FLARE-On 4: Challenge 3 Solution – greek_to_me.exe

Challenge Author: Matt Williams (@0xmwilliams)

`greek_to_me.exe` is a Windows x86 executable whose strings reveal what is likely the desired state of the program at virtual address `0x401101`, shown in Figure 1.

```
004010F5 push    0                ; flags
004010F7 push    2Bh              ; len
004010F9 push    offset aCongratulation ; "Congratulations! But wait, where's..."
004010FE push    [ebp+s]          ; s
00401101 call    ds:send
```

Figure 1 – Challenge completion string

However, the disassembly preceding this address contains odd assembly instructions, shown in Figure 2.

```
004010A0 icebp
004010A1 push    es
004010A2 sbb     dword ptr [esi], 1F99C4F0h
004010A8 les     edx, [ecx+1D81061Ch]
004010AE out     6, al            ; DMA controller, 8237A-5.
004010AE                          ; channel 3 base address
004010AE                          ; (also sets current ad
```

Figure 2 – Instructions preceding desired end-state

At this stage you may have correctly assumed the sample modifies these instructions in order to properly reach `0x401101`. If these instructions execute in their current state the program will likely crash. Another indication of self-modifying code is found in the sample's PE headers. The `.text` section, where the program's entry point resides, is writeable. From here you may have worked backward to determine what causes the program to take the preferred branch at `0x401063`. Another approach involves determining where the socket was created. Let's explore the latter approach.

`greek_to_me.exe` contains a single call to the `socket` function at `0x401151`, shown in Figure 3. Within the `sub_401121` function we observe the sample creating a listening socket on TCP port `2222` (`0x8AE`) using a standard series of Windows API functions: `socket`, `bind`, `listen`, and `accept`.

```
00401147 loc_401147:
00401147 push    esi
00401148 push    edi
00401149 push    6               ; protocol
0040114B push    1               ; type
0040114D push    2
0040114F pop     edi
00401150 push    edi             ; af
00401151 call    ds:socket
00401157 mov     esi, eax
00401159 cmp     esi, 0FFFFFFFFh
0040115C jz      short loc_4011D8
```

```
0040115E push    offset cp       ; "127.0.0.1"
00401163 mov     [ebp+name.sa_family], di
00401167 call    ds:inet_addr
0040116D push    2222            ; hostshort
00401172 mov     dword ptr [ebp+name.sa_data+2], eax
00401175 call    ds:htons
0040117B mov     word ptr [ebp+name.sa_data], ax
0040117F lea     eax, [ebp+name]
00401182 push    10h             ; namelen
00401184 push    eax             ; name
00401185 push    esi             ; s
00401186 call    ds:bind
0040118C cmp     eax, 0FFFFFFFFh
0040118F jz      short loc_4011D1
```

Figure 3 – Socket creation

The sample waits for a connection on the listening port before attempting to receive a maximum of four bytes from the connected client. Received bytes are stored in a buffer passed into `sub_401121` as its single argument. If at least one byte is received, the function returns a socket handle without tearing down the established connection. Note this socket handle may be used later in the program's execution at `0x401071` or `0x401101`.

Execution continues if `sub_401121` returns a valid socket handle, otherwise the sample exits. The next basic block shown in Figure 4 populates registers used in the sample's decoding loop:

```
00401029 mov     ecx, offset loc_40107C
0040102E add     ecx, 79h
00401031 mov     eax, offset loc_40107C
00401036 mov     dl, [ebp+buf]
```

Figure 4 – Populating registers prior to the loop

First, an address of executable code in the `.text` section (`0x40107C`) is moved into the `ECX` register and a constant value (`0x79`) is added to it. This reflects the "stop" address for the decoding loop described below. The address `0x40107C` is moved into the EAX register, representing the start address for the decoding loop. At `0x401036`, the first byte from the `recv` buffer is moved into the lower eight bits of the `EDX` register.

The next basic block, shown in Figure 5, contains a loop that performs the following operations:

1) Extract a single byte at the address stored in `EAX` (`0x40107C`)
2) XOR the extracted byte with the first byte received over the listening socket
3) Add `0x22` to the result of the XOR operation
4) Use the resulting byte to overwrite the byte extracted in Step 1

```
00401039 loc_401039:
00401039      mov     bl, [eax]
0040103B      xor     bl, dl
0040103D      add     bl, 22h
00401040      mov     [eax], bl
00401042      inc     eax
00401043      cmp     eax, ecx
00401045      jl      short loc_401039
```

Figure 5 – Self-modifying code

The address stored in `EAX` is incremented by one and compared to the maximum address stored in `ECX`. The loop continues until `EAX` matches the maximum address (`0x4010F5`). The next basic block, shown in Figure 6, passes the start address of the modified code (`0x40107C`) and the length value (`0x79`) as arguments to `sub_4011E6`. Without diving into this function, we see the lower 16 bits (`AX`) of its return value are moved into the `EAX` register and compared to the hard-coded value `0xFB5E`.

```
00401047 mov     eax, offset loc_40107C
0040104C mov     [ebp+var_C], eax
0040104F push    79h
00401051 push    [ebp+var_C]
00401054 call    sub_4011E6
00401059 pop     ecx
0040105A pop     ecx
0040105B movzx   eax, ax
0040105E cmp     eax, 0FB5Eh
00401063 jz      short loc_40107C
```

Figure 6 – Testing the checksum result

The result of this comparison determines if the program jumps to the modified code at `0x40107C` or falls through to a failure message shown in Figure 7:

```
00401065 push    0               ; flags
00401067 push    14h             ; len
00401069 push    offset buf      ; "Nope, that's not it."
0040106E push    [ebp+s]         ; s
00401071 call    ds:send
```

Figure 7 – FAIL

Given this information, one might correctly assume `sub_4011E6` is used to calculate a verification or checksum value for the bytes modified by the instructions in Figure 5.

At this stage we've determined a single-byte value received over the socket is used as an XOR key to modify the sample's own code between `0x40107C` and `0x4010F4`. The modified code is then verified using a hard-coded checksum value. Given the key is only a single byte, a simple brute-forcer would help us determine the expected byte value.

To determine the brute-forcer's success, we might assume the modified code executes properly and the "`Congratulations`" string is returned over the socket. Based on that assumption, a simple Python script like the one shown in Figure 8 would print the correct byte value. The script works by starting and connecting to an instance of `greek_to_me.exe`, sending a single-byte value, and determining if the "`Congratulations`" string is returned over the socket. This operation is performed in a loop that sends all possible single-byte values to the program.

```python
import sys
import os
import time
import socket

TCP_IP = '127.0.0.1'
TCP_PORT = 2222
BUFFER_SIZE = 1024

for i in range (0,256):
    os.startfile(sys.argv[1])
    time.sleep(0.1)

    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((TCP_IP, TCP_PORT))
    s.send(chr(i))
    data = s.recv(BUFFER_SIZE)
    s.close()

    if 'Congratulations' in data:
        print "Key found: %x" % i
        break
```

Figure 8 – Python socket brute-forcer

But what if we didn't want to operate under the assumption the decoded bytes execute properly and instead confirm the expected checksum value matches? Rather than reverse engineer the checksum algorithm, let's use this as an opportunity to explore an interesting malware analysis technique: emulation.

To begin, let's extract the opcode bytes present in the checksum function `sub_4011E6`. Our only concern is the return value stored in AX after the instruction at `0x401265` is executed, as shown in Figure 9. Thus, there's no need to extract the function epilogue bytes.

```
0040124E 8B C1              mov     eax, ecx
00401250 C1 E1 08           shl     ecx, 8
00401253 25 00 FF 00 00     and     eax, 0FF00h
00401258 03 C1              add     eax, ecx
0040125A 66 8B 4D FC        mov     cx, word ptr [ebp+var_4]
0040125E 66 C1 E9 08        shr     cx, 8
00401262 66 03 D1           add     dx, cx
00401265 66 0B C2           or      ax, dx
00401268 8B E5              mov     esp, ebp
0040126A 5D                 pop     ebp
0040126B C3                 retn
0040126B                    sub_4011E6 endp
```

Figure 9 – End of checksum function

We'll also extract the 0x79 encoded bytes beginning at 0x40107C. Both sets of extracted bytes are shown in the initial Python snippet for our emulation brute-forcer solution, as seen in Figure 10:

```
import binascii
import struct
from unicorn import *
from unicorn.x86_const import *
from capstone import *


CHECKSUM_CODE = binascii.unhexlify(
    '55 8B EC 51 8B 55 0C B9 FF 00 00 00 89 4D FC 85 D2 74 51 53 8B 5D 08 56 57 '
    '6A 14 58 66 8B 7D FC 3B D0 8B F2 0F 47 F0 2B D6 0F B6 03 66 03 F8 66 89 7D '
    'FC 03 4D FC 43 83 EE 01 75 ED 0F B6 45 FC 66 C1 EF 08 66 03 C7 0F B7 C0 89 '
    '45 FC 0F B6 C1 66 C1 E9 08 66 03 C1 0F B7 C8 6A 14 58 85 D2 75 BB 5F 5E 5B '
    '0F B6 55 FC 8B C1 C1 E1 08 25 00 FF 00 00 03 C1 66 8B 4D FC 66 C1 E9 08 66 '
    '03 D1 66 0B C2'.replace(' ', ''))

ENCODED_BYTES = binascii.unhexlify(
    '33 E1 C4 99 11 06 81 16 F0 32 9F C4 91 17 06 81 14 F0 06 81 15 F1 C4 91 1A '
    '06 81 1B E2 06 81 18 F2 06 81 19 F1 06 81 1E F0 C4 99 1F C4 91 1C 06 81 1D '
    'E6 06 81 62 EF 06 81 63 F2 06 81 60 E3 C4 99 61 06 81 66 BC 06 81 67 E6 06 '
    '81 64 E8 06 81 65 9D 06 81 6A F2 C4 99 6B 06 81 68 A9 06 81 69 EF 06 81 6E '
    'EE 06 81 6F AE 06 81 6C E3 06 81 6D EF 06 81 72 E9 06 81 73 7C'.replace(' ',
''))
```

Figure 10 – Extracted checksum function bytes and encoded bytes

The code in Figure 11 defines a function that performs the sample's decoding routine given a byte value between 0x00 and 0xFF:

```
def decode_bytes(i):
    decoded_bytes = ""
    for byte in ENCODED_BYTES:
        decoded_bytes += chr(((ord(byte) ^ i) + 0x22) & 0xFF)

    return decoded_bytes
```

Figure 11 – Python implementation of decoding loop

Next, we'll define a function that utilizes the Unicorn[1] framework to emulate the checksum function given a set of decoded bytes:

[1] http://www.unicorn-engine.org/docs/

```python
def emulate_checksum(decoded_bytes):
    # establish memory addresses for checksum code, stack, and decoded bytes
    address = 0x400000
    stack_addr = 0x410000
    dec_bytes_addr = 0x420000

    # write checksum code and decoded bytes into memory
    mu = Uc(UC_ARCH_X86, UC_MODE_32)
    mu.mem_map(address, 2 * 1024 * 1024)
    mu.mem_write(address, CHECKSUM_CODE)
    mu.mem_write(dec_bytes_addr, decoded_bytes)
```
Figure 12 – Unicorn emulation environment setup

The code in Figure 12 initializes an x86 emulator in 32-bit mode and creates a 2MiB memory range used to store the checksum function code, a stack for use within the function, and the decoded bytes. The checksum code and decoded bytes are written to arbitrary locations within the memory range.

The checksum function receives two arguments that are pushed onto the stack prior to the function call at `0x401054`: the address of the decoded bytes (`0x40107C`) and the number of bytes (`0x79`). Figure 13 illustrates the state of the program stack after the checksum function is called:

ESP	Return address
ESP+4	Address of decoded bytes (0x40107C)
ESP+8	Size of decoded bytes (0x79)

Figure 13 – Stack layout after checksum function call

For the checksum function to emulate properly, we setup the stack to match the layout in Figure 13 and populate the ESP register. After emulation, we can return the calculated checksum from the `emulate_checksum` function as shown in Figure 14.

```python
    # place the address of decoded bytes and size on the stack
    mu.reg_write(UC_X86_REG_ESP, stack_addr)
    mu.mem_write(stack_addr + 4, struct.pack('<I', dec_bytes_addr))
    mu.mem_write(stack_addr + 8, struct.pack('<I', 0x79))

    # emulate and read result in AX
    mu.emu_start(address, address + len(CHECKSUM_CODE))
    checksum = mu.reg_read(UC_X86_REG_AX)

    return checksum
```
Figure 14 – Stack setup and emulation

Now the easy part! We iterate through all the possible single-byte values as XOR keys, decode the bytes, emulate the checksum, and determine which byte results in the expected checksum value. This is shown in the script fragment in Figure 15:

```
for i in range(0, 256):
    decoded_bytes = decode_bytes(i)
    checksum = emulate_checksum(decoded_bytes)
    if checksum == 0xFB5E:
        print 'Checksum matched with byte %X' % i
```

Figure 15 – Attempting each single-byte value

Running the script prints the single-byte value the sample expects to receive over the socket: `0xA2`. However, we still don't understand the nature of what we assume are decoded instructions at `0x40107C`. Let's attempt to disassemble the instructions using the Capstone[2] disassembler and complete the `for` loop we initiated in Figure 15. The result is shown in Figure 16.

```
        print 'Decoded bytes disassembly:'
        md = Cs(CS_ARCH_X86, CS_MODE_32)
        for j in md.disasm(decoded_bytes, 0x40107C):
            print "0x%x:\t%s\t%s" % (j.address, j.mnemonic, j.op_str)
        break
```

Figure 16 – Disassembling the decoded bytes

Running our script provides some interesting disassembly, as shown in Figure 17:

[2] http://www.capstone-engine.org/lang_python.html

```
Success with byte A2
Decoded bytes disassembly:
0x40107c:    mov bl, 0x65
0x40107e:    mov byte ptr [ebp - 0x2b], bl
0x401081:    mov byte ptr [ebp - 0x2a], 0x74
0x401085:    mov dl, 0x5f
0x401087:    mov byte ptr [ebp - 0x29], dl
0x40108a:    mov byte ptr [ebp - 0x28], 0x74
0x40108e:    mov byte ptr [ebp - 0x27], 0x75
0x401092:    mov byte ptr [ebp - 0x26], dl
0x401095:    mov byte ptr [ebp - 0x25], 0x62
0x401099:    mov byte ptr [ebp - 0x24], 0x72
0x40109d:    mov byte ptr [ebp - 0x23], 0x75
0x4010a1:    mov byte ptr [ebp - 0x22], 0x74
0x4010a5:    mov byte ptr [ebp - 0x21], bl
0x4010a8:    mov byte ptr [ebp - 0x20], dl
0x4010ab:    mov byte ptr [ebp - 0x1f], 0x66
0x4010af:    mov byte ptr [ebp - 0x1e], 0x6f
0x4010b3:    mov byte ptr [ebp - 0x1d], 0x72
0x4010b7:    mov byte ptr [ebp - 0x1c], 0x63
0x4010bb:    mov byte ptr [ebp - 0x1b], bl
0x4010be:    mov byte ptr [ebp - 0x1a], 0x40
0x4010c2:    mov byte ptr [ebp - 0x19], 0x66
0x4010c6:    mov byte ptr [ebp - 0x18], 0x6c
0x4010ca:    mov byte ptr [ebp - 0x17], 0x61
0x4010ce:    mov byte ptr [ebp - 0x16], 0x72
0x4010d2:    mov byte ptr [ebp - 0x15], bl
0x4010d5:    mov byte ptr [ebp - 0x14], 0x2d
0x4010d9:    mov byte ptr [ebp - 0x13], 0x6f
0x4010dd:    mov byte ptr [ebp - 0x12], 0x6e
0x4010e1:    mov byte ptr [ebp - 0x11], 0x2e
0x4010e5:    mov byte ptr [ebp - 0x10], 0x63
0x4010e9:    mov byte ptr [ebp - 0xf], 0x6f
0x4010ed:    mov byte ptr [ebp - 0xe], 0x6d
0x4010f1:    mov byte ptr [ebp - 0xd], 0
```

Figure 17 – Script results

For those new to reverse engineering, two aspects of the Figure 17 disassembly should stand out. First, a stack string[3] is being populated. Second, the constant hex values being moved onto the stack fall within the range of printable characters (`0x20-0x7E`). Extracting these printable characters in the order they are moved onto the stack or by viewing the stack in a debugger after providing the correct byte yields the challenge solution:

et_tu_brute_force@flare-on.com

[3] https://www.fireeye.com/blog/threat-research/2016/06/automatically-extracting-obfuscated-strings.html

Appendix A: Python Emulation Script

```
import binascii
import struct
from unicorn import *
from unicorn.x86_const import *
from capstone import *

CHECKSUM_CODE = binascii.unhexlify(
    '55 8B EC 51 8B 55 0C B9 FF 00 00 00 89 4D FC 85 D2 74 51 53 8B 5D 08 56 57 '
    '6A 14 58 66 8B 7D FC 3B D0 8B F2 0F 47 F0 2B D6 0F B6 03 66 03 F8 66 89 7D '
    'FC 03 4D FC 43 83 EE 01 75 ED 0F B6 45 FC 66 C1 EF 08 66 03 C7 0F B7 C0 89 '
    '45 FC 0F B6 C1 66 C1 E9 08 66 03 C1 0F B7 C8 6A 14 58 85 D2 75 BB 5F 5E 5B '
    '0F B6 55 FC 8B C1 C1 E1 08 25 00 FF 00 00 03 C1 66 8B 4D FC 66 C1 E9 08 66 '
    '03 D1 66 0B C2'.replace(' ', ''))
ENCODED_BYTES = binascii.unhexlify(
    '33 E1 C4 99 11 06 81 16 F0 32 9F C4 91 17 06 81 14 F0 06 81 15 F1 C4 91 1A '
    '06 81 1B E2 06 81 18 F2 06 81 19 F1 06 81 1E F0 C4 99 1F C4 91 1C 06 81 1D '
    'E6 06 81 62 EF 06 81 63 F2 06 81 60 E3 C4 99 61 06 81 66 BC 06 81 67 E6 06 '
    '81 64 E8 06 81 65 9D 06 81 6A F2 C4 99 6B 06 81 68 A9 06 81 69 EF 06 81 6E '
    'EE 06 81 6F AE 06 81 6C E3 06 81 6D EF 06 81 72 E9 06 81 73 7C'.replace(' ',
''))


def decode_bytes(i):
    decoded_bytes = ""
    for byte in ENCODED_BYTES:
        decoded_bytes += chr(((ord(byte) ^ i) + 0x22) & 0xFF)

    return decoded_bytes


def emulate_checksum(decoded_bytes):
    # establish memory addresses for checksum code, stack, and decoded bytes
    address = 0x400000
    stack_addr = 0x410000
    dec_bytes_addr = 0x420000

    # write checksum code and decoded bytes into memory
    mu = Uc(UC_ARCH_X86, UC_MODE_32)
    mu.mem_map(address, 2 * 1024 * 1024)
    mu.mem_write(address, CHECKSUM_CODE)
    mu.mem_write(dec_bytes_addr, decoded_bytes)

    # place the address of decoded bytes and size on the stack
    mu.reg_write(UC_X86_REG_ESP, stack_addr)
    mu.mem_write(stack_addr + 4, struct.pack('<I', dec_bytes_addr))
    mu.mem_write(stack_addr + 8, struct.pack('<I', 0x79))
```

```
    # emulate and read result in AX
    mu.emu_start(address, address + len(CHECKSUM_CODE))
    checksum = mu.reg_read(UC_X86_REG_AX)

    return checksum


for i in range(0, 256):
    decoded_bytes = decode_bytes(i)
    checksum = emulate_checksum(decoded_bytes)
    if checksum == 0xFB5E:
        print 'Checksum matched with byte %X' % i
        print 'Decoded bytes disassembly:'
        md = Cs(CS_ARCH_X86, CS_MODE_32)
        for j in md.disasm(decoded_bytes, 0x40107C):
            print "0x%x:\t%s\t%s" % (j.address, j.mnemonic, j.op_str)
        break
```