Episode 30 (Malware Analysis II)

Execution (EXE) file parsing

Kitsune, why do you have to analyze a regular executable file in order to do malware analysis?

Tanuki, there is a wide variety of malware. It is impossible to cover all of them here, and I am not going to give the names of the viruses or explain how they work.

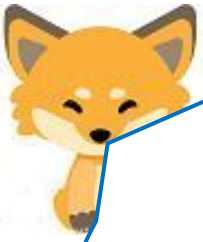
The malware I am interested in are file-infection viruses.

It is the CIH (also known as Chernobyl) virus. It rewrites part of the applications you use in your daily life, invoking the CIH virus program every time you launch the application, and then, with a nonchalant face, leaks your personal information to the outside world while collecting information behind the scenes.

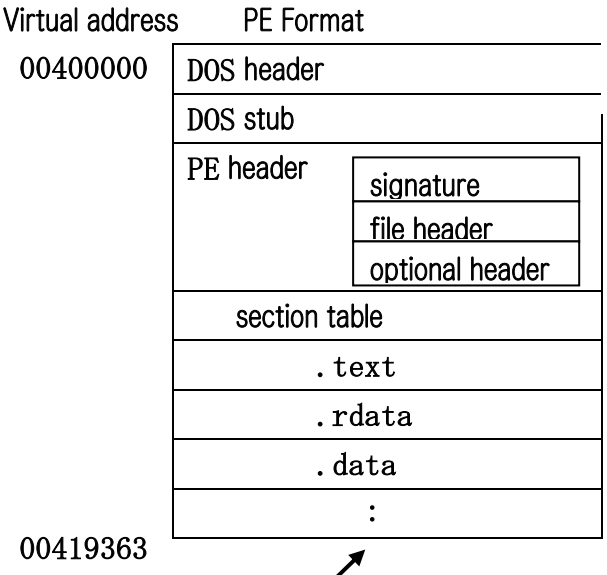
The infected application can be launched and used normally, so the user is unaware of the infection. In addition, the code that invokes the CIH virus is written to the application's free space, so the application's file size does not increase. Furthermore, the CIH virus program is invoked before the application is launched.

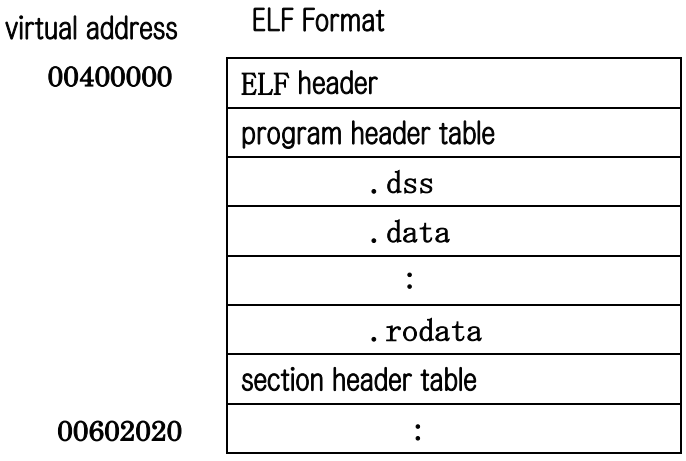
Furthermore, the code that the CIH virus writes to the application is in assembler language, making it ideal for analysis using Ghidra.

Isn't the CIH virus primarily targeting Windows?

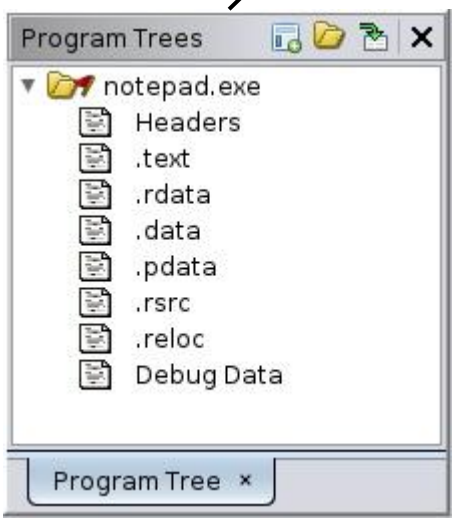


Exactly. That is why it is essential to first understand the specifications of executable files in order to find CIH viruses.
Let's take the case of notepad, which comes standard with Windows.
Windows executables such as notepad.exe are in PE (Portable Executable) format.
However, the emfd.exe used this time is an executable file on Linux, so it is in ELF (Executable and Linkable Format) format.
There are some differences between the two, but the basic parts are similar.
Since emfd.exe is the simplest program with only input/output, I will explain it here.
The differences between PE and ELF structures are illustrated below.



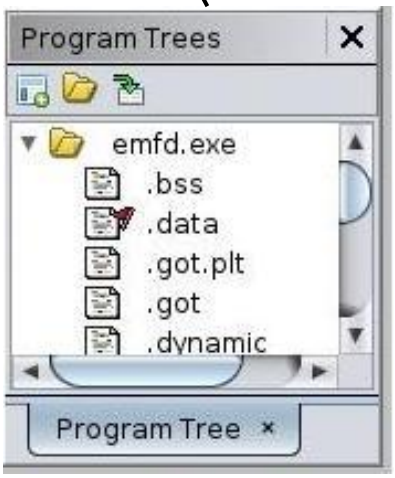


notepad. exe (windows executable file)



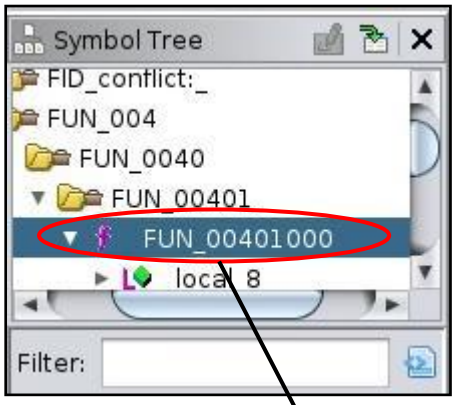
I can see the difference between the two format formats when I compare them on Ghidra!

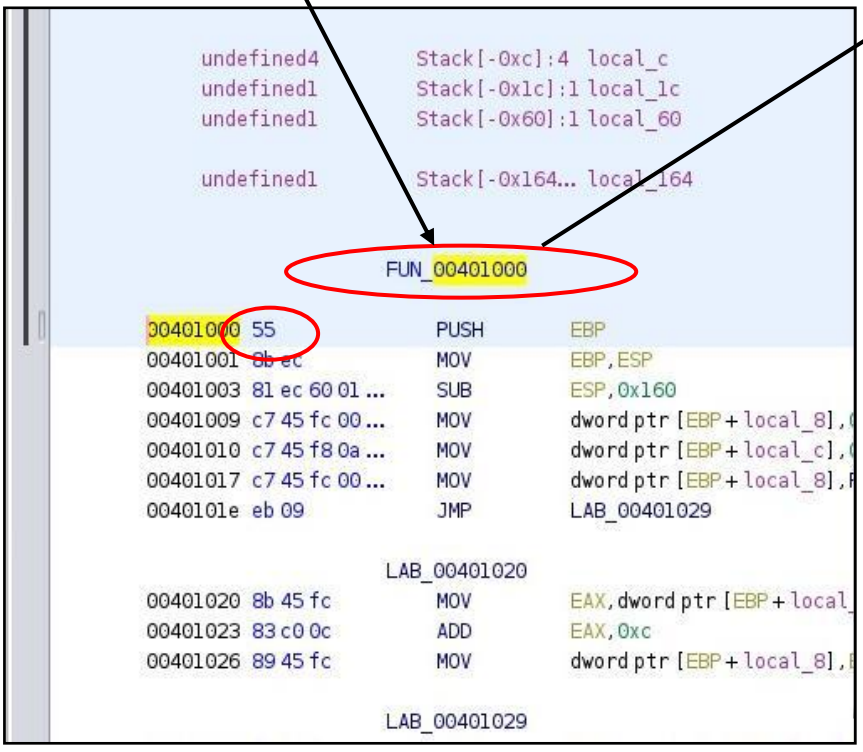
emfd. exe (Linux executable file)

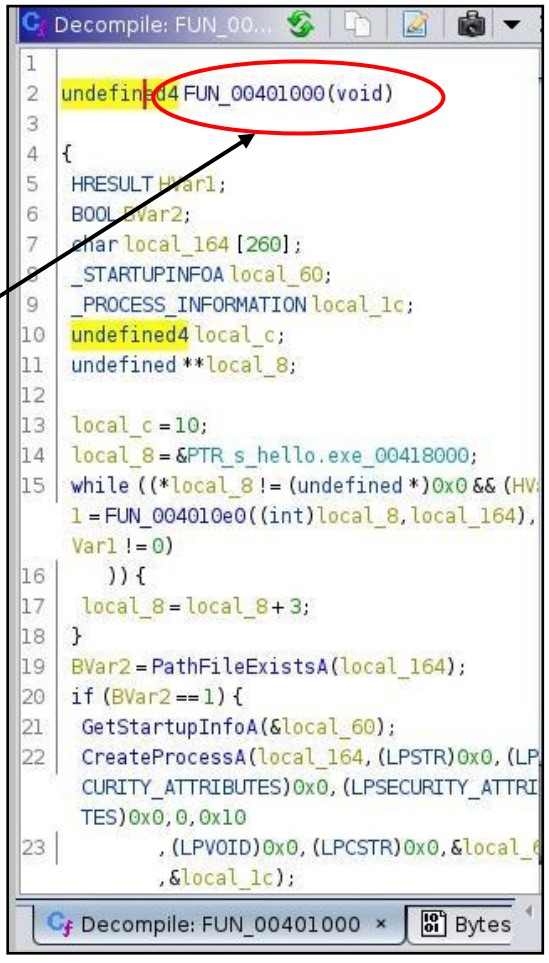


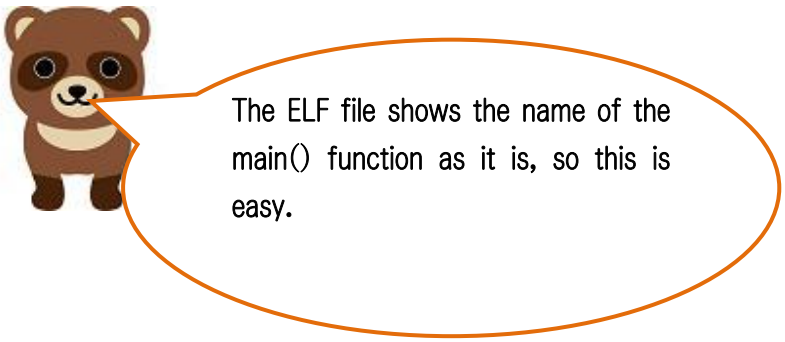
Next, a note on parsing C executables.
The C language is first executed from the main() function. This is the same for PE and ELF files. However, the name of the main() function in the ELF file is displayed as "main", but the main() function in the PE (Portable Executable) file, which is a Windows executable file, is not named "main".
This causes the problem of having to find a function that corresponds to the main function. This is because the analysis will not proceed unless the main() function is found. We can't just blindly analyze. The following figures show examples of main() functions in PE and ELF files.

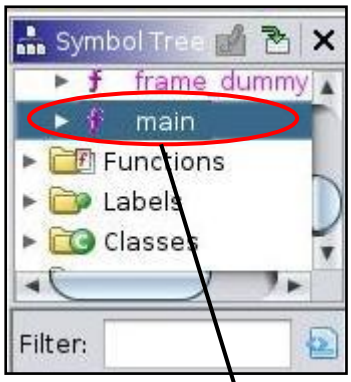
So the "FUN_00401000" function is the main() function in the PE file? That's a tough one to find.
Furthermore, I guess it is not always named "FUN_00401000"!

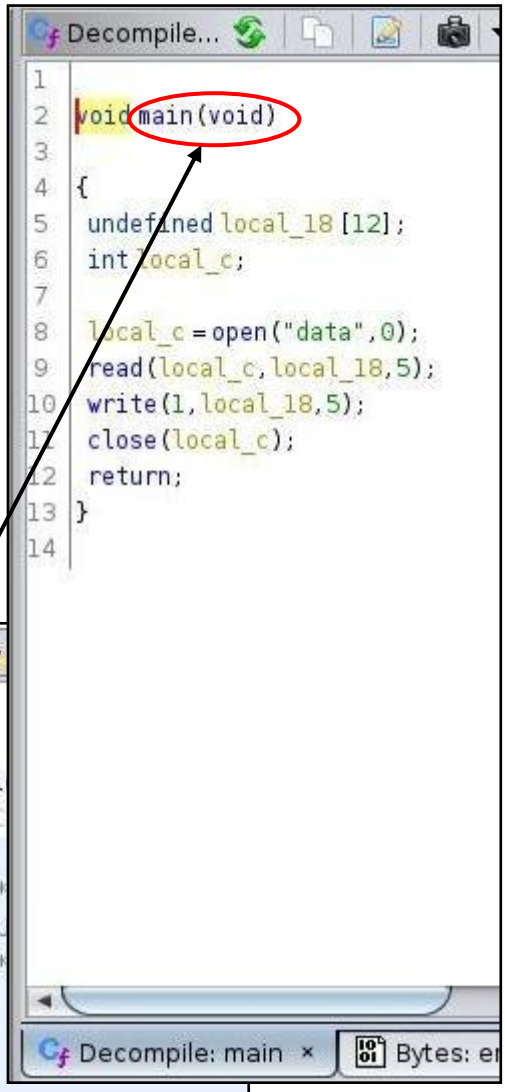


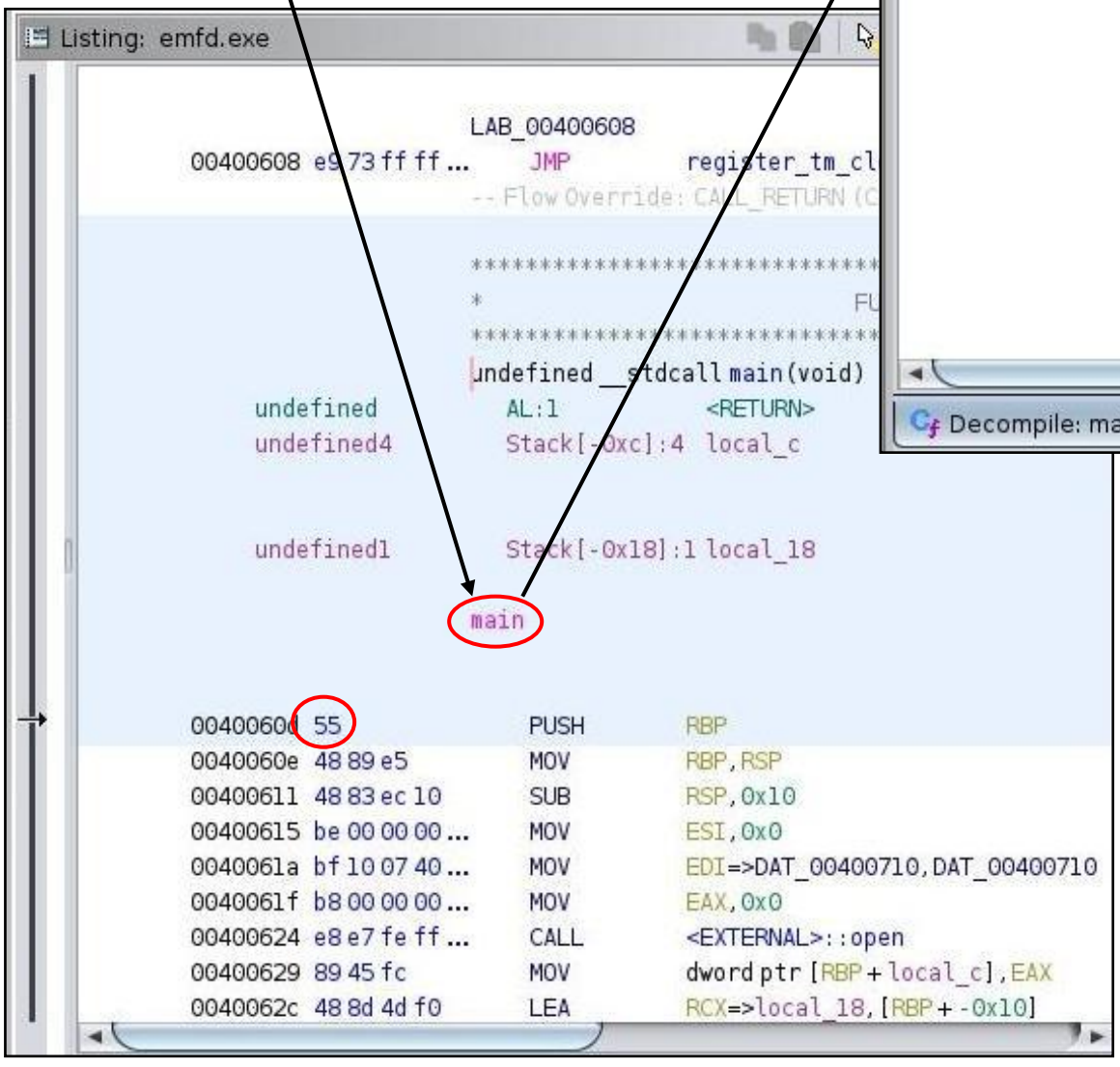


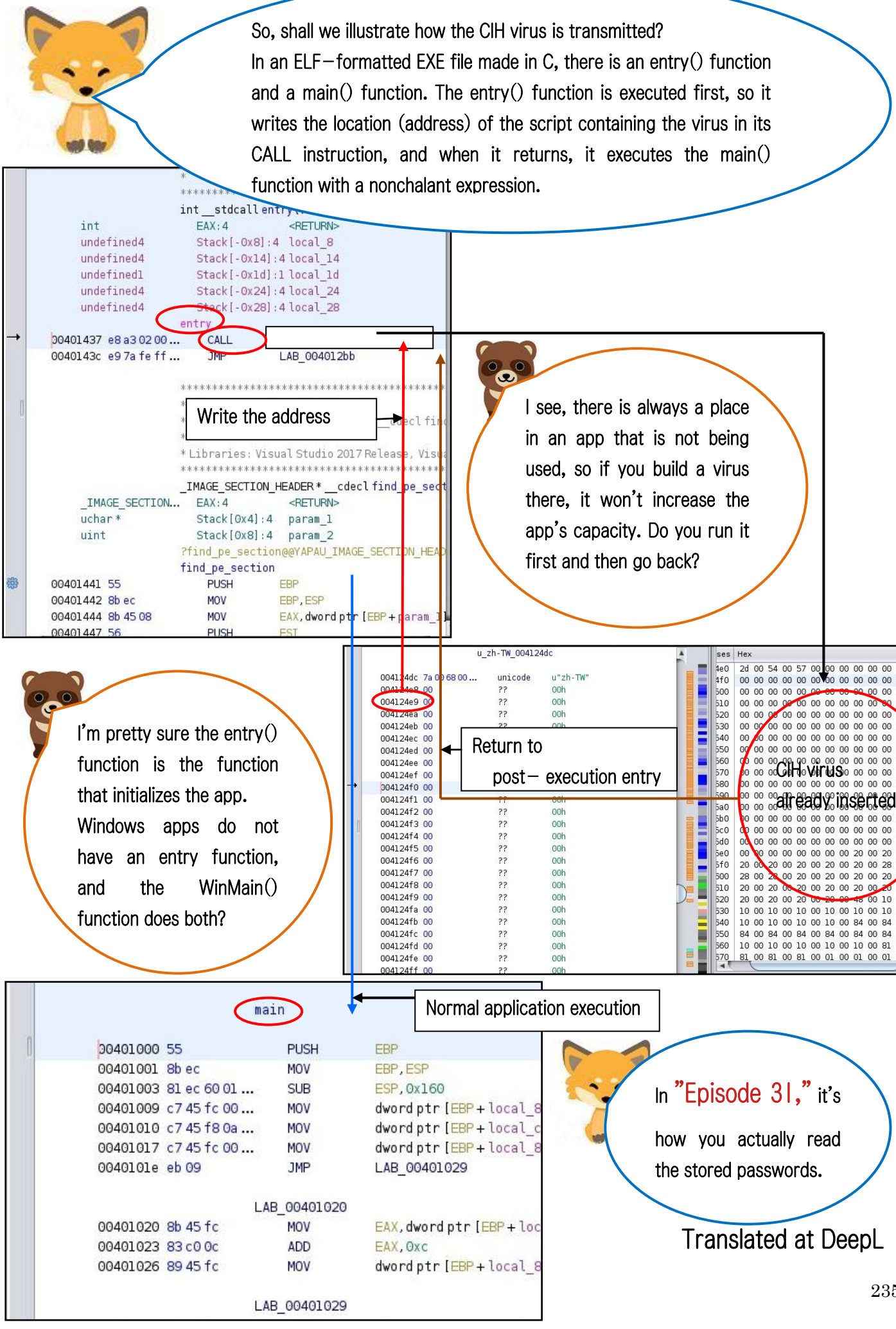



The ELF file shows the name of the main() function as it is, so this is easy.


Symbol Tree
frame_dummy
main
Functions
Labels
Classes
Filter:


Decompile...
1
2 void main(void)
3
4 {
5 undefined local_18[12];
6 int local_c;
7
8 local_c = open("data",0);
9 read(local_c,local_18,5);
10 write(1,local_18,5);
11 close(local_c);
12 return;
13 }
14
Decompile: main
Bytes: er


Listing: emfd.exe
LAB_00400608
00400608 e9 73 ff ff ... JMP register_tm_cl
-- Flow Override: CALL_RETURN (C
undefined __stdcall main(void)
undefined AL:1 <RETURN>
undefined4 Stack[-0xc]:4 local_c
undefined1 Stack[-0x18]:1 local_18
main
00400600 55 PUSH RBP
0040060e 48 89 e5 MOV RBP,RSP
00400611 48 83 ec 10 SUB RSP,0x10
00400615 be 00 00 00 ... MOV ESI,0x0
0040061a bf 10 07 40 ... MOV EDI=>DAT_00400710,DAT_00400710
0040061f b8 00 00 00 ... MOV EAX,0x0
00400624 e8 e7 fe ff ... CALL <EXTERNAL>::open
00400629 89 45 fc MOV dword ptr [RBP + local_c],EAX
0040062c 48 8d 4d f0 LEA RCX=>local_18,[RBP + -0x10]


So, shall we illustrate how the CIH virus is transmitted?
In an ELF-formatted EXE file made in C, there is an entry() function and a main() function. The entry() function is executed first, so it writes the location (address) of the script containing the virus in its CALL instruction, and when it returns, it executes the main() function with a nonchalant expression.
int __stdcall entry(
int EAX:4 <RETURN>
undefined4 Stack[-0x8]:4 local_8
undefined4 Stack[-0x14]:4 local_14
undefined1 Stack[-0x1d]:1 local_1d
undefined4 Stack[-0x24]:4 local_24
undefined4 Stack[-0x28]:4 local_28
entry
00401437 e8 a3 02 00 ... CALL
0040143c e9 7a fe ff ... JMP LAB_004012bb
Write the address
* Libraries: Visual Studio 2017 Release, Visu
_IMAGE_SECTION_HEADER * __cdecl find_pe_sect
_IMAGE_SECTION... EAX:4 <RETURN>
uchar * Stack[0x4]:4 param_1
uint Stack[0x8]:4 param_2
find_pe_section
00401441 55 PUSH EBP
00401442 8b ec MOV EBP,ESP
00401444 8b 45 08 MOV EAX,dword ptr [EBP + param_1]
00401447 56 PUSH ESI
I see, there is always a place in an app that is not being used, so if you build a virus there, it won't increase the app's capacity. Do you run it first and then go back?
u_zh-TW_004124dc
004124dc 7a 00 68 00 ... unicode u"zh-TW"
004124e9 00 ?? 00h
004124ea 00 ?? 00h
004124eb 00 ?? 00h
004124ec 00 ?? 00h
004124ed 00 ?? 00h
004124ee 00 ?? 00h
004124ef 00 ?? 00h
004124f0 00 ?? 00h
Return to post- execution entry
CIH virus already inserted
I'm pretty sure the entry() function is the function that initializes the app. Windows apps do not have an entry function, and the WinMain() function does both?
main
00401000 55 PUSH EBP
00401001 8b ec MOV EBP,ESP
00401003 81 ec 60 01 ... SUB ESP,0x160
00401009 c7 45 fc 00 ... MOV dword ptr [EBP + local_8
00401010 c7 45 f8 0a ... MOV dword ptr [EBP + local_c
00401017 c7 45 fc 00 ... MOV dword ptr [EBP + local_8
0040101e eb 09 JMP LAB_00401029
LAB_00401020
00401020 8b 45 fc MOV EAX,dword ptr [EBP + loc
00401023 83 c0 0c ADD EAX,0xc
00401026 89 45 fc MOV dword ptr [EBP + local_8
LAB_00401029
Normal application execution
In "Episode 31," it's how you actually read the stored passwords.
Translated at DeepL