Software Security

Compilation steps
ELF binary format
AMD64 Assembly introduction
Reverse Engineering

Carlo Ramponi <carlo.ramponi@unitn.it>

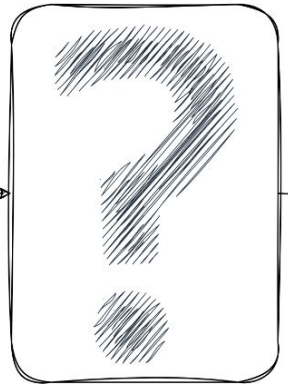
From Code to Processes

Carlo Ramponi

Compilation steps

Compilation *(i.e. black magic)* is the process of transforming high-level, human readable **source code** (*e.g. C*) **into** a machine readable, **executable file**.

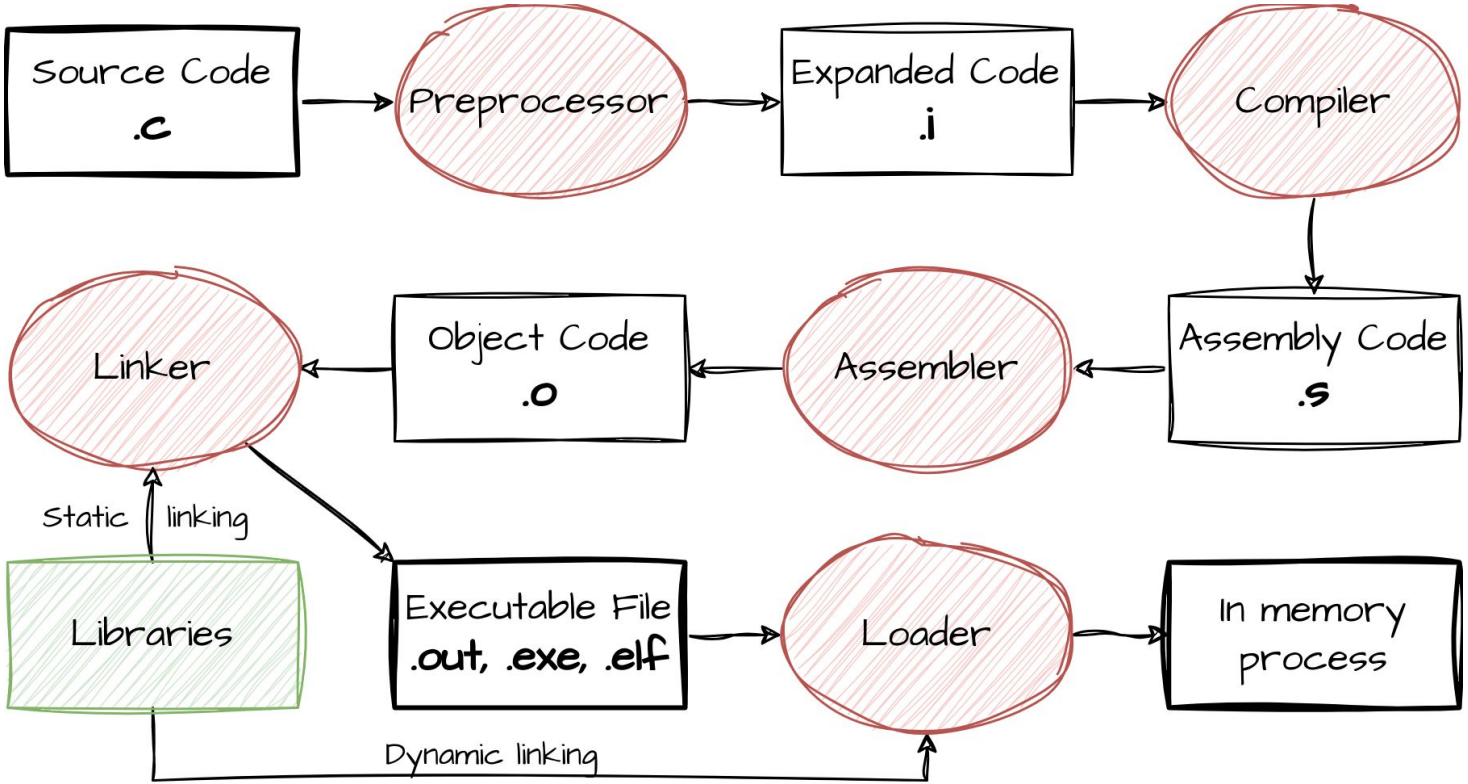
```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```



`gcc main.c`

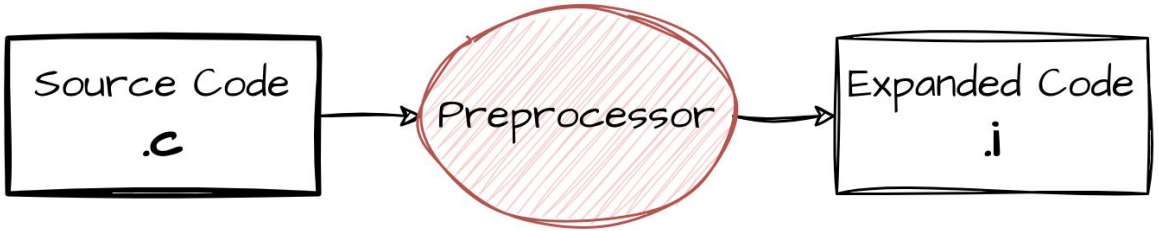
```
00001000  f3 0f 1e fa 48 83 ec 08  48 8b 05 c1 2f 00 00 48  |....H...H.../..H|
00001010  85 c0 74 02 ff d0 48 83  c4 08 c3 00 00 00 00 00  |..t...H.........|
00001020  ff 35 ca 2f 00 00 ff 25  cc 2f 00 00 0f 1f 40 00  |.5./...%./....@.|
00001030  ff 25 ca 2f 00 00 68 00  00 00 00 e9 e0 ff ff ff  |.%./..h.........|
00001040  f3 0f 1e fa 31 ed 49 89  d1 5e 48 89 e2 48 83 e4  |....1.I..^H..H..|
00001050  f0 50 54 45 31 c0 31 c9  48 8d 3d da 00 00 00 ff  |.PTE1.1.H.=.....|
00001060  15 5b 2f 00 00 f4 66 2e  0f 1f 84 00 00 00 00 00  |.[/...f.........|
00001070  48 8d 3d a1 2f 00 00 48  8d 05 9a 2f 00 00 48 39  |H.=./..H.../..H9|
00001080  f8 74 15 48 8b 05 3e 2f  00 00 48 85 c0 74 09 ff  |.t.H..>/..H..t..|
00001090  e0 0f 1f 80 00 00 00 00  c3 0f 1f 80 00 00 00 00  |................|
000010a0  48 8d 3d 71 2f 00 00 48  8d 35 6a 2f 00 00 48 29  |H.=q/..H.5j/..H)|
000010b0  fe 48 89 f0 48 c1 ee 3f  48 c1 f8 03 48 01 c6 48  |.H..H..?H...H..H|
000010c0  d1 fe 74 14 48 8b 05 0d  2f 00 00 48 85 c0 74 08  |..t.H.../..H..t.|
000010d0  ff e0 66 0f 1f 44 00 00  c3 0f 1f 80 00 00 00 00  |..f..D..........|
000010e0  f3 0f 1e fa 80 3d 2d 2f  00 00 00 75 33 55 48 83  |.....=-/...u3UH.|
000010f0  3d ea 2e 00 00 00 48 89  e5 74 0d 48 8b 3d 0e 2f  |=.....H..t.H.=./|
```

Compilation steps

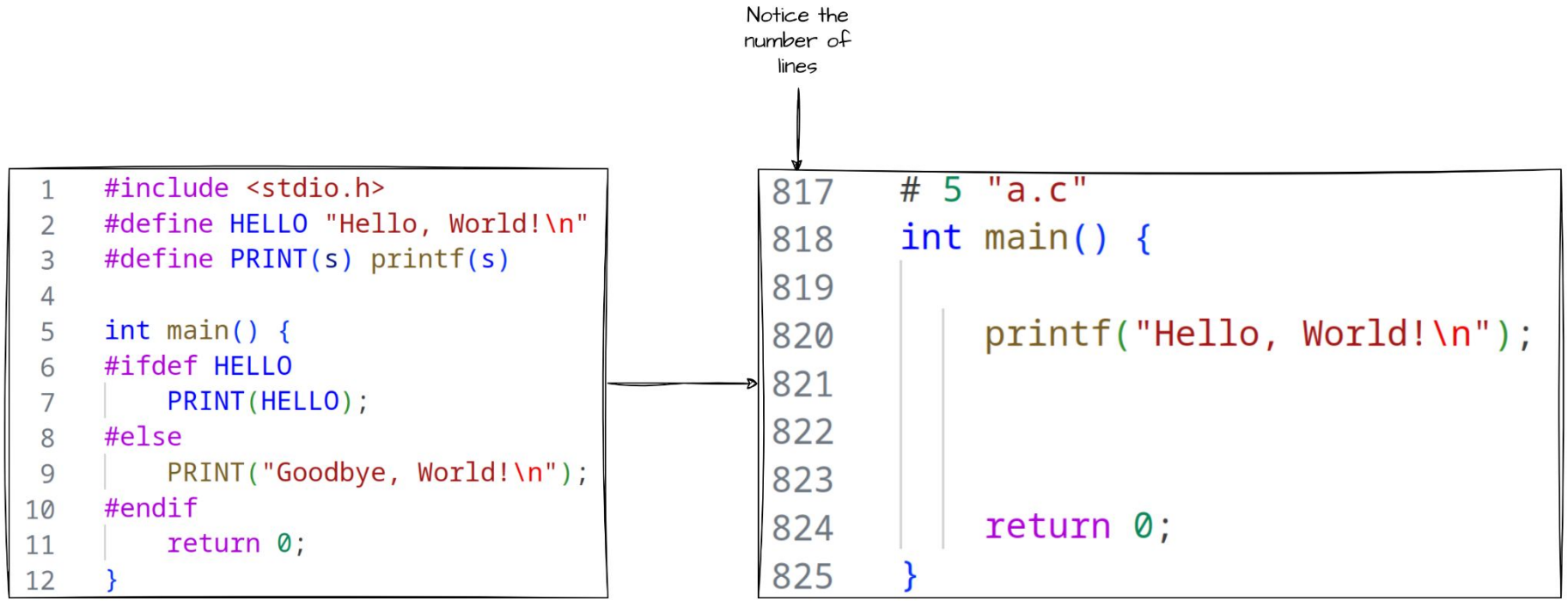


Compilation steps - Preprocessor

- **`Input`**: *Source Code (**.c**)*
- **`Output`**: *Expanded Source Code (**.i**)*
- **`Activities`**:
  - *Comments removal*
  - *Macros expansion*
  - *File inclusion*
- **`Command`**: `gcc -E main.c`

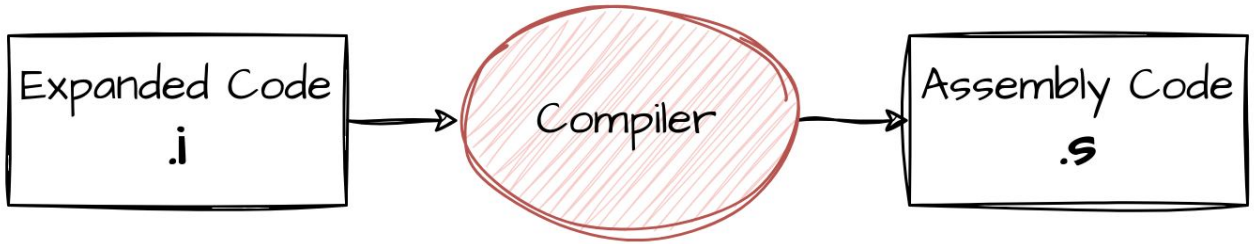


Compilation steps - Preprocessor



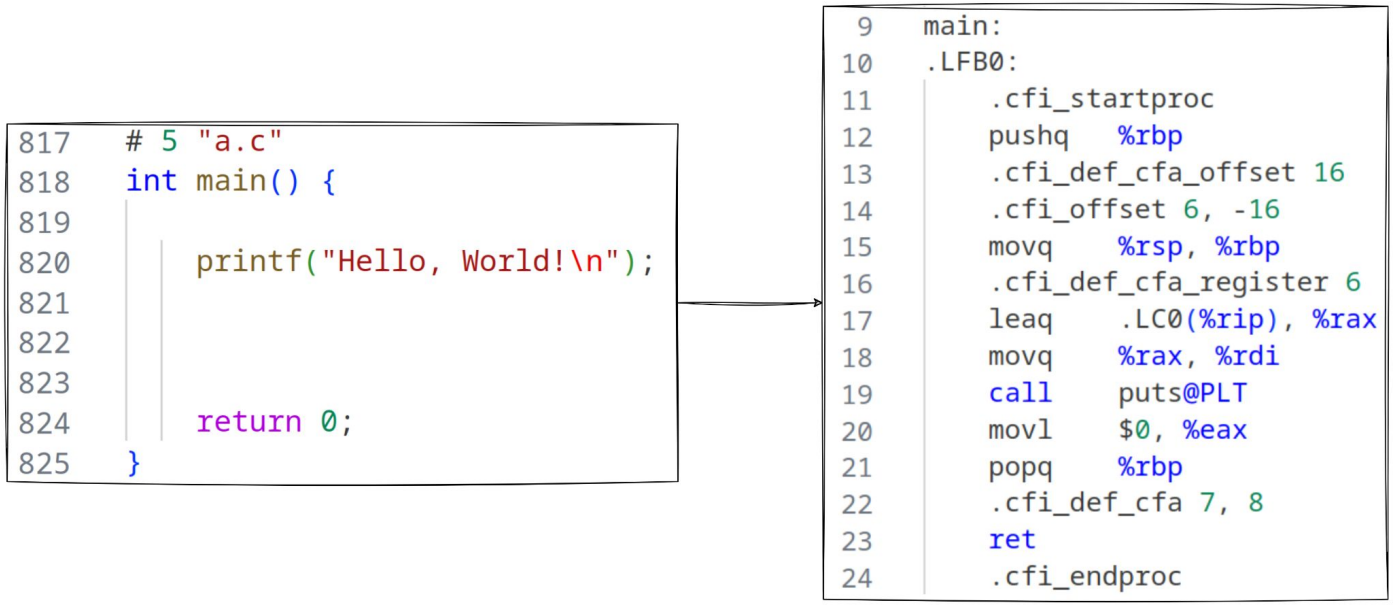
Compilation steps - Compiler

- **`Input`**: *Expanded Source Code (**.i**)*
- **`Output`**: *(Architecture Dependent) Assembly Code (**.s**)*
- **`Activity`**: *The actual compilation of the source code*
- **`Command`**: `gcc -S main.c`



***Note**: Compilers are **extremely complex**, but we won't dig into them*

Compilation steps - Compiler



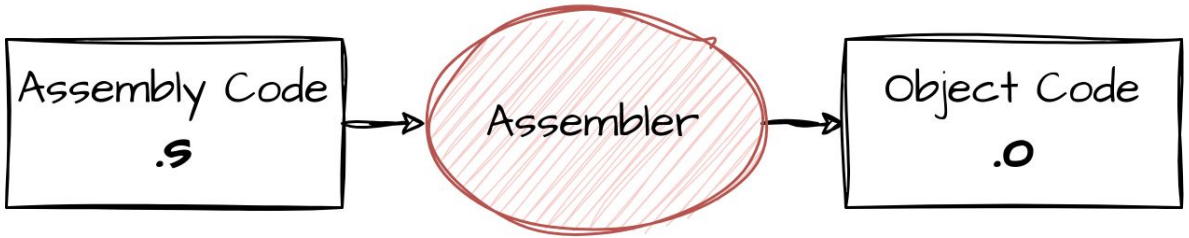
```
817  # 5 "a.c"
818  int main() {
819
820      printf("Hello, World!\n");
821
822
823
824      return 0;
825  }
```

```
 9  main:
10  .LFB0:
11      .cfi_startproc
12      pushq   %rbp
13      .cfi_def_cfa_offset 16
14      .cfi_offset 6, -16
15      movq    %rsp, %rbp
16      .cfi_def_cfa_register 6
17      leaq    .LC0(%rip), %rax
18      movq    %rax, %rdi
19      call    puts@PLT
20      movl    $0, %eax
21      popq    %rbp
22      .cfi_def_cfa 7, 8
23      ret
24      .cfi_endproc
```

Assembly code is a simple English-type language used to write **low-level instructions** in a **textual form**

Compilation steps - Assembler

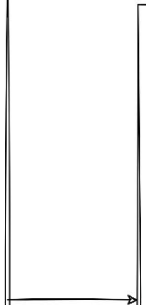
- **`Input`**: *Assembly Code (**.s**)*
- **`Output`**: *Object Code (**.o**)*
- **`Activity`**: *Conversion of assembly code into machine-understandable code*
- **`Command`**: `gcc -c main.c`



The Object code is **not ready for execution**, it still **misses external objects** such as libraries

Compilation steps - Assembler

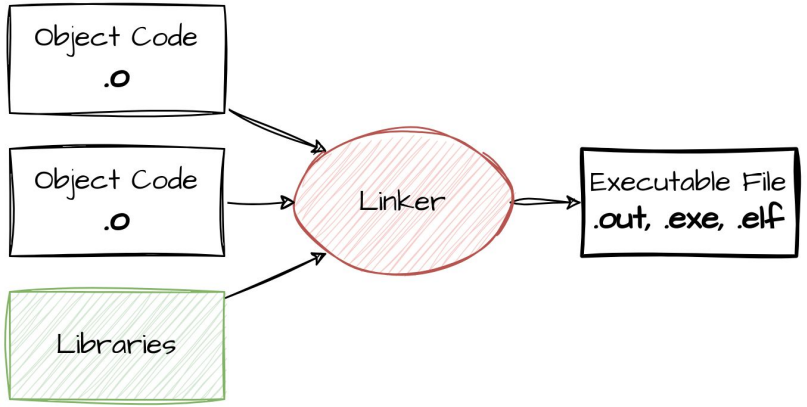
```
 9   main:
10   .LFB0:
11       .cfi_startproc
12       pushq   %rbp
13       .cfi_def_cfa_offset 16
14       .cfi_offset 6, -16
15       movq    %rsp, %rbp
16       .cfi_def_cfa_register 6
17       leaq    .LC0(%rip), %rax
18       movq    %rax, %rdi
19       call    puts@PLT
20       movl    $0, %eax
21       popq    %rbp
22       .cfi_def_cfa 7, 8
23       ret
24       .cfi_endproc
```



```
1   00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
2   00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
3   00000020  00 00 00 00 00 00 00 00  50 02 00 00 00 00 00 00  |........P.......|
4   00000030  00 00 00 00 40 00 00 00  00 00 40 00 0e 00 0d 00  |....@.....@.....|
5   00000040  55 48 89 e5 48 8d 05 00  00 00 00 48 89 c7 e8 00  |UH..H......H....|
6   00000050  00 00 00 b8 00 00 00 00  5d c3 48 65 6c 6c 6f 2c  |........].Hello,|
7   00000060  20 57 6f 72 6c 64 21 00  00 47 43 43 3a 20 28 47  | World!..GCC: (G|
8   00000070  4e 55 29 20 31 33 2e 32  2e 31 20 32 30 32 33 30  |NU) 13.2.1 20230|
9   00000080  38 30 31 00 00 00 00 00  04 00 00 00 20 00 00 00  |801........ ...|
10  00000090  05 00 00 00 47 4e 55 00  02 00 01 c0 04 00 00 00  |....GNU.........|
11  000000a0  00 00 00 00 00 00 00 00  01 00 01 c0 04 00 00 00  |................|
12  000000b0  01 00 00 00 00 00 00 00  14 00 00 00 00 00 00 00  |................|
13  000000c0  01 7a 52 00 01 78 10 01  1b 0c 07 08 90 01 00 00  |.zR..x..........|
14  000000d0  1c 00 00 00 1c 00 00 00  00 00 00 00 1a 00 00 00  |................|
15  000000e0  00 41 0e 10 86 02 43 0d  06 55 0c 07 08 00 00 00  |.A....C..U......|
16  000000f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
17  00000100  00 00 00 00 00 00 00 00  01 00 00 00 04 00 f1 ff  |................|
18  00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
19  00000120  00 00 00 00 03 00 01 00  00 00 00 00 00 00 00 00  |................|
20  00000130  00 00 00 00 00 00 00 00  00 00 00 00 03 00 05 00  |................|
```

Compilation steps - Linker

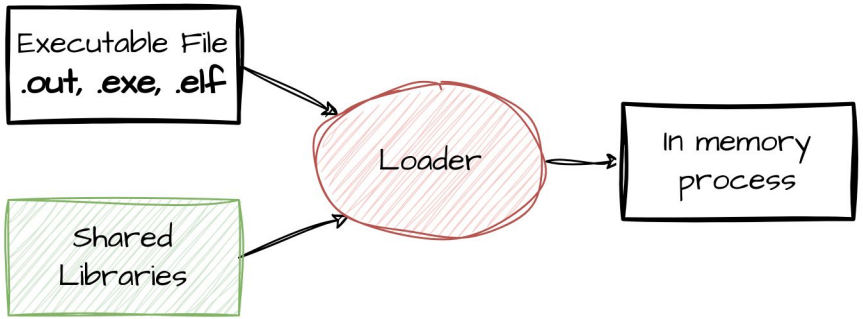
- **`Input`**: *Object Code (**.o**), Libraries (**.a**)*
- **`Output`**: *Executable File (**.out**, **.exe**, **.elf**, ...)*
- **`Activity`**: *Links multiple object files and static libraries and prepares the binary for execution*
- **`Command`**: `gcc main.c`



The **format** of the executable file **depends on the target platform**, we'll look into **ELF** executables (linux)

Compilation steps (not really part of the compilation) - Loader

- **`Input`**: *Executable file, Shared Libraries (**.so**)*
- **`Output`**: *In memory process*
- **`Activity`**: *Creates the process and links shared libraries*
- **`Command`**: `./main.out`



The **format** of the executable file **depends on the target platform**, we'll look into **ELF** executables (linux)

Static vs Dynamic Linking

Two approaches can be used in the linking phase:

1. **Static Link:**
   - Performed at **compile-time**
   - Binaries are **self-contained** and do not depend on any external libraries
   - **Bigger binaries** and possibly multiple copies of the same library in a system

2. **Dynamic Link:**
   - Performed at **load-time**
   - Binaries rely on system libraries that are loaded when needed
   - Mechanisms are needed to **dynamically relocate code**
   - **Lighter binaries** and shared libraries

ELF

Executable and Linkable Format

Carlo Ramponi

Executable and Linkable Format

The Executable and Linkable Format (ELF) is a common file format for object files.

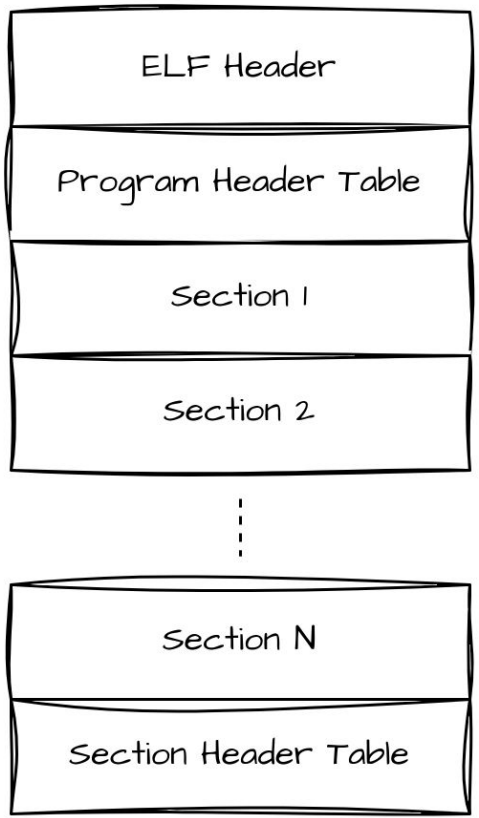
There are three types of object files:

1. **Relocatable file** containing code and data that can be linked with other object files **to create an executable or shared object file**
2. **Executable files** holding a program **suitable for execution**
3. **Shared object files** that can be:
    - linked with other relocatable and shared object files to obtain another object file
    - used by a **dynamic linker** together with other executable files and object files to create a **process image**

Executable and Linkable Format

An ELF file is structured as:

- an **ELF header**, describing the file content
- a **Program Header Table**, providing info about how to create a process image
- a sequence of **Sections**, containing what is needed for linking
- a **Section Header Table**, describing the previous sections



ELF - Header

```
$ readelf -h FILE
```

The ELF header is found at the **start of the file**, it contains **metadata** about the file

For example:

- whether the ELF file is 32-bit or 64-bit
- whether it's using **little-endian** or **big-endian**
- the ELF version
- the **architecture** that the file requires.

The metadata in the ELF header helps different processor architectures to interpret the ELF file.

See next slide →

ELF - Header


```
 carlo@carlo-pc  ~  readelf -h /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x5fa0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          136120 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

ELF - Program Header Table


```
$ readelf -l FILE
```

- The Program Header Table stores **information about segments**.
- Each segment is made up of **one or more sections**.
- The kernel uses this information at run time to **create the process** and **map the segments into memory**.

To run a program, the kernel:

- **loads the ELF header and the program header table** into memory
- loads the contents that are specified in **LOAD in the program header table**
- gives **control to the executable** itself

ELF - Program Header Table


```
carlo@carlo-pc ~ readelf -l /bin/ls
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000003638 0x0000000000003638  R      0x1000
  LOAD           0x0000000000004000 0x0000000000004000 0x0000000000004000
                 0x0000000000013401 0x0000000000013401  R E    0x1000
  DYNAMIC        0x0000000000020a78 0x0000000000021a78 0x0000000000021a78
                 0x00000000000001c0 0x00000000000001c0  RW     0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x000000000001ff70 0x000000000020f70 0x000000000020f70
                 0x0000000000001090 0x0000000000001090  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash
   03     .init .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .data.rel.ro .dynamic .got .data .bss
   06     .dynamic
```

ELF - Section Header Table


```
$ readelf -S FILE
```

- The section header stores information about sections
- It's used during dynamic link time, just before the program is executed.
- Additionally, the section header table contains information that's used by other files to find the symbolic definitions and references of the program.

***Note**: A linker links the binary file with shared libraries that it needs by loading them into memory. We'll see how this works*

ELF - Section Header Table

```
carlo@carlo-pc ~ readelf -S /bin/ls
There are 27 section headers, starting at offset 0x213b8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
    ⋮
  [10] .rela.dyn         RELA             0000000000001790  00001790
       00000000000001ea8  0000000000000018   A       6     0     8
  [11] .init             PROGBITS         0000000000004000  00004000
       000000000000001b  0000000000000000  AX       0     0     4
  [12] .text             PROGBITS         0000000000004020  00004020
       00000000000133d3  0000000000000000  AX       0     0     16
  [13] .fini             PROGBITS         00000000000173f4  000173f4
       000000000000000d  0000000000000000  AX       0     0     4
  [14] .rodata           PROGBITS         0000000000018000  00018000
       000000000000514f  0000000000000000   A       0     0     32
    ⋮
  [21] .got              PROGBITS         0000000000021c38  00020c38
       00000000000003b0  0000000000000008  WA       0     0     8
  [22] .data             PROGBITS         0000000000022000  00021000
       0000000000000278  0000000000000000  WA       0     0     32
  [23] .bss              NOBITS           0000000000022280  00021278
       0000000000001298  0000000000000000  WA       0     0     32
  [24] .comment          PROGBITS         0000000000000000  00021278
       000000000000001b  0000000000000001  MS       0     0     1
  [25] .gnu_debuglink    PROGBITS         0000000000000000  00021294
       0000000000000010  0000000000000000           0     0     4
  [26] .shstrtab         STRTAB           0000000000000000  000212a4
       000000000000010f  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

```c
struct Elf64_Shdr {
  Elf64_Word sh_name;
  Elf64_Word sh_type;
  Elf64_Xword sh_flags;
  Elf64_Addr sh_addr;
  Elf64_Off sh_offset;
  Elf64_Xword sh_size;
  Elf64_Word sh_link;
  Elf64_Word sh_info;
  Elf64_Xword sh_addralign;
  Elf64_Xword sh_entsize;
};
```

ELF - Sections

```
$ readelf -x SECTION_NAME FILE
```

Sections hold the actual content of the executable, e.g. code, data, …
Some common sections are the following:

- **.text**: code
- **.data**: initialised data
- **.rodata**: initialised read-only data
- **.bss**: uninitialized data
- **.plt**: PLT (Procedure Linkage Table)
- **.got**: GOT entries dedicated to dynamically linked global variables
- **.got.plt**: GOT entries dedicated to dynamically linked functions

ELF - Sections


```
 carlo@carlo-pc  ~  readelf -x .rodata /bin/ls | head

Hex dump of section '.rodata':
  0x00018000 01000200 cdcccc3d 6666663f cdcc8c3f .......=fff?...?
  0x00018010 0000803f 0000805f 0000005f 00002041 ...?..._..._.. A
  0x00018020 cdcc4c3f 00000000 00000000 00000000 ..L?............
  0x00018030 00000000 00000000 00000000 00000000 ................
  0x00018040 70e2feff 70e2feff 70e2feff 70e2feff p...p...p...p...
  0x00018050 70e2feff 70e2feff 70e2feff 70e2feff p...p...p...p...
  0x00018060 40e1feff 40e1feff 40e1feff 40e1feff @...@...@...@...
  0x00018070 40e1feff 40e1feff 40e1feff 7ee2feff @...@...@...~...
 carlo@carlo-pc  ~  readelf -x .text /bin/ls | head

Hex dump of section '.text':
  0x00004020 ff156adc 0100ff15 64dc0100 ff155edc ..j.....d.....^.
  0x00004030 0100ff15 58dc0100 ff1552dc 0100ff15 ....X.....R.....
  0x00004040 4cdc0100 ff1546dc 0100660f 1f440000 L.....F...f..D..
  0x00004050 f30f1efa 41574156 41554154 55534881 ....AWAVAUATUSH.
  0x00004060 ecf80000 00488b1e 897c2410 48893424 .....H...|$.H.4$
  0x00004070 64488b04 25280000 00488984 24e80000 dH..%(...H..$...
  0x00004080 0031c048 85db0f84 e21d0000 be2f0000 .1.H........../..
  0x00004090 004889df ff1516dd 01004889 c54885c0 .H........H..H..
```

ELF - From Disk to Memory

When a **program is started**, it becomes a **process**, and a **memory image** will be generated by the kernel, reading the ELF executable.

Segments, that contain one or more sections, will be **copied in memory**.

Address, size and permissions will be taken from the ELF headers.

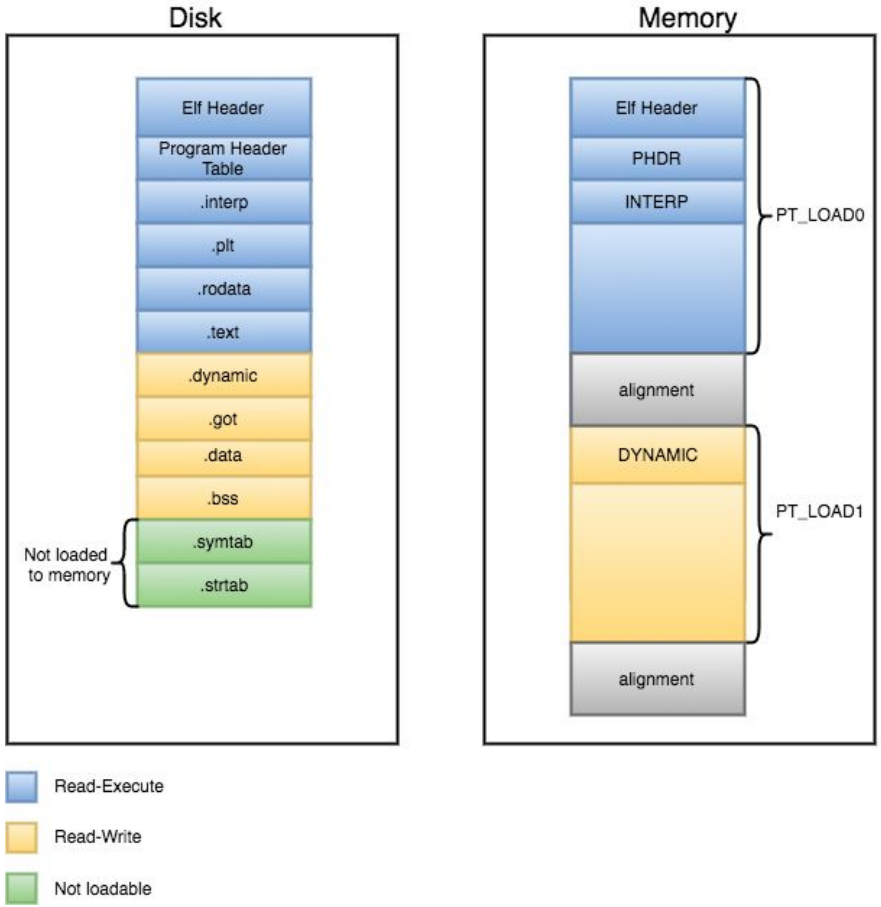


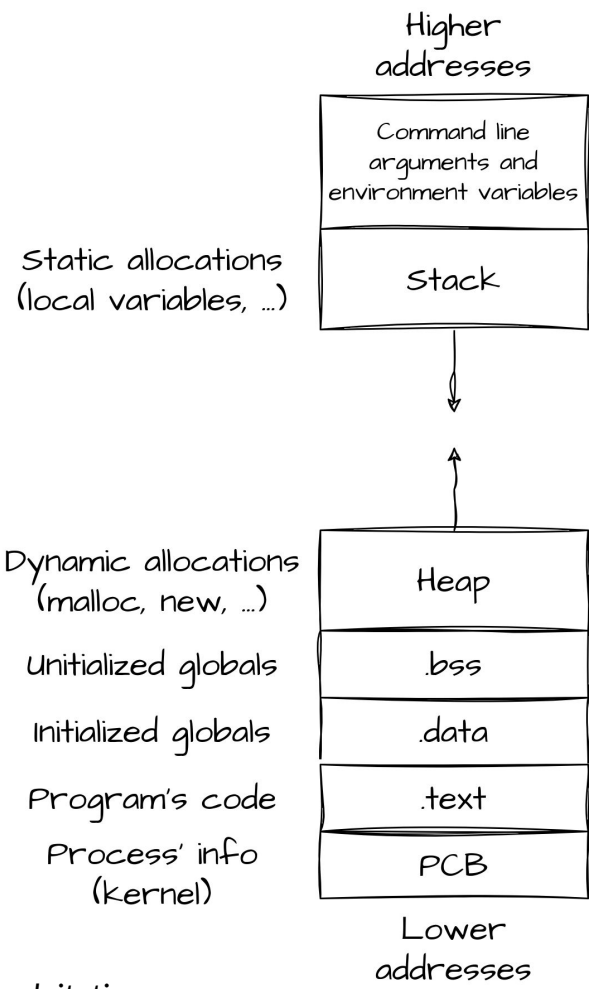
Figure from: https://intezer.com/blog/research/executable-linkable-format-101-part1-sections-segments/

Process Memory

When a program is started, it becomes a **process**, and the **kernel reserves some memory** for it, following a structure that is defined by the ELF executable.

ELF **segments** are mapped to **memory regions**.
Some regions have a **variable size** (stack, heap)

Note**: thanks to **virtual memory**, a process will use the **entire address space



We'll see how the different regions are managed when we'll deal with binary exploitation.

Encoding & Binary Format

Carlo Ramponi

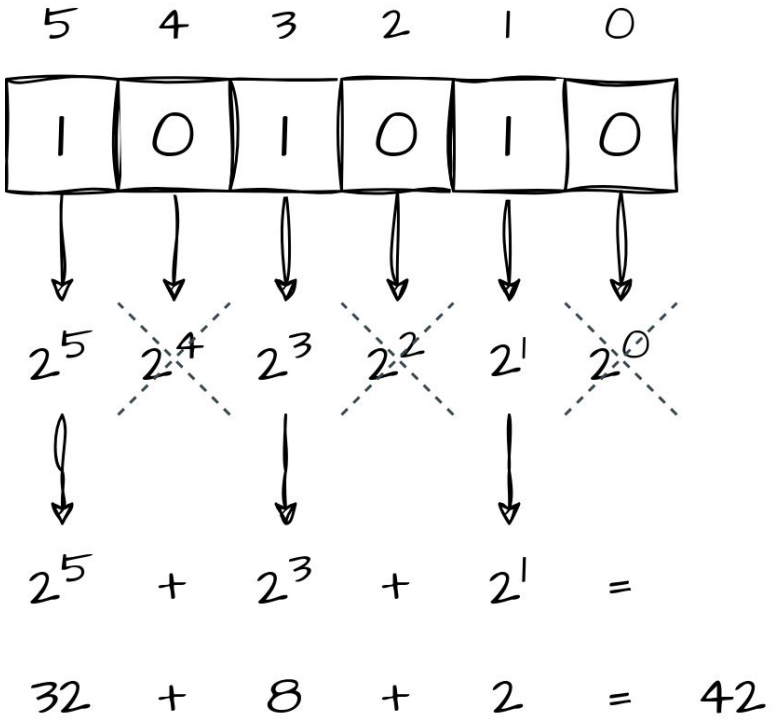
Encoding & Binary Format

Everything inside a Computer is represented using bits (`0` / `1`), but how?

Encoding numbers is trivial (*is it tho?*), we can apply the mathematical conversion from decimal to binary

No one:

Binary:



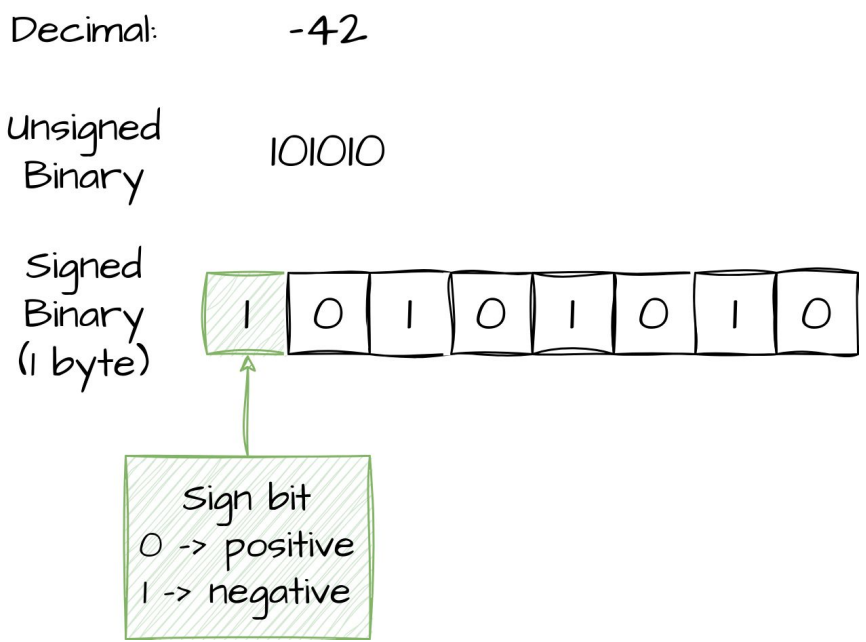
Encoding & Binary Format - Negative numbers

The first solution that comes to mind is **prepending** (or appending) **a sign bit**, like we do for the decimal form (prepending +/-).

This creates **some problems** in low-level operations, such as a **sum**, in which you have to first check the sign and then behave in different ways

2 distinct representations of zero:

- `0000 0000 → +0`
- `1000 0000 → −0`

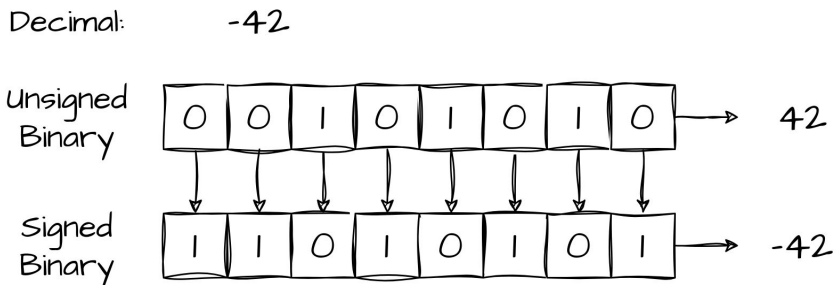


Encoding & Binary Format - 1's complement

Another way to encode negative number is **one's complement** which consists in flipping every bit of the corresponding positive number

2 distinct representations of zero:

- `0000 0000` → `+0`
- `1111 1111` → `−0`



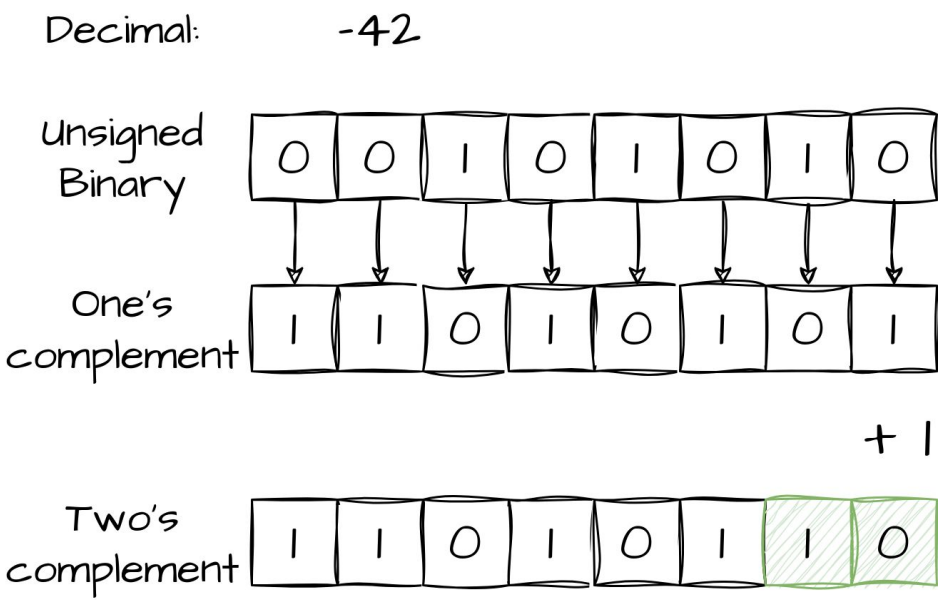
Encoding & Binary Format - 2's complement

The most used way of representing signed numbers is **two's complement**:

1. Take the unsigned binary number
2. Compute its one's complement
3. Add 1

Only one representation of zero and arithmetic operations are easily implemented

*This we'll be useful for **Integer Overflows***



Encoding & Binary Format - 2's complement


```
 carlo@carlo-pc  ~  readelf -h /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x5fa0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          136120 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

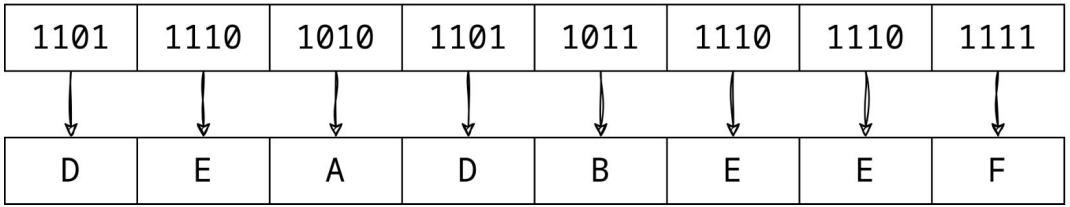
We'll see this later!

Encoding & Binary Format - Hexadecimal

Hexadecimal (aka base 16) is very useful to represent binary numbers in a **more compact way**.

Why not using decimal?
Well, **10 is not a power of 2**, while 16 is, and this makes the **conversion from binary to hexadecimal trivial**

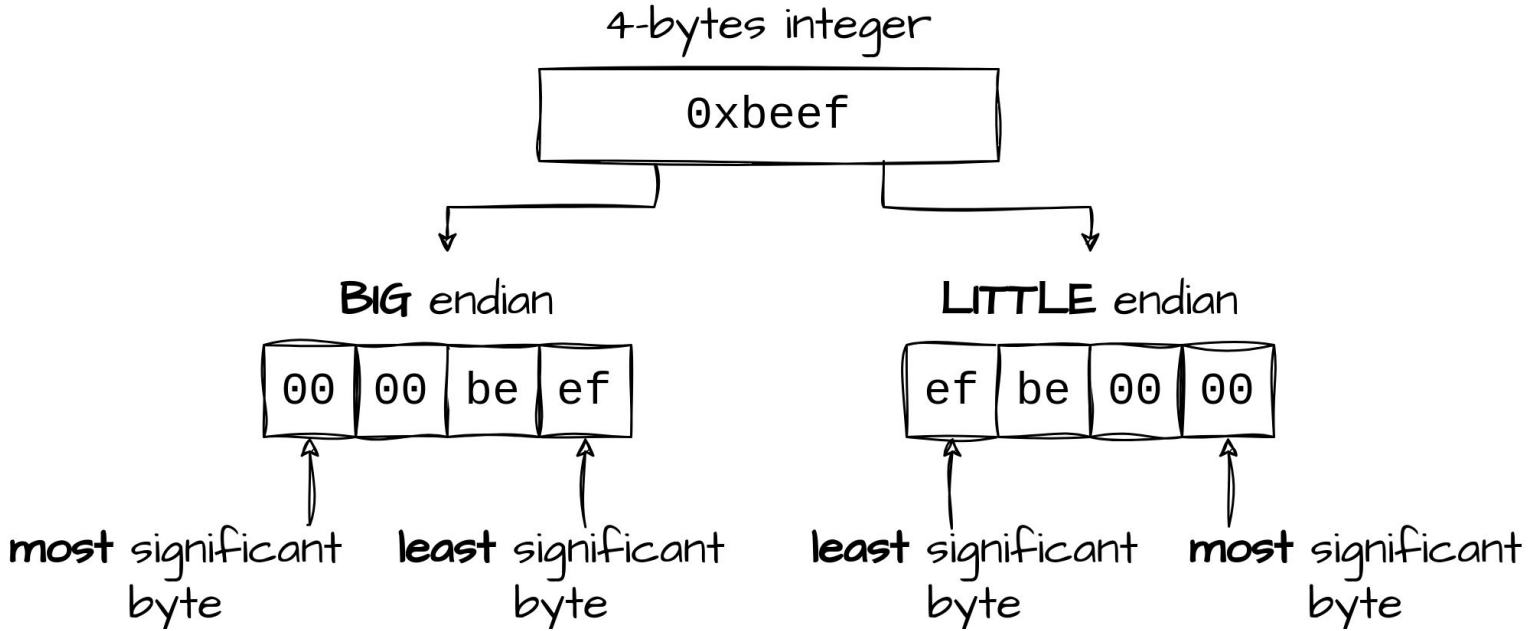
$3735928559_{10} = 1101\ 1110\ 1010\ 1101\ 1011\ 1110\ 1110\ 1111_2$



Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Encoding & Binary Format - Endianness

i.e. in which order bytes are transmitted or stored



Note: You'll most likely always deal with little-endian

Encoding & Binary Format - Endianness


```
 carlo@carlo-pc  ~  readelf -h /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x5fa0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          136120 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

Encoding & Binary Format - ASCII

Enough with numbers, how can we encode text?

The simplest way is to map every character to a number and the use the classical binary encoding

One of these mappings is ASCII, in which every character is 1-byte long (256 possible characters)

Other encodings are, for example UTF-8

```
carlo@carlo-pc  ~  man ascii
   2 3 4 5 6 7       30 40 50 60 70 80 90 100 110 120
  ------------      --------------------------------
0:   0 @ P ` p     0:    (  2  <  F  P  Z  d   n   x
1: ! 1 A Q a q     1:    )  3  =  G  Q  [  e   o   y
2: " 2 B R b r     2:    *  4  >  H  R  \  f   p   z
3: # 3 C S c s     3: !  +  5  ?  I  S  ]  g   q   {
4: $ 4 D T d t     4: "  ,  6  @  J  T  ^  h   r   |
5: % 5 E U e u     5: #  -  7  A  K  U  _  i   s   }
6: & 6 F V f v     6: $  .  8  B  L  V  `  j   t   ~
7: ' 7 G W g w     7: %  /  9  C  M  W  a  k   u   DEL
8: ( 8 H X h x     8: &  0  :  D  N  X  b  l   v
9: ) 9 I Y i y     9: '  1  ;  E  O  Y  c  m   w
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

Encoding & Binary Format - ASCII

ASCII also encodes non-printable characters some examples are:

ASCII value	Character	Description
0 (0x00)	NUL ‘\0’	Null character, usually marks the end of a string (in C)
8 (0x08)	BS ‘\b’	Backspace, deletes the previous character
9 (0x09)	HT ‘\t’	Horizontal TAB
10 (0x0A)	LF ‘\n’	New line

AMD64 Assembly

also called x86-64, x64

Carlo Ramponi

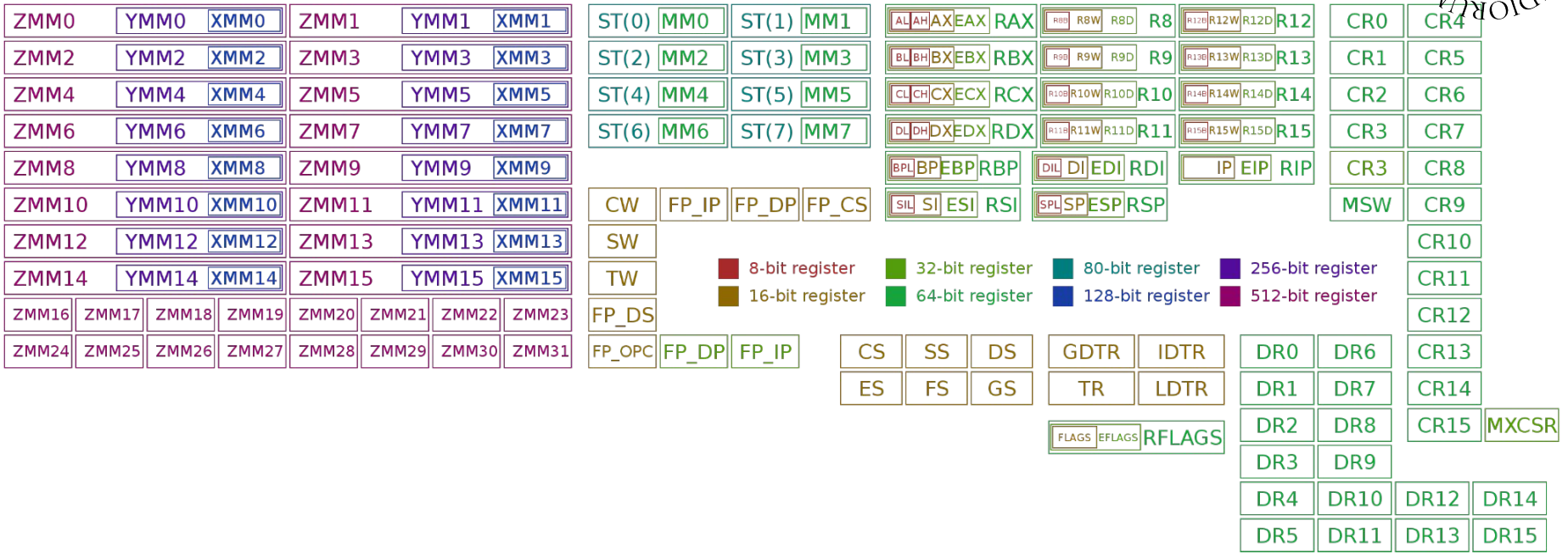
AMD64 Architecture overview

A short introduction of the AMD64 Instruction Set Architecture (ISA).

- **x86 instructions** (integer instructions) ← our focus:
  - originally 16 Bit, later extended to 32 (**IA-32**) and 64 Bit (**AMD64**)
  - This is why a **WORD** is **16-bits** long (2 bytes)
- **FPU**: Floating Point Instructions
  - 32, 64 and 80 bits
- **SIMD** extensions (Single Instruction, Multiple Data):
  - Registers: MMX (64 bit), SSE (128 bit), AVX (256 bit)

For a detailed description, head to the doc:
[official]: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[summarized]: https://www.felixcloutier.com/x86/
[summarized]: https://researcher111.github.io/uva-cso1-F23-DG/readings/x86.html

AMD64 ISA - Registers

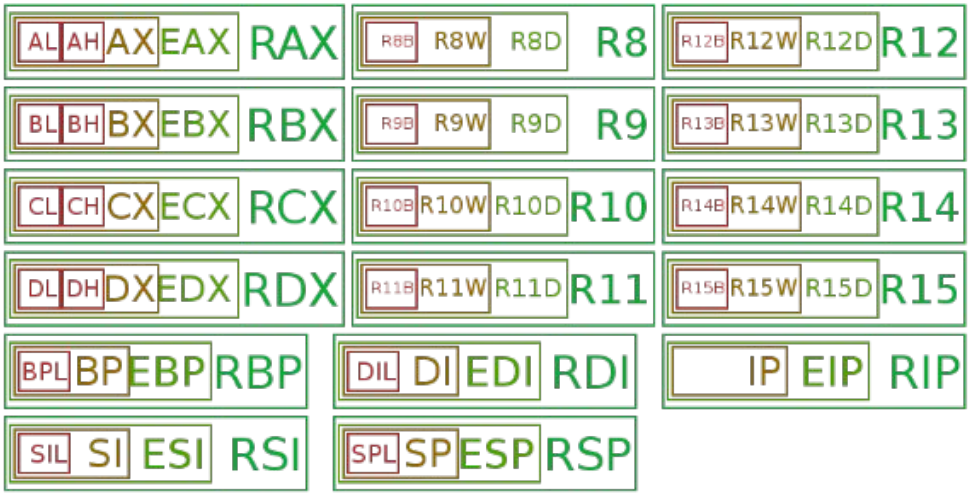


A lot of registers, we'll focus on the General Purpose Registers (**GPR**s)

AMD64 ISA - GPRs

- 14 General Purpose Registers (**GPR**s)
- Base Pointer: **RBP**
- Stack Pointer: **RSP**
- Instruction Pointer (Program Counter): **RIP**

64-bit registers can be accessed using the smaller versions (backward compatibility), e.g.
RAX → EAX → AX → AL/AH



AMD64 ISA - Data movement instructions

- **`mov op1, op2`** - copies the data item referred to by **`op2`** into the location referred to by **`op1`**
- **`push op1`** - places its operand onto the top of the stack
  - equivalent to **`rsp -= 8; memory[rsp] = op1`**
- **`pop op1`** - removes the 8-byte data element from the top of the stack and places it into **`op1`**
  - equivalent to **`op1 = memory[rsp]; rsp += 8`**
- **`lea op1,op2`** - load the memory address indicated by **`op2`** into the register specified by **`op1`**

AMD64 ISA - Arithmetic and Logic Instructions

- **`add op1,op2`**: stores in **`op1`** the result of **`op2+op1`**
- **`sub op1,op2`**: stores in **`op1`** the result of **`op2−op1`**

- **`and op1,op2`**
- **`or op1,op2`**
- **`xor op1,op2`**
- **`...`**

} Perform the specified logical operation on the operands, storing the result in the first operand location

AMD64 ISA - Control Flow Instructions

- **`jmp op`**: jump to the instruction at the memory location specified by the operand **`op`**
- **`cmp op1,op2`**: compares the values of the two specified operands and stores the result in the machine status register
- **`j<condition> op`**: depending on the **`<condition>`** and on the context of machine status register, jumps to instruction at the memory location indicated by the operand
- **`call func`**: push the address of the next instruction and jump to **`func`**
  - **arguments** are prepared **before the call**, following a calling convention
  - equivalent to: **`push rip+4; jmp func`**
- **`ret`**: return to the caller
  - equivalent to: **`pop rip`** ← get the return address from the stack

AMD64 ISA - Addressing modes

Operands of most operations may be either:

- a register
- an immediate value
- or the **contents of memory**

A **memory address** in general is made of an **immediate**, **two registers**, and **a scale** on one of the registers:

```
imm + rA + rB*s
```

where **s** is one of the four specific values 1, 2, 4, or 8.

AMD64 ISA - Assembly syntax(es)

For mostly historical reasons, AMD64 has **two different syntaxes**.

Feature	Intel syntax	AT&T syntax
Register	`rsp`	`%rsp`
Immediate	`23`	`$23`
Reg+Imm Addr	`[rsp+23]`	`23(%rsp)`
R+R*4+Imm Addr	`[rsp+r8*4+23]`	`23(%rsp,%r8,4)`
a += b	`add rax,rbx`	`addq %rbx, %rax`

We'll refer to **Intel syntax**

AMD64 ISA - Assembly syntax(es)

- In general, **AT&T syntax is more explicit**, there are prefixes for types, operations have width suffixes, etc.
  - destination is the last operand
- **Intel syntax**, on the other hand, **is more loose**, and has to add things like **`QWORD PTR`** if the instructions operands do not make the width of a command obvious.
  - destination is the first operand

Intel syntax	**AT&T syntax**
`mov` **`QWORD PTR`** `[rdx+0x227],rax`	`mov`**`q`** `%rax,0x227(%rdx)`
`mov rax, 0`	`mov`**`q`** `$0, %rax`

AMD64 ISA - Assembly syntax(es)

Width specifiers:

bits	historical name	Intel name	AT&T Suffix	register names
8	byte	**BYTE**	**b**	ah, al, r9b, …
16	word, as this was the native size of the 8086 processor	**WORD**	**w**	ax, r9w, …
32	double word	**DWORD**	**l**	eax, r9d, …
64	quad word	**QWORD**	**q**	rax, r9, …

AMD64 ISA - Example

```
1   foo:
2       push    rbp
3       mov rbp, rsp
4       mov DWORD PTR -4[rbp], 0
5       jmp .L2
6   .L3:
7       mov eax, DWORD PTR -4[rbp]
8       lea edx, [rax+rax]
9       mov eax, DWORD PTR -4[rbp]
10      cdqe
11      mov DWORD PTR -48[rbp+rax*4], edx
12      add DWORD PTR -4[rbp], 1
13  .L2:
14      cmp DWORD PTR -4[rbp], 9
15      jle .L3
16      pop rbp
17      ret
```

What does this function do?

AMD64 ISA - Example

```
1   foo:
2       push    rbp
3       mov rbp, rsp
4       mov DWORD PTR -4[rbp], 0
5       jmp .L2
6   .L3:
7       mov eax, DWORD PTR -4[rbp]
8       lea edx, [rax+rax]
9       mov eax, DWORD PTR -4[rbp]
10      cdqe
11      mov DWORD PTR -48[rbp+rax*4], edx
12      add DWORD PTR -4[rbp], 1
13  .L2:
14      cmp DWORD PTR -4[rbp], 9
15      jle .L3
16      pop rbp
17      ret
```

Function's prologue

Function's epilogue

We'll see later what a function prologue and epilogue are

AMD64 ISA - Example

```
1   foo:
2       push    rbp
3       mov rbp, rsp
4       mov DWORD PTR -4[rbp], 0
5       jmp .L2
6   .L3:
7       mov eax, DWORD PTR -4[rbp]
8       lea edx, [rax+rax]
9       mov eax, DWORD PTR -4[rbp]
10      cdqe
11      mov DWORD PTR -48[rbp+rax*4], edx
12      add DWORD PTR -4[rbp], 1
13  .L2:
14      cmp DWORD PTR -4[rbp], 9
15      jle .L3
16      pop rbp
17      ret
```

Set variable at **rbp-4** to **0**

Compare variable at **rbp-4** with **9**

Jump to L3 if the previous **cmp** resulted in **less than or equal**

This must be a loop!
Something similar to:

```
1  void foo() {
2      for(int i = 0; i < 10; i++) {
3          // do things
4      }
5  }
```

AMD64 ISA - Example

```
1   foo:
2       push    rbp
3       mov rbp, rsp
4       mov DWORD PTR -4[rbp], 0
5       jmp .L2
6   .L3:
7       mov eax, DWORD PTR -4[rbp]
8       lea edx, [rax+rax]
9       mov eax, DWORD PTR -4[rbp]
10      cdqe
11      mov DWORD PTR -48[rbp+rax*4], edx
12      add DWORD PTR -4[rbp], 1
13  .L2:
14      cmp DWORD PTR -4[rbp], 9
15      jle .L3
16      pop rbp
17      ret
```

Let's call **i** the variable stored in **rbp-4**

Load **i** in **eax** (32 bit)

edx = rax + rax

Load **i** in **eax** (32 bit)

Store **edx** in **rbp-48** + **rax*4**

i += 1

Another local variable, probably an array

The loop body is writing something at address:

rbp-48 + i * 4

rbp-48 probably points to an array of integers (4-byte values)

AMD64 ISA - Example

The original code was:

```
1   void foo() {
2       int num[10];
3       for(int i = 0; i < 10; i++) {
4           num[i] = i * 2;
5       }
6   }
```

What we just did is called **Reverse Engineering** and consists of analyzing the binary code to understand how the source code was structured

Reverse Engineering

Carlo Ramponi

Reverse Engineering

Reverse engineering (also known as **backwards engineering** or **back engineering**) is a **process** or method through which one **attempts to understand** through deductive reasoning how a previously made device, process, system, or piece of software accomplishes a task **with very little** (if any) **insight** into exactly how it does so.

Software Reverse Engineering (SRE)

“the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction” [1]

Uses: **malware analysis**, modifying software (e.g. ROM hacking), software **cracking**, …

[1]: https://en.wikipedia.org/wiki/Reverse_engineering

SRE - Gathering information from binary files

Given a binary file, we can:

- check if the file is **executable** or not (don't think about the execute permission)
- discover the **architecture** for which the binary has been compiled
- collect **symbols and strings** used in the program
- identify **function names** and used **libraries**

Some useful tools for this tasks are **`file`**, **`strings`**, **`readelf`**, **`objdump`**, ...

SRE - Gathering information from binary files

The **`file`** command shows some information about the content of a file

```
 carlo@carlo-pc  ~/Desktop/EH24/Code  file *
asm_es:   ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
 linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=e336da319f47cd
7c74700ffa7b3fa404d01b310f, for GNU/Linux 4.4.0, not stripped
asm_es.c: C source, ASCII text
asm_es.s: assembler source, ASCII text
Makefile: makefile script, ASCII text
```

SRE - Gathering information from binary files


```
carlo@carlo-pc ~/Desktop/EH24/Code strings asm_es
/lib64/ld-linux-x86-64.so.2
puts
__libc_start_main
__cxa_finalize
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u3UH
Hello World!
;*3$"
GCC: (GNU) 13.2.1 20230801
asm_es.c
_DYNAMIC
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
__libc_start_main@GLIBC_2.34
_ITM_deregisterTMCloneTable
```

The loader program to be used

Library functions

Shared Library

Other symbols

Garbage

A string used by the program

The **`strings`** command prints any sequence of printable characters from a file

You can specify the minimum length with the **`-n`** parameter

Useful to find cleartext, hardcoded secrets

SRE - Gathering information from binary files


```
 carlo@carlo-pc  ~/Desktop/EH24/Code  readelf -h /bin/ls
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x5fa0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          136120 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

We've already seen **`readelf`**: it shows the structured content of an ELF file

SRE - Gathering information from binary files


```
carlo@carlo-pc ~/Desktop/EH24/Code objdump -s -j .text ./asm_es

./asm_es:     file format elf64-x86-64

Contents of section .text:
 1040 f30f1efa 31ed4989 d15e4889 e24883e4  ....1.I..^H..H..
 1050 f0505445 31c031c9 488d3d04 010000ff  .PTE1.1.H.=.....
 1060 155b2f00 00f4662e 0f1f8400 00000000  .[/...f.........
 1070 488d3da1 2f000048 8d059a2f 00004839  H.=./..H.../..H9
 1080 f8741548 8b053e2f 00004885 c07409ff  .t.H..>/..H..t..
 1090 e00f1f80 00000000 c30f1f80 00000000  ................
 10a0 488d3d71 2f000048 8d356a2f 00004829  H.=q/..H.5j/..H)
 10b0 fe4889f0 48c1ee3f 48c1f803 4801c648  .H..H..?H...H..H
 10c0 d1fe7414 488b050d 2f000048 85c07408  ..t.H.../..H..t.
 10d0 ffe0660f 1f440000 c30f1f80 00000000  ..f..D..........
 10e0 f30f1efa 803d2d2f 00000075 33554883  .....=-/...u3UH.
 10f0 3dea2e00 00004889 e5740d48 8b3d0e2f  =.....H..t.H.=./
 1100 0000ff15 d82e0000 e863ffff ffc60504  .........c......
 1110 2f000001 5dc3662e 0f1f8400 00000000  /...].f.........
 1120 c366662e 0f1f8400 00000000 0f1f4000  .ff...........@.
 1130 f30f1efa e967ffff ff554889 e5c745fc  .....g...UH...E.
 1140 00000000 eb138b45 fc8d1400 8b45fc48  .......E.....E.H
 1150 98895485 d08345fc 01837dfc 097ee790  ..T...E...}..~..
 1160 905dc355 4889e5b8 00000000 e8c8ffff  .].UH...........
 1170 ff488d05 8c0e0000 4889c7e8 b0feffff  .H......H.......
 1180 b8000000 005dc3                      .....].
```

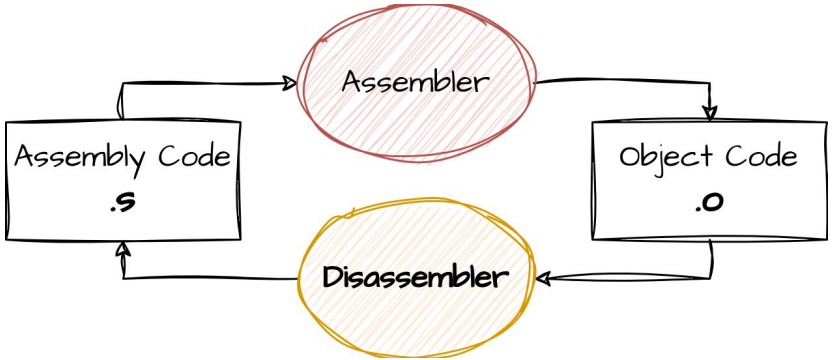
`objdump`, similarly to **`readelf`**, shows the content of binary file, supporting **multiple binary formats**

Can also be used as a a **disassembler!**

SRE - Disassembler

A **disassembler** is a computer program that **translates machine language into assembly language** — the inverse operation to that of an assembler.

Disassembly, the output of a disassembler, is often formatted for human-readability rather than suitability for input to an assembler, making it principally a reverse-engineering tool.



SRE - Disassembler


```
carlo@carlo-pc ~/Desktop/EH24/Code objdump --disassemble=foo -j .text -M intel ./asm_es

./asm_es:     file format elf64-x86-64


Disassembly of section .text:

0000000000001139 <foo>:
    1139:       55                      push   rbp
    113a:       48 89 e5                mov    rbp,rsp
    113d:       c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
    1144:       eb 13                   jmp    1159 <foo+0x20>
    1146:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
    1149:       8d 14 00                lea    edx,[rax+rax*1]
    114c:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
    114f:       48 98                   cdqe
    1151:       89 54 85 d0             mov    DWORD PTR [rbp+rax*4-0x30],edx
    1155:       83 45 fc 01             add    DWORD PTR [rbp-0x4],0x1
    1159:       83 7d fc 09             cmp    DWORD PTR [rbp-0x4],0x9
    115d:       7e e7                   jle    1146 <foo+0xd>
    115f:       90                      nop
    1160:       90                      nop
    1161:       5d                      pop    rbp
    1162:       c3                      ret
```

SRE - Disassembler

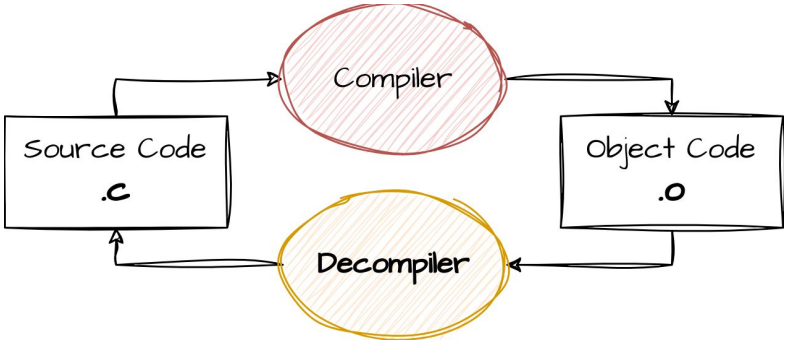
Other tools that can be used as disassemblers are:

- The GNU Project Debugger (**GDB**)
- python-capstone
- python-pwntools
- Radare2
- Ghidra ← also a **decompiler**
- …

SRE - Decompiler

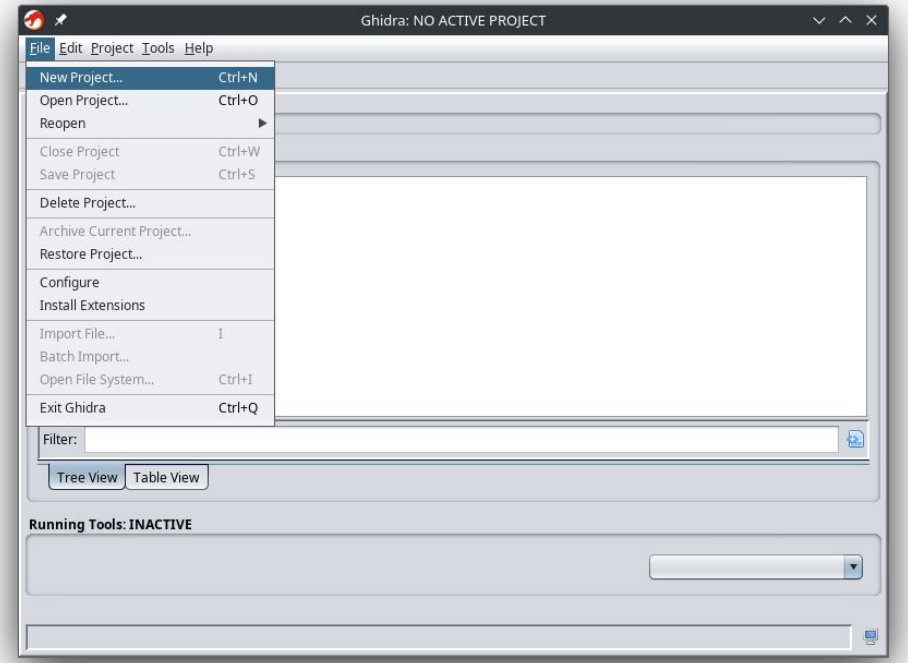
A **decompiler** is a computer program that translates an **executable file** to **high-level source code**.

- The **opposite of a compiler**
- Requires more **sophisticated techniques** than a disassembler.
- Usually **unable** to **perfectly reconstruct** the original **source code**
- Essential tool in the reverse engineering of computer software.

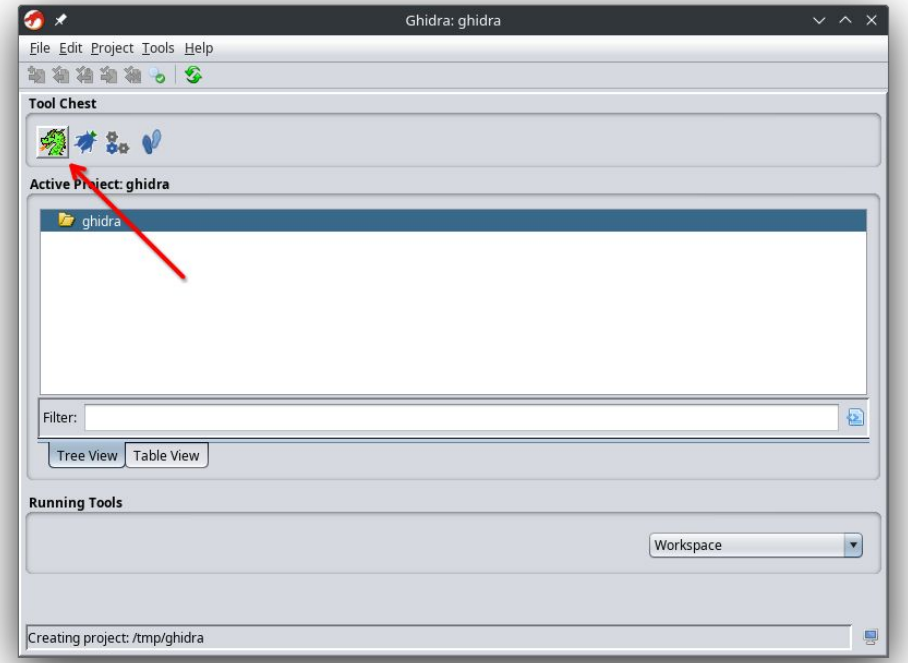


We'll use the open-source decompiler **Ghidra**, from **NSA**

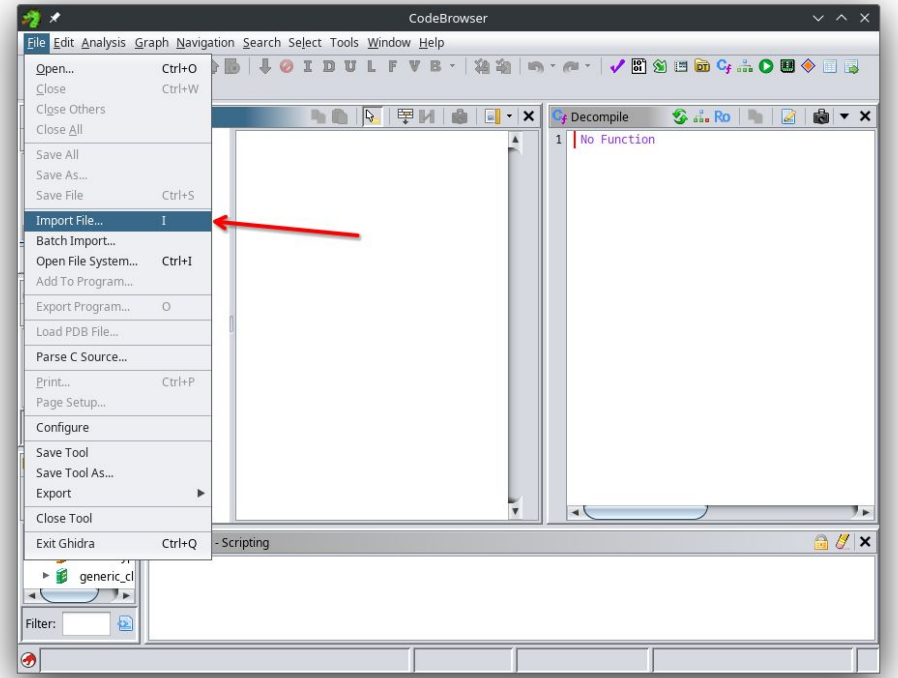
SRE - Decompiler - Ghidra



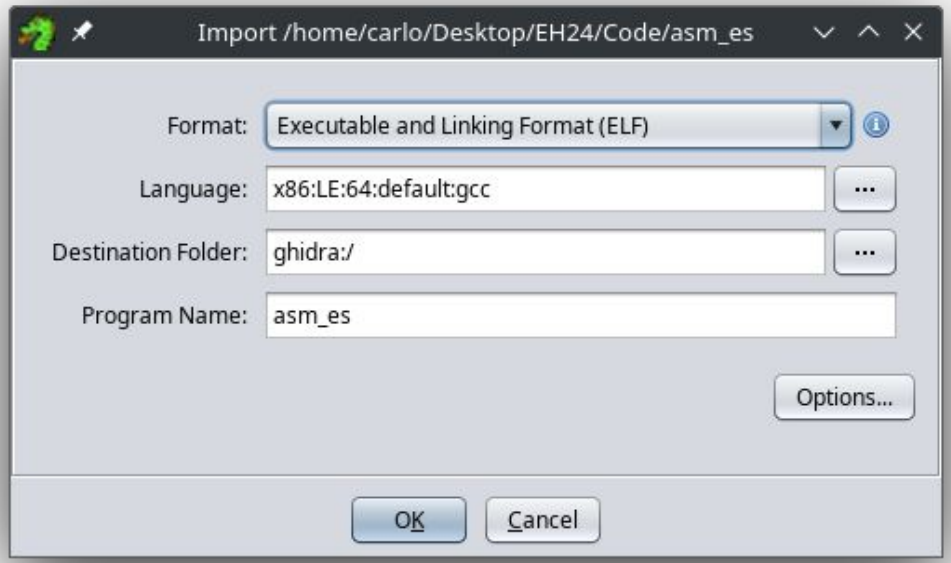
SRE - Decompiler - Ghidra



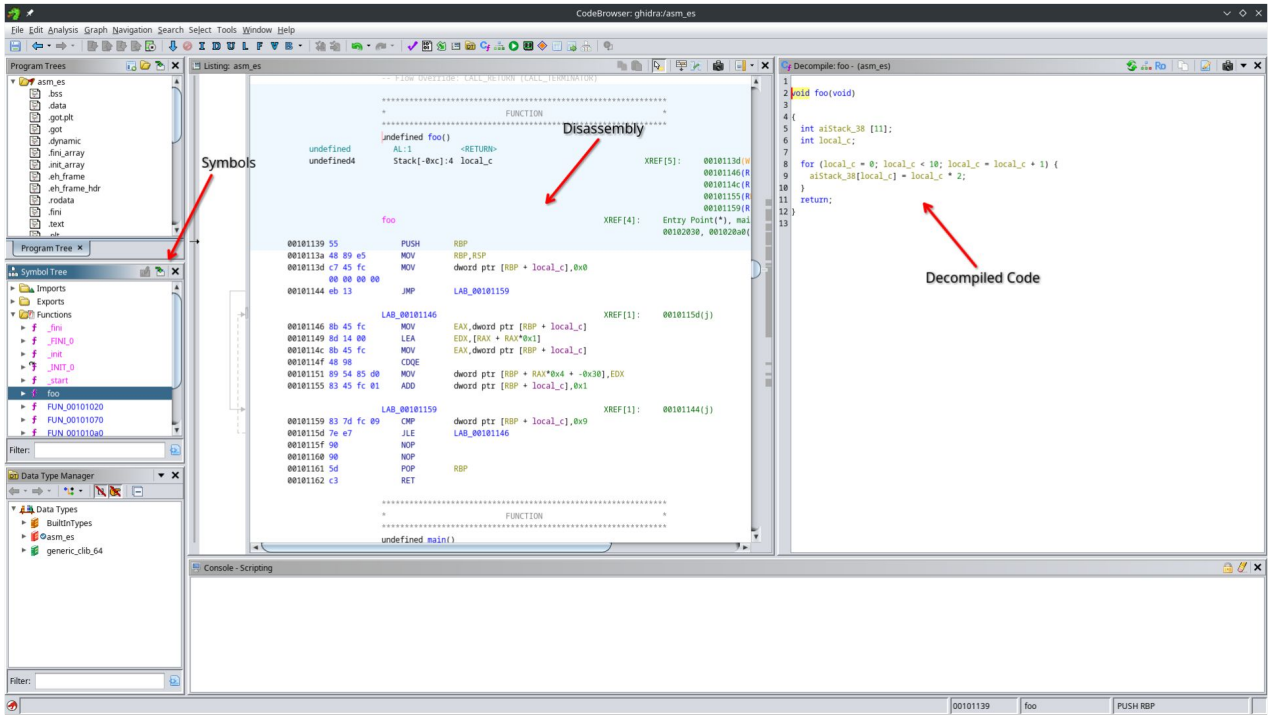
SRE - Decompiler - Ghidra



SRE - Decompiler - Ghidra



SRE - Decompiler - Ghidra



SRE - Decompiler - Ghidra

```
void foo(void)

{
  int aiStack_38 [11];
  int local_c;

  for (local_c = 0; local_c < 10; local_c = local_c + 1) {
    aiSta
  }
  return;
}
```



```
void foo(void)

{
  int num [11];
  int i;

  for (i = 0; i < 10; i = i + 1) {
    num[i] = i * 2;
  }
  return;
}
```

Carlo Ramponi