Linking: from the object file to the executable

An overview of static and dynamic linking

Alessandro Di Federico

ale@clearmind.me

Politecnico di Milano

April 11, 2018

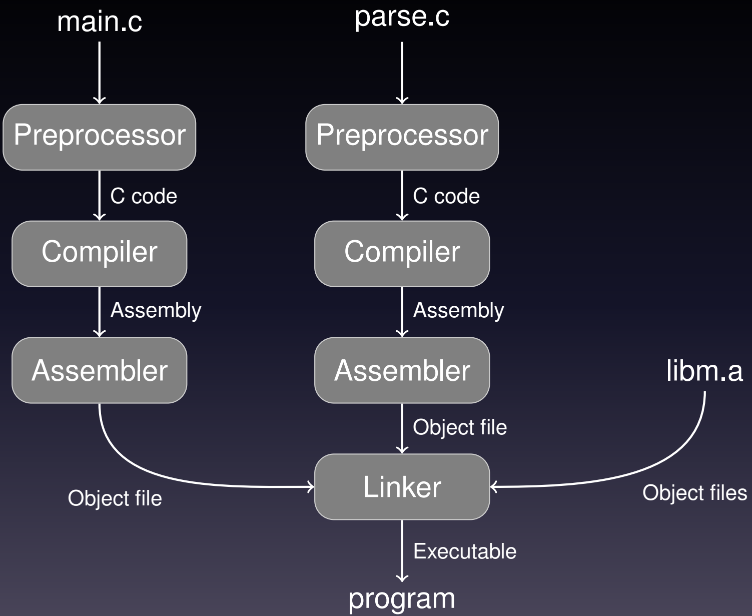
Index

ELF format overview

Static linking

Dynamic linking

Advanced linking features


main.c
parse.c
Preprocessor
Preprocessor
C code
C code
Compiler
Compiler
Assembly
Assembly
Assembler
Assembler
libm.a
Object file
Linker
Object file
Object files
Executable
program

We'll focus on the
ELF format [2]

The object file

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

`.text`	The actual executable code
`.data`	Global, initialized, writeable data
`.bss`	Global, non-initialized, writeable data
`.rodata`	Global read-only data
`.symtab`	The symbol table
`.strtab`	String table for the symbol table
`.rela.section`	The relocation table for *section*
`.shstrtab`	String table for the section names

An example

```
#include <stdint.h>

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

uint32_t a_function() {
  return 42 + uninitialized;
}
```

An example

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

.text

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

Disassembly of section .text:

0000000000000000 <a_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
```

The sections

```
$ readelf -S example.o

Nr  Name        Type      Off  Size  ES  Flg Inf
 0              NULL      000  000   00        0
 1  .text       PROGBITS  040  00f   00   AX   0
 2  .rela.text  RELA      298  018   18    I   1
 3  .data       PROGBITS  050  008   00   WA   0
 4  .rela.data  RELA      2b0  018   18    I   3
 5  .bss        NOBITS    058  004   00   WA   0
 6  .rodata     PROGBITS  05c  004   00    A   0
...
11  .shstrtab   STRTAB    0b8  066   00        0
12  .symtab     SYMTAB    120  138   18        9
13  .strtab     STRTAB    258  039   00        0
```

The sections

```
$ readelf -S example.o

Nr  Name        Type      Off  Size  ES  Flg  Inf
 0              NULL      000  000   00         0
 1  .text       PROGBITS  040  00f   00   AX    0
 2  .rela.text  RELA      298  018   18    I    1
 3  .data       PROGBITS  050  008   00   WA    0
 4  .rela.data  RELA      2b0  018   18    I    3
 5  .bss        NOBITS    058  004   00   WA    0
 6  .rodata     PROGBITS  05c  004   00    A    0
...
11  .shstrtab   STRTAB    0b8  066   00         0
12  .symtab     SYMTAB    120  138   18         9
13  .strtab     STRTAB    258  039   00         0
```

The sections

```
$ readelf -S example.o

Nr  Name          Type      Off  Size  ES  Flg  Inf
 0                NULL      000  000   00         0
 1  .text         PROGBITS  040  00f   00   AX    0
 2  .rela.text    RELA      298  018   18    I    1
 3  .data         PROGBITS  050  008   00   WA    0
 4  .rela.data    RELA      2b0  018   18    I    3
 5  .bss          NOBITS    058  004   00   WA    0
 6  .rodata       PROGBITS  05c  004   00    A    0
...
11  .shstrtab     STRTAB    0b8  066   00         0
12  .symtab       SYMTAB    120  138   18         9
13  .strtab       STRTAB    258  039   00         0
```

The sections

```
$ readelf -S example.o

Nr  Name        Type     Off  Size  ES  Flg  Inf
 0              NULL     000  000   00         0
 1  .text       PROGBITS 040  00f   00   AX    0
 2  .rela.text  RELA     298  018   18    I    1
 3  .data       PROGBITS 050  008   00   WA    0
 4  .rela.data  RELA     2b0  018   18    I    3
 5  .bss        NOBITS   058  004   00   WA    0
 6  .rodata     PROGBITS 05c  004   00    A    0
...
11  .shstrtab   STRTAB   0b8  066   00         0
12  .symtab     SYMTAB   120  138   18         9
13  .strtab     STRTAB   258  039   00         0
```

And the uninitialized variable?

And the uninitialized variable?

They are special, we’ll talk about them later

The sections

```
$ readelf -S example.o

Nr  Name         Type      Off  Size  ES  Flg  Inf
 0               NULL      000  000   00         0
 1  .text        PROGBITS  040  00f   00   AX    0
 2  .rela.text   RELA      298  018   18    I    1
 3  .data        PROGBITS  050  008   00   WA    0
 4  .rela.data   RELA      2b0  018   18    I    3
 5  .bss         NOBITS    058  004   00   WA    0
 6  .rodata      PROGBITS  05c  004   00    A    0
...
11  .shstrtab    STRTAB    0b8  066   00         0
12  .symtab      SYMTAB    120  138   18         9
13  .strtab      STRTAB    258  039   00         0
```

.text

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

Disassembly of section .text:

0000000000000000 <a_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
```

.text

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

Disassembly of section .text:

0000000000000000 <a_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
```

And `.bss`?

And `.bss`?

`.bss` is not stored in the file, it’s all zeros

Custom sections

You can also create your own section, in GCC:

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

Symbols

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

`st_name`	the name of the symbol (stored in `.strtab`).
`st_shndx`	index of the containing section.
`st_value`	the symbol's address/offset in the section.
`st_size`	the size of the represented object.
`st_info`	the symbol type and binding.
`st_other`	the symbol visibility.

Undefined symbols

If `st_shndx` is `0`, the symbol is not defined in the current TUs.
If the linker can't find it in any TUs it will complain.

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

The type of the symbol identifies what it represents:

`STT_OBJECT`	a global variable.
`STT_FUNC`	a function.
`STT_SECTION`	a section.
`STT_FILE`	a translation unit.

Symbol binding

The binding of a symbol, determines if and how can be used by other translation units:

`STB_LOCAL`	local to the TU (`static` in C terms).
`STB_GLOBAL`	available to other TUs (`extern` in C terms).
`STB_WEAK`	like `STB_GLOBAL`, but can be overridden.

Symbol visibility

binding $\Longrightarrow$ is it available to other TUs?
visibility $\Longrightarrow$ is it available to other *modules*?

- A *module* is an executable or a dynamic library
- Visibility determines if a function is exported by the library
- Executables usually ignore visibility (they export nothing)
- To change the visibility in GCC:

```
__attribute__ ((visibility ("...")))
```

Type of visibility

There are several different types of visibility:

STV_DEFAULT	visible, can use an external version.
STV_HIDDEN	not visible, always use own version.
STV_PROTECTED	visible, but use always use own version.

Symbol table example

```
$ readelf -s example.o

 #  Val  Sz Type    Bind   Vis Ndx Name
 0  000   0 NOTYPE  LOCAL  DEF UND
 1  000   0 FILE    LOCAL  DEF ABS example.c
 2  000   0 SECTION LOCAL  DEF   1
 3  000   0 SECTION LOCAL  DEF   3
 4  000   0 SECTION LOCAL  DEF   5
...
 9  000   4 OBJECT  GLOBAL DEF COM uninitialized
10  000   4 OBJECT  GLOBAL DEF   5 zero_intialized
11  000   4 OBJECT  GLOBAL DEF   6 constant
12  000   8 OBJECT  GLOBAL DEF   3 initialized
13  000  15 FUNC    GLOBAL DEF   1 a_function
```

Relocations

A relocation is a directive for the linker

Relocations

A relocation is a directive for the linker

Dear linker, write the value of a certain symbol
in a certain location in a certain way

Structure of a relocation

- Relocations are organized in *relocation tables*
- There can be a relocation table for each section
- A relocation is composed by the following fields:

`r_offset`	where to write (as an offset in the section).
`r_info`	symbol identifier and relocation type.
`r_addend`	value to add to the symbol's value (optional).

Relocation types

- Part of `r_info`, specify *how* to write the symbol's value
- There are several, architecture-specific, relocation types:

  `R_X86_64_64` full symbol value (64 bit).

  `R_X86_64_PC32` offset from the relocation target (32 bit).

  And many others similar to these.

Relocation table example

```
$ readelf -r example.o

Section '.rela.text' contains 1 entries:
Offset    Type              Symbol's Name + Addend
000006    R_X86_64_PC32     uninitialized + 0

Section '.rela.data' contains 1 entries:
Offset    Type            Symbol's Name + Addend
000000    R_X86_64_64     constant + 0
```

Index

ELF format overview

Static linking

Dynamic linking

Advanced linking features

Who is the linker?

- Usually you don't invoke it directly, GCC will do it for you
- Use `gcc -v` if you want to see how it's invoked
- Under unix-like platforms, three main linkers:

`ld.bfd`	wide feature set, slow, high memory usage.
`ld.gold`	ELF-only, fast, reduced memory usage.
`lld`	the new kid in the block, faster, LLVM-based.

The linker

What does the linker do?

The linker

What does the linker do?

1. Take in input several object files
2. Lay out the output binary
3. Build the final symbol table from the inputs
4. Apply all the relocations
5. Output the final executable/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 (*one definition rule*)

Relocations

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

Original .text

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

Disassembly of section .text:

0000000000000000 <a_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
```

Linked .text

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

Disassembly of section .text:

0000000000400514 <a_function>:
400514: 55                push rbp
400515: 48 89 e5          mov  rbp,rsp
400518: 8b 05 72 0b 20 00 mov  eax, [rip+0x200b72]
40051e: 83 c0 2a          add  eax,0x2a
400521: 5d                pop  rbp
400522: c3                ret
```

What about libraries?

- Let's first focus on static libraries, i.e. `.a` files
- Static libraries are *copied* into the final binary
- A `.a` file is just an archive of object files

```
ar rcs libmine.a example.o other.o
ranlib libmine.a
```

- Typically a library is linked with the `-l` parameter[1], e.g.

```
gcc -lmine main.c -o main.o
```

- Not all the object files will be linked in
- Only those providing otherwise undefined symbols

[1] `libmine.a` must be in the library search path (e.g. in `/lib`). Otherwise use `-L` to add a path to the library search path.

We said the linker places similar sections together

Why is that?

Memory mapping

Similar sections will be mapped together:

- Code (e.g., `.text`) will go in a executable page
- Read-only data (e.g., `.rodata`) will go in a read-only page
- Writeable data (e.g., `.data`) will go in a writeable page

Introducing: segments

- A segment groups sections requiring similar permissions
- Segments are defined in the *program headers*
- A segment is composed by:

`p_offset`	offset in the *file* where the segment starts.
`p_vaddr`	virtual address where it should be loaded.
`p_filesz`	size of the segment *in the file*.
`p_memsz`	size of the segment *in memory*.
`p_flags`	permission: executable, writeable, readable.

Standard segments

Typically a program has two segments:

```
+rx .rodata, .text, ...
+rw .data, .bss, ...
```

The kernel reads the program headers and maps the required pages in memory with the appropriate permissions

How comes we have both `p_filesz` and `p_memsz`?

How comes we have both `p_filesz` and `p_memsz`?

They might differ: `.bss` is all zeros, so it’s not stored in the file.
The `p_memsz` portion exceeding `p_filesz` is zero-intialized.

Program headers example

```
$ readelf -l example

Program Headers:
Type Offset VirtAddr FileSiz MemSiz Flg
...
LOAD 0x000  0x400000 0x61c   0x61c  R E
LOAD 0xea8  0x600ea8 0x188   0x1f0  RW
...

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

/proc/$PID/maps

```
$ cat /proc/$EXAMPLE_PID/maps

00400000-00401000 r-xp ... ./example
00600000-00602000 rw-p ... ./example
...
```

Bonus: final section table

```
$ readelf -S example

Nr  Name       Type     Address  Off    Size Flg
...
 8  .text      PROGBITS 400380   000380 1e6  AX
...
10  .rodata    PROGBITS 400570   000570 004   A
...
18  .data      PROGBITS 601020   001020 010  WA
19  .bss       NOBITS   601040   001030 058  WA
...
21  .shstrtab  STRTAB   000000   00104f 0b2
22  .symtab    SYMTAB   000000   001108 468
23  .strtab    STRTAB   000000   001570 148
```

Index

ELF format overview

Static linking

Dynamic linking

Advanced linking features

Why dynamic linking?

- Dynamic linking is required to support dynamic libraries
- Dynamic libraries benefits:
  - Fix a bug in a library $\Rightarrow$ multiple applications will benefit
  - Load the library once $\Rightarrow$ save physical memory
  - Do not replicate code $\Rightarrow$ save disk space

What is dynamic linking?

- A lighter version of the linking process
- It is performed at run-time
- It loads the dynamic libraries in memory
- It provides the address of symbols in dynamic libraries

Who is the dynamic linker?

```
$ readelf -l main

Program Headers:
Type   Offset VirtAddr FileSiz  MemSiz   Flg
...
INTERP 0x270  0x400270 0x00001c 0x00001c R
[Program interpreter: /lib64/ld-linux-x86-64.so.2]
...
```

Differences w.r.t. static linking

- Used sections:

`.dynsym`	Dynamic symbol tables (instead of `.symtab`)
`.dynstr`	String table for `.dynsym` (instead of `.strtab`)
`.rela.dyn`	Dynamic relocation table (instead of `.rela.`...)

- 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 anywhere in the address space

- They can’t have absolute addresses at linking time
- We want to share read-only parts among processes
- We can’t patch the them at run-time!
- So, no dynamic relocations in `.text` or `.rodata`

Position Independent Code

- Libraries are usually compiled as PIC
- Never use absolute addresses
- Use offsets from the current PC
- In the linked binary, the program base address is `0`

The idea

We can't know a symbol's absolute address
but we know its relative distance from the current PC

[2] Usually

The idea

We can't know a symbol's absolute address
but we know its relative distance from the current PC

For data too!
The distance between `.text` and `.data` is constant[2]

---

[2] Usually

Two dynamic libraries

libone.c

```c
extern int libtwo_variable;

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

libtwo.c

```c
#include <stdlib.h>

int libtwo_variable = 0x435;

int libtwo_function(void) {
  system("echo␣hello");
  return 151;
}
```

Building a dynamic library

- Creating a dynamic library:

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

- Linking against a dynamic library:

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

- Dynamic libraries are *not* copied into the final executable
- The `.so` file must be available in predefined paths:
  - `/usr/lib`, `/lib`...
  - Any path in the `LD_LIBRARY_PATH` environment variable

Digression: 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`
- Or use fixed-point linking with `--start-group`/`--end-group`:

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

Symbols in another library

What about symbols in another library?

- We can't patch `.text`
- Two problems:
  1. How to access global variables in another module?
  2. How to call functions in another module?

Introducing: the GOT

- Let's first see how we can access global variables
- The linker will create a *Global Offset Table* (`.got`):
  - It contains a pointer-sized entry for each imported variable
  - It holds their run-time addresses
  - It is populated upon startup by the dynamic loader

libone.so's relocations

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

Nr Name Type      Address Off  Size ES Flg Lk Inf
...
18 .got PROGBITS 200fa8   fa8  58   08 WA  0  0
...
```

.got is at the 0x200fa8-0x201000 range

```
$ readelf -r libone.so

Relocation section '.rela.dyn':
Offset Type              Symbol's Name + Addend
...
200fe0 R_X86_64_GLOB_DAT libtwo_variable + 0
...
```

library_function code

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

00000000000006c0 <library_function>:
 6c0: push rbp
 6c1: mov  rbp,rsp
 6c4: mov  rax,QWORD PTR [rip+0x200915] # 200fe0
 6cb: mov  eax,DWORD PTR [rax]
 6cd: add  eax,0x2a
 6d0: pop  rbp
 6d1: ret
```

What about functions?

Do they work in the same way?

Introducing: lazy loading

- A program could import a lot of functions
- Maybe some of them are not used very often
- Applying all the relocations slows down the startup
- Why don’t we fix the relocation only when needed?

The actors

- To implement lazy loading we use three new sections:

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

- For each imported function we have an entry in all of them

libtwo.so's sections

```
$ readelf -S libtwo.so

Nr  Name       Type     Address  Size ES Flg Inf
...
 7  .rela.plt  RELA     000538   48   18 AI  9
...
 9  .plt       PROGBITS 0005a0   40   10 AX  0
...
19  .got.plt   PROGBITS 200fa8   58   08 WA  0
...
```

`.got.plt` is at the `0x200fa8`-`0x201000` range

```
$ objdump -d libtwo.so

Disassembly of section .text:

00000000000006e0 <libtwo_function>:
...
 700: call 5b0 <system@plt>
...

Disassembly of section .plt:
...
00000000000005b0 <system@plt>:
 5b0: jmp  QWORD PTR [rip+0x200a0a] # 200fc0
...
```

.got.plt relocations

```
$ readelf -r libtwo.so

Relocation section '.rela.plt':
Offset Type              Symbol's Name + Addend
200fc0 R_X86_64_JUMP_SLOT system + 0
```

The PLT

This was what it would look like without lazy loading

Lazy loading

- At startup, `.got.plt` doesn't contain function addresses
- It contains the address of the stub's second instruction
- This second part invokes the dynamic loader
- The dynamic loader will fix the relocation
- From then on, `.got.plt` will contain the correct address

The real PLT

```
$ objdump -d -j .plt libtwo.so

00000000000005b0 <system@plt>:
 5b0: jmp   QWORD PTR [rip+0x200a0a] # 200fc0
 5b6: push 0x0
 5bb: jmp   5a0 <_init+0x20>

$ objdump -s -j .got.plt libtwo.so

libtwo.so:     file format elf64-x86-64

Contents of section .got.plt:
 200fb8 00000000 00000000 b6050000 00000000
```

`b6 05 00 00` is `0x5b6` in little endian.

A note on the dynamic loader

- The dynamic loader ignores the sections
- It just knows about program headers
- In particular it uses `PT_DYNAMIC`, which points to `.dynamic`
- `.dynamic` contains all the information it needs:
  - Needed libraries
  - Address and size of `.dynsym`, `.dynstr`, `.got.plt`, `.rela.plt`
  - ...

.dynamic section

```
$ readelf -l libone.so
Program Headers:
Type    Offset   VirtAddr FileSiz  MemSiz   Flg
...
DYNAMIC 0x000db8 0x200db8 0x0001f0 0x0001f0 RW
...

$ readelf -S libone.so

Section Headers:
Nr Name     Type    Address Off     Size    ES Flg
...
17 .dynamic DYNAMIC 200db8  000db8 0001f0 10   WA
...
```

.dynamic content

```
$ readelf -d libone.so

Type        Name/Value
(NEEDED)    Shared library: [libtwo.so]
(NEEDED)    Shared library: [libc.so.6]
...
(STRTAB)    0x358
(SYMTAB)    0x208
(STRSZ)     206 (bytes)
(SYMENT)    24 (bytes)
(PLTGOT)    0x200fa8
(PLTRELSZ) 48 (bytes)
...
(JMPREL)    0x540
(RELA)      0x468
(RELASZ)    216 (bytes)
(RELAENT)   24 (bytes)
...
```

What's mandatory?

This means that the section table is optional

- The same for `.symtab` and `.strtab`
- They are basically debugging information
- In fact, the `strip` tool can remove them to save space

Index

ELF format overview

Static linking

Dynamic linking

Advanced linking features

Relaxation

- At compile-time we might ignore how long a jump is
- Some ISAs have shorter instructions for short jumps
- In MIPS, a long jump can take up to 3 instructions
- What if the destination is unknown at compile-time?
- The compiler must emit the most conservative option

Introducing: relaxation

- Certain smart linkers offer relaxation
- That is, they're able to move the code up and down
- At link-time!
- Symbols for each basic block must be provided

Reducing code size

- It might happen that unused functions make it to link time
- The compiler ignores if a non-`static` function is unused
- The linker has the required information to understand this

But the ELF linker is dumb!

- If it is asked to link a section, it will link it as a whole
- The Mach-O format instead works at finer granularity

The solution

- The compiler can create a section per-function/data object

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

- Then, we can tell the linker to drop unreferenced sections:

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

Link-time optimization

- Traditionally compilers optimize with a TU granularity
- LTO (`-flto`) optimizes the program as whole
- The compiler does not emit a standard object file
- A high-level, internal representation is serialized:

GCC	an ELF with sections containing GIMPLE.
clang	a bitcode file containing the LLVM IR.

- The linker will merge these IRs and optimize them

Pretty much like...

```
cat *.c > all.c

gcc all.c -o all
```

Pros and cons

- Pros:
  - Global view on the program
  - Larger optimization opportunities
  - Aggressive dead code elimination
- Cons:
  - Resource intensive
  - Only a subset of the optimizations can be run

Security considerations

`-z relro` An attacker [1] could change the `.dynamic` section for malicious purposes. With `-z relro` enabled, once .dynamic has been initialized, it is marked read-only.

`-z now` An attacker [1] could use the lazy loading system to call an arbitrary function. `-z now` completely disables lazy loading.

`-pie` By default executable's position, unlike libraries, is not randomized. An attacker could then reuse code in the executable binary for malicious purposes. `-pie` compiles the executable as PIC and produces a relocatable program (similar to a shared library).

Thanks

Bibliography

- Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.
  How the ELF Ruined Christmas.
  In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658, Washington, D.C., August 2015. USENIX Association.
- Santa Cruz Operation.
  System V Application Binary Interface, 2013.
  `http://www.sco.com/developers/gabi/latest/contents.html`.

License