Basic VM Challenge 1

S Chowdhury

Introduction

In this challenge, we will investigate an obfuscated executable. The obfuscation technique used here is called “virtual machine based obfuscation”. Before you start this challenge, you should know the following topics:

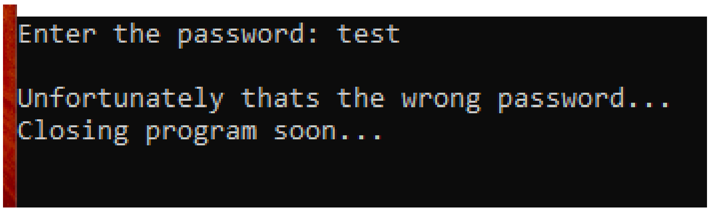
- C programming
- Python programming
- Assembly programming
- Fetch decode execute cycle
- Program simple emulator such as CHIP-8

If this section seems difficult, first learn about those topics. It’s recommended that you’re comfortable with assembly programming and understand fetch decode execute cycle.

Main Function Analysis

In this challenge, we will have a “VM” that runs the code which test’s user input. In this challenge, you will see how the VM takes bytecode from the *.data* section, and based on that bytecode it would perform different actions. The VM in this challenge has instructions for addition, subtraction and moving values. There are 6 registers, including 1 register that stores the instruction pointer. There’s also an array that stores the bytecode.

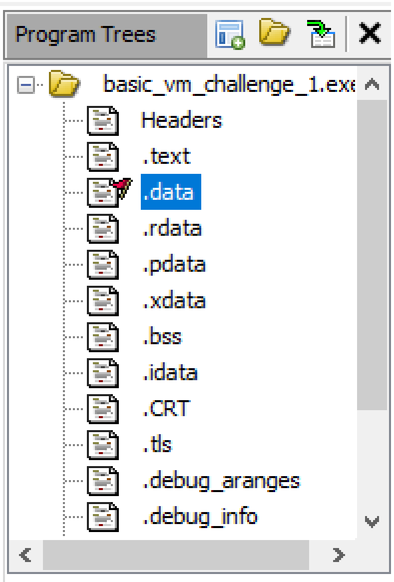
First, run the program.



Note, if you want you can try to brute force the password, you don’t have to investigate the rest of the program. However, lets investigate for education purposes.

Open this program in Ghidra.

Click on the *.data* section to see what’s stored there:



We can see a variable called “bytes”:

```
                            .data
                            bytes
        00404020 aa            ??          AAh
        00404021 00            ??          00h
        00404022 00            ??          00h
        00404023 00            ??          00h
        00404024 10            ??          10h
        00404025 00            ??          00h
        00404026 00            ??          00h
        00404027 00            ??          00h
        00404028 50            ??          50h     P
        00404029 00            ??          00h
        0040402a 00            ??          00h
        0040402b 00            ??          00h
        0040402c 11            ??          11h
        0040402d 00            ??          00h
        0040402e 00            ??          00h
        0040402f 00            ??          00h
        00404030 20            ??          20h
        00404031 00            ??          00h
        00404032 00            ??          00h
        00404033 00            ??          00h
        00404034 20            ??          20h
        00404035 00            ??          00h
        00404036 00            ??          00h
        00404037 00            ??          00h
        00404038 10            ??          10h
        00404039 00            ??          00h
        0040403a 00            ??          00h
        0040403b 00            ??          00h
```

One thing to note, when programming this VM, an “int” array was used to store the bytecode. Since “int” data type is 4 bytes big, that’s why you see 3 extra bytes of “00” after each bytecode. So the actual bytecode would be:

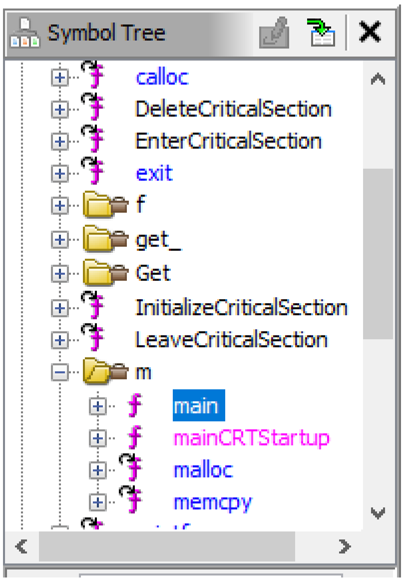
aa 10 50 11…

We can see a password variable:

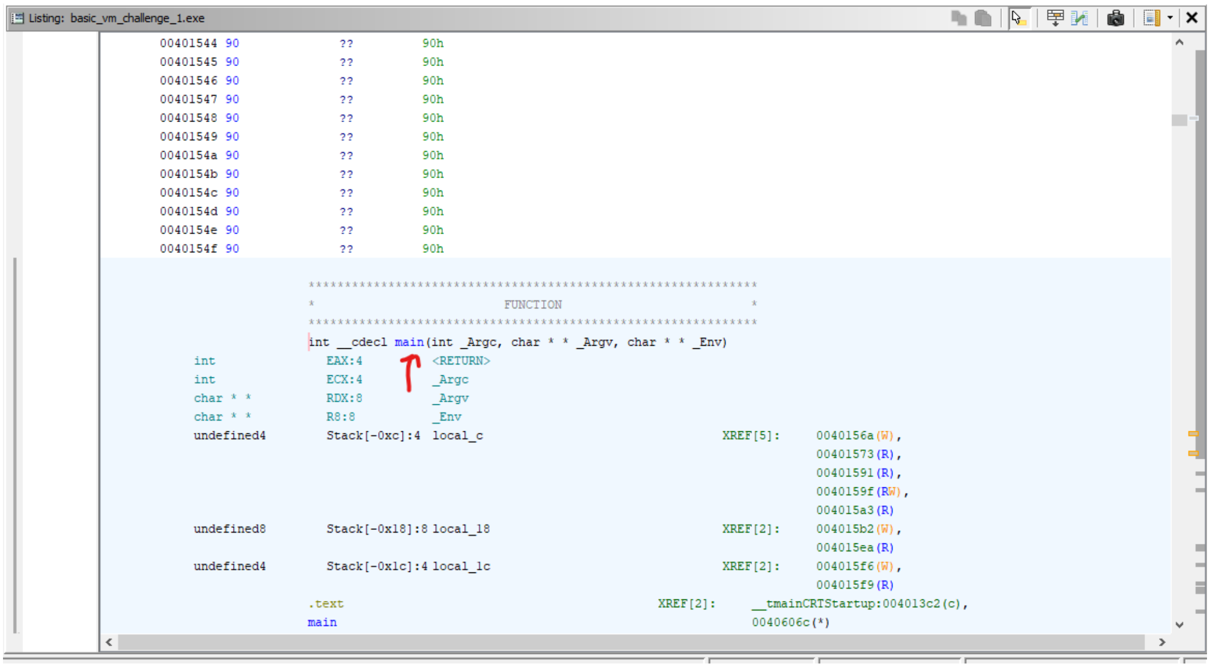
```
                    password
00404070 e8              ??          E8h
00404071 03              ??          03h
00404072 00              ??          00h
00404073 00              ??          00h
00404074 00              ??          00h
00404075 00              ??          00h
00404076 00              ??          00h
00404077 00              ??          00h
00404078 00              ??          00h
00404079 00              ??          00h
0040407a 00              ??          00h
0040407b 00              ??          00h
0040407c 00              ??          00h
0040407d 00              ??          00h
0040407e 00              ??          00h
0040407f 00              ??          00h
```

Does not seem like string. Maybe it’s an int variable? The hex value 0x03e8 represents 1000 in decimal. You can try entering that into the program.

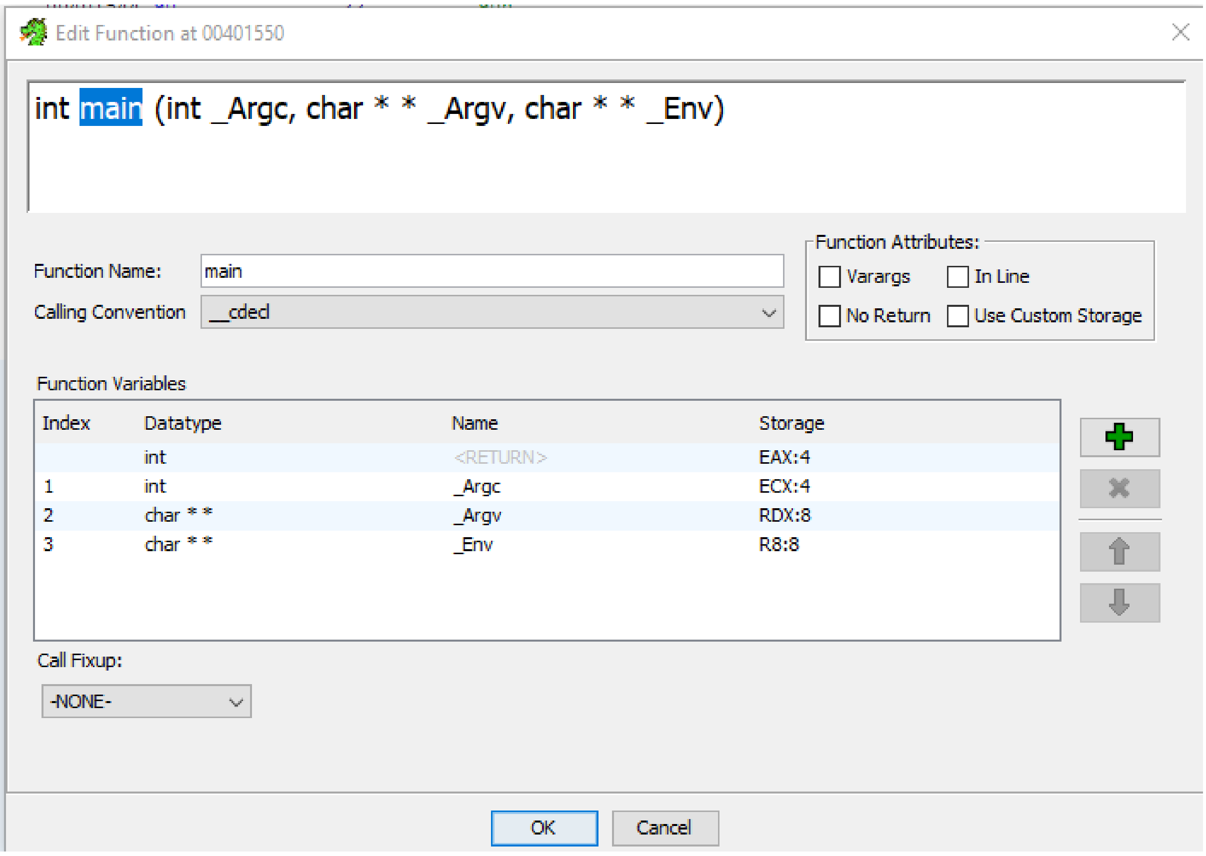
Return to *main* for now by double clicking *main* in the **Symbol Tree**:



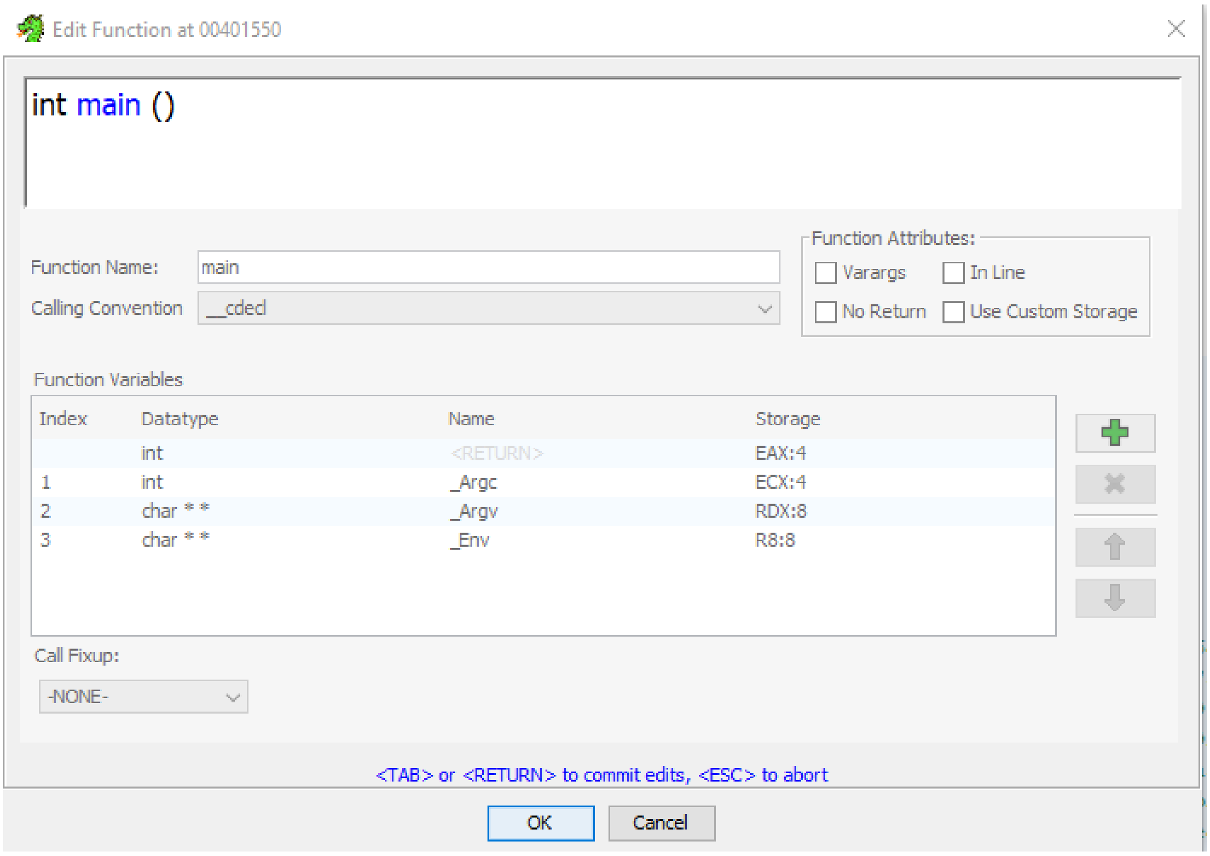
Right click on the main function:



Then select *Edit Function*.



Delete the stuff inside the brackets.



Now click ok. Now the assembly would look nicer. Notice in the main function, we can see references to the “bytes” global variable and a “cpu” global variable.

```
00401578 48 8d 14        LEA        RDX,[RAX*0x4]
         85 00 00
         00 00
00401580 48 8d 05        LEA        RAX,[bytes]                                   = AAh
         99 2a 00 00
00401587 8b 0c 02        MOV        ECX,dword ptr [RDX + RAX*0x1]=>bytes          = AAh
0040158a 48 8d 05        LEA        RAX,[cpu]                                     = ??
         ef 73 00 00
00401591 8b 55 fc        MOV        EDX,dword ptr [RBP + local_c]
00401594 48 63 d2        MOVSXD     RDX,EDX
00401597 48 83 c2 04     ADD        RDX,0x4
0040159b 89 4c 90 08     MOV        dword ptr [RAX + RDX*0x4 + 0x8]=>DAT_00408988,... = ??
0040159f 83 45 fc 01     ADD        dword ptr [RBP + local_c],0x1

                     LAB_004015a3                            XREF[1]:     00401571(j)
004015a3 8b 45 fc        MOV        EAX,dword ptr [RBP + local_c]
004015a6 83 f8 4f        CMP        EAX,0x4f
004015a9 76 c8           JBE        LAB_00401573
004015ab 48 8d 05        LEA        RAX,[cpu]                                     = ??
```

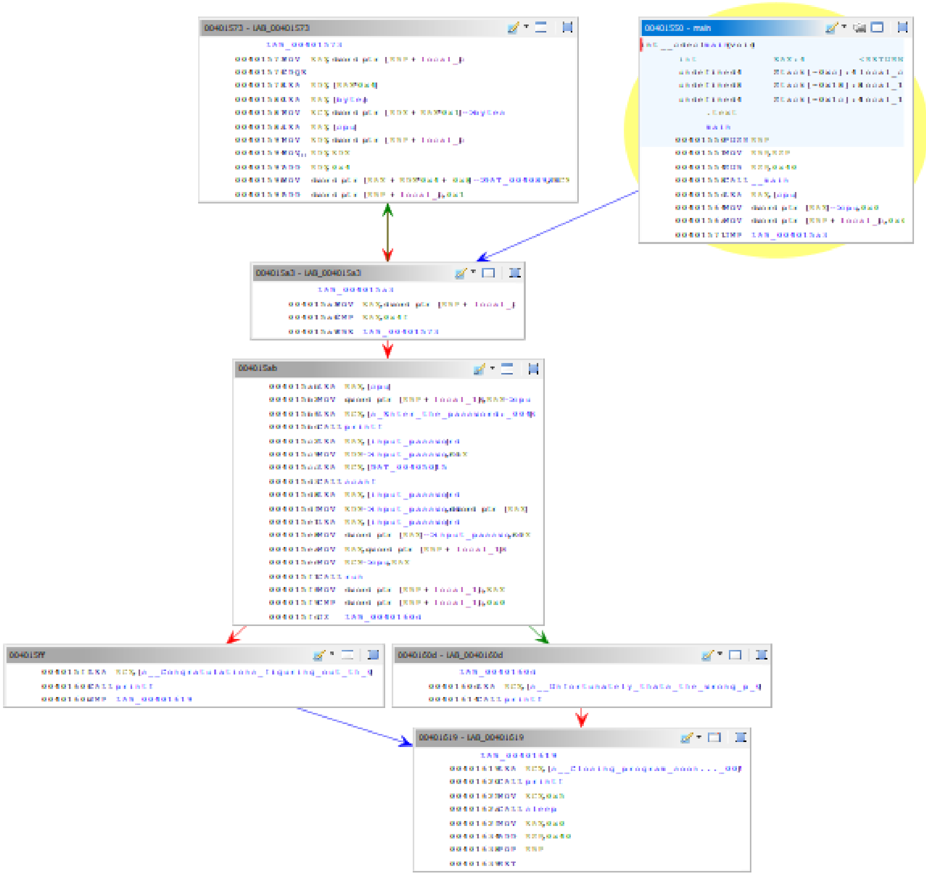
Let’s investigate the “cpu” variable. Double click on “cpu” to see info.

```
                    cpu                                         XREF[6]:     main:0040155d(*),
                                                                             main:00401564(W),
                                                                             main:0040158a(*),
                                                                             main:004015ab(*),
                                                                             main:004015b2(*),
                                                                             main:004015ee(*)
00408980                undefined4  ??
00408984                ??          ??
00408985                ??          ??
00408986                ??          ??
00408987                ??          ??
```

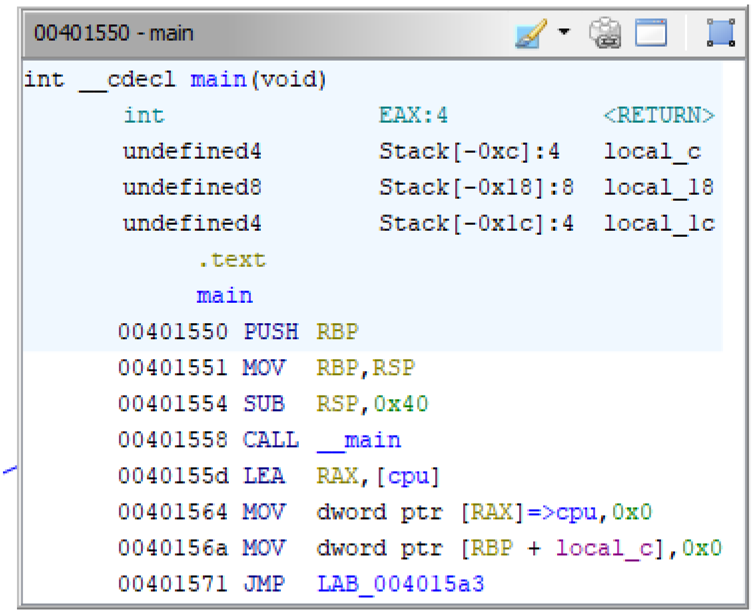
Note that this cpu struct contains information regarding the VM. Things such as instruction pointer and registers get stored here. If it's not clear now, then later we can see how it works. For now, open the decompiler and investigate the *main* function:

```
Decompile: main - (basic_vm_challenge_1.exe)

int __cdecl main(void)

{
  int iVar1;
  uint local_c;

  __main();
  cpu = 0;
  for (local_c = 0; local_c < 0x50; local_c = local_c + 1) {
    *(undefined4 *)(&DAT_00408988 + ((longlong)(int)local_c + 4) * 4) =
         *(undefined4 *)(&bytes + (longlong)(int)local_c * 4);
  }
  printf("Enter the password: ");
  scanf("%ld",&input_password);
  iVar1 = run(&cpu);
  if (iVar1 == 0) {
    printf("\nUnfortunately thats the wrong password...");
  }
  else {
    printf("\nCongratulations figuring out the password!");
  }
  printf("\nClosing program soon...");
  sleep(5);
  return 0;
}
```

As you can see, the decompiled version looks questionable. It can be fixed with a bit of work, but for now lets investigate the assembly. Enter the "graph mode" for the *main* function.



Let’s investigate the first block.



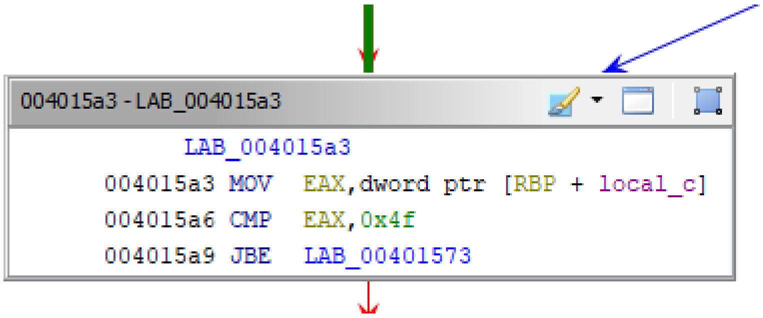
We can see that on **0040155d**, the address of the “cpu” global variable gets saved into RAX. Then, on **00401564**, we store the value 0x0 at the address saved in RAX. Note how it says *dword ptr*, this means that it points to data which would be of size double word. Here’s a quick review:

- Byte – 8 bits

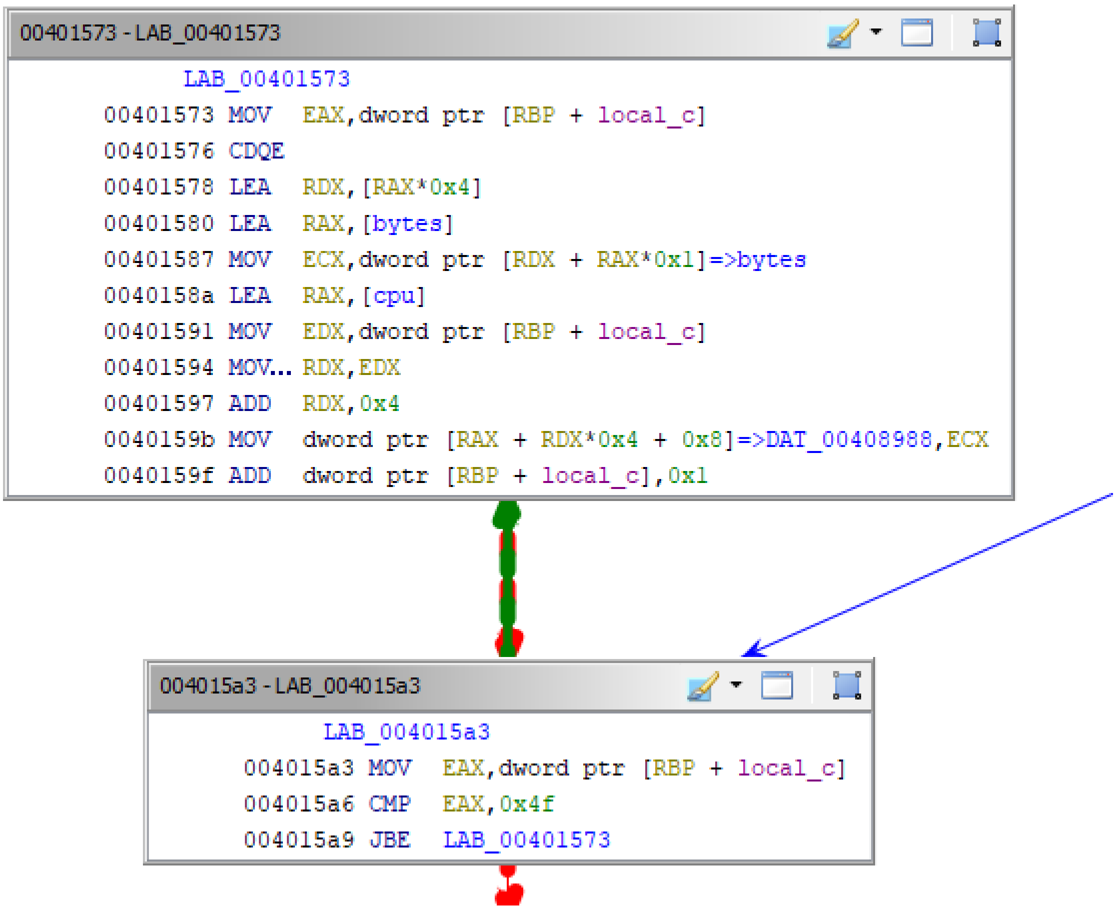
- Word – 2 bytes
- Double word – 2 words
- Quad word – 4 words

One thing to note, an int is 4 bytes size, so if you see double words, then it could be an int. Knowing about the size of different data types could be helpful.

On line **0040156a**, we can see that the value 0x0 gets put into the local variable “local_c”. After that, there’s a jump to an address. Let’s investigate that block.



We can see that “local_c” gets put into the EAX register. Then, the *cmp* instruction is used to compare EAX (which is currently holding the local variable local_c) with 0x4f. However, we know that 0x0 was just put into “local_c”. The *JBE* command will jump to that memory location, because 0x0 is less than 0x4f. Let’s investigate where this jumps to.



One thing you might notice is that at the end of the block starting at **00401573**, we add 1 to “local_c” and then move back into the block **004015a3**, where it performs the check again. The “local_c” variable seems like to loop index.

So at **00401573**, we move the loop index into EAX. After that, there the *CDQE* instruction. This command extends a double word in the EAX register into a quad word in the RAX register. Then we multiply RAX by 4, and store that into RDX. Note that this would be used as an offset to get different items in the array. We times by 4, because maybe each item in the array has size of 4. Then at **00401580**, we store the address of the “bytes” variable in RAX. Then, we have this instruction:

*mov ECX, dword ptr [RDX + RAX*0x1]*

Remember that RAX currently holds the base address of the “bytes” array, and RDX contains the offset. The offset that gets stored in RDX gets calculated by the loop index, or “local_c”. So were storing into ECX whatever in the memory address of RDX + RAX. In other words, a particular item in the “bytes” array.

At **0040158a**, the address of the “cpu” global variable gets stored into RAX. Then, the looping index gets stored into EDX.

```
00401573 - LAB_00401573
            LAB_00401573
    00401573 MOV       EAX,dword ptr [RBP + ii]
    00401576 CDQE
    00401578 LEA       RDX,[RAX*0x4]
    00401580 LEA       RAX,[bytes]
    00401587 MOV       ECX,dword ptr [RDX + RAX*0x1]=>bytes
    0040158a LEA       RAX,[cpu]
    00401591 MOV       EDX,dword ptr [RBP + ii]
    00401594 MOVSXD    RDX,EDX
    00401597 ADD       RDX,0x4
    0040159b MOV       dword ptr [RAX + RDX*0x4 + 0x8]=>DAT_00408988,ECX
    0040159f ADD       dword ptr [RBP + ii],0x1
```

The next command, *MOVSXD*, which is extending the value in EDX to 64-bits. Then we add 4 to RDX. After that, we copy whatever’s in ECX (remember it stores an item from the bytes array) into the memory location “RAX + RDX*0x4 + 0x8”. RAX stores the base address of the cpu global variable, meanwhile “RDX*0x4 + 0x8” stores the offset. However, why do we have to add 4 to RDX at **00401597** and then multiply RDX by 0x4 and have another 0x8? Well we probably multiply by 0x4 because each item in this new array would be 4 bytes apart. The “cpu” global variables a struct, which contains the instruction pointer, registers and an array to store the bytecode. So something like this:

struct cpu_struct{

int instruction_pointer;

int register_array[5];

int bytecode[100];

}

So, from the base address of the "cpu" global variable, an extra offset is required to get to the "bytecode" array, which is a new array in the "cpu_struct" struct.

Then at **0040159f**, the loop index gets incremented by 1. Then we come back to the block which checks the value of the looping index.

```
004015a3 - LAB_004015a3
            LAB_004015a3
    004015a3 MOV        EAX,dword ptr [RBP + ii]
    004015a6 CMP        EAX,0x4f
    004015a9 JBE        LAB_00401573
```

Note that the *CMP* function is used to compare loop index with 0x4f. Even though the "bytes" global variable is an array with 20 items. This is because I made a mistake when programming this challenge, it should compare with 0x14. However, the rest of the challenge still works, so just ignore that for now.

Explanation why it still works:

It still works because the bytecode array which is part of the cpu struct has 100 items, while the "bytes" global variable has 20 items. The VM program is less than that, and exits before any of the junk values get reached. However, just ignore for now.

After the loop stores the bytecode from "bytes" global variable into the bytecode array in the "cpu" struct, we have this block of code:

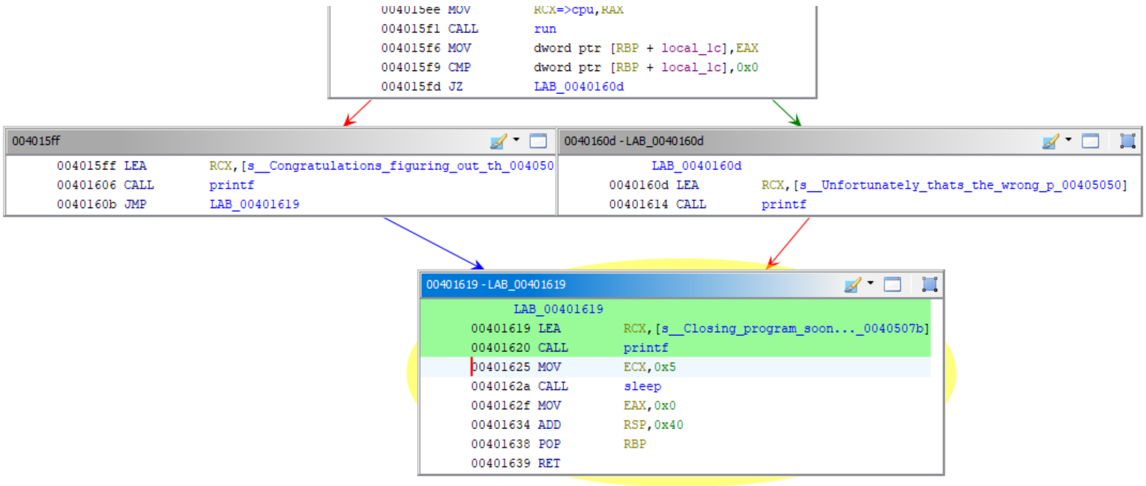
```
004015ab
    004015ab LEA        RAX,[cpu]
    004015b2 MOV        qword ptr [RBP + local_18],RAX=>cpu
    004015b6 LEA        RCX,[s_Enter_the_password:_00405000]
    004015bd CALL       printf
    004015c2 LEA        RAX,[input_password]
    004015c9 MOV        RDX=>input_password,RAX
    004015cc LEA        RCX,[DAT_00405015]
    004015d3 CALL       scanf
    004015d8 LEA        RAX,[input_password]
    004015df MOV        EDX=>input_password,dword ptr [RAX]
    004015e1 LEA        RAX,[input_password]
    004015e8 MOV        dword ptr [RAX]=>input_password,EDX
    004015ea MOV        RAX,qword ptr [RBP + local_18]
    004015ee MOV        RCX=>cpu,RAX
    004015f1 CALL       run
    004015f6 MOV        dword ptr [RBP + local_1c],EAX
    004015f9 CMP        dword ptr [RBP + local_1c],0x0
    004015fd JZ         LAB_0040160d
```

The first two lines stores a pointer to the "cpu" variable into "local_18". The next two lines prompts the user to enter a password. At **004015c2**, we store the memory address of global variable "input password” into RAX. Then we store whatever’s in RAX into RDX. Then we store some data into RCX and call *scanf*. Remember, the RCX register stores the first input into the function and the RDX stores the second input. So we’re taking user input and storing it into the “input_password” global variable, since the memory address of “input_password” gets stored in RDX, which is the second input for the *scanf* function.

At **004015d8**, we store the memory address of “input_password” global variable into RAX. Then we dereference that pointer and store the value of “input_password” into EDX. Then we store the memory address of “input_password” into RAX, and store whatever we have in EDX (which would be the value of input_password) into the memory location pointed to by RAX, in other words we store whatever’s in EDX into the “input_password” variable. I don’t understand the point of those few lines of instructions.

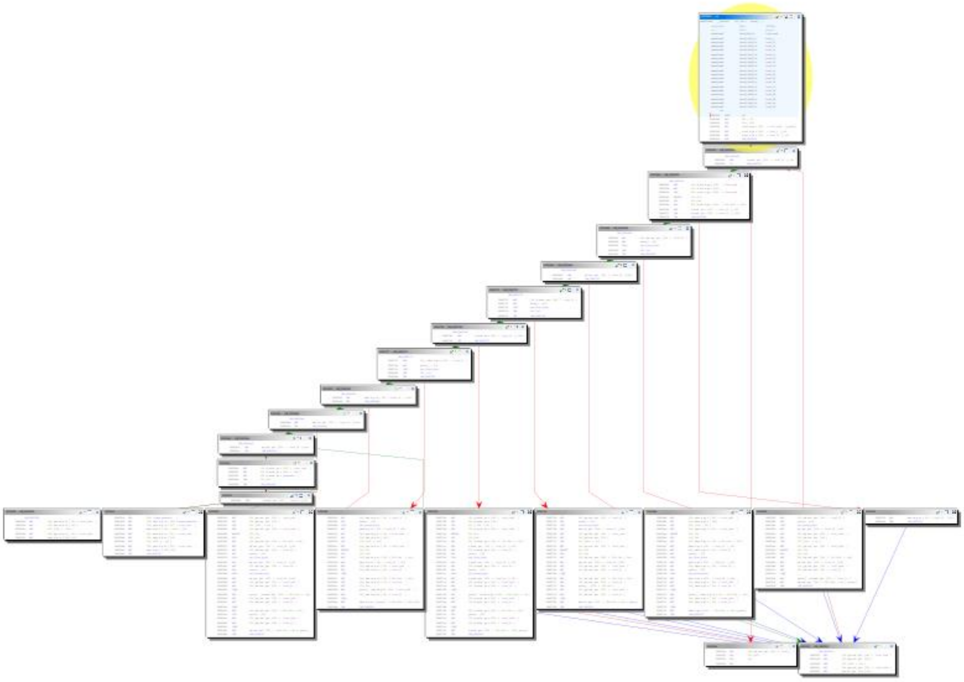
At **004015ea**, we store “local_18” into RAX. Remember, the “local_18” variable’s a pointer to the “cpu” structure. We then move the value in RAX into RCX, then the *run* function gets called. So the pointer to the “cpu” variable gets passed as an input into the *run* function.

At **004015f6**, the return value of the *run* function gets stored into the variable “local_1c”, then this gets compared with 0x0. Based on the result, a jump gets performed. Let’s investigate the next part.



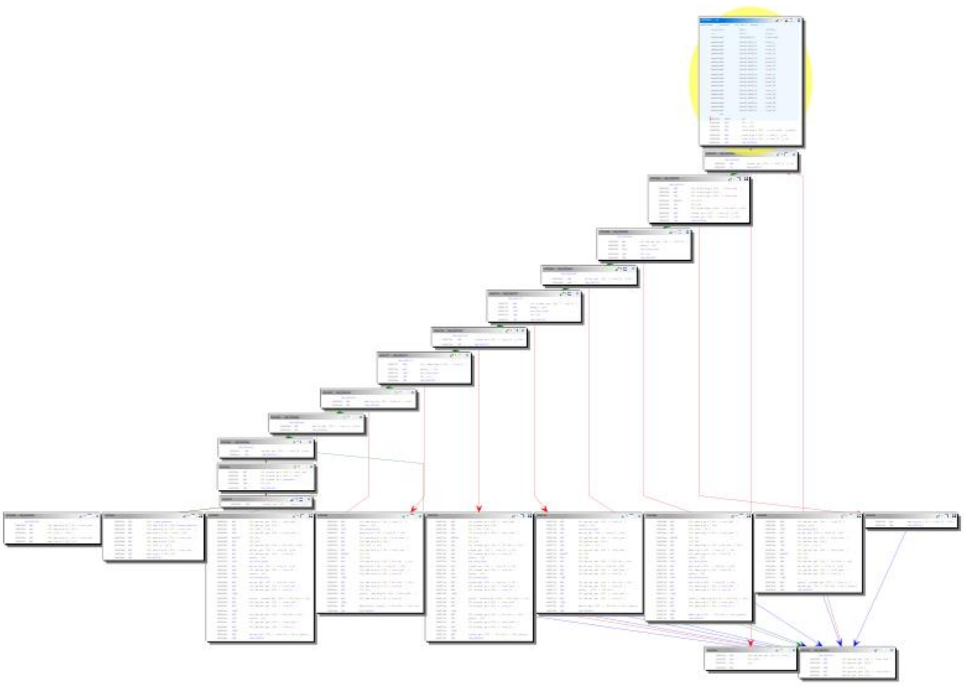
So, the jump determines whether the success message or the unfortunate message gets displayed. The final block of code just exits the program. The “VM” gets executed in the *run* function.

Let’s investigate the *run* function.



Run Function Analysis

In this section, only a brief explanation would be given regarding each block of code, since there's so many different blocks. Lets see the function graph again:



What happens is that the "cpu" structure selects a byte code. There's a huge *if else* statement, and depending on the result, one of the blocks at the start would be chosen. Then, the *virtual instruction pointer* would get incremented and the program returns to the start of the loop to fetch the new bytecode.

Here's some strategies we can use to solve VM based obfuscation:

- Program an emulator and dump intermediate values – we probably won't be doing this, since we have to figure out the password. If this was a challenge where you have to extract the flag, then we can use this strategy.
- Program a disassembler. This is the strategy we would use. After we program the disassembler, we can read this and figure out what the program's doing.

Here's the starting block, remember how the input parameter was a pointer to the "cpu" struct? That's why the variable's called "cpu_ptr".

```
        run
0040163a PUSH      RBP
0040163b MOV       RBP,RSP
0040163e SUB       RSP,0x70
00401642 MOV       qword ptr [RBP + cpu_ptr_local],cpu_ptr
00401646 MOV       dword ptr [RBP + local_c],0x0
0040164d MOV       dword ptr [RBP + vm_running],0x1
00401654 JMP       LAB_00401924
```

After the first block, we have this block:

```
00401646 MOV       dword ptr [RBP + local_c],0x0
0040164d MOV       dword ptr [RBP + vm_running],0x1
00401654 JMP       LAB_00401924

00401924 - LAB_00401924
        LAB_00401924
00401924 CMP       dword ptr [RBP + vm_running],0x1
00401928 JZ        LAB_00401659
```

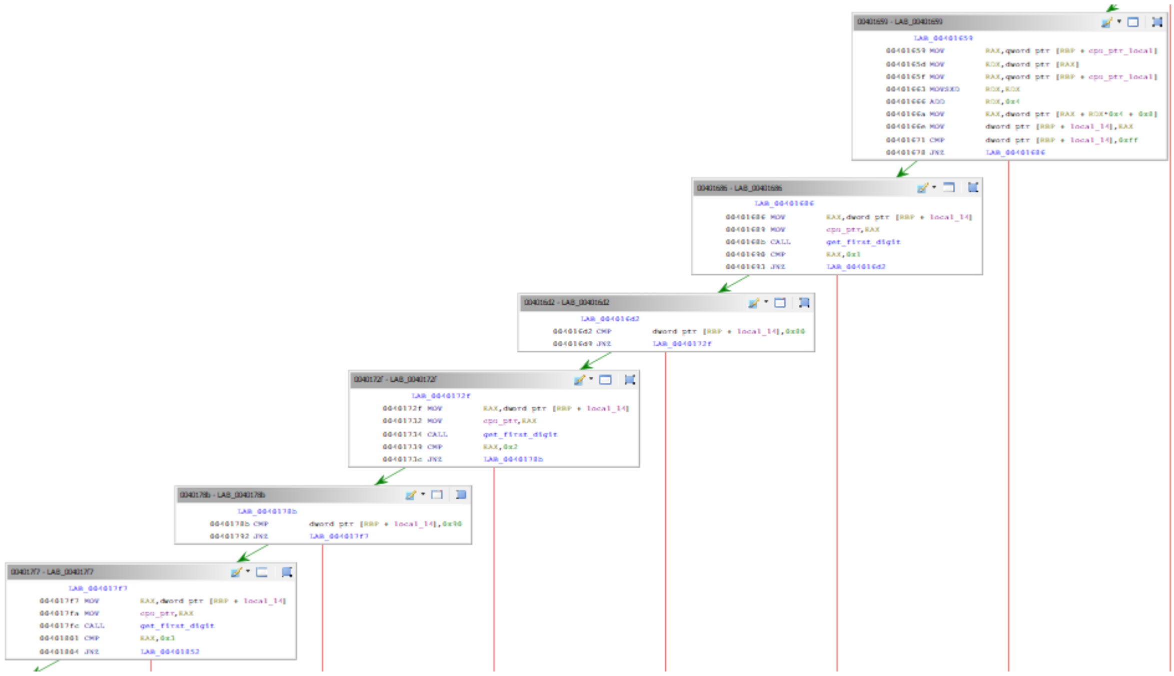
It we can see arrow entering from the first block and also on the bottom right. A comparison is done on "local_10" and a jump is done. This suggests that this is a loop. This would be the loop where the VM runs.

The "local_c" gets returned, so that's variable keeps track of whether true or false gets returned.

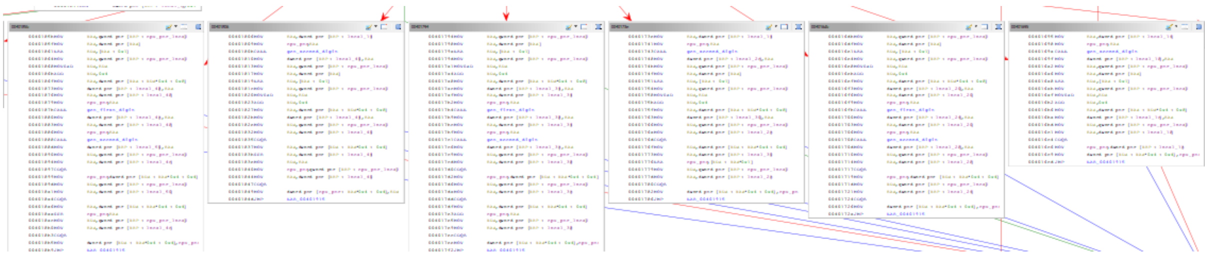
```
0040192e
0040192e MOV       EAX,dword ptr [RBP + local_c]
00401931 ADD       RSP,0x70
00401935 POP       RBP
00401936 RET
```

Rename to "success".

Then we have huge *if else* statement:



Which selects which section of the program to execute:



Let’s investigate the next block.

```
00401659 - LAB_00401659

            LAB_00401659
        00401659 MOV        RAX,qword ptr [RBP + cpu_ptr_local]
        0040165d MOV        EDX,dword ptr [RAX]
        0040165f MOV        RAX,qword ptr [RBP + cpu_ptr_local]
        00401663 MOVSXD     RDX,EDX
        00401666 ADD        RDX,0x4
        0040166a MOV        EAX,dword ptr [RAX + RDX*0x4 + 0x8]
        0040166e MOV        dword ptr [RBP + local_14],EAX
        00401671 CMP        dword ptr [RBP + local_14],0xff
        00401678 JNZ        LAB_00401686
```

From **00401659** to **0040165f**, it’s looking at the “cpu” structure, and saving the value of the “instruction_pointer” into EDX. It then stores the base address of the “cpu” structure in RAX. At **0040166a**, it seems to be accessing an array. Perhaps it’s accessing the bytecode array which is part of the “cpu” struct. We then store this bytecode into “local_14” variable and compare with 0xff, then jump based on that. When investigating further we can see that “local_14” just contains the current instruction. Now, if the variable equals to 0xff, then we jump to this block:

```
0040192e
        0040192e MOV        EAX,dword ptr [RBP + success]
        00401931 ADD        RSP,0x70
        00401935 POP        RBP
        00401936 RET
```

Which exits the function.

Let’s investigate the next *if else* block:

```
00401686 - LAB_00401686
                 LAB_00401686
        00401686 MOV        EAX,dword ptr [RBP + current_instruction]
        00401689 MOV        cpu_ptr,EAX
        0040168b CALL       get_first_digit
        00401690 CMP        EAX,0x1
        00401693 JNZ        LAB_004016d2
```

```
004016d2 - LAB_004016d2
                 LAB_004016d2
        004016d2 CMP        dword ptr [RBP + current_instruction],0x80
        004016d9 JNZ        LAB_0040172f
```

It moves the current instruction into EAX, moves that into “cpu_ptr” and then calls *get_first_digit* function. Then, it checks if the return value 0x1 and jumps based on that. Here’s the *get_first_digit* function:

```
00401937 - get_first_digit
int __fastcall get_first_digit(int param_1)
        int             EAX:4           <RETURN>
        int             ECX:4           param_1
        undefined4      Stack[0x8]:4    local_res8
                get_first_digit
        00401937 PUSH       RBP
        00401938 MOV        RBP,RSP
        0040193b MOV        dword ptr [RBP + local_res8],param_1
        0040193e MOV        EAX,dword ptr [RBP + local_res8]
        00401941 LEA        EDX,[RAX + 0xf]
        00401944 TEST       EAX,EAX
        00401946 CMOVS      EAX,EDX
        00401949 SAR        EAX,0x4
        0040194c POP        RBP
        0040194d RET
```

This just obtains the first digit of a 2 digit hexadecimal number.

Not sure why the function input for *get_first_digit* gets stored in "cpu_ptr" rather than RDX though. Maybe it's a mistake that Ghidra made? Other software such as IDA might not make this same mistake.

Let's investigate where this jumps to:

```
00401695
        00401695 MOV      EAX,dword ptr [RBP + current_instruction]
        00401698 MOV      cpu_ptr,EAX
        0040169a CALL     get_second_digit
        0040169f MOV      dword ptr [RBP + local_18],EAX
        004016a2 MOV      RAX,qword ptr [RBP + cpu_ptr_local]
        004016a6 MOV      EAX,dword ptr [RAX]
        004016a8 LEA      EDX,[RAX + 0x1]
        004016ab MOV      RAX,qword ptr [RBP + cpu_ptr_local]
        004016af MOVSXD   RDX,EDX
        004016b2 ADD      RDX,0x4
        004016b6 MOV      EAX,dword ptr [RAX + RDX*0x4 + 0x8]
        004016ba MOV      dword ptr [RBP + local_1c],EAX
        004016bd MOV      RDX,qword ptr [RBP + cpu_ptr_local]
        004016c1 MOV      EAX,dword ptr [RBP + local_18]
        004016c4 CDQE
        004016c6 MOV      cpu_ptr,dword ptr [RBP + local_1c]
        004016c9 MOV      dword ptr [RDX + RAX*0x4 + 0x4],cpu_ptr
        004016cd JMP      LAB_00401915
```

Again, we can see in this block of code that the input to *get_second_digit* function takes the input from "cpu_ptr". Let's investigate the *get_second_digit* function:

```
0040194e - get_second_digit

int __fastcall get_second_digit(int param_1)
     int              EAX:4          <RETURN>
     int              ECX:4          param_1
     undefined4       Stack[0x8]:4   local_res8
          get_second_digit
     0040194e PUSH       RBP
     0040194f MOV        RBP,RSP
     00401952 SUB        RSP,0x20
     00401956 MOV        dword ptr [RBP + local_res8],param_1
     00401959 MOV        param_1,dword ptr [RBP + local_res8]
     0040195c CALL       get_first_digit
     00401961 SHL        EAX,0x4
     00401964 MOV        EDX,dword ptr [RBP + local_res8]
     00401967 SUB        EDX,EAX
     00401969 MOV        EAX,EDX
     0040196b ADD        RSP,0x20
     0040196f POP        RBP
     00401970 RET
```

Getting the second digit of a 2 digit hexademical number.

Return back to where the program was:

```
00401695

00401695 MOV      EAX,dword ptr [RBP + current_instruction]
00401698 MOV      cpu_ptr,EAX
0040169a CALL     get_second_digit
0040169f MOV      dword ptr [RBP + second_digit_value],EAX
004016a2 MOV      RAX,qword ptr [RBP + cpu_ptr_local]
004016a6 MOV      EAX,dword ptr [RAX]
004016a8 LEA      EDX,[RAX + 0x1]
004016ab MOV      RAX,qword ptr [RBP + cpu_ptr_local]
004016af MOVSXD   RDX,EDX
004016b2 ADD      RDX,0x4
004016b6 MOV      EAX,dword ptr [RAX + RDX*0x4 + 0x8]
004016ba MOV      dword ptr [RBP + next_bytecode],EAX
004016bd MOV      RDX,qword ptr [RBP + cpu_ptr_local]
004016c1 MOV      EAX,dword ptr [RBP + second_digit_value]
004016c4 CDQE
004016c6 MOV      cpu_ptr,dword ptr [RBP + next_bytecode]
004016c9 MOV      dword ptr [RDX + RAX*0x4 + 0x4],cpu_ptr
004016cd JMP      LAB_00401915
```

The variables have been renamed. This block gets the second digit of the current instruction, then looks at the instruction pointer (remember the instruction pointer's at offset 0 on the cpu struct) and adds one to it, then stores that memory address into EDX. Then at **004016b6**, it accesses the next instruction from the bytecode array which is part of the "cpu" struct. This next bytecode gets stored into EAX, then it gets stored into the variable "next_bytecode". Then, the "next_bytecode" gets stored into an array at **004016c9**. At this instruction, RDX has the base address of the CPU struct, and RAX has the value of the second digit of the current instruction. So, the "next_bytecode" gets moved into an array, based on the second digit of the current instruction. Maybe the second digit talks about which register store the "next_bytecode"? So maybe the "cpu" struct has an array of registers. So this instruction would look something like this:

1x yz

The "1" determines that it's a move instruction. The x determines which register to move the value to, and the yz is the constant that will be moved to that register. After this block, we have this:

```
00401915 - LAB_00401915

         LAB_00401915
00401915 MOV      RAX,qword ptr [RBP + cpu_ptr_local]
00401919 MOV      EAX,dword ptr [RAX]
0040191b LEA      EDX,[RAX + 0x2]
0040191e MOV      RAX,qword ptr [RBP + cpu_ptr_local]
00401922 MOV      dword ptr [RAX],EDX
```

Which adds 2 to the instruction pointer.

After that block, we return to this block:

```
00401924 - LAB_00401924
                 LAB_00401924
        00401924 CMP          dword ptr [RBP + vm_running],0x1
        00401928 JZ           LAB_00401659
```

Then we can look at the next block in the *if else* statement:

```
004016d2 - LAB_004016d2
                 LAB_004016d2
        004016d2 CMP          dword ptr [RBP + current_instruction],0x80
        004016d9 JNZ          LAB_0040172f
```

So it compares “current_instruction” with 0x80, then jumps to the next *if else* if they don’t equal. If it does equal, then it will jump to another block:

```
004016db
        004016db MOV         RAX,qword ptr [RBP + cpu_ptr_local]
        004016df MOV         EAX,dword ptr [RAX]
        004016e1 LEA         EDX,[RAX + 0x1]
        004016e4 MOV         RAX,qword ptr [RBP + cpu_ptr_local]
        004016e8 MOVSXD      RDX,EDX
        004016eb ADD         RDX,0x4
        004016ef MOV         EAX,dword ptr [RAX + RDX*0x4 + 0x8]
        004016f3 MOV         dword ptr [RBP + local_20],EAX
        004016f6 MOV         EAX,dword ptr [RBP + local_20]
        004016f9 MOV         cpu_ptr,EAX
        004016fb CALL        get_first_digit
        00401700 MOV         dword ptr [RBP + local_24],EAX
        00401703 MOV         EAX,dword ptr [RBP + local_20]
        00401706 MOV         cpu_ptr,EAX
        00401708 CALL        get_second_digit
        0040170d MOV         dword ptr [RBP + local_28],EAX
        00401710 MOV         RDX,qword ptr [RBP + cpu_ptr_local]
        00401714 MOV         EAX,dword ptr [RBP + local_28]
        00401717 CDQE
        00401719 MOV         cpu_ptr,dword ptr [RDX + RAX*0x4 + 0x4]
        0040171d MOV         RDX,qword ptr [RBP + cpu_ptr_local]
        00401721 MOV         EAX,dword ptr [RBP + local_24]
        00401724 CDQE
        00401726 MOV         dword ptr [RDX + RAX*0x4 + 0x4],cpu_ptr
        0040172a JMP         LAB_00401915
```

From **004016db** to **004016f3**, it looking at the next instruction in the bytecode array which is part of the “cpu” struct, then stores this value into “local_20”. It then gets the first digit of this value and stores it into “local_24” then gets the second digit and stores it into “local_28”. Then, it takes this “local_28” value and takes a value from an array inside the “cpu” structure. Then at **00401726** it places this value in another location in the “cpu” structure. At **00401726**, the value of RAX is the first digit of the next bytecode. It looks like they're moving something from one register to another (there's an array of registers which stores the different registers). The instruction would look something like this:

80 xy

This would move the value in register y into register x.

Then, we move to these blocks again:

```
00401915 - LAB_00401915
                 LAB_00401915
        00401915 MOV          RAX,qword ptr [RBP + cpu_ptr_local]
        00401919 MOV          EAX,dword ptr [RAX]
        0040191b LEA          EDX,[RAX + 0x2]
        0040191e MOV          RAX,qword ptr [RBP + cpu_ptr_local]
        00401922 MOV          dword ptr [RAX],EDX
```

```
00401924 - LAB_00401924
                 LAB_00401924
        00401924 CMP          dword ptr [RBP + vm_running],0x1
        00401928 JZ           LAB_00401659
```

Let's investigate the next *if else* block:

```
0040172f - LAB_0040172f
                 LAB_0040172f
        0040172f MOV          EAX,dword ptr [RBP + current_instruction]
        00401732 MOV          cpu_ptr,EAX
        00401734 CALL         get_first_digit
        00401739 CMP          EAX,0x2
        0040173c JNZ          LAB_0040178b
```

So if the first digit of "current_instruction" equals 2, then it will proceed to this block:

```
0040173e
        0040173e MOV       EAX,dword ptr [RBP + current_instruction]
        00401741 MOV       cpu_ptr,EAX
        00401743 CALL      get_second_digit
        00401748 MOV       dword ptr [RBP + local_2c],EAX
        0040174b MOV       RAX,qword ptr [RBP + cpu_ptr_local]
        0040174f MOV       EAX,dword ptr [RAX]
        00401751 LEA       EDX,[RAX + 0x1]
        00401754 MOV       RAX,qword ptr [RBP + cpu_ptr_local]
        00401758 MOVSXD    RDX,EDX
        0040175b ADD       RDX,0x4
        0040175f MOV       EAX,dword ptr [RAX + RDX*0x4 + 0x8]
        00401763 MOV       dword ptr [RBP + local_30],EAX
        00401766 MOV       RDX,qword ptr [RBP + cpu_ptr_local]
        0040176a MOV       EAX,dword ptr [RBP + local_2c]
        0040176d CDQE
        0040176f MOV       EDX,dword ptr [RDX + RAX*0x4 + 0x4]
        00401773 MOV       EAX,dword ptr [RBP + local_30]
        00401776 LEA       cpu_ptr,[RDX + RAX*0x1]
        00401779 MOV       RDX,qword ptr [RBP + cpu_ptr_local]
        0040177d MOV       EAX,dword ptr [RBP + local_2c]
        00401780 CDQE
        00401782 MOV       dword ptr [RDX + RAX*0x4 + 0x4],cpu_ptr
        00401786 JMP       LAB_00401915
```

This block adds a constant to a register. Try to read the block and see if you can work that out. After this block, it proceeds back to these blocks again:

```
00401915 - LAB_00401915
              LAB_00401915
        00401915 MOV       RAX,qword ptr [RBP + cpu_ptr_local]
        00401919 MOV       EAX,dword ptr [RAX]
        0040191b LEA       EDX,[RAX + 0x2]
        0040191e MOV       RAX,qword ptr [RBP + cpu_ptr_local]
        00401922 MOV       dword ptr [RAX],EDX
```

```
00401924 - LAB_00401924
              LAB_00401924
        00401924 CMP       dword ptr [RBP + vm_running],0x1
        00401928 JZ        LAB_00401659
```

The next *if else* block:

```
0040178b - LAB_0040178b
              LAB_0040178b
        0040178b CMP       dword ptr [RBP + current_instruction],0x90
        00401792 JNZ       LAB_004017f7
```

The instruction block:

```
00401794

        00401794 MOV        RAX,qword ptr [RBP + cpu_ptr_local]
        00401798 MOV        EAX,dword ptr [RAX]
        0040179a LEA        EDX,[RAX + 0x1]
        0040179d MOV        RAX,qword ptr [RBP + cpu_ptr_local]
        004017a1 MOVSXD     RDX,EDX
        004017a4 ADD        RDX,0x4
        004017a8 MOV        EAX,dword ptr [RAX + RDX*0x4 + 0x8]
        004017ac MOV        dword ptr [RBP + local_34],EAX
        004017af MOV        EAX,dword ptr [RBP + local_34]
        004017b2 MOV        cpu_ptr,EAX
        004017b4 CALL       get_first_digit
        004017b9 MOV        dword ptr [RBP + local_38],EAX
        004017bc MOV        EAX,dword ptr [RBP + local_34]
        004017bf MOV        cpu_ptr,EAX
        004017c1 CALL       get_second_digit
        004017c6 MOV        dword ptr [RBP + local_3c],EAX
        004017c9 MOV        RDX,qword ptr [RBP + cpu_ptr_local]
        004017cd MOV        EAX,dword ptr [RBP + local_38]
        004017d0 CDQE
        004017d2 MOV        cpu_ptr,dword ptr [RDX + RAX*0x4 + 0x4]
        004017d6 MOV        RDX,qword ptr [RBP + cpu_ptr_local]
        004017da MOV        EAX,dword ptr [RBP + local_3c]
        004017dd CDQE
        004017df MOV        EAX,dword ptr [RDX + RAX*0x4 + 0x4]
        004017e3 ADD        cpu_ptr,EAX
        004017e5 MOV        RDX,qword ptr [RBP + cpu_ptr_local]
        004017e9 MOV        EAX,dword ptr [RBP + local_38]
        004017ec CDQE
        004017ee MOV        dword ptr [RDX + RAX*0x4 + 0x4],cpu_ptr
        004017f2 JMP        LAB_00401915
```

This adds one register to another register. Read the block and try to figure that out.

The next *if else* statement:

```
004017f7 - LAB_004017f7

                    LAB_004017f7
        004017f7 MOV        EAX,dword ptr [RBP + current_instruction]
        004017fa MOV        cpu_ptr,EAX
        004017fc CALL       get_first_digit
        00401801 CMP        EAX,0x3
        00401804 JNZ        LAB_00401852
```

The next instruction:

```
00401806

00401806 MOV     EAX,dword ptr [RBP + current_instruction]
00401809 MOV     cpu_ptr,EAX
0040180b CALL    get_second_digit
00401810 MOV     dword ptr [RBP + local_40],EAX
00401813 MOV     RAX,qword ptr [RBP + cpu_ptr_local]
00401817 MOV     EAX,dword ptr [RAX]
00401819 LEA     EDX,[RAX + 0x1]
0040181c MOV     RAX,qword ptr [RBP + cpu_ptr_local]
00401820 MOVSXD  RDX,EDX
00401823 ADD     RDX,0x4
00401827 MOV     EAX,dword ptr [RAX + RDX*0x4 + 0x8]
0040182b MOV     dword ptr [RBP + local_44],EAX
0040182e MOV     RDX,qword ptr [RBP + cpu_ptr_local]
00401832 MOV     EAX,dword ptr [RBP + local_40]
00401835 CDQE
00401837 MOV     EAX,dword ptr [RDX + RAX*0x4 + 0x4]
0040183b SUB     EAX,dword ptr [RBP + local_44]
0040183e MOV     EDX,EAX
00401840 MOV     cpu_ptr,qword ptr [RBP + cpu_ptr_local]
00401844 MOV     EAX,dword ptr [RBP + local_40]
00401847 CDQE
00401849 MOV     dword ptr [cpu_ptr + RAX*0x4 + 0x4],EDX
0040184d JMP     LAB_00401915
```

This subtracts a constant from register.

The next *if else* block:

```
00401852 - LAB_00401852

        LAB_00401852
00401852 CMP     dword ptr [RBP + current_instruction],0x91
00401859 JNZ     LAB_004018bb
```

The next instruction:

```
0040185b

        0040185b MOV        RAX,qword ptr [RBP + cpu_ptr_local]
        0040185f MOV        EAX,dword ptr [RAX]
        00401861 LEA        EDX,[RAX + 0x1]
        00401864 MOV        RAX,qword ptr [RBP + cpu_ptr_local]
        00401868 MOVSXD     RDX,EDX
        0040186b ADD        RDX,0x4
        0040186f MOV        EAX,dword ptr [RAX + RDX*0x4 + 0x8]
        00401873 MOV        dword ptr [RBP + local_48],EAX
        00401876 MOV        EAX,dword ptr [RBP + local_48]
        00401879 MOV        cpu_ptr,EAX
        0040187b CALL       get_first_digit
        00401880 MOV        dword ptr [RBP + local_4c],EAX
        00401883 MOV        EAX,dword ptr [RBP + local_48]
        00401886 MOV        cpu_ptr,EAX
        00401888 CALL       get_second_digit
        0040188d MOV        dword ptr [RBP + local_50],EAX
        00401890 MOV        RDX,qword ptr [RBP + cpu_ptr_local]
        00401894 MOV        EAX,dword ptr [RBP + local_4c]
        00401897 CDQE
        00401899 MOV        cpu_ptr,dword ptr [RDX + RAX*0x4 + 0x4]
        0040189d MOV        RDX,qword ptr [RBP + cpu_ptr_local]
        004018a1 MOV        EAX,dword ptr [RBP + local_50]
        004018a4 CDQE
        004018a6 MOV        EAX,dword ptr [RDX + RAX*0x4 + 0x4]
        004018aa SUB        cpu_ptr,EAX
        004018ac MOV        RDX,qword ptr [RBP + cpu_ptr_local]
        004018b0 MOV        EAX,dword ptr [RBP + local_4c]
        004018b3 CDQE
        004018b5 MOV        dword ptr [RDX + RAX*0x4 + 0x4],cpu_ptr
        004018b9 JMP        LAB_00401915
```

This block subtracts a register from another register.

The next *if else* block:

```
004018bb - LAB_004018bb

              LAB_004018bb
        004018bb CMP        dword ptr [RBP + current_instruction],0xaa
        004018c2 JNZ        LAB_004018e5
```

The next instruction:

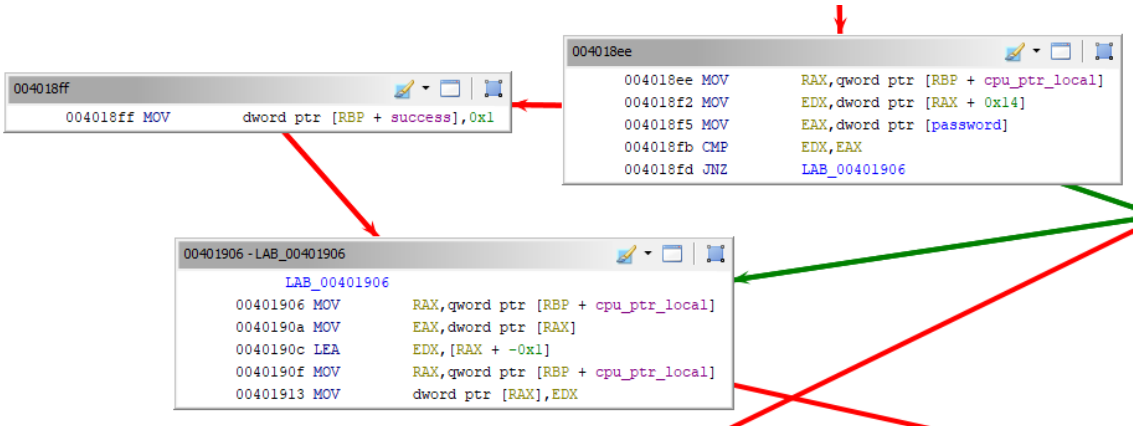
```
004018c4
        004018c4 LEA     RAX,[input_password]
        004018cb MOV     EDX,dword ptr [RAX]=>input_password
        004018cd MOV     RAX,qword ptr [RBP + cpu_ptr_local]
        004018d1 MOV     dword ptr [RAX + 0x14],EDX
        004018d4 MOV     RAX,qword ptr [RBP + cpu_ptr_local]
        004018d8 MOV     EAX,dword ptr [RAX]
        004018da LEA     EDX,[RAX + -0x1]
        004018dd MOV     RAX,qword ptr [RBP + cpu_ptr_local]
        004018e1 MOV     dword ptr [RAX],EDX
        004018e3 JMP     LAB_00401915
```

It's putting the user input into the "cpu" structure, but at offset 0x14. Remember that the first 4 bytes stores the instruction pointer. Then we have the registers array. There's 5 registers r0, r1, r2, r3 and r4. Each of these registers stores an int, therefore they're 4 bytes each. Therefore, to reach r4 we have an offset of 20, or 0x14 in hexadecimal. So it's storing the user input at r4. After that, we decrement the instruction pointer by 1. Remember, after each instruction block gets executed we increment the instruction pointer in the "cpu" struct by 2. However, since this instruction is only 1 byte, we need to subtract by 1 here.

The next *if else* statement:

```
004018e5 - LAB_004018e5
             LAB_004018e5
        004018e5 CMP     dword ptr [RBP + current_instruction],0xa1
        004018ec JNZ     LAB_00401915
```

The next instruction block:



This just checks r4 and checks if it's the same as the "password" global variable. If they're equal, the "success" variable gets set to 0x1. Then, the instruction pointer in the "cpu" struct gets decremented by 1 because this instruction is only 1 byte.

Let's summarize the instruction set:

- FF – stop running
- 1x yz – move constant "yz" into register x
- 80 xy – move register y into register x
- 2x yz – add constant "yz" into register x
- 90 xy – add register y into register x
- 3x yz – subtract constant from register x
- 91 xy – subtract register y from register x
- AA – set register 4 as the input password
- A1 – check if the value in register 4 matches the global variable "password"

So we have the instruction set. We also have the global variable "password" from the *.data* section:

```
                             password
00404070 e8 03 00 00         undefined4   000003E8h
00404074 00                  ??           00h
00404075 00                  ??           00h
00404076 00                  ??           00h
00404077 00                  ??           00h
00404078 00                  ??           00h
00404079 00                  ??           00h
0040407a 00                  ??           00h
0040407b 00                  ??           00h
0040407c 00                  ??           00h
0040407d 00                  ??           00h
0040407e 00                  ??           00h
0040407f 00                  ??           00h
```

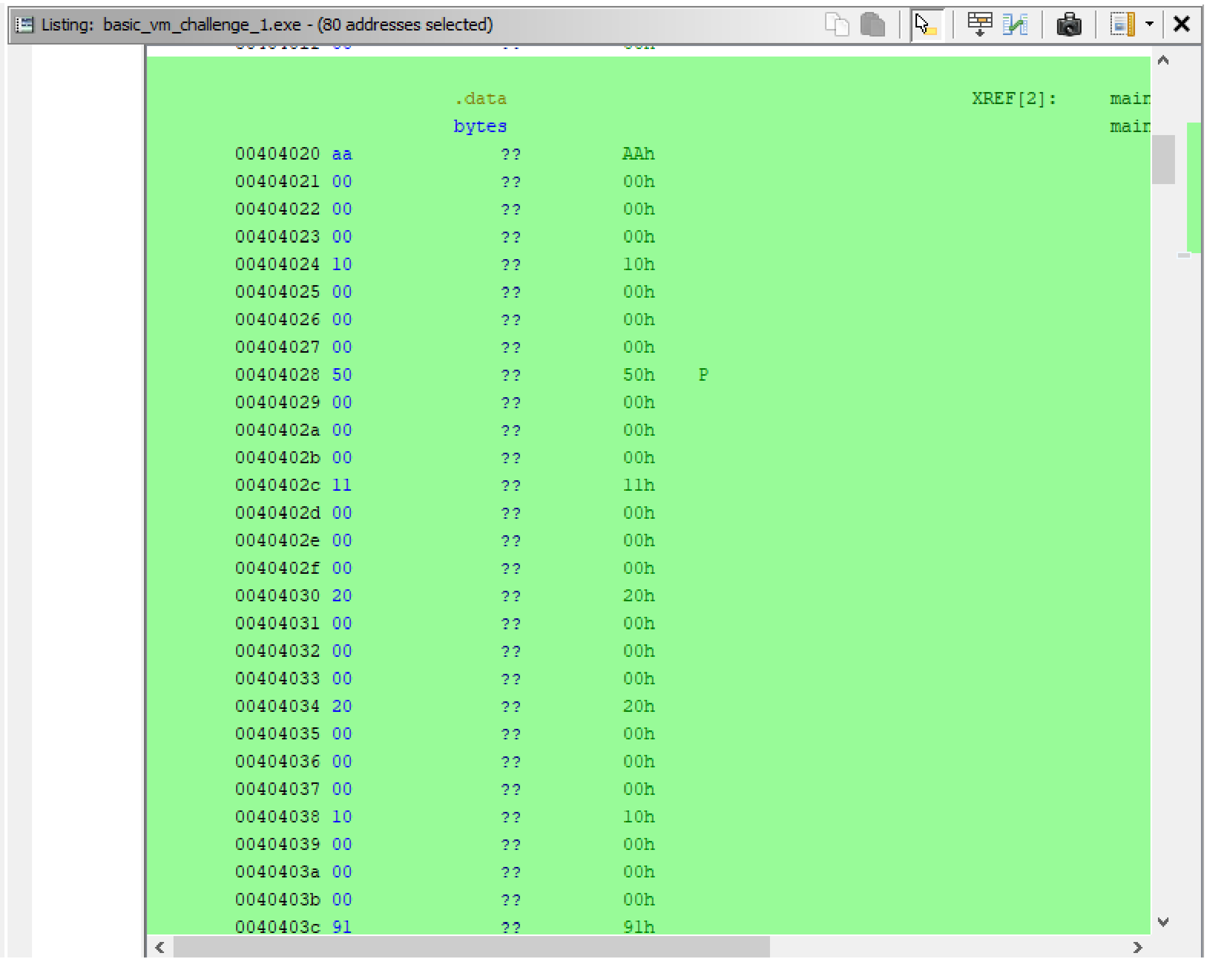
Try entering 1000 into the input:

```
Enter the password: 1000

Unfortunately thats the wrong password...
Closing program soon...
```

We have the original bytecode, which would get copied into the "cpu" structure:

```
                    .data
                    bytes
00404020 aa             ??         AAh
00404021 00             ??         00h
00404022 00             ??         00h
00404023 00             ??         00h
00404024 10             ??         10h
00404025 00             ??         00h
00404026 00             ??         00h
00404027 00             ??         00h
00404028 50             ??         50h    P
00404029 00             ??         00h
0040402a 00             ??         00h
0040402b 00             ??         00h
0040402c 11             ??         11h
0040402d 00             ??         00h
0040402e 00             ??         00h
0040402f 00             ??         00h
00404030 20             ??         20h
00404031 00             ??         00h
00404032 00             ??         00h
00404033 00             ??         00h
00404034 20             ??         20h
00404035 00             ??         00h
00404036 00             ??         00h
00404037 00             ??         00h
00404038 10             ??         10h
00404039 00             ??         00h
0040403a 00             ??         00h
0040403b 00             ??         00h
0040403c 91             ??         91h
```

Remember, when taking the bytecode we have to take the first value then skip 3, then take the next value then skip 3, because when programming this challenge the bytes were stored in an integer variable. However, another thing we can do is to highlight this entire data structure:



Then Right click then select *Data>int*. Now it looks nicer:

```
                  .data
                  bytes
00404020 aa 00 00 00    int         AAh
00404024 10 00 00 00    int         10h
00404028 50 00 00 00    int         50h
0040402c 11 00 00 00    int         11h
00404030 20 00 00 00    int         20h
00404034 20 00 00 00    int         20h
00404038 10 00 00 00    int         10h
0040403c 91 00 00 00    int         91h
00404040 01 00 00 00    int         1h
00404044 90 00 00 00    int         90h
00404048 40 00 00 00    int         40h
0040404c a1 00 00 00    int         A1h
00404050 ff 00 00 00    int         FFh
00404054 ff 00 00 00    int         FFh
00404058 00 00 00 00    int         0h
0040405c 00 00 00 00    int         0h
00404060 00 00 00 00    int         0h
00404064 00 00 00 00    int         0h
00404068 00 00 00 00    int         0h
0040406c 00 00 00 00    int         0h
```

So we can see that the bytecode looks like:

AA 10 50 11 20 20 10 91 01 90 40 A1 FF FF 00 00 00 00 00 00

With this information, we can start to write the disassembler. Or if you want you can analyse the bytecode and look at the instruction set information we figured out to determine what the program does.

Programming Disassembler

Python will get used to write the disassembler. Here's the bytecode array, instruction pointer and register array:

```
bytecode = [0xaa,0x10,0x50,0x11,0x20,0x20,0x10,0x91,0x01,0x90,0x40,0xa1,0xff,0xff,0x00,0x00,0x00,0x00,0x00,0x00]

instruction_pointer = 0x00
registers = [0x00,0x00,0x00,0x00,0x00]
```

Here's the helper functions that will be required:

```
def get_first_digit(number):

    return int(number/16)

def get_second_digit(number):

    return int(number - get_first_digit(number)*16)
```

The vm_running variable to keep track of whether the vm's running or not:

```
vm_running = 1
```

Then we have the vm code:

```
while vm_running:
    # Fetch instruction
    current_instruction = bytecode[instruction_pointer]

    # Decode instruction in the if else statement, then execute instruction
    if (current_instruction == 0xff):
        # Exit vm
        vm_running = 0
        print(f"vip: {hex(instruction_pointer)}  exit vm")
    elif (get_first_digit(current_instruction) == 0x01):
        # 1x yz
        # move constant yz into register x
        register_index = get_second_digit(current_instruction)
        value = bytecode[instruction_pointer+1]
        print(f"vip: {hex(instruction_pointer)}  mov r{register_index}, {hex(value)}")
    elif (current_instruction == 0x80):
        # 80 xy
        # move register y value into register x
        register_1_index = get_first_digit(bytecode(instruction_pointer+1))
        register_2_index = get_second_digit(bytecode(instruction_pointer+1))
        print(f"vip: {hex(instruction_pointer)}  mov r{register_1_index}, r{register_2_index}")

    elif (get_first_digit(current_instruction) == 0x2):
        # 2x yz
        # add constant yz to register x
        register_index = get_second_digit(current_instruction)
        value = bytecode[instruction_pointer+1]
        print(f"vip: {hex(instruction_pointer)}  add r{register_index}, {hex(value)}")
    elif (current_instruction == 0x90):
        # 90 xy
        # add value in register y to register x
        register_1_index = get_first_digit(bytecode[instruction_pointer+1])
        register_2_index = get_second_digit(bytecode[instruction_pointer+1])
        print(f"vip: {hex(instruction_pointer)}  add r{register_1_index}, r{register_2_index}")

    elif (get_first_digit(current_instruction) == 0x3):
        # 3x yz
        # subtract yz from register x
        register_index = get_second_digit(current_instruction)
        value = bytecode[instruction_pointer+1]
        print(f"vip: {hex(instruction_pointer)} sub r{register_index}, {value}")
    elif (current_instruction == 0x91):
        #91 xy
        # subtract value in register y from register x
        register_1_index = get_first_digit(bytecode[instruction_pointer+1])
        register_2_index = get_second_digit(bytecode[instruction_pointer+1])
        print(f"vip: {hex(instruction_pointer)}  sub r{register_1_index}, r{register_2_index}")
    elif (current_instruction == 0xaa):
        print(f"vip: {hex(instruction_pointer)}  mov r4, [input]")
        instruction_pointer -= 1
    elif (current_instruction == 0xa1):
        print(f"vip {hex(instruction_pointer)}  test r4, [password]")
        instruction_pointer -= 1

    instruction_pointer += 2
```

Here's the output of this:

```
Windows PowerShell
PS C:¥Users¥sc¥Desktop¥challenges¥walkthrough¥basic_vm_challenge_1> python .¥disassemble.py
vip: 0x0  mov r4, [input]
vip: 0x1  mov r0, 0x50
vip: 0x3  mov r1, 0x20
vip: 0x5  add r0, 0x10
vip: 0x7  sub r0, r1
vip: 0x9  add r4, r0
vip 0xb  test r4, [password]
vip: 0xc  exit vm
PS C:¥Users¥sc¥Desktop¥challenges¥walkthrough¥basic_vm_challenge_1>
```

So, now we can see what the program does.

- Move input into r4

- Move 0x50 into r0
- Mov 0x20 into r1
- Add 0x10 to r0
- Sub r1 from r0
- Add r0 to r4
- Test r4 with password

Note, we know from before that password equals 1000.

```
                              password
00404070 e8 03 00 00     undefined4  000003E8h
00404074 00              ??          00h
00404075 00              ??          00h
00404076 00              ??          00h
00404077 00              ??          00h
00404078 00              ??          00h
00404079 00              ??          00h
0040407a 00              ??          00h
0040407b 00              ??          00h
0040407c 00              ??          00h
0040407d 00              ??          00h
0040407e 00              ??          00h
0040407f 00              ??          00h
```

So we end up putting the input into r4, adding 0x40 into r4 and testing value in r4 equals 0x3e8 (1000), so the value we need to input would be 0x3e8 – 0x40, so 0x3a8 or 936 in decimal.

So try entering this value:

```
Enter the password: 936

Congratulations figuring out the password!
Closing program soon...
```

This password works. Note, if you wanted you didn't have to investigate the VM, you could have just brute forced the answer. However, it's useful to investigate for education purposes.

Python Program

```
def get_first_digit(number):

    return int(number/16)

def get_second_digit(number):

    return int(number - get_first_digit(number)*16)

bytecode =
[0xaa,0x10,0x50,0x11,0x20,0x20,0x10,0x91,0x01,0x90,0x40,0xa1,0xff,0xff,0x00,0x00,0x00,0x00,0x00
,0x00]

instruction_pointer = 0x00
registers = [0x00,0x00,0x00,0x00,0x00]



vm_running = 1



while vm_running:
    # Fetch instruction
    current_instruction = bytecode[instruction_pointer]

    # Decode instruction in the if else statement, then execute instruction
    if (current_instruction == 0xff):
        # Exit vm
        vm_running = 0
        print(f"vip: {hex(instruction_pointer)}  exit vm")
    elif (get_first_digit(current_instruction) == 0x01):
```

```
    # 1x yz
    # move constant yz into register x
    register_index = get_second_digit(current_instruction)
    value = bytecode[instruction_pointer+1]
    print(f"vip: {hex(instruction_pointer)}  mov r{register_index}, {hex(value)}")
elif (current_instruction == 0x80):
    # 80 xy
    # move register y value into register x
    register_1_index = get_first_digit(bytecode(instruction_pointer+1))
    register_2_index = get_second_digit(bytecode(instruction_pointer+1))
    print(f"vip: {hex(instruction_pointer)}  mov r{register_1_index}, r{register_2_index}")


elif (get_first_digit(current_instruction) == 0x2):
    # 2x yz
    # add constant yz to register x
    register_index = get_second_digit(current_instruction)
    value = bytecode[instruction_pointer+1]
    print(f"vip: {hex(instruction_pointer)}  add r{register_index}, {hex(value)}")
elif (current_instruction == 0x90):
    # 90 xy
    # add value in register y to register x
    register_1_index = get_first_digit(bytecode[instruction_pointer+1])
    register_2_index = get_second_digit(bytecode[instruction_pointer+1])
    print(f"vip: {hex(instruction_pointer)}  add r{register_1_index}, r{register_2_index}")
elif (get_first_digit(current_instruction) == 0x3):
    # 3x yz
    # subtract yz from register x
    register_index = get_second_digit(current_instruction)
    value = bytecode[instruction_pointer+1]
    print(f"vip: {hex(instruction_pointer)} sub r{register_index}, {value}")
elif (current_instruction == 0x91):
```

```
    #91 xy
    # subtract value in register y from register x
    register_1_index = get_first_digit(bytecode[instruction_pointer+1])
    register_2_index = get_second_digit(bytecode[instruction_pointer+1])
    print(f"vip: {hex(instruction_pointer)}  sub r{register_1_index}, r{register_2_index}")
elif (current_instruction == 0xaa):
    print(f"vip: {hex(instruction_pointer)}  mov r4, [input]")
    instruction_pointer -= 1
elif (current_instruction == 0xa1):
    print(f"vip {hex(instruction_pointer)}  test r4, [password]")
    instruction_pointer -= 1

instruction_pointer += 2
```