



CM0588 – Malware Reverse Engineering Lab

Paolo Falcarin – April 2025

1. Disabling a malicious trigger in Ghidra.

Download the **malware-example-ghidra.zip** file and extract the following files:

1. malware.c,
2. malware.exe,
3. Obfmalware.c,
4. Obfmalware.exe

Don't worry, this is not real malware just a toy example! 😊

The program has been obfuscated using the following command:

```
tigress --Environment=x86_64:Linux:Gcc:4.6 --Transform=Flatten --Functions=main --Transform=InitOpaque --Functions=main --out=Obfmalware.c ./malware.c
```

You have been given a program `Obf_malware.exe` that has been generated and obfuscated for you. Using NSA's Ghidra tool, find and disable the malicious trigger.

Examining the program:

`Obf_malware.exe` is a number guessing game. Try different numbers to see if you can get the correct number. Once you get the correct number, you may see something like this.

```
liger@liger:~$ ./malware.exe
Guess a number between 0-9.
5
Yes, that's my number, congratulations.
You've activated the malicious trigger!
```

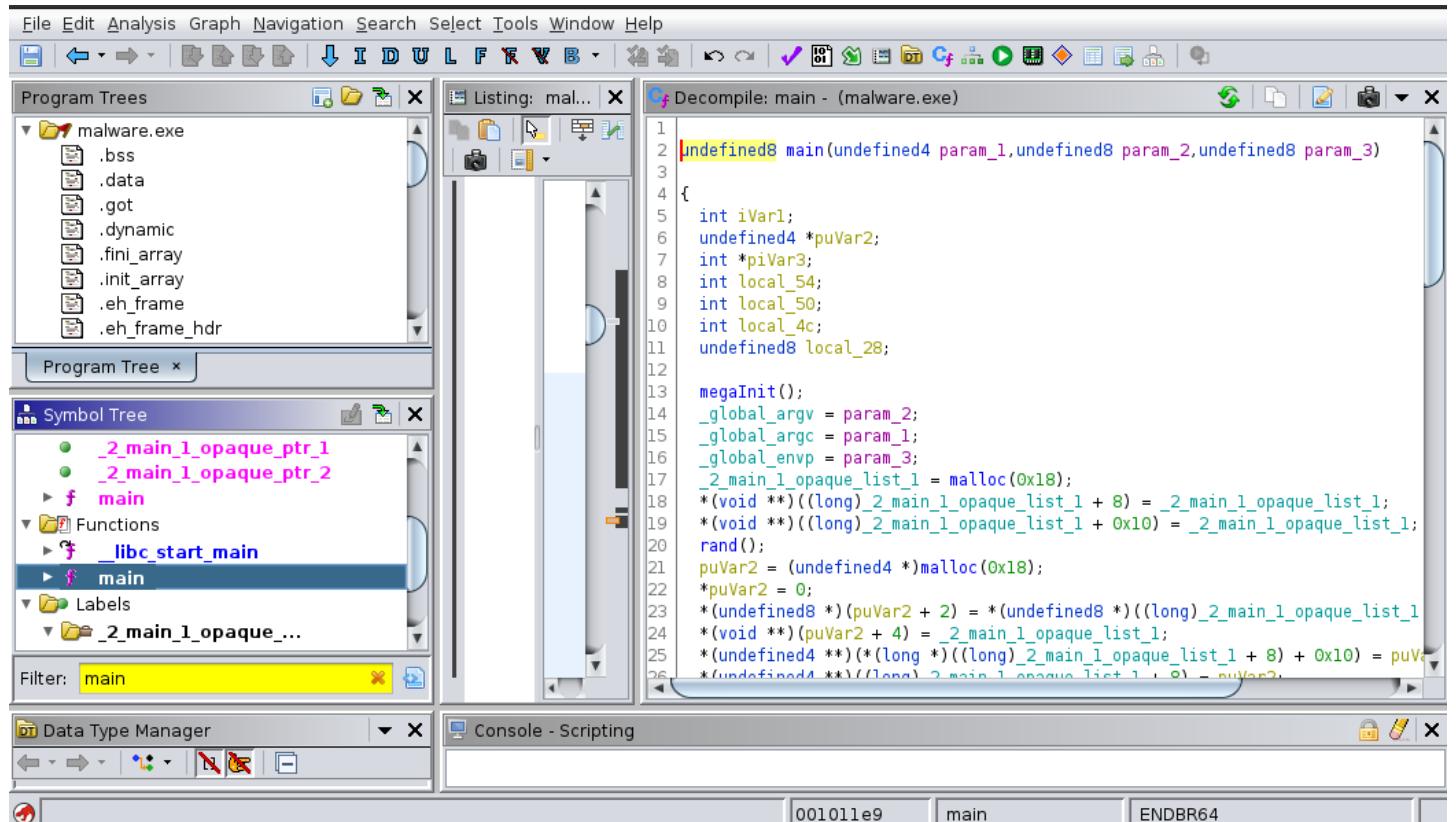
This program activates malicious code once the user guesses the correct number. We want to disable this so that the number guessing game can be played normally, without activating the malicious code.

Analysing the program:

Open ghidra, import `Obf_malware.exe` with default settings, and when asked to perform an analysis, select yes, and analyse with the default settings.

To begin disabling the malicious trigger, let's start by looking in the main function.

In the System Tree window, filter for the function *main*



If you click on the main function, you should see some code in the *Decompile* window.

Let's explore the main function a bit more. If you scroll down, you should run into this.

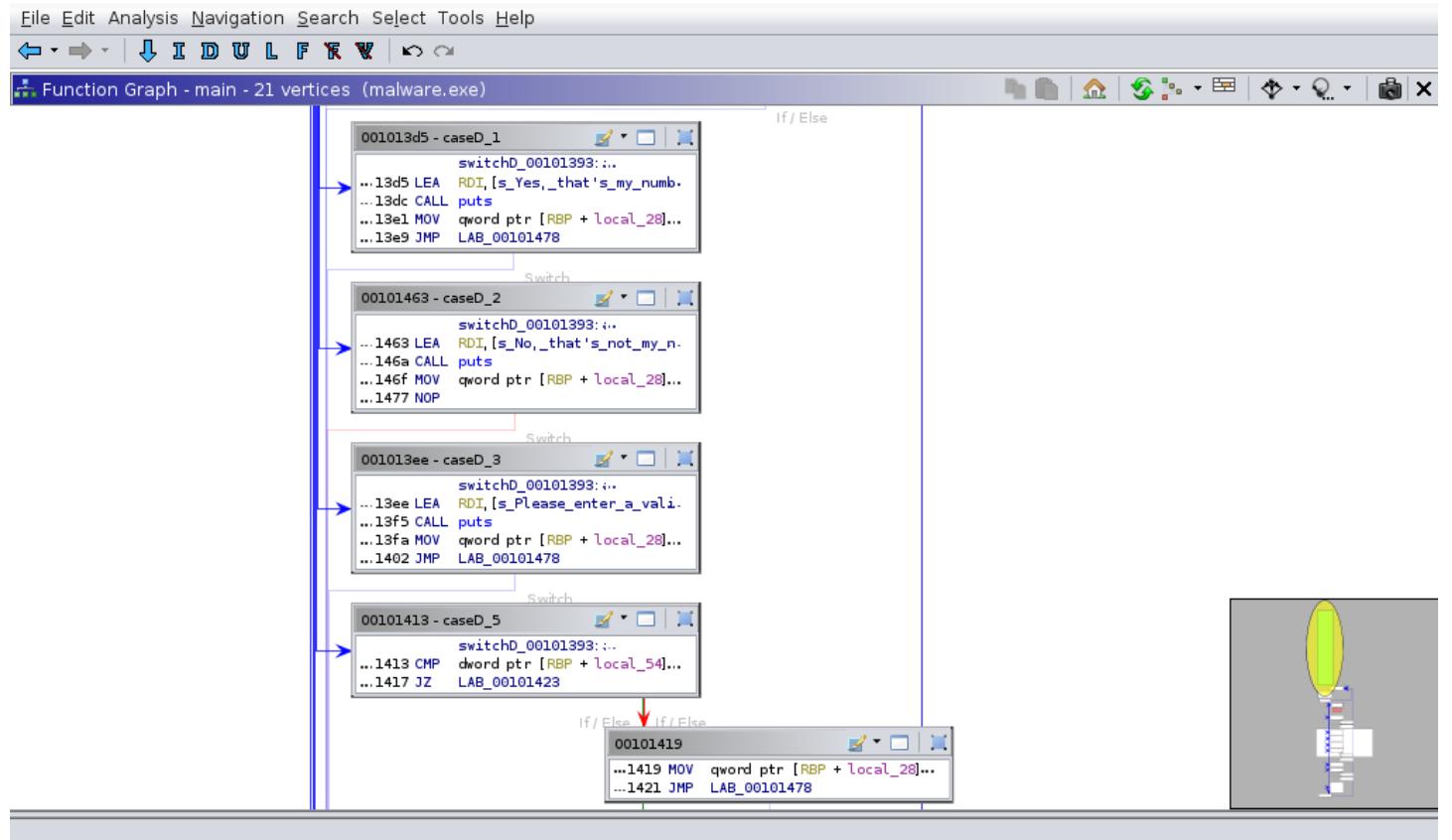
The screenshot shows the Decompile window with the following pseudocode:

```
switch(local_28) {
    case 0:
        if (local_4c == local_50) {
            local_28 = 1;
        }
        else {
            local_28 = 2;
        }
        break;
    case 1:
        puts("Yes, that's my number, congratulations.");
        local_28 = 5;
        break;
    case 2:
        puts("No, that's not my number.");
        local_28 = 4;
        break;
    case 3:
        puts("Please enter a valid number between 0-9.");
        local_28 = 4;
        break;
    case 4:
        return 0;
    case 5:
        if (local_54 == 0) {
```

Now we see where the text that prints out the congratulations message appears. We know that the malicious trigger happens after this text.

You may notice a large switch statement, using the variable `local_28`. Let's follow the switch statement to see what happens after the congratulations message.

(If you have trouble following the switch statement, while the main function is selected, you can open the Function Graph under the Window tab.)



After the program prints the congratulations message, the value of `local_28` changes to 5. Let's see what happens at case 5.

```
49     local_28 = 5;
50     break;
51 case 2:
52     puts("No, that's not my number.");
53     local_28 = 4;
54     break;
55 case 3:
56     puts("Please enter a valid number between 0-9.");
57     local_28 = 4;
58     break;
59 case 4:
60     return 0;
61 case 5:
62     if (local_54 == 0) {
63         local_28 = 4;
64     }
65     else {
66         local_28 = 6;
67     }
68     break;
69 case 6:
70     payload();
71     local_28 = 4;
72     break;|
73 case 7:
74     if (local_4c == -1) {
```

We can see here that after case 5, there are two options. `local_28` changes to 4, or `local_28` changes to 6.

If we look at case 4, it returns 0, ending the program. If we look at case 6, the function activates a payload. This is the malicious trigger that happens.

Patching instructions:

We want to prevent the malicious trigger from occurring. One way to do this is to ensure that case 6 will not occur.

Let's try changing `local_28 = 6` in case 5 to `local_28 = 4`.

The screenshot shows the Immunity Debugger interface with two windows side-by-side. The left window is the 'Listing' window for 'malware.exe', displaying assembly code. The right window is the 'Decompile' window for 'main - ...', showing the corresponding C code. In the assembly listing, there is a MOV instruction at address 0x00101393 that moves the value 0x6 to the register RBP + local_28. This line is highlighted with a blue selection bar. The decompiled C code shows a switch statement where case 6 sets local_28 to 6.

```

switchD_00101393::cased_6
CALL    payload

MOV     qword ptr [RBP + local_28],0x4

JMP     LAB_00101478

switchD_00101393::cased_5
0   CMP    dword ptr [RBP + local_54],0x0
JZ      LAB_00101423
MOV     qword ptr [RBP + local_28],0x6

JMP     LAB_00101478

LAB_00101423
MOV     qword ptr [RBP + local_28],0x4

JMP     LAB_00101478

switchD_00101393::cased_0
MOV     EAX,dword ptr [RBP + local_4c]
CMP    EAX,dword ptr [RBP + local_50]

```

```

local_28 = 5;
break;
case 2:
    puts("No, that's not my number.");
    local_28 = 4;
    break;
case 3:
    puts("Please enter a valid number be");
    local_28 = 4;
    break;
case 4:
    return 0;
case 5:
    if (local_54 == 0) {
        local_28 = 4;
    }
    else {
        local_28 = 6;
    }
    break;
case 6:
    payload();
    local_28 = 4;
    break;
case 7:
    if (local_4c == -1) {

```

If you click on `local_28 = 6`, the Listing window will jump to the assembly instruction that it corresponds to. It appears that the value `0x6` is moved to `local_28` using a `MOV` instruction.

Let's change the `0x6` into `0x4`. Right click on the assembly instruction, and select `Patch Instruction`.

The screenshot shows the Immunity Debugger interface with the assembly listing window. A context menu is open over the assembly instruction at address 0x00101393, specifically over the `MOV qword ptr [RBP + local_28],0x6` line. The 'Patch Instruction' option in the menu is highlighted with a red box.

Context menu options include:

- Bookmark... Ctrl+D
- Clear Code Bytes C
- Clear With Options...
- Clear Flow and Repair...
- Copy Ctrl+C
- Copy Special...
- Paste Ctrl+V
- Comments ▶ e 3: ▶ e 4: ▶ e 5: ▶ e 6: ▶ e 7: ▶ f (local_4c == -1) {
- Instruction Info...
- Patch Instruction Ctrl+Shift+G**
- Processor Manual...
- Processor Options...
- Create Function F
- Create Thunk Function
- Function ▶ local_28 = 6;
- Add Label... L
- Show Label History... H
- Clear Register Values...
- Set Register Values... Ctrl+R
- Colors ▶ local_28 = 4;

You should be able to edit the instruction now. Change `0x6` to `0x4`, then press Enter.

The screenshot shows two panes of the Immunity Debugger interface. The left pane, titled 'Listing: malware.exe', displays assembly code. The right pane, titled 'Decompile: main - ...', shows the generated C code. The assembly code includes instructions like 'CALL payload' and 'MOV qword ptr [RBP + local_28], 0x4'. The C code includes a switch statement with cases 0 through 7, each containing logic to set 'local_28' to 4 or break out of the loop.

```

switchD_00101393::caseD_6 XREF[1]: 00101393
    CALL    payload
    MOV     qword ptr [RBP + local_28], 0x4

    JMP     LAB_00101478

switchD_00101393::caseD_5 XREF[1]: 00101393
0  CMP     dword ptr [RBP + local_54], 0x0
JZ      LAB_00101423
    MOV     qword ptr [RBP + -0x20], 0x4

    JMP     LAB_00101478

LAB_00101423 XREF[1]: 00101417
    MOV     qword ptr [RBP + local_28], 0x4

    JMP     LAB_00101478

switchD_00101393::caseD_0 XREF[1]: 00101393
    MOV     EAX,dword ptr [RBP + local_4c]
    CMP     EAX,dword ptr [RBP + local_50]

```

```

49   local_28 = 5;
50   break;
case 2:
52   puts("No, that's not my number.");
53   local_28 = 4;
54   break;
case 3:
56   puts("Please enter a valid number below 5.");
57   local_28 = 4;
58   break;
case 4:
60   return 0;
case 5:
62   if (local_54 == 0) {
63       local_28 = 4;
64   }
65   else {
66       local_28 = 4;
67   }
68   break;
case 6:
70   payload();
71   local_28 = 4;
72   break;
case 7:
73   if (local_4c == -1) {
74

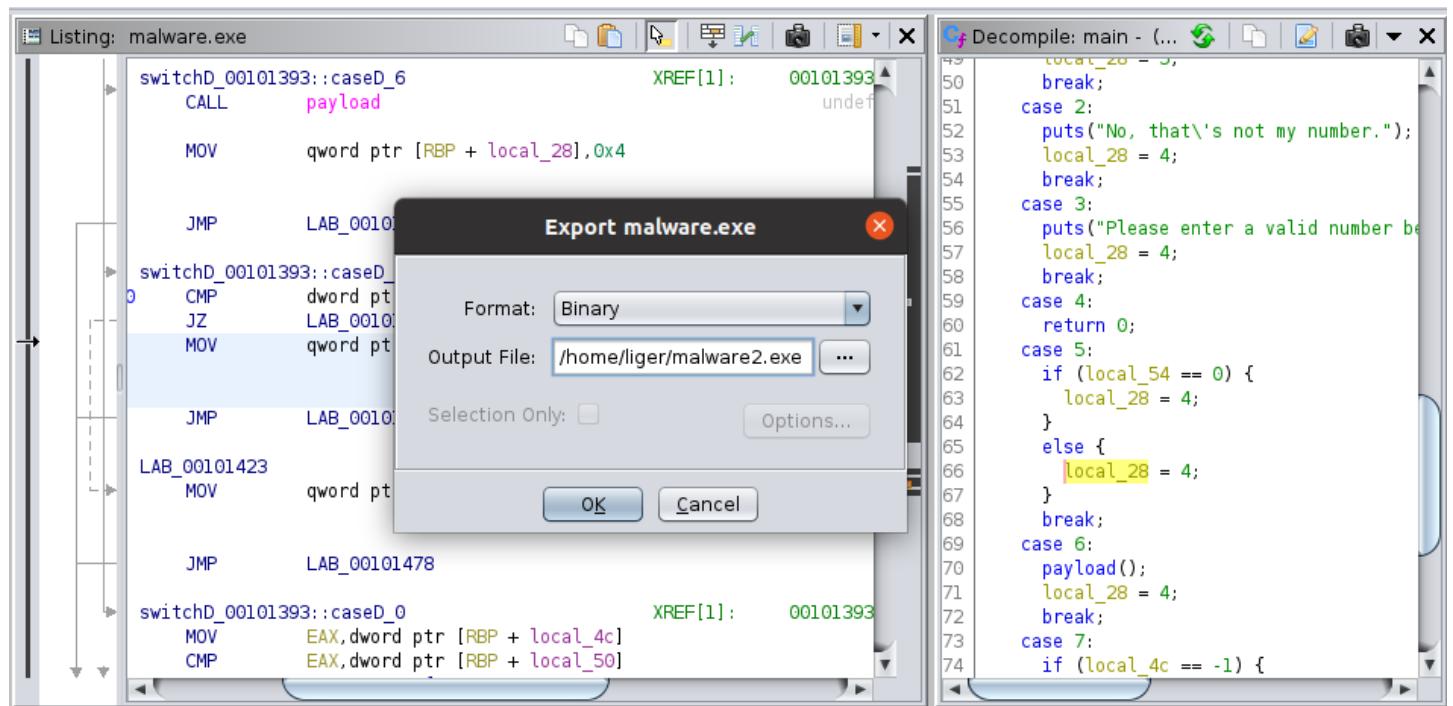
```

You should see now that the instruction has been changed. In the decompile menu, the code has been changed as well.

Now, when the user correctly guesses the number, instead of jumping to the payload, it should simply end the program as normal.

Exporting as binary:

To export the edited program, go to **File > Export Program**, and export as Binary.



You can rename the output file here. In this case, I have renamed it to malware2.

Now, you can run the program that ghidra just exported. When you guess the correct number this time, it should display the congratulations message, and end the program without triggering the malicious code.

If the program works, you are done! However, you may encounter a seg fault instead. This is due to how the export feature works in ghidra. The next section below outlines a way you can work around this with an alternative way to patch the instruction.

Patching using byte window:

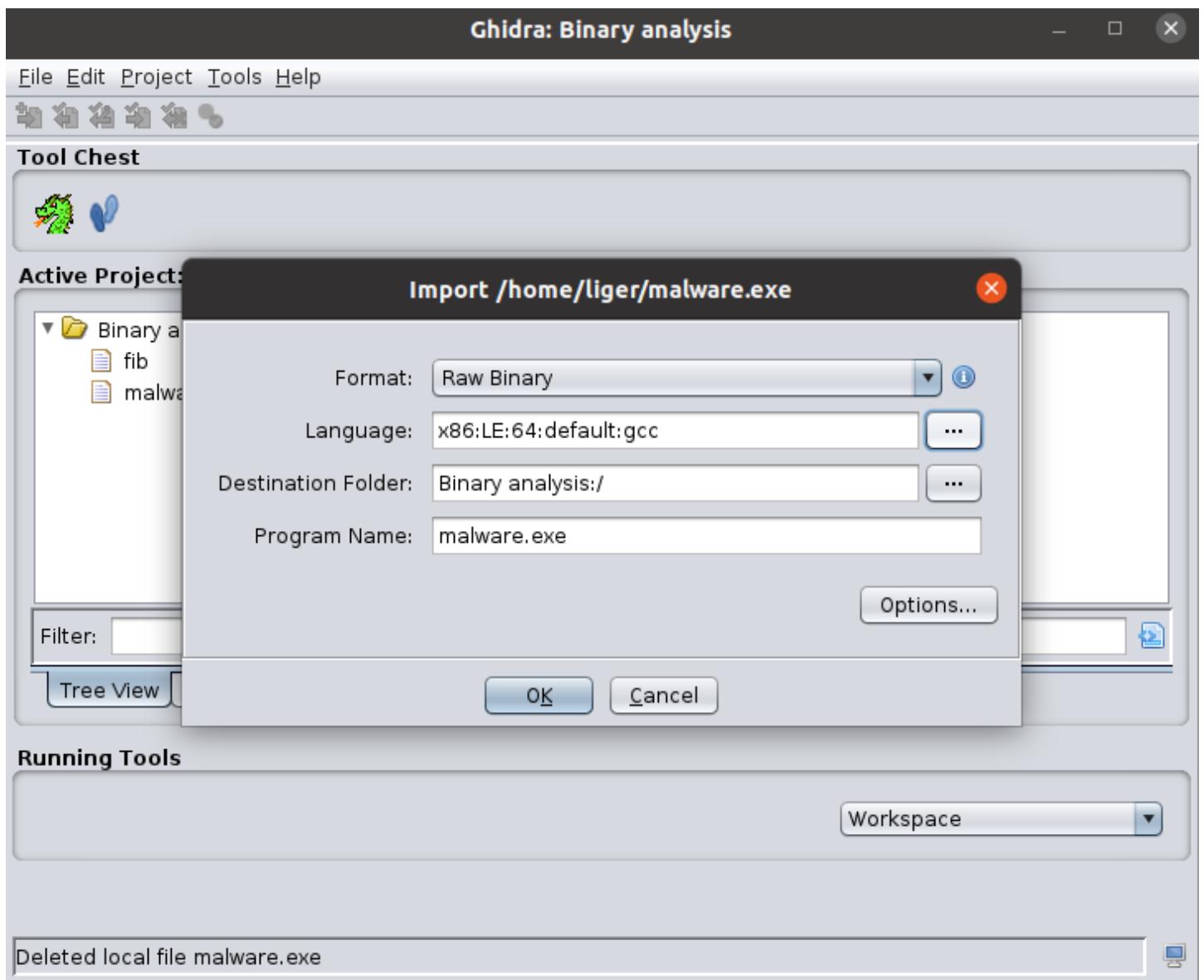
If you run into problems, here's an alternative way to edit a binary in ghidra and export it. Keep note of the location of the instruction you intend to patch.

The screenshot shows the Ghidra interface with two main panes. The left pane is the 'Listing' view, showing assembly code for the 'main' function. The right pane is the 'Decompile' view, showing the corresponding C code. A specific instruction at address 00101419 is highlighted with a red rectangle in the assembly listing. This instruction is a MOV qword from [RBP + local_28] to a register, with the value 0x6. The C code shows a switch statement with case 6 set to 'payload();'. The assembly listing also includes labels like LAB_00101478 and LAB_00101423, and various jumps and moves.

```
Listing: malware.exe
00101402 80 74      JMP     LAB_00101478
00101404 e8 76 00    CALL    payload
00 00
00101409 48 c7 45    MOV     qword ptr [RBP + local_28], 0x4
e0 04 00
00 00
00101411 eb 65      JMP     LAB_00101478
switchD_00101393::caseD_6
00101413 83 7d b4 00  CMP     dword ptr [RBP + local_54], 0x0
00101417 74 0a      JZ      LAB_00101423
00101419 48 c7 45    MOV     qword ptr [RBP + local_28], 0x6
e0 06 00
00 00
00101421 eb 55      JMP     LAB_00101478
00101423 48 c7 45    MOV     qword ptr [RBP + local_28], 0x4
e0 04 00
00 00
0010142b eb 4b      JMP     LAB_00101478
LAB_00101423
0010142d 8b 45 hc    MOV     EAX, dword ptr [RBP + local_4c]
switchD_00101393::caseD_0
Decompile: main
49 local_28 = 5;
50 break;
51 case 2:
52 puts("No, tha");
53 local_28 = 4;
54 break;
55 case 3:
56 puts("Please");
57 local_28 = 4;
58 break;
59 case 4:
60 return 0;
61 case 5:
62 if (local_54)
63   local_28 =
64 }
65 else {
66   local_28 =
67 }
68 break;
69 case 6:
70 payload();
Cf Decompile: main
```

In this case, it's `48 c7 45 e0 06 00 00 00`. We want to change the `06` into a `04`.

Import `Obf_malware.exe` again, this time, with different import instructions. Select the format `Raw Binary`, and select the same language as before.



Run the analysis again. You may notice that the analysis this time is a lot less clear.

Press **s** to open up the search window. Select hex, and search for the hex of the position of the instruction you want to patch.

When you find the instruction, click on it and open up the Bytes window under the Window tab. At the top of the window, select the pencil icon, to allow editing. Now you can edit the instruction.

In this case, we change the **06** to **04**, highlighted here in red.

Bytes: malware.exe

Addresses	Hex
00001340	89 05 f2 2c 00 00 48 8b 05 eb 2c 00 00 48 89 05
00001350	ec 2c 00 00 c7 45 d0 01 00 00 00 c7 45 d4 01 00
00001360	00 00 48 c7 45 e0 08 00 00 00 48 83 7d e0 08 77
00001370	f9 48 8b 45 e0 48 8d 14 85 00 00 00 00 48 8d 05
00001380	f0 0c 00 00 8b 04 02 48 98 48 8d 15 e4 0c 00 00
00001390	48 01 d0 3e ff e0 b8 00 00 00 e9 dd 00 00 00
000013a0	bf 01 00 00 00 e8 ec 00 00 00 89 45 d8 8b 45 d8
000013b0	89 45 b4 c7 45 b8 05 00 00 00 e8 f2 00 00 00 89
000013c0	45 dc 8b 45 dc 89 45 bc 48 c7 45 e0 07 00 00 00
000013d0	e9 a3 00 00 00 48 8d 3d 2c 0c 00 00 e8 bf fc ff
000013e0	ff 48 c7 45 e0 05 00 00 00 e9 8a 00 00 00 48 8d
000013f0	3d 3b 0c 00 00 e8 a6 fc ff 48 c7 45 e0 04 00
00001400	00 00 eb 74 e8 76 00 00 00 48 c7 45 e0 04 00 00
00001410	00 eb 65 83 7d b4 00 74 0a 48 c7 45 e0 04 00 00
00001420	00 eb 55 48 c7 45 e0 04 00 00 00 eb 4b 8b 45 bc
00001430	3b 45 b8 75 0a 48 c7 45 e0 01 00 00 00 eb 39 48
00001440	c7 45 e0 02 00 00 00 eb 2f 83 7d bc ff 75 0a 48
00001450	c7 45 e0 03 00 00 00 eb 1f 48 c7 45 e0 00 00 00
00001460	00 eb 15 48 8d 3d ef 0b 00 00 e8 31 fc ff 48
00001470	c7 45 e0 04 00 00 00 90 e9 ed fe ff ff c9 c3 f3
00001480	0f 1e fa 55 48 89 e5 48 8d 3d 0a 0c 00 00 e8 0d
00001490	fc ff ff 90 5d c3 f3 0f 1e fa 55 48 89 e5 89 7d
000014a0	fc 8b 45 fc 5d c3 f3 0f 1e fa 55 48 89 e5 90 5d

Start: 00000000 End: 000043df Offset: 00000000 Insertion: 0000141e

Cf Decompiler × Bytes: malware.exe ×

After editing the instruction, you can now export it as binary like before.

Run the exported program and try guessing the correct number.

```
liger@liger:~$ ./malware3.exe.bin
Guess a number between 0-9.
5
Yes, that's my number, congratulations.
liger@liger:~$
```

Now the congratulation message appears, and no malicious trigger is activated. You are done!