Understanding Instruction Set Architectures

Reverse Engineering – Understanding Instruction Set Architectures

Albert López - newlog@overflowedminds.net

Goals

What can you expect after this module?

Goals

1. Understand what software does without their the source code

1. Understand assembler for x86 and x64 architectures

1. Learn how to use reverse engineering tools

1. Understand different compiler disassemblies

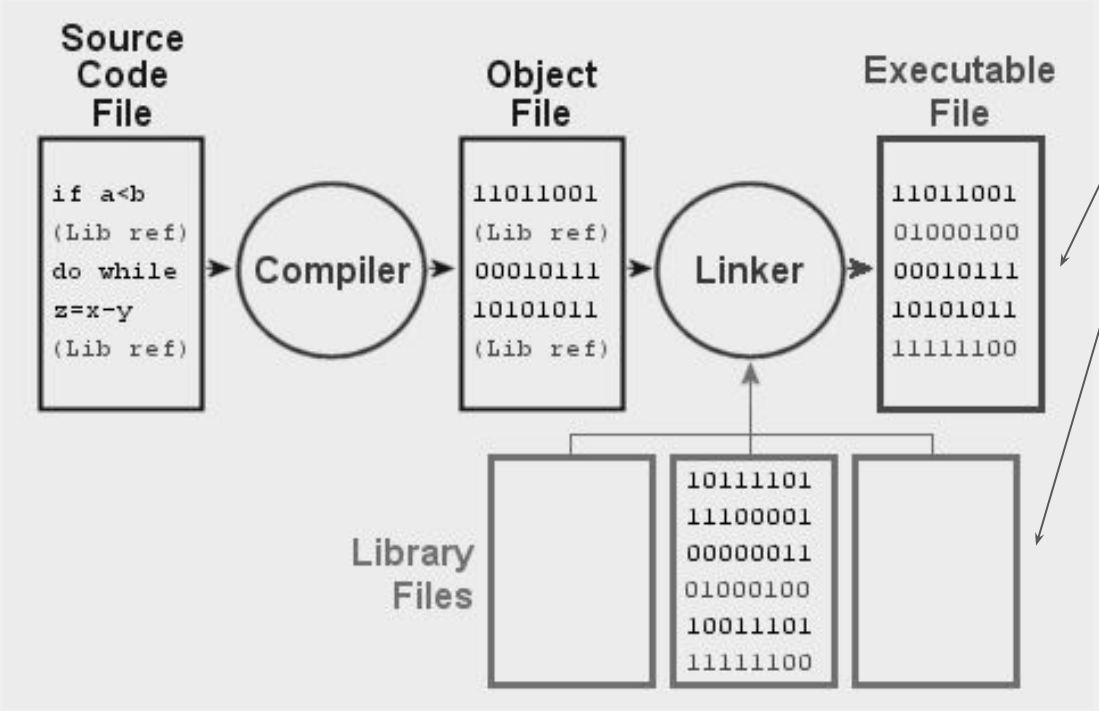
1. Understand different compiler optimizations

The Basics

Our goal is to understand what a binary does:

- Without having the source code
- Without executing it

- From **source code** to **executable**?



Processors, Machine Code & Assembly Language

An executable contains the instructions that the processor will execute.
Do you remember the **.text** section?

Like everything in our world, these instructions are not anything but 0s and 1s (machine code).

The processor has no trouble understanding 0s and 1s, but humans do.

That's why we created the **Assembly/Assembler Language**.

Assembler is just a translation of those 0s and 1s into something more readable.

Compiling & Decompiling

At a high level:

- Compiling means converting source code into machine code.
- Decompiling means converting machine code into source code.

It is worth to point out that going from source code to machine code, and going from machine code to source code is not an idempotent operation.

Decompilers convert machine code into pseudocode so things are easier to understand for a human being. However, this process is not perfect.

Some decompilers:

- Hex-Rays Decompiler
- snowman (open source)
- RetDec (open sourced in 2017 by Avast)

x86 & x64

Workflow

We could use services such as the Compiler Explorer and analyze the assembler from there…

However, in order to get used to static analysis tools and understand how they disassemble binaries we will be working with binaries compiled by different compilers and we will analyze them with different tools.

Compilers:

- GCC [1]
- Visual C++ [2]

Analysis tools:

- IDA Pro (demo)
- radare2 (from git!)

[1] GCC version: gcc (Ubuntu 7.2.0-8ubuntu3) 7.2.0
[2] Visual Studio Express 2017

General Purpose Registers, Assignments

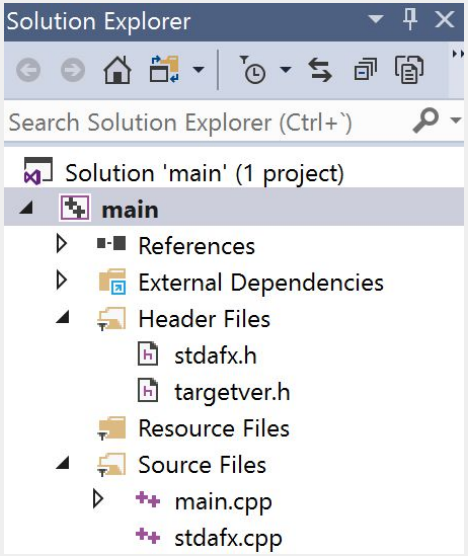
Source Code

- Compile with GCC (in x64 OS):
  - $ gcc main.c -o main_x64
  - $ gcc -m32 main.c -o main_x86

```
int func(void) {
        int a = 2;
        return a;
}

int main(int argc, char **argv) {
        int a = func();
        return a;
}
```

- Compile with Visual C++
  - F7 with proper config


```
#include "stdafx.h"

int func(void) {
        printf("marker!");
        int a = 2;
        return a;
}

int main(int argc, char **argv) {
        int a = func();
        return a;
}
```



[1] If -m32 option does not work: $ sudo apt-get install gcc-multilib

Roadmap

1. radare2, GCC, main_x86
   a. Registers explanation
   b. mov explanation

1. radare2, Visual C++, main.exe (x86) → Debug

1. radare2, Visual C++, main.exe (x86) → Release
   b. Locating main function
   c. Signatures
   d. mov explanation
   e. Generated code comparison

1. radare2, GCC, main_x86 → Stripping symbols

1. IDA Pro, GCC, main_x86

1. IDA Pro, Visual C++, main.exe (x86)

radare2, GCC,
main_x86

radare2, GCC, main_x86

Steps:

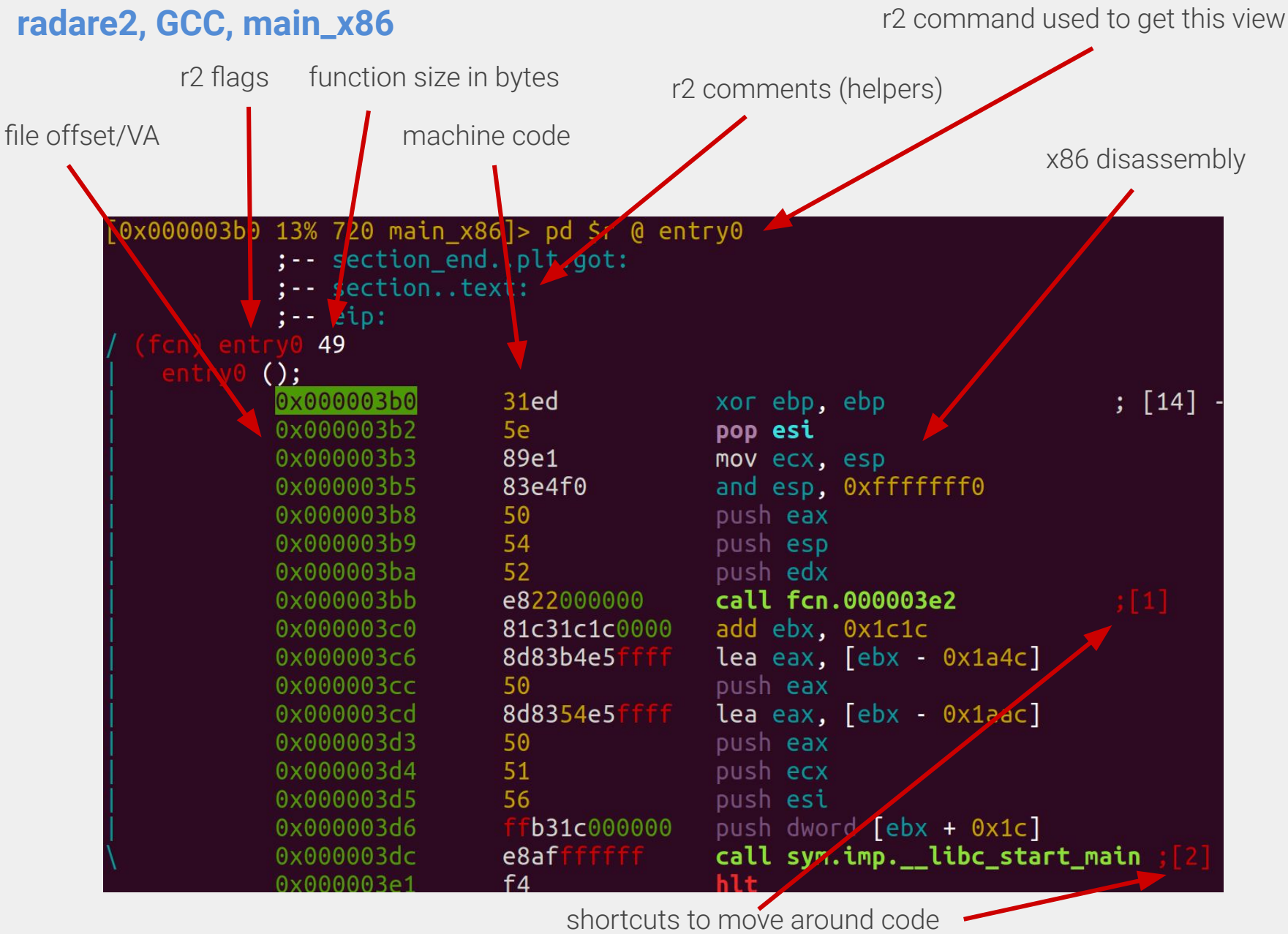
1. Open file:
   a. $ r2 <binary_path>

1. Analyze binary:
   b. > aaa

1. Go to Visual Mode:
   b. > V

1. Go to disassembly view
   b. > p

radare2, GCC, main_x86



radare2, GCC, main_x86

On startup, r2 leaves us in the binary entry point (remember our class on Binary File Formats).

As you have guessed, that's not our **main()** function. Before getting to our main, many things happen:

- What happens before main?

With radare2, most of the times, we can move to our main function with:

- > s main

Which means, "seek to 'main' symbol"

radare2, GCC, main_x86

```
[0x00000507 17% 225 main_x86]> pd $r @ main
            ;-- main:
/ (fcn) sym.main 25
|   sym.main ();
|           0x00000507      55              push ebp
|           0x00000508      89e5            mov ebp, esp
|           0x0000050a      e811000000      call sym.__x86.get_pc_thunk.ax ;[1]
|           0x0000050f      05cd1a0000      add eax, 0x1acd
|           0x00000514      e8d4ffffff      call sym.func                  ;[2]
|           0x00000519      b800000000      mov eax, 0
|           0x0000051e      5d              pop ebp
\           0x0000051f      c3              ret
```

This makes more sense now!

Spoiler alert:

- "**call**" assembler instruction is used to call functions!

Given that our code was compiled with debug symbols ("Binary File Formats" class!), radare2 knows that we called our function "**func**".

radare2, GCC, main_x86

```
[0x00000507 17% 225 main_x86]> pd $r @ main
            ;-- main:
/ (fcn) sym.main 25
|   sym.main ();
|           0x00000507      55              push ebp
|           0x00000508      89e5            mov ebp, esp
|           0x0000050a      e811000000      call sym.__x86.get_pc_thunk.ax ;[1]
|           0x0000050f      05cd1a0000      add eax, 0x1acd
|           0x00000514      e8d4ffffff      call sym.func                  ;[2]
|           0x00000519      b800000000      mov eax, 0
|           0x0000051e      5d              pop ebp
\           0x0000051f      c3              ret
```

- However, what's this call to **sym.__x86.get_pc_thunk.ax**?

This function is added by the compiler (GCC) in order to generate PIC (Position Independent Code).

PIC code (might) generate smaller binaries because it contains less "fixups" (Binary File Formats class!) and decreases binary startup time for the same reason.

radare2, GCC, main_x86

If we click “1”, we will see the sym.__x86.get_pc_thunk.ax function.

```
[0x00000520 18% 225 main_x86]> pd $r @ sym.__x86.get_pc_thunk.ax
/ (fcn) sym.__x86.get_pc_thunk.ax 4
|   sym.__x86.get_pc_thunk.ax ();
|              ; CALL XREF from 0x000004f3 (sym.func)
|              ; CALL XREF from 0x0000050a (sym.main)
|          0x00000520      8b0424          mov eax, dword [esp]
\          0x00000523      c3              ret
```

This is black magic for you now, but at the end of this module you will understand. I promise!

This code retrieves the location (address) of the assembler instructions **in memory** and stores it in the eax register [1].

Write “u” to move back to the previous radare2 location (main function).

[1] *esp = @ret address

radare2, GCC, main_x86

If we click “2”, we will see the sym.func function.

```
/ (fcn) sym.func 28
|   sym.func ();
|           ; var int local_4h @ ebp-0x4
|              ; CALL XREF from 0x00000519 (sym.main)
|           0x000004ed      55              push ebp
|           0x000004ee      89e5            mov ebp, esp
|           0x000004f0      83ec10          sub esp, 0x10
|           0x000004f3      e82e000000      call sym.__x86.get_pc_thunk.ax ;[1]
|           0x000004f8      05e41a0000      add eax, 0x1ae4
|           0x000004fd      c745fc020000.   mov dword [local_4h], 2
|           0x00000504      8b45fc          mov eax, dword [local_4h]
|           0x00000507      c9              leave
\           0x00000508      c3              ret
```

Forget about the first 5 instructions. We only care about the instruction at 0x4fd:

- mov dword [local_4h], 2

This instruction is moving the value 2 and storing it in the memory address pointed by ebp-4.

radare2, GCC, main_x86

- mov dword [local_4h], 2

local_4h is a symbol that r2 defines to ease our comprehension of assembler.

r2 is helping us understand that 2 is stored in a **function local variable**.

The address where this local variable will stores its data is dynamic! And it is defined by the value of the **EBP register** minus 4 bytes...

radare2, GCC, main_x86

- mov dword [local_4h], 2

local_4h is a symbol that r2 defines to ease our comprehension of assembler.

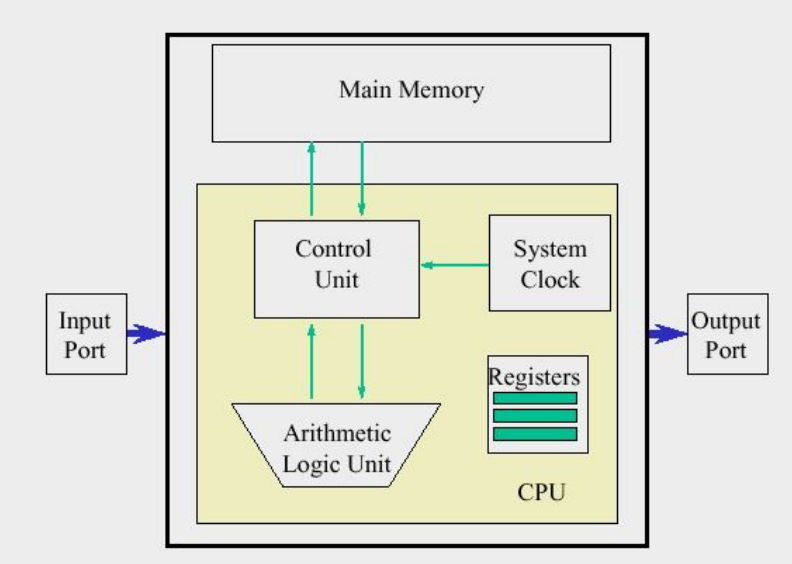
r2 is helping us understand that 2 is stored in a **function local variable**.

The address where this local variable will stores its data is dynamic! And it is defined by the value of the **EBP register** minus 4 bytes...

Registers?? What are registers??

radare2, GCC, main_x86 → Registers

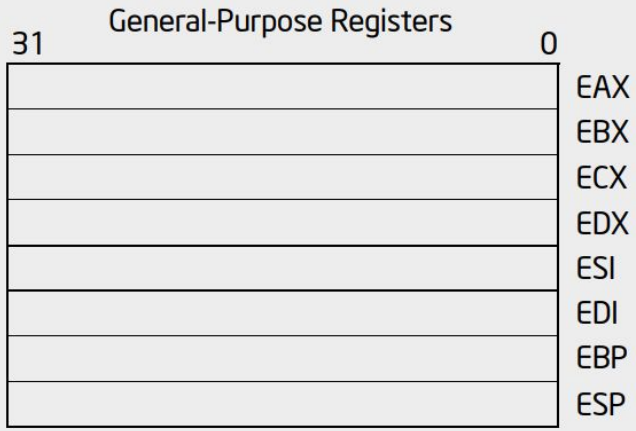
Registers are fast access chips that the processor provides to increase speed on data access and write.



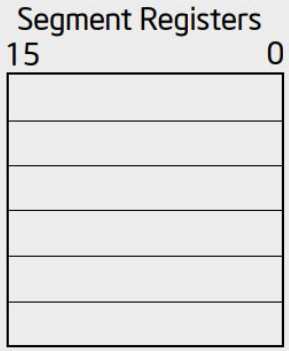
Writing and reading from a register is faster than doing it from RAM memory which is faster than doing it from the hard disk.

Given that the registers are hardware chips, the number of registers is limited.

radare2, GCC, main_x86 → Registers

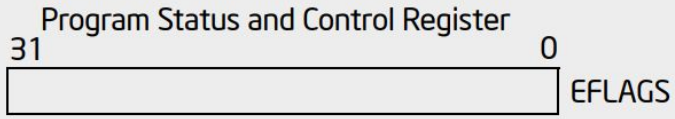


EAX — Accumulator for operands and results data
EBX — Pointer to data in the DS segment
ECX — Counter for string and loop operations
EDX — I/O pointer
ESI — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
EDI — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
EBP — Pointer to data on the stack (in the SS segment)
ESP — Stack pointer (in the SS segment)



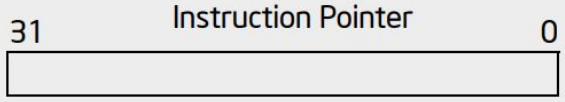
Registers that will be used as Segment Selectors. Given that segmentation is barely used by OS, CS, DS, SS and ES are usually set to 0.

CS — Code segment register
DS — Data segment register
SS — Stack segment register
ES — Extra data segment register
FS — Extra data segment register
GS — Extra data segment register



FS (GS in x64) segment register is used to point to the Thread Information/Environment Block (TIB/TEB) of a process.

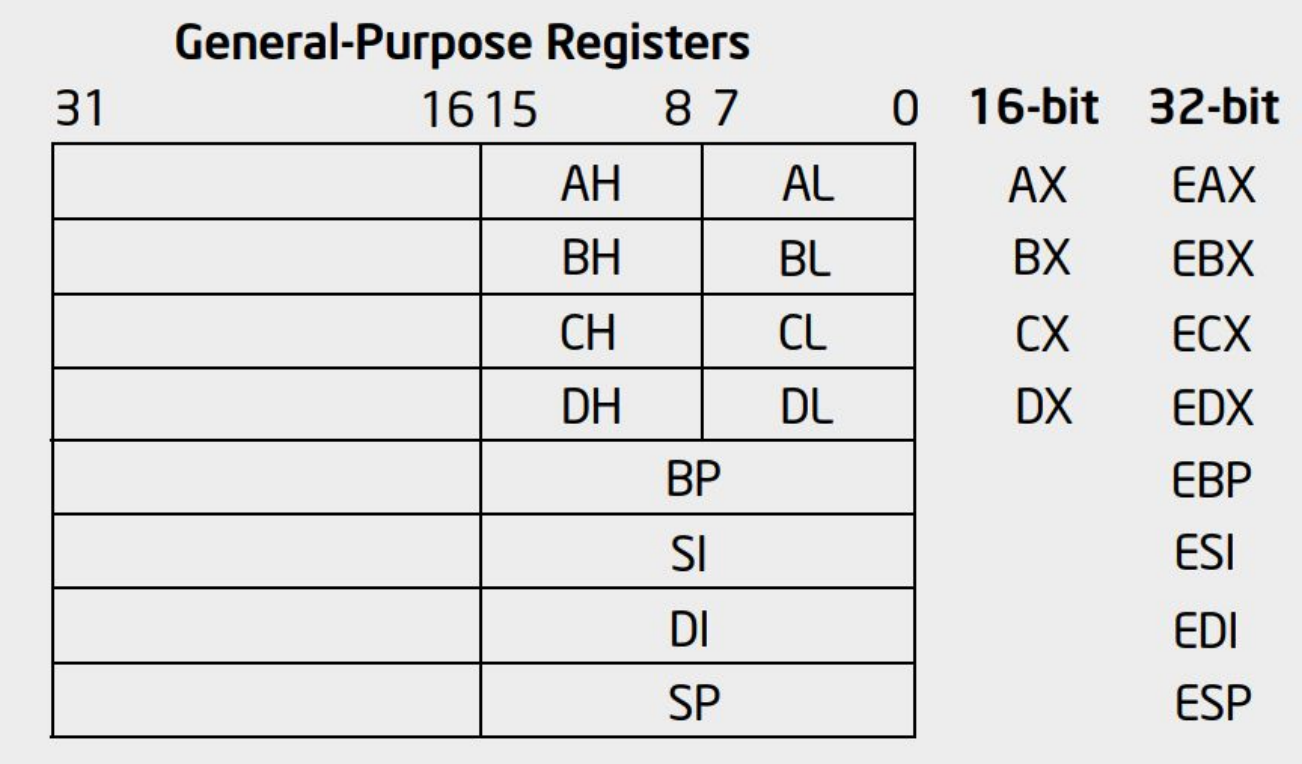
EFLAGS — Flags indicating multiple things such as arithmetic instruction information



EIP — Contains the address of the instruction to be executed

radare2, GCC, main_x86

Registers can also be accessed by parts:



radare2, VisualC++, main.exe (x86)

DEBUG MODE

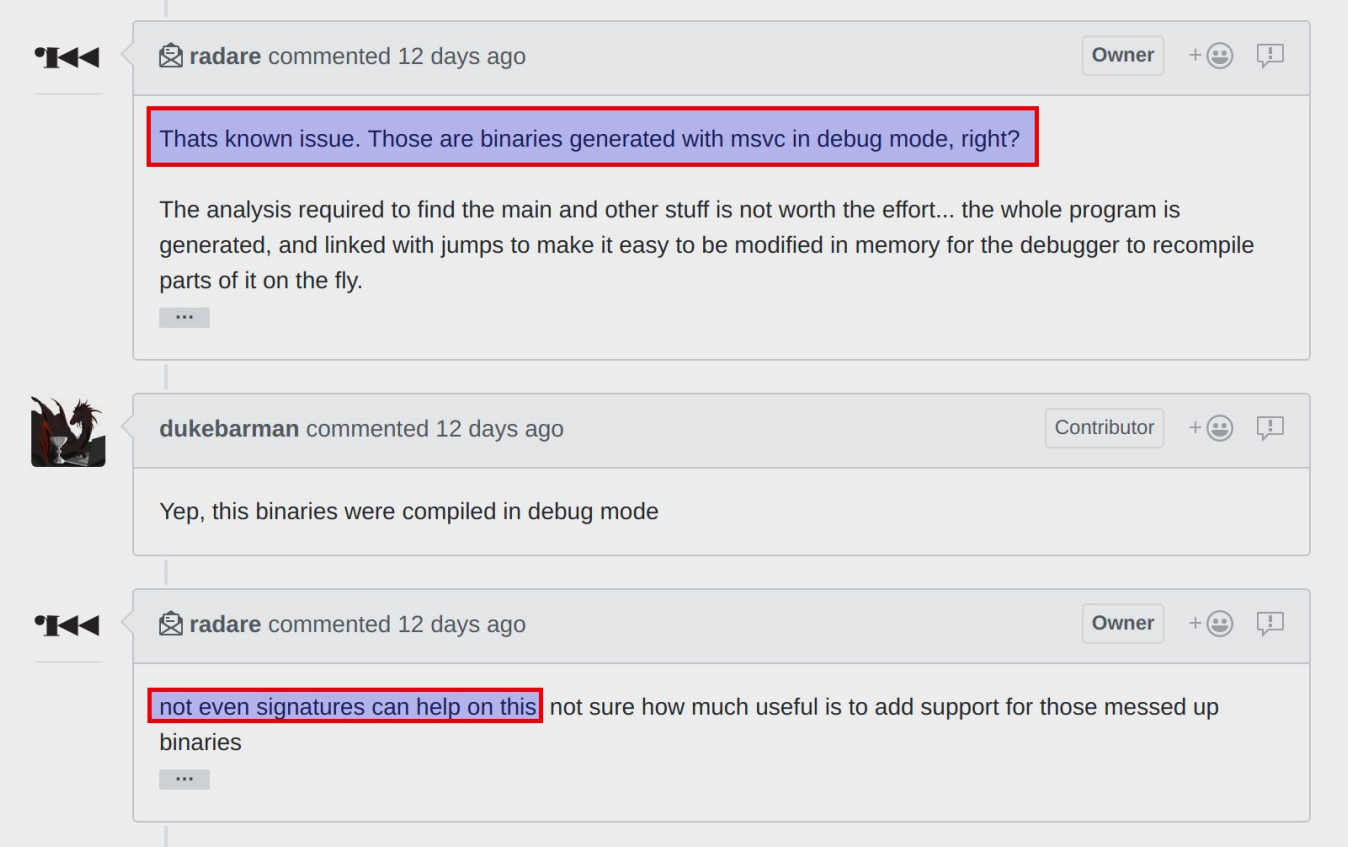
radare2, Visual C++, main.exe (x86)

```
[0x0041105a 2% 720 main.exe]> pd $r @ entry0
/ (fcn) entry0 15
|   entry0 ();
\       ,=< 0x0041105a      e9f10d0000      jmp loc.00411e50            ;[1]
/ (fcn) sub.ucrtbased.dll_wcscpy_s_5f 11
|   sub.ucrtbased.dll_wcscpy_s_5f ();
|       |       ; UNKNOWN XREF from 0x00413ca0 (fcn.00413c30)
|       |       ; CALL XREF from 0x00413ca0 (fcn.00413c30)
|       |       ; CALL XREF from 0x00413cba (fcn.00413c30)
\      ,==< 0x0041105f      e97a3a0000      jmp loc.00414ade            ;[2]
/ (fcn) sub.ucrtbased.dll_strcpy_s_64 11
|   sub.ucrtbased.dll_strcpy_s_64 ();
|      ||       ; UNKNOWN XREF from 0x004121f0 (sub.Stack_corrupted_near_unknown_variable_e6)
|      ||       ; CALL XREF from 0x004121f0 (sub.Stack_corrupted_near_unknown_variable_e6)
|      ||       ; CALL XREF from 0x0041260b (sub.KERNEL32.dll_MultiByteToWideChar_340 + 715)
\     ,===< 0x00411064      e9273a0000      jmp loc.00414a90            ;[3]
     ,====< 0x00411069      e9023b0000      jmp loc.00414b70            ;[4]
/ (fcn) sub.ucrtbased.dll_strcat_s_6e 11
|   sub.ucrtbased.dll_strcat_s_6e ();
|     ||||      ; UNKNOWN XREF from 0x00412202 (sub.Stack_corrupted_near_unknown_variable_e6)
|     ||||      ; CALL XREF from 0x00412202 (sub.Stack_corrupted_near_unknown_variable_e6)
|     ||||      ; CALL XREF from 0x00412218 (sub.Stack_corrupted_near_unknown_variable_e6)
|     ||||      ; CALL XREF from 0x0041261d (sub.KERNEL32.dll_MultiByteToWideChar_340 + 733)
|     ||||      ; CALL XREF from 0x00412633 (sub.KERNEL32.dll_MultiByteToWideChar_340 + 755)
\    ,=====< 0x0041106e      e9233a0000      jmp loc.00414a96            ;[5]
/ (fcn) fcn.00411073 11
|   fcn.00411073 ();
|    |||||      ; UNKNOWN XREF from 0x0041199a (loc.00411970 + 42)
|    |||||      ; CALL XREF from 0x0041199a (loc.00411970 + 42)
|    |||||      ; CALL XREF from 0x004119ca (loc.00411970 + 90)
\   ,======< 0x00411073      e9f8080000      jmp loc.00411970            ;[6]
/ (fcn) sub.KERNEL32.dll_IsDebuggerPresent_78 314
|   sub.KERNEL32.dll_IsDebuggerPresent_78 (int arg_4h, int arg_8h);
|   ||||||    ; var int local_32ch @ ebp-0x32c
|   ||||||    ; var int local_2a0h @ ebp-0x2a0
|   ||||||    ; var int local_29ch @ ebp-0x29c
|   ||||||    ; var int local_298h @ ebp-0x298
```

radare2, Visual C++, main.exe (x86)

What's this madness?

radare2, Visual C++, main.exe (x86)



radare commented 12 days ago — Owner

Thats known issue. Those are binaries generated with msvc in debug mode, right?

The analysis required to find the main and other stuff is not worth the effort... the whole program is generated, and linked with jumps to make it easy to be modified in memory for the debugger to recompile parts of it on the fly.

...

dukebarman commented 12 days ago — Contributor

Yep, this binaries were compiled in debug mode

radare commented 12 days ago — Owner

not even signatures can help on this not sure how much useful is to add support for those messed up binaries

...

radare2,
VisualC++,
main.exe (x86)

RELEASE MODE

radare2, Visual C++, main.exe (x86)

```
[0x00401240 18% 720 main.exe]> pd $r @ entry0
        :   ;-- eip:
/ (fcn) entry0 315
|   entry0 ();
|       :   ; var int local_24h @ ebp-0x24
|       :   ; var int local_19h @ ebp-0x19
|       :   ; var int local_4h @ ebp-0x4
|       :   0x00401240      e8f2030000     call fcn.00401637          ;[1]
\       `=< 0x00401245      e988feffff     jmp 0x4010d2               ;[2]
/ (fcn) sub.KERNEL32.dll_SetUnhandledExceptionFilter_24a 40
|   sub.KERNEL32.dll_SetUnhandledExceptionFilter_24a ();
|             ; UNKNOWN XREF from 0x00401364 (sub.KERNEL32.dll_SetUnhandledExceptionFilter_24a + 282)
|             ; CALL XREF from 0x00401364 (sub.KERNEL32.dll_SetUnhandledExceptionFilter_24a + 282)
|           0x0040124a      55             push ebp
|           0x0040124b      8bec           mov ebp, esp
|           0x0040124d      6a00           push 0
|           0x0040124f      ff1504204000   call dword [sym.imp.KERNEL32.dll_SetUnhandledExceptionFilter]
|           0x00401255      ff7508         push dword [ebp + 8]
|           0x00401258      ff151c204000   call dword [sym.imp.KERNEL32.dll_UnhandledExceptionFilter] ;[4
|           0x0040125e      68090400c0     push 0xc0000409
|           0x00401263      ff1508204000   call dword [sym.imp.KERNEL32.dll_GetCurrentProcess] ;[5] ; 0x4
|           0x00401269      50             push eax
|           0x0040126a      ff152c204000   call dword [sym.imp.KERNEL32.dll_TerminateProcess] ;[6] ; 0x40
|           0x00401270      5d             pop ebp
\           0x00401271      c3             ret
              ; JMP XREF from 0x0040100e (section..text + 14)
            0x00401272      55             push ebp
            0x00401273      8bec           mov ebp, esp
            0x00401275      81ec24030000   sub esp, 0x324
            0x0040127b      6a17           push 0x17                  ; 23
            0x0040127d      e863090000     call sub.KERNEL32.dll_IsProcessorFeaturePresent_be5 ;[7] ; BOO
```

This looks a little bit better, right?

radare2, Visual C++, main.exe (x86)

```
[0x00401240]> f~ entry0
0x00401240 10 entry0
[0x00401240]> f~ main
```

“entry0” flag is defined, but “main” is not :(

radare2, Visual C++, main.exe (x86)

```
[0x00401240]> f~ entry0
0x00401240 10 entry0
[0x00401240]> f~ main
```

“entry0” flag is defined, but “main” is not :(

At this point, there’s only one option. Analyze all that code until main, right?

radare2, Visual C++, main.exe (x86)

```
[0x00401240]> f~ entry0
0x00401240 10 entry0
[0x00401240]> f~ main
```

"entry0" flag is defined, but "main" is not :(

At this point, there's only one option. Analyze all that code until main, right?

Well… We can always try to **cheat**...

radare2, Visual C++, main.exe (x86)

- How do disassemblers know what function is “main”?

Signatures!

Each compiler generates code in a very specific way. Disassemblers identify the specific code generated for the “main” function and then flag that address.

With radare2, you can load signatures with the “z” command

If you happen to have IDA Pro signatures (because you bought an IDA Pro licence...) and they detect those functions, you could load them with:

```
> zfs <signature_file>
```

radare2, Visual C++, main.exe (x86)

Right now, radare2 does not support loading more than one signature at a time, so you can do something like:

- find <path_sigs> -name "*.sig" -fprintf .r2 "zfs %h/%f\n"
- r2 -A -i .r2 <bin>

You'd see how many functions would be flagged. However, radare2 **does not support** the signatures that would detect the main function :(

radare2, Visual C++, main.exe (x86)

Right now, radare2 does not support loading more than one signature at a time, so you can do something like:

- find <path_sigs> -name "*.sig" -fprintf .r2 "zfs %h/%f\n"
- r2 -A -i .r2 <bin>

You'd see how many functions would be flagged. However, radare2 **does not support** the signatures that would detect the main function.

No cheating today :(

radare2, Visual C++, main.exe (x86)

- **Strategies to find main?**

1. Look for known patterns that usually happen before/after calling main.

- Things that can happen **before**:
  Calls to GetCommandLine, functions working with argc or argv, etc.

- Things that can happen **afterwards**:
  Calls to uninitialize_crt, __SEH_epilog4, “exit”, “terminate” functions, etc.

1. Find likely-to-be calls to known windows API made by developer’s code and then go backwards

This requires a little bit of experience, but those type of calls are easy to identify. Windows API call interacting with the filesystem, network, crypto, etc.

radare2, Visual C++, main.exe (x86)

- **Finding our code**

Easy to see this function is a dead end

```
/ (fcn) entry0 315
|   entry0 ();
|       :   ; var int local_24h @ ebp-0x24
|       :   ; var int local_19h @ ebp-0x19
|       :   ; var int local_4h @ ebp-0x4
|       :   0x0040129d      e8f2030000      call fcn.00401694           ;[1]
\       `=< 0x004012a2      e988feffff      jmp 0x40112f                ;[2]
/ (fcn) sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 40
|   sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 ();
|           ; UNKNOWN XREF from 0x004013c1 (sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 + 282)
|           ; CALL XREF from 0x004013c1 (sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 + 282)
|           0x004012a7      55              push ebp
|           0x004012a8      8bec            mov ebp, esp
|           0x004012aa      6a00            push 0
|           0x004012ac      ff1504204000    call dword [sym.imp.KERNEL32.dll_SetUnhandledExceptionFilter]
|           0x004012b2      ff7508          push dword [ebp + 8]
|           0x004012b5      ff151c204000    call dword [sym.imp.KERNEL32.dll_UnhandledExceptionFilter] ;[4
|           0x004012bb      68090400c0      push 0xc0000409
|           0x004012c0      ff1508204000    call dword [sym.imp.KERNEL32.dll_GetCurrentProcess] ;[5] ; 0x4
|           0x004012c6      50              push eax
|           0x004012c7      ff152c204000    call dword [sym.imp.KERNEL32.dll_TerminateProcess] ;[6] ; 0x40
|           0x004012cd      5d              pop ebp
\           0x004012ce      c3              ret
```

Nothing to see here...

radare2, Visual C++, main.exe (x86)

- **Finding our code**

Easy to see this function is a dead end

```
/ (fcn) entry0 315
|   entry0 ();
|       :   ; var int local_24h @ ebp-0x24
|       :   ; var int local_19h @ ebp-0x19
|       :   ; var int local_4h @ ebp-0x4
|       :   0x0040129d      e8f2030000      call fcn.00401694           ;[1]
|       `=< 0x004012a2      e988feffff      jmp 0x40112f                ;[2]
\ (fcn) sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 40
|   sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 ();
|           ; UNKNOWN XREF from 0x004013c1 (sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 + 282)
|           ; CALL XREF from 0x004013c1 (sub.KERNEL32.dll_SetUnhandledExceptionFilter_2a7 + 282)
|           0x004012a7      55              push ebp
|           0x004012a8      8bec            mov ebp, esp
|           0x004012aa      6a00            push 0
|           0x004012ac      ff1504204000    call dword [sym.imp.KERNEL32.dll_SetUnhandledExceptionFilter]
|           0x004012b2      ff7508          push dword [ebp + 8]
|           0x004012b5      ff151c204000    call dword [sym.imp.KERNEL32.dll_UnhandledExceptionFilter] ;[4
|           0x004012bb      68090400c0      push 0xc0000409
|           0x004012c0      ff1508204000    call dword [sym.imp.KERNEL32.dll_GetCurrentProcess] ;[5] ; 0x4
|           0x004012c6      50              push eax
|           0x004012c7      ff152c204000    call dword [sym.imp.KERNEL32.dll_TerminateProcess] ;[6] ; 0x40
|           0x004012cd      5d              pop ebp
\           0x004012ce      c3              ret
```

This is the only thing that remains.
Let's follow it...

Nothing to see here...

Click "2"

radare2, Visual C++, main.exe (x86)

- **Finding our code**

We have a long function now! Let's scroll down:

```
   |   0x004011ff      84c0           test al, al
  ,==< 0x00401201      7408           je 0x40120b                ;[1]
  ||   0x00401203      ff36           push dword [esi]
  ||   0x00401205      e8f5090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__register_thread_local_exe_atexit_callback_bff ;[3]
  ||   0x0040120a      59             pop ecx
  ||      ; JMP XREF from 0x004011f6 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 454)
  ``-> 0x0040120b      e8dd090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll___p___argv_bed ;[4]
       0x00401210      8b38           mov edi, dword [eax]
       0x00401212      e8d0090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll___p___argc_be7 ;[5]
       0x00401217      8bf0           mov esi, eax
       0x00401219      e8a5090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__get_initial_narrow_environment_bc3 ;[6]
       0x0040121e      50             push eax
       0x0040121f      57             push edi
       0x00401220      ff36           push dword [esi]
       0x00401222      e8d9fdffff     call sub.marker_0          ;[7]
       0x00401227      83c40c         add esp, 0xc
       0x0040122a      8bf0           mov esi, eax
       0x0040122c      e848060000     call sub.KERNEL32.dll_GetModuleHandleW_879 ;[8] ; dword GetModuleHandleW(void)
       0x00401231      84c0           test al, al
  ,=<  0x00401233      7506           jne 0x40123b               ;[9]
  |    0x00401235      56             push esi
  |    0x00401236      e89a090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll_exit_bd5 ;[?]
  |       ; JMP XREF from 0x00401233 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 515)
  `-> 0x0040123b      84db           test bl, bl
  ,=< 0x0040123d      7505           jne 0x401244               ;[?]
  |   0x0040123f      e8af090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__cexit_bf3 ;[?]
  |      ; JMP XREF from 0x0040123d (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 525)
  `-> 0x00401244      6a00           push 0
      0x00401246      6a01           push 1                     ; 1
      0x00401248      e880030000     call fcn.004015cd          ;[?]
      0x0040124d      59             pop ecx
```

radare2, Visual C++, main.exe (x86)

- **Finding our code**

We have a long function now! Let's scroll down:

```
  |   0x004011ff      84c0           test al, al
 ,==< 0x00401201      7408           je 0x40120b                 ;[1]
 ||   0x00401203      ff36           push dword [esi]
 ||   0x00401205      e8f5090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__register_thread_local_exe_atexit_callback_bff ;[3]
 ||   0x0040120a      59             pop ecx
 ||      ; JMP XREF from 0x004011f6 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 454)
 ``-> 0x0040120b      e8dd090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll___p___argv_bed ;[4]
      0x00401210      8b38           mov edi, dword [eax]
      0x00401212      e8d0090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll___p___argc_be7 ;[5]
      0x00401217      8bf0           mov esi, eax
      0x00401219      e8a5090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__get_initial_narrow_environment_bc3 ;[6]
      0x0040121e      50             push eax
      0x0040121f      57             push edi
      0x00401220      ff36           push dword [esi]
      0x00401222      e8d9fdffff     call sub.marker_0           ;[7]
      0x00401227      83c40c         add esp, 0xc
      0x0040122a      8bf0           mov esi, eax
      0x0040122c      e848060000     call sub.KERNEL32.dll_GetModuleHandleW_879 ;[8] ; dword GetModuleHandleW(void)
      0x00401231      84c0           test al, al
 ,=<  0x00401233      7506           jne 0x40123b                ;[9]
 |    0x00401235      56             push esi
 |    0x00401236      e89a090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll_exit_bd5 ;[?]
 |       ; JMP XREF from 0x00401233 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 515)
 `-> 0x0040123b      84db           test bl, bl
 ,=< 0x0040123d      7505           jne 0x401244                ;[?]
 |   0x0040123f      e8af090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__cexit_bf3 ;[?]
 |      ; JMP XREF from 0x0040123d (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 525)
 `-> 0x00401244      6a00           push 0
     0x00401246      6a01           push 1                      ; 1
     0x00401248      e880030000     call fcn.004015cd           ;[?]
     0x0040124d      59             pop ecx
```

radare2, Visual C++, main.exe (x86)

- **Finding our code**

Obviously, this looks like our winner...

We have a long function now! Let's scroll down:

```
|   0x004011ff      84c0           test al, al
,==< 0x00401201      7408           je 0x40120b                 ;[1]
||   0x00401203      ff36           push dword [esi]
||   0x00401205      e8f5090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__register_thread_local_exe_atexit_callback_bff ;[3]
||   0x0040120a      59             pop ecx
||      ; JMP XREF from 0x004011f6 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 454)
`-> 0x0040120b      e8dd090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll___p___argv_bed ;[4]
    0x00401210      8b38           mov edi, dword [eax]
    0x00401212      e8d0090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll___p___argc_be7 ;[5]
    0x00401217      8bf0           mov esi, eax
    0x00401219      e8a5090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__get_initial_narrow_environment_bc3 ;[6]
    0x0040121e      50             push eax
    0x0040121f      57             push edi
    0x00401220      ff36           push dword [esi]
    0x00401222      e8d9fdffff     call sub.marker_0           ;[7]
    0x00401227      83c40c         add esp, 0xc
    0x0040122a      8bf0           mov esi, eax
    0x0040122c      e848060000     call sub.KERNEL32.dll_GetModuleHandleW_879 ;[8] ; dword GetModuleHandleW(void)
    0x00401231      84c0           test al, al
,=< 0x00401233      7506           jne 0x40123b                ;[9]
|   0x00401235      56             push esi
|   0x00401236      e89a090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll_exit_bd5 ;[?]
|      ; JMP XREF from 0x00401233 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 515)
`-> 0x0040123b      84db           test bl, bl
,=< 0x0040123d      7505           jne 0x401244                ;[?]
|   0x0040123f      e8af090000     call sub.api_ms_win_crt_runtime_l1_1_0.dll__cexit_bf3 ;[?]
|      ; JMP XREF from 0x0040123d (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 525)
`-> 0x00401244      6a00           push 0
    0x00401246      6a01           push 1                      ; 1
    0x00401248      e880030000     call fcn.004015cd           ;[?]
    0x0040124d      59             pop ecx
```

radare2, Visual C++, main.exe (x86)

- **Finding our code**

There you go! Looks like we can see something similar to what we wrote in our code! We can see the “marker!” string.

```
/ (fcn) sub.marker_0 19
|   sub.marker_0 ();
|              ; UNKNOWN XREF from 0x00401222 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 498)
|              ; CALL XREF from 0x00401222 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 498)
|          0x00401000      68f8204000      push str.marker             ; 0x4020f8 ; "marker!" ; [00] m-r-x s
|          0x00401005      e826000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 ;[1]
|          0x0040100a      83c404          add esp, 4
|          0x0040100d      b802000000      mov eax, 2
\          0x00401012      c3              ret
```

Remember that our code looked like:

```
int func(void) {
        printf(“marker!”);
        int a = 2;
        return a;
}
```

The first 3 assembly instructions are the printf(). Let’s forget about them.

radare2, Visual C++, main.exe (x86)

- **What have we learned?**

$$a = 2 \longrightarrow \text{mov eax, } 2$$

Actually, register “eax” is used here because, **by convention**, functions store the return value in the eax register.

radare2, Visual C++, main.exe (x86)

- **What have we learned?**

$$a = 2 \longrightarrow mov\ eax, 2$$

Actually, register “eax” is used here because, **by convention**, functions store the return value in the eax register.

MOV

Move

Opcode	Mnemonic	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8.
89 /r	MOV r/m16,r16	Move r16 to r/m16.
89 /r	MOV r/m32,r32	Move r32 to r/m32.
8A /r	MOV r8,r/m8	Move r/m8 to r8.
8B /r	MOV r16,r/m16	Move r/m16 to r16.
8B /r	MOV r32,r/m32	Move r/m32 to r32.
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16.
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register.
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL.
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX.
A2	MOV moffs8*,AL	Move AL to (seg:offset).
A3	MOV moffs16*,AX	Move AX to (seg:offset).
A3	MOV moffs32*,EAX	Move EAX to (seg:offset).
B0+ rb	MOV r8,imm8	Move imm8 to r8.
B8+ rw	MOV r16,imm16	Move imm16 to r16.
B8+ rd	MOV r32,imm32	Move imm32 to r32.
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8.
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16.
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32.

http://c9x.me/x86/html/file_module_x86_id_176.html

radare2, Visual C++, main.exe (x86)

- **What have we learned?**

This means there are many ways in which the mov instruction can be used. e.g.:

- mov eax, 2 → Sets eax to 2.

But also:

- mov dword [ebp-4], 2 → The address ebp-4 is set to 2 → C syntax: *(ebp-4) = 2

radare2, Visual C++, main.exe (x86)

- **What have we learned?**

This means there are many ways in which the mov instruction can be used. e.g.:

- mov eax, 2 → Sets eax to 2.

But also:

- mov dword [ebp-4], 2 → The address ebp-4 is set to 2 → C syntax: *(ebp-4) = 2

What a long journey to only learn the **mov** instruction, right?

radare2, Visual C++, main.exe (x86)

- **What have we learned?**

This means there are many ways in which the mov instruction can be used. e.g.:

- mov eax, 2 → Sets eax to 2.

But also:

- mov dword [ebp-4], 2 → The address ebp-4 is set to 2 → C syntax: *(ebp-4) = 2

What a long journey to only learn the **mov** instruction, right?

Well, you've learned much more...

Fast Recap

- **What have you learned so far?**

- Compile binaries from either Windows and Linux
- Compile binaries for different architectures (x86, x64)
- Understand that different compilers generate different assembly (more later)
- Use radare2 to move around code
- Follow “jmp”s and “call”s (although you don’t know yet their inner workings)
- PIC messes disassemby a little bit
- Understand what registers are
- Understand what the mov instruction does
- In x86, function return values are stored in the “eax” register

- **What have you learned so far?**

- Compile binaries from either Windows and Linux
- Compile binaries for different architectures (x86, x64)
- Understand that different compilers generate different assembly (more later)
- Use radare2 to move around code
- Follow “jmp”s and “call”s (although you don’t know yet their inner workings)
- PIC messes disassemby a little bit
- Understand what registers are
- Understand what the mov instruction does
- In x86, function return values are stored in the “eax” register

Time to feel proud of yourself!

- **About: "Understand that different compilers generate different assembly"**

GCC, main_x86 (with symbols)

```
/ (fcn) sym.func 28
|   sym.func ();
|           ; var int local_4h @ ebp-0x4
|              ; CALL XREF from 0x00000519 (sym.main)
|           0x000004ed      55              push ebp
|           0x000004ee      89e5            mov ebp, esp
|           0x000004f0      83ec10          sub esp, 0x10
|           0x000004f3      e82e000000      call sym.__x86.get_pc_thunk.ax ;[1]
|           0x000004f8      05e41a0000      add eax, 0x1ae4
|           0x000004fd      c745fc020000.   mov dword [local_4h], 2
|           0x00000504      8b45fc          mov eax, dword [local_4h]
|           0x00000507      c9              leave
\           0x00000508      c3              ret
```

VC++, main.exe (x86), Release

```
/ (fcn) sub.marker_0 19
|   sub.marker_0 ();
|              ; UNKNOWN XREF from 0x00401222 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 498)
|              ; CALL XREF from 0x00401222 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 498)
|           0x00401000      68f8204000      push str.marker             ; 0x4020f8 ; "marker!" ; [00] m-r-x s
|           0x00401005      e826000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 ;[1]
|           0x0040100a      83c404          add esp, 4
|           0x0040100d      b802000000      mov eax, 2
\           0x00401012      c3              ret
```

- Differences [1]:

- Function prologue/epilogue (more on this later)
- Call to PIC handling function in GCC
- GCC stores value in stack (local variable), VC++ optimizes it.

[1] Aside from the fact that VC++ had a printf()

radare2, GCC, main_x86

STRIPPING SYMBOLS

radare2, GCC, main_x86 → Stripping Symbols

Let's remove all symbols (Binary File Formats class!) from the binary and see how does it affect our analysis

Remember that with our previous binary generated by GCC, radare2 showed us this:

Steps:

1. Strip symbols
   a. $ strip -s main_x86

1. r2 main_x86

radare2, GCC, main_x86 → Stripping Symbols

BEFORE

Remember that with our previous binary generated by GCC, radare2 showed us this:

```
[0x00000507 17% 225 main_x86]> pd $r @ main
            ;-- main:
/ (fcn) sym.main 25
|   sym.main ();
|           0x00000507      55              push ebp
|           0x00000508      89e5            mov ebp, esp
|           0x0000050a      e811000000      call sym.__x86.get_pc_thunk.ax ;[1]
|           0x0000050f      05cd1a0000      add eax, 0x1acd
|           0x00000514      e8d4ffffff      call sym.func                  ;[2]
|           0x00000519      b800000000      mov eax, 0
|           0x0000051e      5d              pop ebp
\           0x0000051f      c3              ret
```

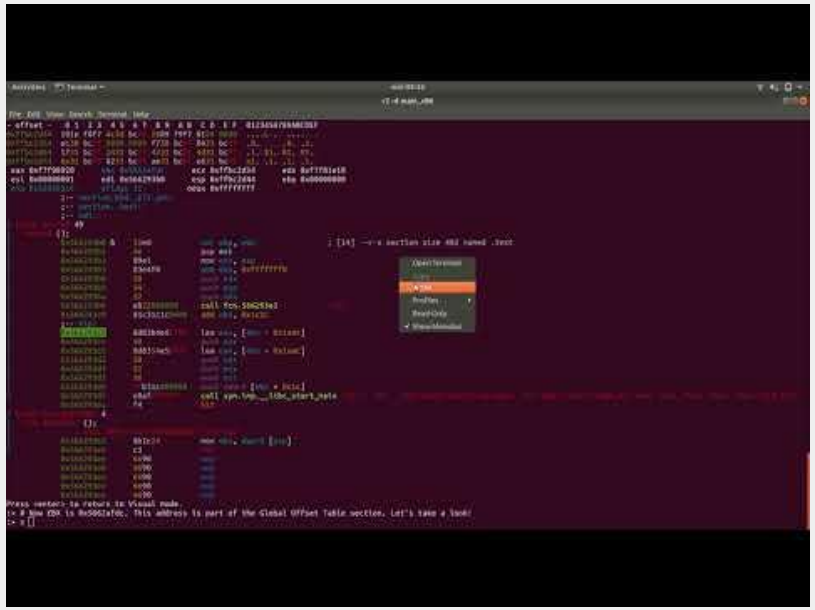
We could even directly move to the “main” function with a seek to main.

radare2, GCC, main_x86 → Stripping Symbols

AFTER

```
[0x000003b0]> f~main
0x00001fe8 4 reloc.__libc_start_main_232
0x00000390 6 sym.imp.__libc_start_main
```

Once you are able to get to main (if you ever do! ;), you'll see that our function "func" is not automatically named anymore!

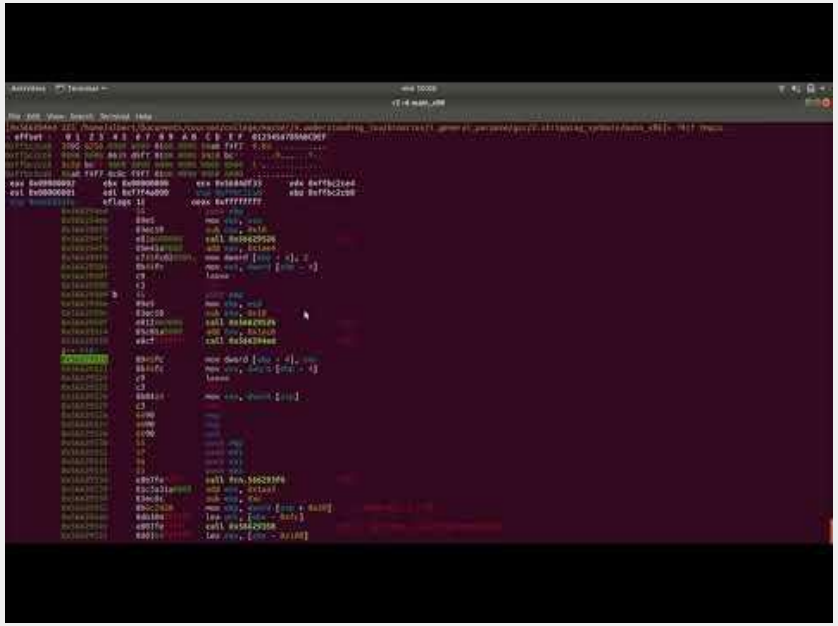


radare2, GCC, main_x86 → Stripping Symbols

AFTER

Our “func” function does not get named (or even detected as a function for what is worth), to fix that:

- af func <address>



Extra Instructions for Assignment

- **LEA**

This instruction is similar to mov and commonly used. It allows operations on the origin registers so extra instructions are avoided.

The origin operand is always enclosed with brackets “[“, “]”, but in this case, it does not mean that the values are dereferenced.

- lea reg1, [reg2] ==> mov reg1, reg2 ==> reg1 = reg2

- lea reg1, [reg2+reg2] → reg1 = reg2 * 2

- lea reg1, [reg2+0xA] → reg1 = reg2 + 10

- lea eax, [esi+ecx*4] → This requires a better explanation!

This is used for iterating over arrays. “esi” is the source register, it points to the base address of the array. “ecx” is the iterator index. 4 bytes is the size of each array item.

This instruction will be inside a loop where “ecx” is incremented.

- **xchg**

- xchg reg1, reg2 → exchange values

- **xor**

xor truth table

0 ^ 0 → 0

0 ^ 1 → 1

1 ^ 0 → 1

1 ^ 1 → 0

- xor reg1, reg1 → Sets reg1 to 0

- Exchange values in an **obfuscated** way:

```
xor eax, ebx
xor ebx, eax        → xchg eax, ebx
xor eax, ebx
```

radare2,
VisualC++,
main.exe (x64)

RELEASE MODE

radare2, Visual C++, main.exe (x64)

- **Finding our code**

```
➜ Release r2 -A main.exe
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
 -- License server overloaded (ETOOMANYCASH)
[0x140001324]> f~ main
0x140001000 26 main
[0x140001324]> # Surprise!
```

Don’t ask me why… But r2 has a successful signature for main within x64 but not for x86!

radare2, Visual C++, main.exe (x64)

- **Differences**

```
/ (fcn) main 26
|   main ();
|              ; CALL XREF from 0x1400012c4 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 + 660)
|           0x140001000      4883ec28        sub rsp, 0x28               ; '(' ; [00] m-r-x section size 4096
|           0x140001004      488d0d051200.   lea rcx, str.marker         ; 0x140002210 ; "marker!"
|           0x14000100b      e820000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_30 ;[1]
|           0x140001010      b802000000      mov eax, 2
|           0x140001015      4883c428        add rsp, 0x28               ; '('
\           0x140001019      c3              ret
```

Two main differences:

- Memory addresses are longer
- The registers have different names

However, our code remains the same!

- mov eax, 2 → That's because a 64 bit register is not needed to store 2.

ISAs → x86 & x64 → General Purpose Registers, Assignments

radare2, Visual C++, main.exe (x64)

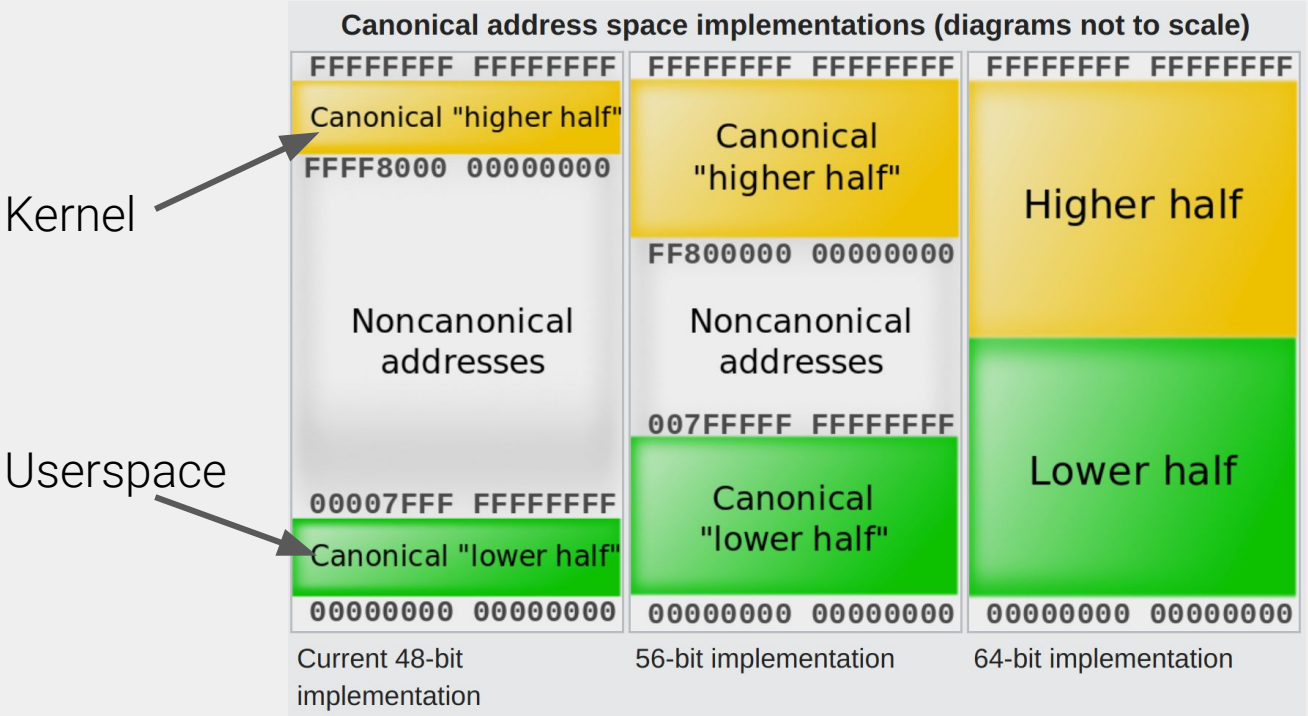
64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

So much new registers!

radare2, Visual C++, main.exe (x64)

- **Canonical Form Addresses**

- Even if in a 64-bit environment, the virtual address space for is not 2^{64}.
- Only the least significant 48 bits are used for addressing.
- The most significant bit (bit 47 starting from 0) is extended.



Arithmetics

Source Code

```
#include <stdio.h>

int func(void) {
        printf("marker!");
        int a = 2;
        int b = 4;
        int c = 0;
        a = a + 2;
        a = a - 2;
        a = a * b;
        c = a / b;
        return a;
}

int main(int argc, char **argv) {
        int a = func();
        return a;
}
```

Disassembly

```
/ (fcn) sym.func 79
|   sym.func ();
|           ; var int local_ch @ rbp-0xc
|           ; var int local_8h @ rbp-0x8
|           ; var int local_4h @ rbp-0x4
|              ; CALL XREF from 0x000006a8 (sym.main)
|           0x0000064a      55              push rbp
|           0x0000064b      4889e5          mov rbp, rsp
|           0x0000064e      4883ec10        sub rsp, 0x10
|           0x00000652      488d3deb0000.   lea rdi, str.marker
|           0x00000659      b800000000      mov eax, 0
|           0x0000065e      e8bdfeffff      call sym.imp.printf
|           0x00000663      c745f4020000.   mov dword [local_ch], 2
|           0x0000066a      c745f8040000.   mov dword [local_8h], 4
|           0x00000671      c745fc000000.   mov dword [local_4h], 0
|           0x00000678      8345f402        add dword [local_ch], 2
|           0x0000067c      836df402        sub dword [local_ch], 2
|           0x00000680      8b45f4          mov eax, dword [local_ch]
|           0x00000683      0faf45f8        imul eax, dword [local_8h]
|           0x00000687      8945f4          mov dword [local_ch], eax
|           0x0000068a      8b45f4          mov eax, dword [local_ch]
|           0x0000068d      99              cdq
|           0x0000068e      f77df8          idiv dword [local_8h]
|           0x00000691      8945fc          mov dword [local_4h], eax
|           0x00000694      8b45f4          mov eax, dword [local_ch]
|           0x00000697      c9              leave
\           0x00000698      c3              ret
```

- **Local variables setup**

```
c745f4020000.  mov dword [local_ch], 2
c745f8040000.  mov dword [local_8h], 4
c745fc000000.  mov dword [local_4h], 0
```

```
int a = 2;
int b = 4;
int c = 0;
```

Let's rename variables.

Approach 1:
- s sym.func → seek to the specific function
- afvn local_ch a; afvn local_8h b; afvn local_4h c

Approach 2:
- In Visual Mode, go to the function, scroll until instruction, press "d", then "n".

```
c745f4020000.  mov dword [a], 2
c745f8040000.  mov dword [b], 4
c745fc000000.  mov dword [c], 0
```

- **Addition and subtraction**

```
8345f402        add dword [a], 2
836df402        sub dword [a], 2
```

a = a + 2;
a = a - 2;

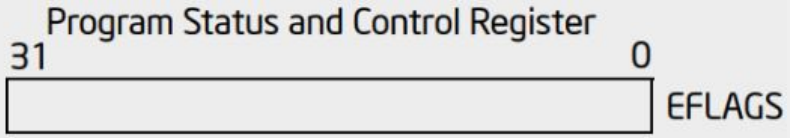
Values are being added/subtracted from a memory address instead of registers.

- *mem_addr = *mem_addr + 2
- *mem_addr = *mem_addr - 2

- Remember the EFLAGS/RFLAGS (x86/x64) register?

E.g. the “sub” instruction sets the following flags:

- The OF, SF, ZF, AF, PF, and CF



- **EFLAGS/RFLAGS Register**

Symbol	Bit	Name	Set if...
CF	0	Carry	Operation generated a carry or borrow
PF	2	Parity	Last byte has even number of 1's, else 0
AF	4	Adjust	Denotes Binary Coded Decimal in-byte carry
ZF	6	Zero	Result was 0
SF	7	Sign	Most significant bit of result is 1
OF	11	Overflow	Overflow on signed operation
DF	10	Direction	Direction string instructions operate (increment or decrement)
ID	21	Identification	Changeability denotes presence of CPUID instruction

- **Multiplication**

```
8b45f4          mov eax, dword [a]
0faf45f8        imul eax, dword [b]
8945f4          mov dword [a], eax
```

a = a * b;

imul → Signed multiplication; **mul** → unsigned multiplication

The Intel Documentation 3-446 Vol. 2A for imul is 4 pages.

In this case:
- eax = eax multiplied by *b → eax = 2 * 4 = 12

There are other syntaxes such as:
- imul reg → e.g. imul rbx → rbx = rbx * rax (the AL, AX, EAX, or RAX is the source)
  - Result is stored as AX, DX:AX, EDX:EAX, or RDX:RAX

- **Multiplication**

With the "**imul reg**" syntax, the result is stored in two registers.

Example, if in x86 you multiply an integer value >2^16 by x, you will need more than 32 bits to store the result!

Your result will be stored as EDX:EAX (EDX upper 32 result bits, EAX low).

However, with the other syntaxes (e.g. imul reg, reg), the result is only stored in one register. **Therefore, some information loss can happen!**

Then you should check CF (Carry Flag) & OF (Overflow Flag) EFLAGS bits to check if information was lost. Simplifying:

- CF: If the operation does not fit in a destination container. Useful for unsigned ops.
- OF: If the MSb of the result is different from both sources. Useful for signed ops.

A quite detailed article about CF & OF.

- **Division**

```
8b45f4          mov eax, dword [a]
99              cdq
f77df8          idiv dword [b]
8945fc          mov dword [c], eax
```

c = a / b;

idiv → Signed multiplication; **div** → unsigned multiplication

div/idiv →
Operation: EDX:EAX / <value>
Quotient (result): EAX
Remainder: EDX

In this case:
EAX = *a,
EDX = 0
EAX = *a / *b
*c = EAX

- **Division**

```
8b45f4          mov eax, dword [a]
99              cdq
f77df8          idiv dword [b]
8945fc          mov dword [c], eax
```

c = a / b;

idiv → Signed multiplication; **div** → unsigned multiplication

div/idiv →
Operation: EDX:EAX / <value>
Quotient (result): EAX
Remainder: EDX

In this case:
EAX = *a,
EDX = 0
EAX = *a / *b
*c = EAX

- **cdq?**

"Converts signed DWORD in EAX to a signed quad word in EDX:EAX by extending the high order bit of EAX throughout EDX."

In this case, EDX will be set to 0 given that EAX = *a and *a = 8.

If **div** instead of **idiv,** an **xor edx, edx** would have been used.

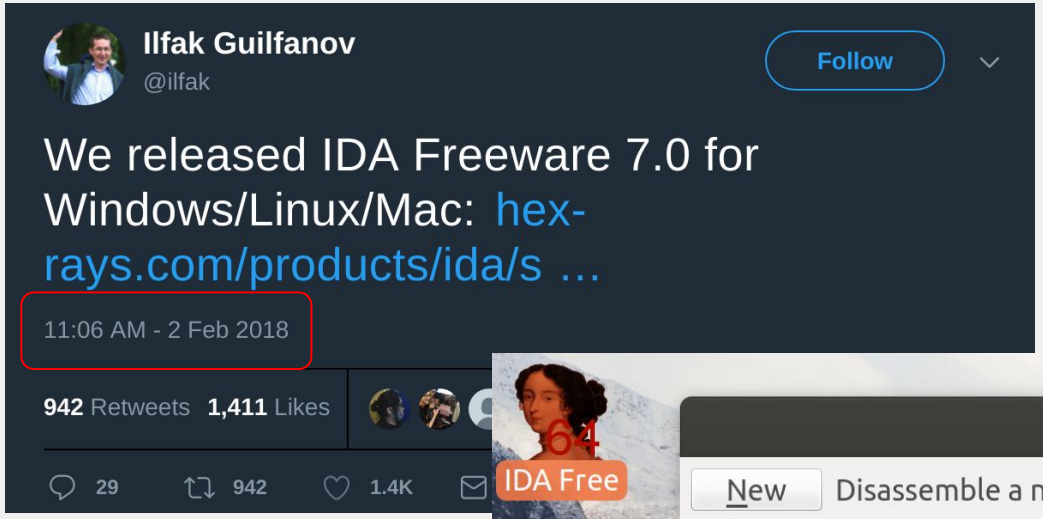
Control Flow

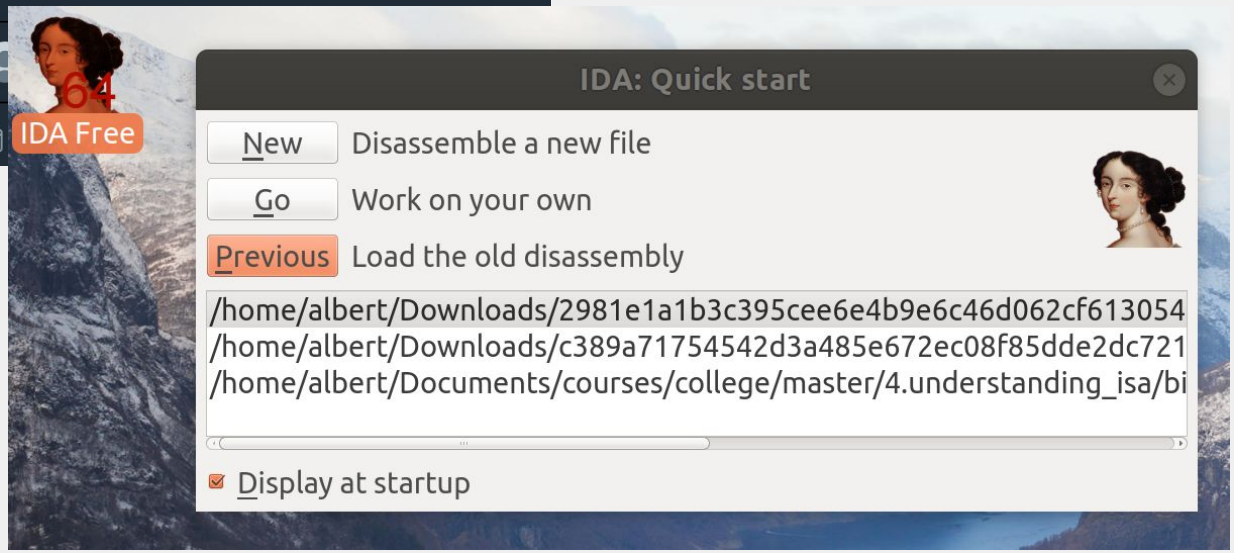
Source Code

- gcc if_equal.c -o if_equal_x64 && strip if_equal_x64

```
#include <stdio.h>

int func(void) {
	int a = 2;
	if (a == 2) {
		printf("a is equal to 2\n");
	} else {
		printf("a is different to 2\n");
	}
	return a;
}

int main(int argc, char **argv) {
	int a = func();
	return a;
}
```

- **Let's use IDA Pro 7 (freeware!)**



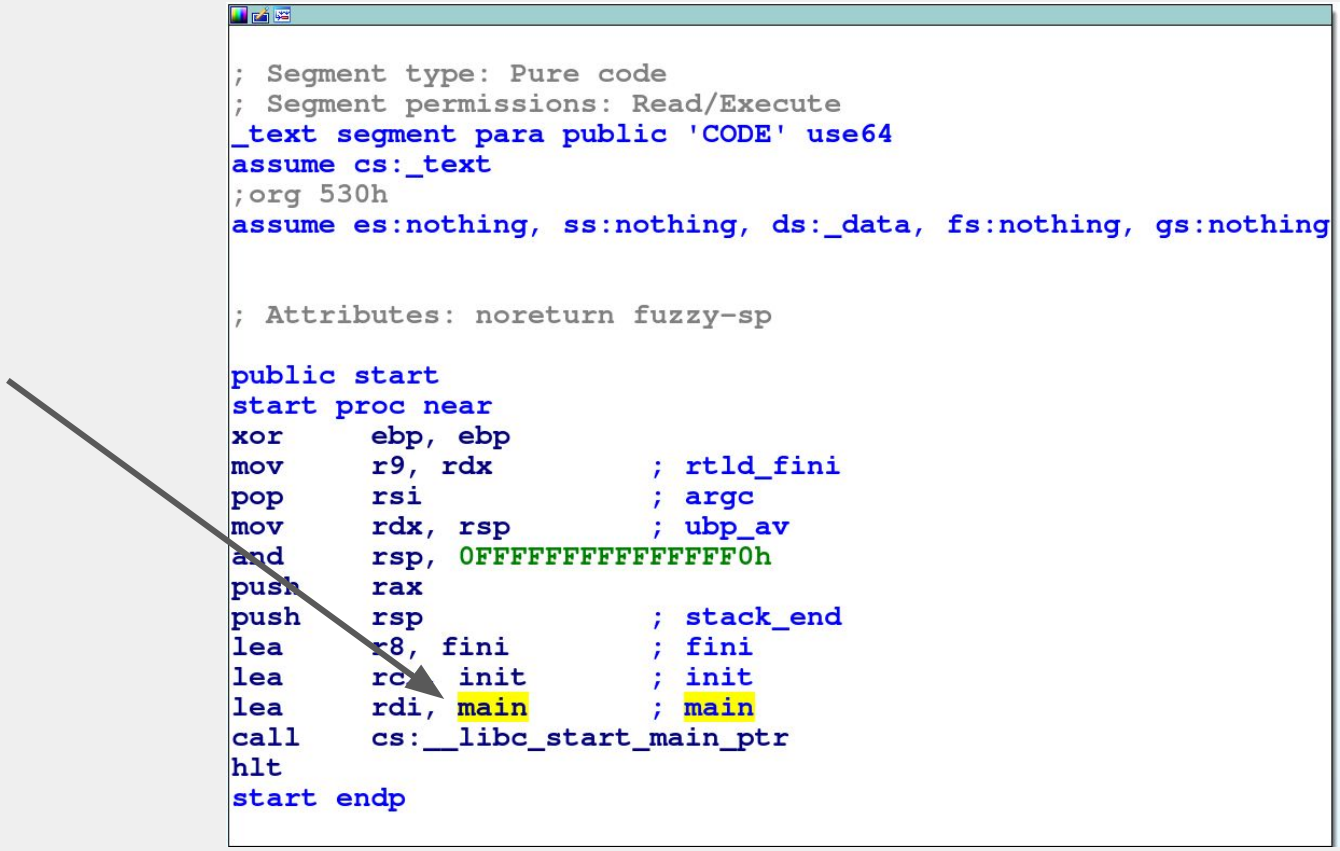


No debugging supported...

IDA Pro “start” is equivalent to r2 “entry0”

IDA Pro detects the parameters to __libc_start_main and names the “main” address parameter appropriately

Let’s double click!

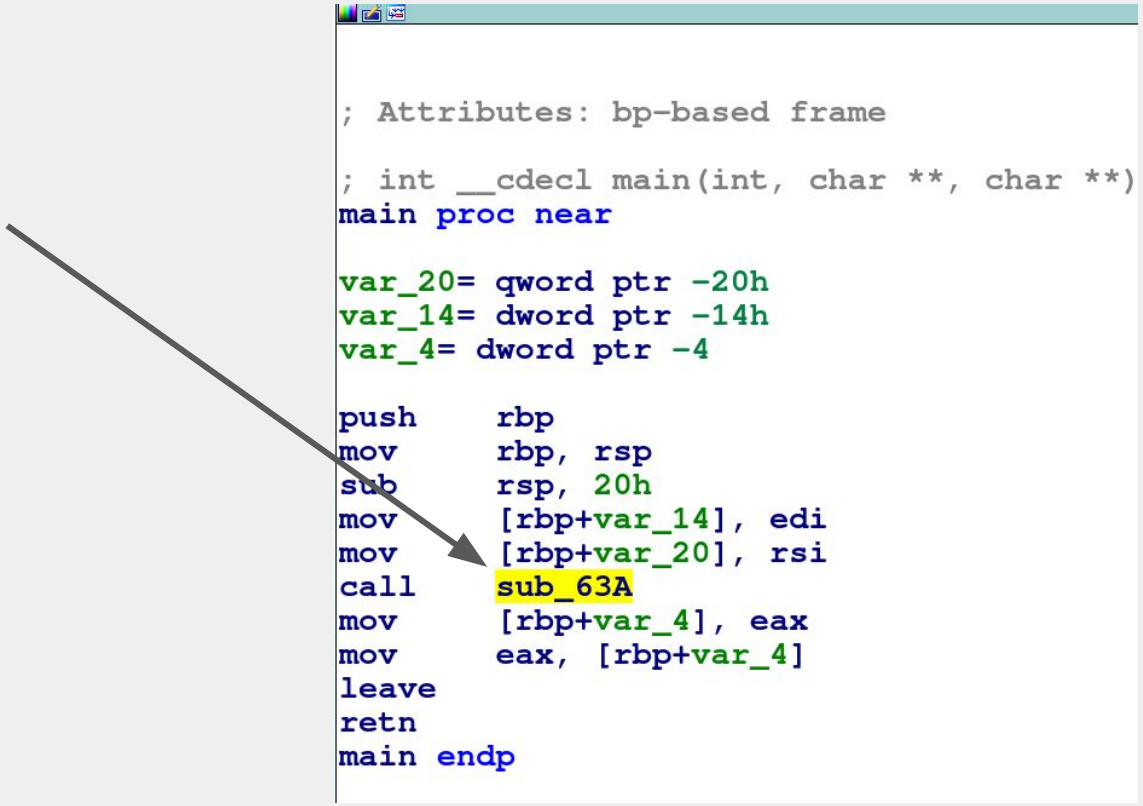


```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 530h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


; Attributes: noreturn fuzzy-sp

public start
start proc near
xor     ebp, ebp
mov     r9, rdx         ; rtld_fini
pop     rsi             ; argc
mov     rdx, rsp        ; ubp_av
and     rsp, 0FFFFFFFFFFFFFFF0h
push    rax
push    rsp             ; stack_end
lea     r8, fini        ; fini
lea     rcx, init       ; init
lea     rdi, main       ; main
call    cs:__libc_start_main_ptr
hlt
start endp
```

IDA Pro “start” is equivalent to r2 “entry0”

This is our “func”, let’s rename it!

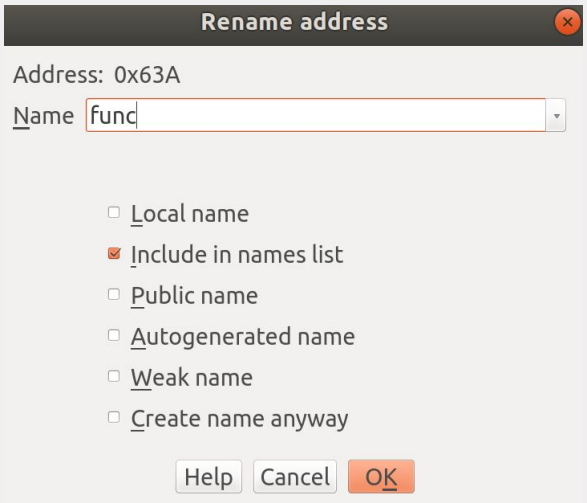
Put the pointer on the function name and press “n”.



IDA Pro “start” is equivalent to r2 “entry0”

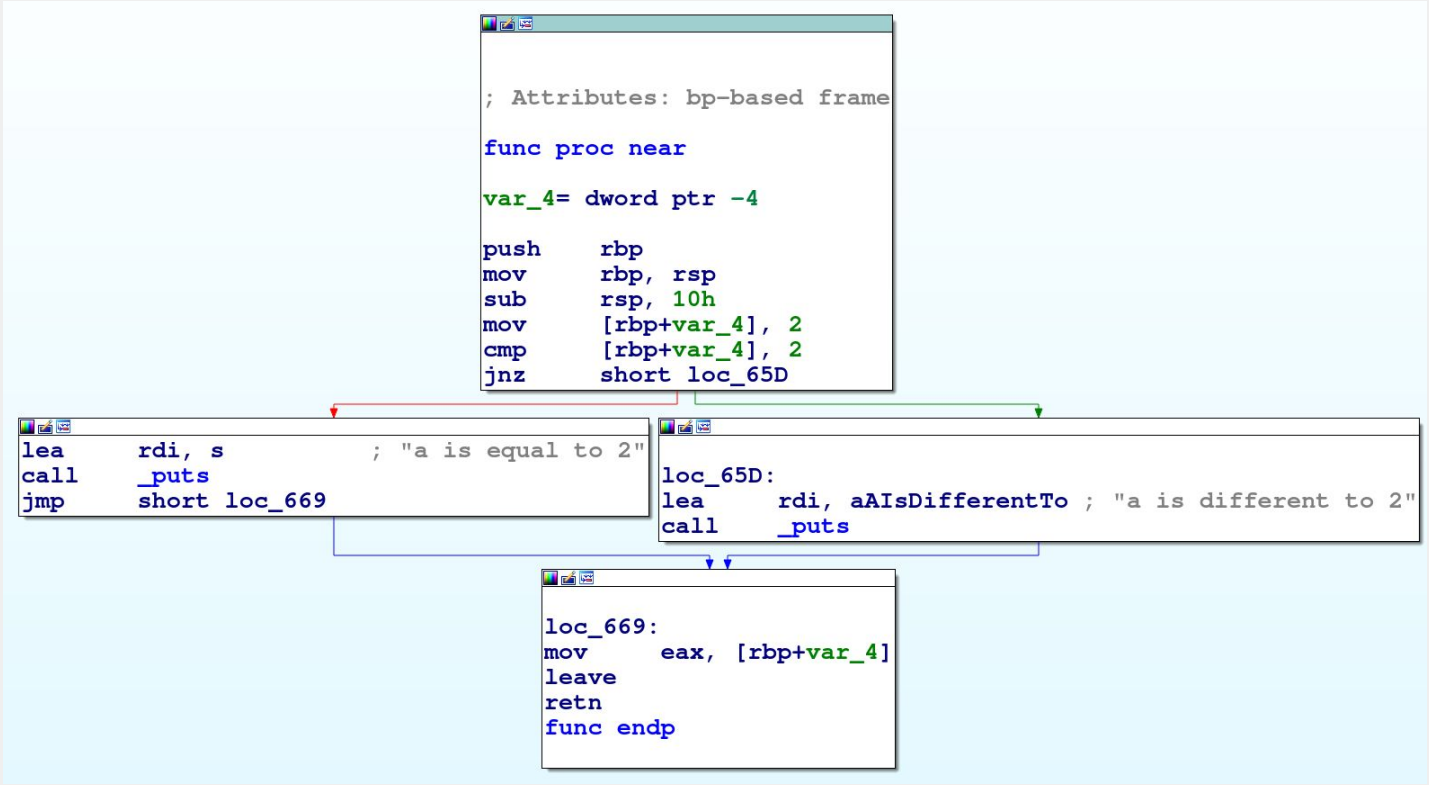
This is our “func”, let’s rename it!

Put the pointer on the function name and press “n”.

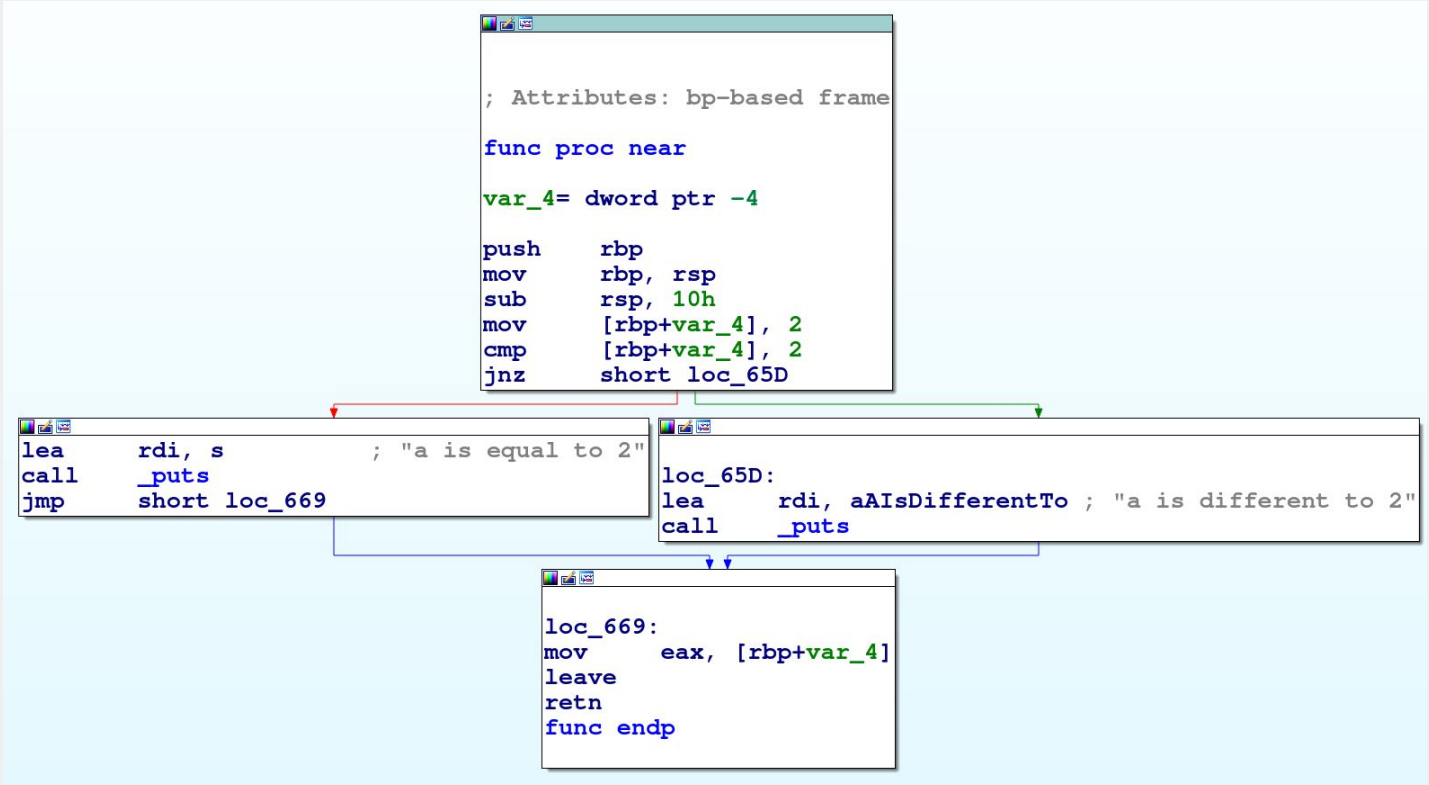



```
; Attributes: bp-based frame

; int __cdecl main(int, char **, char **)
main proc near

var_20= qword ptr -20h
var_14= dword ptr -14h
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_14], edi
mov     [rbp+var_20], rsi
call    sub_63A
mov     [rbp+var_4], eax
mov     eax, [rbp+var_4]
leave
retn
main endp
```

Do you wanna know what each option in IDA Pro means? You’ll have to buy the IDA Pro Book!



cmp & jnz
compare & jump not zero



cmp & jnz
compare & jump not zero

- **Compare (cmp) instruction**

The two operands are compared and the EFLAGS/RFLAGS register is set appropriately.

The second operand is subtracted from the first one and the flags are set the same way the “sub” instruction would do it.

- Most important operations:

if a = b → cmp a, b → ZF set
if a < b → cmp a, b → SF set
if a > b → cmp a, b → ZF and SF not set.

If you ever want to know how instructions affect flags, check the Appendix A, EFLAGS AND INSTRUCTIONS from Intel Manual, Volume 1.

- **Test instruction**

The test instruction is similar to the compare (cmp) instruction in the sense that it compares two operands. The difference is that instead of doing a “sub” between them, it does a logical “and”.

For example, to check if EAX is 0 you can do:

- test eax, eax

The “test eax, eax” will do an “and” and if the result is 0 (because eax is 0) the ZF will be set appropriately for the next instruction to be used.

It is also useful to check **if register is negative**. If eax is negative, the SF (Sign Flag) when “test eax, eax” will be set (given that the “and” still outputs a negative number)

- **Jump Not Zero (jnz) instruction**

The “jnz” instruction is part of the “Jump If Condition Is Met” type of instructions, usually represented with “Jcc”.

The “Jcc” instruction is followed by an offset. This offset is relative to the EIP (actual instruction being executed).

If the condition specified by the jcc instruction is met, the execution flow will jump/execute the instruction at EIP+/-offset.

In the case of “jnz”, if the flag ZF is 0, a the execution flow will “jump”, otherwise it will continue to the “next” instruction.

- **Jump Not Zero (jnz) instruction**

If you want to know what flags do each “jcc” instruction check, you can read the Instruction Set Reference, Jcc section of the Intel Manuals.

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 *cb*	JA *rel8*	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 *cb*	JAE *rel8*	D	Valid	Valid	Jump short if above or equal (CF=0).
72 *cb*	JB *rel8*	D	Valid	Valid	Jump short if below (CF=1).
76 *cb*	JBE *rel8*	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 *cb*	JC *rel8*	D	Valid	Valid	Jump short if carry (CF=1).
E3 *cb*	JCXZ *rel8*	D	N.E.	Valid	Jump short if CX register is 0.
E3 *cb*	JECXZ *rel8*	D	Valid	Valid	Jump short if ECX register is 0.
E3 *cb*	JRCXZ *rel8*	D	Valid	N.E.	Jump short if RCX register is 0.
74 *cb*	JE *rel8*	D	Valid	Valid	Jump short if equal (ZF=1).
7F *cb*	JG *rel8*	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D *cb*	JGE *rel8*	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C *cb*	JL *rel8*	D	Valid	Valid	Jump short if less (SF≠ OF).
7E *cb*	JLE *rel8*	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 *cb*	JNA *rel8*	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 *cb*	JNAE *rel8*	D	Valid	Valid	Jump short if not above or equal (CF=1).

- **Jump Not Zero (jnz) instruction**

In general, many instructions follow the same naming convention than “jcc”. To know what flags you need to check you can read the Appendix B, EFLAGS and Condition Codes section of the Intel Manuals, Vol 1.

Table B-1. EFLAGS Condition Codes

Mnemonic (*cc*)	Condition Tested For	Instruction Subcode	Status Flags Setting
O	Overflow	0000	OF = 1
NO	No overflow	0001	OF = 0
B NAE	Below Neither above nor equal	0010	CF = 1
NB AE	Not below Above or equal	0011	CF = 0
E Z	Equal Zero	0100	ZF = 1
NE NZ	Not equal Not zero	0101	ZF = 0
BE NA	Below or equal Not above	0110	(CF OR ZF) = 1
NBE A	Neither below nor equal Above	0111	(CF OR ZF) = 0
S	Sign	1000	SF = 1
NS	No sign	1001	SF = 0
P PE	Parity Parity even	1010	PF = 1
NP PO	No parity Parity odd	1011	PF = 0
L NGE	Less Neither greater nor equal	1100	(SF XOR OF) = 1
NL GE	Not less Greater or equal	1101	(SF XOR OF) = 0
LE NG	Less or equal Not greater	1110	((SF XOR OF) OR ZF) = 1
NLE G	Neither less nor equal Greater	1111	((SF XOR OF) OR ZF) = 0

- **Jump Not Zero (jnz) instruction**

- **Equivalent Instructions:**

There are many instructions that mean the same but are represented differently.

For example, **“JA” (Jump if Above) “JNBE” (Jump if Not Below or Equal)** are the same. Both are represented by opcode 0x77.

What instruction is shown depends on the disassembly tool.

- **Far vs Short Jumps**

A “jcc” instruction cannot do far jumps. The offset that follows a “jcc” instruction is a maximum of a 32/64 bits signed integer offset:

e.g. A 32 bits signed integer range is $-(2^{31})$ to $(2^{31})-1$.

If there’s the need of bigger jumps, the “jmp” instruction needs to be used.

- **Jump Not Zero (jnz) instruction**

- **Signed vs Unsigned comparisons:**

Values stored in memory/registers can be either signed or unsigned.

Philosophical statement of the day:

Actually, there are no signed and unsigned values, but just different ways of interpreting them.

- What does this mean?

JA (Jump if Above) interprets the comparison made by CMP as if it was a comparison with unsigned values.

JG (Jump if Greater) interprets the comparison made by CMP as if it was a comparison with signed values.

The compiler will chose the appropriate instruction.

More info: https://stackoverflow.com/questions/20906639/difference-between-ja-and-jg-in-assembly

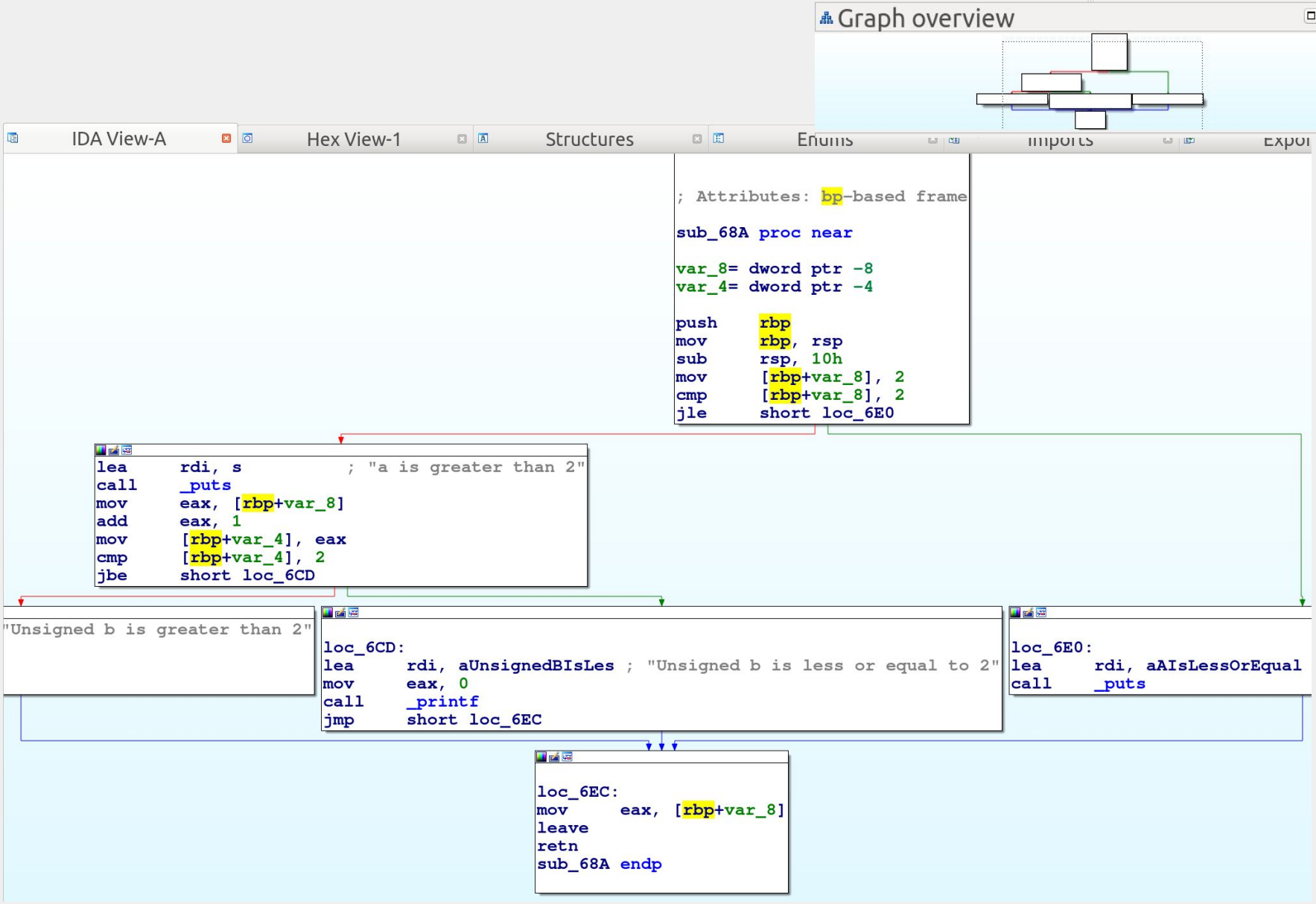
Some nested conditions

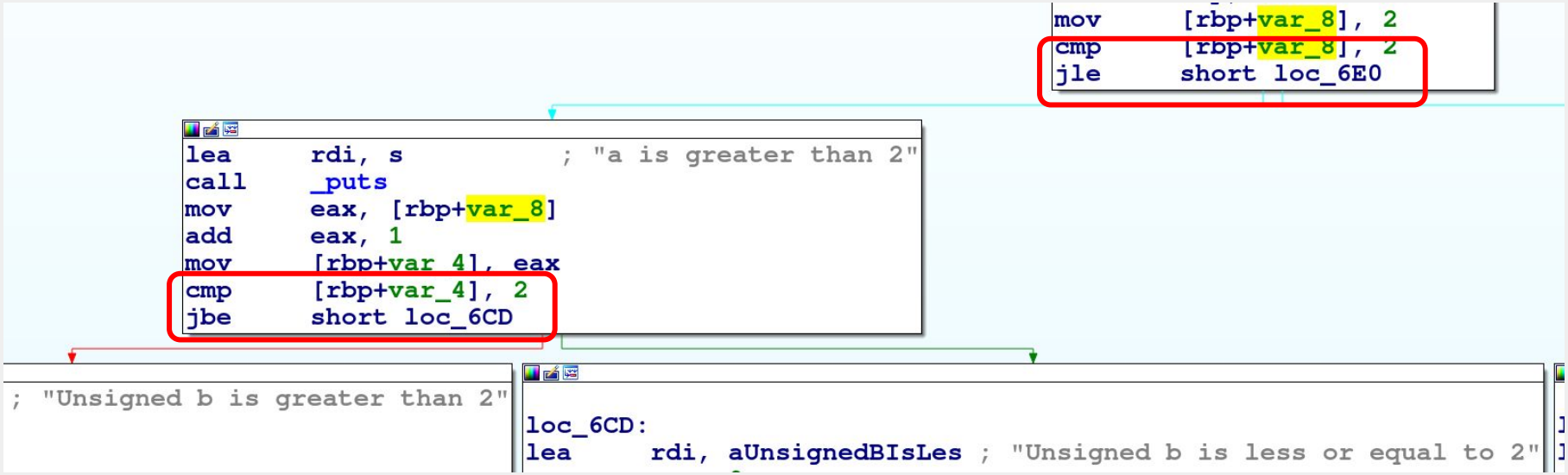
Source Code

- gcc complex_flow.c -o complex_flow_x64 && strip complex_flow_x64

```
#include <stdio.h>

int func(void) {
	int a = 2;
	  if (a > 2) {
		printf("a is greater than 2\n");
		unsigned int b = a + 1;
		if (b > 2)
			printf("Unsigned b is greater than 2");
		else
			printf("Unsigned b is less or equal to 2");

	  } else {
		printf("a is less or equal to 2\n");
	  }
	return a;
}

int main(int argc, char **argv) {
	int a = func();
	return a;
}
```

ISAs → x86 & x64 → Control Flow → Complex Workflow





Here we have two different instructions that mean the “same” but interpret values in a different way.

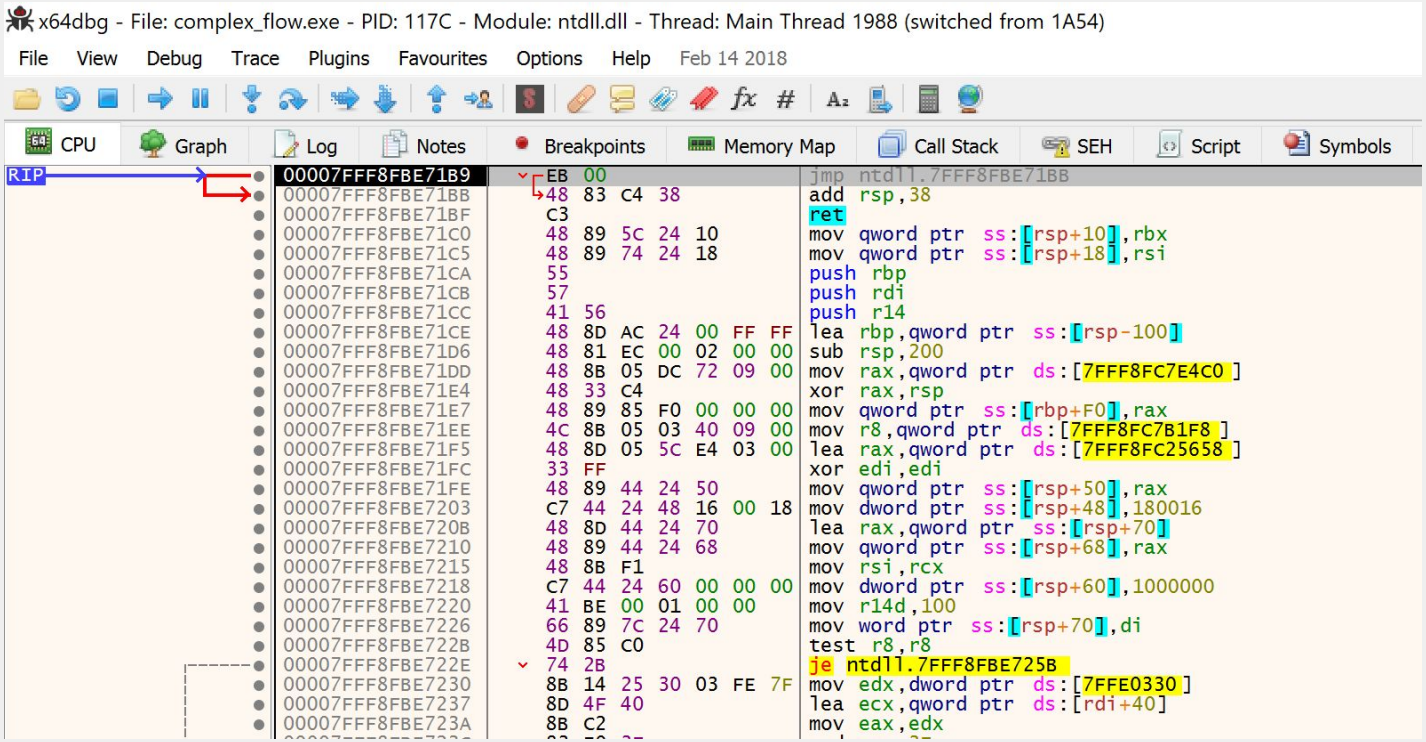
- JLE (Jump Less or Equal)
- JBE (Jump Below or Equal)

With this you can tell VAR_4 is unsigned.

These type of workflows prove how helpful the graph view is.

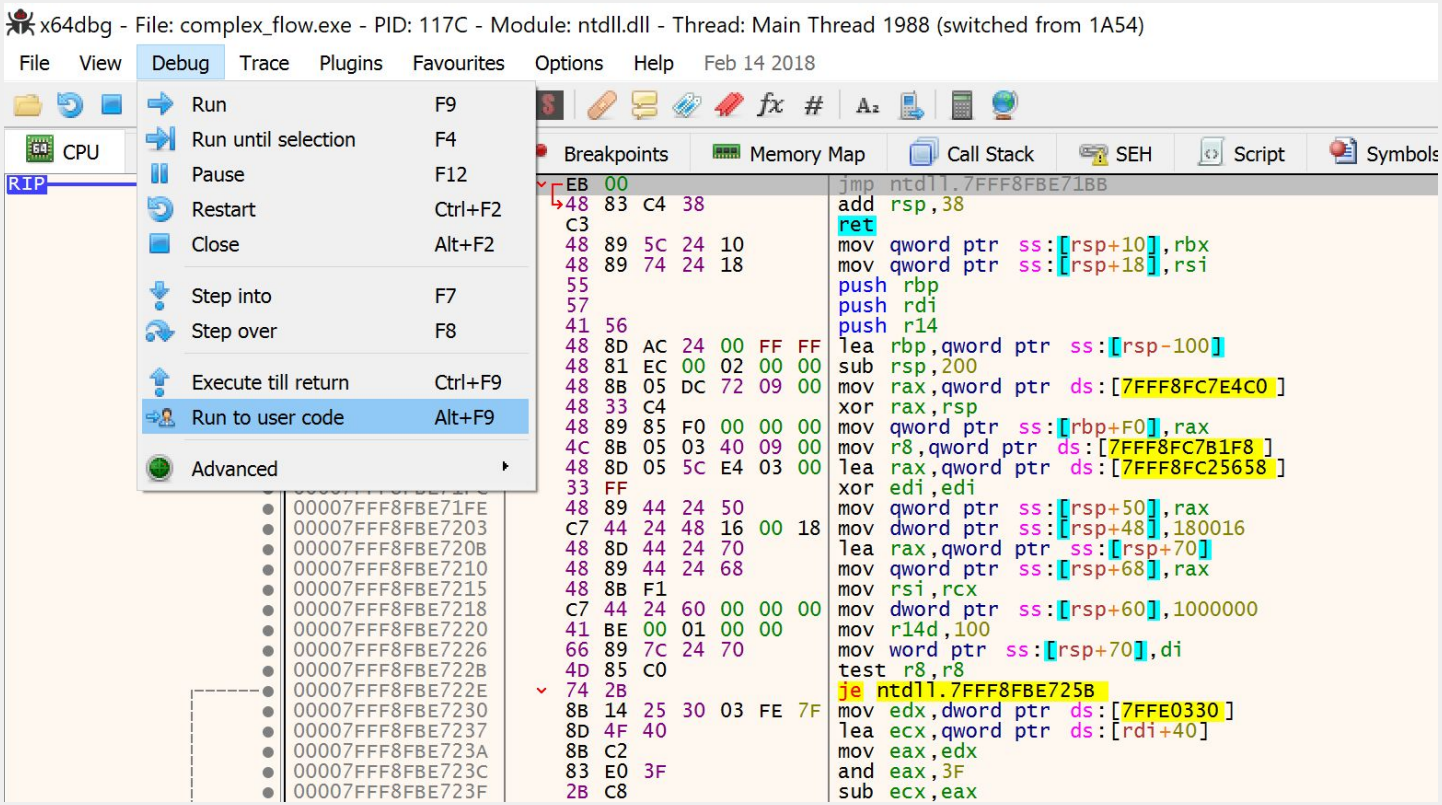
- **Let's Debug it!**

We are going to use x64dbg. You could use OllyDgb, WinDGB, Radare2, IDA Pro, etc. However, x64dbg is open and has an active community.



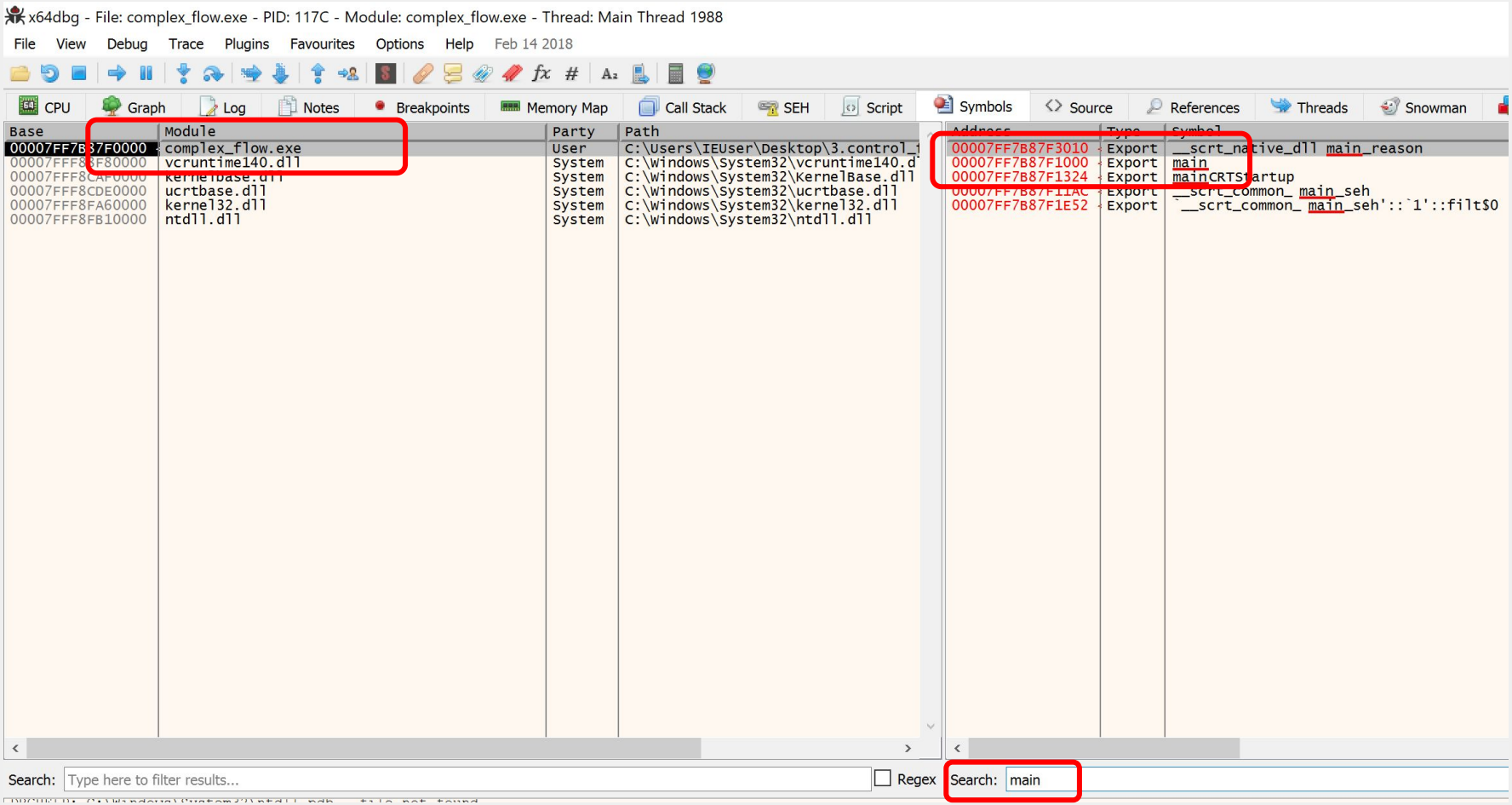
In the beginning we might not know where are we.

x64dbg provides the option to execute code until getting to user code (the main module).

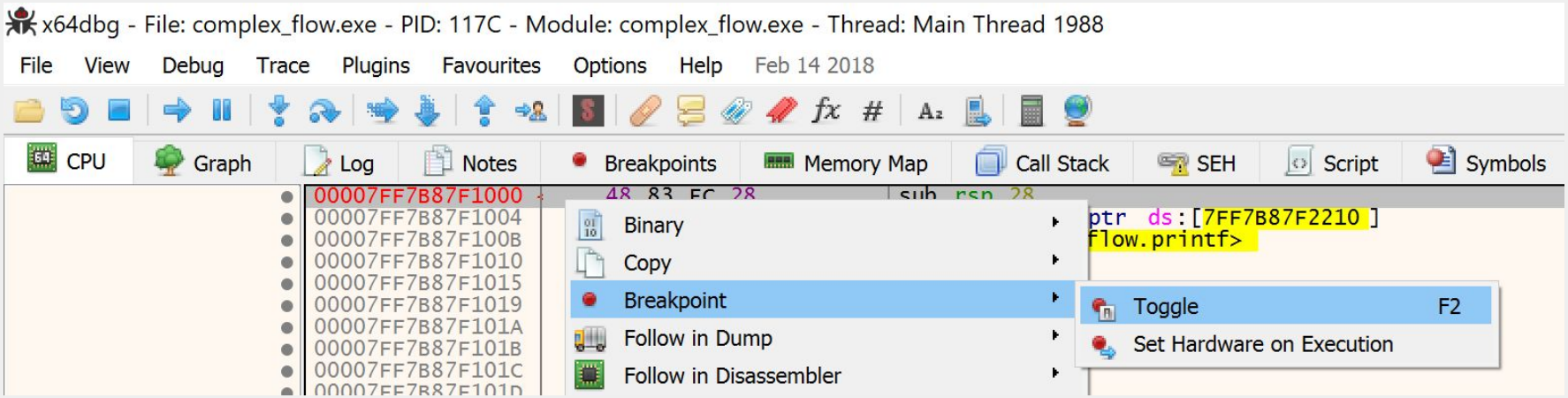


We can also go to Symbols and check if the “main” symbol was found!

If so, just double click in the symbol.

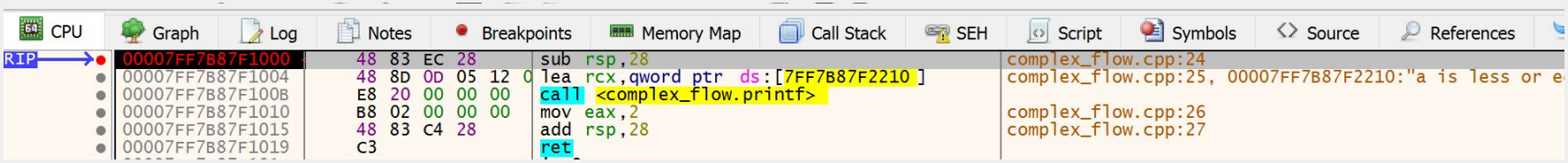


Once you are in the “main” address, press F2.



Now you can automatically execute all instructions until this point by pressing F9.

Once there, we can see the main function is just this!



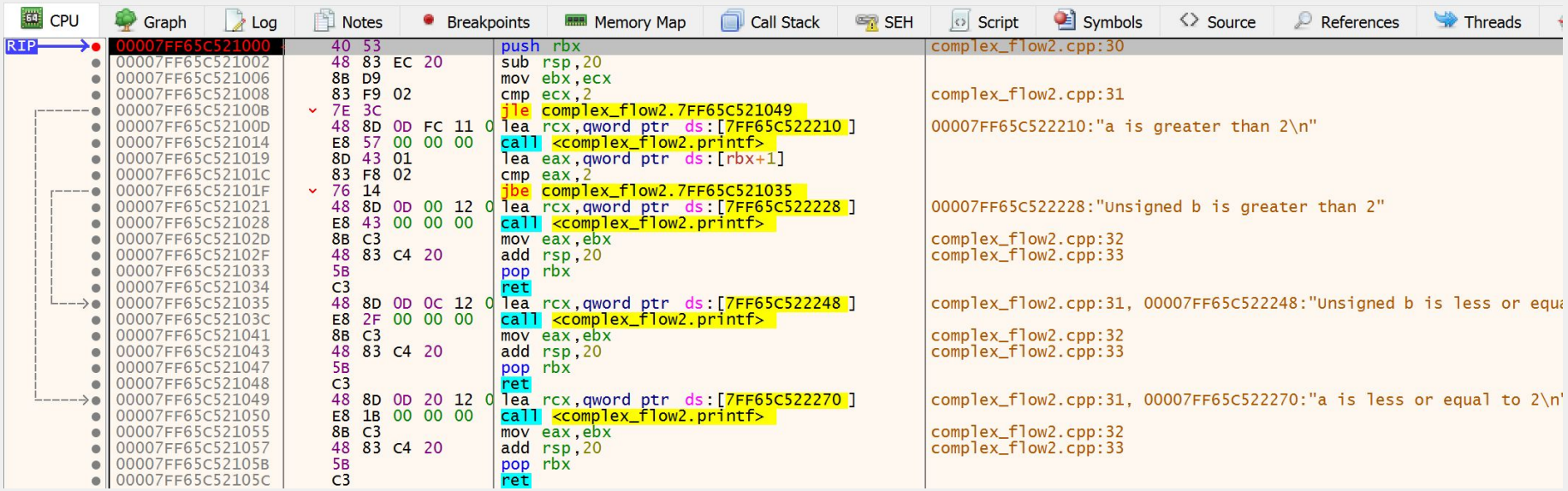
What the heck happened to all our code we saw from GCC disassembly?

Visual Studio removed “dead code”!!

Let’s fix that:

```
#include "stdafx.h"
#include <stdio.h>

int func(int nparams) {
	int a = nparams;
	if (a > 2) {
		printf("a is greater than 2\n");
		unsigned int b = a + 1;
		if (b > 2)
			printf("Unsigned b is greater than 2");
		else
			printf("Unsigned b is less or equal to 2");

	}
	else {
		printf("a is less or equal to 2\n");
	}
	return a;
}

int main(int argc, char **argv) {
	int a = func(argc);
	return a;
}
```

Let’s debug it and see how the jumps and the registers change!



Some mystery

Exercise: Understand the following workflow

```
/ (fcn) main 83
|   main ();
|             ; CALL XREF from 0x1400012f7 (sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_70 + 647)
|             ; CALL XREF from 0x140001000 (main)
|           0x140001000      4053            push rbx                    ; [00] m-r-x section size 4096 named .text
|           0x140001002      4883ec20        sub rsp, 0x20
|           0x140001006      8bd1            mov edx, ecx
|           0x140001008      8bd9            mov ebx, ecx
|           0x14000100a      83ea01          sub edx, 1
|       ,=< 0x14000100d      740c            je 0x14000101b              ;[1]
|       |   0x14000100f      83ea01          sub edx, 1
|      ,==< 0x140001012      7413            je 0x140001027              ;[2]
|      ||   0x140001014      83fa01          cmp edx, 1                  ; 1
|     ,===< 0x140001017      741a            je 0x140001033              ;[3]
|    ,====< 0x140001019      eb24            jmp 0x14000103f             ;[4]
|    ||||      ; JMP XREF from 0x14000100d (main)
|    |||`-> 0x14000101b      488d0dee1100.   lea rcx, str.a_is_one       ; 0x140002210 ; "a is one"
|    |||     0x140001022      e849000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_70 ;[5]
|    |||      ; JMP XREF from 0x140001012 (main)
|    ||`--> 0x140001027      488d0df21100.   lea rcx, str.a_is_two       ; 0x140002220 ; "a is two"
|    ||      0x14000102e      e83d000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_70 ;[5]
|    ||      ; JMP XREF from 0x140001017 (main)
|    |`---> 0x140001033      488d0df61100.   lea rcx, str.a_is_three     ; 0x140002230 ; "a is three"
|    |       0x14000103a      e831000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_70 ;[5]
|    |       ; JMP XREF from 0x140001019 (main)
|    `----> 0x14000103f      488d0dfa1100.   lea rcx, str.who_knows_what_a_is    ; 0x140002240 ; "who knows what a is!"
|           0x140001046      e825000000      call sub.api_ms_win_crt_stdio_l1_1_0.dll___acrt_iob_func_70 ;[5]
|           0x14000104b      8bc3            mov eax, ebx
|           0x14000104d      4883c420        add rsp, 0x20
|           0x140001051      5b              pop rbx
\           0x140001052      c3              ret
```

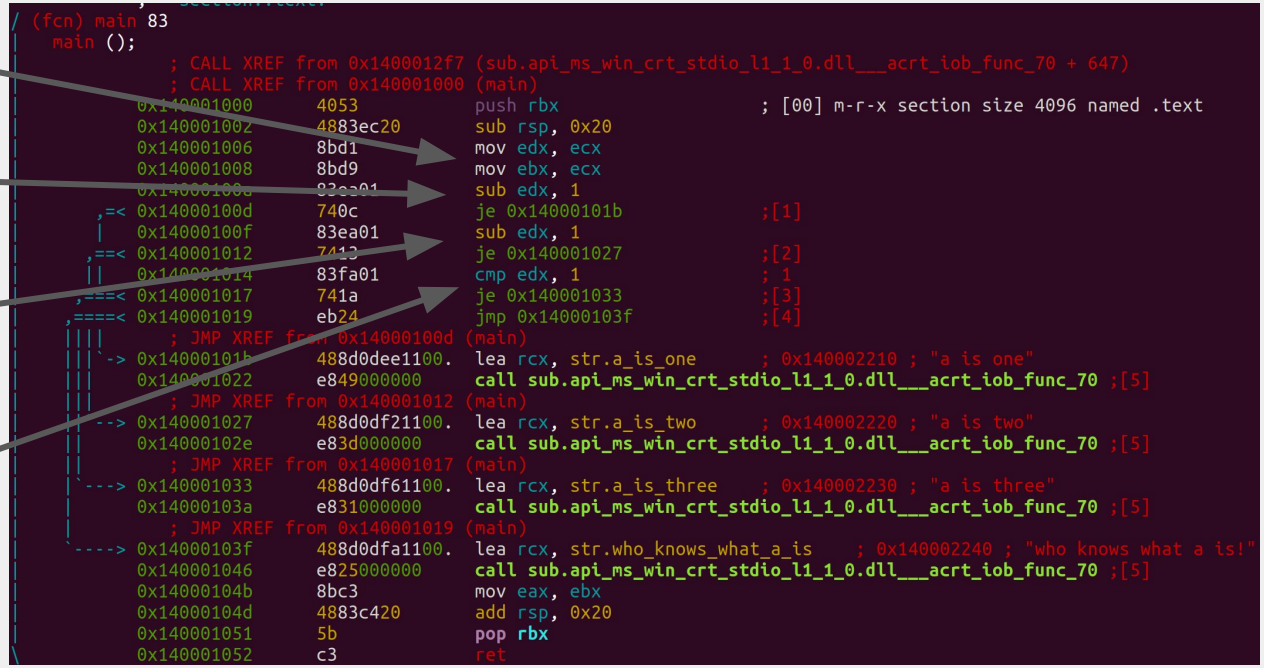
Exercise: Understand the following workflow

ecx contains param.
It's moved to edx

edx = edx - 1.
"je" means ¿ZF = 1?
¿was edx = 1?

edx = edx - 1.
"je" means ¿ZF = 1?
¿edx = 2?

edx = edx - 1.
"je" means ¿ZF = 1?
¿edx = 3?



It was a **switch**!

```
#include "stdafx.h"
#include <stdio.h>

int func(int nparams) {
	int a = nparams;
	switch (a) {
	case 1:
		printf("a is one");
	case 2:
		printf("a is two");
	case 3:
		printf("a is three");
	default:
		printf("who knows what a is!");
		break;
	}
	return a;
}
int main(int argc, char **argv) {
	int a = func(argc);
	return a;
}
```

Stack Operations

- **What's the stack?**

The stack is a region of memory that a given program uses following specific rules and with a specific purpose.

The stack is used by programs to store (among others):
- Function variables
- Function call "history"

By convention, the operations taken on the stack follow a LIFO (Last In First Out) philosophy [1].

This means that a program can not make arbitrary read/writes to this memory region, they must follow strict rules.

Examples:
- Where to read/write is defined by the address stored in the Stack Pointer register
- Your read/write instructions only have one operand
- Read/write instructions affect the Stack Pointer

1 That's, actually, why this memory region is called "stack", coming from the algorithmic world.

- **What's the stack?**

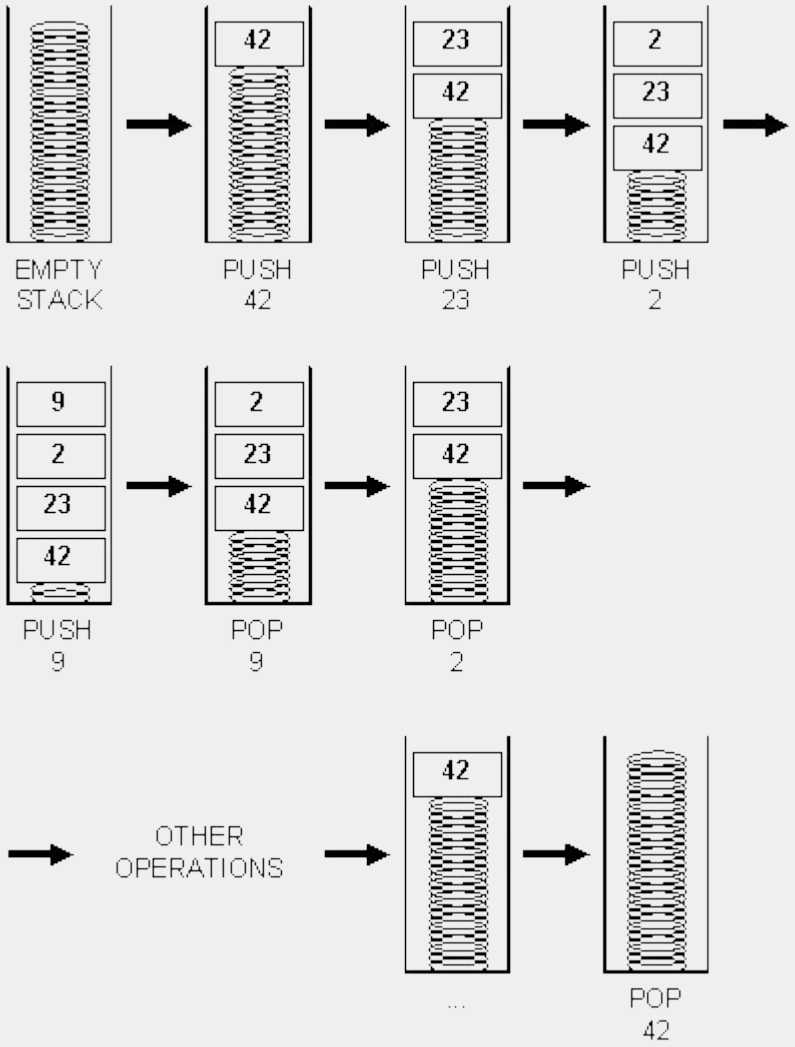
PUSH instruction writes to the stack:
- push eax
- push 0x23

POP instruction reads from the stack:
- pop <reg>
- pop [<reg>]

The stack pointer (SP, ESP, RSP) is what dictates from/to what memory address content will be read/written.

PUSH/POP instructions add/subtract 2, 4 or 8 from the current value of the stack pointer.

This means that the stack grows to lower memory positions.



- **What's the stack?**

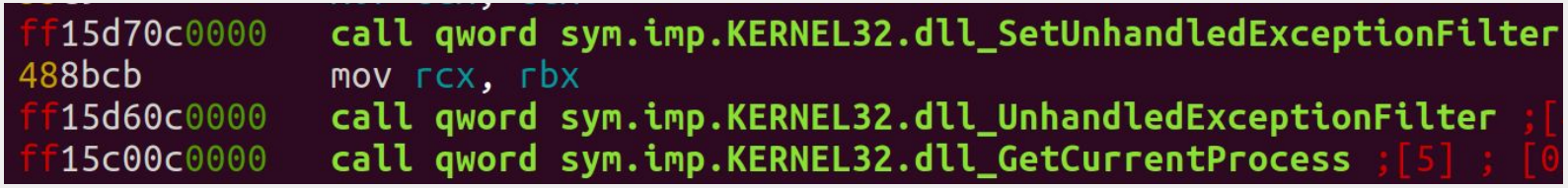
Most of the stack operations are **related to function calls**.

So let's continue...

Functions

At this point, we've seen functions all around!

- A function is just a piece of code/instruction that within a broader structure.



- Functions can receive data and can output data.

- A function is called with the CALL instruction.
  - CALL instruction needs 14 pages of documentation in the Intel Manual.

- You get back from a function with a RET instruction.

- When a function is called, a **stack frame** is built.

Stack Frames

Stack Frames

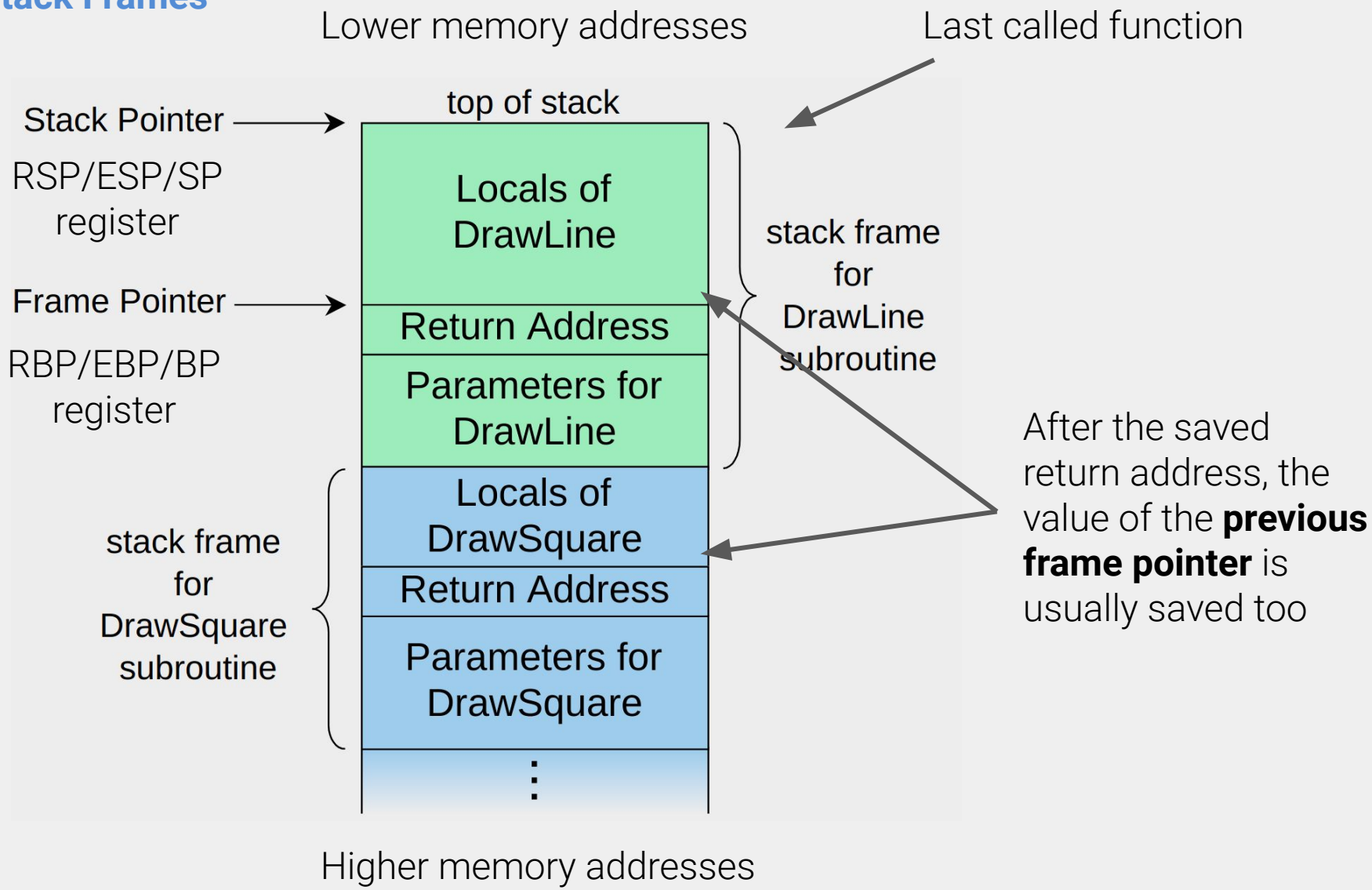
A stack frame is a data structure built in the stack that contains all the information required to execute a function. Including but not limited to:

- Return address
- Address of the previous stack frame (frame pointer)
- Function parameters
- Function variables

Other things that could be included in a stack frame:

- Stack cookies
- Exception handling pointers

Stack Frames



- **How are stack frames built?**

  - Calling Conventions

A calling convention is a definition of whom is responsible (caller or callee) to build each part of the stack frame.

There are a bunch of calling conventions:

- cdecl → common for c code
- stdcall → common for the Windows API calls
- fastcall
- thiscall → common for C++
- 64-bit calling convention for MS
- 64-bit calling convention for Unix
- (...)

- **Calling Conventions**

	cdecl	stdcall	MS x64	Unix x64	fastcall
Prologue	**Caller** pushes params to stack right to left.	**Caller** pushes params to stack right to left. (As cdecl)	**Caller** puts first 4 params in rcx, rdx, r8 and r9 and pushes the rest right to left [1].	**Caller** puts first 6 params in rdi, rsi, rdx, rcx, r8 and r9 and pushes the rest right to left.	**Caller** puts first 2 params in ecx and edx and pushes the rest right to left.
Return Value	Stored in eax.	Stored in eax.	Stored in rax.	Stored in rax.	Stored in eax.
Epilogue	**Caller** cleans up parameters.	**Callee** cleans up parameters. (“ret N”).	**Caller** cleans up the parameters. (as cdecl)	**Caller** cleans up the parameters.	**Callee** cleans up parameters. (“ret N”). (as stdcall)

There also the concept of volatile (caller-saved)/non-volatile (calle-saved) registers, meaning if either the caller/callee must make sure to save those register values (if they need to).

1 Caller will **always** save at the very least space for the 4 registers in the stack.

- **Example with cdecl**

- func(1, 2, 3) → Assume func has 2 local variables and returns 0.

Assembly will be something similar to:

Caller disassembly:

- push 3
- push 2
- push 1
- call func → pushes @ret
- add esp, 0xC

func disassembly:

- push ebp → saves previous frame pointer
- mov ebp, esp → params/var will be ref thr ebp
- sub esp, 8 → save space for local vars
- (...)
- mov eax, 0 → set return value
- leave → mov esp, ebp; pop ebp
- ret → pops saved @ret to eip

func(1, 2, 3) disassembly:

- push 3
- push 2
- push 1
- call func
- add esp, 0xC

func disassembly:

- push ebp
- mov ebp, esp
- sub esp, 8
- (...)
- mov eax, 0
- leave
- ret

Lower memory addresses

esp →

ebp →

Variable	Offset
var1	[ebp-8]
var2	[ebp-4]
saved caller ebp	[ebp]
saved ret address	[ebp+4]
1	[ebp+8]
2	[ebp+0xC]
3	[ebp-0x10]

HIgher memory addresses

A word of advice

Dynamically…

- The stack frame will not contain what you expect...
- The stack frame data won't be where you expect...

calling_conventions, x86

```
20% 225 calling_conventions_x86]> pd $r @ sym.main+4
0x000005f7      83e4f0          and esp, 0xfffffff0
0x000005fa      ff71fc          push dword [ecx - 4]
0x000005fd      55              push ebp
0x000005fe      89e5            mov ebp, esp
0x00000600      53              push ebx
0x00000601      51              push ecx
0x00000602      83ec10          sub esp, 0x10
0x00000605      e816feffff      call sym.__x86.get_pc_thunk.bx ;[1]
0x0000060a      81c3ce190000    add ebx, 0x19ce
0x00000610      c745e8000000.   mov dword [local_18h], 0
0x00000617      c745ec000000.   mov dword [local_14h], 0
0x0000061e      c745f0000000.   mov dword [local_10h], 0
0x00000625      c745f4000000.   mov dword [local_ch], 0
0x0000062c      83ec04          sub esp, 4
0x0000062f      6a03            push 3                      ; "F\x01
0x00000631      6a02            push 2
0x00000633      6a01            push 1
0x00000635      e886ffffff      call sym.func1              ;[2]
0x0000063a      83c410          add esp, 0x10
0x0000063d      8945e8          mov dword [local_18h], eax
0x00000640      83ec0c          sub esp, 0xc
0x00000643      6a06            push 6
0x00000645      ba05000000      mov edx, 5
0x0000064a      b904000000      mov ecx, 4
0x0000064f      e831ffffff      call sym.func2              ;[3]
0x00000654      83c40c          add esp, 0xc
0x00000657      8945ec          mov dword [local_14h], eax
0x0000065a      83ec04          sub esp, 4
0x0000065d      6a09            push 9
0x0000065f      6a08            push 8
0x00000661      6a07            push 7
0x00000663      e8e8feffff      call sym.func3              ;[4]
0x00000668      83c404          add esp, 4
0x0000066b      8945f0          mov dword [local_10h], eax
0x0000066e      83ec04          sub esp, 4
0x00000671      6a0c            push 0xc
0x00000673      6a0b            push 0xb
0x00000675      6a0a            push 0xa
0x00000677      e8a1feffff      call sym.func4              ;[5]
0x0000067c      83c410          add esp, 0x10
0x0000067f      8945f4          mov dword [local_ch], eax
0x00000682      8b55e8          mov edx, dword [local_18h]
0x00000685      8b45ec          mov eax, dword [local_14h]
0x00000688      01c2            add edx, eax
```

Let's identify what call convention is using each funcN

- **func1**

```
83ec04          sub esp, 4
6a03            push 3
6a02            push 2
6a01            push 1
e886ffffff      call sym.func1
83c410          add esp, 0x10
8945e8          mov dword [local_18h], eax
```

Caller pushes parameters to stack

Caller cleans parameters from stack?
Probably, given that caller doesn't do it!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec04          sub esp, 4
e8ea000000      call sym.__x86.get_pc_thunk.ax
050c1a0000      add eax, 0x1a0c
ff7510          push dword [arg_10h]
ff750c          push dword [arg_ch]
ff7508          push dword [arg_8h]
8d90ade7ffff    lea edx, [eax - 0x1853]
52              push edx
89c3            mov ebx, eax
e8c8fdffff      call sym.imp.printf
83c410          add esp, 0x10
8b4508          mov eax, dword [arg_8h]
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c3              ret
```

Callee with common prologue (2 instructions)

Callee with 2 instructions with common epilogue

- **func1 → cdecl**

```
83ec04          sub esp, 4
6a03            push 3
6a02            push 2
6a01            push 1
e886ffffff      call sym.func1
83c410          add esp, 0x10
8945e8          mov dword [local_18h], eax
```

Caller pushes parameters to stack

Caller cleans parameters from stack?
Probably, given that callee doesn't do it!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec04          sub esp, 4
e8ea000000      call sym.__x86.get_pc_thunk.ax
050c1a0000      add eax, 0x1a0c
ff7510          push dword [arg_10h]
ff750c          push dword [arg_ch]
ff7508          push dword [arg_8h]
8d90ade7ffff    lea edx, [eax - 0x1853]
52              push edx
89c3            mov ebx, eax
e8c8fdffff      call sym.imp.printf
83c410          add esp, 0x10
8b4508          mov eax, dword [arg_8h]
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c3              ret
```

Callee with common prologue (2 instructions)

Callee with 2 instructions with common epilogue

- **func2**

```
83ec0c          sub esp, 0xc
6a06            push 6
ba05000000      mov edx, 5
b904000000      mov ecx, 4
e831ffffff      call sym.func2
83c40c          add esp, 0xc
8945ec          mov dword [local_14h], eax
```

Caller pushes 1 parameter to the stack and stores 2 in register edx, ecx

Caller cleans parameters from stack? Probably not, callee cleans it!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec14          sub esp, 0x14
e825010000      call sym.__x86.get_pc_thunk.ax
05471a0000      add eax, 0x1a47
894df4          mov dword [local_ch], ecx
8955f0          mov dword [local_10h], edx
ff7508          push dword [arg_8h]
ff75f0          push dword [local_10h]
ff75f4          push dword [local_ch]
8d9096e7ffff    lea edx, [eax - 0x186a]
52              push edx
89c3            mov ebx, eax
e8fdfdffff      call sym.imp.printf             ;[
83c410          add esp, 0x10
8b45f4          mov eax, dword [local_ch]
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c20400          ret 4
```

Callee with common prologue (2 instructions)

Caller makes sure to save some room for parameter registers in case they need to be saved into stack.

Callee cleans parameter from stack

- **func2 →fastcall**

```
83ec0c          sub esp, 0xc
6a06            push 6
ba05000000      mov edx, 5
b904000000      mov ecx, 4
e831ffffff      call sym.func2
83c40c          add esp, 0xc
8945ec          mov dword [local_14h], eax
```

Caller pushes 1 parameter to the stack and stores 2 in register edx, ecx

Caller cleans parameters from stack? Probably not, callee cleans it!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec14          sub esp, 0x14
e825010000      call sym.__x86.get_pc_thunk.ax
05471a0000      add eax, 0x1a47
894df4          mov dword [local_ch], ecx
8955f0          mov dword [local_10h], edx
ff7508          push dword [arg_8h]
ff75f0          push dword [local_10h]
ff75f4          push dword [local_ch]
8d9096e7ffff    lea edx, [eax - 0x186a]
52              push edx
89c3            mov ebx, eax
e8fdfdffff      call sym.imp.printf             ;[
83c410          add esp, 0x10
8b45f4          mov eax, dword [local_ch]
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c20400          ret 4
```

Callee with common prologue (2 instructions)

Caller makes sure to save some room for parameter registers in case they need to be saved into stack.

Callee cleans parameter from stack

- **func3**

```
83ec04          sub esp, 4
6a09            push 9
6a08            push 8
6a07            push 7
e8e8feffff      call sym.func3
83c404          add esp, 4
8945f0          mov dword [local_10h], eax
```

Caller pushes 3 parameter to the stack

Caller cleans parameters from stack?
Probably not, callee cleans them!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec04          sub esp, 4
e85a010000      call sym.__x86.get_pc_thunk.ax
057c1a0000      add eax, 0x1a7c
ff7510          push dword [arg_10h]
ff750c          push dword [arg_ch]
ff7508          push dword [arg_8h]
8d907fe7ffff    lea edx, [eax - 0x1881]
52              push edx
89c3            mov ebx, eax
e838feffff      call sym.imp.printf             ;[
83c410          add esp, 0x10
8b4508          mov eax, dword [arg_8h]         ;
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c20c00          ret 0xc
```

Callee with common prologue (2 instructions)

Callee cleans parameter from stack

- **func3 → stdcall**

```
83ec04          sub esp, 4
6a09            push 9
6a08            push 8
6a07            push 7
e8e8feffff      call sym.func3
83c404          add esp, 4
8945f0          mov dword [local_10h], eax
```

Caller pushes 3 parameter to the stack

Caller cleans parameters from stack?
Probably not, callee cleans them!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec04          sub esp, 4
e85a010000      call sym.__x86.get_pc_thunk.ax
057c1a0000      add eax, 0x1a7c
ff7510          push dword [arg_10h]
ff750c          push dword [arg_ch]
ff7508          push dword [arg_8h]
8d907fe7ffff    lea edx, [eax - 0x1881]
52              push edx
89c3            mov ebx, eax
e838feffff      call sym.imp.printf             ;[
83c410          add esp, 0x10
8b4508          mov eax, dword [arg_8h]         ;
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c20c00          ret 0xc
```

Callee with common prologue (2 instructions)

Callee cleans parameter from stack

- **func4**

```
83ec04          sub esp, 4
6a0c            push 0xc
6a0b            push 0xb
6a0a            push 0xa
e8a1feffff      call sym.func4
83c410          add esp, 0x10
8945f4          mov dword [local_ch], eax
```

Caller pushes 3 parameter to the stack

Caller cleans parameters from stack?
Probably, callee does not clean them!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec04          sub esp, 4
e88d010000      call sym.__x86.get_pc_thunk.ax
05af1a0000      add eax, 0x1aaf
ff7510          push dword [arg_10h]
ff750c          push dword [arg_ch]
ff7508          push dword [arg_8h]
8d9068e7ffff    lea edx, [eax - 0x1898]
52              push edx
89c3            mov ebx, eax
e86bfeffff      call sym.imp.printf
83c410          add esp, 0x10
8b4508          mov eax, dword [arg_8h]
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c3              ret
```

Callee with common prologue (2 instructions)

Callee does not clean parameter from stack

- **func4 → cdecl**

```
83ec04          sub esp, 4
6a0c            push 0xc
6a0b            push 0xb
6a0a            push 0xa
e8a1feffff      call sym.func4
83c410          add esp, 0x10
8945f4          mov dword [local_ch], eax
```

Caller pushes 3 parameter to the stack

Caller cleans parameters from stack?
Probably, callee does not clean them!

```
55              push ebp
89e5            mov ebp, esp
53              push ebx
83ec04          sub esp, 4
e88d010000      call sym.__x86.get_pc_thunk.ax
05af1a0000      add eax, 0x1aaf
ff7510          push dword [arg_10h]
ff750c          push dword [arg_ch]
ff7508          push dword [arg_8h]
8d9068e7ffff    lea edx, [eax - 0x1898]
52              push edx
89c3            mov ebx, eax
e86bfeffff      call sym.imp.printf
83c410          add esp, 0x10
8b4508          mov eax, dword [arg_8h]
8b5dfc          mov ebx, dword [local_4h]
c9              leave
c3              ret
```

Callee with common prologue (2 instructions)

Callee does not clean parameter from stack

Final Thoughts

Compilers do a lot of stuff reverse engineers don't really care about.

As a reverse engineer, you need to see beyond that and understand what really matters.

Most of the time, that's easy. We care about API calls, strings, etc.

However, there might be times (obfuscated code, algorithms in general, etc) when you might need to look at the instructions one by one. Experience will help you ignore useless instructions.

An example with the Main function in GCC

```
(fcn) sym.main 195
  sym.main ();
          ; var int local_18h @ ebp-0x18
          ; var int local_14h @ ebp-0x14
          ; var int local_10h @ ebp-0x10
          ; var int local_ch @ ebp-0xc
          ; var int local_8h @ ebp-0x8
          ; var int local_4h @ esp+0x4
             ; DATA XREF from 0x00000406 (entry0)
          0x000005f3 b    8d4c2404        lea ecx, [local_4h]
          0x000005f7      83e4f0          and esp, 0xfffffff0
          0x000005fa      ff71fc          push dword [ecx - 4]
          0x000005fd      55              push ebp
          0x000005fe      89e5            mov ebp, esp
          0x00000600      53              push ebx
          0x00000601      51              push ecx
          0x00000602      83ec10          sub esp, 0x10
          0x00000605      e816feffff      call sym.__x86.get_pc_thunk.bx
          0x0000060a      81c3ce190000    add ebx, 0x19ce
          0x00000610      c745e8000000.   mov dword [local_18h], 0
          0x00000617      c745ec000000.   mov dword [local_14h], 0
          0x0000061e      c745f0000000.   mov dword [local_10h], 0
          0x00000625      c745f4000000.   mov dword [local_ch], 0
```

In ECX we store the value of ESP+4. **Backup** of "esp" value before modifying it.

The stack is aligned to an offset multiple of 16. Improves performance and some instructions might require this.

Saved return address is pushed using the new aligned boundary.

Common stack frame operations.

Save register values in case those registers are used inside function.

Set up EBX register to work with PIC

And, honestly, we don't really care about all this!

Homework

Homework

1. Explain how functions like strcpy are implemented with just a couple of assembly instructions.

1. Explain something (short) about assembly we didn't explain in this module.

Bibliography

Bibliography

- Simple Instructions
- Intel Instruction Set Reference Manual
- Calling Conventions

Index of Figures