

Part1. Ghidra tutorial

Installing JDK

- Since Ghidra is written in JAVA, we need to download and install JDK.
<https://adoptopenjdk.net/releases.html?variant=openjdk11&jvmVariant=hotspot>
- Require an Oracal account.

Ghidra GUI

- Download Ghidra from <https://ghidra-sre.org/> and extract the file.

名稱	修改日期	類型	大小
docs	2020/12/15 下午 ...	檔案資料夾	
Extensions	2020/12/15 下午 ...	檔案資料夾	
Ghidra	2020/12/15 下午 ...	檔案資料夾	
GPL	2020/12/15 下午 ...	檔案資料夾	
licenses	2020/12/15 下午 ...	檔案資料夾	
server	2020/12/15 下午 ...	檔案資料夾	
support	2020/12/15 下午 ...	檔案資料夾	
ghidraRun	2020/12/15 下午 ...	檔案	1 KB
ghidraRun.bat	2020/12/15 下午 ...	Windows 批次檔案	1 KB
LICENSE	2020/12/15 下午 ...	檔案	12 KB

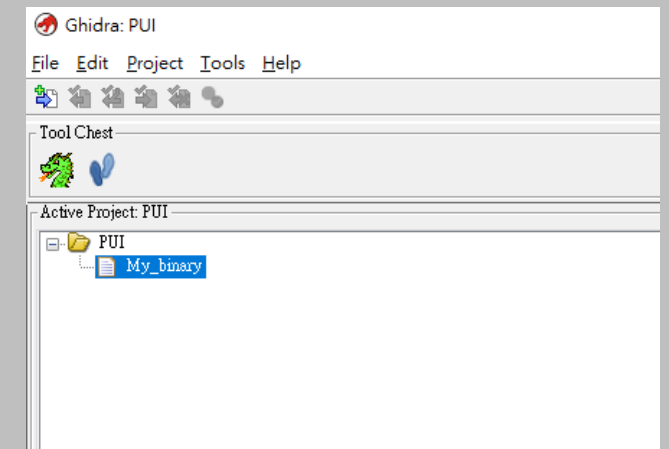
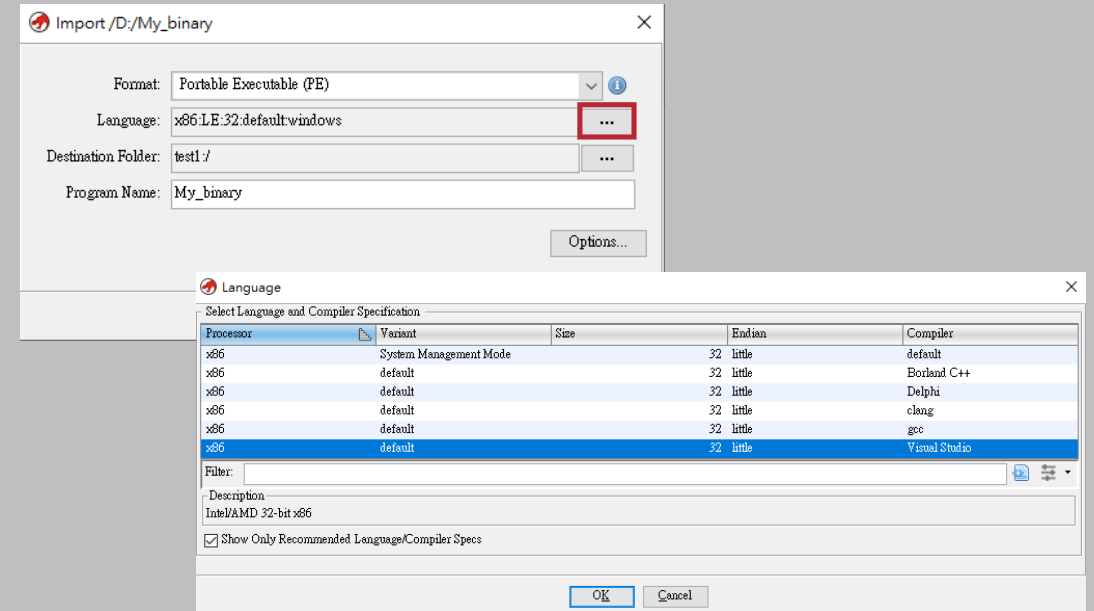
- On windows system, Run ghidraRun.bat to launch GUI.

Create a new project

- [File]/[New Project]
- Non-Shared Project
- Enter project name
- [Finish]

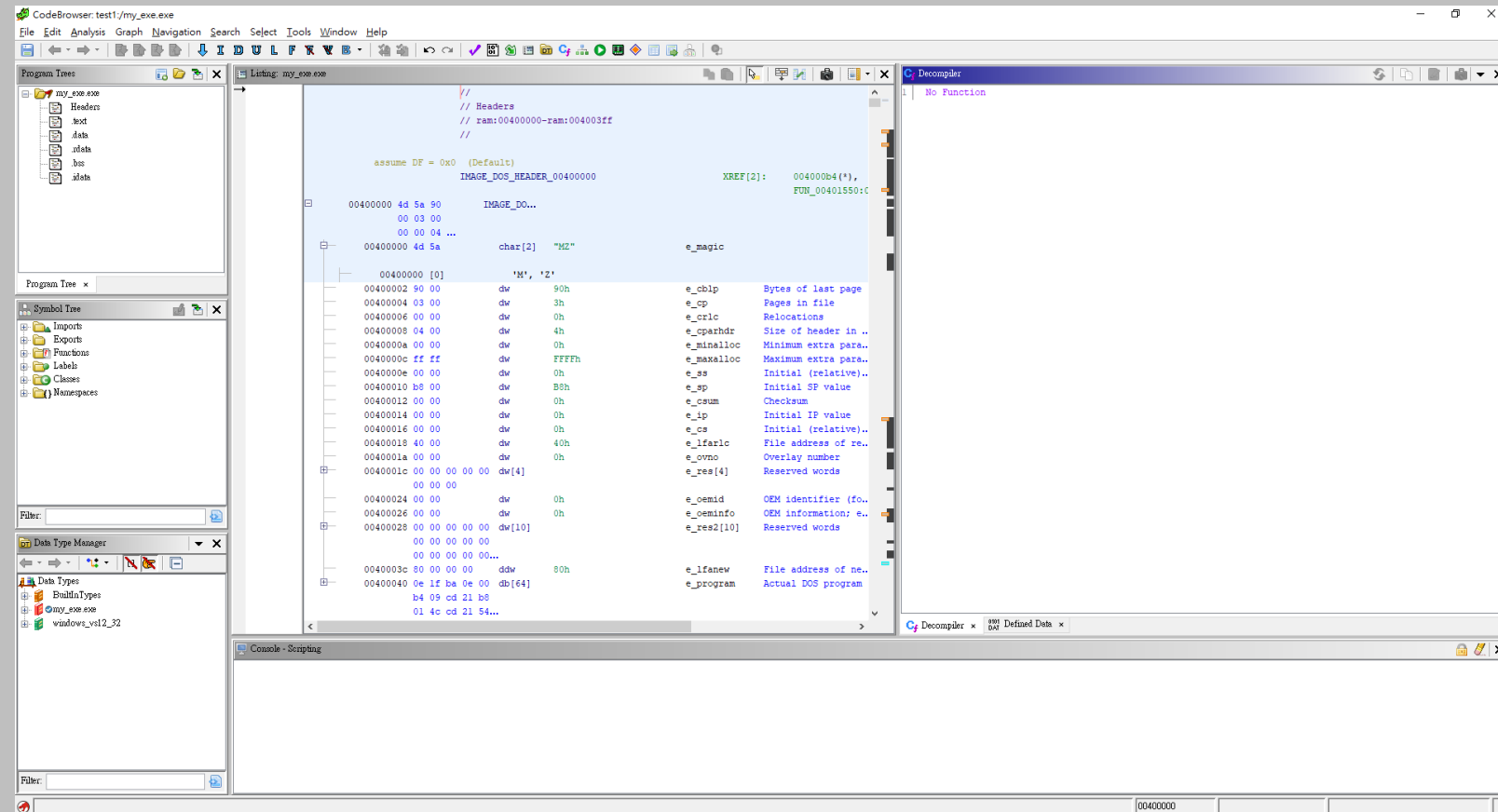
Import binary file

- [File]/[Import file]
- Select the file you want to analysis.
- You can choose the language of the file, or let Ghidra do it itself.
- Press [OK].
- Double click the file to analyze.
- On [Analyze] window, click [Yes] and [Analyze] .

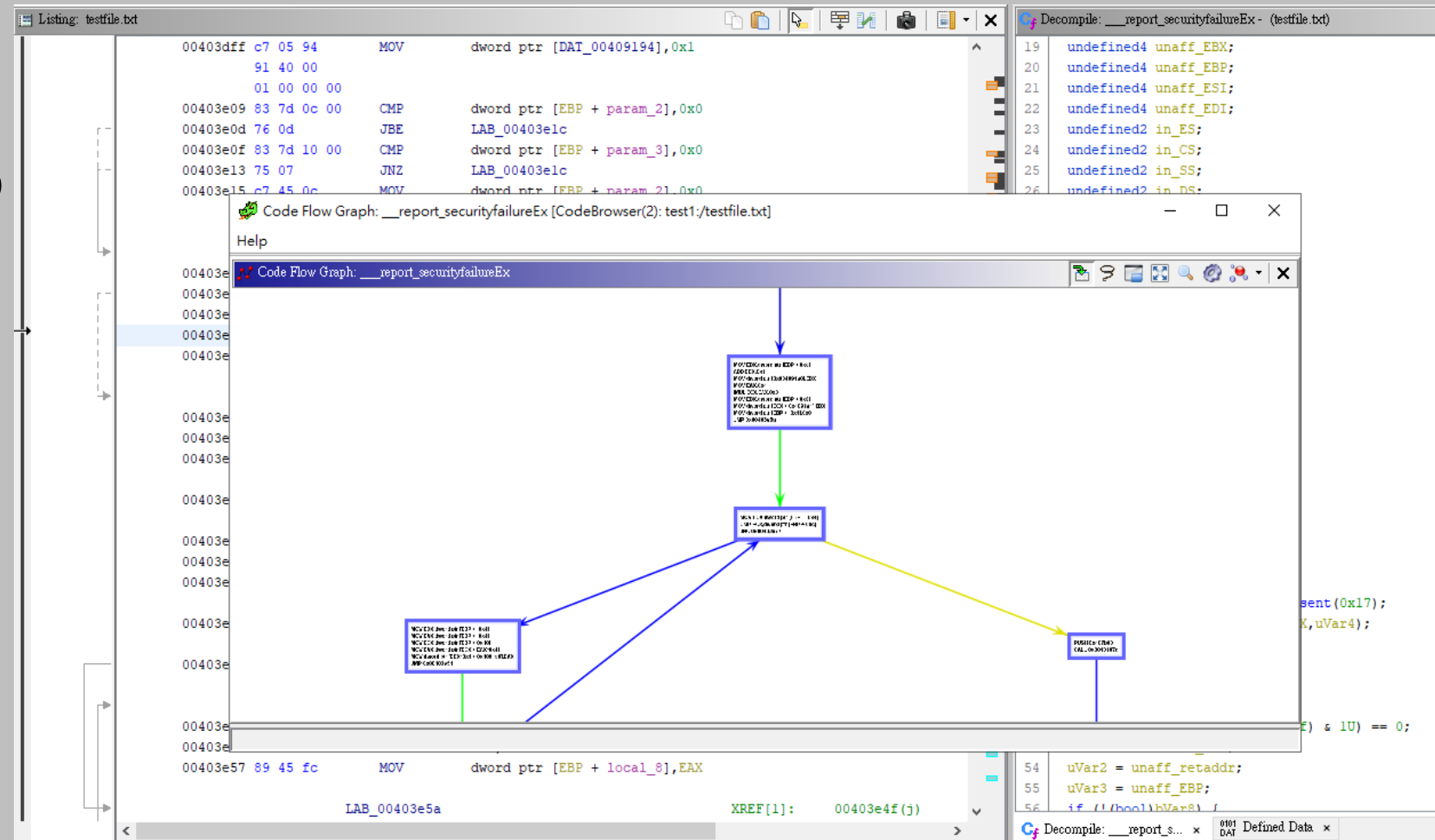


Code browser

- After the analyze is done, you can see the binary disassembly listing and other windows.
- The “MZ” indicate the binary file is a DOS executable format.



- [Graph]/[Block Flow] to see flow graph of current function.

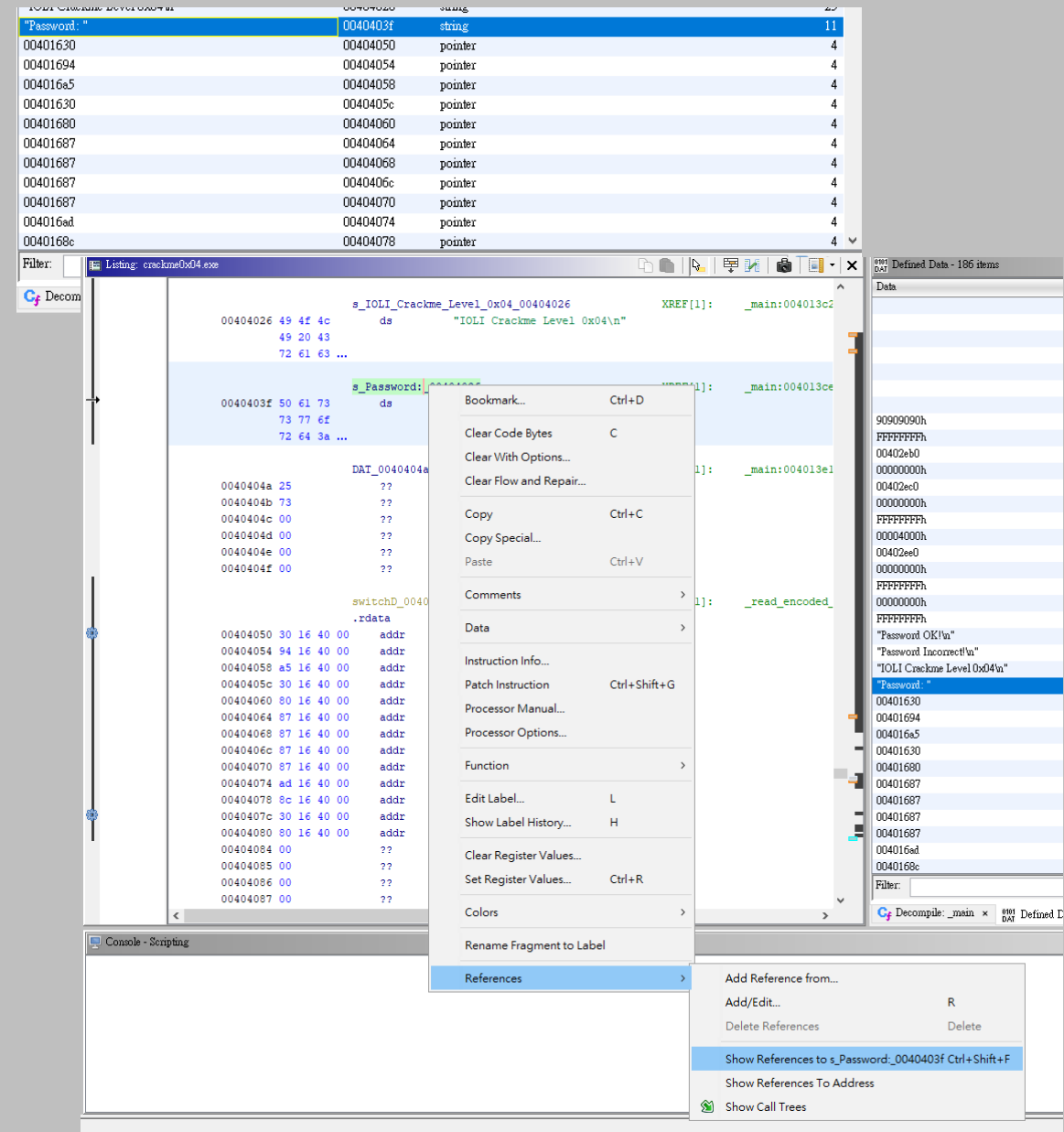


Example

- Here's an example on cracking IOLI crackme0x04
- <https://github.com/Maijin/radare2-workshop-2015/tree/master/IOLI-crackme>

Example

- On Defined Data window, we found a string "Password: " which is interesting, double click to find the location of the string.
- Right click/[References]/[Show References]



Example

- We can find the instruction that read the string, and the Decompiler window showing the function in C.
- The program reads string input by scanf(), then call _check().
- Double click _ckeck() function, let's see what it does.

The screenshot displays a debugger window with two panes. The left pane shows assembly instructions with their addresses, hex values, and mnemonics. The right pane shows the decompiled C code for the function `__cdecl _main`.

Assembly View (Left Pane):

Address	Hex	Mnemonic	Comment
00401398	81 ec 98	SUB	ESP, 0x98
0040139e	83 e4 f0	AND	ESP, 0xffffffff0
004013a1	b8 00 00	MOV	EAX, 0x0
004013a6	83 c0 0f	ADD	EAX, 0xf
004013a9	83 c0 0f	ADD	EAX, 0xf
004013ac	c1 e8 04	SHR	EAX, 0x4
004013af	c1 e0 04	SHL	EAX, 0x4
004013b2	89 45 84	MOV	dword ptr [EBP + local_80], EAX
004013b5	8b 45 84	MOV	EAX, dword ptr [EBP + local_80]
004013b8	e8 13 19	CALL	__alloca
004013bd	e8 0e 01	CALL	__main
004013c2	c7 04 24	MOV	dword ptr [ESP] => local_a0, s_IOLI_Crackme_Level... = "IOLI Cra
004013c9	e8 c2 19	CALL	_printf
004013ce	c7 04 24	MOV	dword ptr [ESP] => local_a0, s_Password: 0040403f = "Password
004013d5	e8 b6 19	CALL	_printf
004013da	8d 45 88	LEA	EAX => local_7c, [EBP + -0x78]
004013dd	89 44 24 04	MOV	dword ptr [ESP + local_9c], EAX
004013e1	c7 04 24	MOV	dword ptr [ESP] => local_a0, DAT_0040404a = 25h %
004013e8	e8 83 19	CALL	_scanf
004013ed	8d 45 88	LEA	EAX => local_7c, [EBP + -0x78]
004013f0	89 04 24	MOV	dword ptr [ESP] => local_a0, EAX
004013f3	e8 18 ff	CALL	_check

Decompiled C Code View (Right Pane):

```
1 |  
2 | int __cdecl _main(int _Argc, char **_Argv, char **_Env)  
3 |  
4 | {  
5 |     size_t in_stack_ffffff60;  
6 |     char local_7c [120];  
7 |  
8 |     __alloca(in_stack_ffffff60);  
9 |     __main();  
10 |    _printf("IOLI Crackme Level 0x04\n");  
11 |    _printf("Password: ");  
12 |    _scanf("%s", local_7c);  
13 |    _check(local_7c);  
14 |    return 0;  
15 | }  
16 |
```

_check()

- in each loop, strlen() function get the length of our input, and call sscanf() function to read each character, convert it to decimal and sum it together. If the sum equals to 0xf(15 in decimal), it break the while loop and print the victory message.

```
void __cdecl _check(char *param_1)
{
    size_t sVar1;
    char local_11;
    uint local_10;
    int local_c;
    int local_8;

    local_c = 0;
    local_10 = 0;
    while( true ) {
        sVar1 = _strlen(param_1);
        if (sVar1 <= local_10) {
            _printf("Password Incorrect!\n");
            return;
        }
        local_11 = param_1[local_10];
        _sscanf(&local_11, "%d", &local_8);
        local_c = local_c + local_8;
        if (local_c == 0xf) break;
        local_10 = local_10 + 1;
    }
    _printf("Password OK!\n");
    /* WARNING: Subroutine does not return */
    _exit(0);
}
```

Example

- Simply input strings which the sum of first N bit is 15 to pass the test.

```
D:\>crackme0x04.exe
IOLI Crackme Level 0x04
Password: 96
Password OK!

D:\>crackme0x04.exe
IOLI Crackme Level 0x04
Password: 44435
Password OK!
```

Part2. OLLVM

- Ollvm wiki: <https://github.com/obfuscator-llvm/obfuscator/wiki>

Install OLLVM

- Run the following command

```
$ git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git  
$ mkdir build  
$ cd build  
$ cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_INCLUDE_TESTS=OFF ../obfuscator/  
$ make
```

Usage

- There are three options you can use while compiling your code.
 - -mllvm -sub for instructions substitution
 - -mllvm -bcf for bogus control flow
 - -mllvm -fla for control flow flattening
- Just pass the flag to compiler like:
\$./build/bin/clang source.c -o test -mllvm -sub -mllvm -fla

Example

- Now let's take a look at how “-sub” flag affect our code.
- Take this simple program for example

```
1  #include <iostream>
2
3  using namespace std;
4
5  int foo(int cnt)
6  {
7      return cnt + 1;
8  }
9
10 int main()
11 {
12     int cnt = 0;
13     while(cnt < 100)
14     {
15         cnt += 1;
16         cnt = foo(cnt);
17     }
18
19     return 0;
20 }
```

Decompile

- Here's the decompile result, it seem that there's nothing wrong with it, let's take a closer look at the assembly code.

main				XREF[5]:	Entry Point (*), _start:0040058d(*), _start:0040058d(*), 00400868(*)
00400680	55	PUSH	RBP		
00400681	48 89 e5	MOV	RBP,RSP		
00400684	48 83 ec 10	SUB	RSP,0x10		
00400688	c7 45 fc	MOV	dword ptr [RBP + local_c],0x0		
	00 00 00 00				
0040068f	c7 45 f8	MOV	dword ptr [RBP + local_10],0x0		
	00 00 00 00				
LAB_00400696				XREF[1]:	004006b8(j)
00400696	83 7d f8 64	CMP	dword ptr [RBP + local_10],0x64		
0040069a	0f 8d 1d	JGE	LAB_004006bd		
	00 00 00				
004006a0	31 c0	XOR	EAX,EAX		
004006a2	8b 4d f8	MOV	ECX,dword ptr [RBP + local_10]		
004006a5	83 e8 01	SUB	EAX,0x1		
004006a8	29 c1	SUB	ECX,EAX		
004006aa	89 4d f8	MOV	dword ptr [RBP + local_10],ECX		
004006ad	8b 7d f8	MOV	EDI,dword ptr [RBP + local_10]		
004006b0	e8 ab ff	CALL	foo		undefined foo(i
	ff ff				
004006b5	89 45 f8	MOV	dword ptr [RBP + local_10],EAX		
004006b8	e9 d9 ff	JMP	LAB_00400696		
	ff ff				
LAB_004006bd				XREF[1]:	0040069a(j)
004006bd	31 c0	XOR	EAX,EAX		
004006bf	48 83 c4 10	ADD	RSP,0x10		
004006c3	5d	POP	RBP		
004006c4	c3	RET			

```
1
2 undefined8 main(void)
3
4 {
5     int local_10;
6
7     local_10 = 0;
8     while (local_10 < 100) {
9         local_10 = foo(local_10 + 1);
10    }
11    return 0;
12 }
13
```

Without -sub flag

vs

With -sub flag

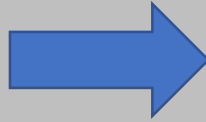
main		XREF[5]:	Entry Point(
		_start:00400	_start:00400
		00400868 (*)	
00400680	55	PUSH	RBP
00400681	48 89 e5	MOV	RBP,RSP
00400684	48 83 ec 10	SUB	RSP,0x10
00400688	c7 45 fc	MOV	dword ptr [RBP + local_c],0x0
	00 00 00 00		
0040068f	c7 45 f8	MOV	dword ptr [RBP + local_10],0x0
	00 00 00 00		
LAB_00400696		XREF[1]:	004006b4(j)
00400696	83 7d f8 64	CMP	dword ptr [RBP + local_10],0x64
0040069a	0f 8d 19	JGE	LAB_004006b9
	00 00 00		
004006a0	8b 45 f8	MOV	EAX,dword ptr [RBP + local_10]
004006a3	83 c0 01	ADD	EAX,0x1
004006a6	89 45 f8	MOV	dword ptr [RBP + local_10],EAX
004006a9	8b 7d f8	MOV	EDI,dword ptr [RBP + local_10]
004006ac	e8 af ff	CALL	foo
	ff ff		undefined
004006b1	89 45 f8	MOV	dword ptr [RBP + local_10],EAX
004006b4	e9 dd ff	JMP	LAB_00400696
	ff ff		
LAB_004006b9		XREF[1]:	0040069a(j)
004006b9	31 c0	XOR	EAX,EAX
004006bb	48 83 c4 10	ADD	RSP,0x10
004006bf	5d	POP	RBP
004006c0	c3	RET	

main		XREF[5]:	Entry Point(
		_start:00400	_start:00400
		00400868 (*)	
00400680	55	PUSH	RBP
00400681	48 89 e5	MOV	RBP,RSP
00400684	48 83 ec 10	SUB	RSP,0x10
00400688	c7 45 fc	MOV	dword ptr [RBP + local_c],0x0
	00 00 00 00		
0040068f	c7 45 f8	MOV	dword ptr [RBP + local_10],0x0
	00 00 00 00		
LAB_00400696		XREF[1]:	004006b8(j)
00400696	83 7d f8 64	CMP	dword ptr [RBP + local_10],0x64
0040069a	0f 8d 1d	JGE	LAB_004006bd
	00 00 00		
004006a0	31 c0	XOR	EAX,EAX
004006a2	8b 4d f8	MOV	ECX,dword ptr [RBP + local_10]
004006a5	83 e8 01	SUB	EAX,0x1
004006a8	29 c1	SUB	ECX,EAX
004006aa	89 4d f8	MOV	dword ptr [RBP + local_10],ECX
004006ad	8b 7d f8	MOV	EDI,dword ptr [RBP + local_10]
004006b0	e8 ab ff	CALL	foo
	ff ff		undefined
004006b5	89 45 f8	MOV	dword ptr [RBP + local_10],EAX
004006b8	e9 d9 ff	JMP	LAB_00400696
	ff ff		
LAB_004006bd		XREF[1]:	0040069a(j)
004006bd	31 c0	XOR	EAX,EAX
004006bf	48 83 c4 10	ADD	RSP,0x10
004006c3	5d	POP	RBP
004006c4	c3	RET	

Example

- We can observe that there's a slightly difference between two assembly code, the add instruction became two sub instruction.
 $a = b - (-c)$

```
EAX = local_10  
EAX = EAX + 1  
local_10 = EAX
```



```
EAX = 0  
ECX = local_10  
EAX = EAX - 1  
ECX = ECX - EAX  
Local_10 = ECX
```

- Although we can't see the difference after decompile, but the obfuscator do did its job.