Worcester Polytechnic Institute
Department of Computer Science

Lab 16: Reverse Engineering with IDA

Objectives

- IDA code analysis with ground truth. Student will write a small C program with a few functionalities and obfuscations to create their own executables. They will then analyze the assembly code to understand how their programs are structured in assembly language.
- IDA code analysis without ground truth. Students will be given another executable (not the one they created in Module 1) to practice what they have learned and reverse engineer the executable.

Task

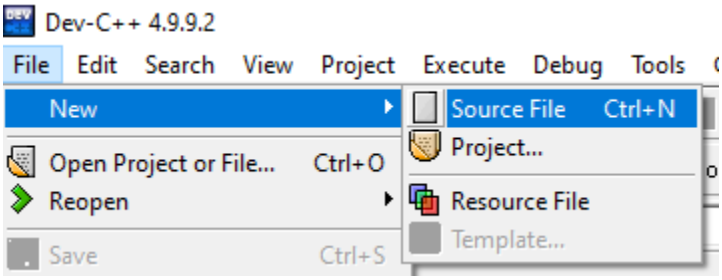
Download IDA Freeware
https://hex-rays.com/ida-free/

Task 1: Create a program using C/C++ (C is preferred)

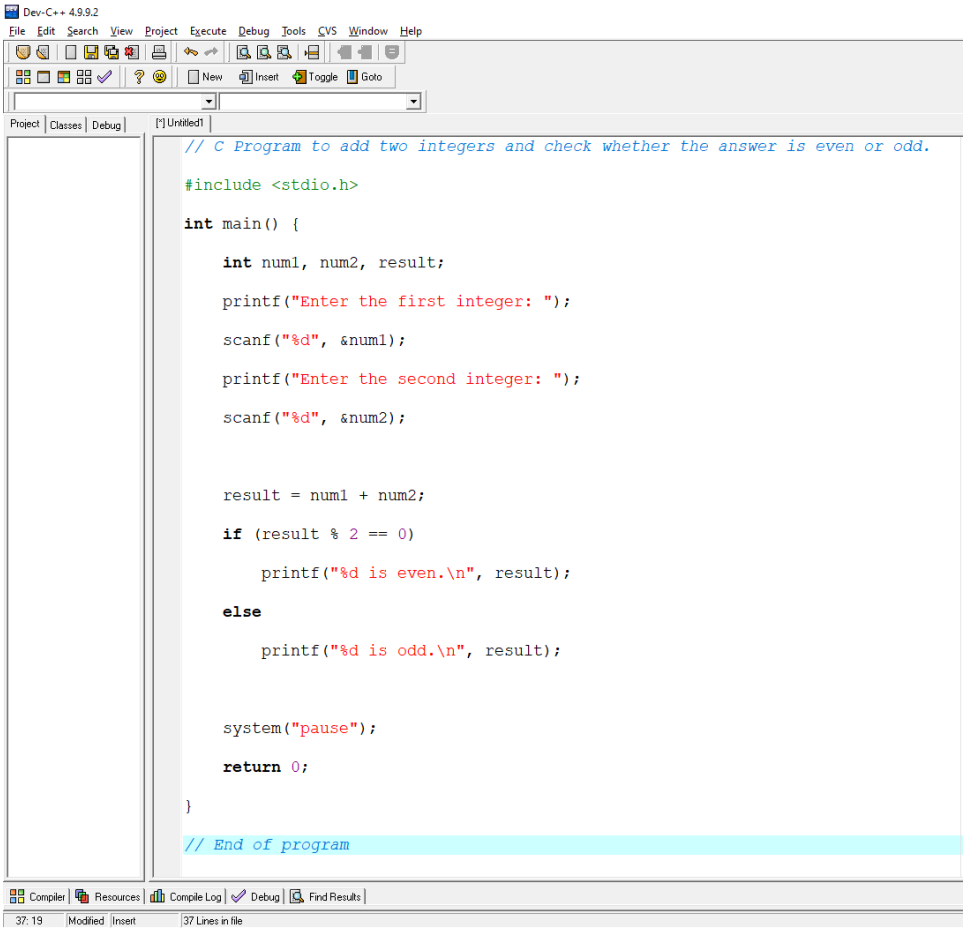
In any IDE, create a simple C program as follow:

Dev C++:
https://www.bloodshed.net/

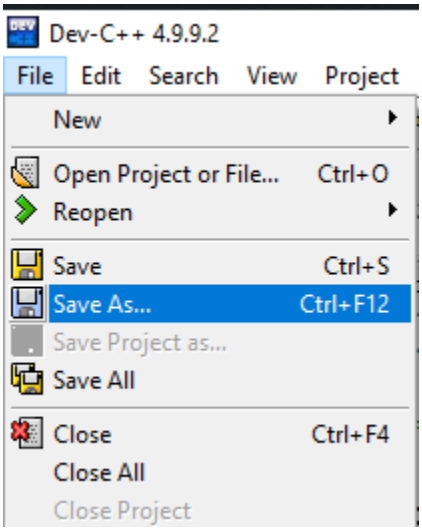
1. File -> New -> Source File



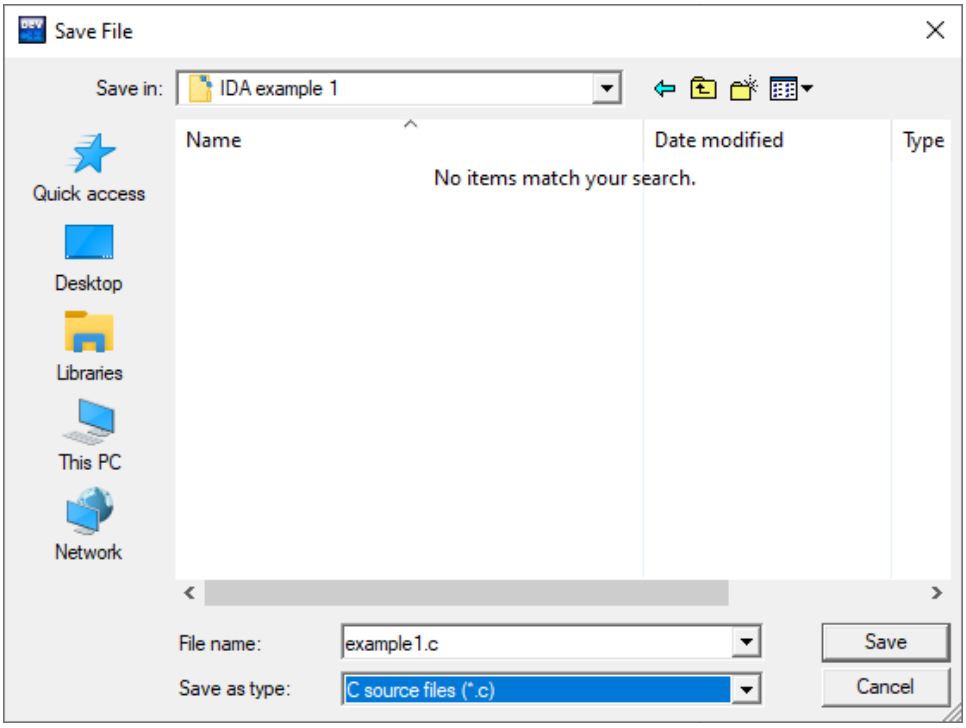
2. Copy the code below to the editor



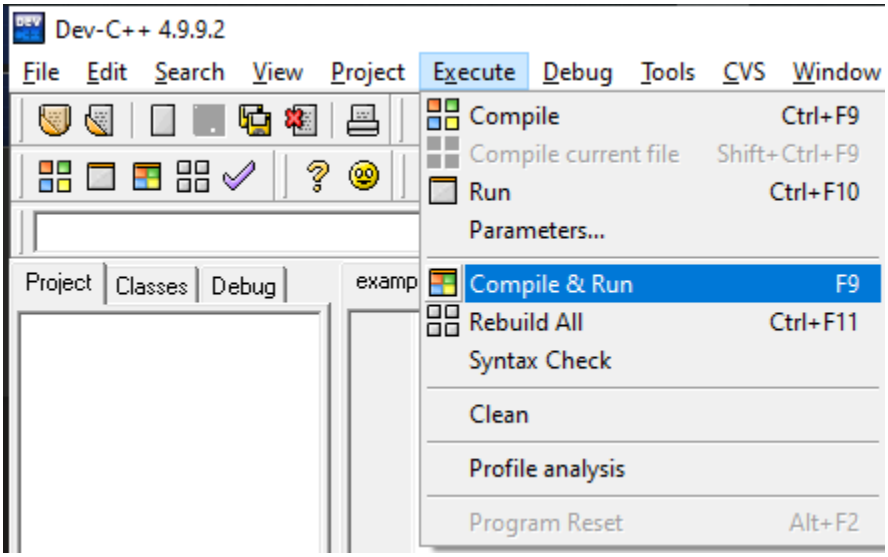
3. File -> Save As...



4. Select the a location, rename file to example1.c and change the type to C source files (*.c)



5. To compile or test the program. Select Execute -> Compile & Run (This will generate the .exe under the location you save with the .c file)

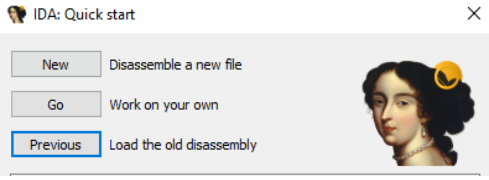


Code:

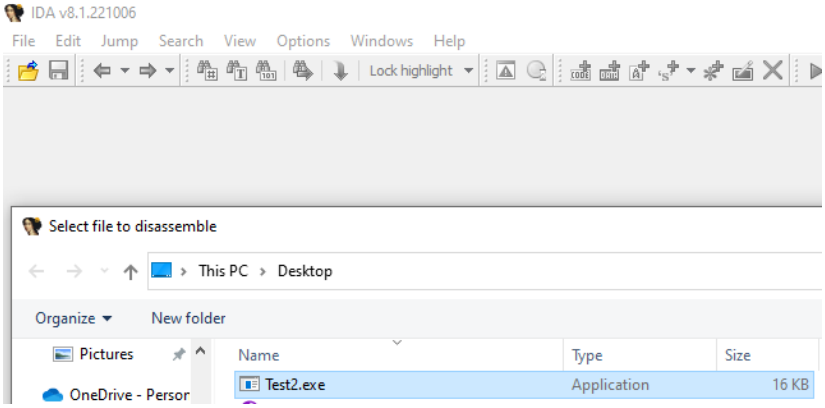
```
// C Program to add two integers and check whether the answer is even or odd.
#include <stdio.h>
int main() {
    int num1, num2, result;
    printf("Enter the first integer: ");
    scanf("%d", &num1);
    printf("Enter the second integer: ");
    scanf("%d", &num2);

    result = num1 + num2;
    if (result % 2 == 0)
        printf("%d is even.\n", result);
    else
        printf("%d is odd.\n", result);

    system("pause");
    return 0;
}
// End of program
```

Task 2: Inspect the program in IDA

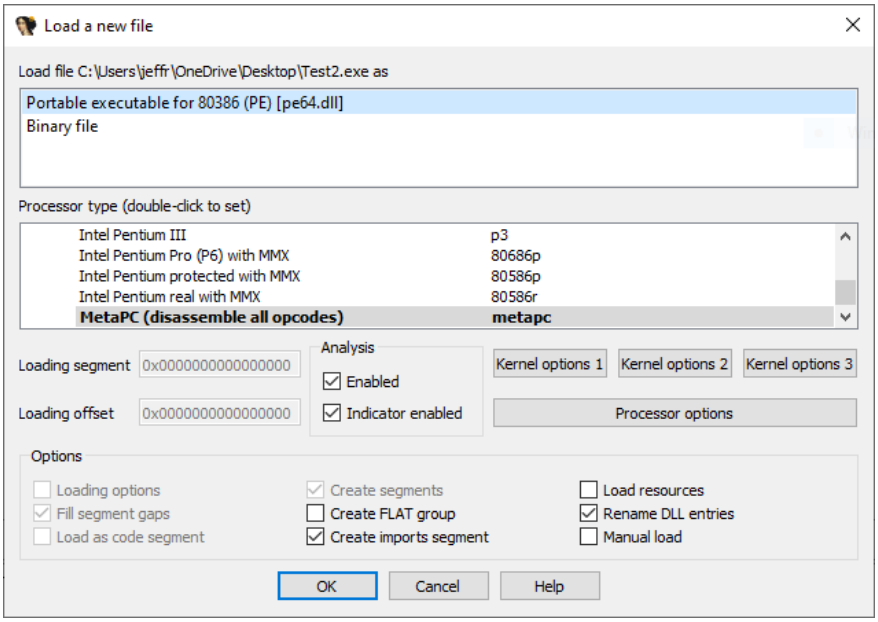
Open IDA, select “New”



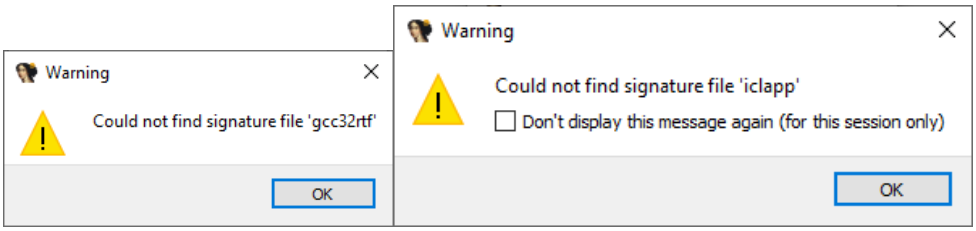
Select the C program .exe application that we created.



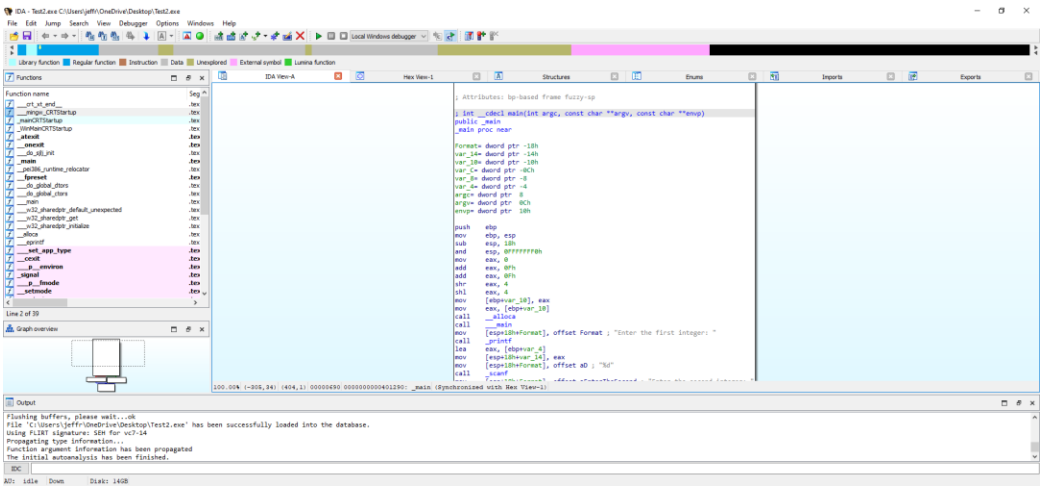
Leave the default setting, press “OK”



Ignore the warning



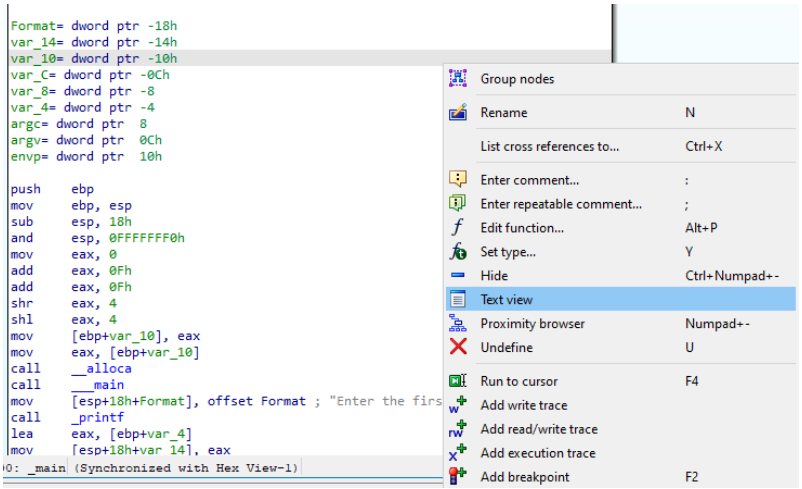
IDA will automatically select the main function.



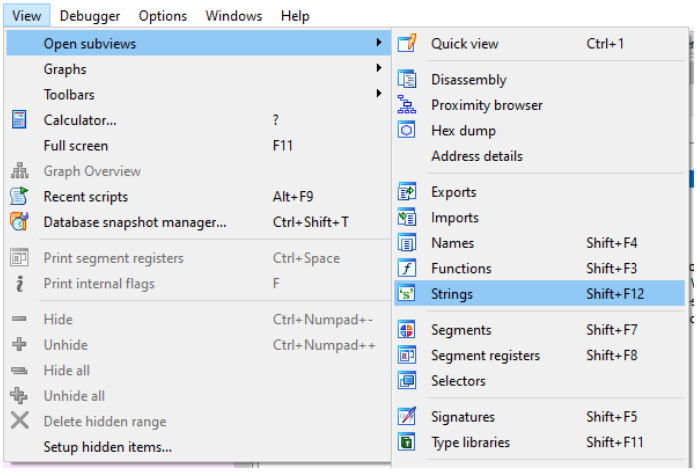
(In some simple programs, IDA can locate the main function, but in most of the real world programs, IDA will not be able to locate the main function.)

IDA basic usage:

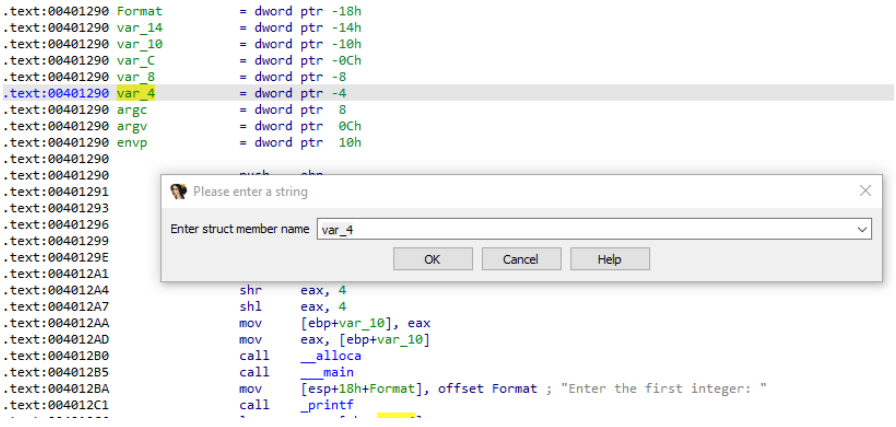
Right click to switch between text view and graph view.



Select “View” → “Open subviews” → “Strings” to open Strings window



Press “N” to rename variables.



(In this example, we can rename var_4 as num1, var_8 as num2 and var_C as result)

Analyze the code:

Here are two link that can help you understand more about x86-assembly:
https://www.aldeid.com/wiki/Category:Architecture/x86-assembly

https://www.aldeid.com/wiki/X86-assembly/Instructions

Set up the base stack
Move the base pointer
Every C program would initialize this part automatically

```
:00401290                push    ebp
:00401291                mov     ebp, esp
:00401293                sub     esp, 18h
:00401296                and     esp, 0FFFFFFF0h
:00401299                mov     eax, 0
:0040129E                add     eax, 0Fh
:004012A1                add     eax, 0Fh
:004012A4                shr     eax, 4
:004012A7                shl     eax, 4
:004012AA                mov     [ebp+var_10], eax
:004012AD                mov     eax, [ebp+var_10]
:004012B0                call    __alloca
:004012B5                call    ___main
```

Get user input

```
:004012BA                mov     [esp+18h+Format], offset Format ; "Enter the first integer: "
:004012C1                call    _printf
:004012C6                lea     eax, [ebp+num1]
:004012C9                mov     [esp+18h+var_14], eax
:004012CD                mov     [esp+18h+Format], offset aD ; "%d"
:004012D4                call    _scanf
:004012D9                mov     [esp+18h+Format], offset aEnterTheSecond ; "Enter the second integer: "
:004012E0                call    _printf
:004012E5                lea     eax, [ebp+num2]
:004012E8                mov     [esp+18h+var_14], eax
:004012EC                mov     [esp+18h+Format], offset aD ; "%d"
:004012F3                call    _scanf
```

Get the sum of num1 and num2 and move to result

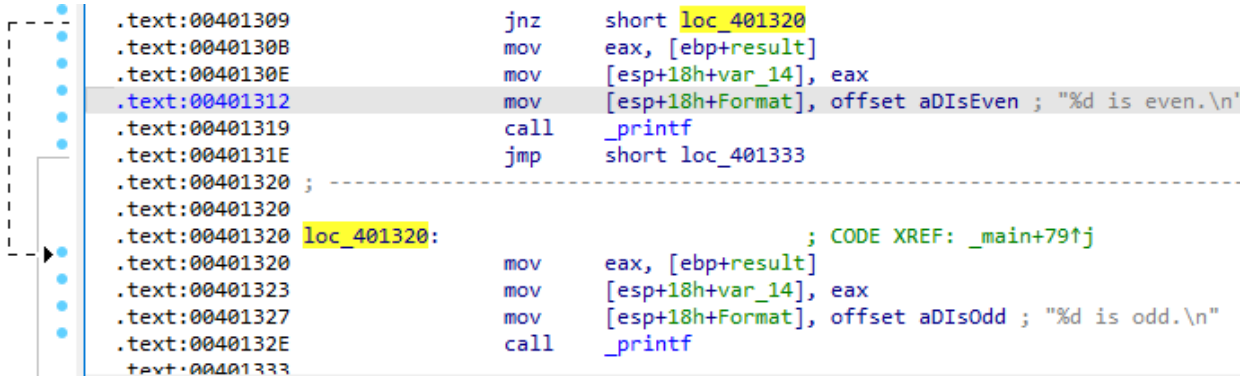
```
:004012F8                mov     eax, [ebp+num2]
:004012FB                add     eax, [ebp+num1]
:004012FE                mov     [ebp+result], eax
:00401301                mov     eax, [ebp+result]
```

The *and* instruction performs a logical AND operation
The *test* instruction is identical to the *and* instruction except it does not affect operands.
For example, *and* eax=0011b(3 in dec) 0001b, after the *and* operation, result would be 0001
test eax, eax will test if eax = 0

```
:00401304                and     eax, 1
:00401307                test    eax, eax
:00401309                jnz     short loc_401320
```

The *jnz* instruction is a conditional jump that follows a test.
It jumps to the specified location if the Zero Flag (ZF) is cleared (0).

jnz is commonly used to explicitly test for something not being equal to zero
so in our example, jump if eax != 0, so it will take the jump and print 3 is odd.



(To understand more jump instruction, visit https://www.aldeid.com/wiki/X86-assembly/Instructions)

Task 3: Examine an unknown program.

Download “unknown4.exe” and examine the program code.
Try to crack the password phase.

Question:

What is the password from unknown4 program? After examining the program from IDA, run the program again and enter the correct password and then take a screenshot.