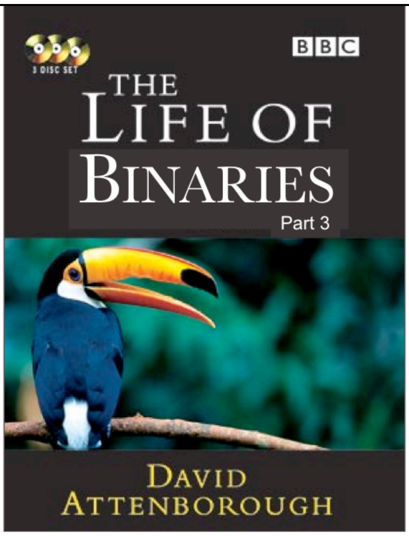


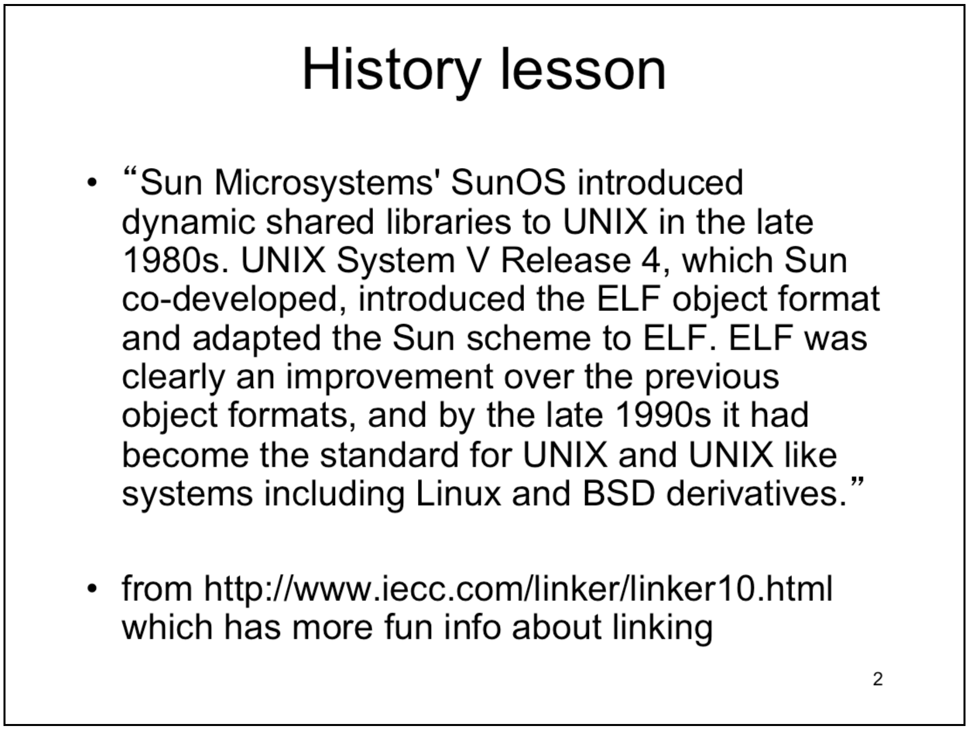
Xeno Kovah – 2012

xkovah at gmail

1

See notes for citation

Image from: http://www.bbcshop.com/content/ebiz/bbc/invt/bbcdvd1020/lobd300.jpg



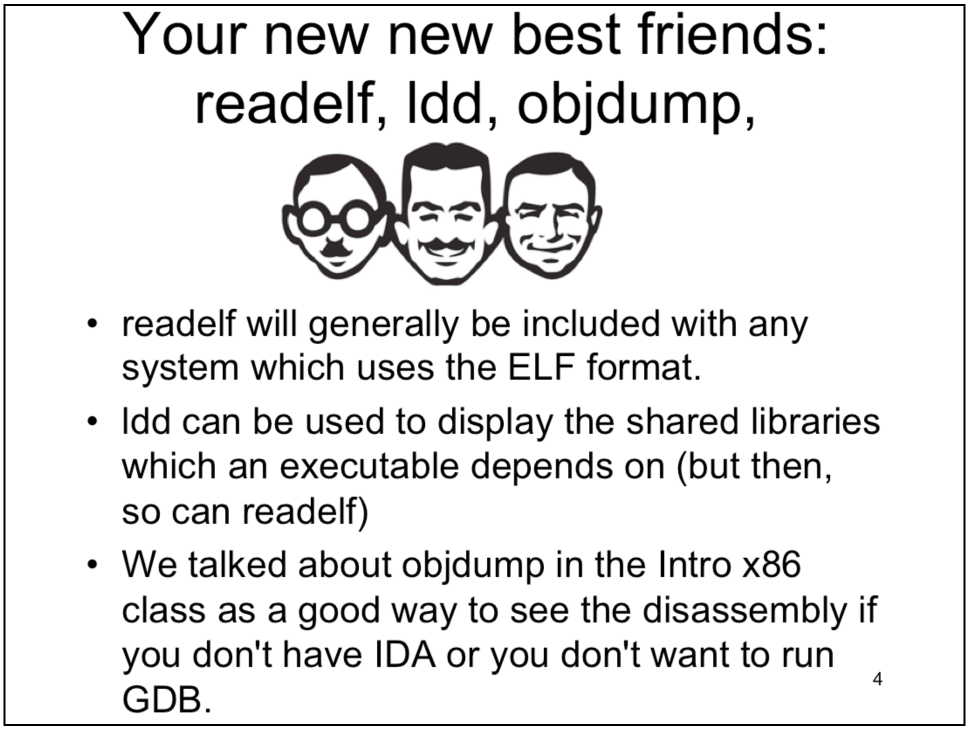
History lesson

- “Sun Microsystems' SunOS introduced dynamic shared libraries to UNIX in the late 1980s. UNIX System V Release 4, which Sun co-developed, introduced the ELF object format and adapted the Sun scheme to ELF. ELF was clearly an improvement over the previous object formats, and by the late 1990s it had become the standard for UNIX and UNIX like systems including Linux and BSD derivatives.”
- from http://www.iecc.com/linker/linker10.html which has more fun info about linking

Executable and Linkable Format (ELF)

- In this class we're going to focus on x86-64
  - See the previous class for the 32 bit notes
- Official Application Binary Interface (ABI)
  http://**www.sco.com**/developers/devspecs/gabi41.pdf
- x86 supplement - http://**www.sco.com**/developers/devspecs/abi386-4.pdf
- Start at chapter 4

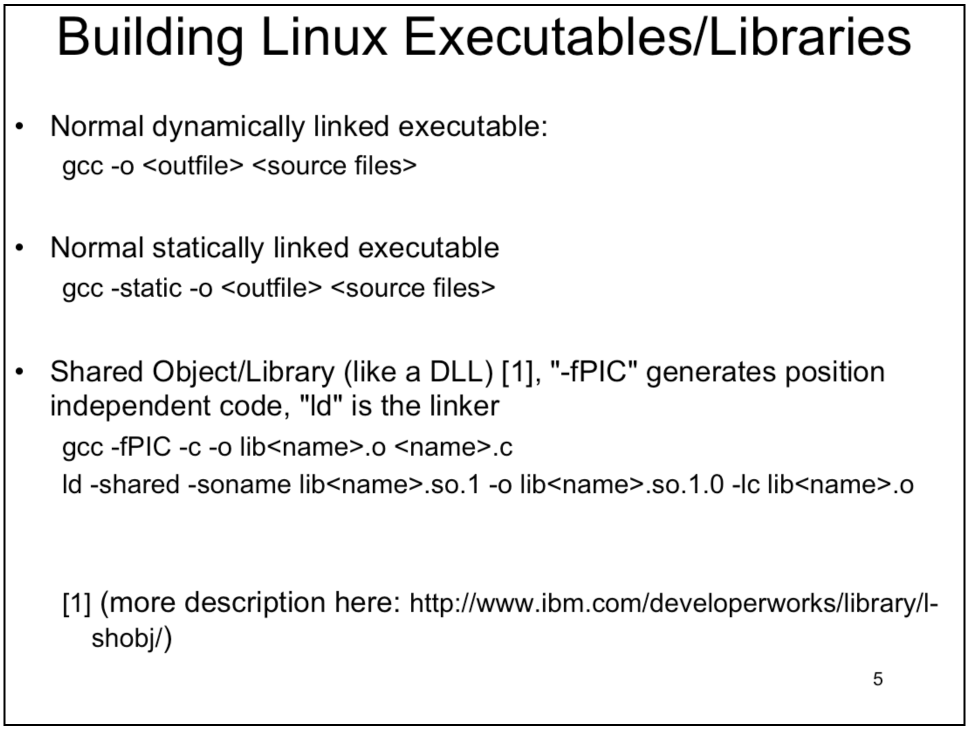
- **X86-64 - http://www.x86-64.org/documentation/abi.pdf (ABI Draft 0.99.6 - July 2, 2012)**
- April 24th 2001 draft update (supposedly widely used) http://refspecs.freestandards.org/elf/gabi4+/contents.html
- Oct 29th 2009 draft update http://www.sco.com/developers/gabi/2009-10-26/contents.html



Your new new best friends: readelf, ldd, objdump,

- readelf will generally be included with any system which uses the ELF format.
- ldd can be used to display the shared libraries which an executable depends on (but then, so can readelf)
- We talked about objdump in the Intro x86 class as a good way to see the disassembly if you don't have IDA or you don't want to run GDB.

http://WWW.busymom.net/reviews/pep_boys.jpg



Building Linux Executables/Libraries

- Normal dynamically linked executable:

```
gcc -o <outfile> <source files>
```

- Normal statically linked executable

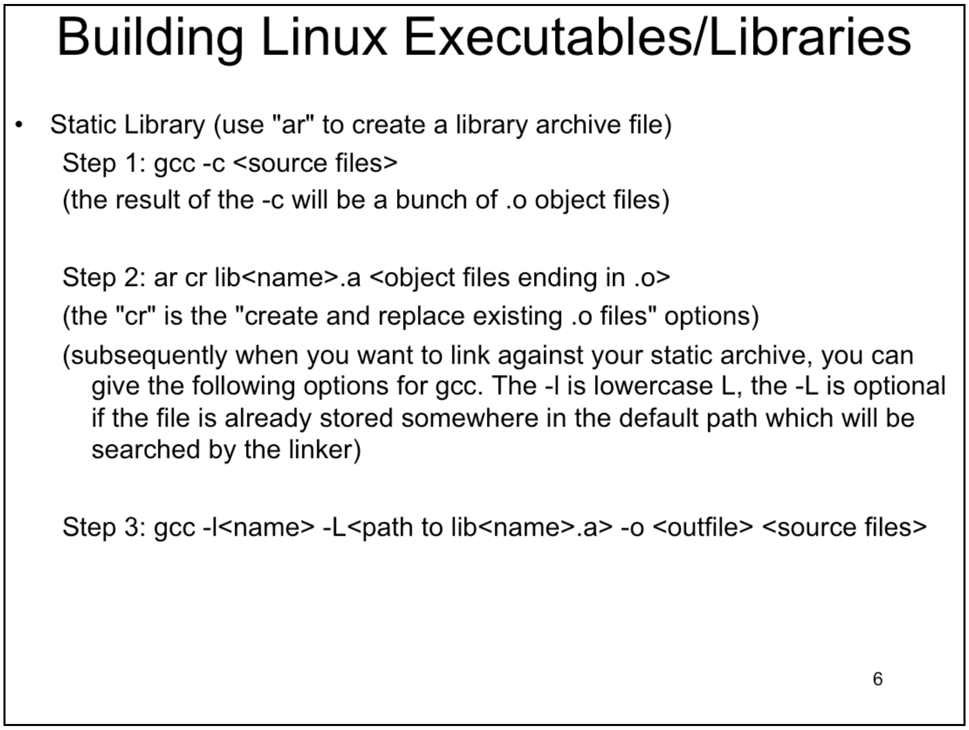
```
gcc -static -o <outfile> <source files>
```

- Shared Object/Library (like a DLL) [1], "-fPIC" generates position independent code, "ld" is the linker

```
gcc -fPIC -c -o lib<name>.o <name>.c
ld -shared -soname lib<name>.so.1 -o lib<name>.so.1.0 -lc lib<name>.o
```

[1] (more description here: http://www.ibm.com/developerworks/library/l-shobj/)

5



Building Linux Executables/Libraries

- Static Library (use "ar" to create a library archive file)

  Step 1: gcc -c <source files>

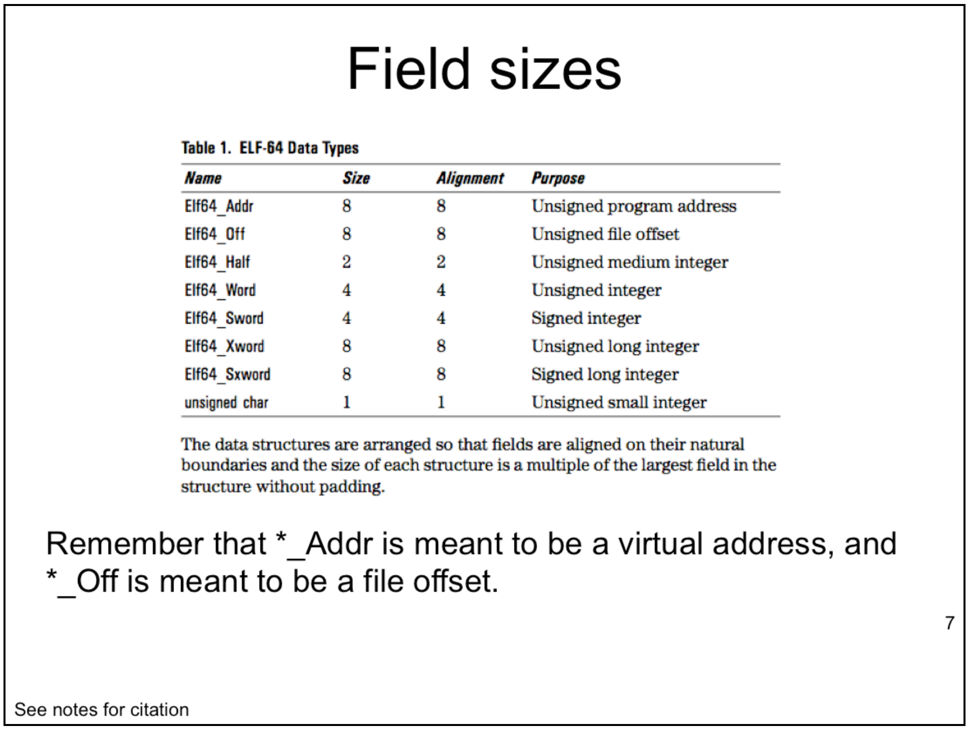
  (the result of the -c will be a bunch of .o object files)

  Step 2: ar cr lib<name>.a <object files ending in .o>

  (the "cr" is the "create and replace existing .o files" options)

  (subsequently when you want to link against your static archive, you can give the following options for gcc. The -l is lowercase L, the -L is optional if the file is already stored somewhere in the default path which will be searched by the linker)

  Step 3: gcc -l<name> -L<path to lib<name>.a> -o <outfile> <source files>



Field sizes

Table 1. ELF-64 Data Types

Name	*Size*	*Alignment*	*Purpose*
Elf64_Addr	8	8	Unsigned program address
Elf64_Off	8	8	Unsigned file offset
Elf64_Half	2	2	Unsigned medium integer
Elf64_Word	4	4	Unsigned integer
Elf64_Sword	4	4	Signed integer
Elf64_Xword	8	8	Unsigned long integer
Elf64_Sxword	8	8	Signed long integer
unsigned char	1	1	Unsigned small integer

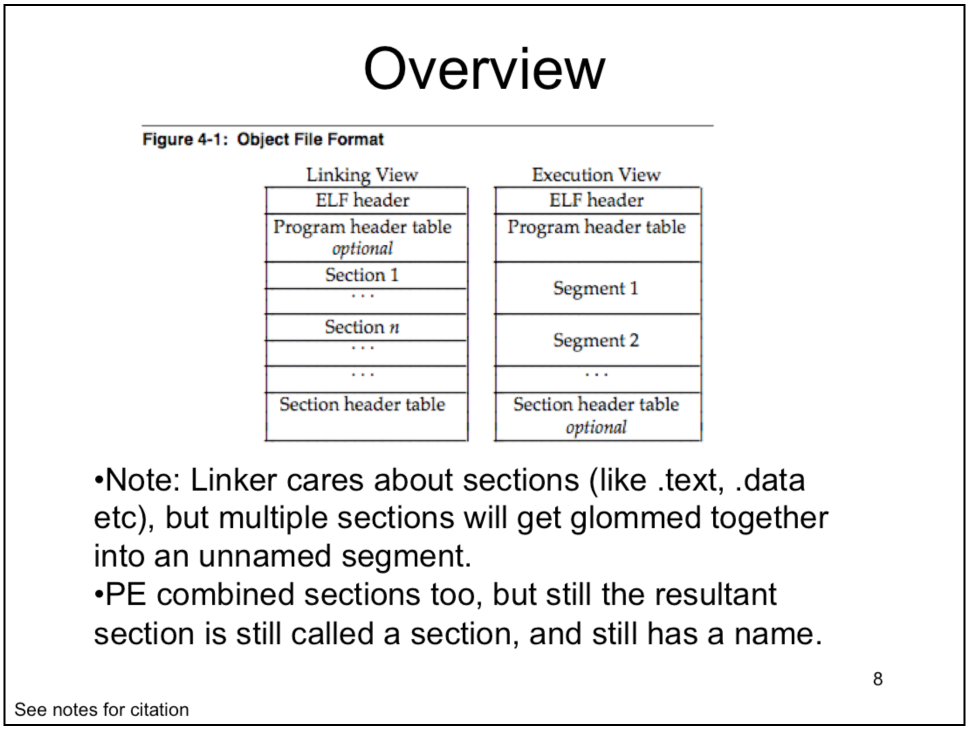
The data structures are arranged so that fields are aligned on their natural boundaries and the size of each structure is a multiple of the largest field in the structure without padding.

Remember that *_Addr is meant to be a virtual address, and *_Off is meant to be a file offset.

7

See notes for citation

Pic from http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf because for some reason the AMD64 Architecture Processor Supplement doesn't have this table



Overview

Figure 4-1: Object File Format

Linking View	Execution View
ELF header	ELF header
Program header table *optional*	Program header table
Section 1	Segment 1
. . .	
Section *n*	Segment 2
. . .	
. . .	. . .
Section header table	Section header table *optional*

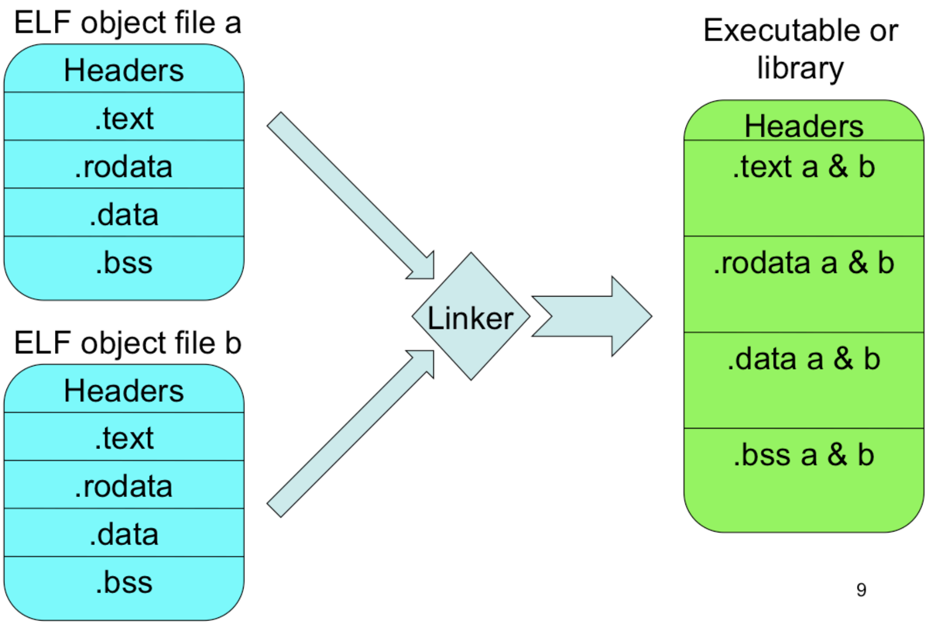
- Note: Linker cares about sections (like .text, .data etc), but multiple sections will get glommed together into an unnamed segment.
- PE combined sections too, but still the resultant section is still called a section, and still has a name.

8

See notes for citation

http://www.sco.com/developers/devspecs/gabi41.pdf

Linker Overview



ELF object file a

Headers

.text

.rodata

.data

.bss

ELF object file b

Headers

.text

.rodata

.data

.bss

Linker

Executable or library

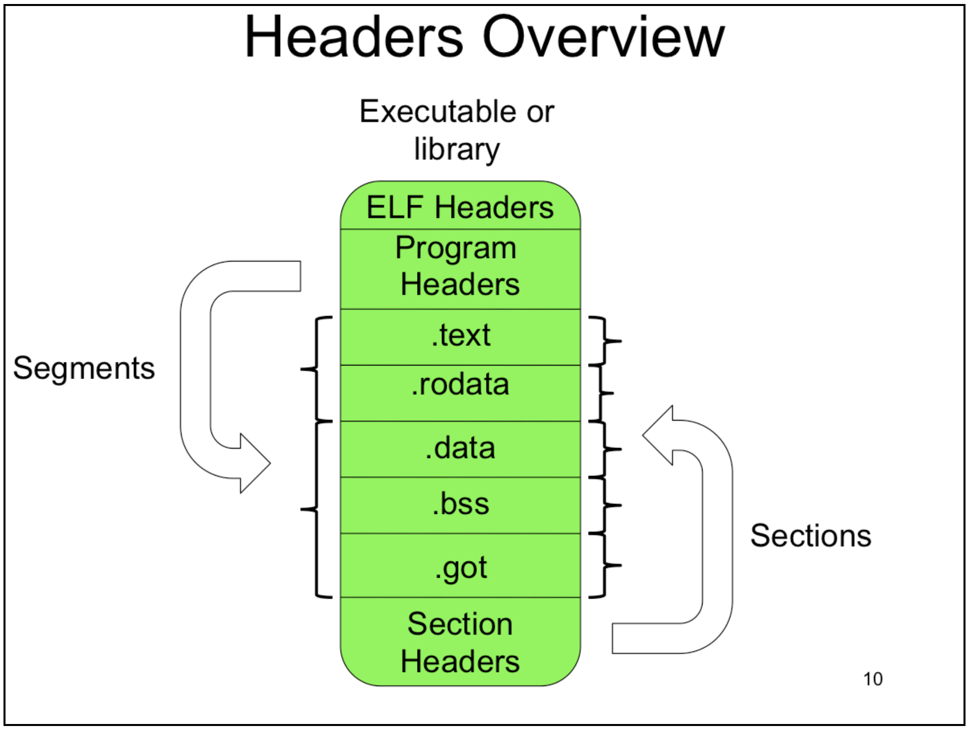
Headers

.text a & b

.rodata a & b

.data a & b

.bss a & b



Headers Overview

Executable or library

ELF Headers

Program Headers

.text

.rodata

.data

.bss

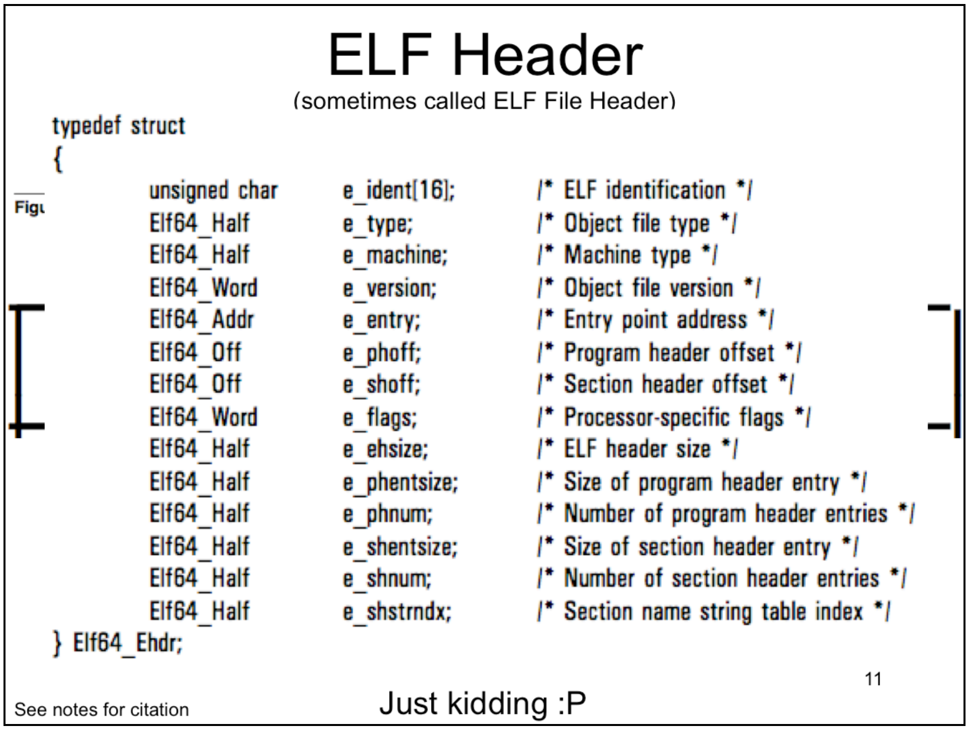
.got

Section Headers

Segments

Sections

10



ELF Header

(sometimes called ELF File Header)

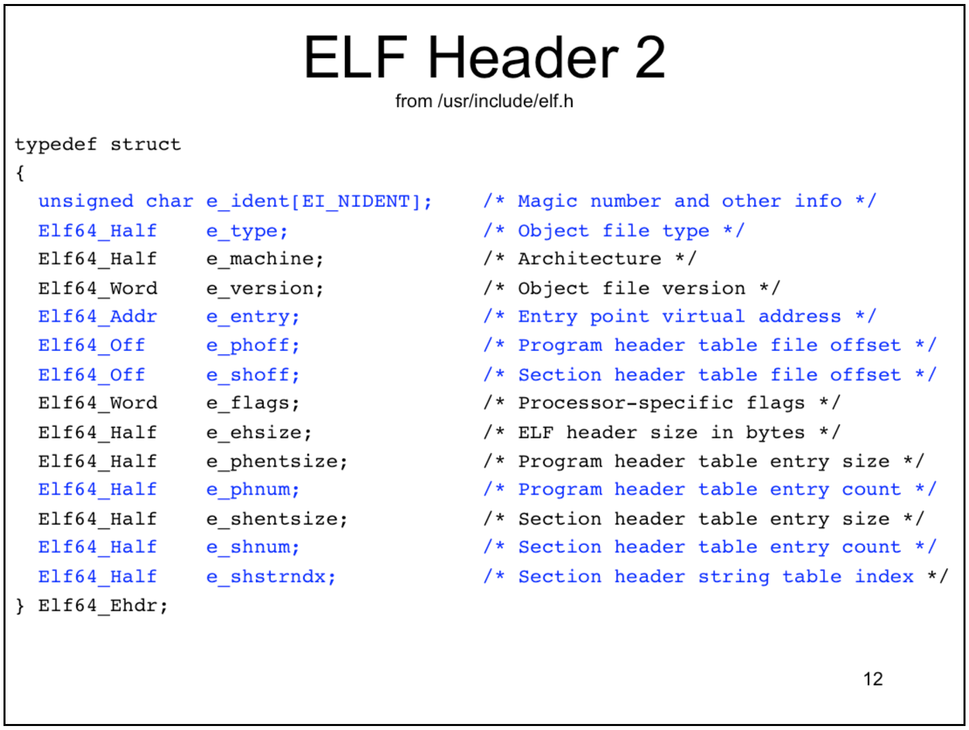
```
typedef struct
{
        unsigned char   e_ident[16];    /* ELF identification */
        Elf64_Half      e_type;         /* Object file type */
        Elf64_Half      e_machine;      /* Machine type */
        Elf64_Word      e_version;      /* Object file version */
        Elf64_Addr      e_entry;        /* Entry point address */
        Elf64_Off       e_phoff;        /* Program header offset */
        Elf64_Off       e_shoff;        /* Section header offset */
        Elf64_Word      e_flags;        /* Processor-specific flags */
        Elf64_Half      e_ehsize;       /* ELF header size */
        Elf64_Half      e_phentsize;    /* Size of program header entry */
        Elf64_Half      e_phnum;        /* Number of program header entries */
        Elf64_Half      e_shentsize;    /* Size of section header entry */
        Elf64_Half      e_shnum;        /* Number of section header entries */
        Elf64_Half      e_shstrndx;     /* Section name string table index */
} Elf64_Ehdr;
```

Figu

See notes for citation

Just kidding :P

http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf



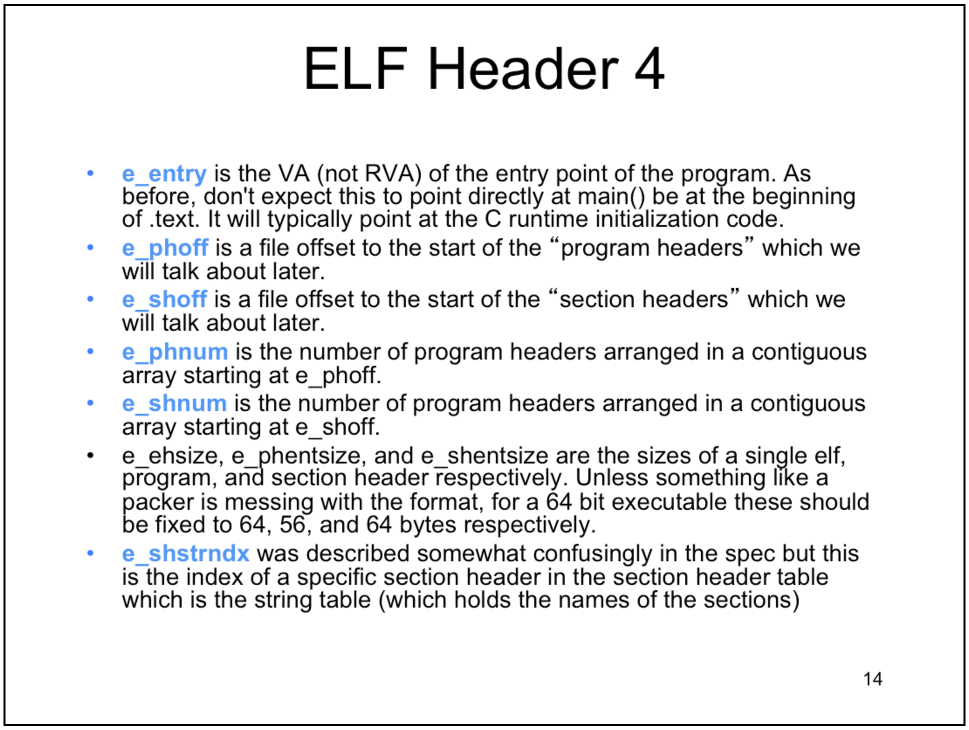
ELF Header 2

from /usr/include/elf.h

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];    /* Magic number and other info */
  Elf64_Half    e_type;                /* Object file type */
  Elf64_Half    e_machine;             /* Architecture */
  Elf64_Word    e_version;             /* Object file version */
  Elf64_Addr    e_entry;               /* Entry point virtual address */
  Elf64_Off     e_phoff;               /* Program header table file offset */
  Elf64_Off     e_shoff;               /* Section header table file offset */
  Elf64_Word    e_flags;               /* Processor-specific flags */
  Elf64_Half    e_ehsize;              /* ELF header size in bytes */
  Elf64_Half    e_phentsize;           /* Program header table entry size */
  Elf64_Half    e_phnum;               /* Program header table entry count */
  Elf64_Half    e_shentsize;           /* Section header table entry size */
  Elf64_Half    e_shnum;               /* Section header table entry count */
  Elf64_Half    e_shstrndx;            /* Section header string table index */
} Elf64_Ehdr;
```

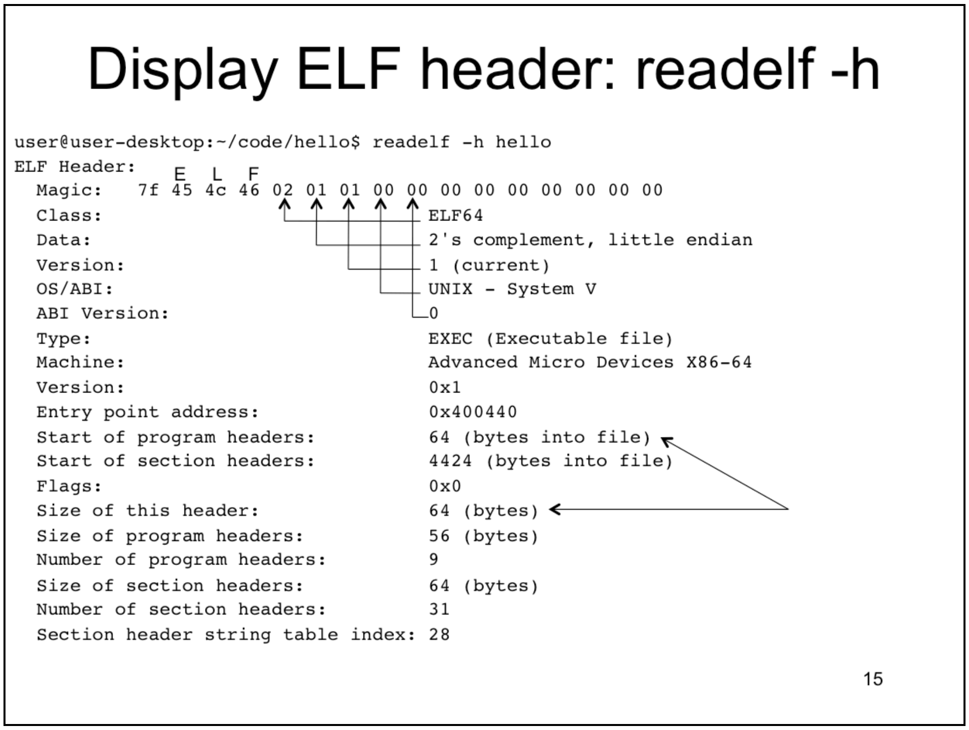
ELF Header 3

- **e_ident**[] starts with the magic number which is [0] = 0x7f, [1] = 'E', [2] = 'L', [3] = 'F'. This would be equivalent to the MZ signature at the start of PE files. Come on man, why's it always got to be about PEs? Hey, I'm just saying is all. Don't bite my head off. Whatever man, INFORMATION WANTS TO BE FREE! OPEN SOURCE FOREVER! Are you done? Yes. Good. There is other data encoded in e_ident, but we don't care about it that much and you can just look it up in the ABI if you're interested.
- **e_type** values that we care about are ET_REL (1) a relocatable file, ET_EXEC (2) an executable, ET_DYN (3) a shared object, and maybe ET_CORE (4)



ELF Header 4

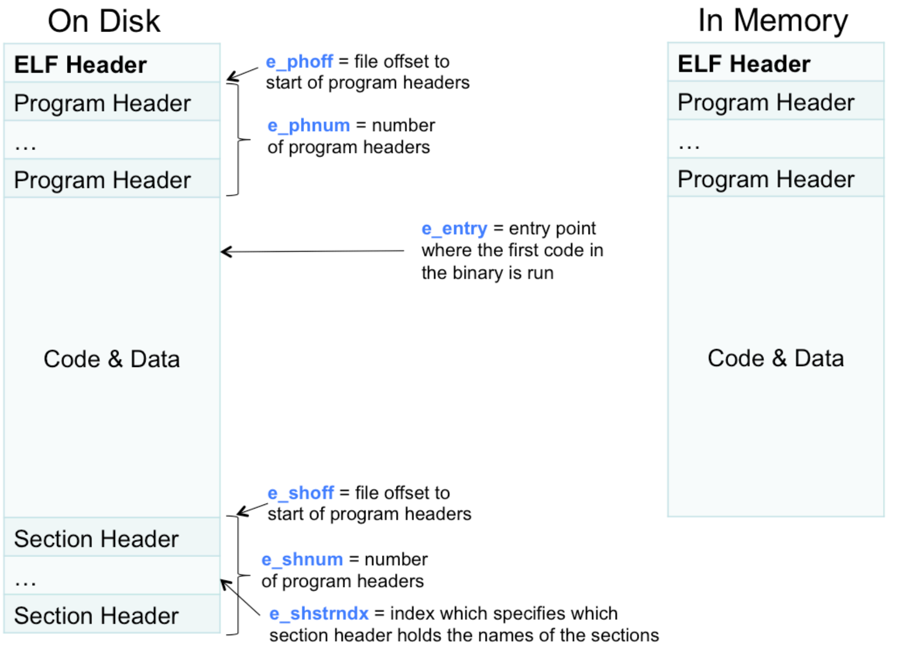
- **e_entry** is the VA (not RVA) of the entry point of the program. As before, don't expect this to point directly at main() be at the beginning of .text. It will typically point at the C runtime initialization code.
- **e_phoff** is a file offset to the start of the “program headers” which we will talk about later.
- **e_shoff** is a file offset to the start of the “section headers” which we will talk about later.
- **e_phnum** is the number of program headers arranged in a contiguous array starting at e_phoff.
- **e_shnum** is the number of program headers arranged in a contiguous array starting at e_shoff.
- e_ehsize, e_phentsize, and e_shentsize are the sizes of a single elf, program, and section header respectively. Unless something like a packer is messing with the format, for a 64 bit executable these should be fixed to 64, 56, and 64 bytes respectively.
- **e_shstrndx** was described somewhat confusingly in the spec but this is the index of a specific section header in the section header table which is the string table (which holds the names of the sections)



Display ELF header: readelf -h

```
user@user-desktop:~/code/hello$ readelf -h hello
ELF Header:    E  L  F
  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:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400440
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4424 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
```

ELF Header Fields



e_phoff = file offset to start of program headers

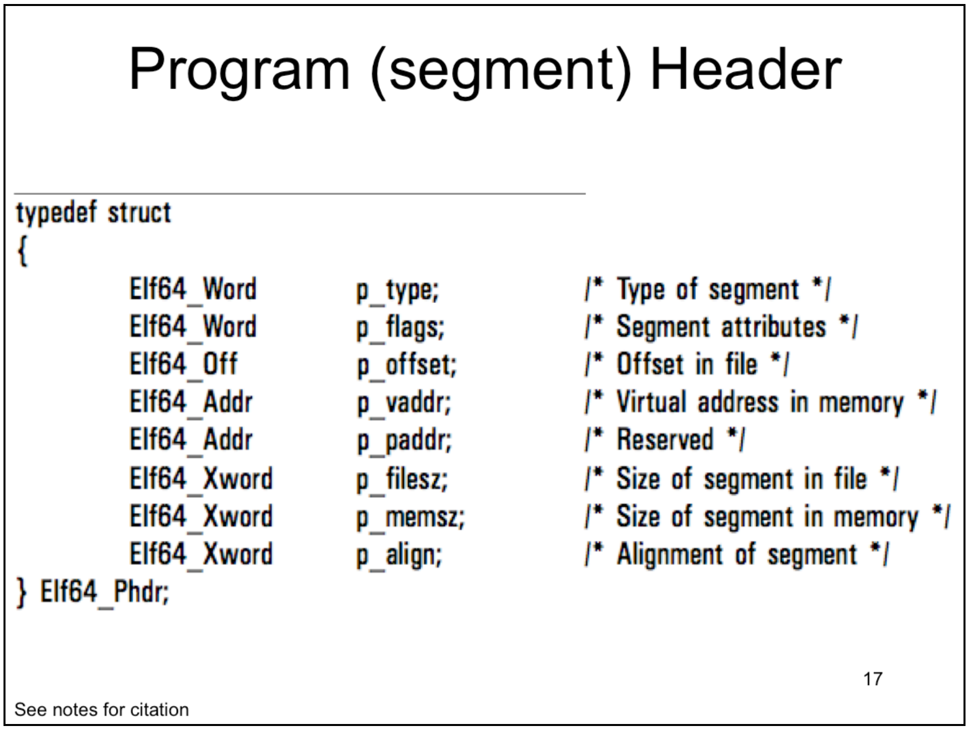
e_phnum = number of program headers

e_entry = entry point where the first code in the binary is run

e_shoff = file offset to start of program headers

e_shnum = number of program headers

e_shstrndx = index which specifies which section header holds the names of the sections

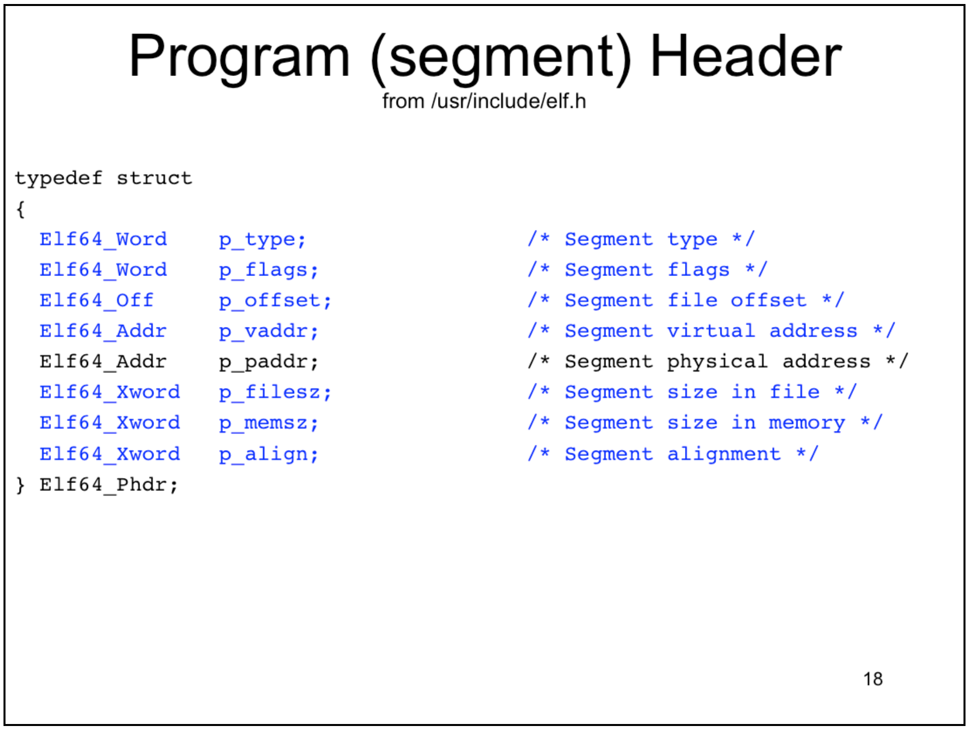


Program (segment) Header

```
typedef struct
{
        Elf64_Word      p_type;         /* Type of segment */
        Elf64_Word      p_flags;        /* Segment attributes */
        Elf64_Off       p_offset;       /* Offset in file */
        Elf64_Addr      p_vaddr;        /* Virtual address in memory */
        Elf64_Addr      p_paddr;        /* Reserved */
        Elf64_Xword     p_filesz;       /* Size of segment in file */
        Elf64_Xword     p_memsz;        /* Size of segment in memory */
        Elf64_Xword     p_align;        /* Alignment of segment */
} Elf64_Phdr;
```

See notes for citation

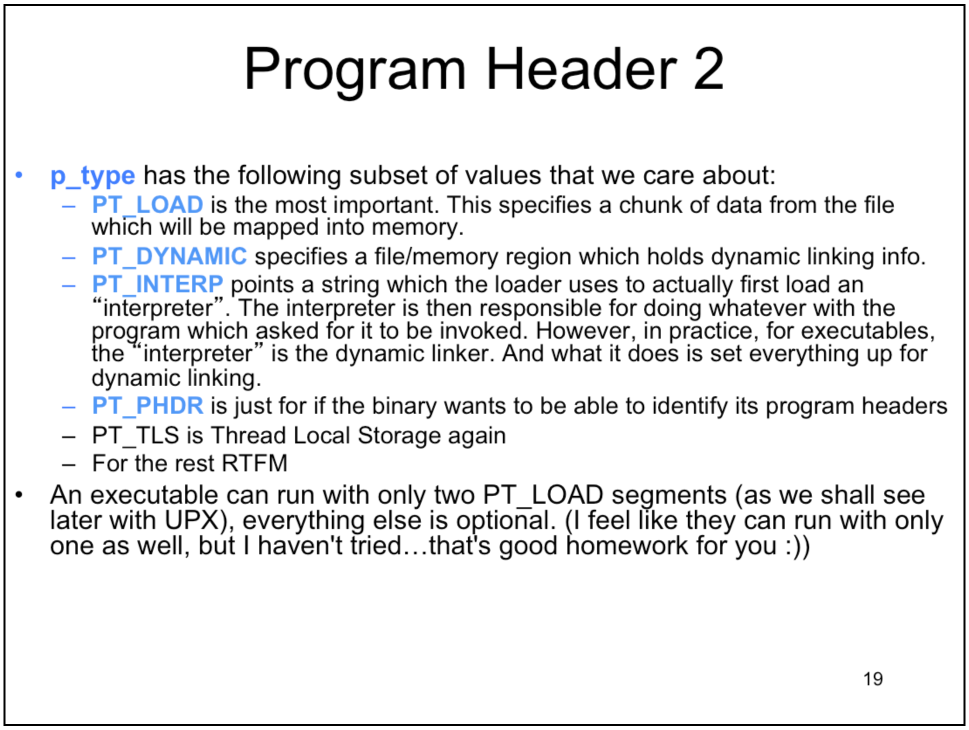
http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf



Program (segment) Header

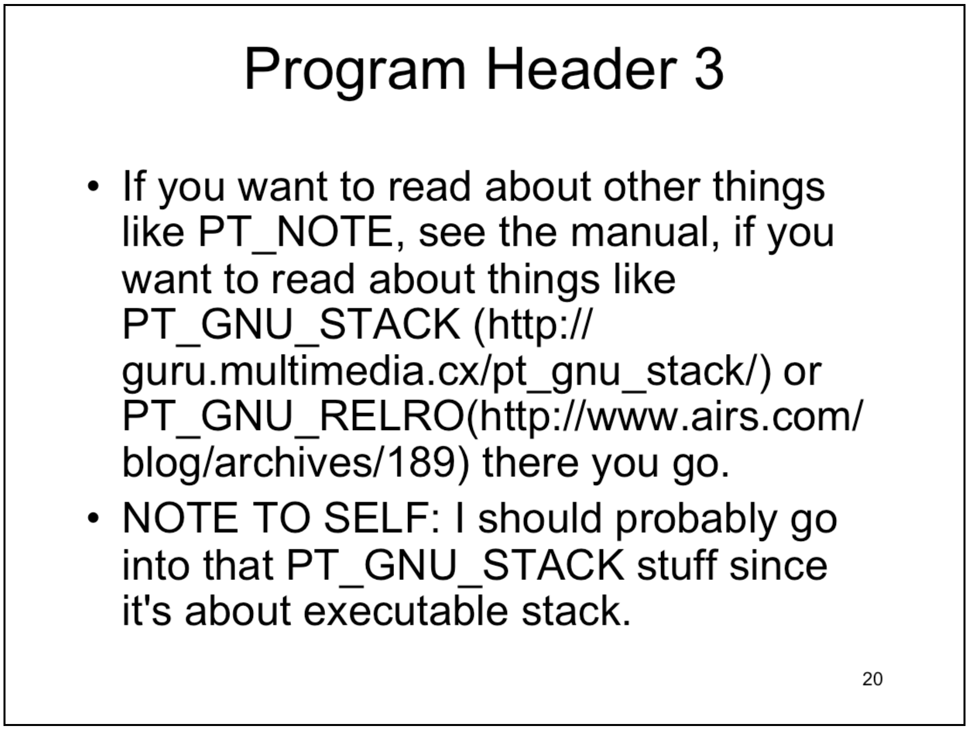
from /usr/include/elf.h

```
typedef struct
{
  Elf64_Word    p_type;                 /* Segment type */
  Elf64_Word    p_flags;                /* Segment flags */
  Elf64_Off     p_offset;               /* Segment file offset */
  Elf64_Addr    p_vaddr;                /* Segment virtual address */
  Elf64_Addr    p_paddr;                /* Segment physical address */
  Elf64_Xword   p_filesz;               /* Segment size in file */
  Elf64_Xword   p_memsz;                /* Segment size in memory */
  Elf64_Xword   p_align;                /* Segment alignment */
} Elf64_Phdr;
```



Program Header 2

- **p_type** has the following subset of values that we care about:
  - **PT_LOAD** is the most important. This specifies a chunk of data from the file which will be mapped into memory.
  - **PT_DYNAMIC** specifies a file/memory region which holds dynamic linking info.
  - **PT_INTERP** points a string which the loader uses to actually first load an “interpreter”. The interpreter is then responsible for doing whatever with the program which asked for it to be invoked. However, in practice, for executables, the “interpreter” is the dynamic linker. And what it does is set everything up for dynamic linking.
  - **PT_PHDR** is just for if the binary wants to be able to identify its program headers
  - PT_TLS is Thread Local Storage again
  - For the rest RTFM
- An executable can run with only two PT_LOAD segments (as we shall see later with UPX), everything else is optional. (I feel like they can run with only one as well, but I haven't tried…that's good homework for you :))



Program Header 3

- If you want to read about other things like PT_NOTE, see the manual, if you want to read about things like PT_GNU_STACK (http://guru.multimedia.cx/pt_gnu_stack/) or PT_GNU_RELRO(http://www.airs.com/blog/archives/189) there you go.
- NOTE TO SELF: I should probably go into that PT_GNU_STACK stuff since it's about executable stack.

Program Header 4

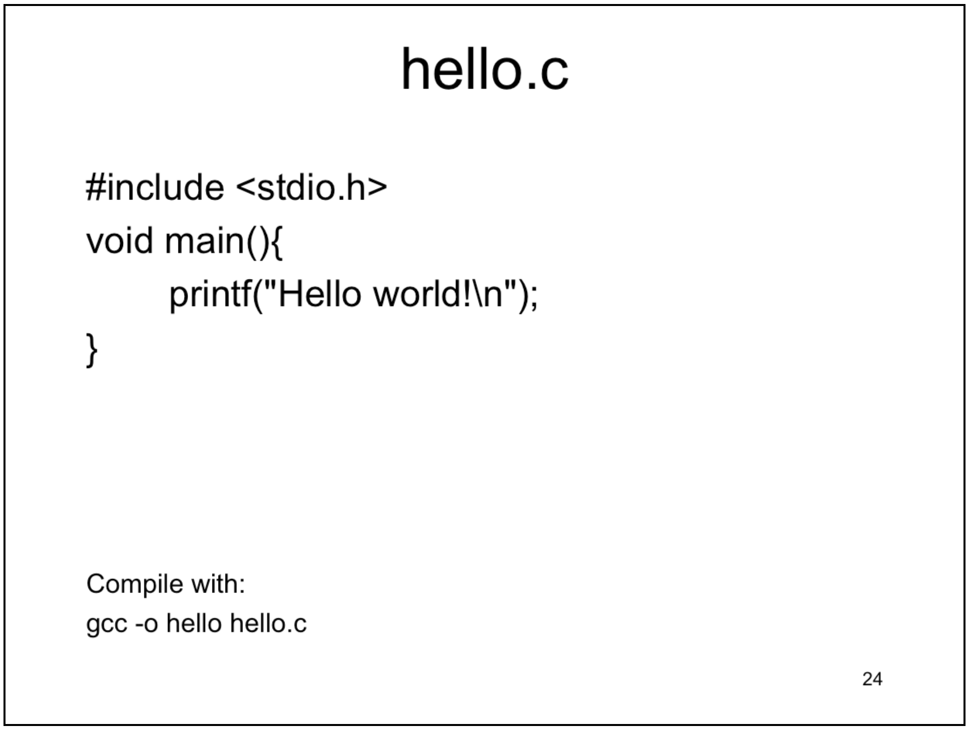
- **p_offset** is where the data you want to map into memory starts in the file. But again, remember, only for PT_LOAD segments does data actually get read from the file and mapped into memory.
- **p_vaddr** is the virtual address where this segment will be mapped at.
- p_paddr is supposed to be the physical address, but "Because System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects." (from the ABI) This is frequently set to the same thing as p_vaddr, but you can basically just ignore it.

Program Header 5

- **p_filesz** is how much data is read from the file and mapped into memory.
- **p_memsz** is the size of the virtual memory allocation for this segment. If p_memsz is > p_filesz, the extra space is used for the .bss area. (Remember that conceptually the .bss section is all about allocating some space in memory for uninitialized variables which you don't need/ want to store in the file.)
- Also recall that the only way that the size in memory (IMAGE_SECTION_HEADER.Misc.VirtualSize) could be smaller than the size on disk (IMAGE_SECTION_HEADER.RawData) for PE files was due to the RawData being small and being aligned up to a multiple of the file alignment size. ELF has no notion of file alignment
  - therefore the following always is true: p_filesz <= p_memsz.

Program Header 6

- **p_flags** are the memory permission flags.
- PF_R = 0x4, PF_W = 0x2, PF_X = 0x1, or any number of processor-specific things as long as the MSB is set (which can be checked by ANDing with PF_MASKPROC). So the lower 3 bits are RWX like normal UNIX filesystem permissions.
- **p_align** is the segment alignment. The segment must start on a virtual address which is a multiple of this value. For normal PT_LOAD segments, the alignment is 0x1000.
- In contrast, PE sections don't *have* to be 0x1000 aligned, so it's sort of like ELF enforces memory alignment while PE doesn't, and PE enforces file alignment while ELF doesn't.

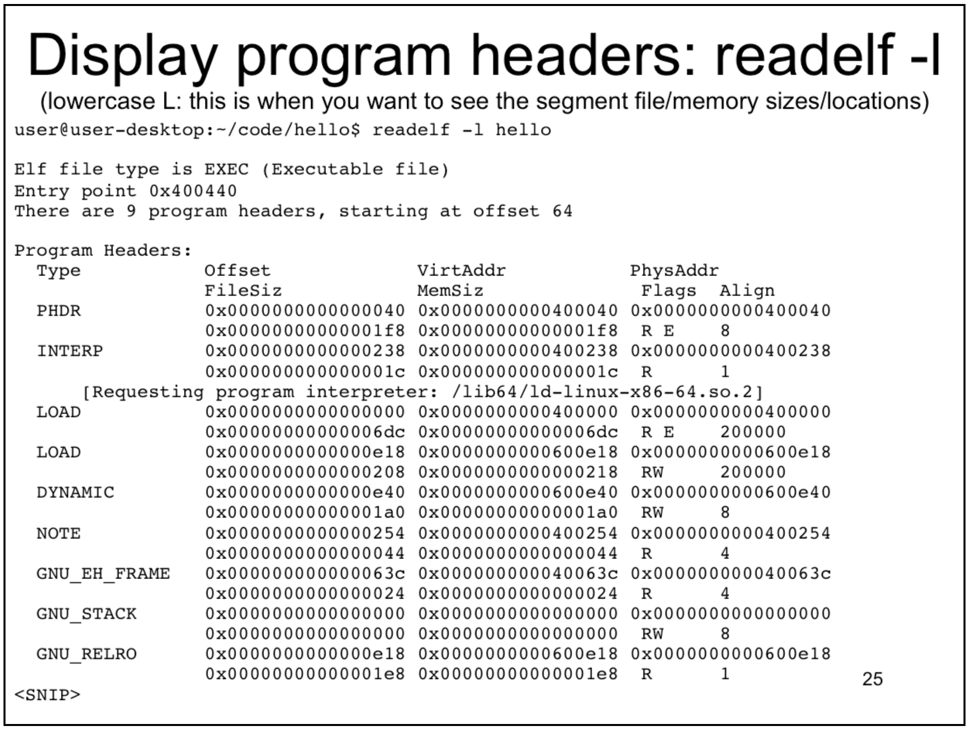


hello.c

```
#include <stdio.h>
void main(){
	printf("Hello world!\n");
}
```

Compile with:

```
gcc -o hello hello.c
```



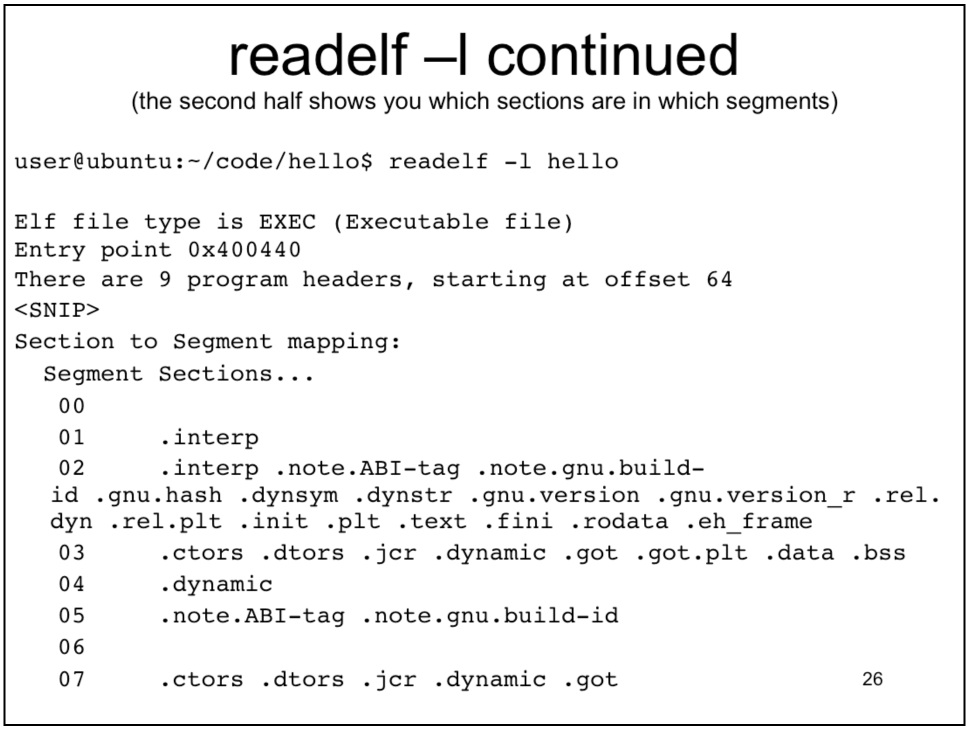
Display program headers: readelf -l

(lowercase L: this is when you want to see the segment file/memory sizes/locations)

```
user@user-desktop:~/code/hello$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x400440
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006dc 0x00000000000006dc  R E    200000
  LOAD           0x0000000000000e18 0x0000000000600e18 0x0000000000600e18
                 0x0000000000000208 0x0000000000000218  RW     200000
  DYNAMIC        0x0000000000000e40 0x0000000000600e40 0x0000000000600e40
                 0x00000000000001a0 0x00000000000001a0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x000000000000063c 0x000000000040063c 0x000000000040063c
                 0x0000000000000024 0x0000000000000024  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8
  GNU_RELRO      0x0000000000000e18 0x0000000000600e18 0x0000000000600e18
                 0x00000000000001e8 0x00000000000001e8  R      1
<SNIP>
```

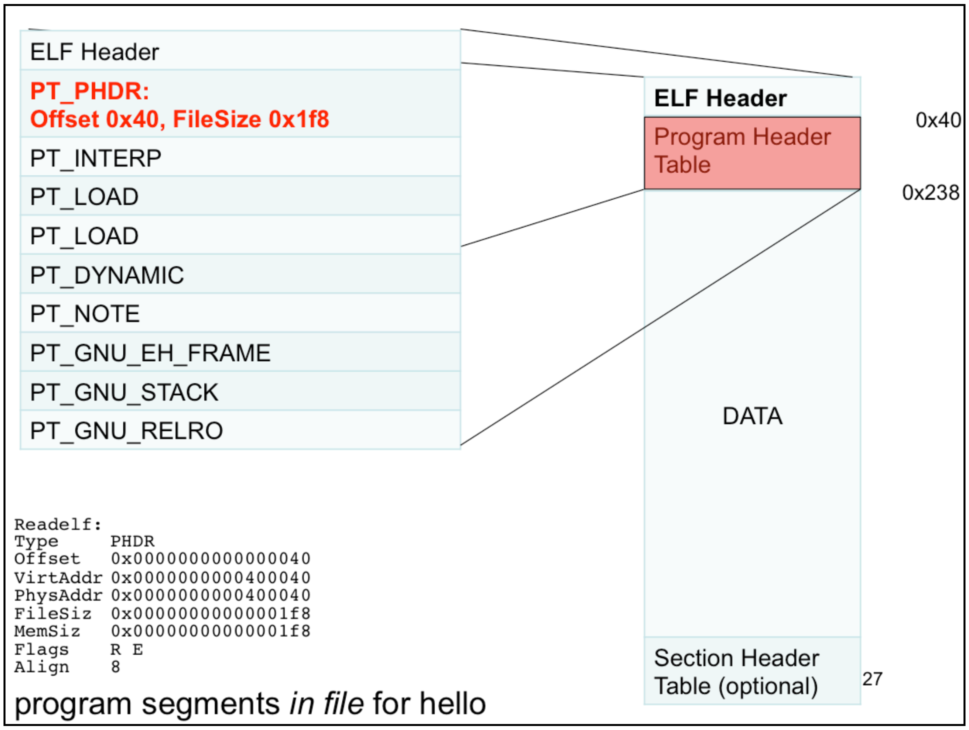


readelf –l continued

(the second half shows you which sections are in which segments)

```
user@ubuntu:~/code/hello$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x400440
There are 9 program headers, starting at offset 64
<SNIP>
Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-
  id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.
  dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06
   07     .ctors .dtors .jcr .dynamic .got
```



ELF Header

PT_PHDR:
Offset 0x40, FileSize 0x1f8

PT_INTERP

PT_LOAD

PT_LOAD

PT_DYNAMIC

PT_NOTE

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

ELF Header

Program Header Table

0x40

0x238

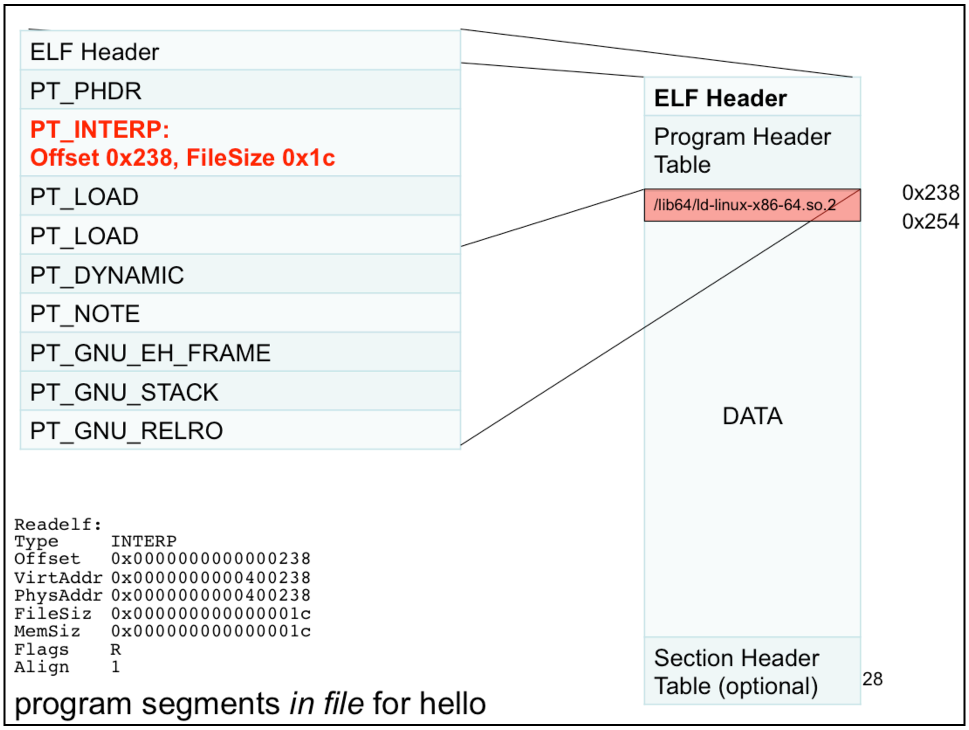
DATA

Section Header Table (optional)

```
Readelf:
Type     PHDR
Offset   0x0000000000000040
VirtAddr 0x0000000000400040
PhysAddr 0x0000000000400040
FileSiz  0x00000000000001f8
MemSiz   0x00000000000001f8
Flags    R E
Align    8
```

program segments *in file* for hello

27



ELF Header

PT_PHDR

PT_INTERP:
Offset 0x238, FileSize 0x1c

PT_LOAD

PT_LOAD

PT_DYNAMIC

PT_NOTE

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

ELF Header

Program Header Table

/lib64/ld-linux-x86-64.so.2

0x238

0x254

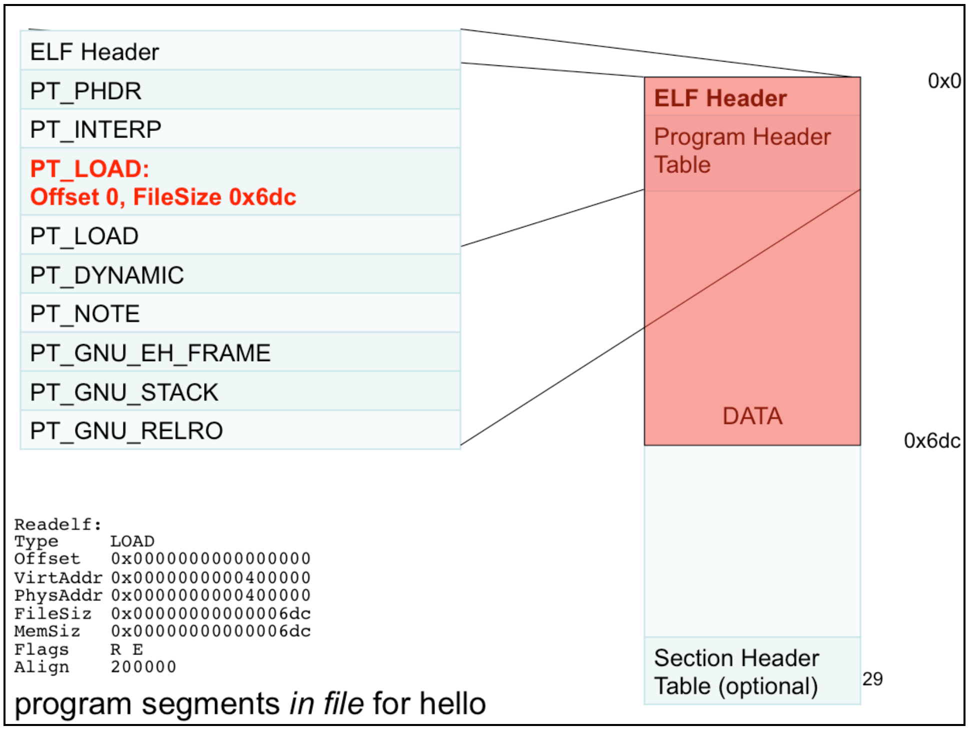
DATA

Section Header Table (optional)

```
Readelf:
Type     INTERP
Offset   0x0000000000000238
VirtAddr 0x0000000000400238
PhysAddr 0x0000000000400238
FileSiz  0x000000000000001c
MemSiz   0x000000000000001c
Flags    R
Align    1
```

program segments *in file* for hello

28



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD:
Offset 0, FileSize 0x6dc

PT_LOAD

PT_DYNAMIC

PT_NOTE

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

ELF Header

Program Header Table

DATA

Section Header Table (optional)

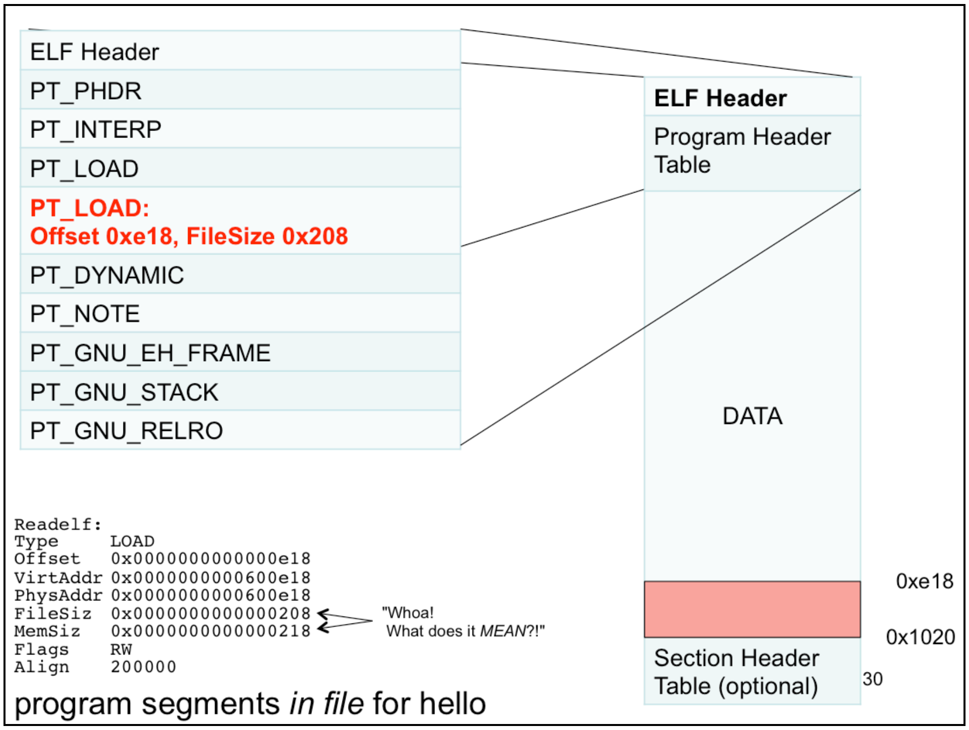
0x0

0x6dc

```
Readelf:
Type     LOAD
Offset   0x0000000000000000
VirtAddr 0x0000000000400000
PhysAddr 0x0000000000400000
FileSiz  0x00000000000006dc
MemSiz   0x00000000000006dc
Flags    R E
Align    200000
```

program segments *in file* for hello

29



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD

PT_LOAD:
Offset 0xe18, FileSize 0x208

PT_DYNAMIC

PT_NOTE

PT_GNU_EH_FRAME

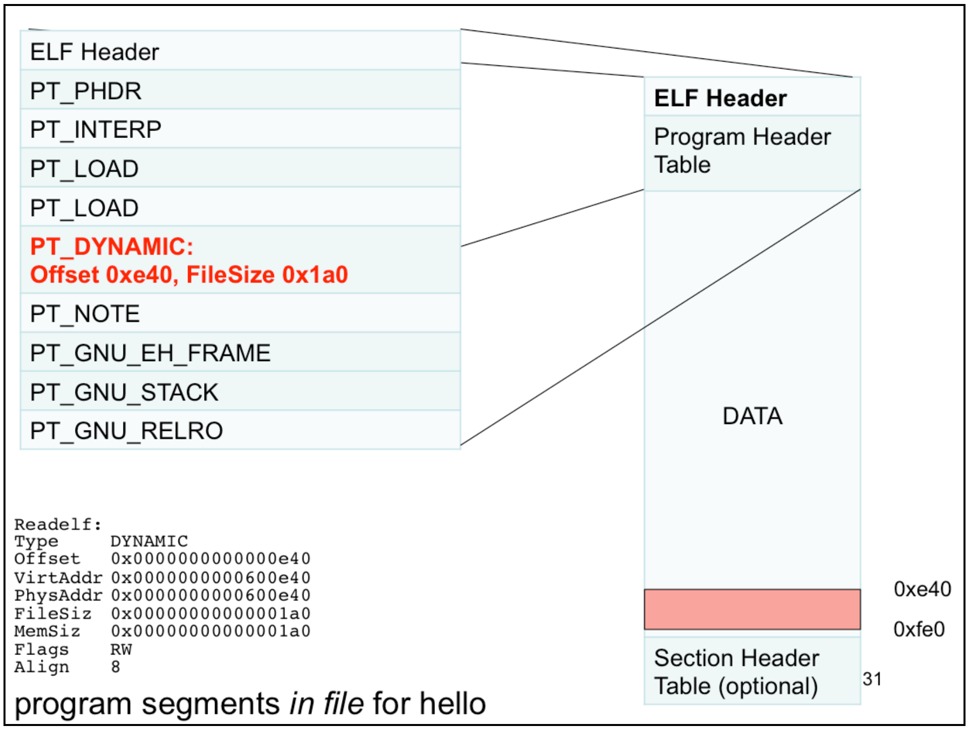
PT_GNU_STACK

PT_GNU_RELRO

```
Readelf:
Type     LOAD
Offset   0x0000000000000e18
VirtAddr 0x0000000000600e18
PhysAddr 0x0000000000600e18
FileSiz  0x0000000000000208
MemSiz   0x0000000000000218
Flags    RW
Align    200000
```

"Whoa!
What does it *MEAN*?!"

program segments *in file* for hello



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD

PT_LOAD

PT_DYNAMIC:
Offset 0xe40, FileSize 0x1a0

PT_NOTE

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

ELF Header

Program Header Table

DATA

0xe40

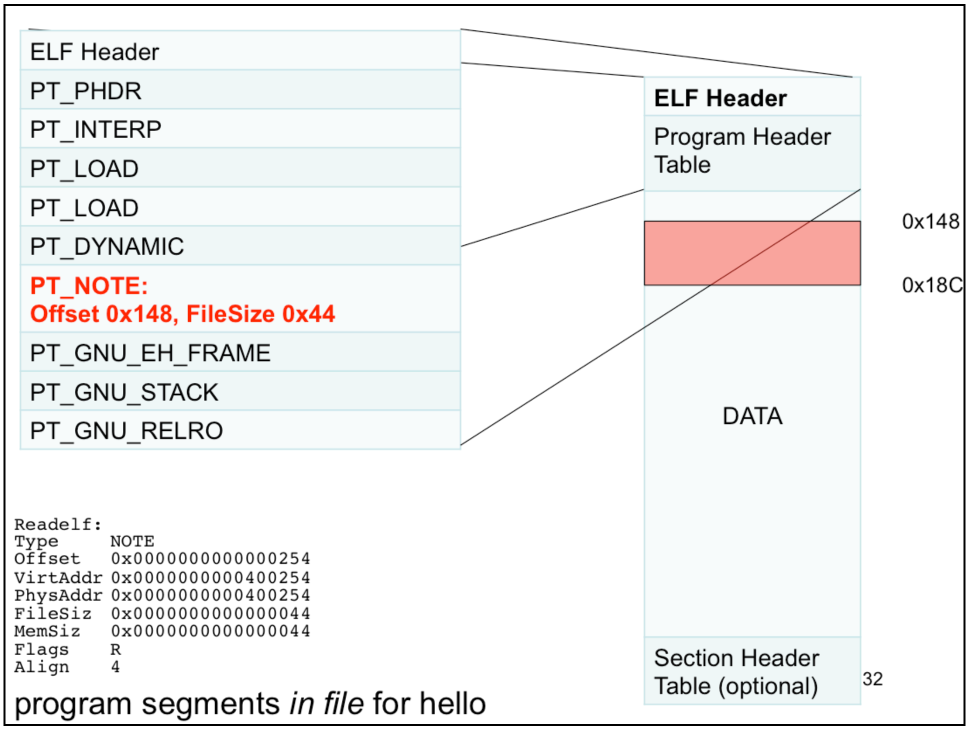
0xfe0

Section Header Table (optional)

```
Readelf:
Type     DYNAMIC
Offset   0x0000000000000e40
VirtAddr 0x0000000000600e40
PhysAddr 0x0000000000600e40
FileSiz  0x00000000000001a0
MemSiz   0x00000000000001a0
Flags    RW
Align    8
```

program segments *in file* for hello

31



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD

PT_LOAD

PT_DYNAMIC

PT_NOTE:
Offset 0x148, FileSize 0x44

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

ELF Header

Program Header Table

0x148

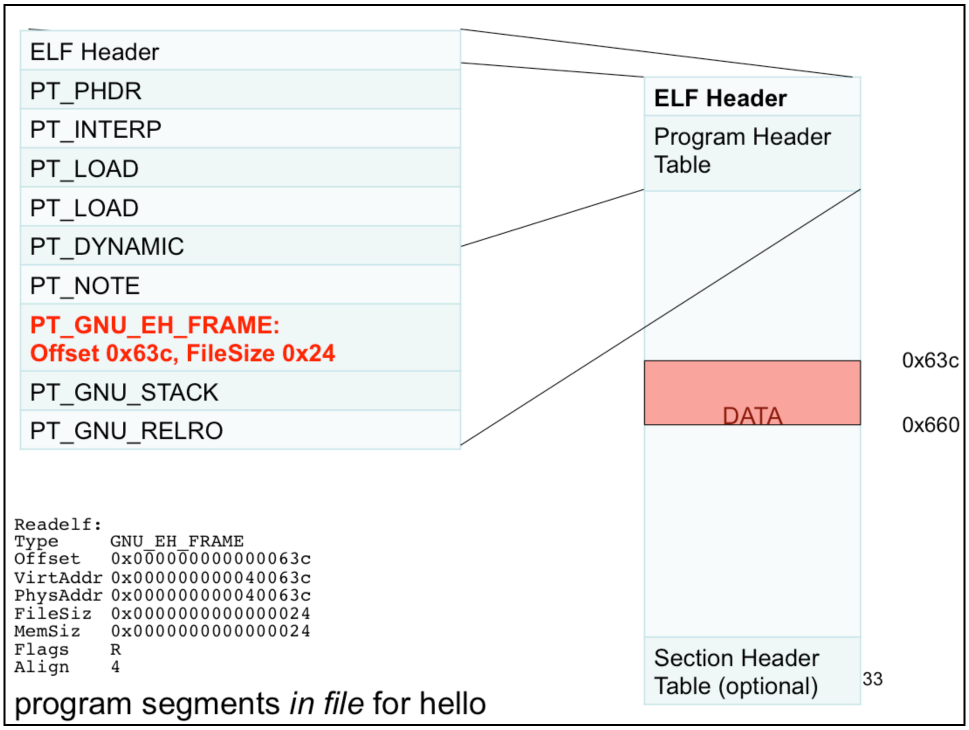
0x18C

DATA

Section Header Table (optional)

```
Readelf:
Type     NOTE
Offset   0x0000000000000254
VirtAddr 0x0000000000400254
PhysAddr 0x0000000000400254
FileSiz  0x0000000000000044
MemSiz   0x0000000000000044
Flags    R
Align    4
```

program segments *in file* for hello



ELF Header
PT_PHDR
PT_INTERP
PT_LOAD
PT_LOAD
PT_DYNAMIC
PT_NOTE
PT_GNU_EH_FRAME:
Offset 0x63c, FileSize 0x24
PT_GNU_STACK
PT_GNU_RELRO

ELF Header
Program Header Table

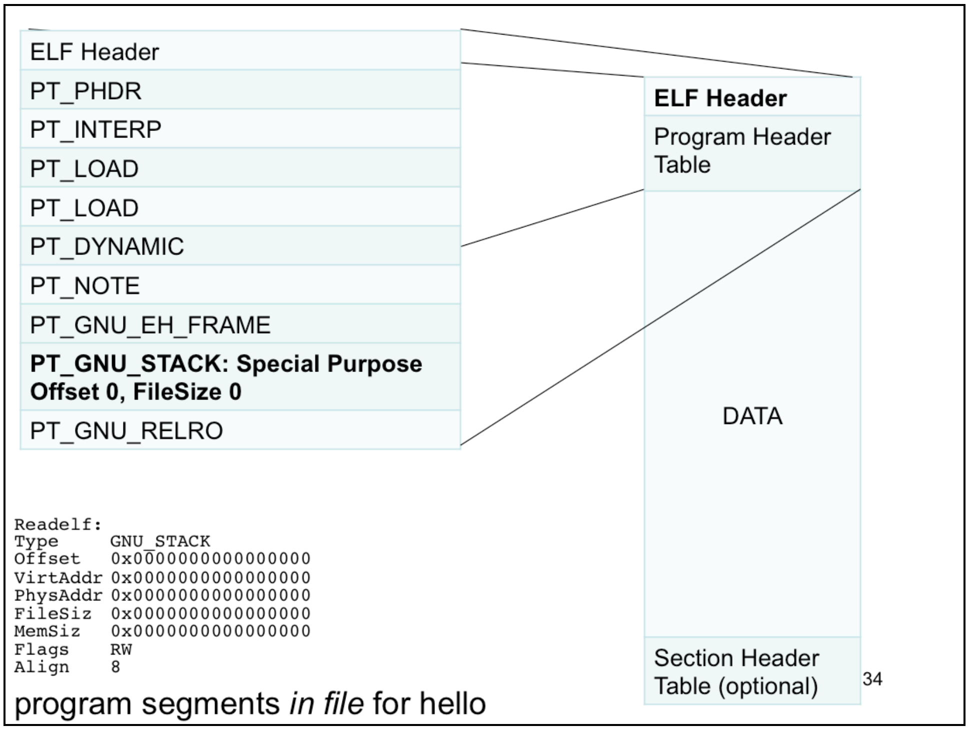
0x63c
DATA
0x660

Section Header Table (optional)

```
Readelf:
Type     GNU_EH_FRAME
Offset   0x000000000000063c
VirtAddr 0x000000000040063c
PhysAddr 0x000000000040063c
FileSiz  0x0000000000000024
MemSiz   0x0000000000000024
Flags    R
Align    4
```

program segments *in file* for hello

33



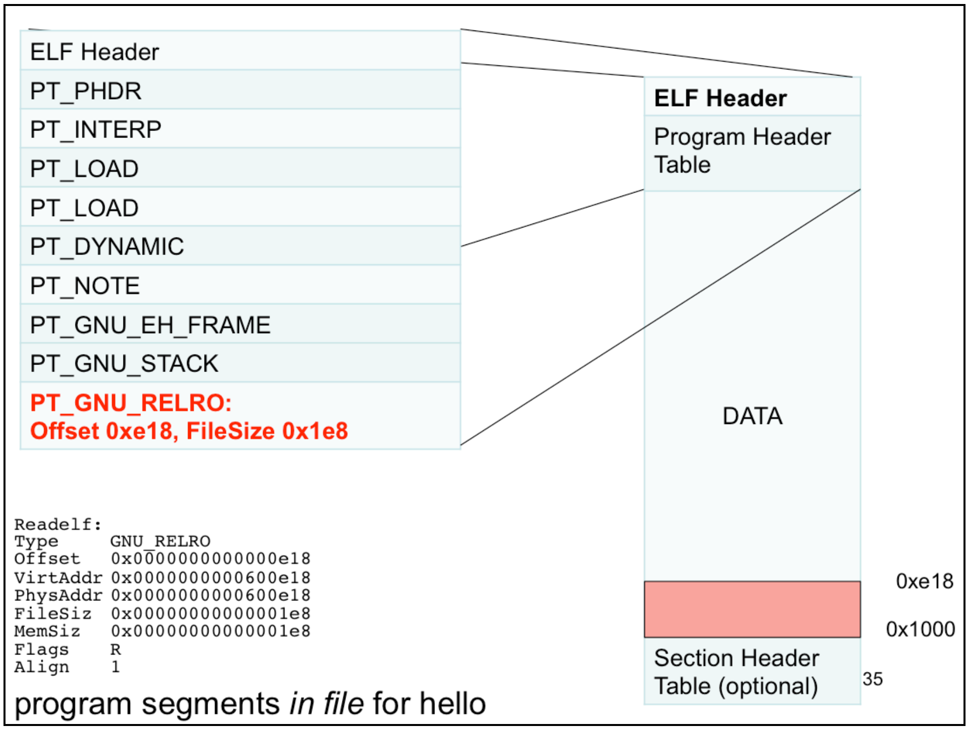
ELF Header
PT_PHDR
PT_INTERP
PT_LOAD
PT_LOAD
PT_DYNAMIC
PT_NOTE
PT_GNU_EH_FRAME
PT_GNU_STACK: Special Purpose
Offset 0, FileSize 0
PT_GNU_RELRO

ELF Header
Program Header Table
DATA
Section Header Table (optional)

```
Readelf:
Type     GNU_STACK
Offset   0x0000000000000000
VirtAddr 0x0000000000000000
PhysAddr 0x0000000000000000
FileSiz  0x0000000000000000
MemSiz   0x0000000000000000
Flags    RW
Align    8
```

program segments *in file* for hello

34

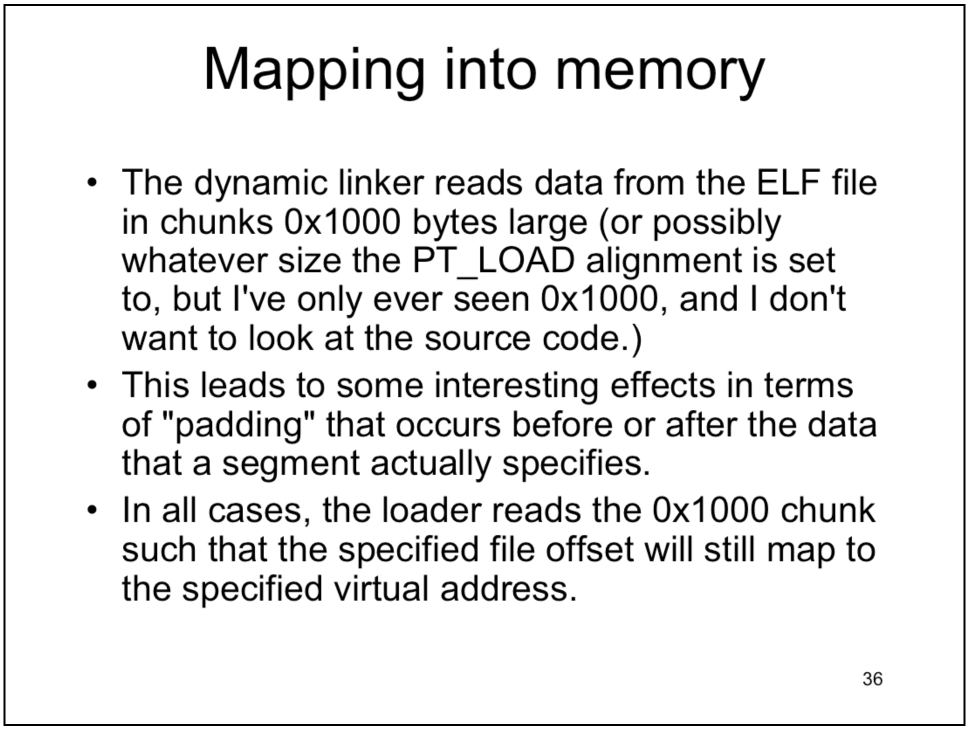


ELF Header
PT_PHDR
PT_INTERP
PT_LOAD
PT_LOAD
PT_DYNAMIC
PT_NOTE
PT_GNU_EH_FRAME
PT_GNU_STACK
PT_GNU_RELRO:
Offset 0xe18, FileSize 0x1e8

ELF Header
Program Header Table
DATA
Section Header Table (optional)

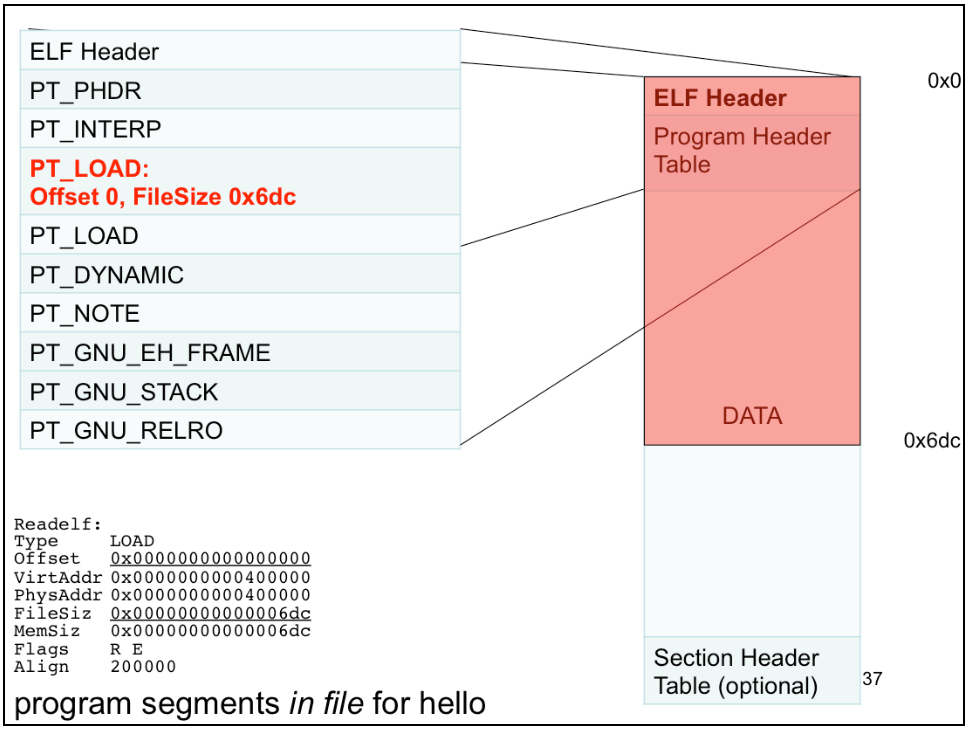
```
Readelf:
Type     GNU_RELRO
Offset   0x0000000000000e18
VirtAddr 0x0000000000600e18
PhysAddr 0x0000000000600e18
FileSiz  0x00000000000001e8
MemSiz   0x00000000000001e8
Flags    R
Align    1
```

program segments *in file* for hello



Mapping into memory

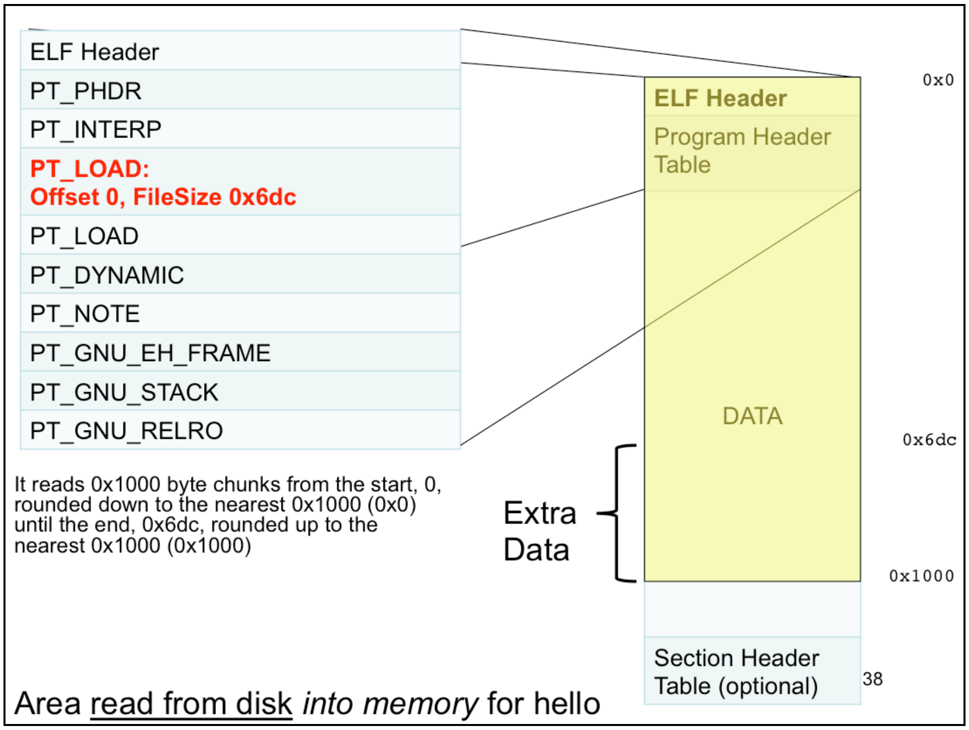
- The dynamic linker reads data from the ELF file in chunks 0x1000 bytes large (or possibly whatever size the PT_LOAD alignment is set to, but I've only ever seen 0x1000, and I don't want to look at the source code.)
- This leads to some interesting effects in terms of "padding" that occurs before or after the data that a segment actually specifies.
- In all cases, the loader reads the 0x1000 chunk such that the specified file offset will still map to the specified virtual address.



ELF Header
PT_PHDR
PT_INTERP
PT_LOAD:
Offset 0, FileSize 0x6dc
PT_LOAD
PT_DYNAMIC
PT_NOTE
PT_GNU_EH_FRAME
PT_GNU_STACK
PT_GNU_RELRO

```
Readelf:
Type     LOAD
Offset   0x0000000000000000
VirtAddr 0x0000000000400000
PhysAddr 0x0000000000400000
FileSiz  0x00000000000006dc
MemSiz   0x00000000000006dc
Flags    R E
Align    200000
```

program segments *in file* for hello



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD:
Offset 0, FileSize 0x6dc

PT_LOAD

PT_DYNAMIC

PT_NOTE

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

ELF Header

Program Header Table

DATA

Extra Data

Section Header Table (optional)

0x0

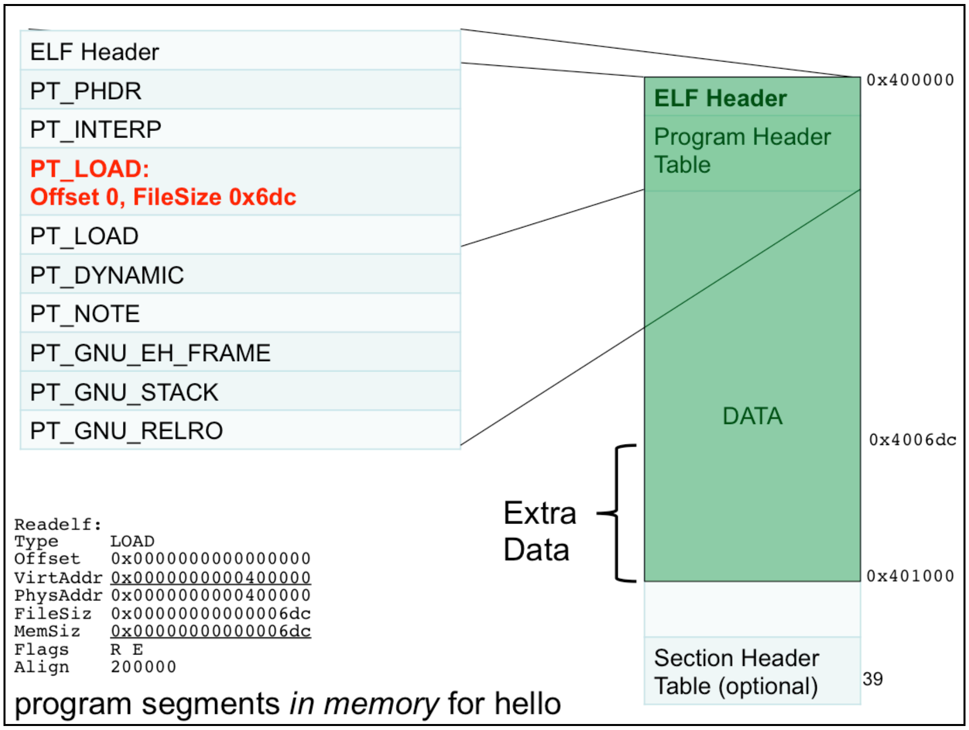
0x6dc

0x1000

It reads 0x1000 byte chunks from the start, 0, rounded down to the nearest 0x1000 (0x0) until the end, 0x6dc, rounded up to the nearest 0x1000 (0x1000)

Area <u>read from disk</u> *into memory* for hello

38



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD:
Offset 0, FileSize 0x6dc

PT_LOAD

PT_DYNAMIC

PT_NOTE

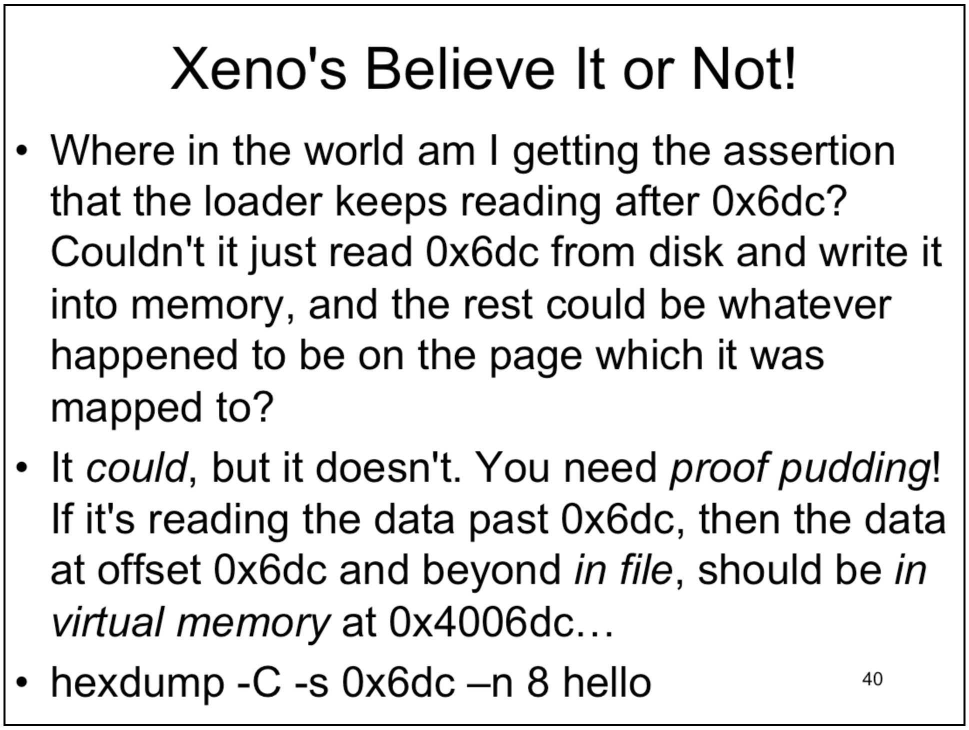
PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

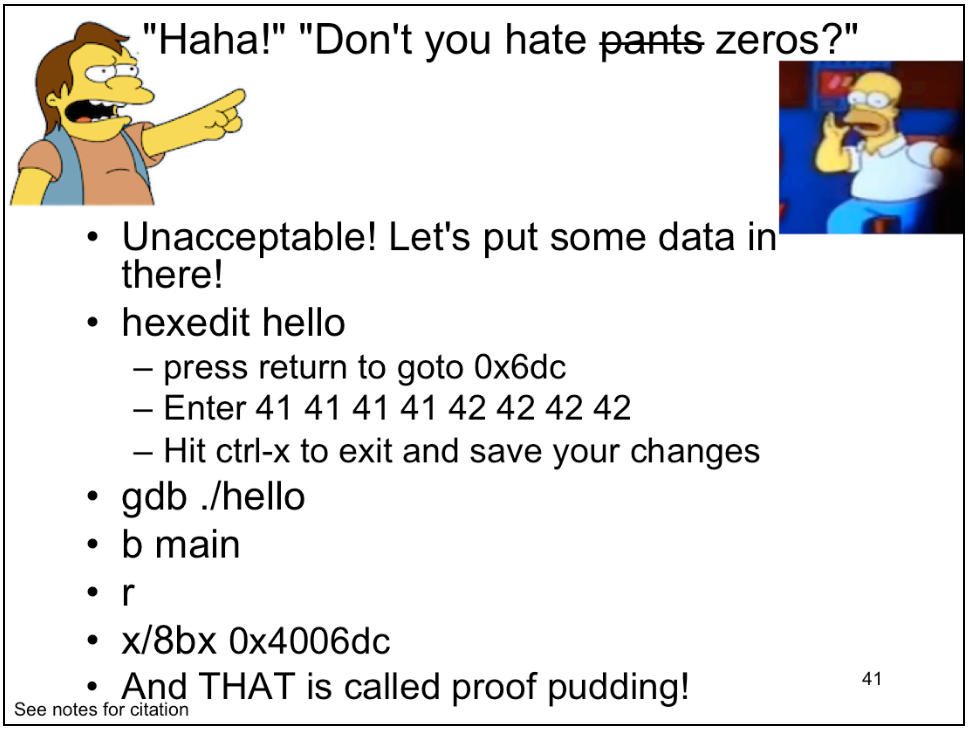
```
Readelf:
Type     LOAD
Offset   0x0000000000000000
VirtAddr 0x0000000000400000
PhysAddr 0x0000000000400000
FileSiz  0x00000000000006dc
MemSiz   0x00000000000006dc
Flags    R E
Align    200000
```

program segments *in memory* for hello



Xeno's Believe It or Not!

- Where in the world am I getting the assertion that the loader keeps reading after 0x6dc? Couldn't it just read 0x6dc from disk and write it into memory, and the rest could be whatever happened to be on the page which it was mapped to?
- It *could*, but it doesn't. You need *proof pudding*! If it's reading the data past 0x6dc, then the data at offset 0x6dc and beyond *in file*, should be *in virtual memory* at 0x4006dc…
- hexdump -C -s 0x6dc –n 8 hello



"Haha!" "Don't you hate ~~pants~~ zeros?"

- Unacceptable! Let's put some data in there!
- hexedit hello
  - press return to goto 0x6dc
  - Enter 41 41 41 41 42 42 42 42
  - Hit ctrl-x to exit and save your changes
- gdb ./hello
- b main
- r
- x/8bx 0x4006dc
- And THAT is called proof pudding!

See notes for citation

41

http://community.secondlife.com/t5/image/serverpage/image-id/21161i18A255DCE485562B/image-size/original?v=mpbl-1

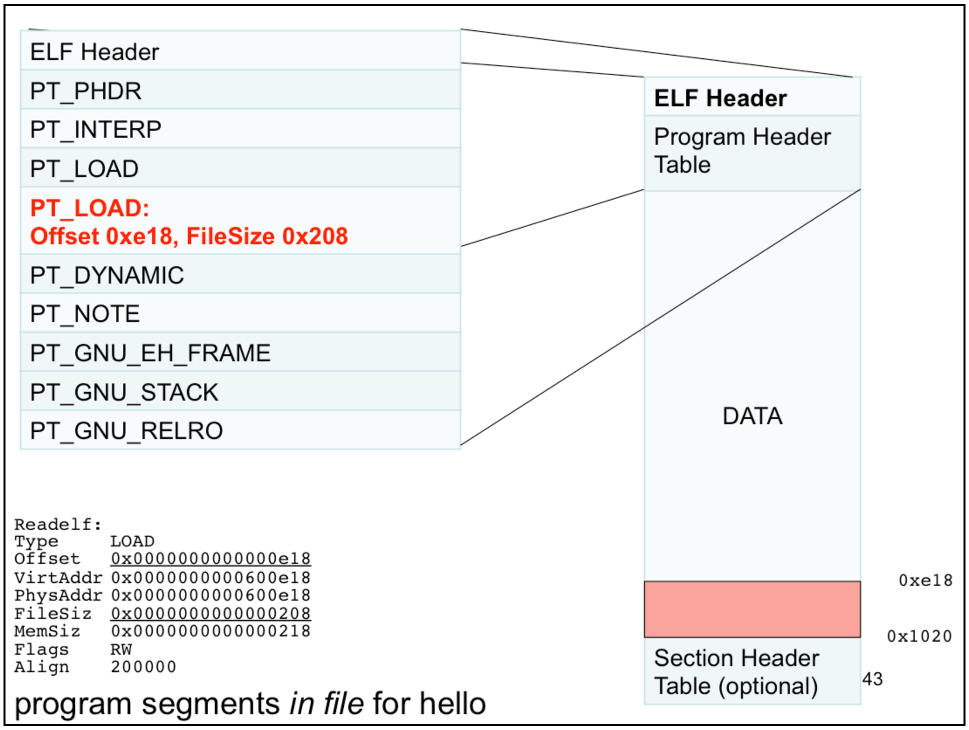
http://www.youtube.com/watch?v=1srzsTJOW2c

42

See notes for citation

One of my best slides to date
"Disco Stu doesn't fraternize…with zeros."

http://4.bp.blogspot.com/-qlLwD3ouTPQ/TpxnX-SXIlI/AAAAAAAAGCQ/0Ieb0RFeJK8/s1600/homerhatespants.png



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD

PT_LOAD:
Offset 0xe18, FileSize 0x208

PT_DYNAMIC

PT_NOTE

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

ELF Header

Program Header Table

DATA

0xe18

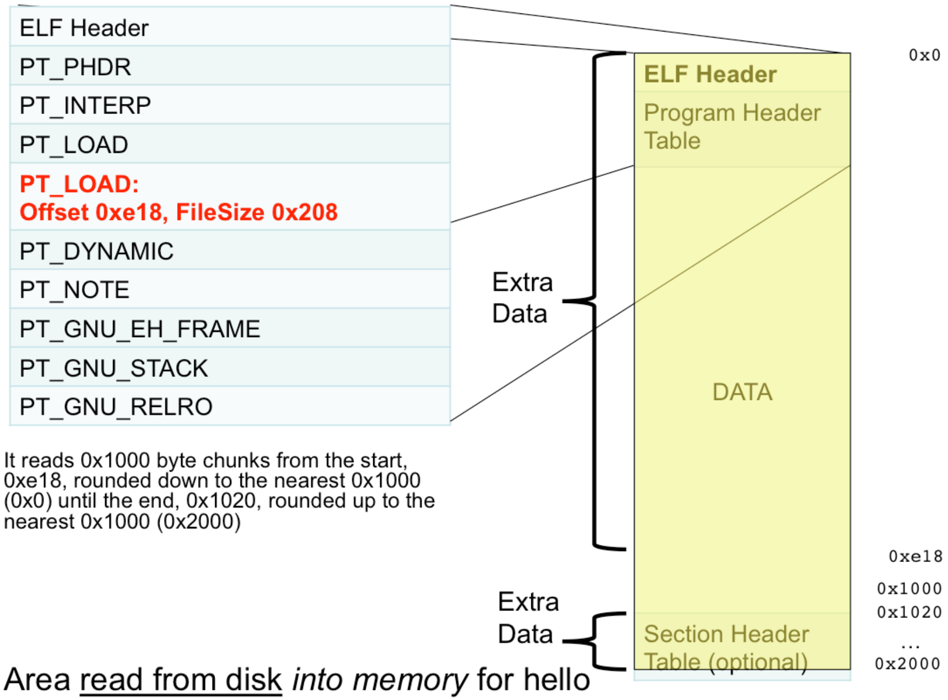
0x1020

Section Header Table (optional)

```
Readelf:
Type      LOAD
Offset    0x0000000000000e18
VirtAddr 0x0000000000600e18
PhysAddr 0x0000000000600e18
FileSiz  0x0000000000000208
MemSiz   0x0000000000000218
Flags    RW
Align    200000
```

program segments *in file* for hello

43



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD

PT_LOAD:
Offset 0xe18, FileSize 0x208

PT_DYNAMIC

PT_NOTE

PT_GNU_EH_FRAME

PT_GNU_STACK

PT_GNU_RELRO

Extra Data

ELF Header

Program Header Table

DATA

Section Header Table (optional)

Extra Data

0x0

0xe18

0x1000

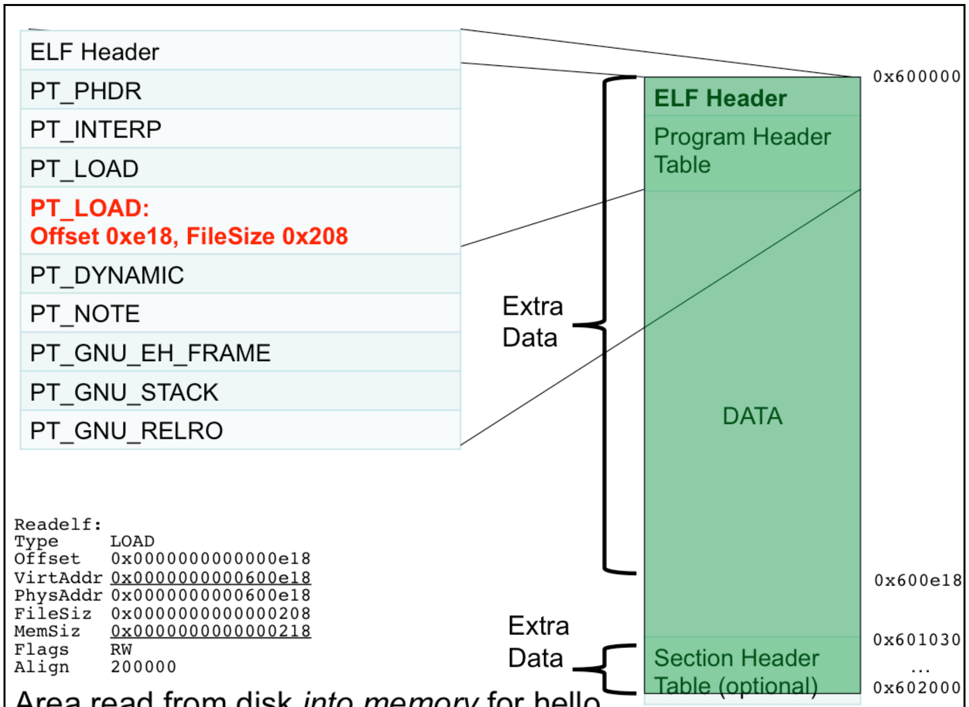
0x1020

...

0x2000

It reads 0x1000 byte chunks from the start, 0xe18, rounded down to the nearest 0x1000 (0x0) until the end, 0x1020, rounded up to the nearest 0x1000 (0x2000)

Area read from disk *into memory* for hello



```
Readelf:
Type     LOAD
Offset   0x0000000000000e18
VirtAddr 0x0000000000600e18
PhysAddr 0x0000000000600e18
FileSiz  0x0000000000000208
MemSiz   0x0000000000000218
Flags    RW
Align    200000
```

Area read from disk *into memory* for hello

ELF Header

PT_PHDR

PT_INTERP

PT_LOAD:
VirtAddr 0x400000, MemSize 0x6dc

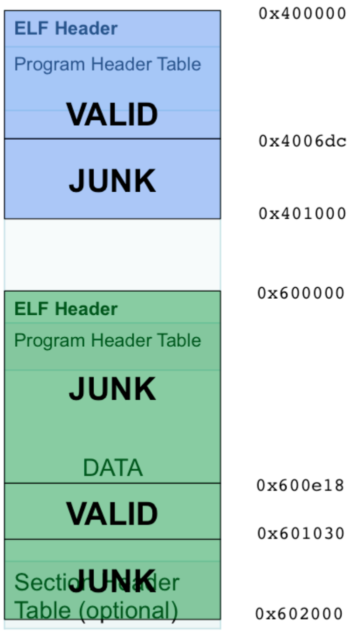
PT_LOAD:
VirtAddr 0x600e18, MemSize 0x210

PT_DYNAMIC

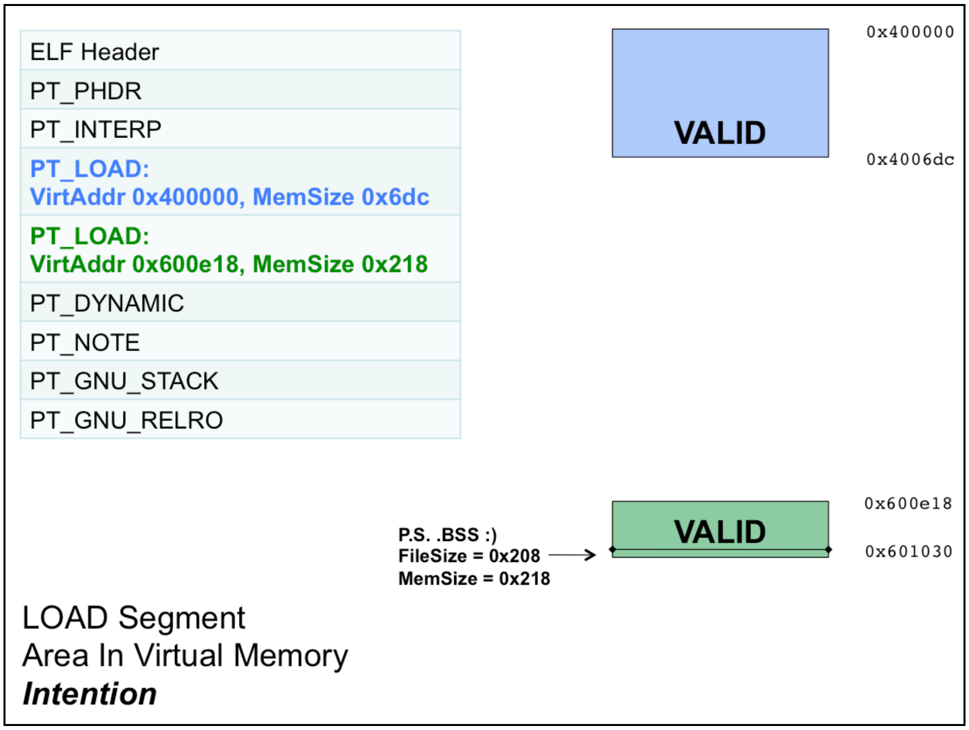
PT_NOTE

PT_GNU_STACK

PT_GNU_RELRO



LOAD Segment
Area In Virtual Memory for hello
NOTE: I CHANGED SCALE!



ELF Header

PT_PHDR

PT_INTERP

PT_LOAD:
VirtAddr 0x400000, MemSize 0x6dc

PT_LOAD:
VirtAddr 0x600e18, MemSize 0x218

PT_DYNAMIC

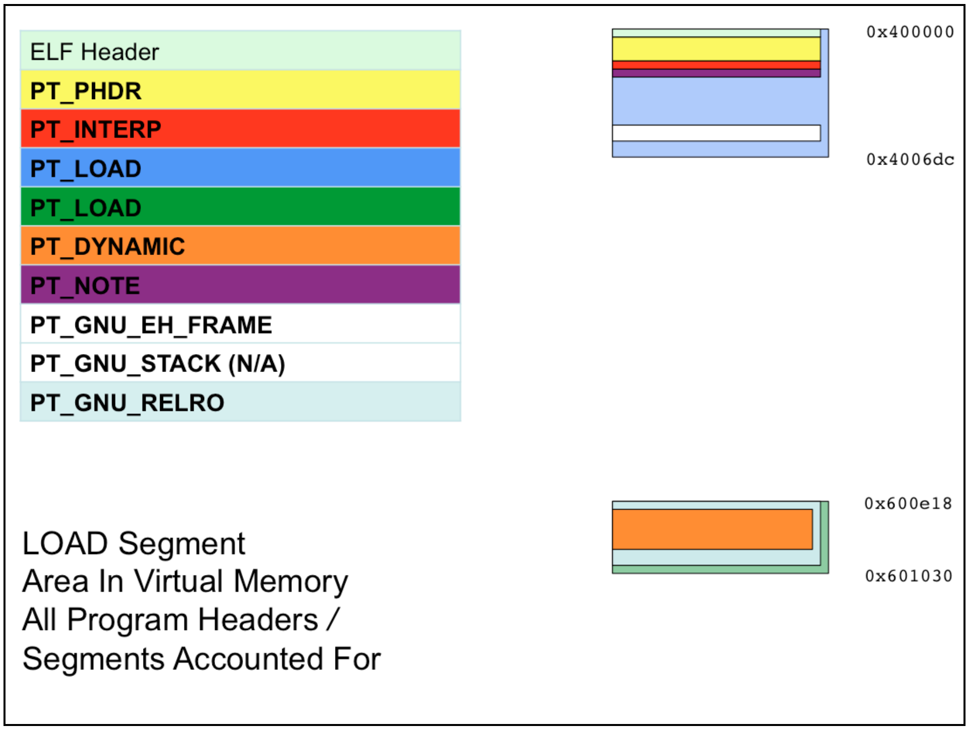
PT_NOTE

PT_GNU_STACK

PT_GNU_RELRO

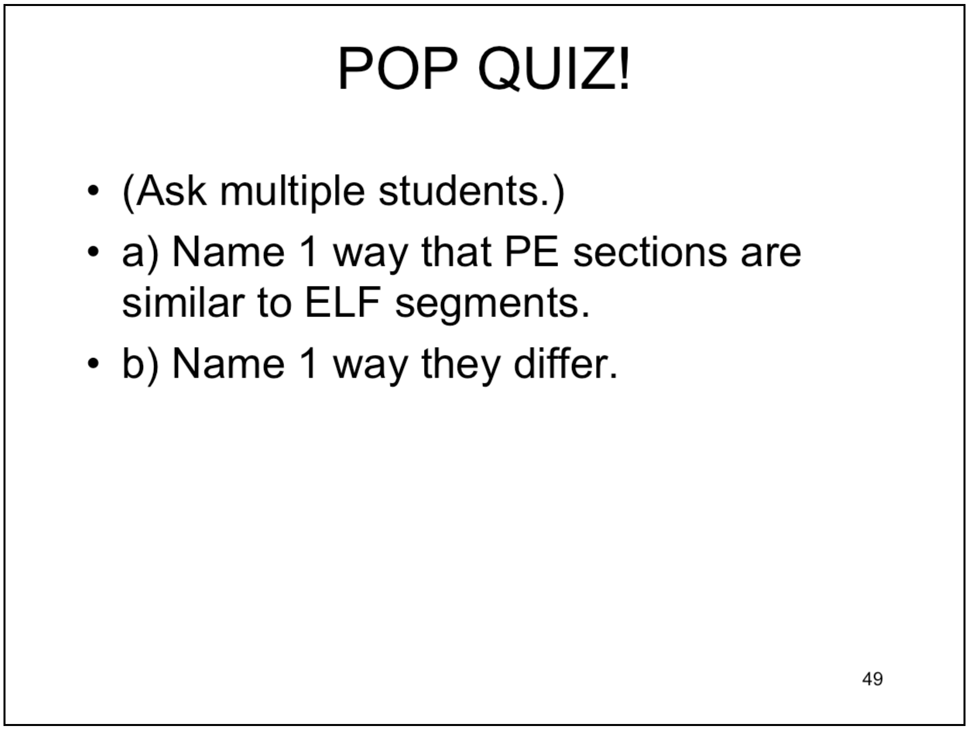
P.S. .BSS :)
FileSize = 0x208
MemSize = 0x218

LOAD Segment
Area In Virtual Memory
Intention



ELF Header
PT_PHDR
PT_INTERP
PT_LOAD
PT_LOAD
PT_DYNAMIC
PT_NOTE
PT_GNU_EH_FRAME
PT_GNU_STACK (N/A)
PT_GNU_RELRO

LOAD Segment
Area In Virtual Memory
All Program Headers /
Segments Accounted For



POP QUIZ!

- (Ask multiple students.)
- a) Name 1 way that PE sections are similar to ELF segments.
- b) Name 1 way they differ.

Possible correct answers for a)

They both specify read/write/execute permissions on memory.

They both specify a region of memory which was mapped in from disk.

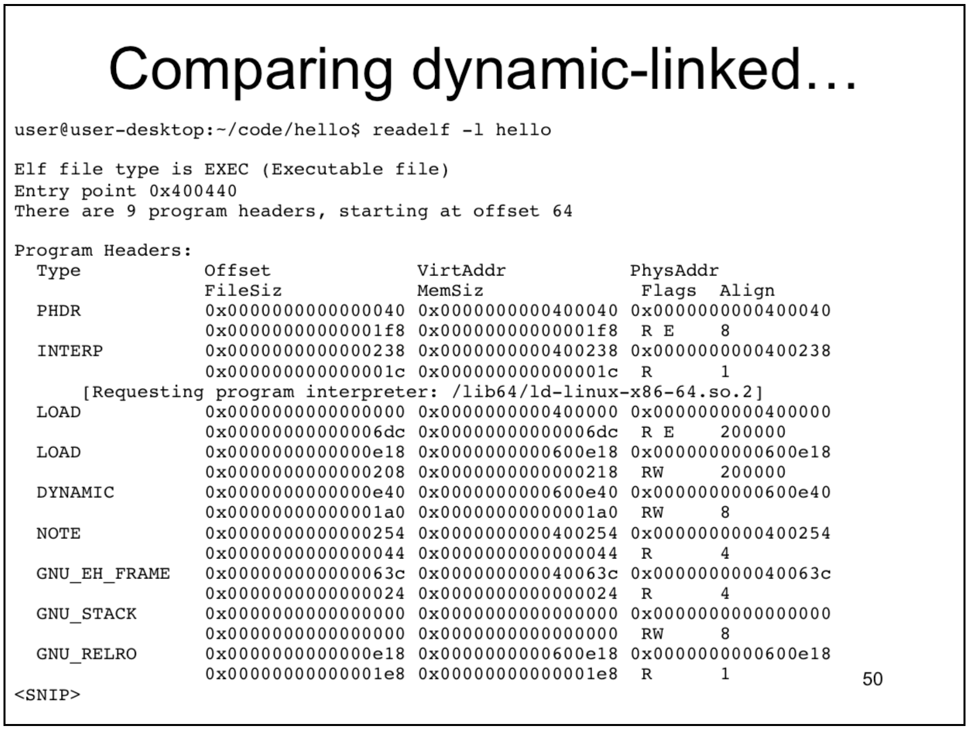
A typical PE section is multiple linker sections merged into

Possible correct answers for b)

Elf can never have a file size smaller than the memory size / PE can have either of file or memory size be larger than the other.

An ELF segment encompases

An ELF segment doesn't have a name, only an ELF section has a name (



Comparing dynamic-linked…

```
user@user-desktop:~/code/hello$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x400440
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006dc 0x00000000000006dc  R E    200000
  LOAD           0x0000000000000e18 0x0000000000600e18 0x0000000000600e18
                 0x0000000000000208 0x0000000000000218  RW     200000
  DYNAMIC        0x0000000000000e40 0x0000000000600e40 0x0000000000600e40
                 0x00000000000001a0 0x00000000000001a0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x000000000000063c 0x000000000040063c 0x000000000040063c
                 0x0000000000000024 0x0000000000000024  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8
  GNU_RELRO      0x0000000000000e18 0x0000000000600e18 0x0000000000600e18
                 0x00000000000001e8 0x00000000000001e8  R      1
<SNIP>
```

50

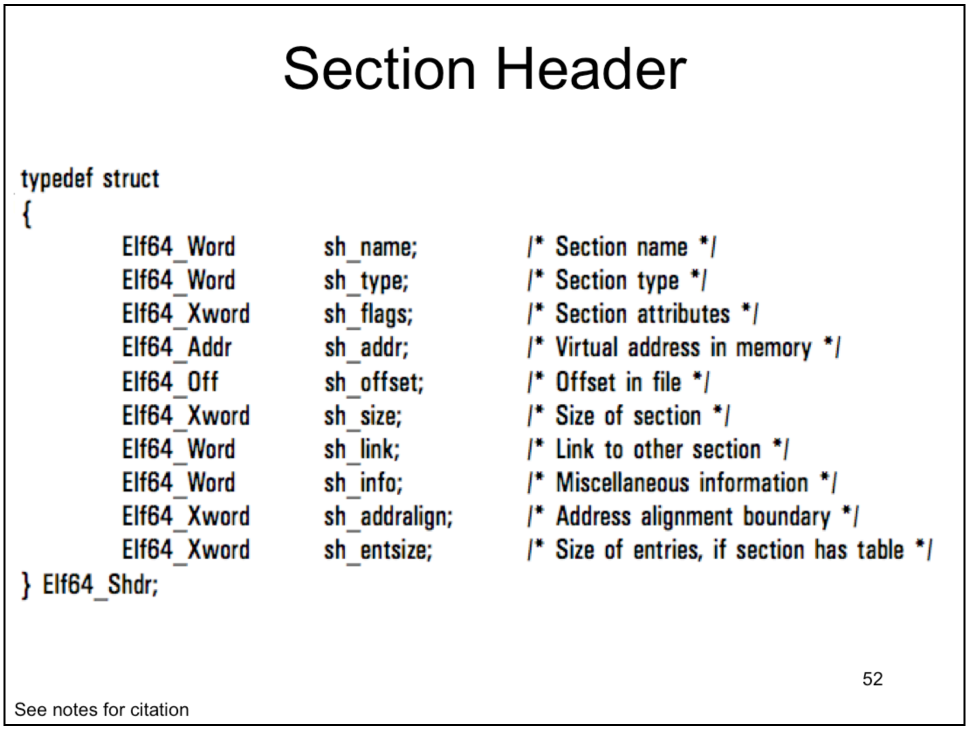
To static linked…

```
user@user-desktop:~/code/hello$ gcc -static -o hello-static hello.c
user@user-desktop:~/code/hello$ readelf -l hello-static

Elf file type is EXEC (Executable file)
Entry point 0x400320
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000a0bd9 0x00000000000a0bd9  R E    200000
  LOAD           0x00000000000a0f00 0x00000000006a0f00 0x00000000006a0f00
                 0x0000000000000db0 0x0000000000003ea8  RW     200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000a0f00 0x00000000006a0f00 0x00000000006a0f00
                 0x0000000000000020 0x0000000000000050  R      8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8
  GNU_RELRO      0x00000000000a0f00 0x00000000006a0f00 0x00000000006a0f00
                 0x0000000000000100 0x0000000000000100  R      1
<SNIP>
```

```
Note: no segments of type PHDR, INTERP, DYNAMIC, GNU_EH_FRAME. Added type TLS
```



Section Header

```
typedef struct
{
        Elf64_Word      sh_name;        /* Section name */
        Elf64_Word      sh_type;        /* Section type */
        Elf64_Xword     sh_flags;       /* Section attributes */
        Elf64_Addr      sh_addr;        /* Virtual address in memory */
        Elf64_Off       sh_offset;      /* Offset in file */
        Elf64_Xword     sh_size;        /* Size of section */
        Elf64_Word      sh_link;        /* Link to other section */
        Elf64_Word      sh_info;        /* Miscellaneous information */
        Elf64_Xword     sh_addralign;   /* Address alignment boundary */
        Elf64_Xword     sh_entsize;     /* Size of entries, if section has table */
} Elf64_Shdr;
```

52

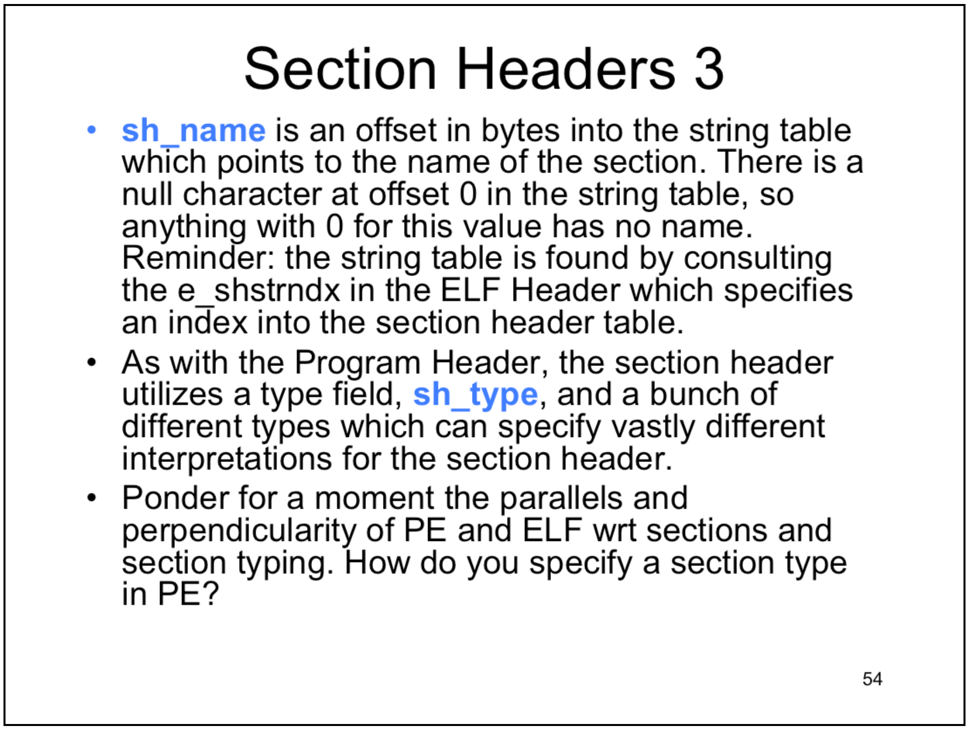
See notes for citation

http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf

Section Headers 2

from /usr/include/elf.h

```
typedef struct
{
  Elf64_Word    sh_name;               /* Section name (string tbl index) */
  Elf64_Word    sh_type;               /* Section type */
  Elf64_Xword   sh_flags;              /* Section flags */
  Elf64_Addr    sh_addr;               /* Section virtual addr at execution */
  Elf64_Off     sh_offset;             /* Section file offset */
  Elf64_Xword   sh_size;               /* Section size in bytes */
  Elf64_Word    sh_link;               /* Link to another section */
  Elf64_Word    sh_info;               /* Additional section information */
  Elf64_Xword   sh_addralign;          /* Section alignment */
  Elf64_Xword   sh_entsize;            /* Entry size if section holds table */
} Elf64_Shdr;
```



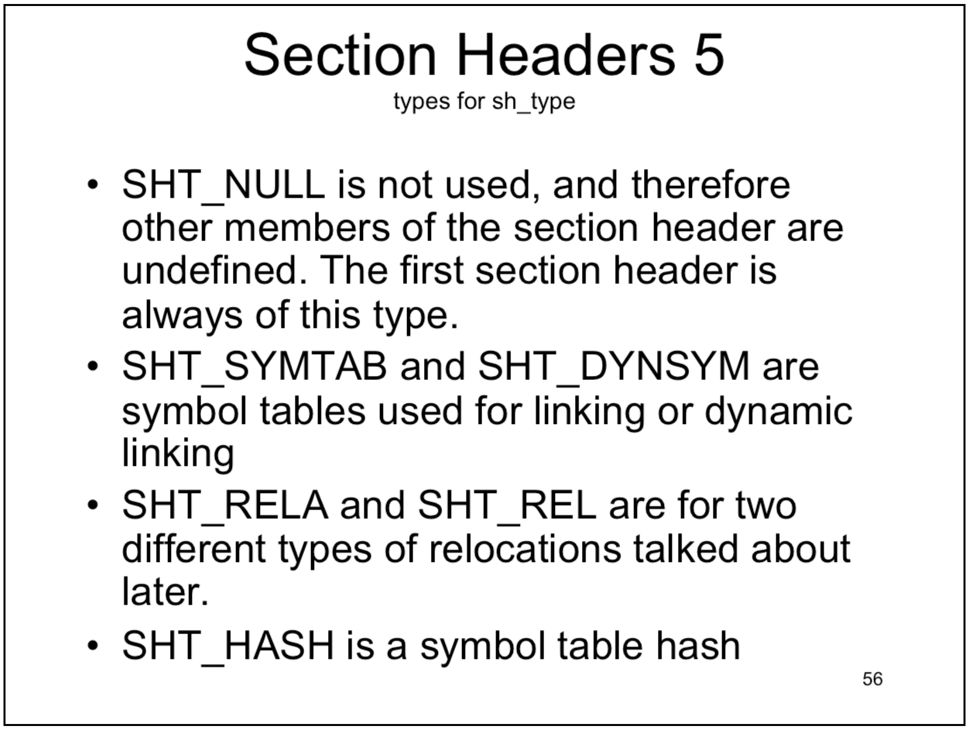
Section Headers 3

- **sh_name** is an offset in bytes into the string table which points to the name of the section. There is a null character at offset 0 in the string table, so anything with 0 for this value has no name. Reminder: the string table is found by consulting the e_shstrndx in the ELF Header which specifies an index into the section header table.
- As with the Program Header, the section header utilizes a type field, **sh_type**, and a bunch of different types which can specify vastly different interpretations for the section header.
- Ponder for a moment the parallels and perpendicularity of PE and ELF wrt sections and section typing. How do you specify a section type in PE?

Section Headers 4

types for sh_type

- **SHT_PROGBITS** is a sort of the catch-all for anything which is valid but doesn't have some other special predefined type
- **SHT_STRTAB** is for string tables
- **SHT_DYNAMIC** is for dynamic linking information
- **SHT_NOBITS** is for things which take no space in the file but do take space in memory (like .bss)



Section Headers 5

types for sh_type

- SHT_NULL is not used, and therefore other members of the section header are undefined. The first section header is always of this type.
- SHT_SYMTAB and SHT_DYNSYM are symbol tables used for linking or dynamic linking
- SHT_RELA and SHT_REL are for two different types of relocations talked about later.
- SHT_HASH is a symbol table hash

Section Headers 6

- **sh_flags** can have the values SHF_WRITE = 0x1, SHF_ALLOC = 0x2, SHF_EXECINSTR = 0x4, or any number of processor-specific things as long as the MSB is set (which can be checked by ANDing with SHF_MASKPROC). Of these, SHF_ALLOC probably needs a little more explaining. SHF_ALLOC declares whether or not this section is going to occupy memory during program execution. The .debug* or .shstrtab are examples of sections that don't set this bit (and also set the sh_addr to 0 as was previously described as a way to indicate it won't reside in memory.)
- **sh_addr** is the virtual address where this section starts in memory. It's set to 0 if the section won't reside in memory.
- **sh_offset** is the file offset to the start of this data. This is still set for things with type SHT_NOBITS as it is the "conceptual" offset ;)

Section Headers 7

- **sh_size** is the size of the section in bytes

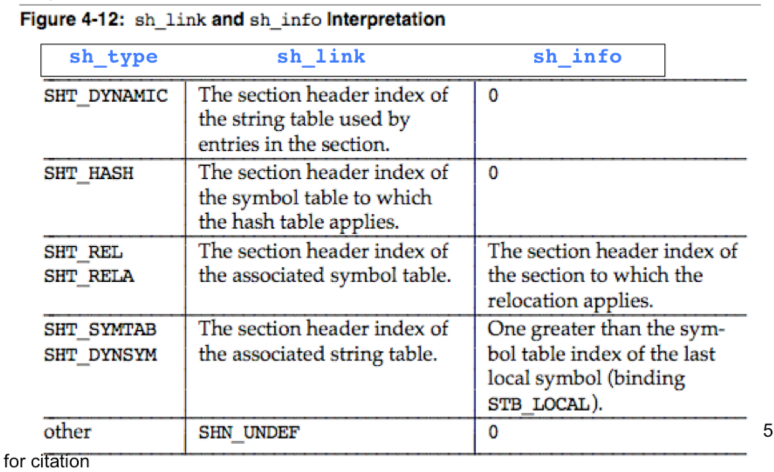
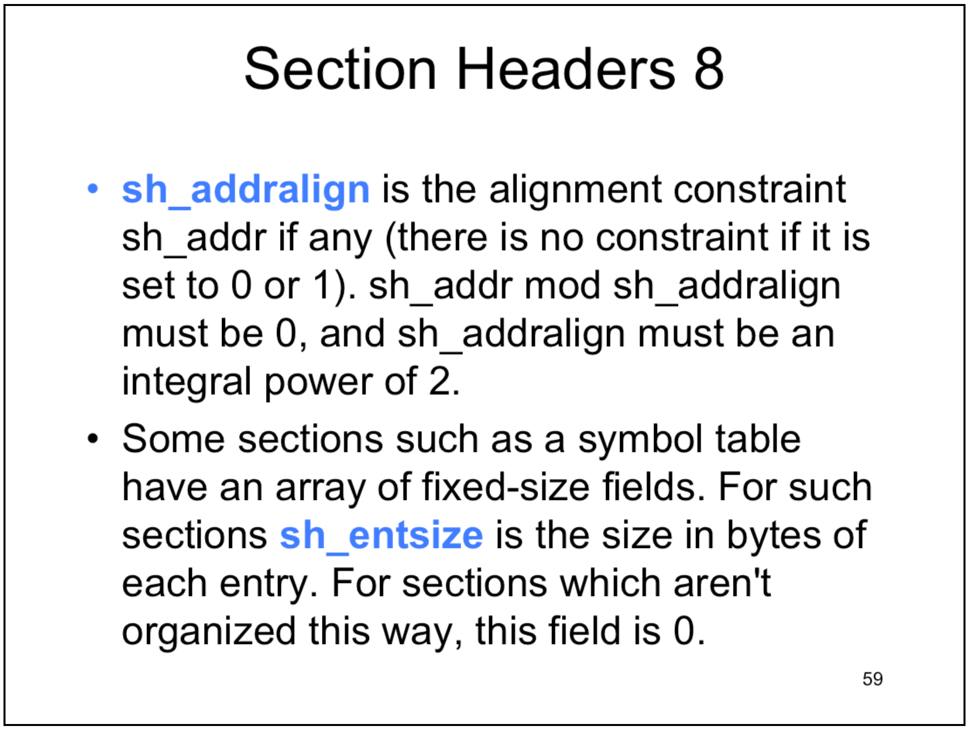


Figure 4-12: sh_link **and** sh_info **Interpretation**

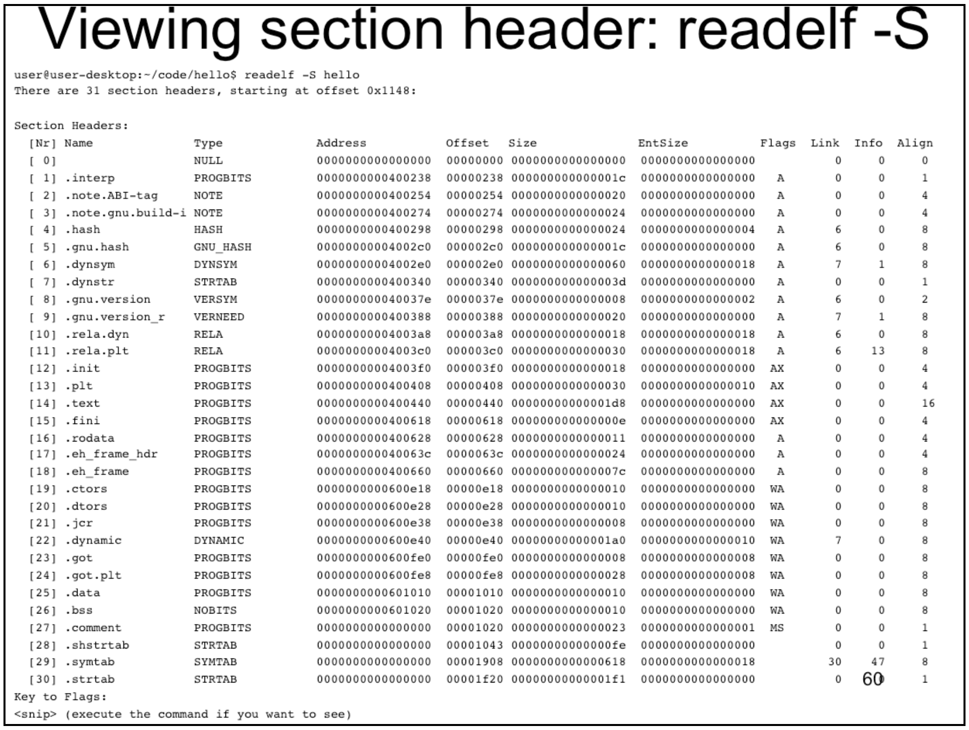
sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the string table used by entries in the section.	0
SHT_HASH	The section header index of the symbol table to which the hash table applies.	0
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
other	SHN_UNDEF	0

See notes for citation



Section Headers 8

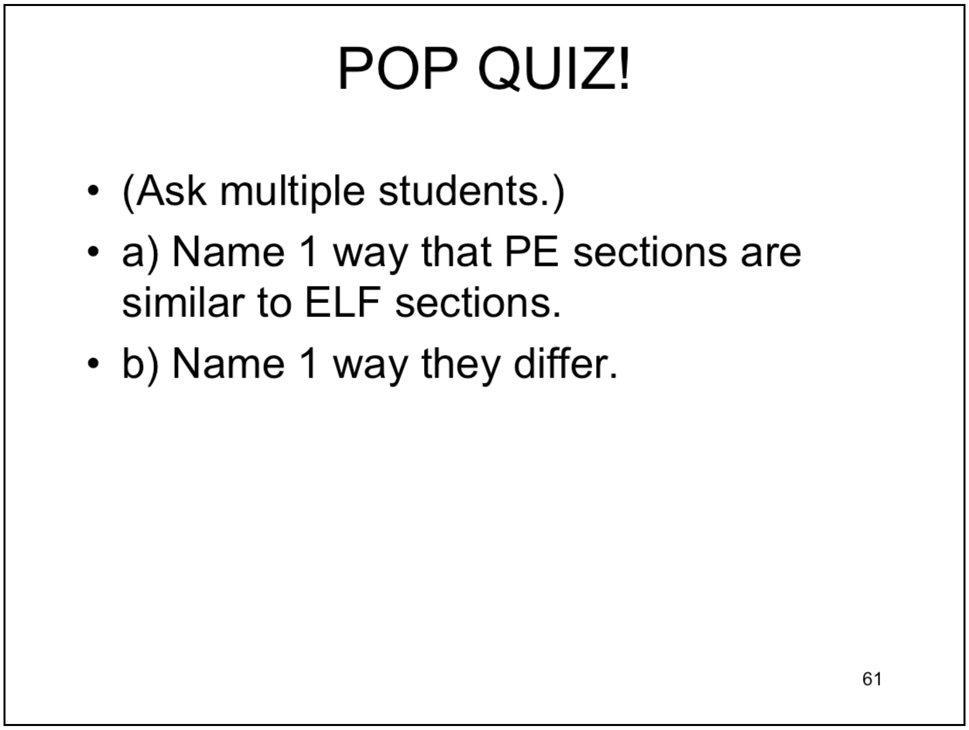
- **sh_addralign** is the alignment constraint sh_addr if any (there is no constraint if it is set to 0 or 1). sh_addr mod sh_addralign must be 0, and sh_addralign must be an integral power of 2.
- Some sections such as a symbol table have an array of fixed-size fields. For such sections **sh_entsize** is the size in bytes of each entry. For sections which aren't organized this way, this field is 0.



Viewing section header: readelf -S

```
user@user-desktop:~/code/hello$ readelf -S hello
There are 31 section headers, starting at offset 0x1148:

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         0000000000400238  00000238  000000000000001c  0000000000000000    A       0     0      1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254  0000000000000020  0000000000000000    A       0     0      4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274  0000000000000024  0000000000000000    A       0     0      4
  [ 4] .hash             HASH             0000000000400298  00000298  0000000000000024  0000000000000004    A       6     0      8
  [ 5] .gnu.hash         GNU_HASH         00000000004002c0  000002c0  000000000000001c  0000000000000000    A       6     0      8
  [ 6] .dynsym           DYNSYM           00000000004002e0  000002e0  0000000000000060  0000000000000018    A       7     1      8
  [ 7] .dynstr           STRTAB           0000000000400340  00000340  000000000000003d  0000000000000000    A       0     0      1
  [ 8] .gnu.version      VERSYM           000000000040037e  0000037e  0000000000000008  0000000000000002    A       6     0      2
  [ 9] .gnu.version_r    VERNEED          0000000000400388  00000388  0000000000000020  0000000000000000    A       7     1      8
  [10] .rela.dyn         RELA             00000000004003a8  000003a8  0000000000000018  0000000000000018    A       6     0      8
  [11] .rela.plt         RELA             00000000004003c0  000003c0  0000000000000030  0000000000000018    A       6    13      8
  [12] .init             PROGBITS         00000000004003f0  000003f0  0000000000000018  0000000000000000   AX       0     0      4
  [13] .plt              PROGBITS         0000000000400408  00000408  0000000000000030  0000000000000010   AX       0     0      4
  [14] .text             PROGBITS         0000000000400440  00000440  00000000000001d8  0000000000000000   AX       0     0     16
  [15] .fini             PROGBITS         0000000000400618  00000618  000000000000000e  0000000000000000   AX       0     0      4
  [16] .rodata           PROGBITS         0000000000400628  00000628  0000000000000011  0000000000000000    A       0     0      4
  [17] .eh_frame_hdr     PROGBITS         000000000040063c  0000063c  0000000000000024  0000000000000000    A       0     0      4
  [18] .eh_frame         PROGBITS         0000000000400660  00000660  000000000000007c  0000000000000000    A       0     0      8
  [19] .ctors            PROGBITS         0000000000600e18  00000e18  0000000000000010  0000000000000000   WA       0     0      8
  [20] .dtors            PROGBITS         0000000000600e28  00000e28  0000000000000010  0000000000000000   WA       0     0      8
  [21] .jcr              PROGBITS         0000000000600e38  00000e38  0000000000000008  0000000000000000   WA       0     0      8
  [22] .dynamic          DYNAMIC          0000000000600e40  00000e40  00000000000001a0  0000000000000010   WA       7     0      8
  [23] .got              PROGBITS         0000000000600fe0  00000fe0  0000000000000008  0000000000000008   WA       0     0      8
  [24] .got.plt          PROGBITS         0000000000600fe8  00000fe8  0000000000000028  0000000000000008   WA       0     0      8
  [25] .data             PROGBITS         0000000000601010  00001010  0000000000000010  0000000000000000   WA       0     0      8
  [26] .bss              NOBITS           0000000000601020  00001020  0000000000000010  0000000000000000   WA       0     0      8
  [27] .comment          PROGBITS         0000000000000000  00001020  0000000000000023  0000000000000001   MS       0     0      1
  [28] .shstrtab         STRTAB           0000000000000000  00001043  00000000000000fe  0000000000000000            0     0      1
  [29] .symtab           SYMTAB           0000000000000000  00001908  0000000000000618  0000000000000018           30    47      8
  [30] .strtab           STRTAB           0000000000000000  00001f20  00000000000001f1  0000000000000000            0     0      1
Key to Flags:
<snip> (execute the command if you want to see)
```



POP QUIZ!

- (Ask multiple students.)
- a) Name 1 way that PE sections are similar to ELF sections.
- b) Name 1 way they differ.

Possible correct answers for a)

They both have an associated name

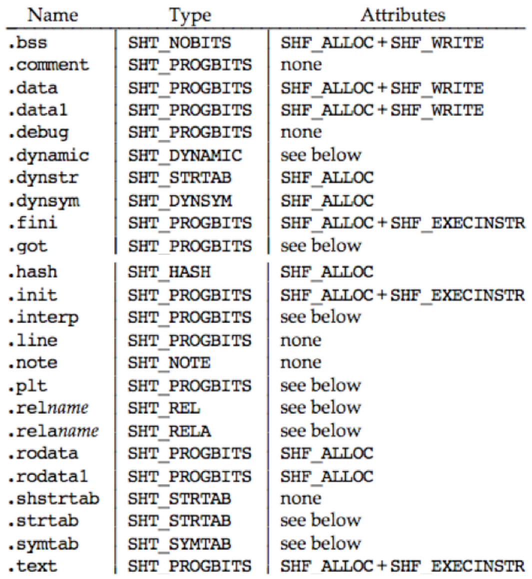
Possible correct answers for b)

Section names can be a max of 8 characters in PE, and they can be longer in ELF

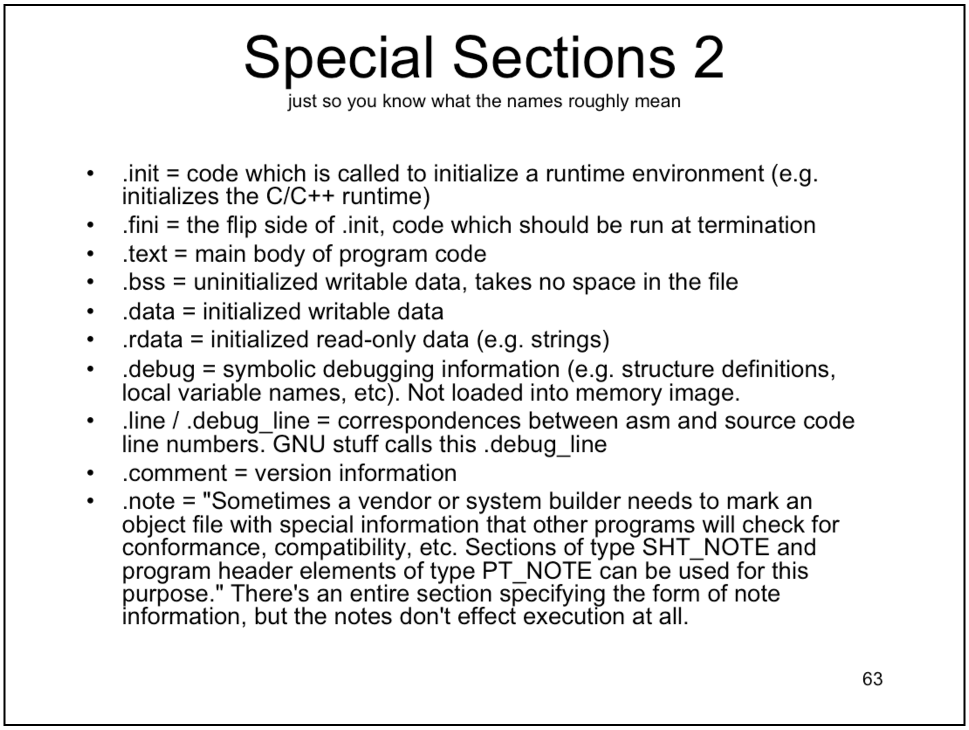
"Special Sections"

awwwww, idnt dat *special*

Figure 4-13: Special Sections



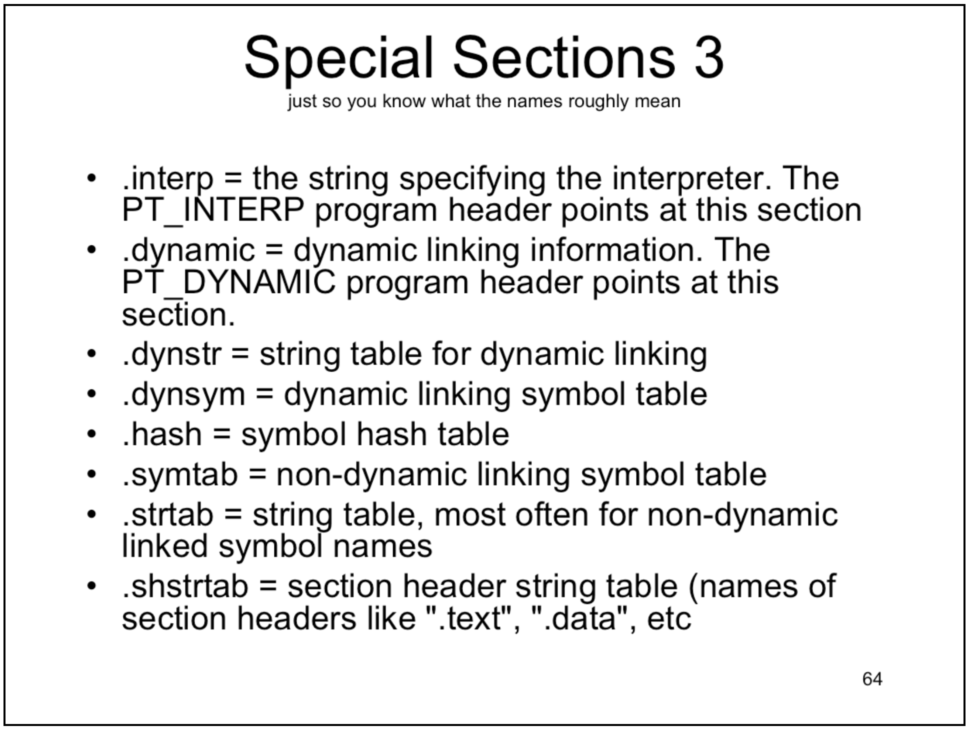
Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	see below
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	see below
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	see below
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.plt	SHT_PROGBITS	see below
.rel*name*	SHT_REL	see below
.rela*name*	SHT_RELA	see below
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.text	SHT PROGBITS	SHF ALLOC + SHF EXECINSTR



Special Sections 2

just so you know what the names roughly mean

- .init = code which is called to initialize a runtime environment (e.g. initializes the C/C++ runtime)
- .fini = the flip side of .init, code which should be run at termination
- .text = main body of program code
- .bss = uninitialized writable data, takes no space in the file
- .data = initialized writable data
- .rdata = initialized read-only data (e.g. strings)
- .debug = symbolic debugging information (e.g. structure definitions, local variable names, etc). Not loaded into memory image.
- .line / .debug_line = correspondences between asm and source code line numbers. GNU stuff calls this .debug_line
- .comment = version information
- .note = "Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, etc. Sections of type SHT_NOTE and program header elements of type PT_NOTE can be used for this purpose." There's an entire section specifying the form of note information, but the notes don't effect execution at all.



Special Sections 3

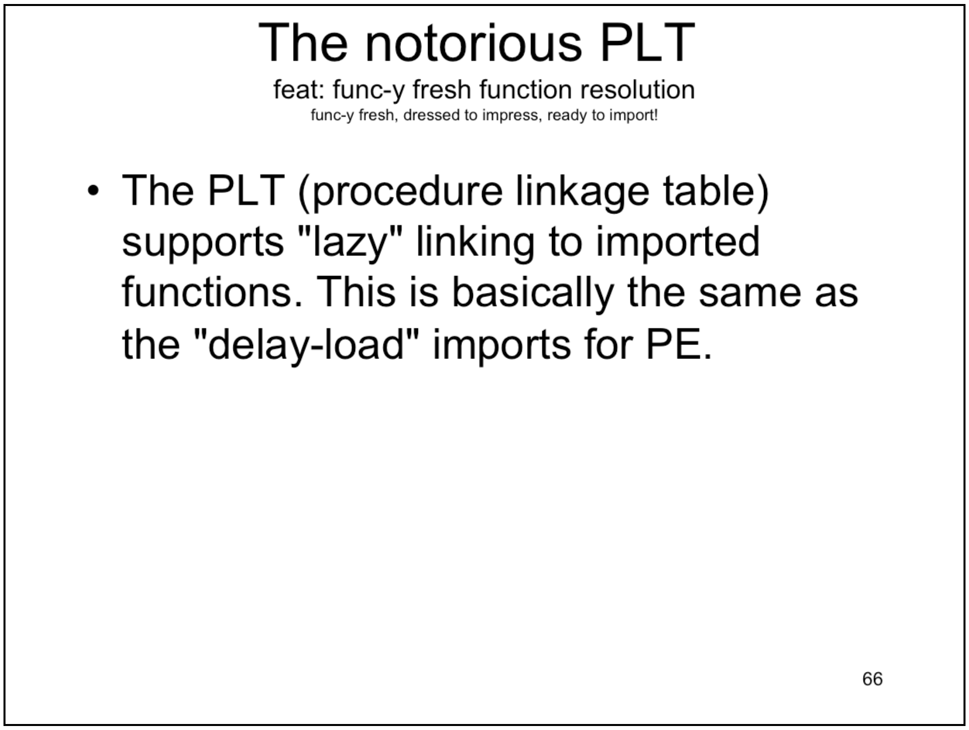
just so you know what the names roughly mean

- .interp = the string specifying the interpreter. The PT_INTERP program header points at this section
- .dynamic = dynamic linking information. The PT_DYNAMIC program header points at this section.
- .dynstr = string table for dynamic linking
- .dynsym = dynamic linking symbol table
- .hash = symbol hash table
- .symtab = non-dynamic linking symbol table
- .strtab = string table, most often for non-dynamic linked symbol names
- .shstrtab = section header string table (names of section headers like ".text", ".data", etc

Special Sections 4

just so you know what the names roughly mean

- .got = Global Offset Table used to help out position independent code
- .plt = Procedure Linkage Table used for "delay-load" aka "dynamic" aka "lazy" linking/resolution of imported functions.
- .got.plt = chunk of GOT used to support the PLT
- .rel* or .rela* = relocation information
- .ctors/.dtors = constructors/destructors not necessarily C++ con/destructors. You can use "__attribute__ ((constructor));" on functions to make them constructors, and then they will run before main(). You can apply "__attribute__ ((destructor));" and the function will run just after main() exits.
  - You may remember .ctors from such movies as "Corey's Exploits 1" and "Dial C for CTORS"!
- .jcr = java class registration
- .eh_frame = exception frame

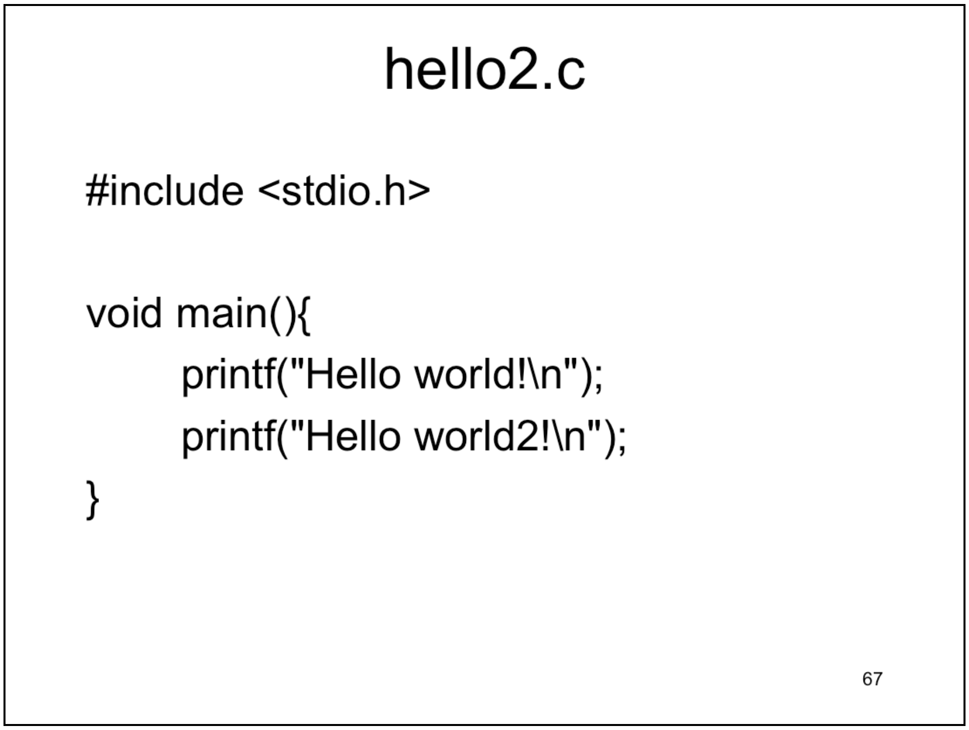


The notorious PLT

feat: func-y fresh function resolution

func-y fresh, dressed to impress, ready to import!

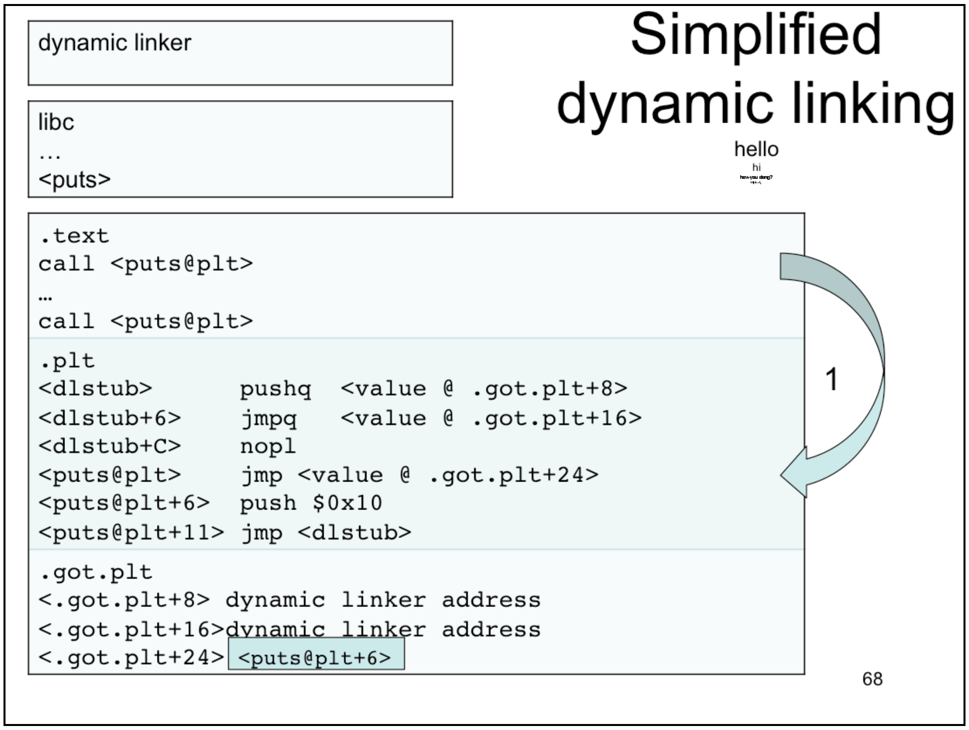
- The PLT (procedure linkage table) supports "lazy" linking to imported functions. This is basically the same as the "delay-load" imports for PE.



hello2.c

```
#include <stdio.h>

void main(){
	printf("Hello world!\n");
	printf("Hello world2!\n");
}
```



Simplified dynamic linking

hello

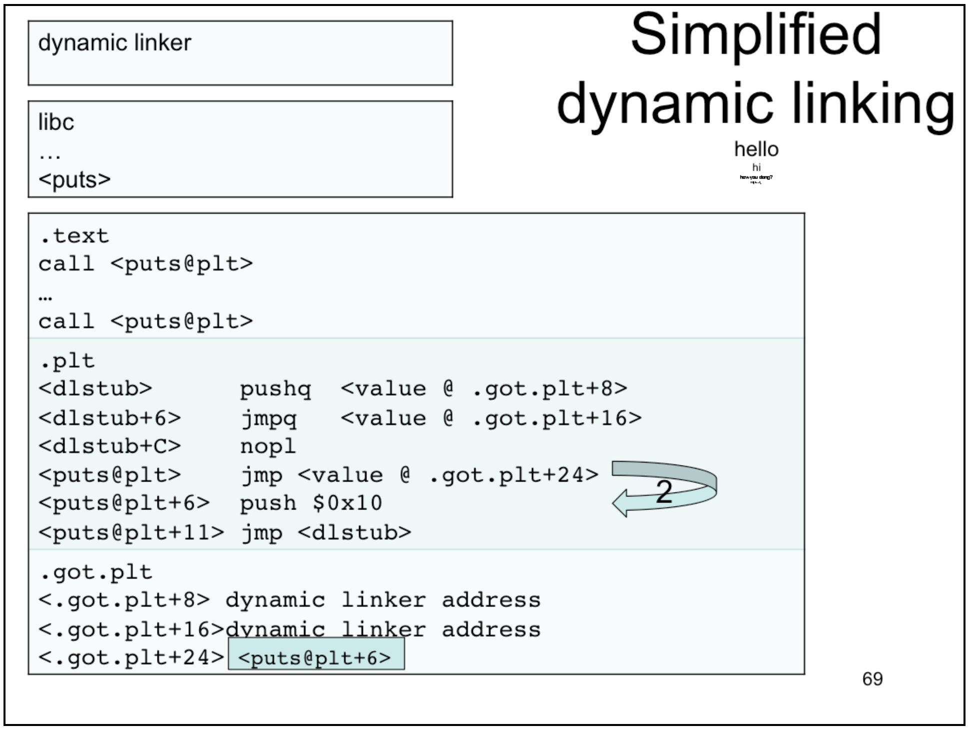
hi

how you doing?

```
dynamic linker
```

```
libc
…
<puts>
```

```
.text
call <puts@plt>
…
call <puts@plt>
.plt
<dlstub>      pushq  <value @ .got.plt+8>
<dlstub+6>    jmpq   <value @ .got.plt+16>
<dlstub+C>    nopl
<puts@plt>    jmp <value @ .got.plt+24>
<puts@plt+6>  push $0x10
<puts@plt+11> jmp <dlstub>
.got.plt
<.got.plt+8> dynamic linker address
<.got.plt+16>dynamic linker address
<.got.plt+24> <puts@plt+6>
```



Simplified dynamic linking

hello

hi

how you doing?

[illegible]

```
dynamic linker
```

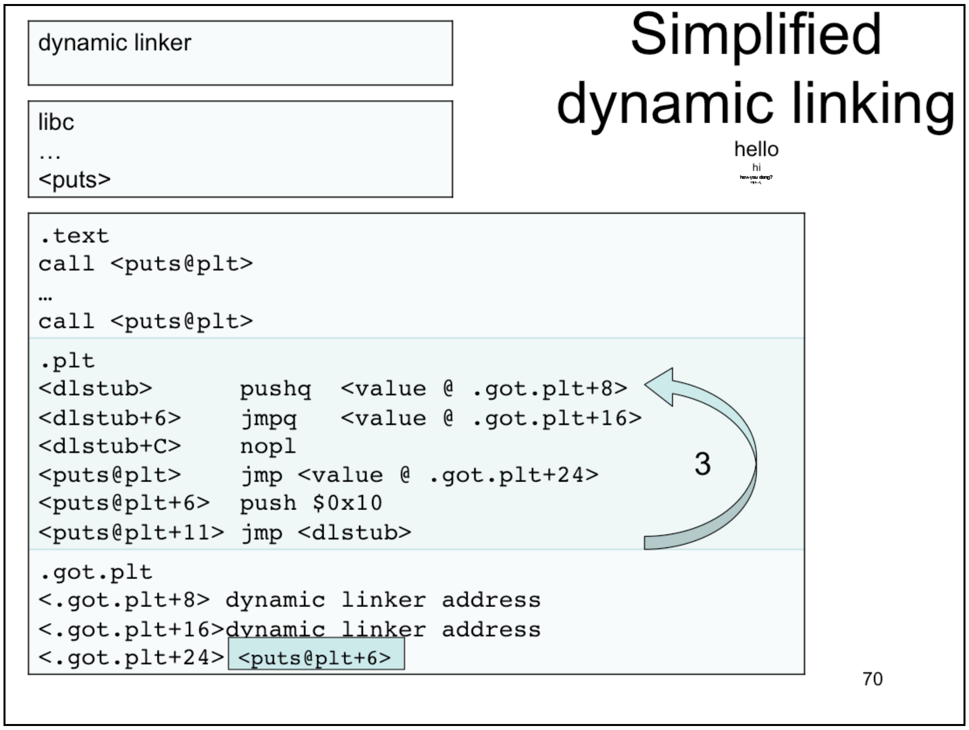
```
libc
…
<puts>
```

```
.text
call <puts@plt>
…
call <puts@plt>
```

```
.plt
<dlstub>       pushq  <value @ .got.plt+8>
<dlstub+6>     jmpq   <value @ .got.plt+16>
<dlstub+C>     nopl
<puts@plt>     jmp <value @ .got.plt+24>
<puts@plt+6>   push $0x10
<puts@plt+11>  jmp <dlstub>
```

```
.got.plt
<.got.plt+8> dynamic linker address
<.got.plt+16>dynamic linker address
<.got.plt+24> <puts@plt+6>
```

69



Simplified dynamic linking

hello

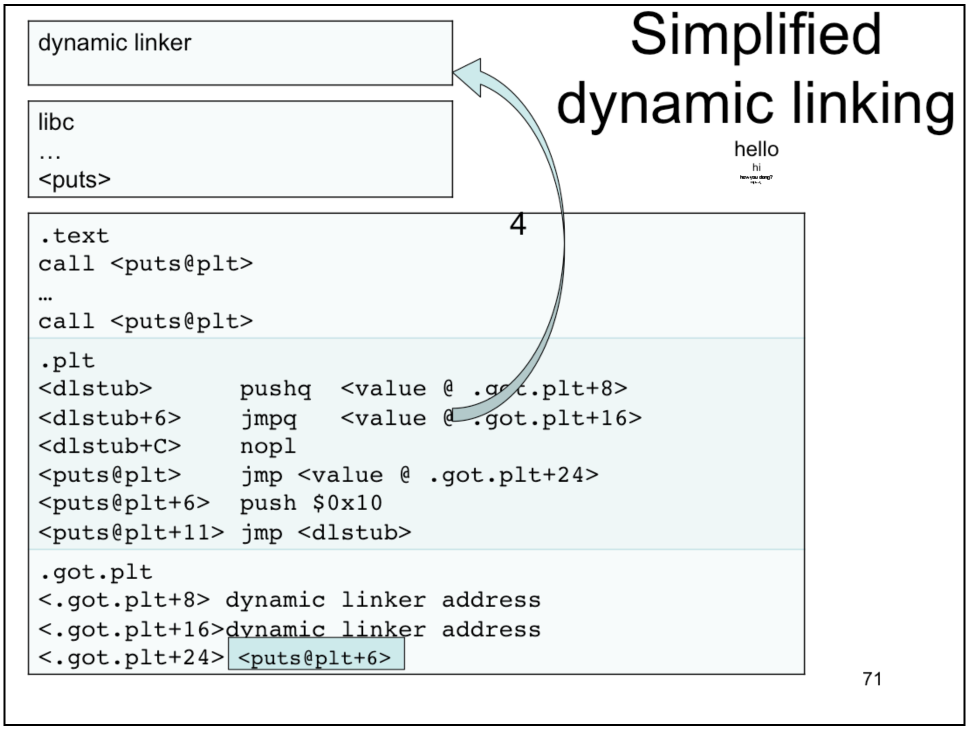
hi

how you doing?

dynamic linker

libc
…
<puts>

```
.text
call <puts@plt>
…
call <puts@plt>
.plt
<dlstub>      pushq  <value @ .got.plt+8>
<dlstub+6>    jmpq   <value @ .got.plt+16>
<dlstub+C>    nopl
<puts@plt>    jmp <value @ .got.plt+24>
<puts@plt+6>  push $0x10
<puts@plt+11> jmp <dlstub>
.got.plt
<.got.plt+8> dynamic linker address
<.got.plt+16>dynamic linker address
<.got.plt+24> <puts@plt+6>
```



Simplified dynamic linking

hello

hi

how you doing?

[illegible]

dynamic linker

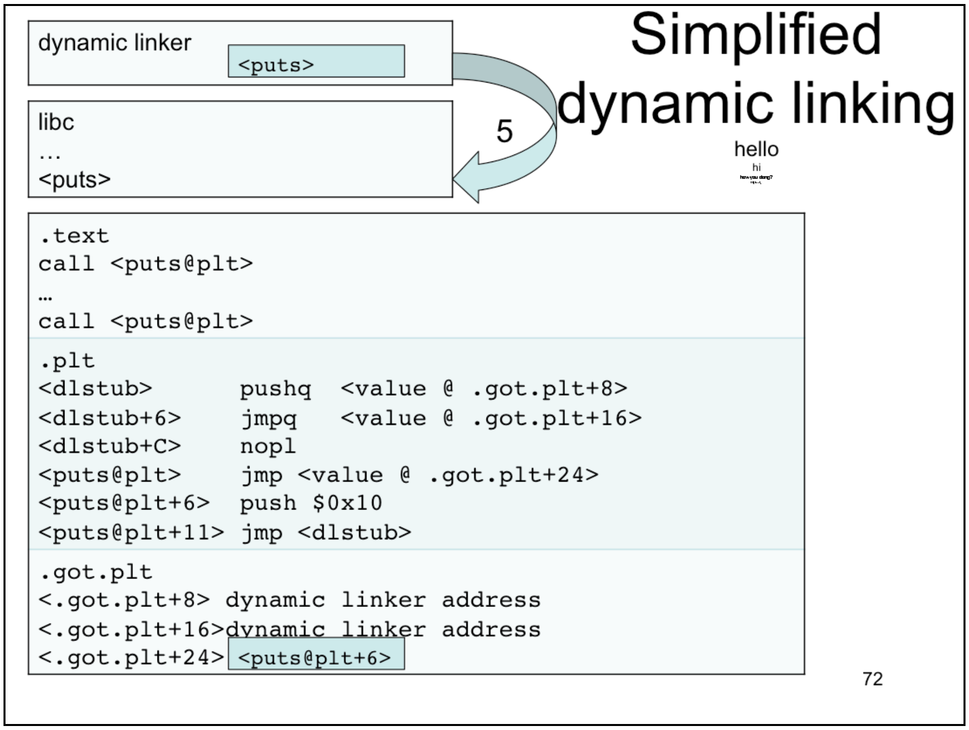
libc
…
<puts>

4

```
.text
call <puts@plt>
…
call <puts@plt>
```

```
.plt
<dlstub>      pushq  <value @ .got.plt+8>
<dlstub+6>    jmpq   <value @ .got.plt+16>
<dlstub+C>    nopl
<puts@plt>    jmp <value @ .got.plt+24>
<puts@plt+6>  push $0x10
<puts@plt+11> jmp <dlstub>
```

```
.got.plt
<.got.plt+8> dynamic linker address
<.got.plt+16>dynamic linker address
<.got.plt+24> <puts@plt+6>
```



Simplified dynamic linking

hello

hi

how you doing?

[illegible]

dynamic linker

```
<puts>
```

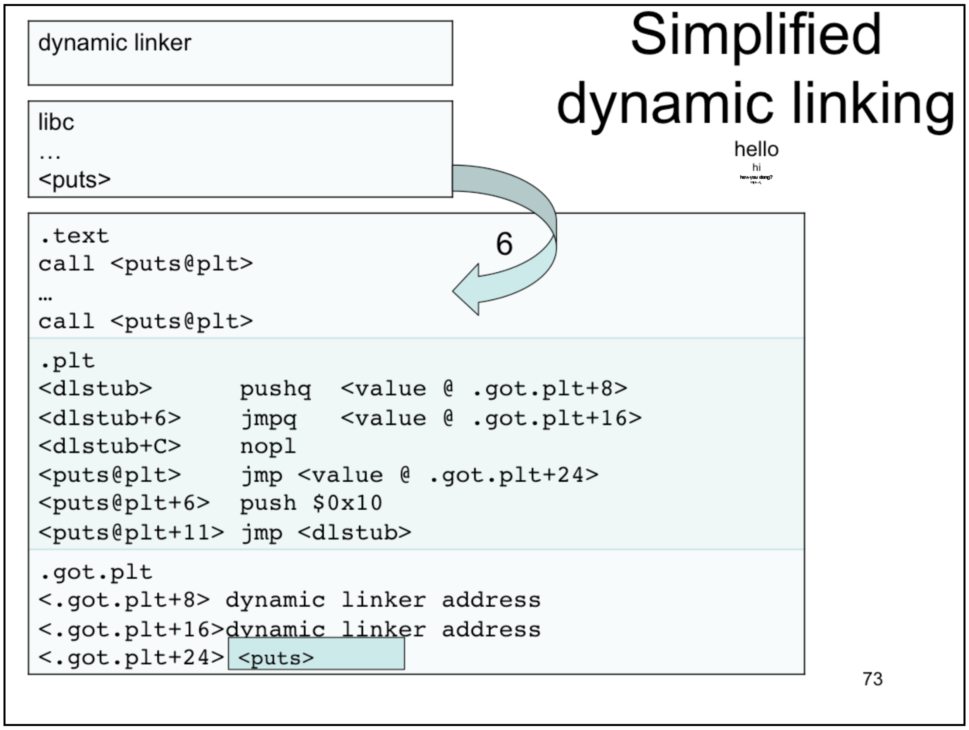
5

libc
…
<puts>

```
.text
call <puts@plt>
…
call <puts@plt>
```

```
.plt
<dlstub>       pushq  <value @ .got.plt+8>
<dlstub+6>     jmpq   <value @ .got.plt+16>
<dlstub+C>     nopl
<puts@plt>     jmp <value @ .got.plt+24>
<puts@plt+6>   push $0x10
<puts@plt+11>  jmp <dlstub>
```

```
.got.plt
<.got.plt+8> dynamic linker address
<.got.plt+16>dynamic linker address
<.got.plt+24> <puts@plt+6>
```



Simplified dynamic linking

hello

hi

how you doing?

[illegible]

dynamic linker

libc
…
<puts>

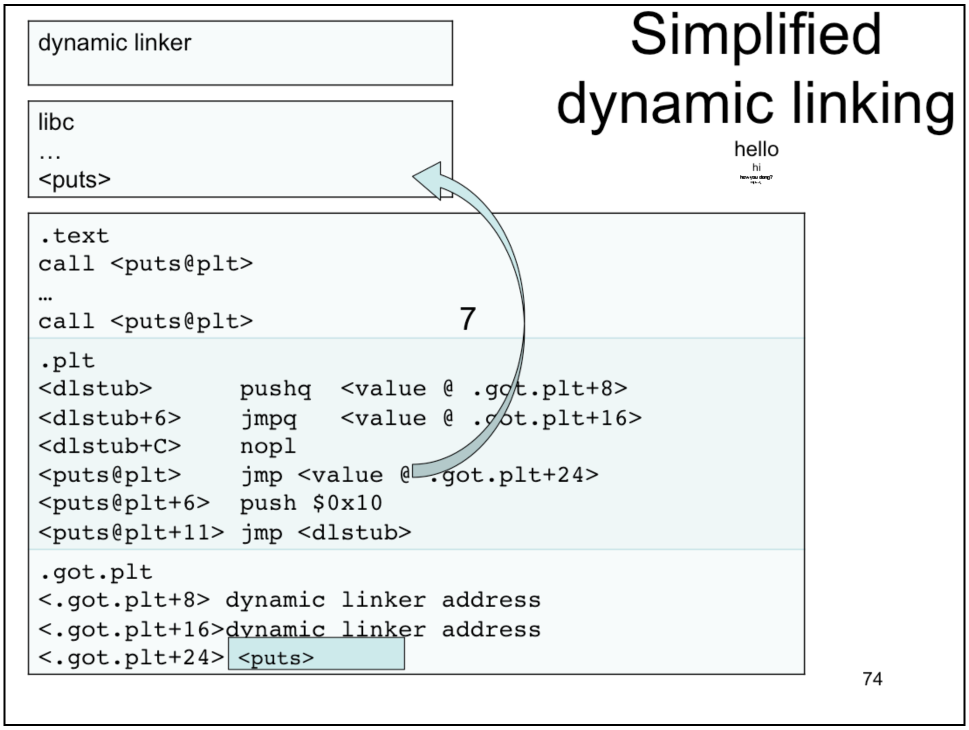
6

```
.text
call <puts@plt>
…
call <puts@plt>
```

```
.plt
<dlstub>       pushq  <value @ .got.plt+8>
<dlstub+6>     jmpq   <value @ .got.plt+16>
<dlstub+C>     nopl
<puts@plt>     jmp <value @ .got.plt+24>
<puts@plt+6>  push $0x10
<puts@plt+11> jmp <dlstub>
```

```
.got.plt
<.got.plt+8> dynamic linker address
<.got.plt+16>dynamic linker address
<.got.plt+24> <puts>
```

73



Simplified dynamic linking

hello

hi

how you doing?

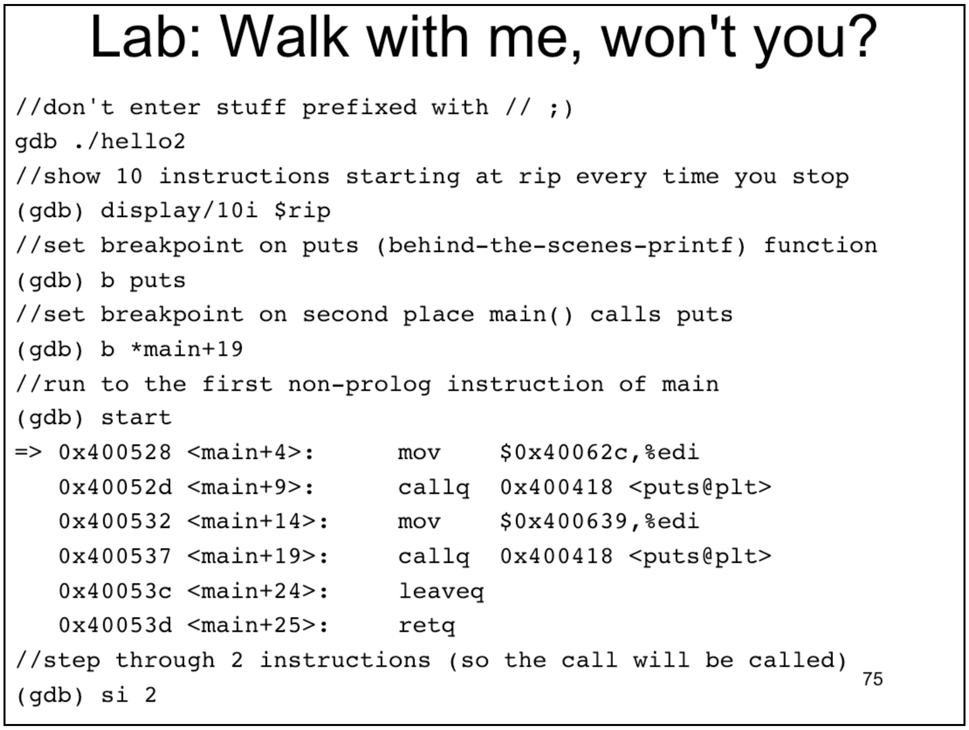
dynamic linker

libc
…
<puts>

```
.text
call <puts@plt>
…
call <puts@plt>
```

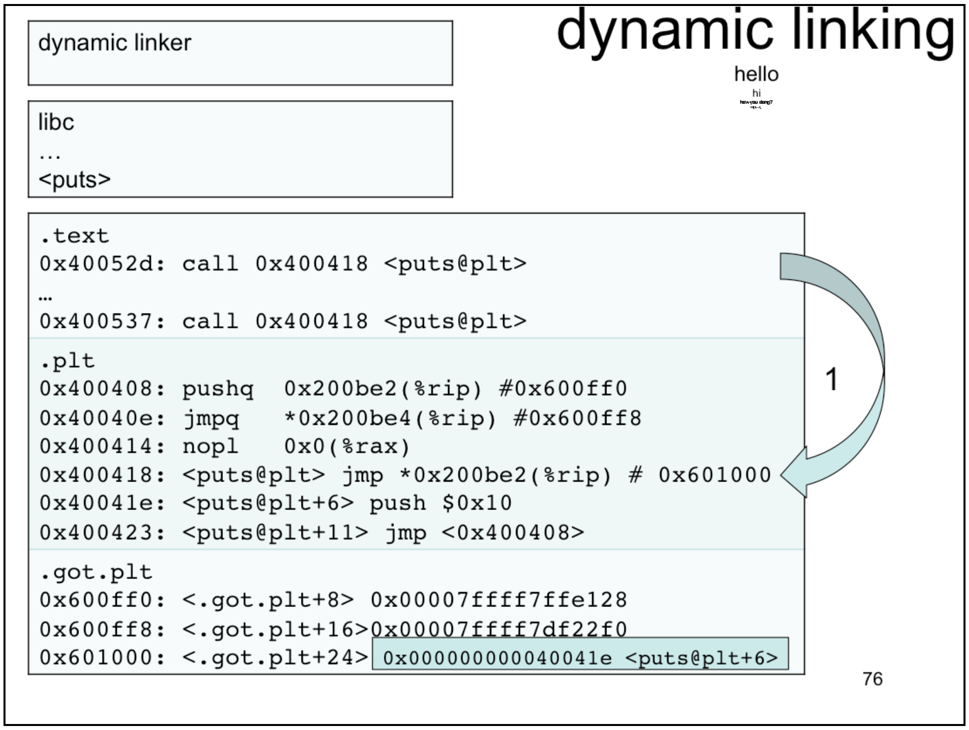
```
.plt
<dlstub>      pushq  <value @ .got.plt+8>
<dlstub+6>    jmpq   <value @ .got.plt+16>
<dlstub+C>    nopl
<puts@plt>    jmp <value @ .got.plt+24>
<puts@plt+6>  push $0x10
<puts@plt+11> jmp <dlstub>
```

```
.got.plt
<.got.plt+8> dynamic linker address
<.got.plt+16>dynamic linker address
<.got.plt+24> <puts>
```



Lab: Walk with me, won't you?

```
//don't enter stuff prefixed with // ;)
gdb ./hello2
//show 10 instructions starting at rip every time you stop
(gdb) display/10i $rip
//set breakpoint on puts (behind-the-scenes-printf) function
(gdb) b puts
//set breakpoint on second place main() calls puts
(gdb) b *main+19
//run to the first non-prolog instruction of main
(gdb) start
=> 0x400528 <main+4>:      mov    $0x40062c,%edi
   0x40052d <main+9>:      callq  0x400418 <puts@plt>
   0x400532 <main+14>:     mov    $0x400639,%edi
   0x400537 <main+19>:     callq  0x400418 <puts@plt>
   0x40053c <main+24>:     leaveq
   0x40053d <main+25>:     retq
//step through 2 instructions (so the call will be called)
(gdb) si 2
```



dynamic linking

hello

hi

how you doing?

dynamic linker

libc
…
<puts>

```
.text
0x40052d: call 0x400418 <puts@plt>
…
0x400537: call 0x400418 <puts@plt>
.plt
0x400408: pushq  0x200be2(%rip) #0x600ff0
0x40040e: jmpq   *0x200be4(%rip) #0x600ff8
0x400414: nopl   0x0(%rax)
0x400418: <puts@plt> jmp *0x200be2(%rip) # 0x601000
0x40041e: <puts@plt+6> push $0x10
0x400423: <puts@plt+11> jmp <0x400408>
.got.plt
0x600ff0: <.got.plt+8> 0x00007ffff7ffe128
0x600ff8: <.got.plt+16>0x00007ffff7df22f0
0x601000: <.got.plt+24> 0x000000000040041e <puts@plt+6>
```

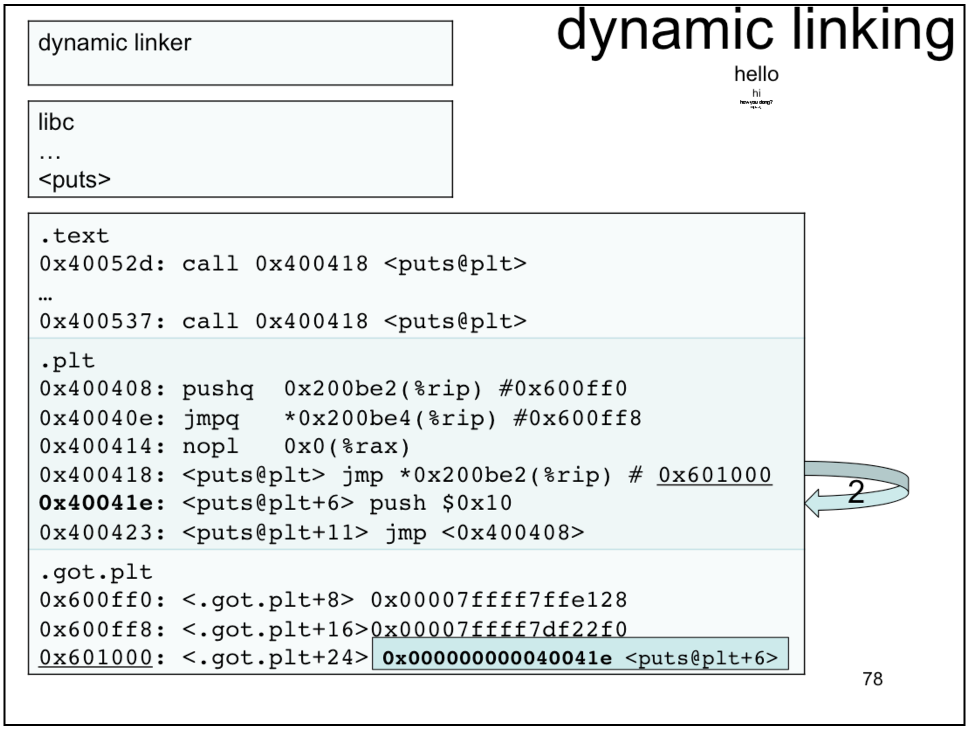
A lovely day for a stroll

```
=> 0x400418 <puts@plt>:    jmpq   *0x200be2(%rip)
                           # 0x601000 <_GLOBAL_OFFSET_TABLE_+24>
   0x40041e <puts@plt+6>:  pushq  $0x0
   0x400423 <puts@plt+11>: jmpq   0x400408
//rip-relative addressing always starts at the next instruction
(gdb) x/x 0x40041e + 0x200be2
0x601000 <_GLOBAL_OFFSET_TABLE_+24>:    0x0040041e
```

```
Section Headers:
Name      Type       Address           Offset   Size              EntSize           Flags  Link  Info  Align
.got.plt  PROGBITS   0000000000600fe8  00000fe8 0000000000000028  0000000000000008  WA       0     0     8

(range = 0x600fe8 to 0x601010)
```

So basically this is going to look up a value from the .got.plt, and then jump there. It just so happens that the value is the address of the next instruction! But the whole point of this exercise is that later on that value in the .got.plt will change to the address of puts()



dynamic linking

hello

hi

how you doing?

dynamic linker

libc
…
<puts>

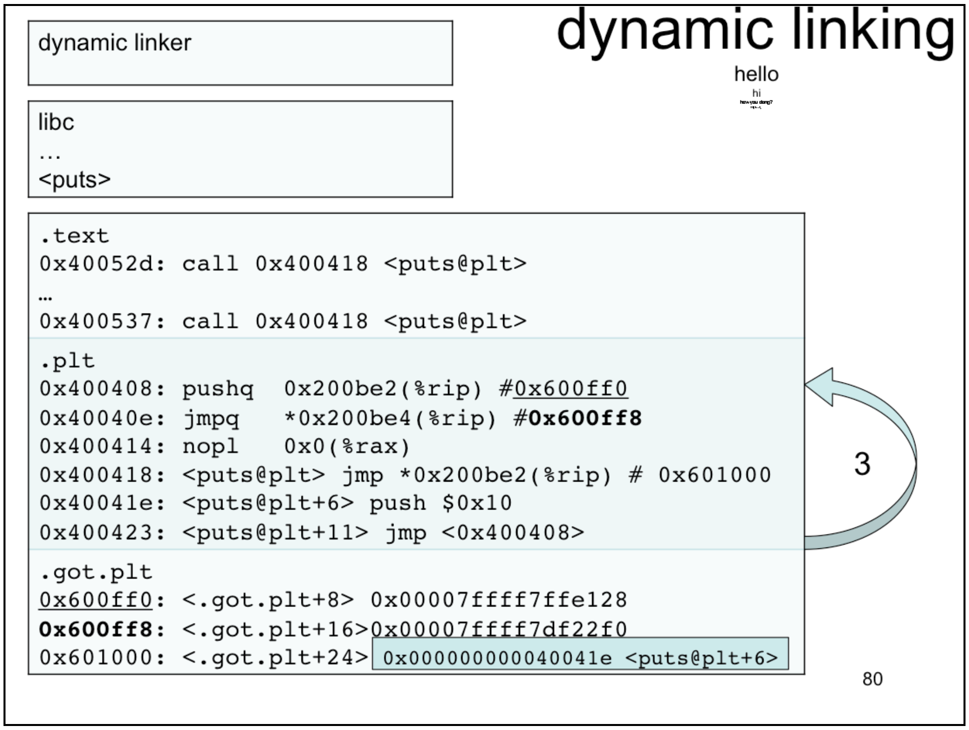
```
.text
0x40052d: call 0x400418 <puts@plt>
…
0x400537: call 0x400418 <puts@plt>
.plt
0x400408: pushq  0x200be2(%rip) #0x600ff0
0x40040e: jmpq   *0x200be4(%rip) #0x600ff8
0x400414: nopl   0x0(%rax)
0x400418: <puts@plt> jmp *0x200be2(%rip) # 0x601000
0x40041e: <puts@plt+6> push $0x10
0x400423: <puts@plt+11> jmp <0x400408>
.got.plt
0x600ff0: <.got.plt+8> 0x00007ffff7ffe128
0x600ff8: <.got.plt+16>0x00007ffff7df22f0
0x601000: <.got.plt+24> 0x000000000040041e <puts@plt+6>
```

78

What dread spectre approaches yonder?

```
//step one instruction to confirm it just jmps to the next
(gdb) si 1
=> 0x40041e <puts@plt+6>: pushq  $0x0
   0x400423 <puts@plt+11>:      jmpq   0x400408
//execute the next 2 instructions to go to the final stub
//that looks up the dynamic linker's location
(gdb) si 2
=> 0x400408: pushq  0x200be2(%rip)
                   # 0x600ff0 <_GLOBAL_OFFSET_TABLE_+8>
   0x40040e: jmpq   *0x200be4(%rip)
                   # 0x600ff8 <_GLOBAL_OFFSET_TABLE_+16>
   0x400414: nopl   0x0(%rax)

(gdb) x/xg 0x600ff0
0x600ff0 <_GLOBAL_OFFSET_TABLE_+8>:   0x00007ffff7ffe128
(gdb) x/xg 0x600ff8
0x600ff8 <_GLOBAL_OFFSET_TABLE_+16>:  0x00007ffff7df22f0
```



dynamic linking

hello

hi

how you doing?

[illegible]

dynamic linker

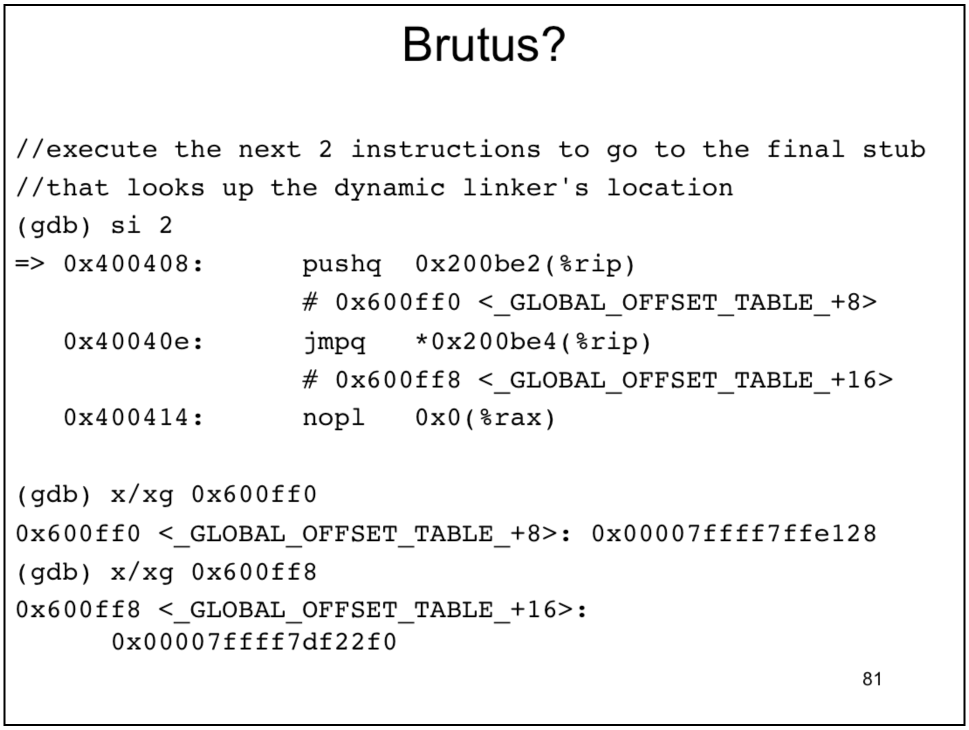
libc
…
<puts>

```
.text
0x40052d: call 0x400418 <puts@plt>
…
0x400537: call 0x400418 <puts@plt>
```

```
.plt
0x400408: pushq  0x200be2(%rip) #0x600ff0
0x40040e: jmpq   *0x200be4(%rip) #0x600ff8
0x400414: nopl   0x0(%rax)
0x400418: <puts@plt> jmp *0x200be2(%rip) # 0x601000
0x40041e: <puts@plt+6> push $0x10
0x400423: <puts@plt+11> jmp <0x400408>
```

```
.got.plt
0x600ff0: <.got.plt+8> 0x00007ffff7ffe128
0x600ff8: <.got.plt+16>0x00007ffff7df22f0
0x601000: <.got.plt+24> 0x000000000040041e <puts@plt+6>
```

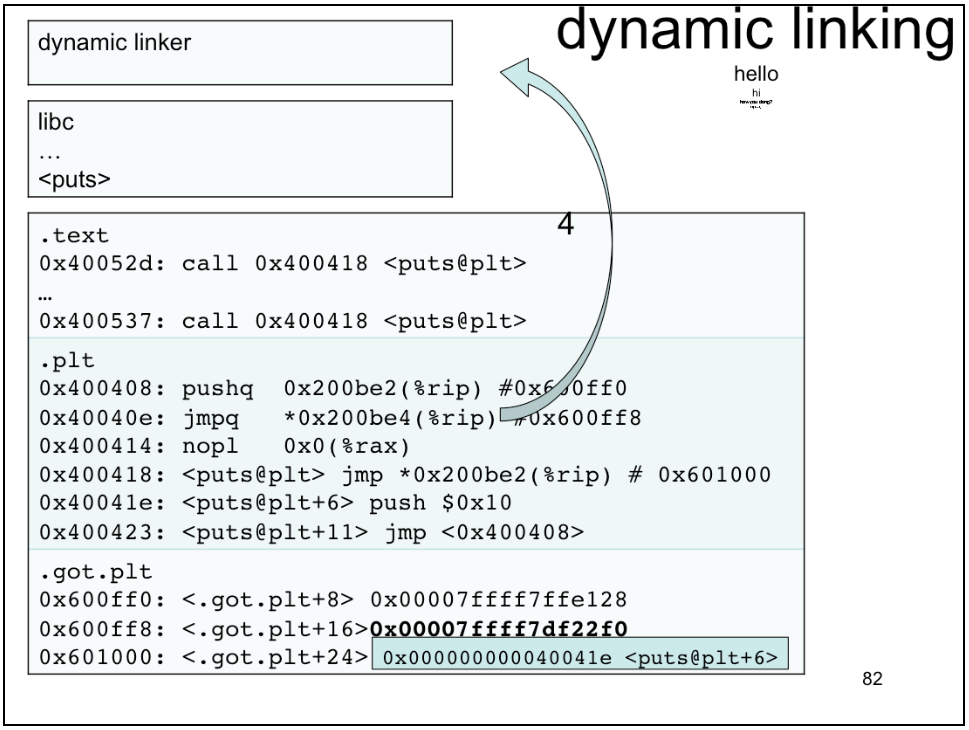
80



Brutus?

```
//execute the next 2 instructions to go to the final stub
//that looks up the dynamic linker's location
(gdb) si 2
=> 0x400408:      pushq  0x200be2(%rip)
                  # 0x600ff0 <_GLOBAL_OFFSET_TABLE_+8>
   0x40040e:      jmpq   *0x200be4(%rip)
                  # 0x600ff8 <_GLOBAL_OFFSET_TABLE_+16>
   0x400414:      nopl   0x0(%rax)

(gdb) x/xg 0x600ff0
0x600ff0 <_GLOBAL_OFFSET_TABLE_+8>: 0x00007ffff7ffe128
(gdb) x/xg 0x600ff8
0x600ff8 <_GLOBAL_OFFSET_TABLE_+16>:
      0x00007ffff7df22f0
```



dynamic linking

hello

hi

how you doing?

dynamic linker

libc
…
<puts>

```
.text
0x40052d: call 0x400418 <puts@plt>
…
0x400537: call 0x400418 <puts@plt>
.plt
0x400408: pushq  0x200be2(%rip) #0x600ff0
0x40040e: jmpq   *0x200be4(%rip) #0x600ff8
0x400414: nopl   0x0(%rax)
0x400418: <puts@plt> jmp *0x200be2(%rip) # 0x601000
0x40041e: <puts@plt+6> push $0x10
0x400423: <puts@plt+11> jmp <0x400408>
.got.plt
0x600ff0: <.got.plt+8> 0x00007ffff7ffe128
0x600ff8: <.got.plt+16>0x00007ffff7df22f0
0x601000: <.got.plt+24> 0x000000000040041e <puts@plt+6>
```

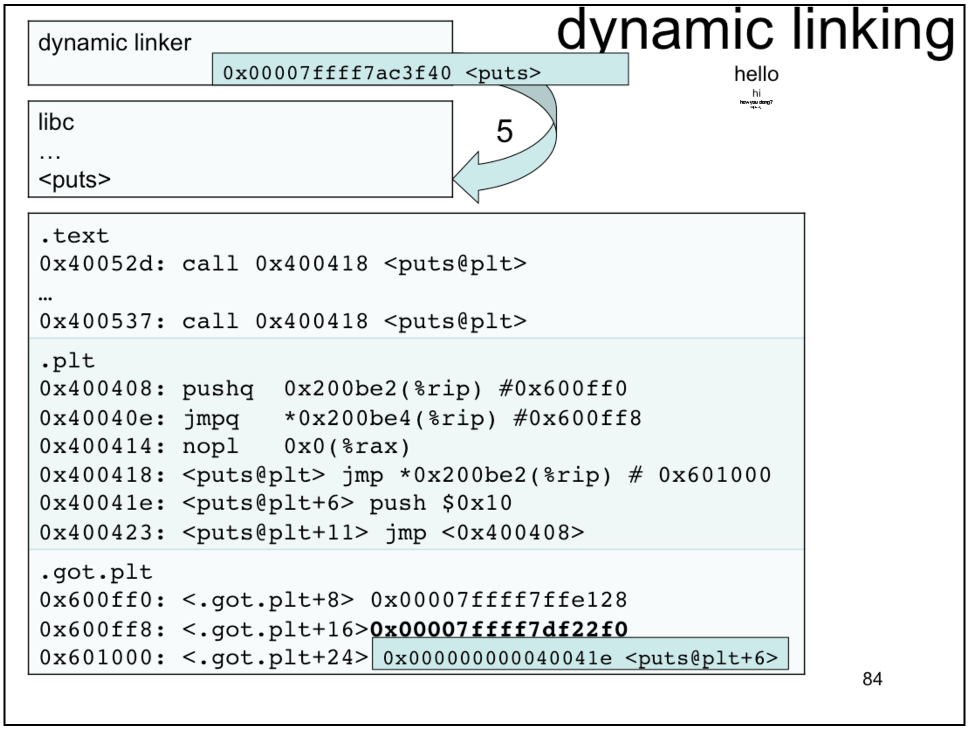
82

Foofus?

```
//execute the jmp to the dynamic linker
(gdb) si 2
0x00007ffff7df22f0 in ?? () from /lib64/ld-linux-x86-64.so.2
1: x/10i $rip
=> 0x7ffff7df22f0:  sub    $0x38,%rsp
   0x7ffff7df22f4:  mov    %rax,(%rsp)
   0x7ffff7df22f8:  mov    %rcx,0x8(%rsp)

The dynamic linker will fill in
```

In the dynamic linker



dynamic linking

hello

hi

how you doing?

dynamic linker

```
0x00007ffff7ac3f40 <puts>
```

5

libc
…
<puts>

```
.text
0x40052d: call 0x400418 <puts@plt>
…
0x400537: call 0x400418 <puts@plt>
```

```
.plt
0x400408: pushq  0x200be2(%rip) #0x600ff0
0x40040e: jmpq   *0x200be4(%rip) #0x600ff8
0x400414: nopl   0x0(%rax)
0x400418: <puts@plt> jmp *0x200be2(%rip) # 0x601000
0x40041e: <puts@plt+6> push $0x10
0x400423: <puts@plt+11> jmp <0x400408>
```

```
.got.plt
0x600ff0: <.got.plt+8> 0x00007ffff7ffe128
0x600ff8: <.got.plt+16>0x00007ffff7df22f0
0x601000: <.got.plt+24> 0x000000000040041e <puts@plt+6>
```

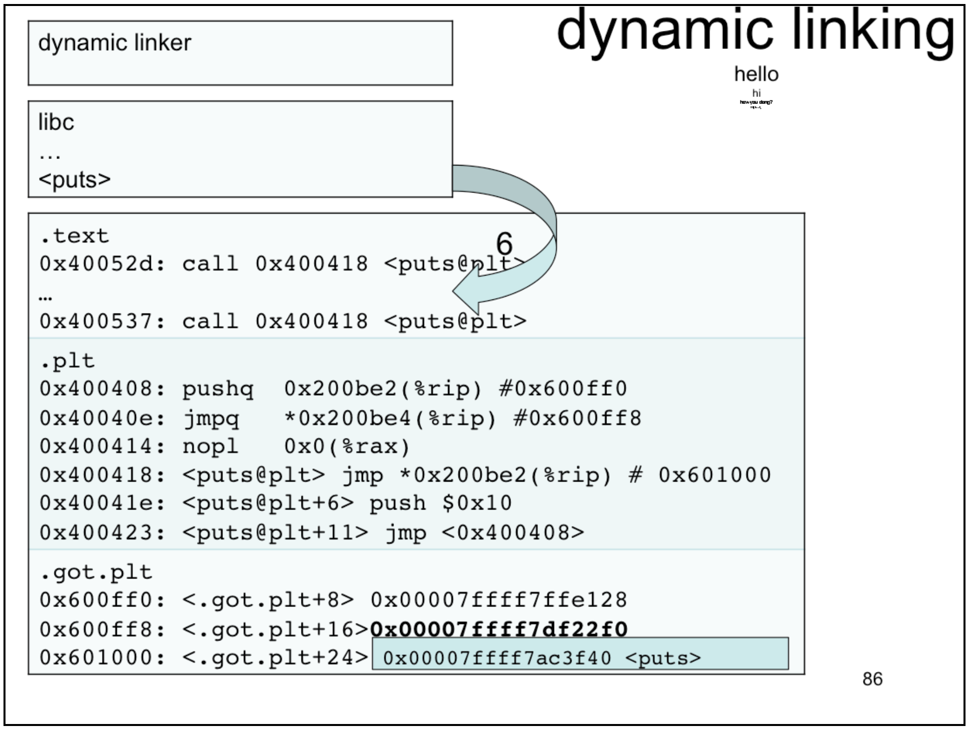
84

The stillness of the air portends our doom, wouldn't you say?

```
//let it continue until it hits our breakpoint on puts
(gdb) c
Continuing.
Breakpoint 1, 0x00007ffff7ac3f40 in puts () from /lib/libc.so.6
1: x/10i $rip
=> 0x7ffff7ac3f40 <puts>:     mov    %rbx,-0x18(%rsp)
   0x7ffff7ac3f45 <puts+5>:   mov    %rbp,-0x10(%rsp)

//let it continue and get through the puts and get back to our main()
(gdb) c
Continuing.
Hello world!

Breakpoint 2, 0x0000000000400537 in main ()
1: x/10i $rip
=> 0x400537 <main+19>: callq  0x400418 <puts@plt>
   0x40053c <main+24>: leaveq
   0x40053d <main+25>: retq
```



dynamic linking

hello

hi

how you doing?

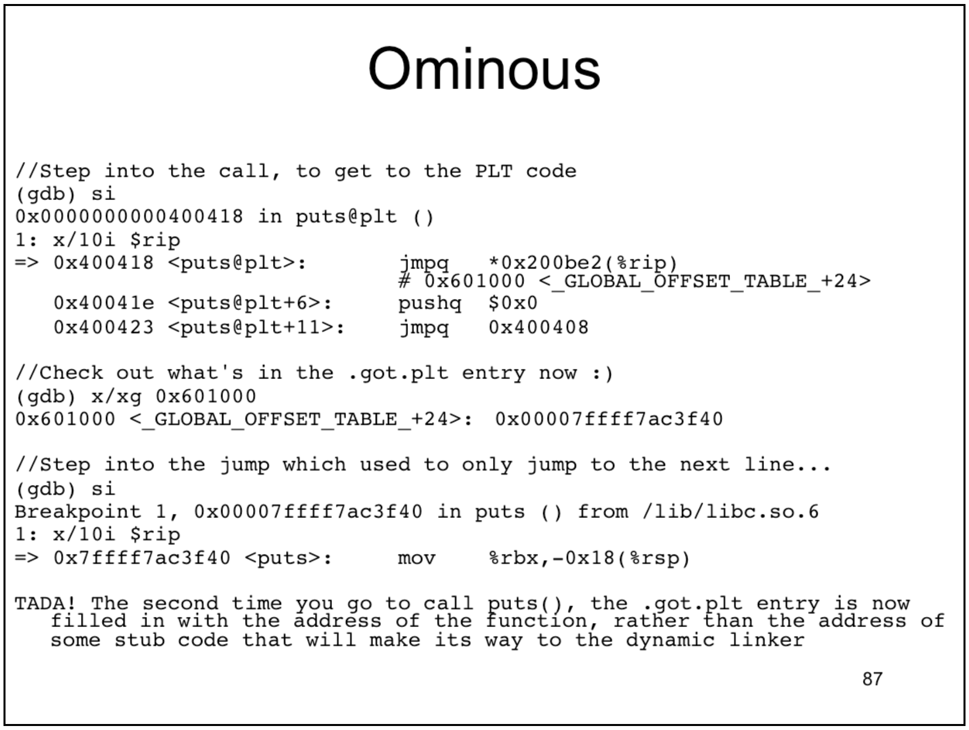
dynamic linker

libc
…
<puts>

```
.text
0x40052d: call 0x400418 <puts@plt>
…
0x400537: call 0x400418 <puts@plt>
```

```
.plt
0x400408: pushq  0x200be2(%rip) #0x600ff0
0x40040e: jmpq   *0x200be4(%rip) #0x600ff8
0x400414: nopl   0x0(%rax)
0x400418: <puts@plt> jmp *0x200be2(%rip) # 0x601000
0x40041e: <puts@plt+6> push $0x10
0x400423: <puts@plt+11> jmp <0x400408>
```

```
.got.plt
0x600ff0: <.got.plt+8> 0x00007ffff7ffe128
0x600ff8: <.got.plt+16>0x00007ffff7df22f0
0x601000: <.got.plt+24> 0x00007ffff7ac3f40 <puts>
```



Ominous

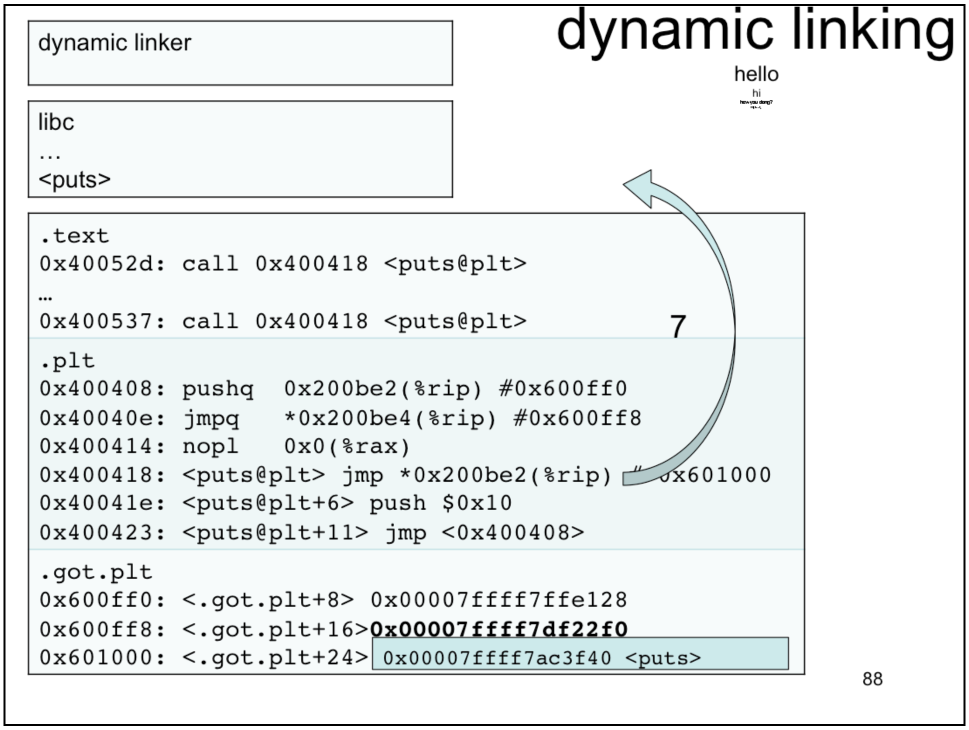
```
//Step into the call, to get to the PLT code
(gdb) si
0x0000000000400418 in puts@plt ()
1: x/10i $rip
=> 0x400418 <puts@plt>:       jmpq   *0x200be2(%rip)
                              # 0x601000 <_GLOBAL_OFFSET_TABLE_+24>
   0x40041e <puts@plt+6>:     pushq  $0x0
   0x400423 <puts@plt+11>:    jmpq   0x400408

//Check out what's in the .got.plt entry now :)
(gdb) x/xg 0x601000
0x601000 <_GLOBAL_OFFSET_TABLE_+24>:  0x00007ffff7ac3f40

//Step into the jump which used to only jump to the next line...
(gdb) si
Breakpoint 1, 0x00007ffff7ac3f40 in puts () from /lib/libc.so.6
1: x/10i $rip
=> 0x7ffff7ac3f40 <puts>:     mov    %rbx,-0x18(%rsp)

TADA! The second time you go to call puts(), the .got.plt entry is now
   filled in with the address of the function, rather than the address of
   some stub code that will make its way to the dynamic linker
```

87



dynamic linking

hello

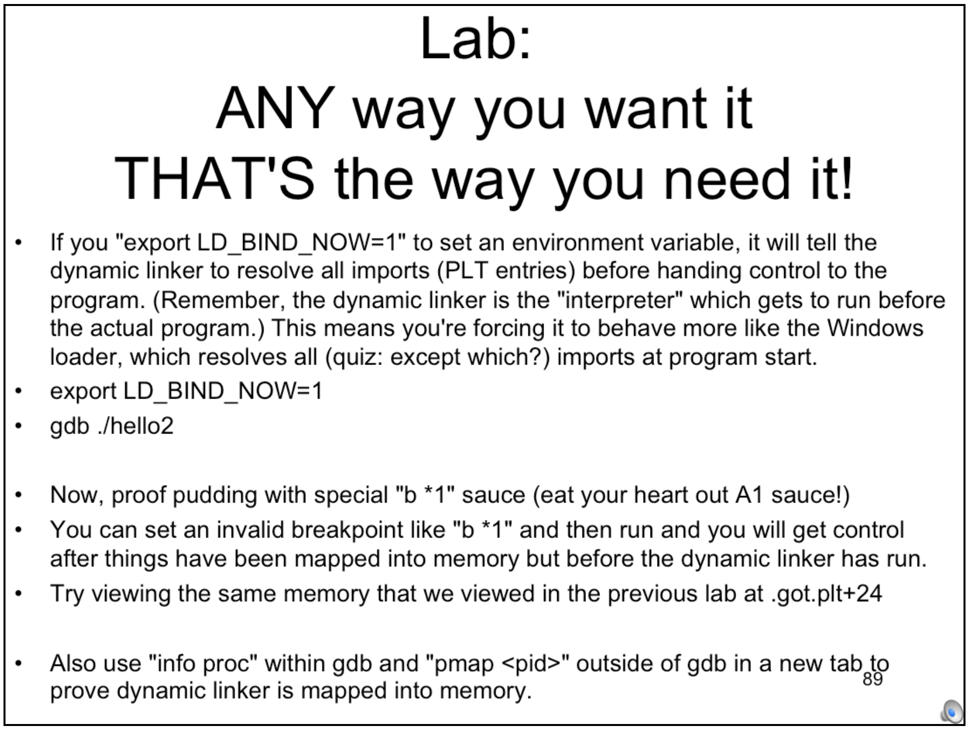
hi

how you doing?

dynamic linker

libc
…
<puts>

```
.text
0x40052d: call 0x400418 <puts@plt>
…
0x400537: call 0x400418 <puts@plt>
.plt
0x400408: pushq  0x200be2(%rip) #0x600ff0
0x40040e: jmpq   *0x200be4(%rip) #0x600ff8
0x400414: nopl   0x0(%rax)
0x400418: <puts@plt> jmp *0x200be2(%rip) #0x601000
0x40041e: <puts@plt+6> push $0x10
0x400423: <puts@plt+11> jmp <0x400408>
.got.plt
0x600ff0: <.got.plt+8> 0x00007ffff7ffe128
0x600ff8: <.got.plt+16>0x00007ffff7df22f0
0x601000: <.got.plt+24> 0x00007ffff7ac3f40 <puts>
```



Lab:
ANY way you want it
THAT'S the way you need it!

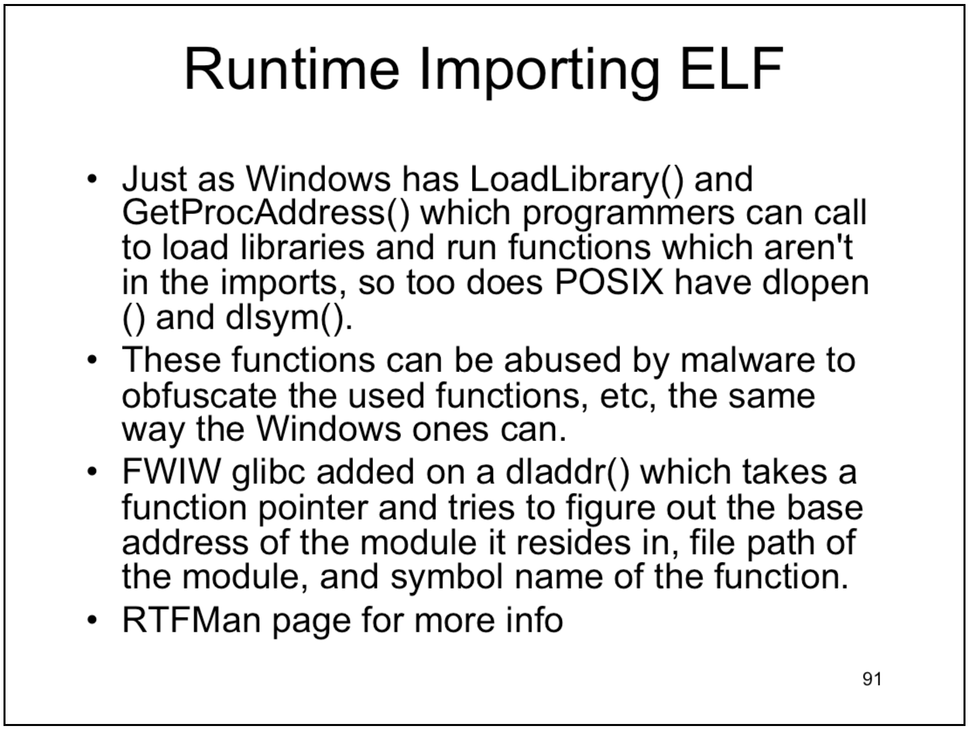
- If you "export LD_BIND_NOW=1" to set an environment variable, it will tell the dynamic linker to resolve all imports (PLT entries) before handing control to the program. (Remember, the dynamic linker is the "interpreter" which gets to run before the actual program.) This means you're forcing it to behave more like the Windows loader, which resolves all (quiz: except which?) imports at program start.
- export LD_BIND_NOW=1
- gdb ./hello2

- Now, proof pudding with special "b *1" sauce (eat your heart out A1 sauce!)
- You can set an invalid breakpoint like "b *1" and then run and you will get control after things have been mapped into memory but before the dynamic linker has run.
- Try viewing the same memory that we viewed in the previous lab at .got.plt+24

- Also use "info proc" within gdb and "pmap <pid>" outside of gdb in a new tab to prove dynamic linker is mapped into memory.

PLT hooking

- Just like with IAT hooking on PE, we can do PLT hooking on ELF. But as with before, we first need a way to get into the memory address space.
- We will use the cheap LD_PRELOAD environment variable way of "shared object injection" (like DLL injection). With this variable set, the specified shared object will be loaded into memory earlier than any of the other imported shared objects, for every executable which is started



Runtime Importing ELF

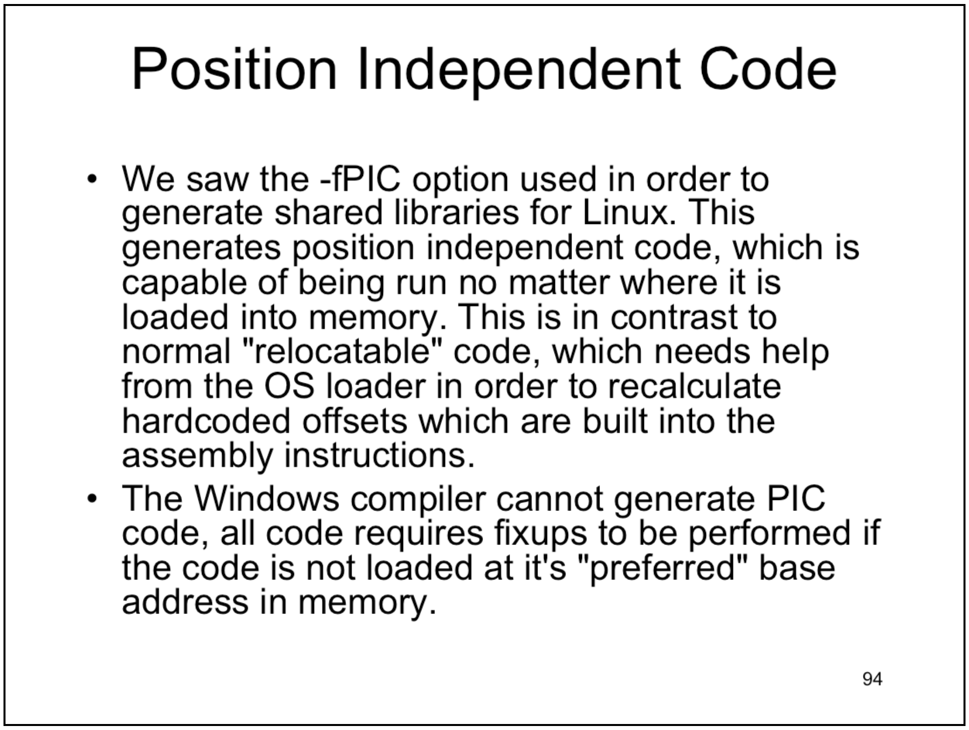
- Just as Windows has LoadLibrary() and GetProcAddress() which programmers can call to load libraries and run functions which aren't in the imports, so too does POSIX have dlopen () and dlsym().
- These functions can be abused by malware to obfuscate the used functions, etc, the same way the Windows ones can.
- FWIW glibc added on a dladdr() which takes a function pointer and tries to figure out the base address of the module it resides in, file path of the module, and symbol name of the function.
- RTFMan page for more info

Thread Local Storage

- Much more about TLS:
- http://www.akkadia.org/drepper/tls.pdf
- Sorry, no callback functions like PE, thus making it fairly uninteresting. Instead, there's just an initialization blob that gets written into the location of the vars, so that they have their initialization values.

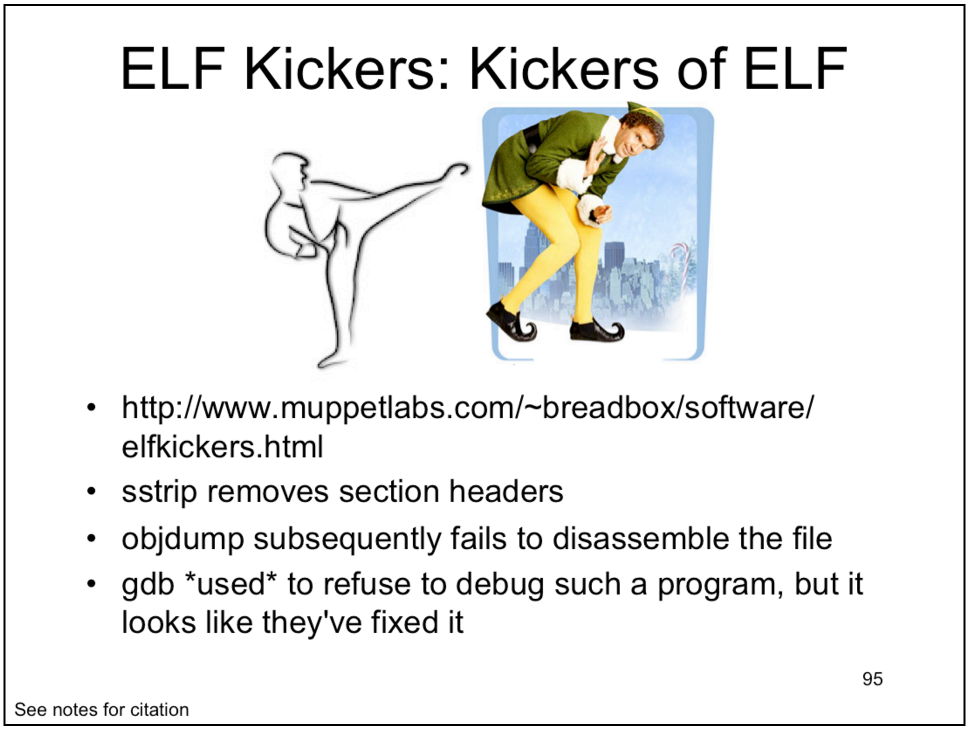
TODO:

- Start talking about linking process again
- Then cover some compiler options and linker options



Position Independent Code

- We saw the -fPIC option used in order to generate shared libraries for Linux. This generates position independent code, which is capable of being run no matter where it is loaded into memory. This is in contrast to normal "relocatable" code, which needs help from the OS loader in order to recalculate hardcoded offsets which are built into the assembly instructions.
- The Windows compiler cannot generate PIC code, all code requires fixups to be performed if the code is not loaded at it's "preferred" base address in memory.



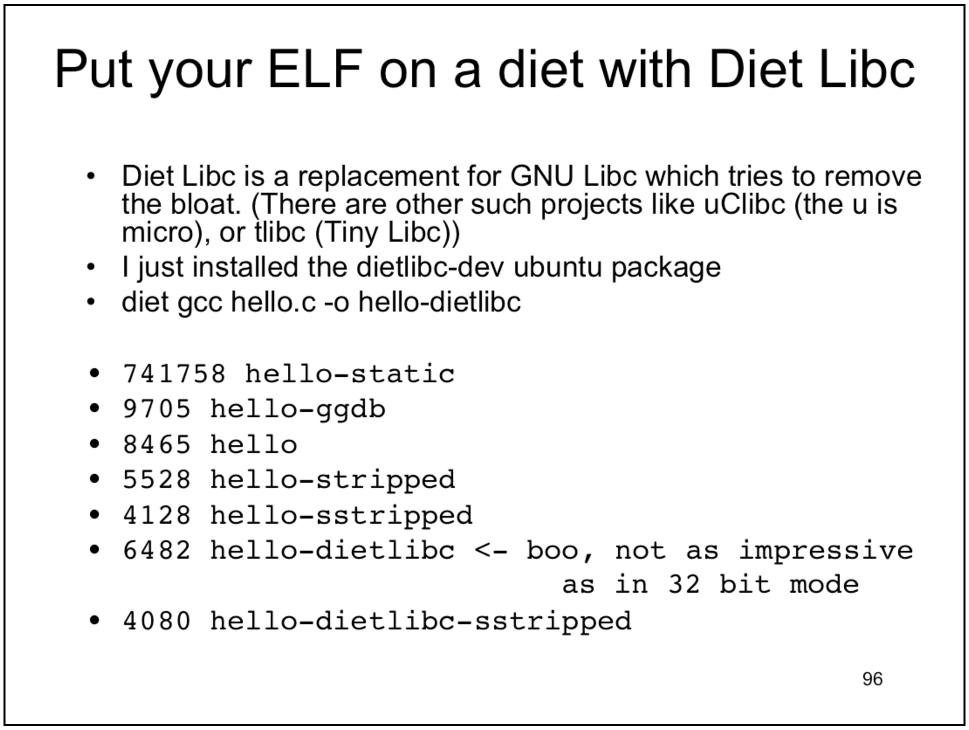
ELF Kickers: Kickers of ELF

- http://www.muppetlabs.com/~breadbox/software/elfkickers.html
- sstrip removes section headers
- objdump subsequently fails to disassemble the file
- gdb *used* to refuse to debug such a program, but it looks like they've fixed it

See notes for citation

http://api.ning.com/files/5sO-AF7JHU65FO2*bEaVplCjT25GsRtNdE6irPdbz-SLMAdLCL3AbEtHCahv4aCgKDUhkzA61sDnPoLtorAE0GEX**deFgQ8/Elf.jpg

http://isobe.typepad.com/photos/illustrations/karatekickweb.jpg



Put your ELF on a diet with Diet Libc

- Diet Libc is a replacement for GNU Libc which tries to remove the bloat. (There are other such projects like uClibc (the u is micro), or tlibc (Tiny Libc))
- I just installed the dietlibc-dev ubuntu package
- diet gcc hello.c -o hello-dietlibc

```
• 741758 hello-static
• 9705 hello-ggdb
• 8465 hello
• 5528 hello-stripped
• 4128 hello-sstripped
• 6482 hello-dietlibc <- boo, not as impressive
                         as in 32 bit mode
• 4080 hello-dietlibc-sstripped
```

Created the files with:

* gcc -static -o hello-static hello.c

* gcc -ggdb –o hello-ggdb hello

* gcc –o hello hello

* strip hello -o hello-stripped

* cd ~/code/ELFkickers-3.0a/sstrip; cp ../../hello/hello .; ./sstrip hello; cp hello hello-sstripped

* sudo apt-get install dietlibc-dev; diet gcc -o hello-dietlibc hello.c

* cd ~/code/ELFkickers-3.0a/sstrip; cp ../../hello/hello-dietlibc .; ./sstrip hello-dietlibc ; cp hello-dietlibc hello-dietlibc-sstripped