IMT4116 - Mandatory Assignment 2

Ole André Hauge

April 26, 2022

1 Advanced Static Analysis

1.1 Find Function

- How many times is the function fopen called?

  Answer: We locate the fopen function in IDA by looking at the 'Names' window. Highlighting the unique name IDA has given the function we can press 'x' to bring up the 'Xref' window for the function. Here we can see that the 'fopen' function is called 7 times in total, as shown in Figure 1.

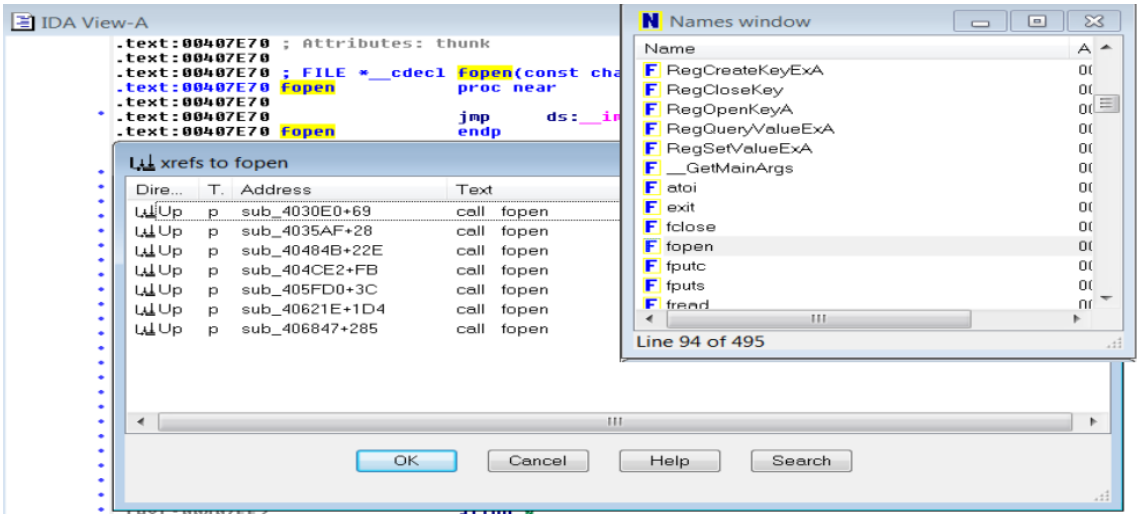

Figure 1: IDA Result: xref to fopen

- Go to the first (lowest address) fopen in the list? State the address. (The next 4 questions are related to this specific instance of fopen)

  Answer: We find and go to the lowest address of 'fopen' by double clicking on the upmost 'fopen' in the list seen in Figure 1. The lowest (first) address for 'fopen' is 00403149 as seen in Figure 2.

Figure 2: IDA Result: address of first fopen call

- What is a prologue in general and specific for this instance of call fopen?

  Answer: A prologue in general is the preparation of the stack and registers for transfer of control to a function. Before a function is called the values that are going to be used by the function are pushed on

the stack so that they are ready to be used. Often these values are results from previous operations. The prologue then pushes the current base pointer (ebp) on to the stack. This way it can be stored for later and used to return to the correct address in the code that the function was called from. Next the value of the base pointer is assigned to the stack pointer (esp) to have the base pointer pointing at the top of the stack, which is where the next execution of the code takes place. Then, the base pointer might be moved down (in x86) to make room for the local variables that are going to be used by the function.

For the instance of the first fopen call at address 00403149 we can see that the offset 'aw' is pushed on the stack before address, resulting from 'ebp' + a local variable, is loaded into 'eax'. 'eax' is then pushed on the stack before the function call to 'fopen' (see Figure 3).

```
.text:00403131    lea     eax, [ebp+var_AD4]
.text:00403137    push    eax             ; char *
.text:00403138    call    sprintf
.text:0040313D    push    offset aAw      ; "aw"
.text:00403142    lea     eax, [ebp+var_AD4]
.text:00403148    push    eax             ; char *
.text:00403149    call    fopen
.text:0040314E    add     esp, 18h
```

Figure 3: IDA Result: prologue of first fopen call

If we know that the syntax for fopen is: FILE *fopen(const char *filename, const char *mode), we can deduce (remembering that the last value pushed on the stack is the first read) that the filename is stored at the address location [ebp+var_AD4], and that the access mode that is requested is 'aw', where 'a' appends text to the end of an existing file and 'w' opens an empty i.e. new file for writing. In short, a file is created if it does not already exist and appended to if it does.

Looking at the assembly code before 'fopen', we can see that the 'GetSystemDirectoryA' function and 'sprintf' are called. The result of the first function returns the length of the string (the system directory path) that is copied to a buffer. The second function formats and stores characters and values to a buffer. Thus, it is reasonable to believe that the value at ebp+var_AD4 equates to full path of the 'keylog.txt' file that the malware wants to create. The prologue for this instance of 'fopen' pushes these two variables on the stack.

- What is an epilogue in general and specific for this instance of call fopen?

  Answer: In general, the epilogue restores the stack and registers to the state before the function call so that we can return to the correct place in the previous code. It starts by cleaning up the stack by removing the added values used by the function, usually by adding some value to 'esp' i.e. 'esp+4'. Effectively freeing up the space. Then it sets the value of 'esp' to the value of 'ebp', before it pops 'ebp' to return to the correct place on the stack i.e. the value it had before the prologue. It then gets rid of the current stack frame and puts the stack frame of the caller back into effect. Lastly, it returns to the calling function by poping the value that is on top of the stack.

  In our instance we can see in Figure 3 that the clean up is done after the function call by adding 18 hex *esp* i.e. moving the stack pointer.

- What calling convention is used here? Explain how you found your answer.

  Answer: Looking at the epilogue in Figure 3 we see that the clean up is done outside of the function i.e. the caller is the one who cleans up the stack. The calling convention that does that is the C declaration called CDECL. We can double check this as is stated when we enter on 'fopen' function in the function list.

- Explain the purpose of the 4 next assembly instructions, after "call fopen"?

  Answer: Looking at Figure 4:

  – The *add* instruction is used to clean up the stack after the function call.

- The *mov* instruction moves the value stored in *eax* to *ebx*. *eax* is often used to store the return value from functions. In the case of *fopen*, the return value is the pointer to the open file (given no errors).
- The *or* instruction executes a bitwise or operation on *ebx* with itself. Or'ing a value with itself just returns the value. This is way to check if the value is 0.
- The *jz* instruction checks the zero flag (ZF) and if it is set (1), jumps to the location at address 004031CD. In this case that would only happen if the result of *fopen* was zero, due to the previous or operation. What this means is that we continue to execute the next instruction below address 00403155 (see Figure 4) unless we have an error with opening the file.

Figure 4: IDA Result: 4 next assembly instructions

1.2 Opcode Knowledge

Explain the single instructions found at the following addresses. You do not have to find the actual value of arguments used, e.g. if eax is involved, it is enough to state that "the value of eax. . . ".

- 403109: Moves the value of eax into the memory location pointed to by [ebp+var_AD8].
- 403142: Loads the effective address from [ebp+var_4AD] into eax, i.e. the value resulting from the addition is loaded as an address into eax.
- 403231: The value of eax is or'ed with FFFFFFFFh. Possibly to mask the value.
- 403270: Pushes the dword stored in the data segment (ds) on the stack. This is a global variable.
- 403258: Adds the hex-value C to the esp register. Seems to be a clean up after a function call.
- 4032FD: The test instructions runs a bitwise AND operation on the 16-bit value stored in the di register and the hex-value 8000 (might be to test for singed value/non-negative). If the values match i.e. returns 1 the ZF is set to 0, while it is set to 1 if there is no match. Test also modifies the 0F, CF, SF, and PF flags, which can be used for tests, however ZF is most relevant to the next line in the code as it uses the jz instruction.
- 403342: Compares the value stored in eax with the hex-value 42.
- 403345: The jle instruction is a conditional jump following a test. If the result of the test of the destination operand is less than or equal to the source operand it will jump to loc_40335C address.

1.3 Key Logging

- At what addresses are keys examined?

  Answer: Looking at the functions in the 'Name window' we can identify 4 that concerns keys: GetKeyState, GetAsyncKeyState, MapVirtualKey, and keybd_event. However, looking at the code it is evident that the examination of keys are done by GetKeyState, GetAsyncKeyState:

  - GetKeyState: 4032DB, 403306, 403332
  - GetAsyncKeyState: 4032F3

- What keys are examined?

  Answer: GetAsyncKeyState checks if a key is down or up, and if it has been pressed since the last iteration. The iteration goes through 92 keys (5Ch). GetKeyState checks for caps lock and shift.

- Goto loc: 403579. The conditional jump at 403580 defines two loops.
  - What is the purpose of ebp+var_4?

    Answer: ebp+var_4 is a counter for the loop. If the value is less than 5Ch (92) the program jumps to loc_4032D9.

  - What is the purpose of the short loop?

    Answer: The short loop registers the key that are pressed by polling the key defined by ebp+var_4, as well as checking the for caps lock, shift, and windows. It does also have code to write to the 'keylog.txt' file (becomes evident by entering the function calls).

  - What is the purpose of the longer loop?

    Answer: The longer loop checks if the window is changed and writes the change to the 'keylog.txt' file. This happens when the short loop is done. It sleeps a short period before polling for keys again.

  - How often are keys polled?

    Answer: We can see that the keys are polled every 8 milliseconds. This is given in 40320D where the PUSH instruction is used put 8 on the stack before the 'sleep' function is called.

1.4 Mutex

We suspect this sample to use mutex (also known as mutant)

- Why do we suspect this?

  Answer: The functions window shows that there is a CreateMutexA function loaded. By using xref to see where it i used we can identify that it is used once in the malicious code in address 4014C6.

- What is the most likely purpose of using mutex/mutant?

  Answer: The mutex is likely used to avoid infecting the system twice. This is a common method used by malware authors.

- What is the mutex/mutant for this sample?

  Answer: Looking in address 4014BD, we can see that the string 'krnel' is pushed on the stack as an argument for the mutex function. This is likely the name it uses.

- Identify the address where the mutex is created.

  Answer: Assuming created refers to where the call function for CreateMutexA is called: 4014C6. Otherwise the name can be found hardcoded in memory in 412074.

- How is the mutex used?

  Answer: The malicious code tries to create the mutex. Checks error code. If the error return value is '0B7h' (ERROR_ALREADY_EXISTS) it exits the process. Otherwise it continues to import libraries before installing the malware. The mutex checks whether the computer is already infected, and if it is it exits the process to not infect the computer twice.

2 Advanced Dynamic Analysis

In this section we use OllyDbg to answer the following questions.

- What is the input and output of the function call to 402B81, done twice early in the execution?

  Answer: The function takes two inputs arguments. The first is a memory address pointing to scrambled ASCII text. The second is an offset used in the operation to de-scramble the ASCII text at the memory location. The output from the functions are registry paths resulting from the de-scrambled ASCII text in the input.

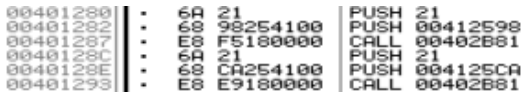

Figure 5: OllyDbg result: function call to 402B81

  Input function 1

  - arg1: address 412598 (ASCII data is stored here but looks scrambled)
  - arg2: 21 (offset value used to de-scrable ASCII values)

  Output function 1

  - SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce

  Input function 2

  - arg1: address 4125CA (ASCII data is stored here but looks scrambled)
  - arg2: 21 (offset value used to de-scrable ASCII values)

  Output function 2

  - SOFTWARE\Microsoft\Windows\CurrentVersion\Run

- Show all filenames that are used in CopyFileA at 401452. Confirm the creation on your filesystem also.

  Answer: The function is inside a loop. During the loop an existing filename and new filename is used by the CopyFileA function. The filenames that occur during the loop is lisetd below. After executing the call we validate that the file is created in C:\Windows\system32\kazaabackupfiles\by checking the folder as the loop progresses as seen in Figure 6.

  Existing filename: C:\Users\rev_eng\Desktop\assignemnet2.exe.exe

  New filename: zoneallarm_pro_crack.exe, AVP_Crack.exe, Porn.exe, Norton_Anti-Virus_2002_Crack.exe, GeneralsNo-CD_Crack.exe, Renegade_No-CD_Crack.exe, Red_Faction_2_No-CD_Crack.exe, Postal_2_Crack.exe, FlashFXP_Crack.exe, DreamweaverMX_Crack.exe, PlanetSide.exe, Winamp_installer.exe, Sitebot.exe, EDU_Hack.exe,

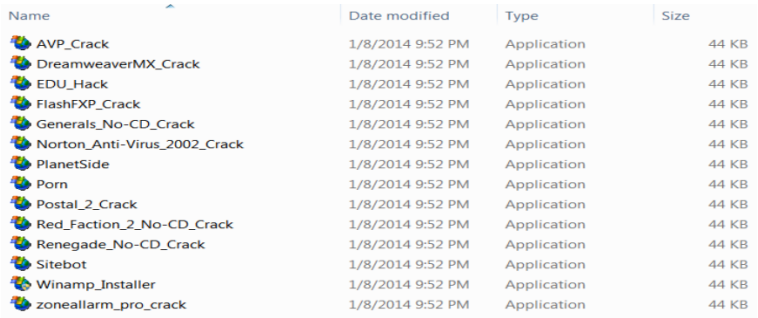

Figure 6: OllyDbg result: Files created by CopyFileA

- Show what happens to the filename in 40134D if it already exists.

  Answer: Normally, if it does not exist, it jumps to the PUSH instruction in 40133D and creates 'wuamqr.exe'. If it exists, it does not jump and enter the code section from the CALL function in 4012F8 to the PUSH instruction in 40133D. In this section the filename is scrambled to a random sequence of letters. We simply changed the value of EAX used in the jump test after the function call in 40134D to simulate that the file already existed. This resulted in the filename: 'obzqjnk.exe' as seen in Figure 7. We verified that the file was created in the system32 directory.

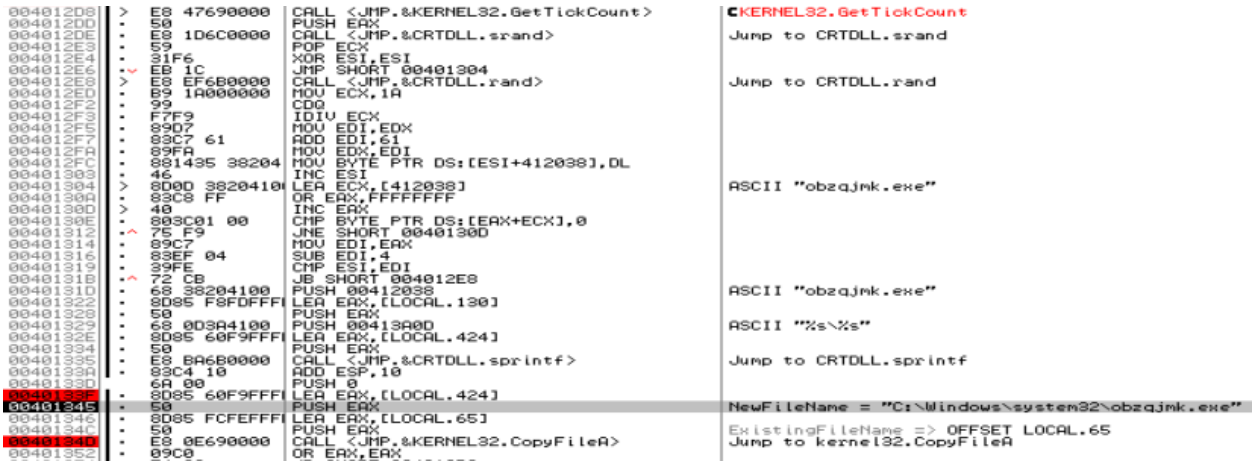

Figure 7: OllyDbg result: Filename modification in 40134D

- Show what directory is created in 4013A5 by CreateDirectoryA

  Answer: The created directory is: C:\Windows\system32\kazaabackupfiles\as seen in Figure 8. We verified it by checking the system32 directory before and after running the function.

Figure 8: OllyDbg result: Directory created in 4013A5

- How do we get the debugger to move past ExitProcess in 40147D and get to 401482?

  Answer: To get past the 'ExitProcess' funciton we can simply replace it by 'NOP' instructions as seen in Figure 9. This way the code that is replaced is not executed and the 'NOP' instructions does not affect the code in a negative way (in terms of debugging).

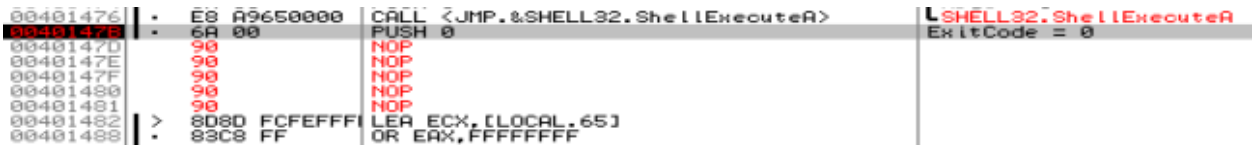

Figure 9: OllyDbg result: Nop-sled in 40147D