Episode 29 (Malware Analysis I)

Using the Ghidra Tool

Tanuki, in order to perform reverse engineering, you need the right tools.
Reverse engineering tools include "IDA Freewire" and "Ghidra. IDA Freewire is a free version of the commercial product IDA Pro. Ghidra is an open source software developed by the National Security Agency (NSA). We will use this one because it is a tool that is currently attracting a lot of attention.
I do not intend to explain the operation of "Ghidra" in the same way as before, so please study the details of "Ghidra" by yourself. The manual of "Ghidra" alone is more than 600 pages long.

Kitsune, I understand. I'll study the necessary parts of the operation myself on the net and in books. By the way, what are the prerequisites for understanding the contents of "Ghidra" ?

Tanuki, the prerequisites for understanding the contents of "Ghidra" are an understanding of assembler language, C language and binary (hexadecimal) files.
But don't worry. I will explain everything except the operations of "Ghidra" in an easy-to-understand manner using real-life examples.
Through reverse engineering, you will be reminded of the importance of assembler and C.

What's a real example? What are you going to use?

A simple C program (mfd.c) that uses only standard input/output (stdio.h) to read a text file (data) and display its contents on the screen.
Using a simple known source program, the script is only about four lines long, so it is clear how it is decompiled and disassembled by "Ghidra". However, it should be noted that when the executable file is created from the source code, stdio.h, which is also a C language program, is included. The following figure illustrates the situation.

mfd.c (source)

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int fd;
    char buf[10];
    fd = open("data",0);
    read(fd,buf,5);
    write(1,buf,5);
    close(fd);
}
```

Creating C executables on CentOS

```
$ gcc -o emfd.exe mfd.c
        executable file  source
```

Import stdio.h at the same time

Forcing memfd.exe to open

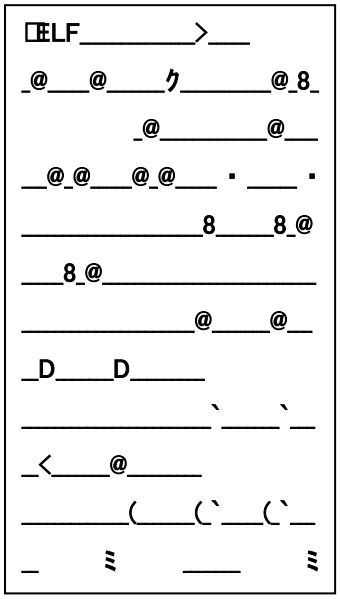
```
□ELF_____>___
_@___@____ク_____@_8_
        _@_____@__
__@_@___@_@___ ▪ ___ ▪
_____8____8_@
___8_@_____
_____@____@__
_D____D_____
_____`____`__
_<____@_____
_____(____(`___(`__
_   ≡    ____    ≡
```

Decompile & Disassemble with "Ghidra".

I see. I see...even if you open the executable file (emfd.exe) with notepad or something, it is still a mess and confusing. So, you revert (emfd.exe) to the assembler language or the original source language and study the mechanism of "Ghidra" by comparing them. Nice work, Kitsune.

Tanuki! Malware is embedded in an executable file (exe).
The next step is reverse engineering from the executable file.

emfd.exe



Open in Binary Editta

emfd.exe (machine language)

```
31 ed 49 89 d1 5e 48 89 e2 48 83 e4 f0 50 54 49
c7 c0 f0 06 40 00 48 c7 c1 80 06 40 00 48 c7 c7
0d 06 40 00 e8 a7 ff ff ff f4 66 0f 1f 44 00 00
b8 57 10 60 00 55 48 2d 50 10 60 00 48 83 f8 0e
48 89 e5 77 02 5d c3 b8 00 00 00 00 48 85 c0 74
f4 5d bf 50 10 60 00 ff e0 0f 1f 80 00 00 00 00
b8 50 10 60 00 55 48 2d 50 10 60 00 48 c1 f8 03
48 89 e5 48 89 c2 48 c1 ea 3f 48 01 d0 48 d1 f8
75 02 5d c3 ba 00 00 00 00 48 85 d2 74 f4 5d 48
89 c6 bf 50 10 60 00 ff e2 0f 1f 80 00 00 00 00
80 3d 85 0a 20 00 00 75 11 55 48 89 e5 e8 7e ff
ff ff 5d c6 05 72 0a 20 00 01 f3 c3 0f 1f 40 00
48 83 3d 38 08 20 00 00 74 1e b8 00 00 00 00 48
85 c0 74 14 55 bf 20 0e 60 00 48 89 e5 ff d0 5d
e9 7b ff ff ff 0f 1f 00 e9 73 ff ff ff 55 48 89
e5 48 83 ec 10 be 00 00 00 00 bf 10 07 40 00 b8
00 00 00 00 e8 e7 fe ff ff 89 45 fc 48 8d 4d f0
8b 45 fc ba 05 00 00 00 48 89 ce 89 c7 b8 00 00
00 00 e8 99 fe ff ff 48 8d 45 f0 ba 05 00 00 00
48 89 c6 bf 01 00 00 00 b8 00 00 00 00 e8 5e fe
ff ff 8b 45 fc 89 c7 b8 00 00 00 00 e8 5f fe ff
ff c9 c3 66 2e 0f 1f 84 00 00 00 00 00 0f 1f 00
41 57 41 89 ff 41 56 49 89 f6 41 55 49 89 d5 41
54 4c 8d 25 78 07 20 00 55 48 8d 2d 78 07 20 00
```

disassembly

Assembler language of "emfd.exe"

```
0040060d 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]
00400630 8b 45 fc        MOV     EAX,dword ptr [RBP + local_c]
00400633 ba 05 00 00 ...  MOV     EDX,0x5
00400638 48 89 ce        MOV     RSI,RCX
0040063b 89 c7           MOV     EDI,EAX
0040063d b8 00 00 00 ...  MOV     EAX,0x0
00400642 e8 99 fe ff ...  CALL    <EXTERNAL>::read
00400647 48 8d 45 f0     LEA     RAX=>local_18,[RBP + -0x10]
0040064b ba 05 00 00 ...  MOV     EDX,0x5
00400650 48 89 c6        MOV     RSI,RAX
00400653 bf 01 00 00 ...  MOV     EDI,0x1
00400658 b8 00 00 00 ...  MOV     EAX,0x0
0040065d e8 5e fe ff ...  CALL    <EXTERNAL>::write
00400662 8b 45 fc        MOV     EAX,dword ptr [RBP + local_c]
00400665 89 c7           MOV     EDI,EAX
00400667 b8 00 00 00 ...  MOV     EAX,0x0
0040066c e8 5f fe ff ...  CALL    <EXTERNAL>::close
00400671 c9              LEAVE
00400672 c3              RET
```

main () Beginning of function

I can see the reverse engineering going on here.
So 55 in hex is the beginning of the main function?
That's great fox!

decompile

Yes, you can decompile from the assembler language back to the script in the original source file.
So, by comparing the decompiled script with the script in the original source file, you can see that the contents of the main() function are almost the same. The only difference is the variable names. The difference is only in the variable names, since we can't read the names of the variables in the original source file.

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
```

[Comparison of the decompiled file with the main source file (mfd.c)]

decompile

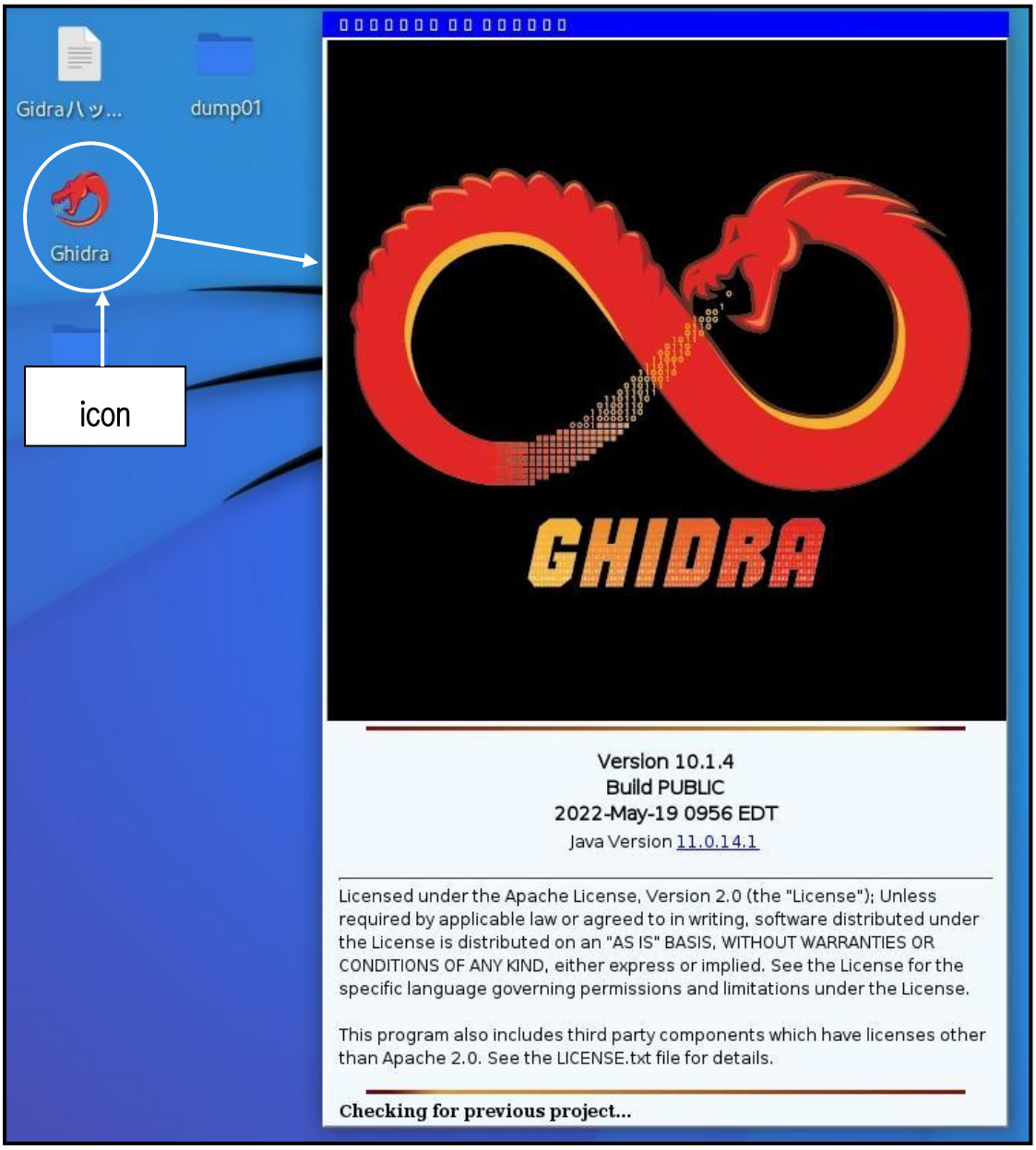
```
1
2  void main(void)
3
4  {
5   undefined local_18[12];
6   int local_c;
7
8   local_c = open("data",0);
9   read(local_c,local_18,5);
10  write(1,local_18,5);
11  close(local_c);
12  return;
13 }
14
```

The Big Difference variable name

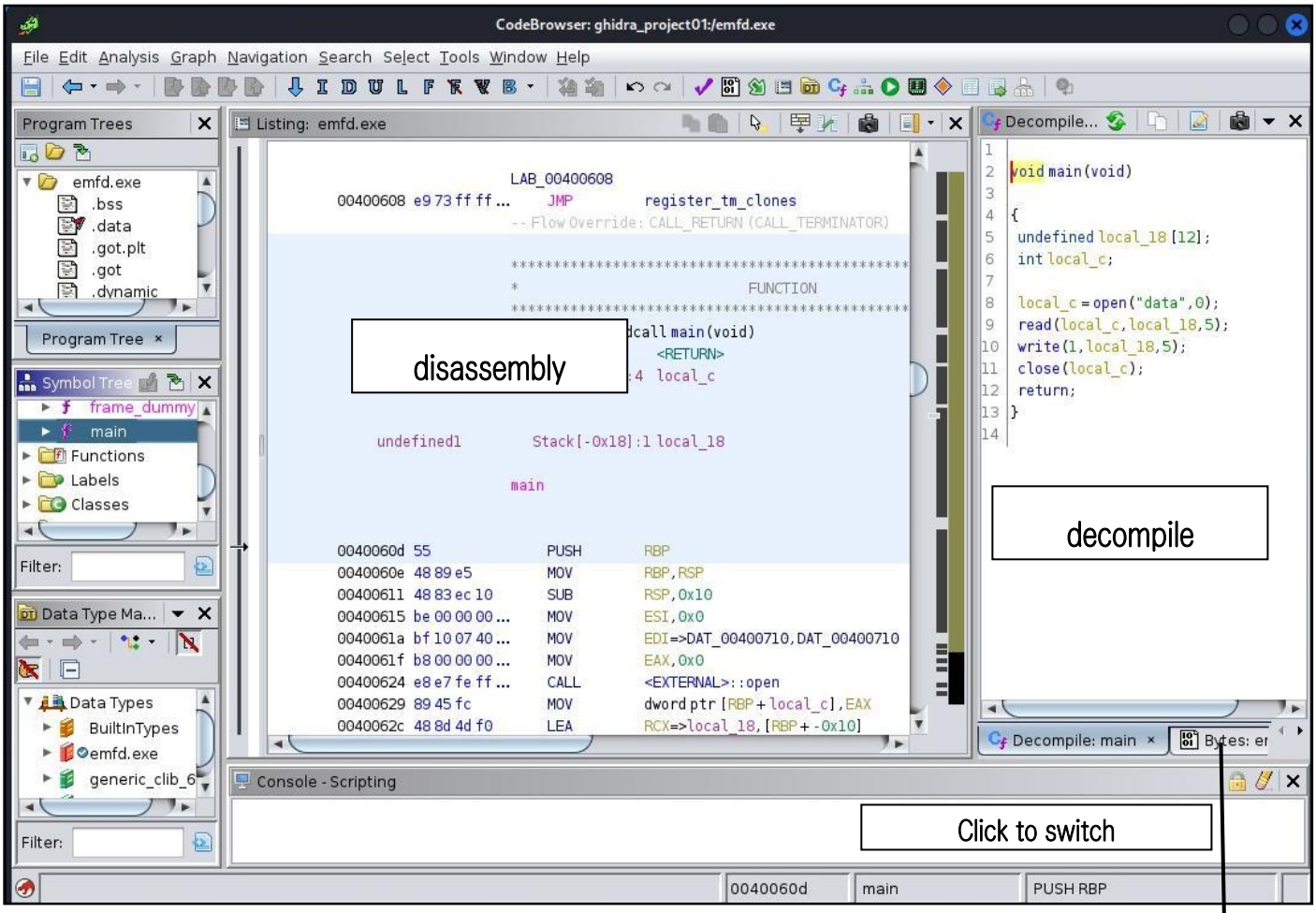
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int fd;
    char buf[10];

    fd = open("data",0);
    read(fd,  buf,  5);
    write(1,  buf,  5);
    close(fd);
}
```

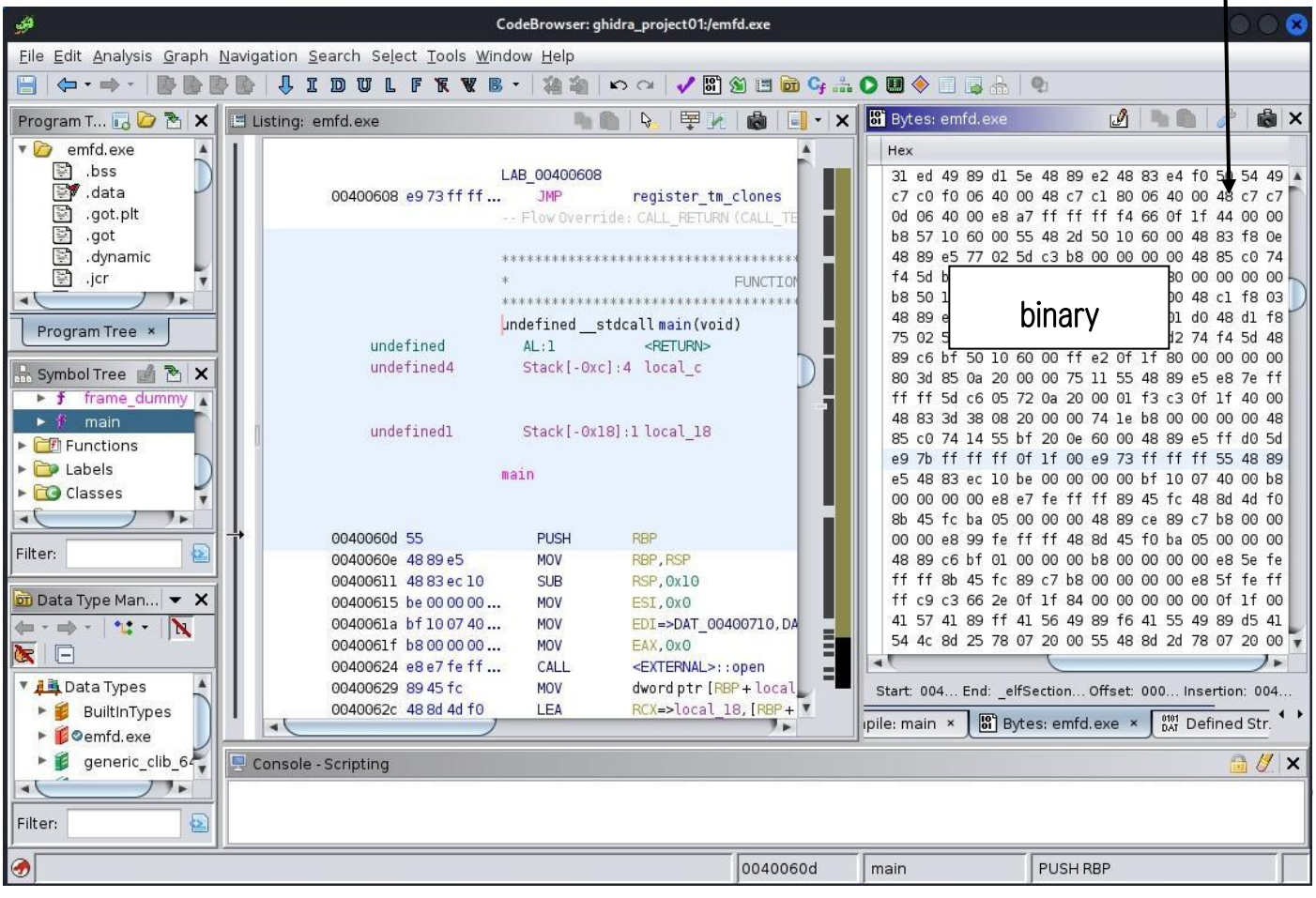
Tanuki, the "Ghidra" tool reads the executable file (emfd.exe), performs binary editing, disassembly, and decompilation all at once, and displays the results. Below is a diagram of the Ghidra tool after loading (emfd.exe). However, as usual, the installation procedure of "Ghidra" to "kali Linux" is not explained here, so please look it up on the net if necessary.

[Launch Ghidra]



[Importing (loading) “emfd.exe” into Ghidra]





Kitsune, "Ghidra" (Ghidorah), a virtual monster, looks amazing in the opening screen.
It looks like it's going to do something bad.

Reverse engineering itself involves stealing the contents of a developed program, you know. But here, it is used to analyze malware, learn how it works, and protect your PC.

In "Episode 30," we will study emfd.exe using "Ghidra.

Translated at DeepL