Lab 04-1.malware

1. **Set a breakpoint at 0x00401092, what is this sample calling?**

Address	Value	Comment
0023FC30	00A61094	CALL to GetProcAddress from Lab_04-1.00A61092
0023FC34	76CF0000	hModule = 76CF0000 (kernel32)
0023FC38	0023FC3C	ProcNameOrOrdinal = "VirtualAlloc"
0023FC3C	74726956	
0023FC40	416C6175	

This CALL instruction calls the *GetProcAddress* (Get Procedure Address) function from KERNEL32.dll, in order to load the address of the *VirtualAlloc* function.

2. **What is being called at 0x004010A6? What is the callee doing?**

Address	Value	Comment
002BFC64	00A610A8	CALL to VirtualAlloc from Lab_04-1.00A610A6
002BFC68	0C000000	Address = 0C000000
002BFC6C	0000B000	Size = B000 (45056.)
002BFC70	00003000	AllocationType = MEM_COMMIT|MEM_RESERVE
002BFC74	00000040	Protect = PAGE_EXECUTE_READWRITE
002BFC78	76D3C65A	kernel32.VirtualAlloc
002BFC7C	74726956	

The *VirtualAlloc* function is called to reserve and commit 44 KB of memory starting at address 0xC000000. Since the PAGE_EXECUTE_READWRITE was passed in, this may suggest that code will be loaded into this portion of memory.

3. **What is sub_401360 doing? What about sub_401372 and sub_401388?**

sub_401360, sub_401372, and sub_401388 all make calls to *GetProcAddress*, but through different DLLs. sub_401360 uses KERNEL32.dll, sub_401372 uses ADVAPI32.dll, and sub_401388 uses USER32.dll. Examples of these function calls are shown below.

Address	Value	Comment
0012F938	00A6136B	CALL to GetProcAddress from Lab_04-1.00A61368
0012F93C	76CF0000	hModule = 76CF0000 (kernel32)
0012F940	0012F944	ProcNameOrOrdinal = "GetModuleFileNameA"

sub_401360 using KERNEL32.dll's *GetProcAddress* function to call *GetModuleFileNameA*.

Address	Value	Comment
0025FA54	00A61381	CALL to GetProcAddress from Lab_04-1.00A6137E
0025FA58	76AC0000	hModule = 76AC0000 (ADVAPI32)
0025FA5C	0025FA60	ProcNameOrOrdinal = "RegCreateKeyA"

sub_401372 using ADVAPI.dll's *GetProcAddress* function to call *RegCreateKeyA*.

Address	Value	Comment
002FFAC8	00A61397	CALL to GetProcAddress from Lab_04-1.00A61394
002FFACC	76580000	hModule = 76580000 (USER32)
002FFAD0	002FFAD4	ProcNameOrOrdinal = "MessageBoxA"

sub_401388 using USER32.dll's *GetProcAddress* function to call *MessageBoxA*.

4. **What Windows API functions did the sample import?**

The sample imported *RegCloseKey, MessageBoxA, LoadLibraryA, VirtualAloc, GetModuleFileNameA, ExitProcess, RegCreateKeyA, CopyFileA, RegSetKeyValueA, RegCloseKey, GetWindowsDirectoryA*.

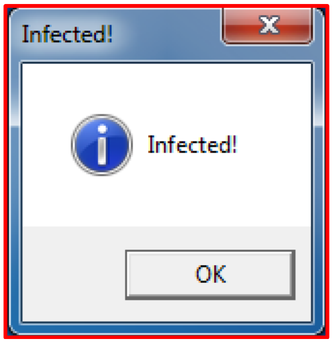
5. **How did you find the imported functions?**

You can see which functions were imported by setting a breakpoint at the function *kernel32.GetProcAddress*. The second argument to this function is *ProcNameOrOrdinal*, which contains an ASCII string of the function being called, like *MessageBoxA*, as shown below.

Address	Value	Comment
0026F774	01011397	CALL to GetProcAddress from Lab_04-1.01011394
0026F778	751B0000	hModule = 751B0000 (USER32)
0026F77C	0026F780	ProcNameOrOrdinal = "MessageBoxA"

6. **What does this sample do?**

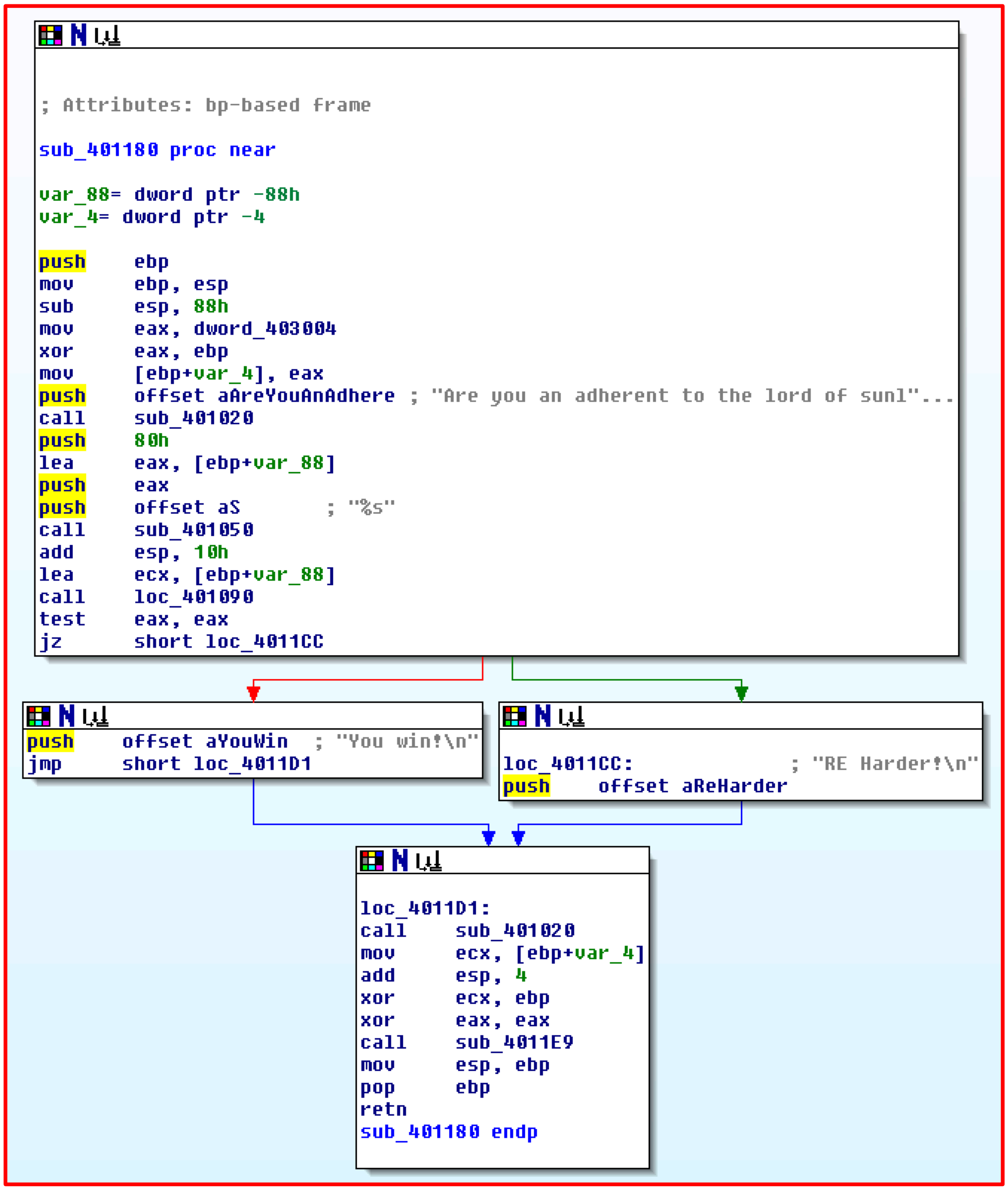
This sample copies itself to *C:\Windows\virus.exe*, places itself in the user's registry value for auto-run at login, and displays a message to the user that they are infected, as seen below.



Lab 04-2.malware

1. **What is the address of the win/lose function?**

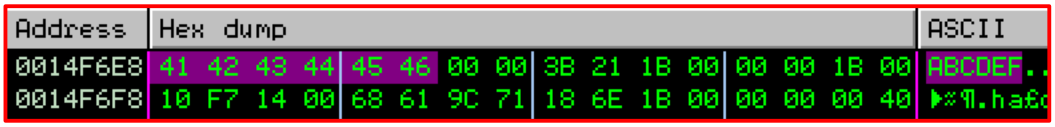
sub_401180 (0x401180) is the win/lose function.



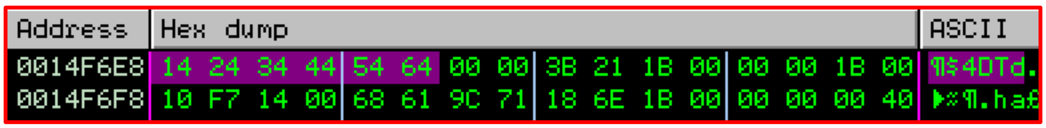
The left branch is the ‘win’ output, and the right branch is the ‘lose’ output, which ouputs “[Reverse Engineer] Harder!”

2. **What does this sample do with the user input?**

It calculates the string length, and then reverses the bytes of each character in memory (shown below)



Original input string was "ABCDEF".



The digits of the hex bytes for the input string were reversed, i.e. 0x41 → 0x14.

Once the character/hex bytes of the user-input string are reversed in memory, they get XOR'd with a hardcoded variable. This variable is a string containing the text *flag{Th1s_!s_n0t_the_acTua1_F1ag}*. Once this XOR is calculated, each byte value gets compared to the encrypted flag's corresponding byte value. At the first difference, the program will exit and inform the user they don't have the right password ("RE Harder!"). If there is no difference, the user has supplied the right password/flag, and the program outputs a success message ("You win!").

Interestingly, the code has a major bug. As long as the user string is a substring of the real flag, starting at the first byte, the program will inform the user that they have succeeded. For example, if the user inputs a single character, *f*, the program will claim they put in the 'right' password. *fl, fla, flag, flag{, flag{P* – and continuing on until you get the full-length flag- will all return successful. I assume this is due to a faulty call to *strcmp* inside an if statement (the programmer probably used a '<' or '>' comparator instead of '!=').

3. **What is the address of the encrypted flag?**

The address of the encrypted flag is 0x40303C. This is referenced by 0x40114E when the code compares the character in the AL register to the corresponding character in the encrypted flag.

```
.data:00403018 7B 67 61 31 46 5F+fake_flag       db '{ga1F_1auTca_eht_t0n_s!_s1hT}galf',0
.data:00403018 31 61 75 54 63 61+                                ; DATA XREF: .text:004010EB↑o
.data:0040303A 00 00                             align 4
.data:0040303C 1D A1 77 47 F1 5A+encrypted_flag  db 1Dh,'íwG±Z',16h,'wfc5ö',18h,'p[üj#+|ê',0,0,0
.data:0040303C 16 77 66 63 35 94+                                ; DATA XREF: .text:0040114E↑r
```

4. What is the flag?

The flag is *flag{Pra1se_th3_Sun!}*. The code to reverse the encryption steps and decipher the flag is shown below.

```python
#!/usr/bin/python

#str1 must be larger string
def xor(str1, str2):
    final_str = ""
    counter = 0
    for letter in str1:
        new_char = chr(ord(letter) ^ ord(str2[counter % len(str2)]))
        final_str += new_char
        counter += 1

    return final_str

reversed_fake_flag = "{ga1F_1auTca_eht_t0n_s!_s1hT}galf"
encrypted_flag = "1DA17747F15A16776663359418E35B816A23D67C88".decode("hex")
flag = ""

reversed_flag = xor(reversed_fake_flag, encrypted_flag) # 0x401106 XOR [EDX-1], AL
print "Reversed flag, pre bit shift: " + reversed_flag

for letter in reversed_flag:
    letter = ord(letter)
    shift_left = (letter & 0xF) << 4 # SHL AL, 4 (0x4010B5)
    shift_right = (letter >> 4)      # SAR CL, 4 (0x4010B8)
    reversed_character = shift_left + shift_right # OR CL, AL (0x4010BB)
    flag += chr(reversed_character)

print "flag: " + flag
```

```
Reversed flag, pre bit shift: fΔv∑'7VıGÜ3ı5WÊ◊nÄ(4¿2B
YÚ
flag: flag{Pra1se_th3_Sun!}ÇC
#$† ï/
```

Output of python script

```
C:\Users\John Smith\Desktop\Lab_04-2.exe
Are you an adherent to the lord of sunlight?
flag{Pra1se_th3_Sun!}
You win!
```

Example of successful user interaction