Linking: from the object file to the executable

An overview of static and dynamic linking

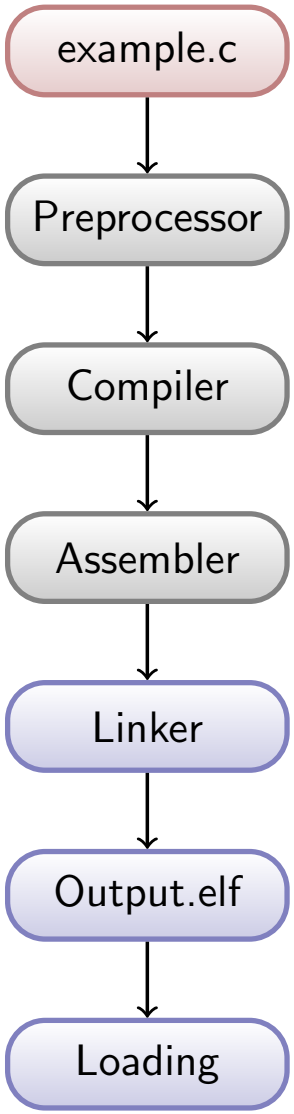
Andrea Gussoni
andrea1.gussoni at polimi.it

Politecnico di Milano

May 10, 2019

Table of Contents

Compiler Pipeline



ELF file structure

- An object file is the final form of a translation unit,
- It's divided in several sections:

`.text`	Machine code
`.data`	Initialized writable data
`.bss`	Non-initilized writable data
`.rodata`	Read-only data
`.eh_frame`	Stack-frame information for C++ exception handling
`.symtab`	Symbol table (variable and function names)
`.strtab`	Strings
`.rela.text`	Relocation section

Running Example

```c
#include <stdint.h>

uint32_t uninitialized;
uint32_t zero_initialized = 0;
const uint32_t constant = 0x41424344 ;
const uint32_t *initialized = &constant ;

uint32_t my_function(void) {
  return 42 + uninitialized;
}
```

Running Example

```
$ gcc -c example.c -o example.o -fno-PIC -fno-stack-protector
```

Running Example

```
$ readelf -h example.o
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:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          768 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         14
  Section header string table index: 13
```

Running Example

```
$ objdump -d -j .text example.o

Disassembly of section .text:

0000000000000000 <my_function>:
   0:   55                          push   rbp
   1:   48 89 e5                    mov    rbp,rsp
   4:   8b 05 00 00 00 00           mov    eax,[rip+0x0]
   a:   83 c0 2a                    add    eax,0x2a
   d:   5d                          pop    rbp
   e:   c3                          ret
```

Sections

```
$ readelf -S --wide example.o

There are 14 section headers, starting at offset 0x300:

Section Headers:
  [Nr] Name              Type            Off    Size   ES Flg Lk Inf
  [ 0]                   NULL            000    000    00      0   0
  [ 1] .text             PROGBITS        040    00f    00  AX  0   0
  [ 2] .rela.text        RELA            250    018    18   I 11   1
  [ 3] .data             PROGBITS        050    008    00  WA  0   0
  [ 4] .rela.data        RELA            268    018    18   I 11   3
  [ 5] .bss              NOBITS          058    004    00  WA  0   0
  [ 6] .rodata           PROGBITS        058    004    00   A  0   0
  ...
  [11] .symtab           SYMTAB          0c0    150    18     12   9
  [12] .strtab           STRTAB          210    03f    00      0   0
  [13] .shstrtab         STRTAB          298    066    00      0   0
```

Sections

```
$ readelf -S --wide example.o

There are 14 section headers, starting at offset 0x300:

Section Headers:
  [Nr] Name              Type            Off    Size   ES Flg Lk Inf
  [ 0]                   NULL            000    000    00      0   0
  [ 1] .text             PROGBITS        040    00f    00  AX  0   0
  [ 2] .rela.text        RELA            250    018    18   I 11   1
  [ 3] .data             PROGBITS        050    008    00  WA  0   0
  [ 4] .rela.data        RELA            268    018    18   I 11   3
  [ 5] .bss              NOBITS          058    004    00  WA  0   0
  [ 6] .rodata           PROGBITS        058    004    00   A  0   0
  ...
  [11] .symtab           SYMTAB          0c0    150    18     12   9
  [12] .strtab           STRTAB          210    03f    00      0   0
  [13] .shstrtab         STRTAB          298    066    00      0   0
```

Sections

```
$ readelf -S --wide example.o

There are 14 section headers, starting at offset 0x300:

Section Headers:
  [Nr] Name              Type            Off    Size   ES Flg Lk Inf
  [ 0]                   NULL            000    000    00      0   0
  [ 1] .text             PROGBITS        040    00f    00  AX  0   0
  [ 2] .rela.text        RELA            250    018    18   I 11   1
  [ 3] .data             PROGBITS        050    008    00  WA  0   0
  [ 4] .rela.data        RELA            268    018    18   I 11   3
  [ 5] .bss              NOBITS          058    004    00  WA  0   0
  [ 6] .rodata           PROGBITS        058    004    00   A  0   0
  ...
  [11] .symtab           SYMTAB          0c0    150    18     12   9
  [12] .strtab           STRTAB          210    03f    00      0   0
  [13] .shstrtab         STRTAB          298    066    00      0   0
```

Sections

```
$ readelf -S --wide example.o

There are 14 section headers, starting at offset 0x300:

Section Headers:
  [Nr] Name              Type            Off    Size   ES Flg Lk Inf
  [ 0]                   NULL            000    000    00      0   0
  [ 1] .text             PROGBITS        040    00f    00  AX  0   0
  [ 2] .rela.text        RELA            250    018    18   I 11   1
  [ 3] .data             PROGBITS        050    008    00  WA  0   0
  [ 4] .rela.data        RELA            268    018    18   I 11   3
  [ 5] .bss              NOBITS          058    004    00  WA  0   0
  [ 6] .rodata           PROGBITS        058    004    00   A  0   0
  ...
  [11] .symtab           SYMTAB          0c0    150    18     12   9
  [12] .strtab           STRTAB          210    03f    00      0   0
  [13] .shstrtab         STRTAB          298    066    00      0   0
```

Sections

```
$ readelf -S --wide example.o

There are 14 section headers, starting at offset 0x300:

Section Headers:
  [Nr] Name              Type            Off    Size   ES Flg Lk Inf
  [ 0]                   NULL            000    000    00      0   0
  [ 1] .text             PROGBITS        040    00f    00  AX  0   0
  [ 2] .rela.text        RELA            250    018    18   I 11   1
  [ 3] .data             PROGBITS        050    008    00  WA  0   0
  [ 4] .rela.data        RELA            268    018    18   I 11   3
  [ 5] .bss              NOBITS          058    004    00  WA  0   0
  [ 6] .rodata           PROGBITS        058    004    00   A  0   0
  ...
  [11] .symtab           SYMTAB          0c0    150    18     12   9
  [12] .strtab           STRTAB          210    03f    00      0   0
  [13] .shstrtab         STRTAB          298    066    00      0   0
```

Sections

We'll see where uninitialized variables are stored later.

.text

```
$ objdump -d -j .text example.o

Disassembly of section .text:

0000000000000000 <my_function>:
   0:   55                          push   rbp
   1:   48 89 e5                    mov    rbp,rsp
   4:   8b 05 00 00 00 00           mov    eax,[rip+0x0]
   a:   83 c0 2a                    add    eax,0x2a
   d:   5d                          pop    rbp
   e:   c3                          ret
```

.data, .rodata and .bss

```
$ objdump -s -j .data -j .rodata -j .bss example.o

Contents of section .data:
 0000 00000000 00000000                    ........
Contents of section .rodata:
 0000 44434241                             DCBA
```

.bss

.bss is not stored on file, since its content it's all zeroes

Create custom sections

To create a custom section with GCC, you can use the following attribute:

```
__attribute__ ((section ("mysection")))
```

Symbols

- A symbol is a label for a piece of code or data
- A symbol is composed by:

`st_name`	Index of the name in the symbol string table
`st_value`	Symbol offset in the table
`st_size`	Symbol size
`st_info`	Symbol type and binding
`st_other`	Symbol visibility
`st_shndx`	Index of the containing section

Undefined symbols

- If the value of `st_shndx` is equal to `0`, it means that the symbol is not defined in the current translation unit.
- Later on, the linker will be tasked with the resolution of all these symbols.

Common symbols

If st_shndx is `COM (0xfff2)`, it's a common symbol

- Used for uninitialized global variables
- You can have multiple definition in different TUs
- They can have different size, the largest will be chosen.
- It has no storage associated in any object file
- It ends up in .bss

Symbol Types

A symbol have an associated type:

`STT_OBJECT`	Data object (variable, array)
`STT_FUNC`	Function
`STT_SECTION`	Section
`STT_FILE`	Source file associated

Symbol binding

The binding property of a symbol specifies how it can be used in other translation units:

STB_LOCAL	Symbol is not visible outside the current translation unit
STB_GLOBAL	Symbol visible to all the translation units
STB_WEAK	Like GLOBAL, but the definition has lower precedence

Symbol visibility

- A *module* is an executable or a dynamic library
- The symbol visibility specifies the symbol exported from the current executable or library
- Standard executables usually do not export symbols

Symbol visibility

The visibility type can be:

`STV_DEFAULT`	Global and weak symbols available
`STV_PROTECTED`	Available, but uses own version
`STV_HIDDEN`	Unavailable, always use own version

Symbol table example

```
$ readelf -r example.o

 Num:  Value   Size  Type     Bind   Vis       Ndx  Name
   0:  00000      0  NOTYPE   LOCAL  DEFAULT   UND
   1:  00000      0  FILE     LOCAL  DEFAULT   ABS  example.c
   2:  00000      0  SECTION  LOCAL  DEFAULT     1
  ...
   9:  00004      4  OBJECT   GLOBAL DEFAULT   COM  uninitialized
  10:  00000      4  OBJECT   GLOBAL DEFAULT     5  zero_initialized
  11:  00000      4  OBJECT   GLOBAL DEFAULT     6  constant
  12:  00000      8  OBJECT   GLOBAL DEFAULT     3  initialized
  13:  00000     15  FUNC     GLOBAL DEFAULT     1  my_function
```

Relocations

Relocations are directives for the loader, that instruct it to write the value of a certain symbol in a certain location

Structure of a relocation

- Relocations are organized in *relocation tables*
- There can be a relocation table for each section
- The field composing a relocation are:

  `r_offset` Offset in the section where to write
  `r_info` Type of the relocation and symbol identifier
  `r_addend` Constant value to add to the value

Relocation types

- In `r_info` it is specified how the relocation should write the symbol's value
- Relocation types are architecture-specific, e.g.:

`R_X86_64_64`	Full symbol value (64 bit)
`R_X86_64_PC32`	Offset from the relocation target (32 bit)

Relocation Example

```
$ readelf -r example.o

Relocation section '.rela.text' contains 1 entry:
  Offset   Type            Sym. Name + Addend
  000006   R_X86_64_PC32   uninitialized + 0

Relocation section '.rela.data' contains 1 entry:
  Offset   Type            Sym. Name + Addend
  000000   R_X86_64_64     constant + 0
```

Table of Contents

The Linker

- The linker is usually invoked directly by the compiler (with gcc you can see the invocation with the -v flag).
- The main linkers are:
  - `ld.bfd` High retrocompatibility, not optimized
  - `ld.gold` Elf-only, optimized
  - `lld` LLVM-based, "limited" application, hugely optimized

The Linker

- Take all the input object files
- Lay out the output binary
- Build the final symbol table
- Apply all the relocations
- Output the final executable or dynamic library

Binary layout

- Fix a starting address for each section name
- Concatenate all the sections with the same name
- Keep sections with same features close to each other

Building the symbol table

- Scan all the input symbol tables and merge them
- Set symbol values to their final virtual address
- Check all the undefined symbols have been resolved
- Check no symbol is defined twice

Relocations

Simply apply all the relocations directives using the symbols' final address

Before linking

```
$ objdump -d -j .text example.o

Disassembly of section .text:

0000000000000000 <my_function>:
   0:   55                      push   rbp
   1:   48 89 e5                mov    rbp,rsp
   4:   8b 05 00 00 00 00       mov    eax,[rip+0x0]
   a:   83 c0 2a                add    eax,0x2a
   d:   5d                      pop    rbp
   e:   c3                      ret
```

After linking

```
$ gcc main.c example.c -o main -fno-PIC -fno-stack-protector
$ objdump -d -j .text main

Disassembly of section .text:

0000000000000631 <my_function>:
 631:   55                      push   rbp
 632:   48 89 e5                mov    rbp,rsp
 635:   8b 05 e5 09 20 00       mov    eax,[rip+0x2009e5] # 201020
 63b:   83 c0 2a                add    eax,0x2a
 63e:   5d                      pop    rbp
 63f:   c3                      ret
```

Static libraries

- Static libraries (`.a` files) are copied in the final executable
- They are an archive composed by object files

```
ar rcs mylib.a example.o foo.o
ranlib mylib.a
```

- Typically a library is linked with the `-l` parameter

```
gcc -lmylib main.c -o main.o
```

- Only the object files providing otherwise undefined symbols will be linked in

Memory mapping

Similar sections of different objects will be mapped together in the final linking:

- Code will go in an executable page
- Read-only data will go in a read-only page
- Writeable data will go in a writeable page

Memory Layout

- In memory the sections will be grouped together if they require identical permissions
- Segments are defined in the program headers
- A segment is described by:

`p_offset`	Offset from the beginning of the file
`p_vaddr`	Virtual address in memory where it should be loaded
`p_filesz`	Size of the segment in the file
`p_memsz`	Size of the segment in memory
`p_flags`	Permissions

Standard segments

Programs are usually composed by two segments:

+rx `.text`, `.rodata`, `.plt`

+rw `.got`, `.data`, `.bss`

Loader

- At program startup, the kernel reads the program headers and maps the required pages in memory setting the appropriate permissions
- Why `p_filesz` and `p_memsize` are both specified?
- They may differ (e.g., `.bbs` which is all zeroes)
- The exceeding portion is zero-initialized

Loader

- At program startup, the kernel reads the program headers and maps the required pages in memory setting the appropriate permissions
- Why `p_filesz` and `p_memsize` are both specified?
- They may differ (e.g., `.bbs` which is all zeroes)
- The exceeding portion is zero-initialized

```
$ readelf -l main

Program Headers:
  Type   Offset VirtAddr FileSiz  MemSiz   Flg Align
  ...
  LOAD   0x0000 0x000000 0x000838 0x000838 R E 0x200000
  LOAD   0x0df0 0x200df0 0x000228 0x000238 RW  0x200000
  ...

 Section to Segment mapping:
  Segment Sections...
  ...
   02     .text .rodata
   03     .data .bss
  ...
```

Table of Contents

Dynamic linking

- Dynamic linking is needed to support dynamic libraries
- Why dynamic linking?:
  - Fix a bug in a library $\Rightarrow$ fix all the binaries that uses it
  - Library loaded in memory only once $\Rightarrow$ save memory
  - Do not replicate code $\Rightarrow$ save disk space

Dynamic linking

- It is performed at run-time
- Loads the dynamic libraries in memory
- Resolves the address of symbols in dynamic libraries

Dynamic linker

```
$ readelf -l main
Elf file type is DYN (Shared object file)
Entry point 0x540
There are 9 program headers, starting at offset 64

Program Headers:
  Type      Offset   VirtAddr FileSiz MemSiz Flg Align
  ...
  INTERP    0x000238 0x000238 0x0001c 0x001c R   0x1
  [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Differences w.r.t. static linking

- Used sections:

  `.dynsym` Dynamic symbol table
  `.dynstr` String table for `.dynsym`
  `.rela.dyn` Dynamic relocation table

- Uses a different set of relocation types
- `r_offset` is not an offset in a section, but a virtual address

Dynamic libraries

- Dynamic libraries can be placed anywhere in the memory at runtime
- They can't have absolute addresses at linking time
- We can't patch them at run-time
- No dynamic relocations in `.text` or `.rodata`

Position Independent Code

- Dynamic libraries are usually compiled as `PIC`
- We can't use absolute addresses, since we do not know where they will be loaded
- The solution is to use the value of the program counter as a base index, and express all the addresses as offsets with respect to this value.
- Also the distance between `.text` and `.data` should be constant

Example

libone.c:

```
extern int libtwo_variable;

int libone_function(void) {
  return 42 + libtwo_variable;
}
```

libtwo.c:

```
int libtwo_variable = 0x42;

int libtwo_function(void) {
  return 0;
}
```

Building a dynamic library

- To create a dynamic library:

```
gcc -fPIC -c libone.c -o libone.o
gcc -shared -fPIC -o libone.so libone.o
```

- Linking against a dynamic library:

```
gcc main -c -lone -o main
```

- Dynamic libraries are not copied into the final executable
- They must be available in the predefined system paths (e.g. `/usr/lib`, `/lib`, ...)

Linking order

- Input object files are always linked in
- Their order doesn't matter
- Static and dynamic libraries order instead is important
- Suppose `libone.so` requires `libtwo.so`
- The `-ltwo` parameter must be passed before `-lone`
- You can use fixed-point linking with `--start-group`/`--end-group`:

```
gcc -Wl,--start-group -lone -ltwo -Wl,--end-group
```

Symbols in another library

How can we access symbols in another library?

- We can't patch `.text`
- Two main problems:
  - How can we access global variables defined in other modules?
  - How can we call functions defined in other modules?

The GOT

- The GOT (*Global Offset Table*), is the mechanism that enables dynamic resolution of external symbols at runtime.
- For global variables:
  - Contains a pointer-sized entry for each imported variables
  - It is populated at startup by the dynamic loader
  - Holds the runtime address of the variables

Example relocations

```
$ gcc -shared -fPIC -L. -ltwo libone.c -o libone.so
$ readelf -S libone.so

Section Headers:
  [Nr] Name  Type          Address Off  Size ES Flg Lk Inf
  ...
  [16] .got     PROGBITS  0200fd8 0fd8 0028 08  WA  0   0
  ...

$ readelf -r libone.so

Relocation section '.rela.dyn':
  Offset  Info      Type              Sym. Name + Addend
  ...
  200fe0  200000006 R_X86_64_GLOB_DAT libtwo_variable + 0
  ...
```

Example relocations

```
$ objdump -j .text -d libone.so

00000000000005ca <libone_function>:
 5ca:   push   rbp
 5cb:   mov    rbp,rsp
 5ce:   mov    rax,[rip+0x200a0b] # 200fe0 <libtwo_variable>
 5d5:   mov    eax,[rax]
 5d7:   add    eax,0x2a
 5da:   pop    rbp
 5db:   ret
```

External functions

- In principle we could apply the same procedure also for the functions.
- This means that at startup the dynamic loader should apply all the relocations
- Often programs import a lot of functions, but they are not actually used very often
- *Lazy loading* is a mechanism to resolve the address only when a function is actually used

New sections

- Dynamic loading requires three new sections:

`.plt`	Small code stubs to call library functions
`.got.plt`	Lazy GOT for library functions addresses
`.rela.plt`	Relocation table relative to `.got.plt`

Example

```
#include <stdio.h>

int main (void) {
  int a = 1;
  int b = 2;
  int c = a + b;

  printf("%d\n", c);
  return 0;
}

$ gcc example.c
```

Sections

```
$ readelf -S a.out
Section Headers:
  [Nr] Name              Type            Address          Size   ES Flg Lk Inf
  ...
  [10] .rela.plt         RELA            00004a0 000018 18   AI  5  23
  [12] .plt              PROGBITS        00004d0 000020 10   AX  0   0
  ...
  [14] .text             PROGBITS        0000500 0001c2 00   AX  0   0
  ...
  [22] .got              PROGBITS        0200a20 000028 08   WA  0   0
  [23] .got.plt          PROGBITS        0200a48 000020 08   WA  0   0
  ...
```

.text

```
$ objdump -d -j .text a.out
Disassembly of section .text:

000000000000060a <main>:
...
 63c: e8 9f fe ff ff   callq  4e0 <printf@plt>
...
```

`.plt` section

- For each function, in the `.plt` section we have a little stub of code which is used for the lazy loading
- The first entry in the `.plt` is the code used to invoke the dynamic loader during lazy loading

.plt section

```
$ objdump -d -j .plt a.out

Disassembly of section .plt:

00000000000004d0 <.plt>:
 4d0:   ff 35 7a 05 20 00      pushq  0x20057a(%rip)
 4d6:   ff 25 7c 05 20 00      jmpq   *0x20057c(%rip)
 4dc:   0f 1f 40 00            nopl   0x0(%rax)

00000000000004e0 <printf@plt>:
 4e0:   ff 25 7a 05 20 00      jmpq   *0x20057a(%rip) # 200a60
 4e6:   68 00 00 00 00         pushq  $0x0
 4eb:   e9 e0 ff ff ff         jmpq   4d0 <.plt>
```

`.plt` section

- The first instruction is a jump to an address contained in the `GOT`.

Relocations

```
$ readelf -r a.out

Relocation section '.rela.plt':
Offset          Info           Type                     Sym. Name + Addend
200a60          200000007 R_X86_64_JUMP_SLO printf@GLIBC_2.2.5 + 0
```

.got section

```
Contents of section .got.plt:
  200a48 30082000 00000000 00000000 00000000  0. ............
  200a58 00000000 00000000 e6040000 00000000  ................
```

Relocations

- If we interpret that address in little endian, we obtain the address of the second instruction of the stub.
- The stub pushes `0` on the stack and call the dynamic loader
- The value which is pushed on the stack is simply an identifier for the function for which we need to resolve the address.

Relocations

- Isn't this mechanism expensive?
- Actually, only the first time the external function is called this mechanism is invoked
- The slot that we saw earlier in the GOT is overwritten with the actual value in memory of the external function
- In this way, from the second time on, no extra cost is paid

Section vs segments

- The loader knows only about program headers
- It can ignore at all the information about sections
- It uses `PT_DYNAMIC` which points directly to `.dynamic` which contains:
  - Needed libraries
  - Address and size of `.dynsym`, `.dynstr`, `.got.plt`, `.rela.plt`

.dynamic

```
$ readelf -l a.out

Elf file type is DYN (Shared object file)
Entry point 0x540
There are 9 program headers, starting at offset 64

Program Headers:
  Type      Offset   VirtAddr FileSiz  MemSiz   Flg
  ...
  DYNAMIC   0x000830 0x200830 0x0001f0 0x0001f0 RW
  ...

$ readelf -S a.out

Section Headers:
  [Nr] Name       Type      Address Off  Size ES Flg
  ...
  [21] .dynamic   DYNAMIC   0200830 830  01f0 10  WA
  ...
```

```
$ readelf -d a.out

Dynamic section at offset 0xdc8 contains 27 entries:
 Type              Name/Value
 (NEEDED)          Shared library: [libc.so.6]
 ..
 (STRTAB)          0x328
 (SYMTAB)          0x280
 (STRSZ)           132 (bytes)
 (SYMENT)          24 (bytes)
 (DEBUG)           0x0
 (PLTGOT)          0x200a48
 (PLTRELSZ)        24 (bytes)
 (PLTREL)          RELA
 (JMPREL)          0x4a0
 (RELA)            0x3e0
 (RELASZ)          192 (bytes)
 (RELAENT)         24 (bytes)
 ...
```

Section vs segments

- The section table is optional
- Also `.symtab` and `.strtab` are optional
- They are debugging information
- The `strip` tool can remove them to save space

Table of Contents

Relaxation

- Certain architectures offer different instructions for performing jumps depending on how "distant" the target is.
- In principle, the compiler cannot know how long the jump will be, and will therefore need to apply the most conservative assumptions
- This could lead to some inefficiency

Relaxation

- The compiler can introduce *relaxation hints*
- A smart linker can use this hints to reassign addresses and substitute inefficient instructions with more efficient ones

Reducing the code size

- It may happen that some functions, even if unused, are included in the final binary
- The compiler in fact cannot know if a non *static* function will be used
- Only the linker has the necessary visibility to know that

Reducing the code size

- Unfortunately the ELF linker works with a *per-section* granularity, and therefore will link sections as a whole.
- This is not true for example for the Mach-O format, which instead has a finer granularity

A workaround

- We can tell the compiler to emit a section for each function or data object:

```
gcc -ffunction-sections -fdata-sections
```

- Then we instruct the linker to drop all the sections which are not used:

```
gcc -Wl,--gc-sections
```

LTO

- Traditionally compilers optimize with a TU granularity
- LTO (`-flto`) optimizes the program as a whole
- The compiler emit an high-level, internal representation:

  `GCC` an ELF with sections containing GIMPLE

  `clang` a bitcode file containing the LLVM IR
- The linker will merge these IRs and optimize them

LTO

```
cat *.c > all.c

gcc all.c -o all
```

Pros and Cons

Pros:

- Larger visibility across multiple
- Larger optimization opportunities, as aggressive dead code optimization

Cons:

- Resource intensive
- Only a subset of the possible optimizations can be run

Security considerations

`-z relro` Some sections, including `.dynamic`, are marked read-only after they have been initialized by the dynamic loader (now default in Ubuntu 18.04)

`-z now` Disable lazy loading. All import symbols are resolved at startup, and `.got.plt` is marked read-only

`-pie` Compile also standard executables (not only libraries) as PIC, and produce a program which is completely relocatable in memory (default in current versions of GCC)

References I


Kishu Agarwal. *Life of a Binary*. URL: https://kishuagarwal.github.io/life-of-a-binary.html.


Alessandro Di Federico et al. "How the {ELF} Ruined Christmas". In: *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015, pp. 643–658.


Santa Cruz Operation. *System V Application Binary Interface*. URL: http://www.sco.com/developers/gabi/latest/contents.html.

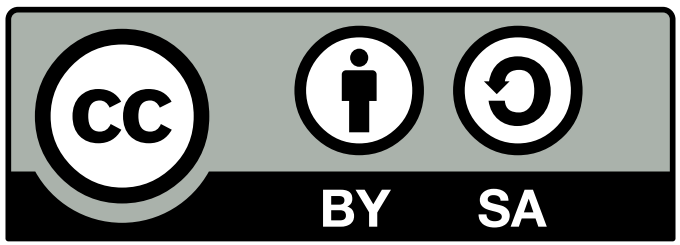
Special Thanks

- I want to thank the author of the last edition of this talk, Alessandro Di Federico[1], whose material I used as a starting point for preparing this talk. The original material was distributed with the CC-BY-SA 3.0 license.
- Some credits also goes to this[2] nice write-up by Kishu Agarwal.

[1] https://home.deib.polimi.it/agosta/lib/exe/fetch.php?id=teaching%3Acompilers&cache=cache&media=teaching:cot_slides_linker.pdf

[2] https://kishuagarwal.github.io/life-of-a-binary.html

License



[3] https://creativecommons.org/licenses/by-sa/4.0/