, `DEBUG` and `LOAD_CONFIG`. The `IMPORT` is the directory that holds all of the information regarding the APIs imported by the crackme. That can be functions that are used to interact with external resources. We have already covered that in previously in figure 8. The `BASERELOC` is the directory that holds all of the information regarding base relocation addresses used by the PE EXE during run-time in memory. This information is not very important to us, since it is not going to affect our analysis at all. The `DEBUG` is the directory that holds all of the information regarding debugging symbols for the binary file. This can be very useful during the analysis because we can uncover some code that can be identified using its debugging symbols, thereby making the readability of the code later on, much easier. The `LOAD_CONFIG` is the

directory that holds all of the information regarding the crackme's initial startup code. It's important also to notice whether a `TLS` section exists or not, as of the possibility of hiding code inside that section, in which that could confuse some people while doing the analysis.

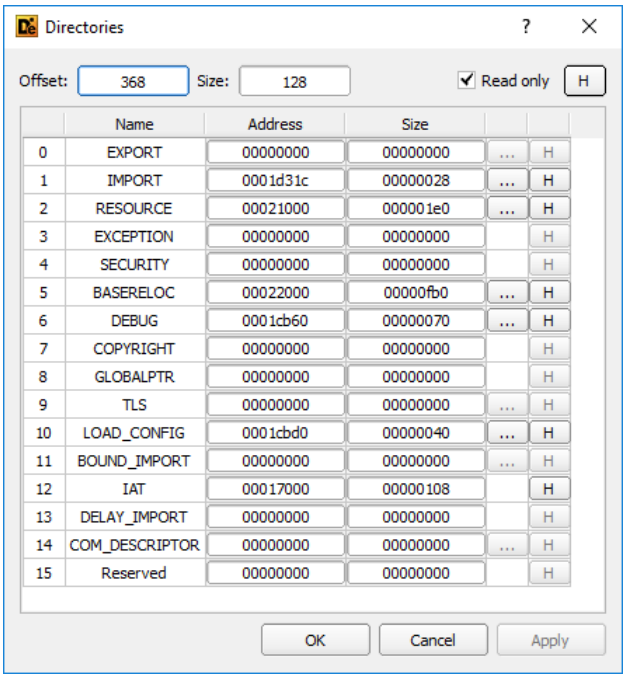

Figure 9: PE Directories

In figure 10, we can see the information for the `DEBUG` directory which contains information regarding the debugging symbols of the binary file. The debug directory is an array of `IMAGE_DEBUG_DIRECTORY` structures. These structures hold information about the type, size, and location of the various types of debug information stored in the file. Four main types of debug information appear: CodeView, 12, 13 and 14. This is just some information that is useful to know while analyzing the crackme. It seems that the binary file wasn't stripped from the debugging information that is associated with the binary file.

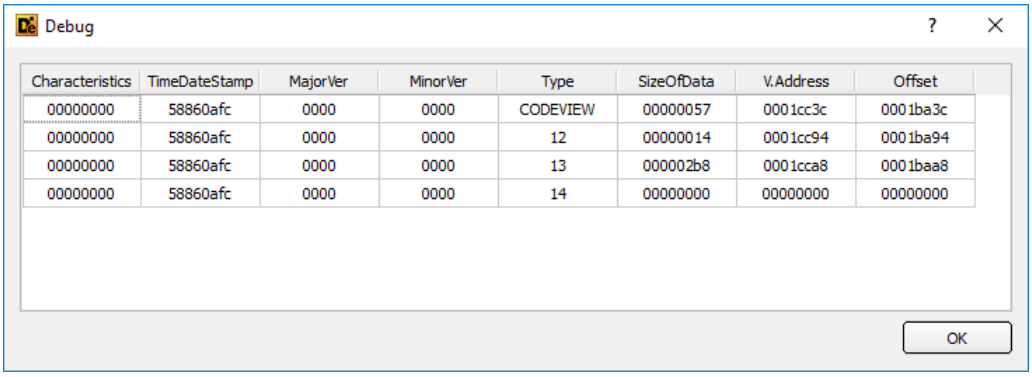

Figure 10: PE Debug Directory Info

In figure 11, we can see the information for the `TLS` (Thread Local Storage) directory which contains all of the information regarding that section of the PE EXE file. This is useful because it can cause certain code to start executing before the main startup code of the entry-point. Also it holds other information regarding `SEH` (Structured Exception Handling) which is needed for the OS in order to catch exceptions if they ever occur during the execution of the code.

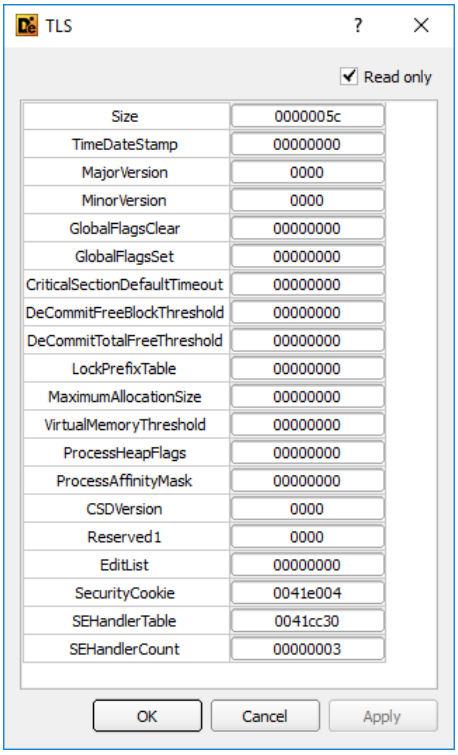

Figure 11: PE Load Config (TLS) Info

4 Inner Layer Analysis

In this section, we will examine the binary file from the inner layer. We will examine the assembly dead listing obtained by disassembling the binary file.

4.1 Disassembly

This section will show and explain the identification of various important locations throughout the code of the binary file. It is imperative that we pin point important locations to analyze so as not to waste our time nor resources with the analysis of runtime/shared libraries.

4.1.1 Startup Entry-Point

After disassembling the binary file, we look for the main entry-point as shown in listing 1 in order to start tracing through the code for some interesting locations. I have already renamed the locations, local variable name and function parameters as well so it can be understood and followed easily.

This looks very typical for a Visual C application written under Visual Studio. All we need to do right now is to find out the location for the `main`/`wmain` function so we can start analyzing the main code for the crackme. We go ahead and follow the jump located at address `004013F0` to the function `?__scrt_common_main_seh@@YAHXZ` located at address `0040127C`.

```
1 .text:004013EB                public start
2 .text:004013EB start          proc near
3 .text:004013EB                call    ___security_init_cookie
4 .text:004013EB
5 .text:004013F0                jmp     ?__scrt_common_main_seh@@YAHXZ
6 .text:004013F0
7 .text:004013F0 start          endp
```

Listing 1: Entry-Point 1

Jumping to the address `0040127C` as shown in listing 2, we come across a classic entry point for the Visual C Run-time library produced by Visual Studio. We examine the function starting with the prologue of setting up a SEH handler, following some other code for other initialization that the library might need, after that we can see the call to the `_main` function, then following that comes the end of the function with the epilogue to perform some clean-ups before exiting the function. Since we are only interested in the `_main` function, we will follow it to address `004010F0`.

```
.text:0040127C                  push    14h
.text:0040127E                  push    offset stru_41CF70
.text:00401283                  call    __SEH_prolog4
.text:00401283
.text:00401283                  [...]
.text:00401283
.text:0040136B                  push    eax             ; envp
.text:0040136C                  push    dword ptr [edi] ; argv
.text:0040136E                  push    dword ptr [esi] ; argc
.text:00401370                  call    _main
.text:00401370
.text:00401370                  [...]
.text:004013E5
.text:004013E5                  call    __SEH_epilog4
.text:004013E5
.text:004013EA                  retn
```

Listing 2: Entry-Point 2

The standard definition for the `main` function in C language is shown in listing 3. We can use this definition in uncovering the parameters passed to the function as well as the manipulation of those parameters inside the function.

```
int __cdecl main(int argc, const char **argv, const char **envp)
```

Listing 3: Standard C Entry-Point Definition

4.1.2 main/wmain Entry-Point

Arriving to address `004010F0`, we find the `_main` function. Starting from this point, we could consider to begin the analysis of the main logic of the crackme.

Listing 4 shows the parameters passed to the function as well as the variables being used internally. As defined earlier in listing 3 we can see the three parameters `argc`, `argv` and `envp`. We also see that the function is using the two variables `var_4` and `var_8` internally.

```
.text:004010F0 _main           proc near
.text:004010F0
.text:004010F0 var_8           = dword ptr -8
.text:004010F0 var_4           = dword ptr -4
.text:004010F0 argc            = dword ptr  8
.text:004010F0 argv            = dword ptr  0Ch
.text:004010F0 envp            = dword ptr  10h
.text:004010F0
.text:004010F0 [...]
```

Listing 4: wmain 1

In listing 5, address `004010F0` marks the initial entry point for `_main` function. It starts by saving the current base pointer and allocating space on the stack for the variables. Some code from the Visual C Run-time library is observed for security purposes. We can neglect that part as it has no value in the analysis of the crackme.

Starting from address `00401104` to address `00401110`, the crackme simply prints the first banner to the console. Moving forward from address `00401115` to address `0040111A`, the crackme prints out the second banner to the console. Those are just simple instructions to the reverser and a little bit of copyright to the crackme associated with the conference OPCDE_.

A comparison at address `00401122` and address `00401126` is performed to check whether the user has passed 2 or more arguments to the crackme or not. Therefore, a check against `argc` is carried out and a decision is then made to jump to address `0040114B` if the comparison yielded `TRUE` or jump continue execution the following address `00401128`. If the comparison yielded `FALSE`. In the case of the comparison resulting in `TRUE`, then the crackme moves on to the following part where it should start its core logic. However, in the case of the comparison resulting in `FALSE`, then the crackme will print out to the console the instructions needed in order to execute it properly. This is noticed between address `00401128` and address `00401132`. At this point, the crackme simply performs some clean-ups and then exits.

```
.text:004010F0                push    ebp
.text:004010F1                mov     ebp, esp
.text:004010F3                sub     esp, 8
.text:004010F6                mov     eax, ___security_cookie
.text:004010FB                xor     eax, ebp
.text:004010FD                mov     [ebp+var_4], eax
.text:00401100                push    esi
.text:00401101                mov     esi, [ebp+argv]
.text:00401104                push    offset aOpcde_2017Crac
.text:00401109                mov     [ebp+var_8], 0
.text:00401110                call    sub_401020
.text:00401110
.text:00401115                push    offset aHttpWww_opcde_
.text:0040111A                call    sub_401020
.text:0040111A
.text:0040111F                add     esp, 8
.text:00401122                cmp     [ebp+argc], 2
.text:00401126                jge     short loc_40114B
.text:00401126
.text:00401128                push    offset aChallenge_exe
.text:0040112D                push    offset aUsageSCode
.text:00401132                call    sub_401020
.text:00401132
.text:00401137                add     esp, 8
.text:0040113A                xor     al, al
.text:0040113C                pop     esi
.text:0040113D                mov     ecx, [ebp+var_4]
.text:00401140                xor     ecx, ebp
.text:00401142                call    @__security_check_cookie@4
.text:00401142
.text:00401147                mov     esp, ebp
.text:00401149                pop     ebp
.text:0040114A                retn
.text:0040114A
.text:0040114A                [...]
```

Listing 5: wmain 2

Earlier, we mentioned that there is a comparison performed on `argc` which will require the user to input an argument and pass it to the crackme from the command prompt. Given such requirement for the `TRUE` condition (branch) to succeed, this leads us to the core logic of the crackme.

In listing 6 starting from address `0040114E` to address `00401157` will pass the first argument in the array `argv` (`argv[1]`), the string "%x" (string format denoting that it is in a hexadecimal format) and finally `eax` (will hold the value of the output of the function `ConvertHexStrToHexDWORD`). It was relatively easy to actually guess what the function does based on the parameters passed to it. Hence, that is why this function was named to `ConvertHexStrToHexDWORD`. Therefore, we can understand that the crackme expects us to input a string as the argument passed to it from the command prompt. This string will require to be some sort of a hexadecimal value in which the calculation of the magic code will definitely depend on it.

In the next code block that starts from address `0040115C` to address `0040115F`, the function `VM_EXECUTE` will take the value returned from the previous function `ConvertHexStrToHexDWORD` as a parameter (referenced by `eax`). This function was named to `VM_EXECUTE` after some analysis was carried out that showed that the crackme has an internal VM empowering some processing that will affect the comparison and calculation of such magic code. We will investigate this function and its internals more thoroughly later on.

Moving forward to the next code block starting from address `00401167` to address `00401176`, a comparison is carried out between the result from the previous function `VM_EXECUTE` and the value `0FF37F33Bh`. This is a clear hint that the previous function `VM_EXECUTE` is in fact the one responsible for the calculation of the magic code. At this point a decision has to be made whether to print out a success message to the reverser on the command prompt denoting that this is the `TRUE` condition or print out a failed message to the reverser on the command prompt denoting that this is the `FALSE` condition. The `TRUE` condition will continue the execution of the crackme to the following address `0040116E`, while the `FALSE` condition will jump out of this code block and change the execution of the crackme to the following address `0040118F`.

The remaining code block starting from address `0040117B` to address `0040118E` basically adjusts the stack and ensures that the security check cookie of the function before it returns was correct (this is a security feature) and then simply returns or exits.

```
.text:0040114B                  [...]
.text:0040114B
.text:0040114B loc_40114B:
.text:0040114B                  lea     eax, [ebp+var_8]
.text:0040114E                  push    eax
.text:0040114F                  push    offset asc_41C9DC
.text:00401154                  push    dword ptr [esi+4]
.text:00401157                  call    ConvertHexStrToHexDWORD
.text:00401157
.text:0040115C                  push    [ebp+var_8]
.text:0040115F                  call    VM_EXECUTE
.text:0040115F
.text:00401164                  add     esp, 10h
.text:00401167                  cmp     eax, 0FF37F33Bh
.text:0040116C                  jnz     short loc_40118F
.text:0040116C
.text:0040116E                  push    dword ptr [esi+4]
.text:00401171                  push    offset aGoodboy_Please
.text:00401176                  call    sub_401020
.text:00401176
.text:0040117B                  add     esp, 8
.text:0040117E                  mov     al, 1
.text:00401180                  pop     esi
.text:00401181                  mov     ecx, [ebp+var_4]
.text:00401184                  xor     ecx, ebp
.text:00401186                  call    @__security_check_cookie@4
.text:00401186
.text:0040118B                  mov     esp, ebp
.text:0040118D                  pop     ebp
.text:0040118E                  retn
.text:0040118E
.text:0040118E                  [...]
```

Listing 6: wmain 3

In listing 6, there was a comparison that was performed and based on the result from such comparison a decision to branch was made. This code block is the `FALSE` condition for such comparison. It denotes that the reverser has entered the wrong magic code and can still have a chance in buying a ticket from the website.

```
.text:0040118F                  [...]
.text:0040118F
.text:0040118F loc_40118F:
.text:0040118F                  push    offset aYouCanStillBuy
.text:00401194                  call    sub_401020
.text:00401194
.text:00401199                  mov     ecx, [ebp+var_4]
.text:0040119C                  add     esp, 4
.text:0040119F                  xor     ecx, ebp
.text:004011A1                  mov     al, 1
.text:004011A3                  pop     esi
.text:004011A4                  call    @__security_check_cookie@4
.text:004011A4
.text:004011A9                  mov     esp, ebp
.text:004011AB                  pop     ebp
.text:004011AC                  retn
.text:004011AC
.text:004011AC _main            endp
```

Listing 7: wmain 4

That's it for the analysis of the `_main` function. In figure 12, we can summarize the call-flow-graph (CFG) to have a better view of what is going on inside the function. We can also clearly see the decision branches taken from the two comparisons that were made for both `argc` and `eax`, which are the count of arguments passed to the crackme from the command prompt and the value returned from the function `VM_EXECUTE` respectively.

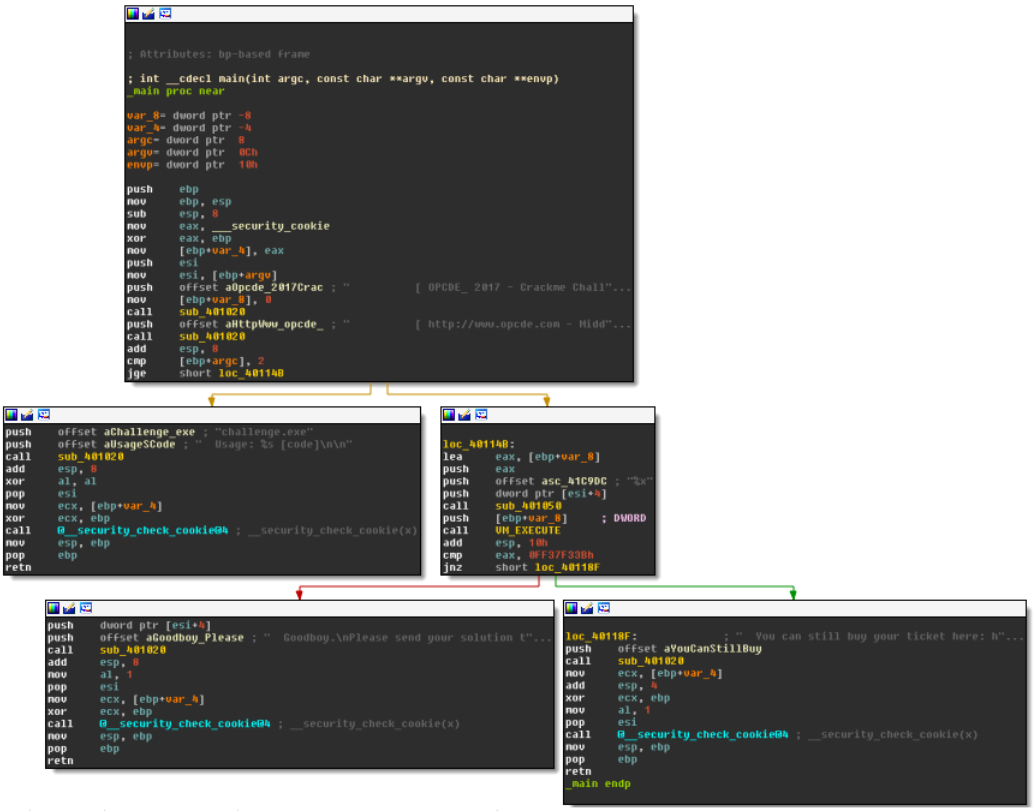

Figure 12: main CFG

4.1.3 VM Interpreter/Dispatcher Entry-Point

In listing 8, we can see that the function expects only one parameter and it shows the size of such parameter, so we can go ahead and assume that this should be a double-word value, since the comparison that was earlier made in listing 6 at address `00401167` was also a double-word.

```
1 .text:00401080 VM_EXECUTE      proc near
2 .text:00401080
3 .text:00401080 arg_0           = dword ptr  8
4 .text:00401080
5 .text:00401080 [...]
```

Listing 8: VM_EXECUTE 1

In listing 9, we can see the main logic of `VM_EXECUTE` function. This part is the core of the problem. It is imperative to understand how the VM interpreter handles each instruction and locate each dispatcher for each handler. This process is the same process as what a regular CPU does. Now, custom VMs will be different in design and operation. However, they will all still share similarities from the following list:

- **Fetch the instruction**: The next instruction is fetched from the memory address that is currently stored in the program counter (PC), and stored in the instruction register (IR). At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.
- **Decode the instruction**: During this cycle the encoded instruction present in the IR (Instruction Register) is interpreted by the decoder.
- **Read the effective address**: In case of a memory instruction (direct or indirect) the execution phase will be in the next clock pulse. If the instruction has an indirect address, the effective address is read from main memory, and any required data is fetched from main memory to be processed and then placed into data registers (Clock Pulse: T_3). If the instruction is direct, nothing is done at this clock pulses. If this is an I/O instruction or a Register instruction, the operation is performed (executed) at clock Pulse.
- **Execute the instruction**: The control unit of the CPU passes the decoded information as a sequence of control signals to the relevant function units of the CPU to perform the actions required by the instruction such as reading values from registers, passing them to the ALU to perform mathematical or logic functions on them, and writing the result back to a register. If the ALU is involved, it sends a condition signal back to the CU. The result generated by the operation is stored in the main memory, or sent to an output device. Based on the condition of any feedback from the ALU, Program Counter may be updated to a different address from which the next instruction will be fetched.

Having said that, we can then examine the VM's interpreter to understand how we can de-construct it internals. Once that is done, we can move

on to reversing its process and writing a disassembler to disassemble the VM's P-CODE into readable instructions. Starting from this point is where the real analysis begins. This is where we are going to find out how the magic code is being calculated and by turn we can work on reversing the algorithm in order to find the magic code.

In listing 9 at address `00401090`, we can see that the first byte of the P-CODE is being read and stored in `EAX` with a zero-extend preservation, which means that the data is moved from a smaller register into a bigger register, and the sign is ignored. Then we compare value stored in `EAX` with `10h`. If the comparison yields a `TRUE` condition, then it jumps to `loc_4010AA` located at address `004010AA`. However, if the comparison yields a `FALSE` condition, then it continues execution to the next instruction located at address `0040109C`. We derive from this part that the byte `10h` is in fact an opcode and the handler for that opcode is `loc_4010BF` located at address `004010BF`. We will analyze each opcode and its handler in its own part as we follow.

Next, we have another comparison between `EAX` with the value `1h`. If the comparison yields a `TRUE` condition, then it jumps to `loc_4010AA` located at address `004010AA`. However, if the comparison yields a `FALSE` condition, then it continues execution to the next instruction located at address `004010A1`. An astute reader will realize that there is a small trick here. The trick is that this small portion of the VM's interpreter code was trying to identify another opcode with the value of `11h`. We can find that out by easily starting forward from address `00401097` and going downwards till address `0040109C`. All we need to do is find the right value that won't satisfy the constraints of the first `SUB` instruction. So we have two options, either start with a higher number than `10h`, or start with a number lower than `10h`. If we assumed that the value of `EAX` was `11h`, this will store the value `10h` in `EAX`, which won't satisfy the conditional jump `JZ` to be taken. Then the second `SUB` operation is encountered and it subtracts `1h` from `EAX` which holds the value `10h` currently. This will result in storing the value `0h` in `EAX`, which will satisfy the constraint to jump to its own handler `loc_4010AA` located at address `004010AA`. This means that if we chose to store a smaller value in `EAX` than `10h`, this will result in having a negative value in `EAX`, which doesn't satisfy our constraints at all. We derive from this part that the byte `11h` is in fact an opcode and the handler for that opcode is `loc_4010AA` located at address `004010AA`. We will analyze each opcode and its handler in its own part as we follow.

The following code segment located between address 004010A1 and address 004010A8 will basically check for the lower part of EAX for the value 0EEh, since at address 00401090 there was a MOVZX instruction which was overridden by the BYTE cast. Therefor, the lower part of EAX will hold the value of the byte pointed to by ECX while the higher part of EAX will be zero'ed out. For example, the value of 01000h will become 00001000h and so forth. This comparison exactly the same as the one previously discussed. this small portion of the VM's interpreter code was trying to identify another opcode with the value of FFh. We can find that out by easily starting forward from address 00401097 and going downwards till address 004010A1. All we need to do is to find the right value that won't satisfy the constraints of the first SUB instruction located at address 00401097 and the second SUB instruction located at address 0040109C. We can also just add all three values 10h, 1h and 0EEh all together to get the value of the opcode we are looking for. Adding all three values together will output the value FFh. If we try out the value FFh on all three SUB instructions located at addresses 00401097, 0040109C and 004010A1 respectively, we will find out that the constraint is indeed satisfied and the conditional jump JZ can be taken to its own handler loc_4010D8 located at address 4010D8. We will analyze each opcode and its handler in its own part as we follow.

The remaining code block located between address 004010DC and address 004010E2 will simply be called only if the VM's interpreter encountered something wrong in the VM's P-CODE, resulting in returning a rather funny result which is the value of DEADBEEFh being stored in EAX and returned back by the function. Typically, we shouldn't hit this constraint unless the P-CODE was patched incorrectly. So the author took care of the patching path. This shouldn't be a problem to a more experienced reverser though. However, I find it better to just analyze the VM's interpreter and disassemble the P-CODE then reverse it, other than try to patch the P-CODE itself. Its more elegant and clean. Aside from the fact that more advanced and complex VMs should have the ability to protect against being patched or even detecting the presence of patched code, but this is not the case here and the author clearly stated that patching is not allowed.

```
.text:00401090                  [...]
.text:00401090
.text:00401090 loc_401090:
.text:00401090                  movzx   eax, byte ptr PCODE[ecx]
.text:00401097                  sub     eax, 10h
.text:0040109A                  jz      short loc_4010BF
.text:0040109A
.text:0040109C                  sub     eax, 1
.text:0040109F                  jz      short loc_4010AA
.text:0040109F
.text:004010A1                  sub     eax, 0EEh
.text:004010A6                  jz      short loc_4010D8
.text:004010A6
.text:004010A8                  jmp     short loc_4010D2
.text:004010A8
.text:004010AA loc_4010AA
.text:004010AA                  [...]
.text:004010BF loc_4010BF:
.text:004010BF                  [...]
.text:004010D2 loc_4010D2:
.text:004010D2                  [...]
.text:004010D8 loc_4010D8:
.text:004010D8                  [...]
.text:004010DC loc_4010DC:
.text:004010DC                  mov     eax, 0DEADBEEFh
.text:004010E1                  pop     ebp
.text:004010E2                  retn
```

Listing 9: VM_EXECUTE 2

We jumped to location `loc_4010AA` located at address `004010AA` as shown in listing 10. In the previous code block the dispatcher assigned this handler for the opcode `11h`. A byte is read from the position where `ECX` was pointing to previously, and stored in `AL`. We increment `ECX` by two bytes, basically to point `ECX` to the next byte in the P-CODE. We can deduct from this point that `ECX` is the program counter (PC) mentioned earlier. Then `AL` is compared with the value `6h` to check the length of the instruction. We will discuss this later in more detail in a separate section.

After that, `EDX` is subtracted by the double-word (DWORD) value read from `ECX`, which seems to be an *Immediate* (constant) value. We can deduct from this that `EDX` holds the value at which the magic code is being calculated

from and also that it holds the initial value to start with. So far, we have obtained some good information on how the VM is working internally. We move forward to the last unconditional jump `JMP` located at address `004010BD` which will surely have to prepare the internal state of the CPU to increment the program counter (PC) assigned to by `ECX` and so on. This was well previously explained on how CPUs work internally. We will also examine that part later on.

```
1 .text:004010AA loc_4010AA:
2 .text:004010AA                  mov     al, byte ptr (VM_PCODE+1)[ecx]
3 .text:004010B0                  add     ecx, 2
4 .text:004010B3                  cmp     al, 6
5 .text:004010B5                  jnb     short loc_4010DC
6 .text:004010B5
7 .text:004010B7                  sub     edx, VM_PCODE[ecx]
8 .text:004010BD                  jmp     short loc_4010D2
```

Listing 10: VM_EXECUTE 3 (SUB Handler)

We jumped to location `loc_4010BF` located at address `004010BF` as shown in listing 11. In listing 9 the dispatcher's code block assigned this handler for the opcode `10h`. A byte is read from the position where `ECX` was pointing to previously, and stored in `AL`. We increment `ECX` by two bytes, basically to point `ECX` to the next byte in the P-CODE. We can deduct from this point that `ECX` is the program counter (PC) mentioned earlier. Then `AL` is compared with the value `6h` to check the length of the instruction. We will discuss this later in more detail in a separate section.

After that, `EDX` is added with the double-word (DWORD) value read from `ECX`, which seems to be an *Immediate* (constant) value. We can deduct from this that `EDX` holds the value at which the magic code is being calculated from and also that it holds the initial value to start with. So far, we have obtained some good information on how the VM is working internally. We move forward to the code segment located between address `004010D2` and address `004010D6`, we find that the internal state of the CPU is being prepared by incrementing the program counter (PC) assigned to by `ECX`. Then it is being compared with the value `19h` which is the value `25` in decimal format. If we actually follow the address of `VM_PCODE` inside the binary file, we can find that the VMs P-CODE consists of `19h` (25d) bytes only (ending with `FFh`). This

shows that the VM's interpreter is checking to see if it has finished iterating over the VM's P-CODE or not so it can either continue or stop execution. At the point of `ECX` being less than `19h` (25d) then the VM's interpreter will reset the state of the CPU again and jump to location `loc_401090` located at address `00401090` to repeat this process once again till it finishes interpreting all of the VM's P-CODE.

```
.text:004010BF loc_4010BF:
.text:004010BF                  mov     al, byte ptr (VM_PCODE+1)[ecx]
.text:004010C5                  add     ecx, 2
.text:004010C8                  cmp     al, 6
.text:004010CA                  jnb     short loc_4010DC
.text:004010CA
.text:004010CC                  add     edx, VM_PCODE[ecx]
.text:004010CC
.text:004010D2
.text:004010D2 loc_4010D2:
.text:004010D2                  inc     ecx
.text:004010D3                  cmp     ecx, 19h
.text:004010D6                  jb      short loc_401090
```

Listing 11: VM_EXECUTE 4 (ADD Handler)

In listing 12, we can see the same code segment we have previously analyzed from listing 11. So we don't have to repeat the analysis, however, this part can be deduced as the "move-forward" or "continue" iterating and validating the execution of the VM's P-CODE.

```
.text:004010D2 loc_4010D2:
.text:004010D2                  inc     ecx
.text:004010D3                  cmp     ecx, 19h
.text:004010D6                  jb      short loc_401090
```

Listing 12: VM_EXECUTE 5

We jumped to location `loc_4010D8` located at address `004010D8` as shown in listing 13. In listing 9 the dispatcher's code block assigned this handler for the opcode `FFh`. It seems that this handler does nothing except store the

value of the calculated magic code inside `EAX` and then the function returns the magic code in `EAX` as well. So we can deduct from this point that the opcode `FFh` is in fact a `RET` instruction.

```
.text:004010D8 loc_4010D8:
.text:004010D8                 mov     eax, edx
.text:004010DA                 pop     ebp
.text:004010DB                 retn
```

Listing 13: VM_EXECUTE 6 (RET Handler)

In listing 14, we can see the following code segment, which we previously briefly discussed as being the error landing location on whether the instruction length was greater than the value of `6h` bytes. This code segment will simply get called only if the VM's interpreter encountered something wrong in the VM's P-CODE, resulting in returning a rather funny result which is the value of `DEADBEEFh` being stored in `EAX` and returned back by the function.

```
.text:004010DB
.text:004010DC loc_4010DC:
.text:004010DC                 mov     eax, 0DEADBEEFh
.text:004010E1                 pop     ebp
.text:004010E2                 retn
.text:004010E2
.text:004010E2 VM_EXECUTE      endp
```

Listing 14: VM_EXECUTE 7

In figure 13, we can notice the branching and different nodes of which several comparison instructions are being processed and executed as well as the appropriate opcode handler for such opcodes. In most cases the CFG of the VM's interpreter has a flat CFG so its easy to identify it while analyzing the code. Since there is a lookup table of some sort ora switch case. Therefor, we will be able to see all handlers on the same level (in terms of depth) in the CFG.

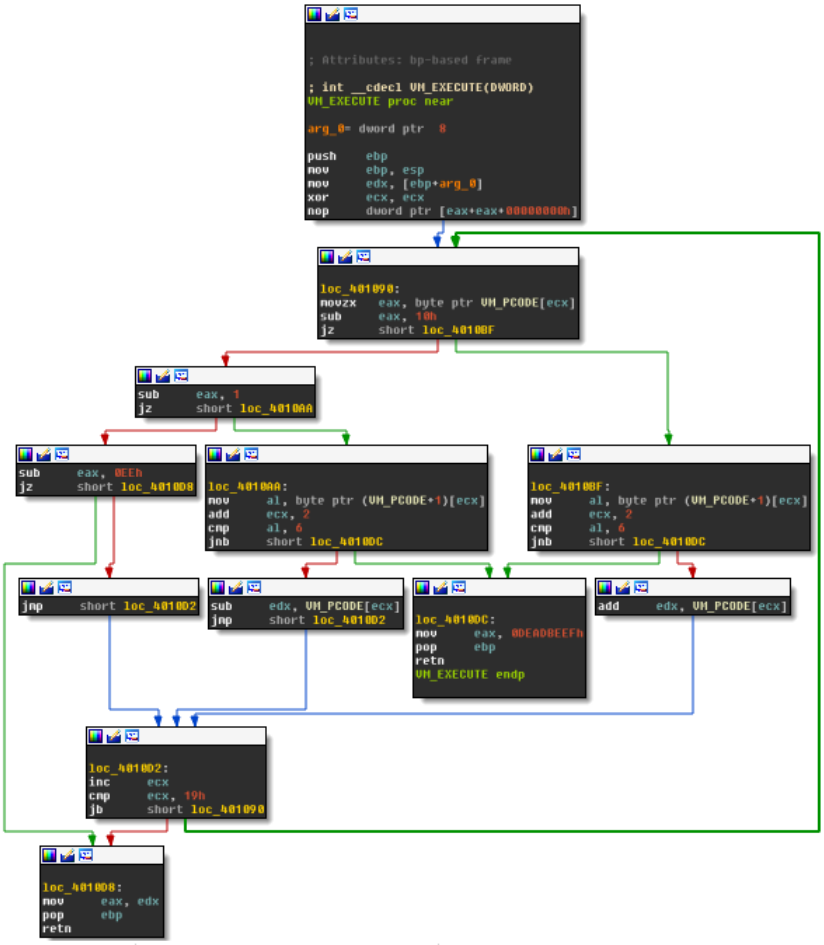

Figure 13: VM Dispatcher CFG

After examining the previous CFG in figure 13 and the disassembled code in listing 9, we can actually define an exact replica of the algorithm of the VM's interpreter as it was reversed (not optimized), which is shown in algorithm 1.

Algorithm 1: VM's Interpreter and Dispatcher

Input: PCODE array P, Initialized register R
Result: Integer m

```
1  PC ⇐ 0
2  RI ⇐ R
3  RV ⇐ 0
4  while P[PC ← PC + 1] ≠ 16 do
5  |  if P[PC] is 17 then
6  |  |  RI ← P[PC + 1]
7  |  |  PC ← PC + 2
8  |  |  if RI ⩾ 6 then
9  |  |  |  return 3735928559
10 |  |  RV ← RV - P[PC]
11 |  else if P[PC] is 255 then
12 |  |  return RV;
13 |  if (PC ← PC + 1) ⩾ 25 then
14 |  |  return RV
15 end
16 RI ← P[PC + 1]
17 PC ← PC + 2
18 if RI < 6 then
19 |  RV ← RV + P[PC]
20 |  if (PC ← PC + 1) ⩾ 25 then
21 |  |  return RV
22 |  return 3735928559
```

In listing 15, we can see the actual table for the VM's P-CODE. As we discussed earlier, it should be a byte array as of the way the VM's interpreter parses it. We should not care on how the disassembler has actually decided to interpret them in terms of display purposes.

```
.data:0041E7B0 VM_PCODE        dd 2010010h
.data:0041E7B4                 dd 110403h
.data:0041E7B8                 dd 1A7895Eh
.data:0041E7BC                 dd 2FE90011h
.data:0041E7C0                 dd 100F83h
.data:0041E7C4                 dd 31B98692h
.data:0041E7C8                 db 0FFh
```

Listing 15: VM P-CODE

4.2 VM Analysis

This section will explain the process in identifying and analyzing VM based code protection. Although, due to the nature of this challenge, not all parts of a realistic code virtualizer exists. This is because the author was nice enough to not make it complicated for students initially. However, it's a good idea to still talk about it as it serves a good educational reference for anyone who wishes to get into such challenges.

Typically, in code virtualizers, there are core components that are required to exist in order for the virtualized code execution to work properly. Those core components are explained to a certain detail level in the following sub-sections.

4.2.1 VM Architecture

Just like any traditional x86/x64 micro-processor chip-set, a custom VM requires almost the same architecture and components that exists inside such processor. The process of designing a custom VM involves a lot of knowledge about micro-processors. Therefore, hand crafting a custom VM is not only a tedious and cumbersome job, but also requires a great deal of attention to small details that affect the execution cycle(s), branching, or the micro-processor state itself.

Luckily this VM architecture is extremely simple and straight-forward, probably due to the nature of the targeted audience. The following subsections will explain each component of the VM.

4.2.2 VM Instruction Format

A custom VM will implement its own custom instruction encoding format. This is up to the author to decide while designing the VM in the first place. But, there are certain universal standards that have to be present and exist in the instruction encoding format in order to construct a proper one.

Each instruction will be 12 bytes in size. The first byte will represent the opcode. The second byte will represent the register number (we only have 1 register though). The remaining 4 bytes (double-word) will represent and *Immediate* value. Figure 14 illustrates such encoding format.

Figure 14: VM Instruction Encoding

4.2.3 VM Instructions

The VM consists only of three instructions, the `ADD`, the `SUB` and `RET` instruction. The algorithm used in calculating the magic code will depend on the usage of those three instructions.

That shouldn't be difficult at all. Given the fact that this is just a simple crackme, we should be okay in finding the algorithm and reversing it. List 15 and 16 shows the syntax, semantics and the proper usage of the VM instructions.

Instruction: ADD

Syntax:	`ADD <reg>,<Imm>`
Semantics:	The ADD instruction adds the data item referred to by its second operand with the data item referred to by the first operand, saving it back into the location referred to by its first operand.
Examples:	`ADD REG1, 12345678h` ; REG1 += 12345678h

Figure 15: VM Instruction: ADD

Instruction: SUB

Syntax:	`SUB <reg>,<Imm>`
Semantics:	The SUB instruction subtracts the data item referred to by its second operand with the data item referred to by the first operand, saving it back into the location referred to by its first operand.
Examples:	`SUB REG1, 12345678h` ; REG1 -= 12345678h

Figure 16: VM Instruction: SUB

Instruction: RET

Syntax:	`RET`
Semantics:	The RET instruction returns back the control flow and exits the current procedure. It could also be an indicator to halt the CPU.
Examples:	`RET` ; return

Figure 17: VM Instruction: RET

4.2.4 VM P-CODE

We discussed earlier that the VM needs to have its own code to interpret. This is idea of implementing a custom VM in the first place. We learned about the construction of the VM earlier with respect to a real CPU. The P-CODE is the same as assembly language compiled, but in its own encoding format which follows the architecture or construction of the custom VM. As previously shown in listing 15, we can observe how a VM's P-CODE looks like.

4.2.5 VM Interpreter/Dispatcher

The VM's interpret is the brain of the VM in general. It is what makes it work. As we have seen previously in listing 9, the VM's interpret iterates over its P-CODE and starts to decode such array in a linear form. The interpreter then starts to decode each byte in the P-CODE array in order to find out each of it's opcodes and then dispatches it to its appropriate handler. The handler then starts to mimic the operation of an x86 assembler code according to the opcode or operation that is undergoing at that moment. This process is

repeated until it reaches the opcode where it tells the VM's interpreter to stop fetching more instructions or just halt.

4.2.6 VM Disassembler

In order to understand the VM's P-CODE, we need to write a disassembler that can decode it into meaningful instructions so that we can understand accordingly. We are basically re-creating the code of the VM's interpreter/dispatcher that was found earlier in the binary file. We just need to re-write it in a High Level Language. Algorithm 2 will demonstrate what the disassembler algorithm should look like, while listing 16 will show what the disassembler could be written like.

Algorithm 2: VM's Disassembler

```
   Input: P-CODE array P
   Result: Disassembled P-CODE
 1 for PC ← 0 to PC ≤ getArraySize(P) do
 2 |   if P[PC] is 16 then
 3 |   |   PC ← PC + 1
 4 |   |   RI ← P[PC ← PC + 1]
 5 |   |   *R ← &P[PC]
 6 |   |   RV ← *R
 7 |   |   print "add reg1, " + RV
 8 |   else if P[PC] is 17 then
 9 |   |   PC ← PC + 1
10 |   |   RI ← P[PC ← PC + 1]
11 |   |   *R ← &P[PC]
12 |   |   RV ← *R
13 |   |   print "sub reg1, " + RV
14 |   else if P[PC] is 255 then
15 |   |   print "ret"
16 end
```

```
#include <stdio.h>
#include <stdlib.h>

#define UBOUND(x) (sizeof(x) / sizeof(x[0]))

uint8_t PCODE[] = {
    0x10, 0x0, 0x1, 0x2, 0x3, 0x4, 0x11, 0x0, 0x5E,
    0x89, 0x0A7, 0x1, 0x11, 0x0, 0x0E9, 0x2F, 0x83,
    0x0F, 0x10, 0x0, 0x92, 0x86, 0x0B9, 0x31, 0xFF
};

int main(int argc, const char * argv[]) {
    for (uint8_t Offset = 0; Offset <= UBOUND(PCODE); Offset++) {
        if (PCODE[Offset] == 0x10) {
            Offset++;
            u_long ulRegIndex = PCODE[Offset++];
            uint32_t *ulReg = (uint32_t *)&PCODE[Offset];
            uint32_t ulImm = *ulReg;

            fprintf(stdout, "add reg1, %Xh\n", ulImm);
        }
        else if (PCODE[Offset] == 0x11) {
            Offset++;
            u_long ulRegIndex = PCODE[Offset++];
            uint32_t *ulReg = (uint32_t *)&PCODE[Offset];
            uint32_t ulImm = *ulReg;

            fprintf(stdout, "sub reg1, %Xh\n", ulImm);
        }
        else if (PCODE[Offset] == 0xFF) {
            fprintf(stdout, "ret\n");
        }
    }

    return EXIT_SUCCESS;
}
```

Listing 16: VM Disassembler

5 Possible Solutions

In the following sub-sections, I will try to demonstrate 3 different methods in solving this crackme challenge, although, it is not really necessary. However I feel this can be used as a good educational reference in the future for anyone who is trying to get into solving VM based protection crackme challenges. In reality, sometimes a combination of all three solutions might be required to solve more complex crackme challenges. Fortunately, this is not the case.

5.1 Abstract Static Analysis

In this solution we will attempt to break the protection and figure out the correct input parameters for the binary file without having to run or debug it in the first place. This will heavily rely on our solid foundation and understanding of the Assembly language and its underlying architecture.

Earlier we stopped at writing the disassembler for the VM in order to disassemble the P-CODE. Continuing to execute the code in listing 16, we will get the disassembly listing of the VM's P-CODE. Listing 17 will show such output.

```
1 add    reg1, 4030201h
2 sub    reg1, 1A7895Eh
3 sub    reg1, F832FE9h
4 add    reg1, 31B98692h
5 ret
```

Listing 17: VM Disassembled P-CODE

Excellent progress so far, and at this point we can clearly see the algorithm used to calculate the magic code. All we have to do now is to reverse the algorithm to compute the right magic code. We already know that from listing 6 at address `00331167`, there was a comparison with the value `FF37F33Bh` which was used for the computed magic code. So we should work our way from the bottom up. Listing 18 will show the reversed algorithm.

```
1 sub    reg1, 4030201h
2 add    reg1, 1A7895Eh
3 add    reg1, F832FE9h
4 sub    reg1, 31B98692h
5 ret
```

Listing 18: Reversed VM Disassembled P-CODE

Starting with the value `FF37F33Bh` stored in `reg1`. We can then follow the reversed algorithm to find the right magic code.

$$\begin{aligned} reg_1 &= \texttt{FF37F33B} \\ reg_1 &= \texttt{FF37F33B} - \texttt{04030201} = \texttt{FB34F13A} \\ reg_1 &= \texttt{FB34F13A} + \texttt{01A7895E} = \texttt{FCDC7A98} \\ reg_1 &= \texttt{FCDC7A98} + \texttt{0F832FE9} = \texttt{10C5FAA81} \\ reg_1 &= \texttt{10C5FAA81} - \texttt{31B98692} = \texttt{DAA623EF} \end{aligned} \tag{1}$$

$$\Rightarrow reg_1 = \texttt{DAA623EF}$$

Now we need to prove that the value we just reversely calculate would properly calculate the hard-coded value in listing 6 at address `00331167` which is `FF37F33B`. All we need to do is to just forwardly trace the disassembled P-CODE in listing 17 to prove that the original and reversed algorithm are equal. Listing

$$\begin{aligned} reg_1 &= \texttt{DAA623EF} \\ reg_1 &= \texttt{DAA623EF} + \texttt{04030201} = \texttt{DEA925F0} \\ reg_1 &= \texttt{DEA925F0} - \texttt{01A7895E} = \texttt{DD019C92} \\ reg_1 &= \texttt{DD019C92} - \texttt{0F832FE9} = \texttt{CD7E6CA9} \\ reg_1 &= \texttt{CD7E6CA9} + \texttt{31B98692} = \texttt{FF37F33B} \end{aligned} \tag{2}$$

$$\Rightarrow reg_1 = \texttt{FF37F33B}$$

Perfect results! As we expected the output from both algorithms matched. Going backwards to find out the magic code, we traced listing 18, which can be seen in equation 1. With the magic code found, we consequently move back again to listing 17 and trace through its code, which can be seen in

equation 2. The exact value of `FF37F33B` was the output of such calculation, which proves that our analysis to the original and reversed P-CODE disassembly was accurate.

As stated before, this approach to solve the crackme relied only on abstract static analysis, without the aid of a debugger or having to run the crackme. I find this to be the most elegant solution. A little bit too extreme in not requiring to use a debugger nor run the crackme at all, but it teaches us a lot.

5.2 Dynamic Analysis via Debugger

In this solution we will attempt to break the protection and figure out the correct input parameters for the binary file with the help of the debugger. This is typically the standard or regular method when analyzing any sort of binary file. The reason to that is that sometimes there are conditions that have to be met during runtime and can not really be distinguished under static analysis only.

We Load up the crackme under the debugger (x64dbg), as shown in figure 18. I chose to use x64dbg as the debugger and not the local debugger in IDA, simply because I find this one more versatile and quiet packed with a lot of features. Moving along, we can see the debugger broke on entry of execution, this is the first step that happens in the OS. We are not interested in this section so we will let it run until it reaches user code. We can go ahead and do that.

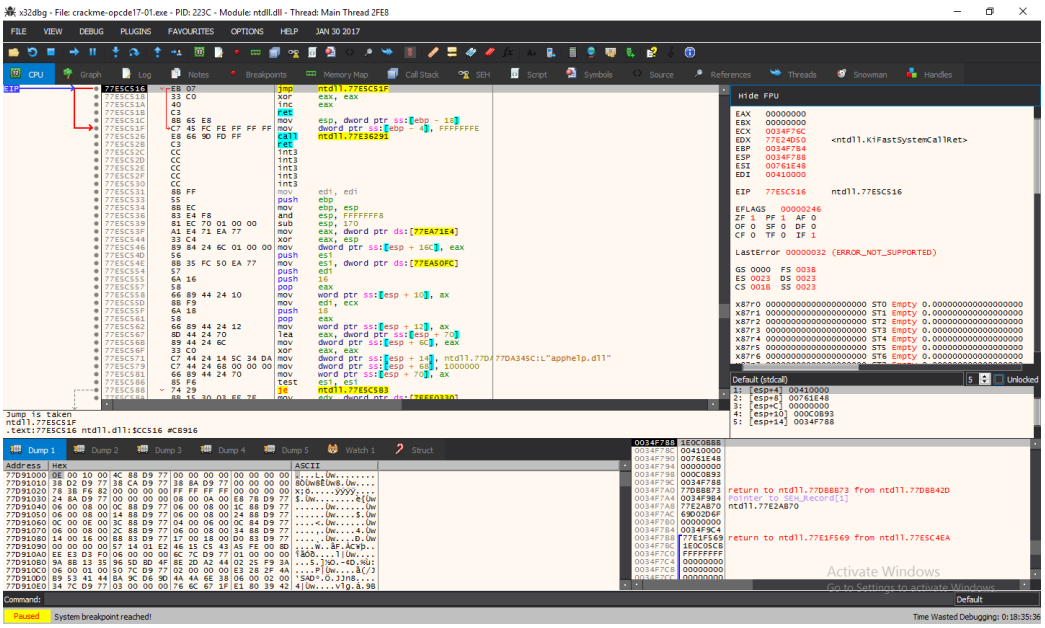

Figure 18: Break On Entry of Crackme

In figure 19, we can see we stopped at address `00E213EB`. We already identified that this binary file was compiled with Visual C Compiler/Linker with Visual Studio earlier in figure 2. By experience I already know where to jump to and what to skip too, but for a starter or a not very experienced reverser, he/she could get lost a little but or confused between all that code, which is the Visual C Run-Time library. Since we have already covered this before, we will move on and and just skip that call and only follow the unconditional jump `JMP` located at address `00E213F0`.

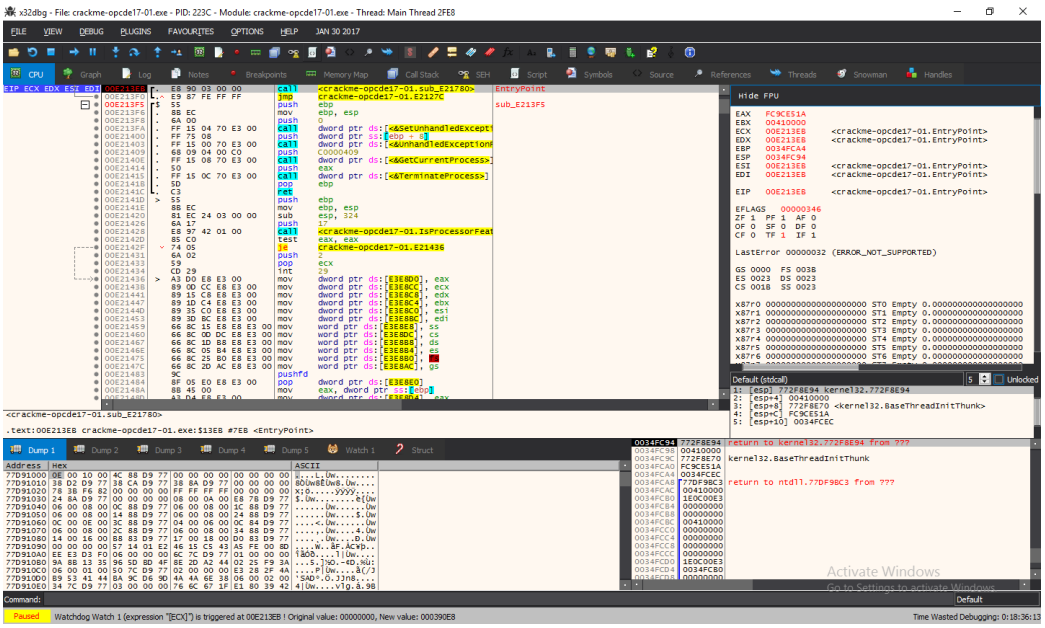

Figure 19: Startup Entry-Point 1

In figure 20, we see that the previous unconditional jump `JMP` located at address `00E213F0` took us to this current address `00E2127C`. Starting from this location is the code that basically runs before the `main`/`wmain` function. This long code segment is also known as the `WinMainCRTStartup` or `wWinMainCRTStartup` depending on whether Unicode or ANSI was used during compilation. This is actually the real entry-point for any C/C++ written application compiled with Visual Studio. This function does quiet a lot, but the following is a small summary of what goes underneath the hood.

- Initializes the global state needed by the CRT.
- Initializes some global state that is used by the compiler. Run-time checks such as the security cookie used by the compiler flag option `/GS`.
- Calls constructors on C++ objects.
- Retrieves command line and start up information provided by the OS and passes it the `main`/`wmain` function.

Now we know what is going on under the hood of `WinMainCRTStartup` or `wWinMainCRTStartup` and how it actually works so all we need to do now is to skip the unnecessary code and locate the code segment which calls the `main/wmain` function. We already know that it takes 3 parameters as previously shown in listing 3, so it shouldn't be too difficult to locate.

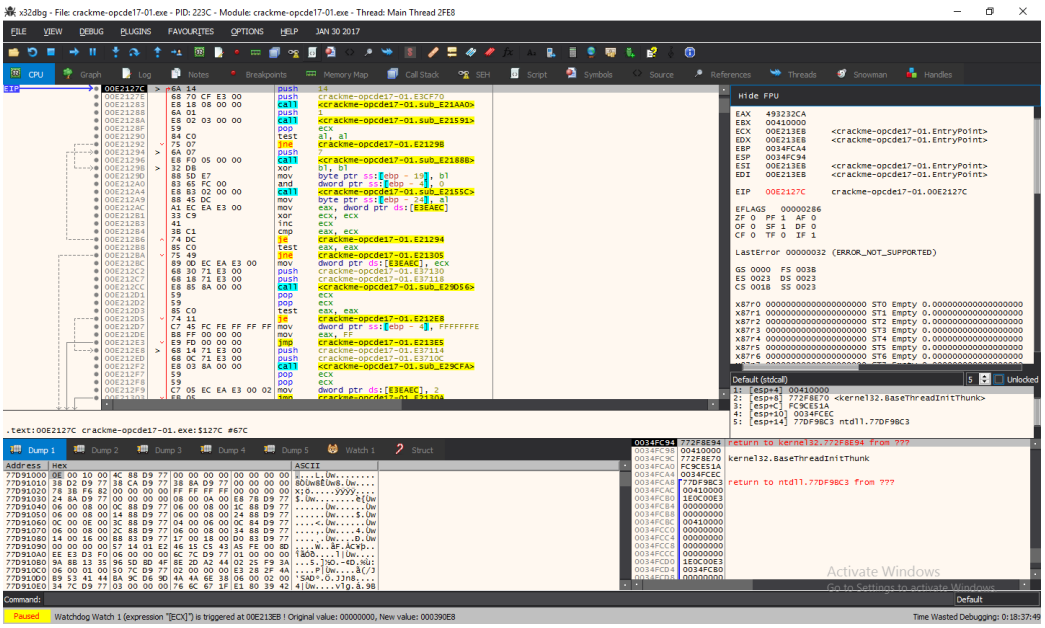

Figure 20: Startup Entry-Point 2

After scrolling down a little bit as shown in figure 21, we can see where `main/wmain` is being called. If we look at address `00E21370`, we can see the call to `main/wmain` and the 3 parameters being pushed at address `00E2136E`, address `00E2136C` and address `00E2136B` from the bottom up, respectively. So we need to follow that call to address `00E210F0` to begin our analysis.

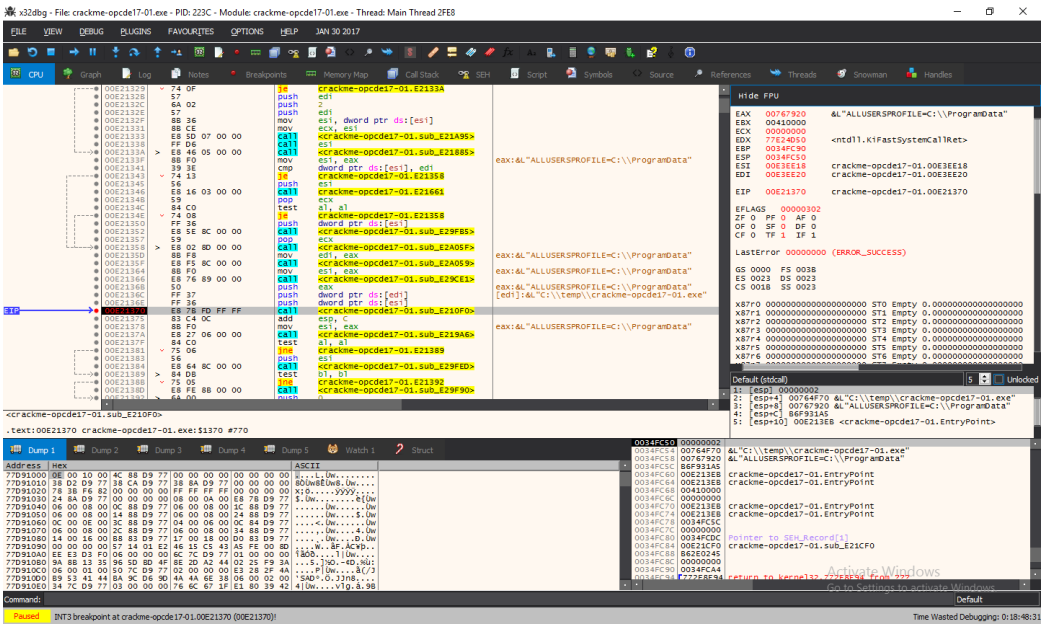

Figure 21: Startup Entry-Point 3

In figure 22, we see the first part of the main entry-point to the `main`/`wmain` function's code and in figure 23, we see the second part of the main entry-point to `main`/`wmain` function's code. In the beginning of figure 22, we can see the function prologue between addresses `00E210F0` and `00E210F3`. Looking at the code for a moment, we can then quickly realize that function `sub_E21020` is being called 5 times at address `00E21110`, address `00E2111A`, address `00E21132`, address `00E21176` and address `00E21194` whenever there is an output to the command prompt or `stdout` in general. So we can definitely skip analyzing this function, as it is of no importance whatsoever.

Next we look at the comparison `CMP` located at address `00E21122`, it looks like it is checking if the number of arguments passed to the crackme from the command prompt is more than or equals to the value `2`. It is understood from the fact that `EBP` is the Base Pointer, and therefore, `EBP+8` should point to the first parameter passed to the function. We should reload the crackme in the debugger, only this time we should make sure to set the debuggee's command line to something to test with, for example, lets use "12345678" as our input.

Moving forward to figure 23, we jump to address `00E2114B`. We notice that there is a call at address `00E21157`, which takes 3 parameters. This also doesn't seem to be an interesting function for us, because if we look at figure at address `00E2115F` we will notice a call which takes 1 parameter, and a comparison right after it with the value `FF37F33B`. Based on that comparison there will be a decision to be made in order to branch to either the `TRUE` condition which means that the command line argument supplied was correct; or the `FALSE` condition which means that the command line argument supplied was incorrect.

We conclude from this that the important function to consider checking first is actually `sub_E21080`. Because most likely the magic code calculation algorithm should be found there.

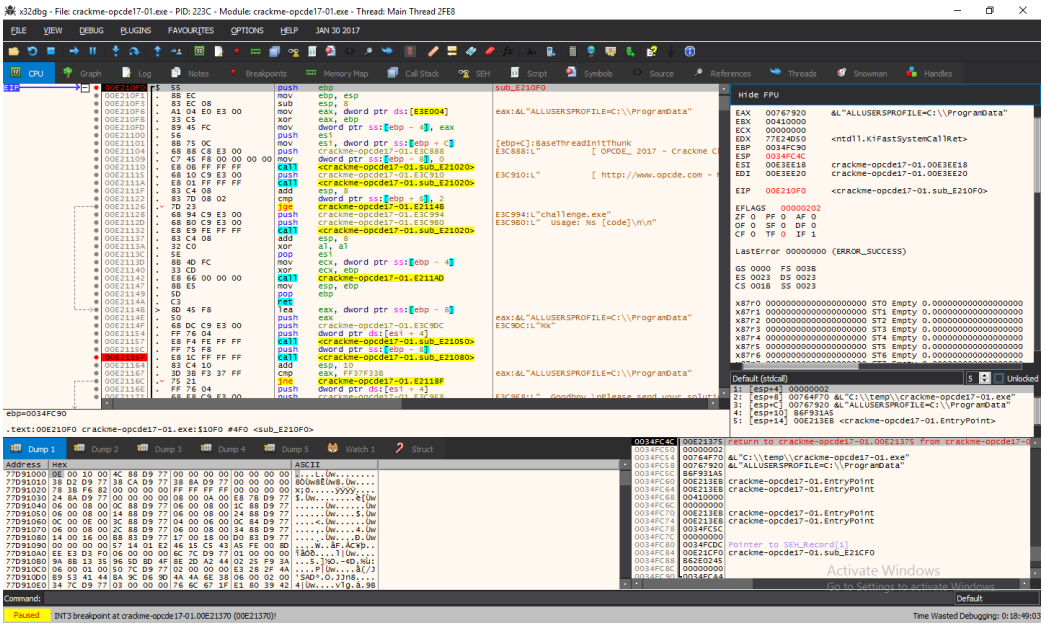

Figure 22: main/wmain Entry-Point 1

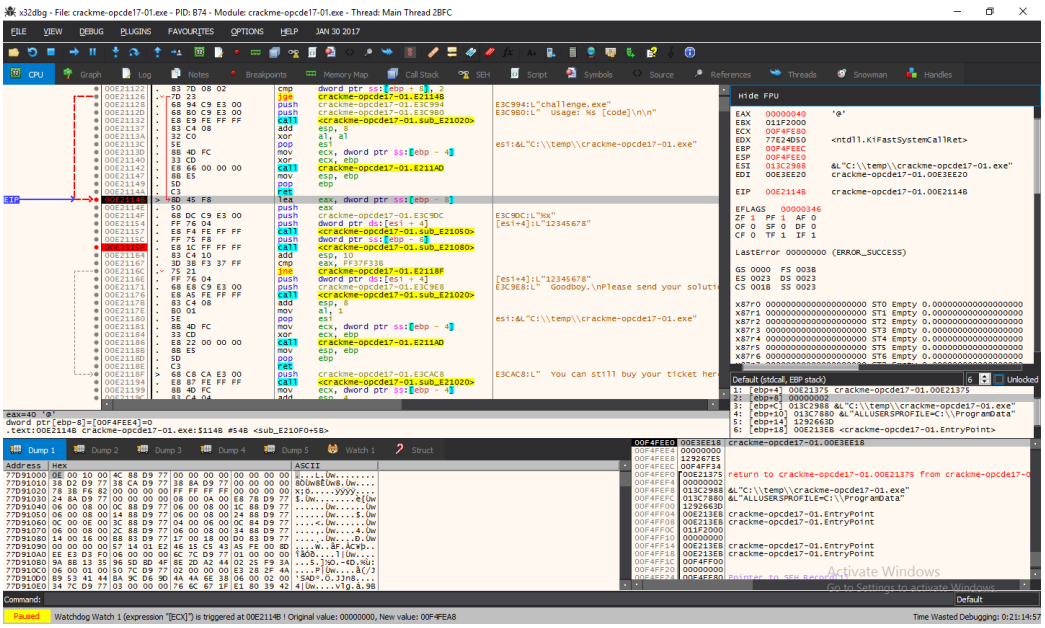

Figure 23: main/wmain Entry-Point 2

In figure 24, we can see the code segment that is probably responsible for the magic code calculation algorithm. From a dynamic analysis approach, we will most likely end up going through this function a number of times, reloading the crackme every time from the start. For the first time, we just need to examine it for a minute or two, just to understand what's going on inside this function.

It seems that at first, the function stores the value of the parameter passed to it earlier in `EDX`, this is observed at address `00E21083`. At address `00E21090`, stepping in through the code we notice that a byte is being read from a table located at address `00E3E7B0`. If we follow that address in the dump window, we will see a sequence of `25` bytes exactly ending with `FF`.

We continue stepping-in through the code, and notice that at address `00E21097`, the subtraction from `EAX` with `10h`, will yield a zero value and the comparison instruction `CMP` right after it will take us to the `TRUE` branch located at address `00E210BF`.

Next we store a byte from address `ECX+00E3E7B1`, which will store the value `0h` in `AL`. We compare `AL` and the value `6h` in order to decide whether to follow the conditional jump `JAE` to address `00E210DC` or not. This jump won't be taken as `EAX` has the value `0h`.

A double-word (DWORD) is read from the address `ECX+E3E7B0` which is `4030201h` to add it to the current value store in `EDX`. `ECX` is incremented by one and then compared for whether it is below or equal to the value `19h`, which is the total size of the array we just dumped earlier. Since this is our first iteration, the jump is taken, and we go to address `00E21090`.

We keep tracing through the code until we hit the address `00E210D8`, which moves the contents of `EDX` and stores it in `EAX`. At this point `EAX` should have the value `36C625C4`. We exit this function and return back to the `main/wmain` function.

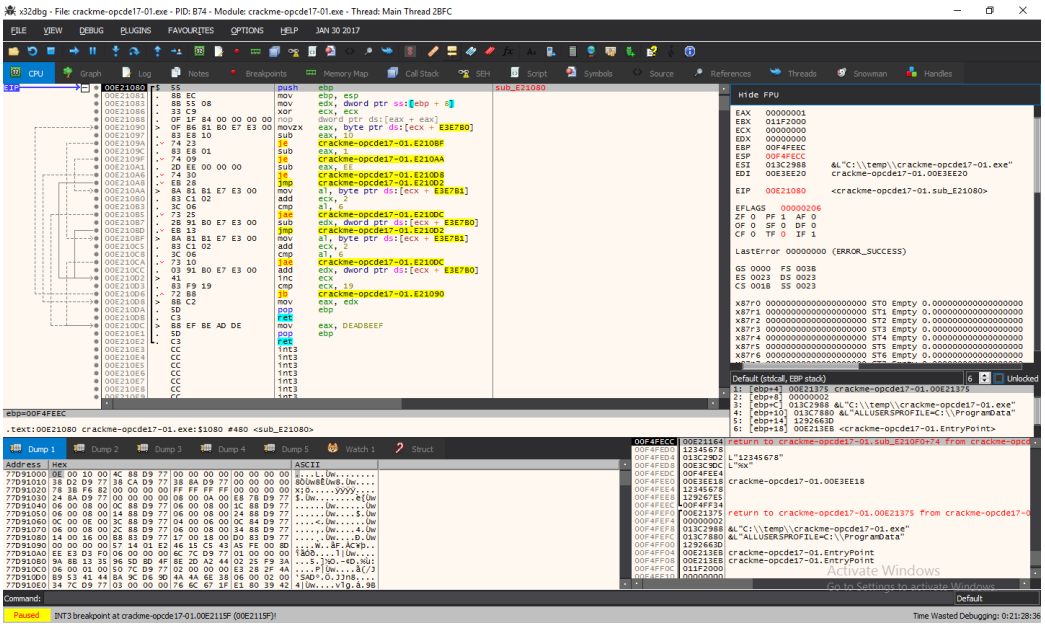

Figure 24: Magic Code Calculation

If we go back to figure 23, we can see that `EAX` is being compared to check whether the hard-code value `FF37F33B` is not equal to each other. Obviously, they do not match, so the conditional jump `JNE` will take us to the `TRUE` branch where we will hit the part of the crackme which tells us to try again.

This was just a dry test run to see what is happening dynamically under the debugger. At this point we should have understood what's going on in this function. We reload the crackme again and run it till we hit the function we were just analyzing located at address `00E21080` as shown in figure 24.

It is easy to spot a weakness in the crackme at this point. Let me explain where is it located and why is it a weakness.

As we look inside the function `sub_E21080` at address `00E21083`, we can notice that the parameter passed to the function is being stored in `EDX`. Also, right before the function terminates, we can see that the contents of `EDX` is being moved and stored in `EAX`. After the function exists and returns back to the `main/wmain` function, `EAX` is compared with the hard-coded value `FF37F33Bh`. Clearly we can see a pattern or connection between `EDX` and `EAX`.

That is great, but how can we leverage this to our own advantage? If you are still reading this paper with good amount of focus, you can quickly realize that we can trick the crackme with this relation between the two registers `EDX` and `EAX`.

If we restarted our debugging session again and let the crackme run till `sub_E21080` function located at address `00E21080` and then placed a break-point at address `00E21086`. We can then reset the contents of `EDX` to the value `0h` instead of holding the value of the parameter passed to it, which is the value of the argument passed to it from the command line. This way we can control the initial value of `EDX` from first iteration. We go ahead and place another break-point at address `00E21086` and let the crackme run till we hit that break-point we can then find out that the value of `EAX` is `2491CF4Ch`.

Perfect! Now we successfully hacked the crackme to let have it give us the offset to the magic code that we are supposed to actually find out. This can be achieved by subtracting the value `2491CF4Ch` from `FF37F33Bh` to get the value `DAA623EFh` as shown in equation 3. If we tried running the crackme from the command prompt while passing this value as the argument, we will see that the crackme congratulates us.

$$\begin{aligned} x &= \texttt{FF37F33B} - \texttt{2491CF4C} \\ \Rightarrow x &= \texttt{DAA623EF} \end{aligned} \tag{3}$$

Finally, to be honest, this is a poor man's approach. In no way did we fully understood the algorithm for calculating the magic code. We didn't really reverse anything at all, instead we just hacked our way through the crackme to solve it.

5.3 Concolic and Symbolic Execution

In this solution we will attempt to break the protection and figure out the correct input parameters for the binary file with using a rather novel and has been an academic topic mostly. Except only recently, this technique has been starting to get adapted more frequently especially with the evolution of tools and engines that allows us to do that easily and in a programmatic fashion. Although, this is not really an easy topic and considered to be complex as it relies on a lot of mathematical foundations and topics that should be covered first in order to fully understand how to leverage such technique in the automation of finding solutions to certain problems. The goal of static analysis in here is to derive a computable semantic interpretation at some point.

Before we dive into the solution, let me first give a quick introduction to the topic and explain some important principles for such approach. Generally speaking, Concolic execution is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs) path. Symbolic execution is used in conjunction with an automated theorem prover or constraint solver based on constraint logic programming to generate new concrete inputs (test cases) with the aim of maximizing code coverage.

Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

Abstract interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g., control-flow, data-flow) without performing all the calculations.

Given a programming or specification language, abstract interpretation consists of giving several semantics linked by relations of abstraction. A semantic is a mathematical characterization of a possible behavior of the program. The most precise semantics, describing very closely the actual

execution of the program, are called the concrete semantics. For instance, the concrete semantics of an imperative programming language may associate to each program the set of execution traces it may produce an execution trace being a sequence of possible consecutive states of the execution of the program; a state typically consists of the value of the program counter and the memory locations (globals, stack and heap). More abstract semantics are then derived; for instance, one may consider only the set of reachable states in the executions (which amounts to considering the last states in finite traces).

We can then summarize the whole process of Concolic and symbolic execution into the combination of the following points:

- **IR lifting**: Translates the program binary into a side-effects-free intermediate representation for program analysis. Examples of IRs are the BAP BIL, OpenREIL, Valgrind VEX and LLVM IR.

- **Symbolic execution**: Expresses the constraints on the inputs needed to reach parts of a program and to aid in path exploration. Examples of symbolic execution engines can be found in KLEE, S2E and Triton.

- **Constraint solving**: Determines the feasible range of inputs to reach a desired part of a program. Examples of some constraint solvers are Z3 and STP.

- **Taint analysis**: Tracks the data and control flows to determine constraints that can be controlled by (parts of) user-controllable inputs.

Two great Concolic and symbolic execution engines exists as of now; Angr and Triton. I prefer to use Triton over Angr, just because of personal preference. So we will continue this part and this solution method using Triton.

Triton is a Pin-based Concolic execution framework which provides some advanced classes to perform dynamic binary analysis (DBA), which was written and currently sponsored by Quarkslab. Triton provides components like a taint engine, a dynamic symbolic execution engine, a snapshot engine, translation of x64 instruction into SMT2-LIB, a Z3 interface to solve constraints and Python bindings. Figure 25 illustrates the architecture of Triton.

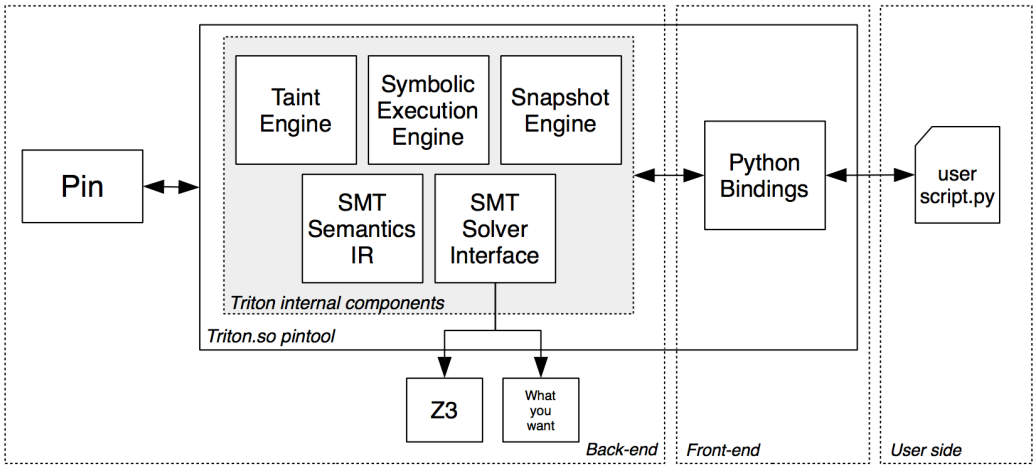

Figure 25: Triton's Architecture

Triton provides a taint engine. This engine applies an over-approximation but that doesn't affect the precision. For instance, in exploit development, what the user wants in reality is knowing if the register is controllable by himself and to know what values can hold this register. Answering this question only with the taint analysis is pretty difficult because a lot of instructions have an influence on the value that can hold a register (e.g. Path conditions, arithmetic operations, etc.). So, our personal reflection about this is: How can we gain time without losing precision?

Symbolic execution offers us the possibility to answer the question: What value can hold a register? However, by applying a symbolic execution and asking a model at each program point if a register is controllable is pretty expensive, therefore, we use an over-approximation to fix the loss of time and if a register is tainted, we ask a model for the precision.

For example, let's imagine a 16-bits register `[x-x-x---x-xx-x-x]` where "`x`" are bits that the user can control and "`-`" bits that the user can't control. This state of register is setup like this due to arithmetic operations but may be something else with a different input value. In this case, it's not useful to know what bits are controllable by the user because they will probably change with another input value. In this case, using a perfect-approximation or an under-approximation is not useful. What we want is to know what values can hold this register according to the user's input.

That's why Triton uses symbolic execution for precision, and uses over-approximated tainting to know if we can ask a model to the SMT solver.

The current step to take is to use IDA along with Ponce plug-in, which allows us to use the integration of Triton directly with IDA during debugging and tracing. This is way easier actually than writing a lengthy external script to do the same thing.

We go ahead and configure the plug-in as shown in figure 26. It is very important to use the same configuration I chose, so we can do the Dynamic Symbolic Execution (DSE) correctly.

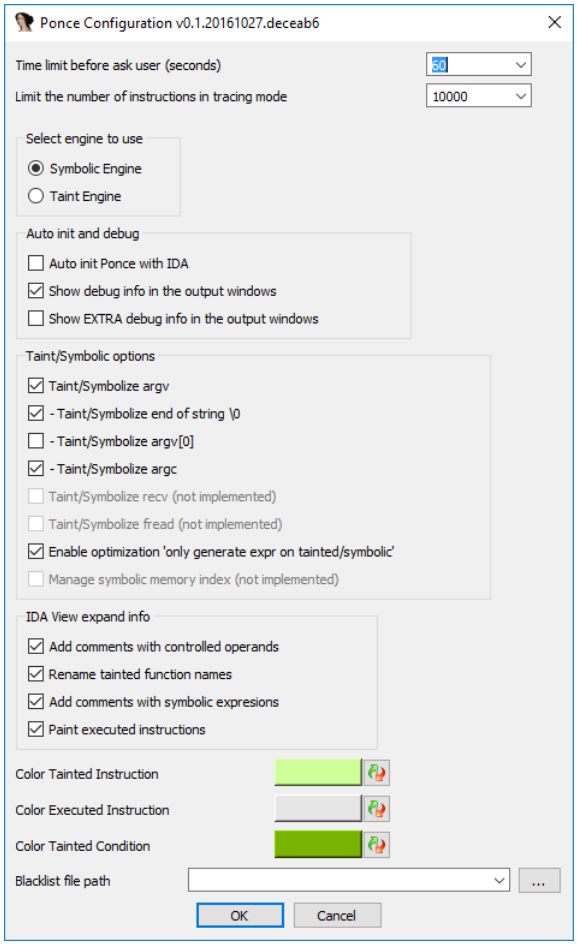

Figure 26: IDA Ponce Plugin

Once we have configured the plugin, we can go ahead and start our debugging and tracing session. In order to do it effectively, we will need to place a couple of breakpoints in certain locations so we can be able to stop the debugging/tracing session to symbolize the registers we are interested in as well as the range of memory addresses too. Table 1 will show every breakpoint address along with the function it is located at as well as the instruction too.

Breakpoint Address	Function	Instruction
004010F0	_main	push ebp
0040116C	_main	jnz short loc_40118F
00401086	VM_EXECUTE	xor ecx, ecx

Table 1: Breakpoints

We can go ahead and start out debugging/tracing session. Once the debugger starts it will stop at address `004010F0`, as shown in figure 27.

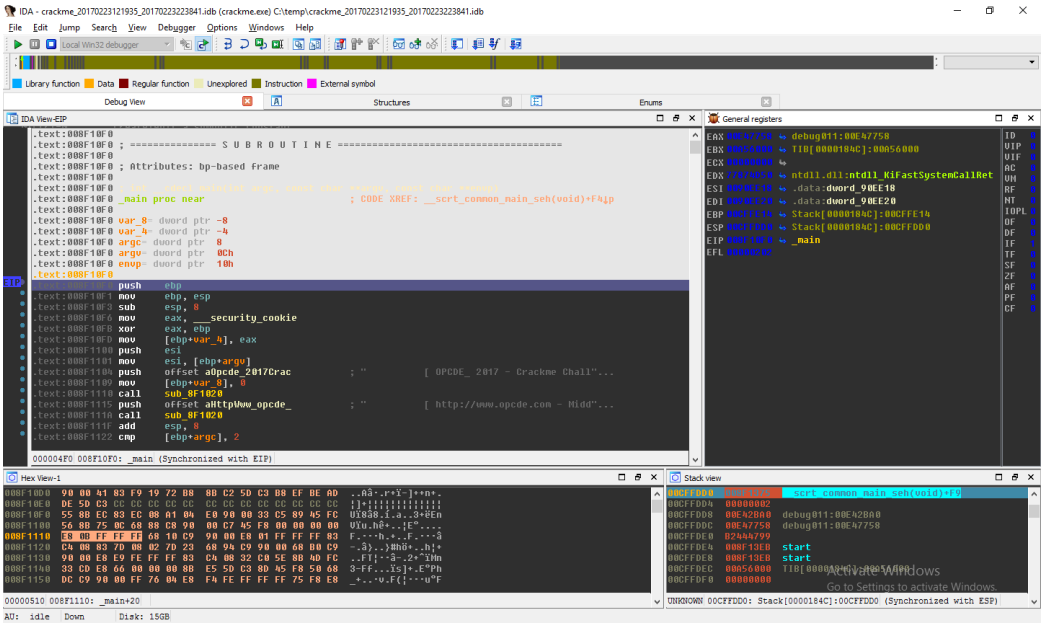

Figure 27: Dynamic Symbolic Execution 1

We can continue the execution once more till the debugger stops at address `00401086`, as shown in figure 28.

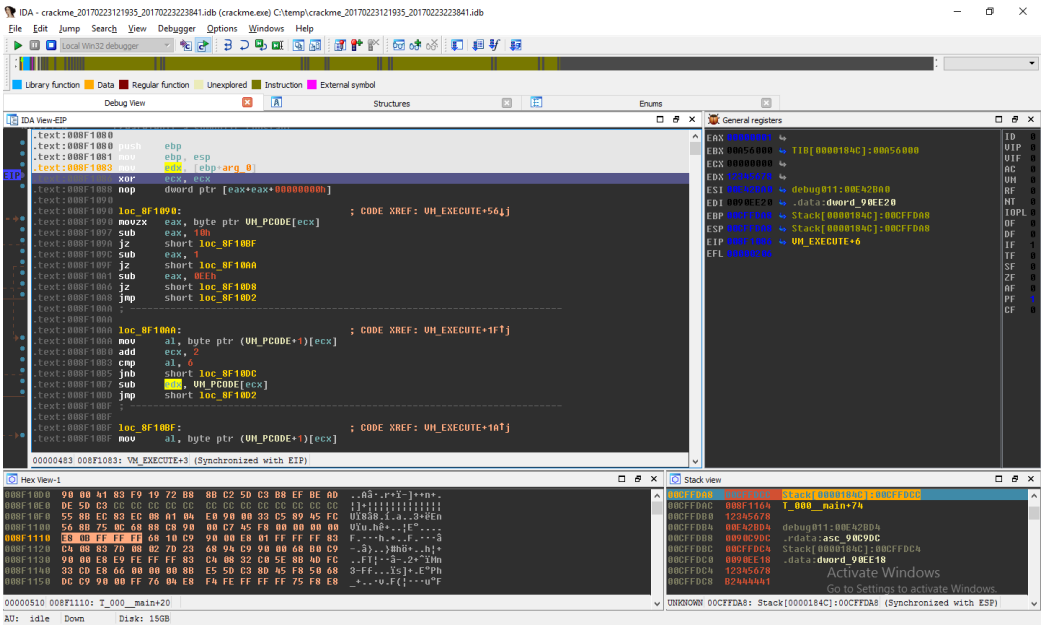

Figure 28: Dynamic Symbolic Execution 2

At this point we can go ahead and symbolize `EDX`. The reason we had the breakpoint at address `00401086` and not at address `00401080` is because we needed to load `EDX` first with the parameter passed to the function in `ebp+arg_0`. This is very important for the purpose of our dynamic symbolic execution. We can see this in figure 29. The highlighted instruction is the one of interest.

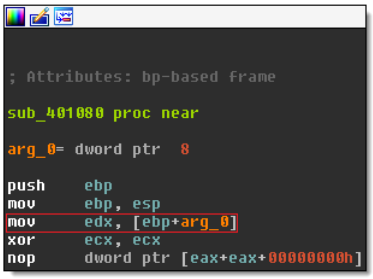

Figure 29: Symbolized Register (EDX)

We will scroll down towards the end of the function to address `004010D8` so we can symbolize `EAX` as well. This can also be seen in figure 30. The highlighted instruction is the one of interest.

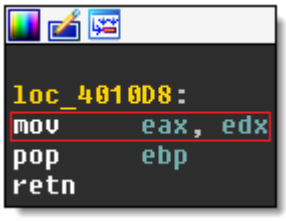

Figure 30: Symbolized Register (EAX)

So far so good, we have symbolized the two registers that are crucial for the dynamic symbolic execution process. Now we need to symbolize the memory region of `VM_EXECUTE` function. This is highly essential for the tracer to be able to trace through the instructions and recognize branches for the purpose of solving the SMT constraint e need in order to get to the right branch of the `TRUE` condition. This is located at the breakpoint we placed earlier at address `0040116C`, which can be seen in figure 31.

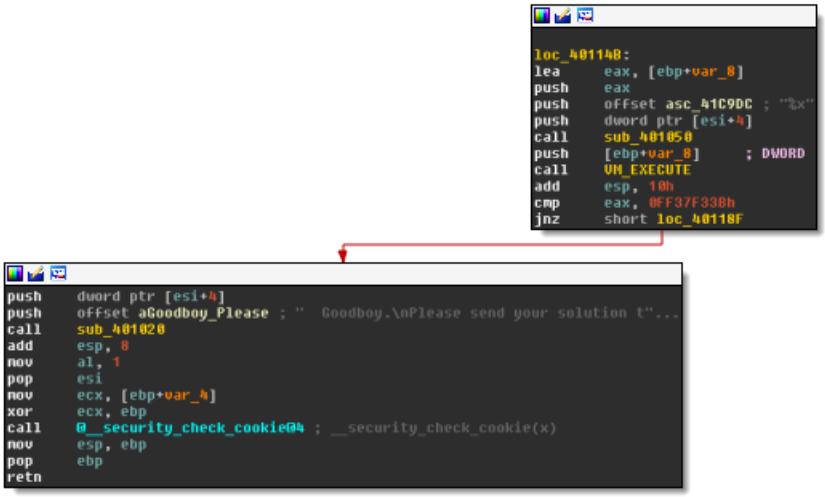

Figure 31: SMT Constraint

In listing 19, we can see the `VM_EXECUTE` function one more. As previously mentioned, we will have to symbolize the whole memory region of that function. We start from address `00401080` to address `004010E2`. Mathematics 101, the function is `99` bytes in size. This can be obtained by subtracting `004010E2` from `00401080`. Symbolizing the the whole memory region will take `100` bytes, just to return back to `_main` function.

```
1  .text:00401080 VM_EXECUTE      proc near
2  .text:00401080 arg_0           = dword ptr  8
3  .text:00401080
4  .text:00401080                 push    ebp
5  .text:00401081                 mov     ebp, esp
6  .text:00401083                 mov     edx, [ebp+arg_0]
7  .text:00401086                 xor     ecx, ecx
8  .text:00401088                 nop     dword ptr [eax+eax+00000000h]
9  .text:00401090 loc_401090:
10 .text:00401090                 [...]
11 .text:004010DC loc_4010DC:
12 .text:004010DC                 mov     eax, 0DEADBEEFh
13 .text:004010E1                 pop     ebp
14 .text:004010E2                 retn
15 .text:004010E2 VM_EXECUTE      endp
```

Listing 19: Symbolized Memory

We will continue execution once more and let the debugger run, till it stops at address `0040116C`. This is the main problem we are trying to solve primarily. We need to tell the SMT engine to solve this constraint so we can go to the `TRUE` branch in which the condition of the crackme will show us the congratulation message. This can be seen in figure 32.

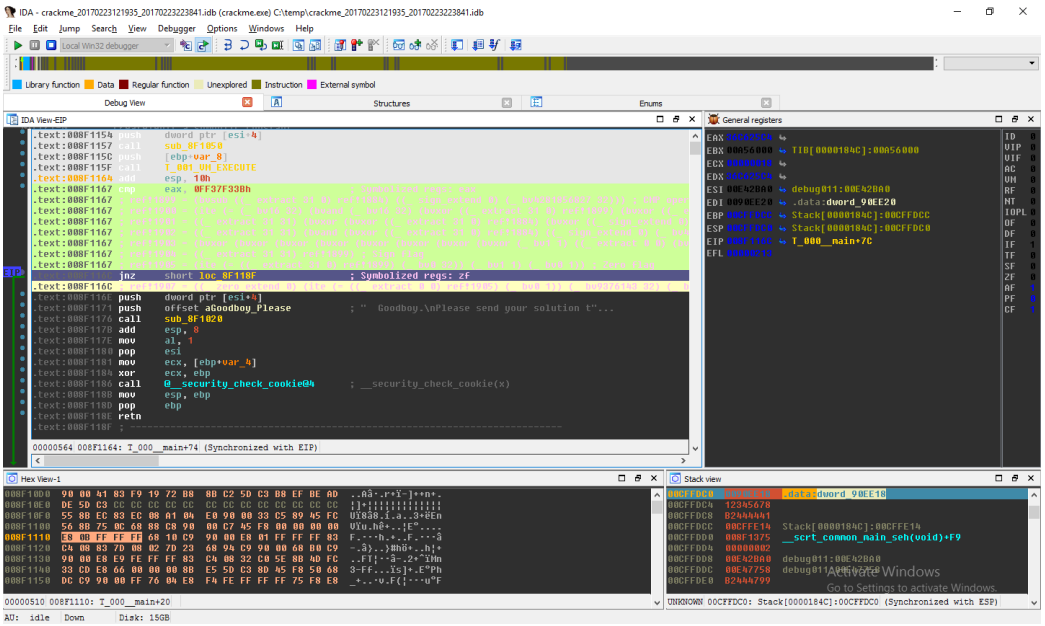

Figure 32: Dynamic Symbolic Execution 3

Finally we will ask the SMT engine to solve the formula from address `0040116C` to address `0040118F`. Of course we do not want to end up at that address since this is the `FALSE` branch in which the condition of the crackme will tell us that we have failed. However, we want to satisfy such condition so we can reverse the logic of the crackme to actually go to address `0040116E` instead. Doing so will give us the solution to this constraint, which is the correct magic code required to be passed to the crackme as an argument from the command line. This can be seen in listing 20.

```
[+] Solving condition at 1
[+] Solving formula...
[+] Solution found! Values:
- SymVar_0 (argc):0x000004
- SymVar_103 (Reg edx at address: 0x008f1083):0xdaa623ef
```

Listing 20: Triton DSE Output

5.4 Constraint Solver

Basically if we look into figure 33, we can see both the `TRUE` and `FALSE` condition that takes us to either of the try again or congratulations branch respectively.

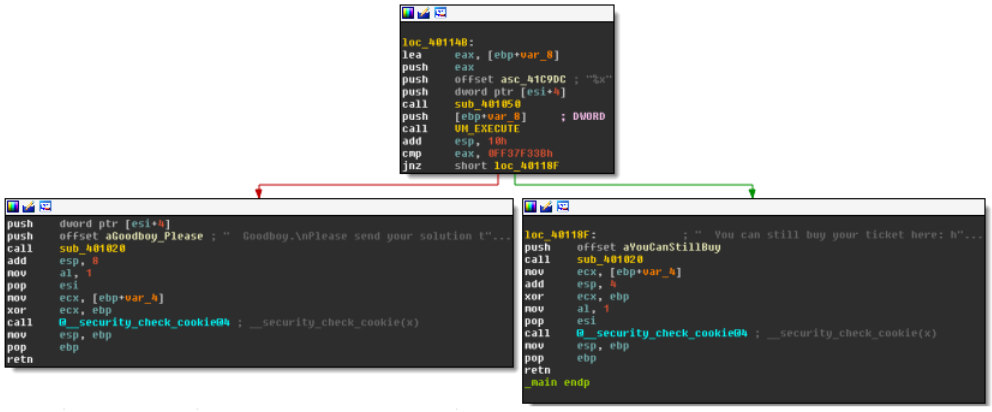

Figure 33: Success/Fail in main CFG

We have already identified that its a VM based protection used to calculate the magic code. We wrote a disassembler to disassemble the P-CODE of the VM and already have the disassembly listing of that VM's P-CODE. Although the algorithm is short and can just be solved by our hands and brain. We can imagine if it was a bit more complicated than that and we couldn't just understand how to manually solve it. We can then use the Z3 Theorem Prover to construct a small script to give us the answer automatically by finding out whether the constraints we have given it in the script is actually modulo satisfiable or not. If we take a look at listing 21 we can examine such script.

```
from z3 import *

# Declare a 32 bit vector representing register eax
reg_eax = BitVec('eax', 32)

# add eax, 4030201
eax = reg_eax + 0x4030201

# sub eax, 1A7895E
eax = eax - 0x1A7895E

# sub eax, F832FE9
eax = eax - 0xF832FE9

# add eax, 31B98692
eax = eax + 0x31B98692

# Create a solver instance
s = Solver()

# Add the constraint
s.add(eax == 0xFF37F33B)

# Check if satisfied
if s.check() == sat:
    print 'sat'

    print '[reg_eax = ' + hex(s.model()[reg_eax].as_long()) + ']'
```

Listing 21: Magic Code Constraint Solver

After we execute the previous script, we will have find our magic code printed out to the command prompt. Listing 22 shows such output.

```
$ python solve_opcde.py
sat
[reg_eax = 0xdaa623ef]
```

Listing 22: Python Script Output

6 Extra

We have fully analyzed the crackme and at this time we are capable of actually decompiling the whole crackme into readable C/C++ source code to match the original source code of the crackme, if not to a near approximation at least. In listing 23, we can see the decompiled version of the crackme. Although, I have changed certain parts on purpose to just make it smaller, but the functionality and structure should be the same.

```
#include <stdio.h>
#include <stdlib.h>

uint8_t PCODE[] = {
    0x10, 0x00, 0x01, 0x02, 0x03, 0x04, 0x11, 0x00, 0x5E, 0x89, 0xA7,
    0x01, 0x11, 0x00, 0xE9, 0x2F, 0x83, 0x0F, 0x10, 0x00, 0x92, 0x86,
    0xB9, 0x31, 0xFF };

u_long vm_execute(u_long ulParam) {
    u_long ulValue = ulParam;
    for (uint8_t Offset = 0;  Offset < 0x19; Offset++) {
        if (PCODE[Offset] == 0x10) {
            Offset += 2; uint32_t *ulReg = (uint32_t *)&PCODE[Offset];
            ulValue += *ulReg;
        } else if (PCODE[Offset] == 0x11) {
            Offset += 2; uint32_t *ulReg = (uint32_t *)&PCODE[Offset];
            ulValue -= *ulReg;
        } else if (PCODE[Offset] == 0xFF) { return ulValue; }
    }
    return 0xDEADBEEF;
}

int main(int argc, const char *argv[]) {
    if (argc <= 1) { printf("usage: opcde-crackme [magic-code]\n"); }
    else { printf("%s\n", (vm_execute(atol(argv[1]))
                            == 0x0FF37F33B) ? "SUCCESS!":"FAIL!"); }
    return EXIT_SUCCESS;
}
```

Listing 23: Decompiled Crackme

We have managed to fit the crackme in exactly 28 lines of code. I could imagine Matt's version to be longer and definitely more structured. However, due to the fact that during compilation a lot of information is lost and it is impossible to recover at all. Therefor, we can only rewrite it to what the assembly output generated from what the compiler looks like in the disassembly listing. Our code is exactly the same and mimicking the assembly output generated from the compiler. At this point, the crackme is overly simplified and it doesn't get any simpler than this!

7 Conclusion

The crackme was simple and straight-forward. Analyzing the code was not complicated either. However, the introduction of a VM based protection for the calculation of the magic code is something that most people are not familiar with. Therefore, it might be problematic a for some to understand what to do or where to begin. Although the magic code was embedded in a VM, the main algorithm was too simple and was reversed easily. This crackme is a perfect example for people who wants to get into reversing VM based protected crackmes. At the end, we have explored five different methods of solving this crackme. Each of which having its own advantage and disadvantage. It is up to the reverser to figure out which method is appropriate at any given time he/she wishes to solve something similar. My goal was to present different ideas for different tastes.

8 Acknowledgments

I would like to thank my friend Matthieu Suiche, first for hosting the first and only technical Security Conference in the Arab region. Also, I would like to thank him for allowing me to submit this paper to serve as an educational reference for students who wishes to learn more about reverse engineering or to understand how to break code virtualization protection schemes.

9 Contact

You can contact me directly on my email provided in the first page of this paper. If you feel the need to send encrypted emails, I am also providing my GPG key below. Please do not hesitate to email me about further questions or comments regarding this paper. Everything is highly appreciated of course.

10 GPG Key

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Comment: GPGTools - https://gpgtools.org

mQINBFTR0AgBEAClXG2OnaHU7tfe/AA1WCTCSwx5/484TSU/7w/NFjWmt6gGWOn5
dF8YlQUsP3bs+bzrMo3/+64pTEUsH2pnt0crDgJyHcQZb6IRMWPx1pFi2a36wO05
mP9jhDUmCYiLRlxhNi+1HTLbT5TcjKwHRynysDfbJn4Y6QSoYTqr05vmqUFvFPCC
fmqg/EmHf5D3JiOgxSKy8ID5uzH4JYS4snhJ5cke2zIoF4aCL59Vntk/GQyGlmfB
d4LzaOKQKi4SUW51BpYdqdNjcJqkLrIQFRdyAEHHi9Xff9evudB4CsOGZy3dy8mr
6BiHUXCLuHc4a635oeIMqUMeUq2c/xrcNM0YY9gPNosBzV5i0TzJ/XVlZXYdkc04
D3z6PjPFOP84Z5lEIFmXMbxJ0bHzCGKa3xBu04dhXvBTczFiZv7h8v6kaJSyBEgj
1zRIYABfj29G5s7QWtL6oiN2zUHcA1zmKvUrSoaVwldUTHOaMOHTWmNLYuwy9VaN
BEG+SuI1ZBjzDRTobbb2NiV1Gsgyx4xyPSNNxlLOkttpu+LDmIDhIK3/qC7AvCB3
N9PrT1hfUUD6Eq7M6s/OsiGGHyXdS5OJvdqsQ5bGkb/qjBgE9ZgZj1hQZDBi9Xq+
6CCHnRnXM7ysUCZLtLLQbS5sC/Vw5zzEU9BG1V3nk1FQMIOxo6Ar0fUkgwARAQAB
tCRNb2hhbWVkIFNhaGVyIDxpYW1oYWxzdGVuQGdtYWlsLmNvbT6JAj0EEwEKACcF
AlTR0AgCGwMFCQeGH4AFCwkIBwMFFQoJCAsFFgIDAQAChgECF4AACgkQ6EzIVXd8
ISxUcA//WqRR4ZcI9kxfAPYlZ8GnottGP6QpCpObdyHcuJlkv4Q2Ez+O5lTGyl+H
b7hbEBfl9EAgTY5FIup28Y5Tco4SyeEjb2EyNxuJjWuJFwpbdeXCz4iTc5hjwWXY
UecRpZmSBbTBWmkxeTn84APatnmIFEriTBGarEZVGpWbrHjKstfp0YzsVnVLsKUt
yp+2qWpYEvCof7GC0s5UbF/J9dNl7I4EXWxdcbnOo6zTfiHocS3yxzFtK3lbleNS
NjgaXG730tkMZDEcxMRfkzW2Rl6sYFDK3E+guQhEDzLBt1zVoqD5oqDNjXvy4TmM
mbe6bx+NOU1LavU/X3AqWt+XDynMvTt9l8WDlaQ0SXaauGbC1Fh9Dmw1A5rjhjw5
qADS9c9b8VJa1zC/LBULsskAsOWQeNjbQzx0y22gNb6stEqv0t3HiTbv5BWa8obS
DCvyQv43n9R67MBlqPkiF9S/jns7vIGHW9AUBoO5IVHo0jt5R65pHU2Wzl/cgiob
oexcTx11XIldkVtaZ8XUfu+dXz4Xhp6gFirK49NHpDR2jHuj5rpWX0i+AiSzYtVr
Jm/t3UWNUZ92BP8zJlKbQAahlvRJCPc2GROruBQI4Q7C1js/cTGZDyQXeadQCaId
0Ov+e0NguAtKa6NoWZ0rLZ/3KMB9bErCdAdeY1nGhUNfS9O6+je5Ag0EVNHQCAEQ
ALOddN7xcVoGkBgl1S3gadE6hSnAzJawRtwDi3ow+cPinjhys9D4h+yss3uzbHw6
AqzlgtE12rOkyiNWKA/ZmkrYseAhXJxkNCkuPSdsW91zHmNdFbkuGtqexROka7FL
uUKoehRNERzxFT5I8Z4jUuVuEVdg1frAVw07f13Q3T/fP/wqjCsONqNmEslk68LP
MFIFSoLlfl4jwRo+rrqLh5h3bhDAEDgOwqHjx9C3cwnVALJvLoZjuiJJIYSGGS7q
SeVQZVsWwITBAifBadQQbl5P3wyJtJwNur6YGbqI+UsPrDyExxzSXqbiPsjhcvUb
EvfcpuDqQE9qmcAPr6F8ouZ8uE6trg0XjgoD9PH85Pm2e/aWk1qTyltV5zDnnYzS
/6OLzU+t7OQkZxdz5EbNs8e7WDMwlxSvZOG4iIP1nReZFNIQZhDwI89I+e/nsD6W
jk5snA6RTKxbTycd4KF6NeeKd/IDClVd5EIekt+2gn/EnYXc6gAKFwcJ3nOw6HN9
BP4e6LGDcBrZQMBn50gavka20vd6sIPSboA6qZyOhVheiR7QfmYjJrx16/XqLabs
Vty+XwePxDdZXD0bmEba+ZN/2caOUQYn6M7L4nO88BymuiLlwiPw6/LKuLvAAANH
PbAIlumcK/62ZAXyvR1jBbdHKCA8I3dScq8fwkD33i+XABEBAAGJAiUEGAEKAA8F
AlTR0AgCGwwFCQeGH4AACgkQ6EzIVXd8ISy++xAAnhJlCJmcGld7CSM4ZUKTfJZC
krok45Ch3IpEtJcWHtvq5AAzjtls4XBnq/fg18rsnDyXG7DgvUnGiJSwiBar0/f1
gySBKHlR+z0ZAhXF2kMcuP6Y6+SxkzdNT2DHChtauQbqpCKiKEHavQ28ui98FMPO
MBz6LBg9X9QSbcTEU3Wy1/lfzCdODfebp4t6tFOUH/ElH48uD3Nt1Iv68vKmp5lu
QQsUR6bmT5bM663banGFii5T4NTlOBkq+78xk8pgcUAvXnCT0Awdrqhjja NQUkuG
zq4dSVuaRUR785yRCs7HR/isVduILeoqrHslfKKO4pLw4Dumd3VpYU7I3n+yL8A9
pn/BwxAp/4hFjR9zRXzCi8F3G0lm6UKdVmU58q5uIbl3Ch+5zbBJLdkOV/T4cU9z
cmPeW5vA6NxeXc72+mltOVzIdnuNS5QIN3/TOVltx1SIFkVO05cv0faya5WwIKL3
o08TIbvgCkArzllexzycHQzbqXH1ZOOvf5RFQ/yLxU8xt3E1HV1LzoS9M0IHsaS1
opbEr/f6gmjOw+dGAdJRLLDZ5tkI3fqsoNuHGGjod/7oPfiy9AmyrZqMK3s6LDzL
OsO5avn9iVSVx7u4gLsMGrHB/bNKSaNhUN8Q04EpNRdnPseG7R4LGvceOTzTPvmk
gbI5T7lFL+uS4iyFk7Y=
=/wP1
-----END PGP PUBLIC KEY BLOCK-----
```