

## Flare-On 3: Challenge 8 Solution - CHIMERA.EXE

Challenge Author: Nick Harbour (@nickharbour)

*chi-me-ra*  
*kī'mirā,kā'mirā/*

*noun*

- 1. an imaginary monster compounded of incongruous parts*
- 2. an individual, organ, or part consisting of tissues of diverse genetic constitution*

Chimera is both a valid Win32 binary as well as a valid DOS binary. Reverse engineering the Win32 half of the binary proves to be very easy but yields the key “this is the wrong password” which, due to a checksum, will still cause the program to output “You have fail. Please try harder.” The message “You have succeed” lies just before it in the binary, taunting you.

This challenge preys on the reverse engineer’s tendency to jump ahead, and ignore the usually mundane file format details and metadata. This is a very small binary (2.5KB) and running strings over it only yields 49 lines. Let’s examine the very first line of this output:

```
!This program cannot not be run in DOS mode.
```

Figure 1 - DOS Stub String from CHIMERA.EXE

There is a big clue, in the form of a double negative, about how you need to run this binary to solve the challenge. You need to run it in 16-bit DOS instead of Windows. If you are curious about this artifact, remember that most 32-bit Windows programs have a very small 16-Bit DOS Program inside them. The 32-bit PE File Format is built upon the old 16-bit executable format, so if you run a 32-bit program inside of 16-bit DOS, it runs a small program that displays the string “This program cannot be run in DOS mode.” and then exits. Let’s examine the 16-bit DOS Stub program from CHIMERA.EXE by loading it into IDA Pro.

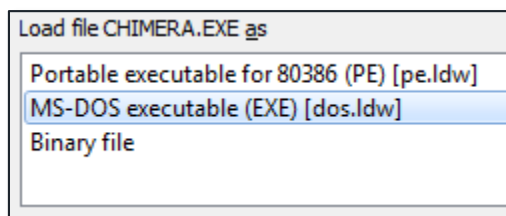


Figure 2 - IDA Pro Loading Options for CHIMERA.EXE

Care must be taken at the loading screen of IDA Pro, as its default options will not represent any of CHIMERA's DOS functionality. You must manually select MS-DOS executable from the top of the loading options dialog. This will cause IDA Pro to disassemble the DOS Stub program and operate in 16-bit disassembly mode for the duration of the file. Figure 3 shows the disassembled DOS Stub for CHIMERA.EXE.

```

push    cs
pop     ds

mov     dx, 11h
mov     ah, 9
int     21h                ; DOS - PRINT STRING
                          ; DS:DX -> string terminated by "$"

jmp     loc_107C6
    
```

Figure 3 - Disassembled DOS Stub from CHIMERA.EXE

We will compare this code to that of a standard DOS Stub program (this one taken from notepad.exe) below in Figure 4.

```

push    cs
pop     ds

mov     dx, 0Eh
mov     ah, 9
int     21h                ; DOS - PRINT STRING
                          ; DS:DX -> string terminated by "$"

mov     ax, 4C01h
int     21h                ; DOS - 2+ - QUIT WITH EXIT CODE (EXIT)
                          ; AL = exit code
    
```

Figure 4 - Disassembled DOS Stub from notepad.exe

Both samples perform the INT 21h with the AH register set to 9, which is a DOS system call to display a string to the screen. The fragment from notepad.exe then executes the function 4Ch (exit) whereas the CHIMERA.EXE sample jumps to a mysterious location.

Before you proceed too far forward in your analysis you may wish to get a DOS analysis environment up and running. I recommend using DOSBox and finding a version of the venerable DOS Debugger and

WinDBG ancestor DEBUG.COM. Once you get up and running and launch CHIMERA.EXE in DOS the results may be initially discouraging, as shown in Figure 5.

```
C:\>CHIMERA.EXE
This program cannot not be run in DOS mode.
C:\>
```

Figure 5 - CHIMERA.EXE Output

You're on the right path though, even if there is no password prompt yet. Let's go back to the disassembly to figure out how to proceed. If we follow the `jmp` instruction at the end of the DOS Stub program it takes us to the following fragment of code shown in Figure 6.

```
loc_107C6:      mov     cx, 70h ; 'p' ; CODE XREF: seg000:0009
loc_107C9:      mov     bx, cx ; CODE XREF: seg000:07D2
                dec     bx
                add     bx, bx
                add     [bx+7D4h], cx
                loop   loc_107C9
loc_107D4:      jmp     short near ptr loc_107D4+1 ; CODE XREF: seg000:loc_107D4
```

Figure 6 - Unpack loop in CHIMERA.EXE

This fragment of code is an unpacker loop that will decode the remainder of the DOS program. This small algorithm simply adds a value to every WORD starting at `loc_107D4`, for 112 (0x70) iterations (224 bytes). The following fragment of IDAPython code will perform this operation in your IDA database so you don't have to worry about dumping from DEBUG.COM.

```
for i in xrange(1, 0x70+1):
    ea = 0x107D4 + i*2
    wval = Word(ea)
    wval += i+1
    PatchWord(ea, wval)
```

Figure 7 - IDAPython Script to Unpack CHIMERA.EXE

Once the program is decoded IDA will still not disassemble it correctly. This is due to the use of Anti -

Disassembly caltrops sprinkled throughout the code. The first is the `jmp` instruction at 107D4 which is shown at the very end of the disassembly fragment shown previously in Figure 6. This instruction is a two byte jump instruction whose target is the second byte of itself, therefore the second byte of this instruction is the first byte of the next instruction. It is impossible to display both this instruction and the next instruction at the same time because they share the same byte. You can use the ‘d’ key in IDA pro to change this instruction to data and then use the ‘c’ key to change the second byte into an instruction if you place your cursor there. If done correctly, the instruction at address 107D5 should be “`inc ax`”.

Starting at address 107D7 is a fragment of code, which may explain why we aren’t getting a command prompt.

```

seg000:07D7  mov     ax, 2A00h
seg000:07DA  int     21h           ; DOS - GET CURRENT DATE
seg000:07DA                      ; Return: DL = day, DH = month, CX = year
seg000:07DA                      ; AL = day of the week (0=Sunday, 1=Monday, etc.)
seg000:07DC  sub     cx, 1990
seg000:07E0  jg      loc_108B0

```

Figure 8 - Year Check code in CHIMERA.EXE

The first system call in this fragment is AH=2Ah which is the Get Current Date command. It will populate the current year into the CX register. The instruction at address 107DC compares the date with 1990 and 107E0 jumps to another location if the date was greater than 1990. The code that it jumps to will simply exit the program. So to continue analysis we either need to modify the program or set our clock back to the 1980’s! I went for the easy route and set the clock back. If you have done so you should see the following promising sign when you launch CHIMERA.EXE, shown in Figure 9 below.

```

C:\>CHIMERA.EXE
This program cannot not be run in DOS mode.
This one's for the geezers.
Enter the password>

```

Figure 9 - DOS Mode input prompt from CHIMERA.EXE

This should look familiar if you ran CHIMERA.EXE in Windows, as it is the exact same prompt though printed to the screen using entirely different APIs. It should be noted that strings in Windows are terminated with null bytes and strings in DOS are terminated by the ‘\$’ character, but these strings

have both at the end so they may be used in either. If we follow the code immediately after the date check (not the branch that is for years over 1990) we find the following fragment.

```

seg000:07EC loc_107EC:                                ; CODE XREF: seg000:07E7
seg000:07EC                                ; seg000:07E9
seg000:07EC          mov     ah, 9
seg000:07EE          int     21h          ; DOS - AH=9 Print String
seg000:07F0          mov     dx, 75Eh
seg000:07F3          mov     ah, 0Ah
seg000:07F5          jnz    short loc_107FB
seg000:07F7          jz     short loc_107FB
seg000:07F7 ; -----
seg000:07FB loc_107FB:                                ; CODE XREF: seg000:07F5
seg000:07FB                                ; seg000:07F7
seg000:07FB          int     21h          ; DOS - AH=A Read Keyboard Input
seg000:07FB

```

Figure 10 - Prompt and Keyboard input sequence in CHIMERA.EXE

This code displays the prompt and performs a buffered keyboard input, storing the length of the input received at memory address 1075F and the buffer of input data at 10760. Once we have isolated the input data we can begin to determine how the program operates on the data to determine if it is the correct key or not.

CHIMERA does not decode the answer key in memory but rather encodes your input using the same method the key is encoded with and then compares the two. Determining the plaintext input needed to produce the correct answer will therefore involve understanding the logic in the program and producing the inverse algorithm in a script. For the sake of brevity, all disassembly fragments shown will have the anti-disassembly code removed and a single comment line “; -----” will be left in its place.

```

seg000:0802 loc_10802:                                ; CODE XREF: seg000:07FF
seg000:0802          mov     dl, 97h ; 'ù'
seg000:0804          mov     ah, byte ptr bInputLength
seg000:0808
seg000:0808 loc_10808:                                ; CODE XREF: seg000:084C
seg000:0808          cmp     cl, ah
seg000:080A          jz     short loc_1084E

```

Figure 11 - Encoding Loop Start

In Figure 11 we see the beginning of the loop to encode the user-supplied input in the same manner as

the key. Once this loop is finished then the entire input will be compared with the entire key. The loop starts from the end of the input buffer and works its way back to the beginning, with each byte being XOR'ed with a rotated and cyphered version of the previous byte. The value 97h seen in the first line of this fragment serves as the “previous” value for the first iteration of the loop. The loop starts at loc\_10808 with the value in the CL register serving as the loop counter with the AH register serving as the maximum loop count, which received its value from the DOS function which received the keyboard input.

```

seg000:080F      mov     al, ah
seg000:0811      dec     al
seg000:0813      sub     al, cl
seg000:0815      test    cl, cl
seg000:0817      jz      short loc_1082D
seg000:0817 ; -----
seg000:081E      mov     bx, 760h           ; readbuffer
seg000:0821      movzx  dx, ah
seg000:0824      add     bx, dx
seg000:0829 ; -----
seg000:0829      sub     bx, cx
seg000:082B      mov     dl, [bx]

```

Figure 12 - Getting a character of Input

The next code the program comes to, shown in Figure 12, will set the BX register to point to the character of input specified by the loop counter. This starts at the end of the input and works its way back to the beginning of the input with each loop iteration. The buffer of input data is located at address 10760 as indicated by the instruction at line labeled “seg000:081E”. The final line of this fragment is retrieving the character and loading into the DL register where it will be used in the next sequence of code shown below in Figure 13.

```

seg000:082D loc_1082D:                ; CODE XREF: seg000:0817
seg000:082D      rol     dl, 3
seg000:0830      movzx  bx, dl
seg000:0830 ; -----
seg000:0838      movzx  bx, byte ptr [bx+461h]
seg000:083D      mov     dl, [bx+461h]
seg000:0841      movzx  bx, al
seg000:0844      xor     [bx+760h], dl
seg000:0848      inc     cx
seg000:0848 ; -----
seg000:084C      jmp     short loc_10808

```

Figure 13 - First Cypher loop in CHIMERA.EXE

The input byte is first rotated left by bits as shown at loc\_1082D. It is then used as an index into data at address 10461 as shown in line 10838. Since the possible byte values can be any valid 8 bit number,

there must be at least 256 bytes at this address. This data actually contains part of the valid 32-bit portion of the program. The next instruction retrieves a byte from this location. Line 10844 XORs this byte with the byte of input data we are pointing to this iteration of the loop. In Figure 14 is a Python array of the 256 bytes of data found at location 10461, which serves as a cypher.

```
cypher = [255, 21, 116, 32, 64, 0, 137, 236, 93, 195, 66, 70, 192, 99, 134, 42,
171, 8, 191, 140, 76, 37, 25, 49, 146, 176, 173, 20, 162, 182, 103, 221, 57, 216,
95, 63, 123, 92, 194, 178, 246, 46, 117, 155, 97, 148, 207, 206, 106, 152, 80, 242,
91, 240, 69, 48, 14, 56, 235, 59, 108, 102, 127, 36, 61, 223, 136, 151, 185, 179,
241, 203, 131, 153, 26, 13, 239, 177, 3, 85, 158, 154, 122, 16, 224, 54, 232, 211,
228, 50, 193, 120, 7, 183, 107, 199, 112, 201, 44, 160, 145, 53, 109, 254, 115, 94,
244, 164, 217, 219, 67, 105, 245, 141, 238, 68, 125, 72, 181, 220, 75, 2, 161, 227,
210, 166, 33, 62, 47, 163, 215, 187, 132, 90, 251, 143, 18, 28, 65, 40, 197, 118,
89, 156, 247, 51, 6, 39, 10, 11, 175, 113, 22, 74, 233, 159, 79, 111, 226, 15, 190,
43, 231, 86, 213, 83, 121, 45, 100, 23, 149, 167, 189, 124, 29, 88, 147, 165, 101,
248, 24, 19, 234, 188, 229, 243, 55, 4, 150, 168, 30, 1, 41, 130, 81, 60, 104, 31,
142, 218, 138, 5, 34, 114, 73, 250, 135, 169, 84, 98, 198, 170, 9, 180, 253, 214,
209, 172, 133, 17, 71, 58, 157, 230, 77, 27, 204, 82, 128, 35, 252, 237, 139, 126,
96, 205, 110, 87, 186, 222, 174, 202, 196, 119, 12, 78, 212, 208, 200, 225, 184,
249, 38, 144, 129, 52]
```

Figure 14 - Cypher data for CHIMERA.EXE

A python representation of this loop is shown in Figure 15.

```
for i in xrange(len(input_buffer)) :
    if i == 0:
        prev = 0x97
    else:
        prev = input_buffer[len(input_buffer) - i]
    input_buffer[len(input_buffer) - (1 + i)] ^= cypher[cypher[rol_byte(prev, 3)]]
```

Figure 15 - Python translation of Cypher Loop

The next sequence of code is a straight-forward rolling XOR loop over the input data, which was already altered by the previous loop. This loop XORs each byte with the previous, and the very first byte with C5h. The disassembly for this sequence is provided in Figure 16 and a Python translation of this code is provided as well in Figure 17.

```

seg000:084E loc_1084E:                                ; CODE XREF: seg000:080A
seg000:084E                xor     cx, cx
seg000:084E ; -----
seg000:0854                mov     dl, 0C5h ; '+'
seg000:0856 loc_10856:                                ; CODE XREF: seg000:0873
seg000:0856                cmp     cl, ah
seg000:0858                jz     short loc_10875
seg000:085A                test    cl, cl
seg000:085C                jz     short loc_10869
seg000:085C ; -----
seg000:0862                mov     bx, cx
seg000:0864                dec     bx
seg000:0865                mov     dl, [bx+760h]
seg000:0869 loc_10869:                                ; CODE XREF: seg000:085C
seg000:0869                mov     bx, cx
seg000:086B                xor     [bx+760h], dl
seg000:086F                inc     cx
seg000:086F ; -----
seg000:0873                jmp    short loc_10856

```

Figure 16 - Rolling XOR loop in CHIMERA.EXE

```

for i in xrange(len(input_buffer)):
    if i == 0:
        prev = 0xC5
    else:
        prev = input_buffer[i-1]
    input_buffer[i] ^= prev

```

Figure 17 - Python translation of the Reverse XOR Loop

The cypher translation and rolling XOR are the last of the input manipulation. Now it's time for CHIMERA to compare the input you supplied with the correct answer. It accomplishes this by XORing every byte of modified input with every byte of the correct answer key, and adding the result to a running total. If the result of the running total is zero, then it's a match! You can see from the code in Figure 18 that its XOR a byte of data from address 10760 (the input buffer) with a byte of data from address 107AC, which must be our answer key. A rough Python translation of this code is also provided in Figure 19.



```

seg000:087C          xor     cx, cx
seg000:087E  loc_1087E:          ; CODE XREF: seg000:0894
seg000:087E          cmp     bx, 1Ah
seg000:0881          jz     short loc_10896
seg000:0881 ; -----
seg000:0886          mov     cl, [bx+760h]
seg000:088A          xor     cl, [bx+7ACh]
seg000:088E          add     ax, cx
seg000:0890          inc     bx
seg000:0890 ; -----
seg000:0894          jmp     short loc_1087E
seg000:0896  loc_10896:          ; CODE XREF: seg000:0881
seg000:0896          test    ax, ax
seg000:0898          jz     short near ptr success_case

```

Figure 18 - Comparison loop in CHIMERA.EXE

```

running_total = 0
if len(realkey_bytes) <= len(inputdata_bytes):
    for i in xrange(len(realkey_bytes)):
        running_total += realkey_bytes[i] ^ inputdata_bytes[i]
else:
    running_total = 1

if running_total == 0:
    print "Correct Key"
else:
    print "Wrong Key"

```

Figure 19 - Python Translation of Comparison Loop

If we take the cyphered key data from offset 107AC (1Ah bytes worth) and produce the inverse logic that is used against the input data, we can deduce the plaintext key. In other words, the input we need to supply to make the program print Success. The script shown in Figure 20 is the end result of that, and will print the plain text solution based on the encoded key found in the binary.

```
cyphered_key = [56, 225, 74, 27, 12, 26, 70, 70, 10, 150, 41, 115, 115, 164, 105,
3, 0, 27, 168, 248, 184, 36, 22, 214, 9, 203]

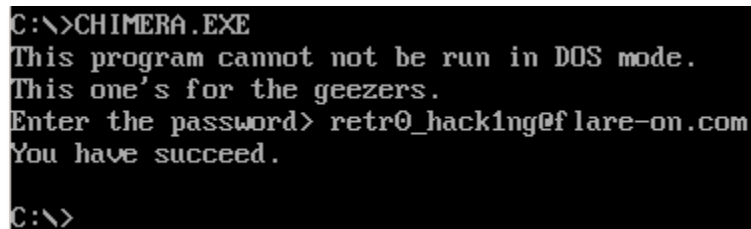
#undo rolling XOR loop
for i in xrange(len(cyphered_key)):
    if i == len(cyphered_key) - 1:
        prev = 0xC5
    else:
        prev = cyphered_key[len(cyphered_key) - (2 + i)]
        cyphered_key[len(cyphered_key) - (1 + i)] ^= prev

#undo cypher lookup loop
for i in xrange(len(cyphered_key)):
    if i == len(cyphered_key) - 1:
        next = 0x97
    else:
        next = cyphered_key[i + 1]
        cyphered_key[i] ^= cypher[cypher[rol_byte(next, 3)]]

#print out the answer!
print ''.join([chr(c) for c in cyphered_key])
```

Figure 20 - Answer Decoding Script

This script should produce the correct answer “retr0\_hack1ng@flare-on.com”. If we try it out on our DOSBox instance with the clock set to the 1980’s, we should feel the warm comfort of our success.



```
C:\>CHIMERA.EXE
This program cannot not be run in DOS mode.
This one's for the geezers.
Enter the password> retr0_hack1ng@flare-on.com
You have succeed.
C:\>
```

Figure 21 - Successful Password output from CHIMERA.EXE