x86/x64 Assembly on Linux

A short introduction for reverse engineers

Giovanni Lagorio

giovanni.lagorio@unige.it
https://zxgio.sarahah.com

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova
Italy

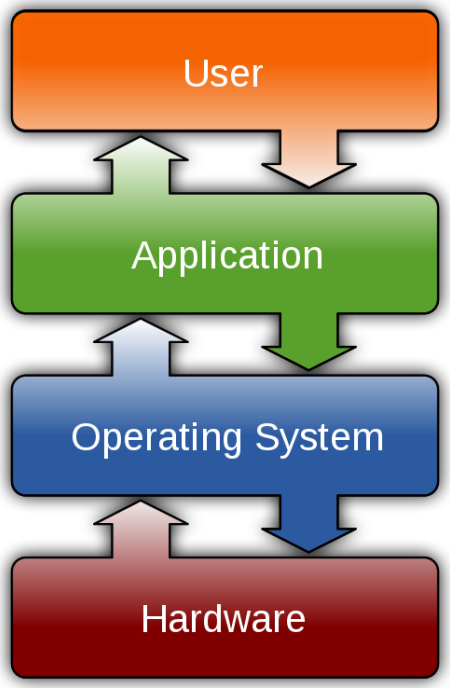
December 12, 2017

Outline

Introduction

Unless you plan to work on the bare metal, the ingredients of your abstract machine are:

1. Application Programming Interface of the OS
2. Instruction Set Architecture, an abstract model of a CPU
3. Application Binary Interface, a set of specifications detailing:
   - executable and object file formats
   - fundamental types (e.g. size and alignment for int, long, ...)
   - (sys)calling conventions
   - dynamic linking semantics



https://en.wikipedia.org/wiki/File:Operating_system_placement.svg

CPU: the Intel x86/x64

We'll deal with user-mode:

- x86/IA-32 ("Intel Architecture, 32-bit", sometimes called i386)
- x86-64/x64/IA-64 (IA-32 with 64-bit extension)/AMD64 is an extension to original IA-32

Documentation:

- Intel 64 and IA-32 Architectures Software Developer Manuals
  `https://software.intel.com/en-us/articles/intel-sdm`
  at the time of writing, a handy 4768 (!) page reference
- Cheat-sheet: `http://www.jegerlehner.ch/intel/` (32 bit only)
- Wikipedia:
  `https://en.wikipedia.org/wiki/X86_instruction_listings`
- In r2: `?d` and `asm.describe`

ABI

We'll deal with:

- Executable and Linking Format (ELF), and
- System V ABI

used by major "Unixes"

- i386 (32-bit)
  `http://refspecs.linuxbase.org/elf/abi386-4.pdf`
- x86-64 (64-bit)
  `http://refspecs.linuxbase.org/elf/x86_64-abi-0.95.pdf`

Outline

A little history

1978 16-bit processors: 8086 and 8088 (8-bit bus); segmentation, $2^{20} = 1$ MB address space. So, 16 bit words, forever ☺

1982 286, protected mode using segment registers as selector, $2^{24} = 16$ MB address space

1985 386, 32-bit processor, virtual-8086 mode, $2^{32} = 4$ GB address space, segmented-memory model and flat memory model, paging with 4k pages

1989 486, integrated x87 FPU

1993 Pentium, 4k and 4M pages

1995-1999 P6 family, MMX and SSE $\rightarrow$ SIMD parallelism

2000-2007 Pentium 4/Xeon family, SSE2 and SS3; introduction of IA64, hyperthreading and VT

2008 Core i7 family, SSE4.2, 2nd generation Virtualization Technology

...

Modes of operation

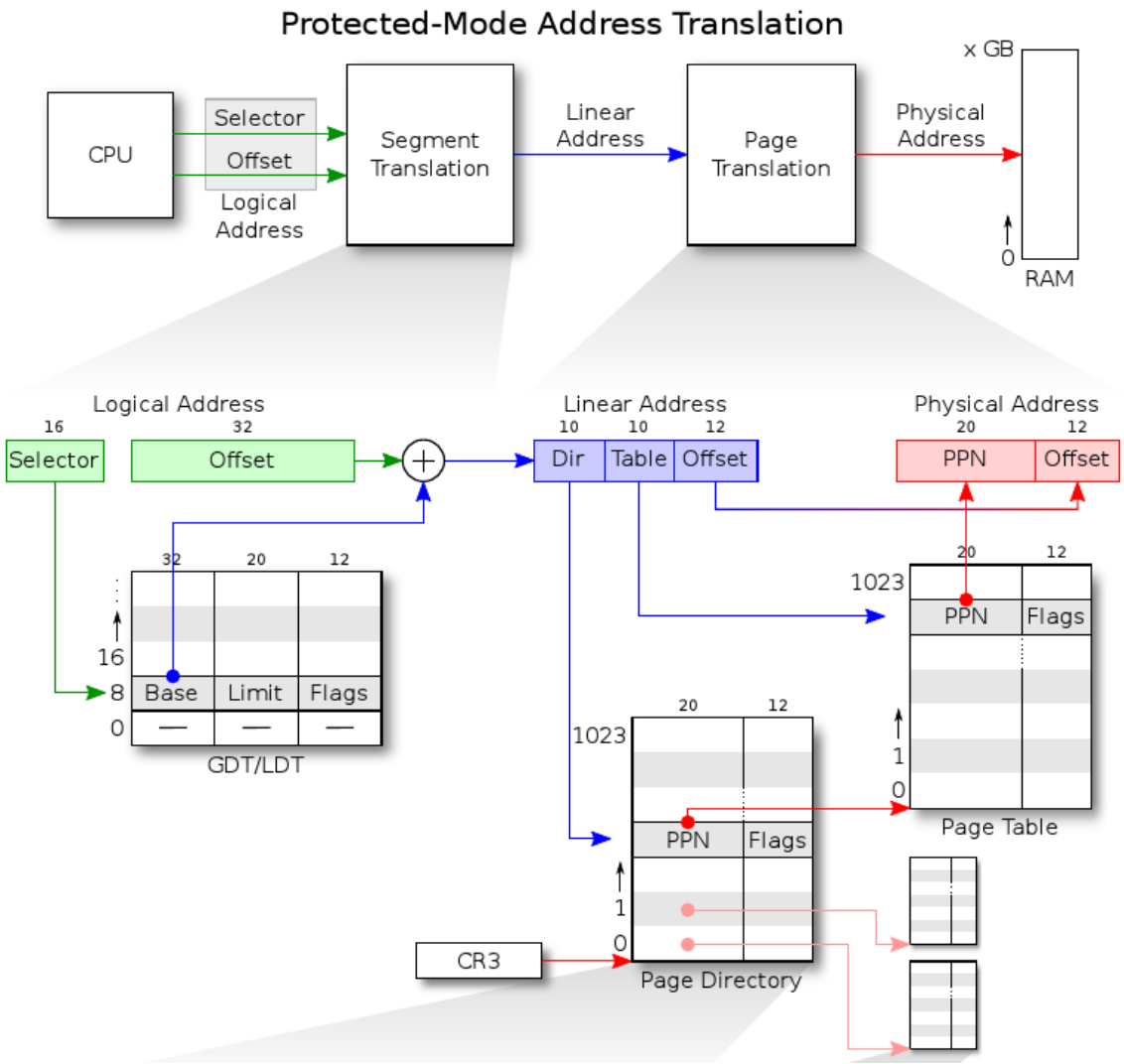
32 bit:

- real-address mode, the "8086 mode"; activated at power-up
- protected mode, the "normal" mode
  - four protection rings, two used: ring 0 (kernel) and ring 3 (user)
  - applications run with a paged 32-bit flat address space
- *system-management mode*, special-purpose operating mode intended for use only by firmware

64 bit:

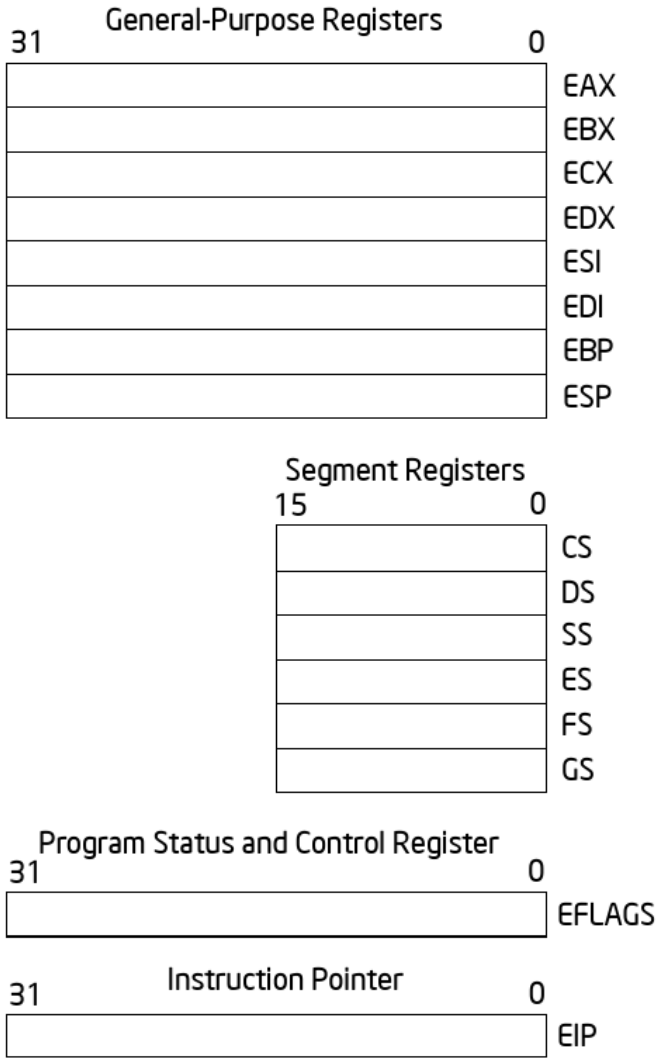
- IA32e, with two submodes:
  - compatibility mode, similar to 32-bit protected mode, permits legacy 16/32-bit application to run without recompilation on a 64-bit OS. Enabled on code-segment basis. Allows to access $2^{36} = 64$ GB of physical memory using Physical Address Extensions
  - 64-bit mode, allows to run 64-bit applications, extending general purpose (and SIMD) registers from 8 to 16

Memory translation



https://pdos.csail.mit.edu/6.828/2009/lec/x86_translation.pdf

Basic execution registers (1/3)

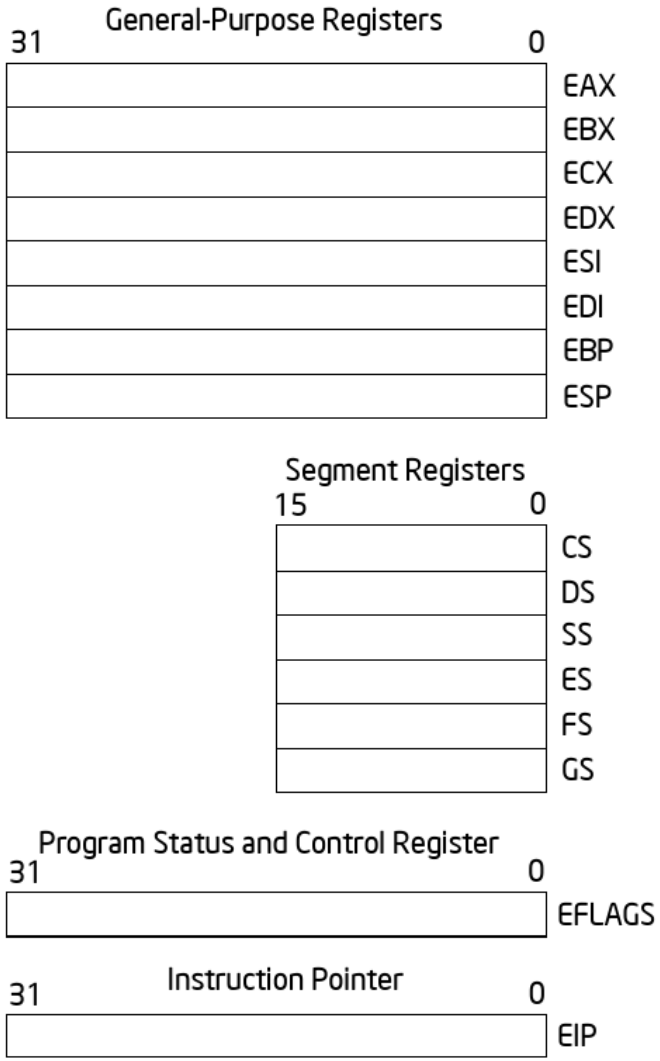


Modern OSes uses paging (only), to offer a 32/64-bit virtual flat address space

- in 32-bit mode:
  - segment registers hold 16-bit "useless" segment selectors
- in 64-bit mode:
  - `CS`, `DS`, `ES`, `SS` are treated as if each segment base is 0
  - all limit checks are disabled
- `FS`/`GS` can be used to point to TLS

  `wiki.osdev.org/Thread_Local_Storage`

Basic execution registers (2/3)



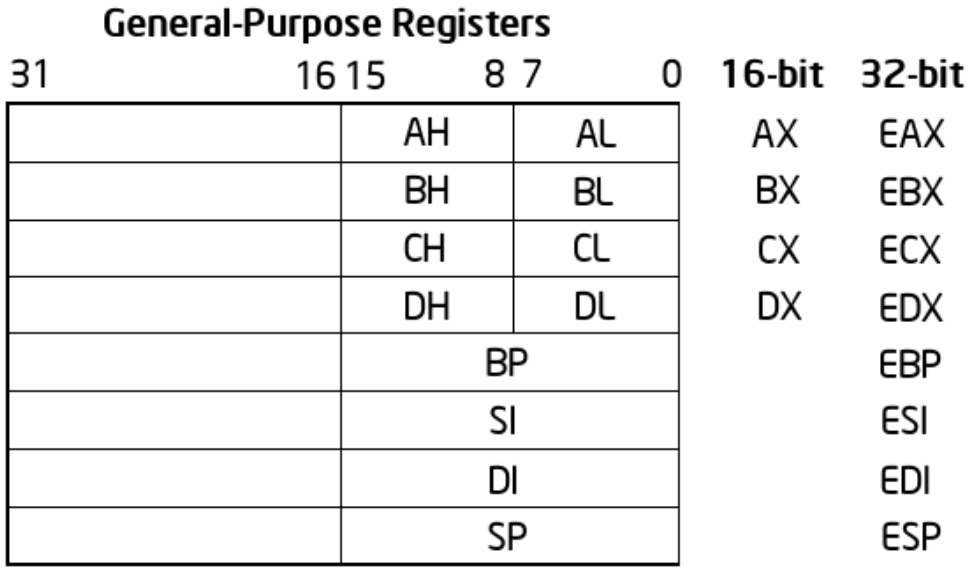
(Mostly) General purpose:

- `EAX` – accumulator
- `EBX` – pointer to data
- `ECX` – counter
- `EDX` – I/O pointer
- `ESI` – source pointer for string ops
- `EDI` – destination pointer

Execution registers:

- `EIP` – instruction pointer
- `ESP` – stack pointer
- (`EBP` – base pointer)
- `EFLAGS` – carry, sign, zero, parity, ...

Basic execution registers (2/3)



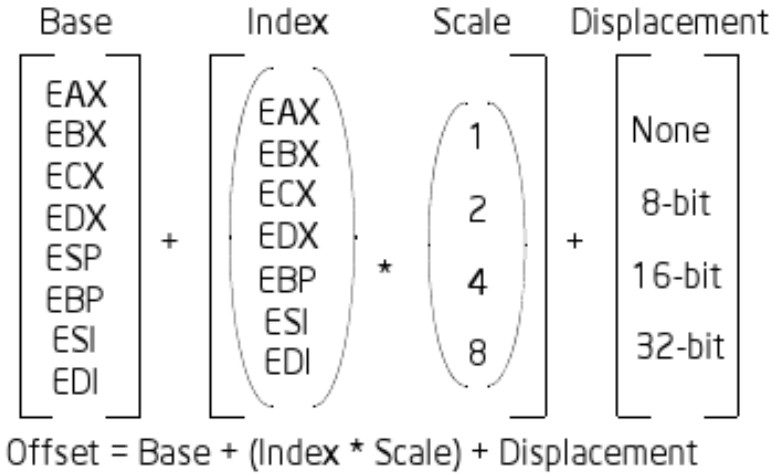
In 64 bit mode:

- `R{A,B,C,D}X`, `R{S,D}I`, `R{B,S,I}P` and `RFLAGS`
- `R8-R15[D,W,B]`

Operands

Machine-instructions act on zero or more operands
The data for an operand can be located in:

- the instruction itself, in case of an immediate operand
- a register
- a memory location, via (segment selector +) offset



- (an I/O port)

Two syntaxes: AT&T vs Intel

- Source-destination ordering
- Register naming: AT&T prefixes register names with %
- Immediate operands: AT&T prefixes immediate operands with $
- Operand size
  - in AT&T determined from the last character of the op-code name
  - Intel prefixes memory operands with *size* `ptr`
- Memory operands
  - in Intel syntax the base register is enclosed in `[` and `]`
  - in AT&T they change to `(` and `)`

moreover, Intel indirect memory references like `[base + index*scale + disp]`, changes to `disp(base, index, scale)` in AT&T

`http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s3`

AT&T vs Intel: examples

Intel Code	AT&T Code
`mov eax,1`	`movl $1,%eax`
`mov ebx,0ffh`	`movl $0xff,%ebx`
`int 80h`	`int $0x80`
`mov ebx, eax`	`movl %eax, %ebx`
`mov eax,[ecx]`	`movl (%ecx),%eax`
`mov eax,[ebx+3]`	`movl 3(%ebx),%eax`
`mov eax,[ebx+20h]`	`movl 0x20(%ebx),%eax`
`add eax,[ebx+ecx*2h]`	`addl (%ebx,%ecx,0x2),%eax`
`lea eax,[ebx+ecx]`	`leal (%ebx,%ecx),%eax`
`sub eax,[ebx+ecx*4h-20h]`	`subl -0x20(%ebx,%ecx,0x4),%eax`

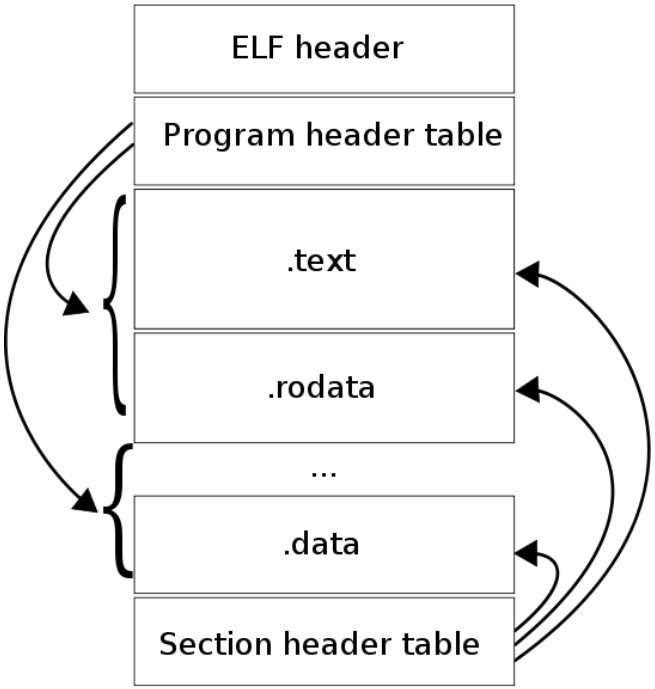
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s3

Executable and Linkable Format

Very flexible file format, can be used for

- executables
- dynamic libraries (AKA shared objects; in Windows: DLL)
- object files (AKA relocatable files)

Two views: segments and sections



https://commons.wikimedia.org/wiki/File:Elf-layout--en.svg

- ELF header at the beginning is a "road map"
- Program header, if present, tells how to create a process image
- Section header, if present, holds linking information

Section header

Sections *can* be present without their header

File format

References:

- `ELF(5)`
- `en.wikipedia.org/wiki/Executable_and_Linkable_Format`
- `refspecs.linuxbase.org/elf/elf.pdf`

Hint

Use `mmap` to ease the "parsing" of an ELF file

Data representation

Here:

- word $\rightarrow$ 32-bit object
- a null pointer has the value zero

Type	C	sizeof	Alignment (bytes)	Intel386 Architecture
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed word

	`unsigned int` `unsigned long`	4	4	unsigned word
Pointer	*any-type* `*` *any-type* `(*)()`	4	4	unsigned word
Floating-point	`float`	4	4	single-precision (IEEE)
	`double`	8	4	double-precision (IEEE)
	`long double`	12	4	extended-precision (IEEE)

Function calling (32 bit)

- stack always kept word-aligned
- argument words pushed in reverse order; i.e. C calling convention
  - argument size padded (if necessary) to keep word alignment
- EBP is the optional frame-pointer
- EBP, EBX, EDI, ESI, and ESP must be preserved for the caller
- integral and pointer return values are stored in EAX
- EBX is the GOT base register (for PIC code only)
- flag *direction* of EFLAGS must be zero on entry and upon exit
- ...

Standard stack frame

Position	Contents	Frame	
4*n*+8 (%ebp)	argument word *n*	Previous	*High addresses*
	...		
8 (%ebp)	argument word 0		
4 (%ebp)	return address	Current	
0 (%ebp)	previous %ebp (optional)		
-4 (%ebp)	unspecified		
	...		
0 (%esp)	variable size		*Low addresses*

Let's check sum.c

Exec-ing a program

The kernel cares about three types of program header entries only:

- `PT_LOAD`, areas of the new program's running memory
- `PT_INTERP`, which identifies the run-time linker
- `PT_GNU_STACK`, containing a bit indicating whether the stack should be made executable

To setup the (virtual) memory:

- stack typically moved downward by a random offset
  - stack layout $\rightarrow$ next slide
- all `PT_LOAD` segments are mapped
- special pages are mapped
  - the virtual Dynamic Shared Object (vDSO)
  - in `PER_SRV4` (non-s{u,g}id programs), an empty page is mapped at 0
  - ...

Source: `https://lwn.net/Articles/631631/`

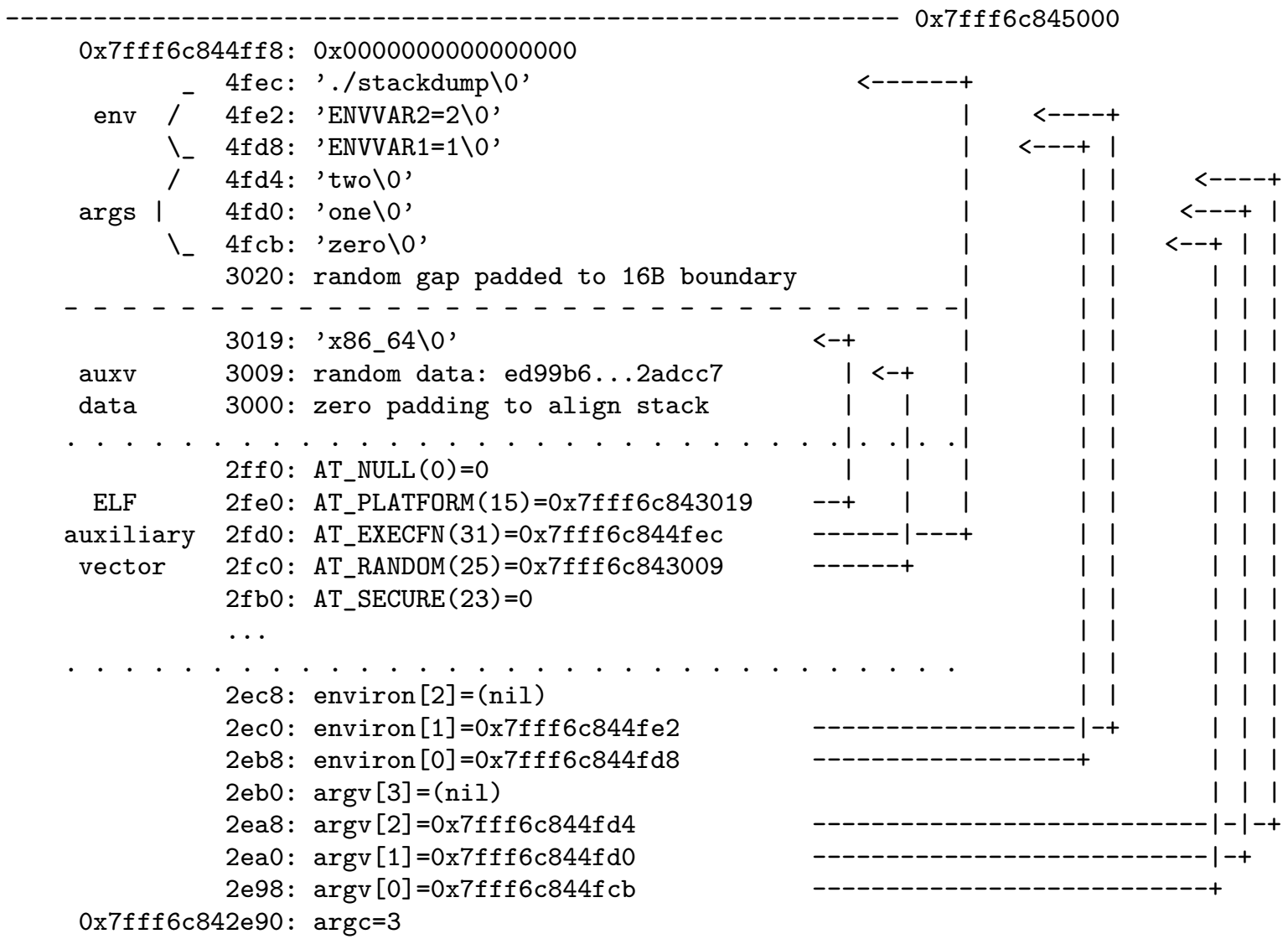
Stack Layout (32 and 64 bit)

From higher to lower addresses:

- NULL
- program name
- environment strings
- argv strings
- ELF Auxiliary Table
- NULL that ends envp[]
- environment pointers (at address β)
- NULL that ends argv[]
- argv pointers (at address α)
- argc

entry-point (indirectly) calls `main` passing: argc, argv ($=\alpha$), envp ($=\beta$)

Stack layout (at entry0), an example



Source: `https://lwn.net/Articles/631631/`

Syscall (32 bit)

Parameters are passed by setting:

- EAX = syscall #
  - syscall tables (x86 and x64 use different syscall-#): `https://w3challs.com/syscalls/`
- EBX, ECX, EDX, ESI, EDI, EBP = parameters 1 – 6

and issuing `INT 0x80`

On return, EAX contains the return value

Syscall example (32 bit)

```
.data
msg: .ascii "Hello World\n"

.text
.global main

main:
    movl $4, %eax   # use the write syscall
    movl $1, %ebx   # write to stdout
    movl $msg, %ecx # use string "Hello World"
    movl $12, %edx  # write 12 characters
    int $0x80       # make syscall

    movl $1, %eax   # use the exit syscall
    movl $0, %ebx   # error code 0
    int $0x80       # make syscall
```

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Data representation ("word" is not used anymore!)

Type	C	sizeof	Alignment (bytes)	AMD64 Architecture
Integral	`_Bool`†	1	1	boolean
	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed twobyte
	`unsigned short`	2	2	unsigned twobyte

	`int`
	`signed int`
	`enum`
		4	4	signed fourbyte

	`unsigned int`	4	4	unsigned fourbyte

	`long`
	`signed long`
	`long long`
	`signed long long`
		8	8	signed eightbyte

	`unsigned long`	8	8	unsigned eightbyte
	`unsigned long long`	8	8	unsigned eightbyte
	`__int128`†† `signed __int128`††	16 16	16 16	signed sixteenbyte signed sixteenbyte
	`unsigned __int128`††	16	16	unsigned sixteenbyte
Pointer	*`any-type`* `*` *`any-type`* `(*)()`	8	8	unsigned eightbyte
Floating-point	`float`	4	4	single (IEEE)
	`double`	8	8	double (IEEE)
	`long double`	16	16	80-bit extended (IEEE)
	`__float128`††	16	16	128-bit extended (IEEE)
Packed	`__m64`††	8	8	*MMX* and 3DNow!
	`__m128`††	16	16	SSE and SSE-2

† This type is called `bool` in C++.

†† These types are optional.

Standard stack frame

Position	Contents	Frame
`8n+16(%rbp)`	argument eightbyte n	Previous
	...	
`16(%rbp)`	argument eightbyte 0	
`8(%rbp)`	return address	Current
`0(%rbp)`	previous `%rbp` value	
`-8(%rbp)`	unspecified	
	...	
`0(%rsp)`	variable size	
`-128(%rsp)`	red zone	

Function calling (64 bit)

- arguments (of simple scalar types) are passed
  - first six: using registers: RDI, RSI, RDX, RCX, R8 and R9
  - the rest: using the stack
- RBP, RBX, R12-R15 and RSP must be preserved for the caller
  - Note: in 32-bit also EDI and ESI must be preserved
- the end of argument area shall be aligned on a 16 byte boundary
- the *red-zone*, a 128-byte area beyond the location pointed to by RSP, is considered to be reserved and shall not be modified by signal or interrupt handlers
  - compilers can optimize leaf-function frames

Check `sum64` out

And add other four arguments to `sum`

Syscall (64 bit)

Parameters are passed by setting:

- EAX = syscall #
  - syscall tables (x86 and x64 use different syscall-#): `https://w3challs.com/syscalls/`
- RDI, RSI, RDX, R10, R8, R9 = parameters 1 – 6

and issuing `syscall`

On return, EAX contains the return value

Syscall example (64 bit)

```
.data
msg: .ascii "Hello World\n"

.text
.global main

main:
    movq $1, %rax   # use the write syscall
    movq $1, %rdi   # write to stdout
    movq $msg, %rsi # use string "Hello World"
    movq $12, %rdx  # write 12 characters
    syscall         # make syscall

    movq $60, %rax  # use the exit syscall
    movq $0, %rdi   # error code 0
    syscall         # make syscall
```

https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux

Outline

Compilation and linking process

- Default-happy version
  `gcc *.c # i.e.  main.c func[23].c`
- What is really happening?
  `gcc -v *.c`
  - For a simpler output:
    `gcc -v *.c 2>&1|egrep '(cpp|cc1|as|collect2) '`
    - collect2 is a GCC internal utility; calls `ld`
      `https://gcc.gnu.org/onlinedocs/gccint/Collect2.html`
- Compilation: `*.c + *.h` $\xrightarrow{\texttt{(cpp)+cc1}}$ `*.s` $\xrightarrow{\texttt{as}}$ `*.o`
  - Object files = m.c. + relocation entries (+ debugging/other info)
- Linking: `*.o (+ *.a + *.so)` $\xrightarrow{\texttt{ld}}$ `a.out`
  - Symbol resolution
  - Relocation

Running a program

- Running: *executable* $\overset{\texttt{fork+exec}}{\longrightarrow}$ running process
- Dynamically linked $\Rightarrow$ an "interpreter": the dynamic linker
  - specified inside `PT_INTERP` segment (`.interp` section)
- `ld.so(8)`, actually
  - `ld.so` handles *a.out binaries*
  - `ld-linux.so*` handles *ELF*
    - `/lib/ld-linux.so.1` for libc5
    - `/lib/ld-linux.so.2` for glibc2, which has been used for years

  same behavior, and same support files; for details: `ldconfig(8)`

Two linkers

- Compile-time linking: ld
- Run-time linking: ld-linux.so*
  - Chicken and egg situation: the linker itself is a shared object!

Assembly files (1/2)

- Option `-S` makes gcc stop after compilation

  `gcc -S -fno-asynchronous-unwind-tables -masm=intel *.c`
- `func3`:
  - `puts` used without "declaration" → undefined symbol
  - `.glob`: `y` and `glob_func3` → globally exported symbol
  - sections: `.data`, `.rodata`, `.text` (and `.note.GNU-stack`); no `.bss`
- `func2`:
  - undefined: `z`, `printf` and `glob_func3`
  - exported: `glob_fun2`
  - `.comm` = "`.section bss` + `.glob`"
    - `https://sourceware.org/binutils/docs/as/Comm.html`
    - `https://sourceware.org/binutils/docs/as/Local.html`
  - BSS: `x2` and `y`
  - (file) local symbols:
    - (implicitly) `x1` and `stat_func2`
    - because `.local`: `x2`
  - . . . (next slide)

Assembly files (2/2)

- `func2`:
  - no code difference for
    - calling (static/extern) functions
    - accessing (static/global/extern) variables
- `main`:
  - no code difference for
    - calling (static/global/extern) functions
    - accessing (static/global) variables

At this level, (most) addresses are still expressed as names

Object Files

- `objdump --file-header *.o`; start address is 0
- let's check some code:
  `objdump --disassemble -M intel func3.o`
  a lot of 00 around ☺
  `objdump --reloc func3.o`
- in `func2.o` it is interesting to compare the call to `stat_func2` with the others
  - We'll talk later about compile/link/runtime library interposition
- let's merge `func2.o` and `func3.o` and check the result:
  `ld -o test1.o --relocatable func2.o func3.o`
  `ld -o test2.o -r func3.o func2.o # -r == --relocatable`

Libraries

- Static libraries
  - Collection of `.o`, created with `ar`
    - `gcc -c func[23].c`
    - `ar rcs libfuncs.a func[23].o`; then
    - `gcc main.c ./libfuncs.a`, or
      `gcc main.c -L. -lfuncs`
    - `./a.out`
  - Only object files containing referenced symbols are included
    - order of `*.o` and `*.a` matters
    - when processing a library, linker finds a "fixed point"
- Dynamic libraries
  - ELF shared objects, created by using `-shared`
    - `gcc -shared -fPIC -o libfuncs.so func[23].c`
    - `gcc main.c -L. -lfuncs`
    - `LD_LIBRARY_PATH=.  ./a.out`

Dynamically link by default

Unless `-static` is specified; for instance, in both cases `libc.so` has been dynamically linked to `a.out`

Position (in)dependent code

- code within an executable is typically absolute/position-dependent (more efficient than PIC), and tied to a fixed address in memory
- shared objects are typically loaded at different addresses in different processes
  - with position-dependent code, text segment would require modifications at load-time $\Rightarrow$ private copy for the process that cannot be shared
  - when built from PIC, relocatable references are generated as indirections through data in the shared object's data segment: text segment requires no modification entries within the code segment

On x86-64 shared objects "must" be PIC

In the default code model (`-mcmode=small`) symbols must be within 4GB of each other, so such code cannot be relocated to a 64-bit space

PIC/PIE

GCC options

- `-fpic`/`-fPIC` generate position-independent code, which accesses all constant addresses through a global offset table
- `-fpie`/`-fPIE` similar to `-fpic`/`-fPIC`, but generated position independent code can be only linked into executables; typically used with `-pie`

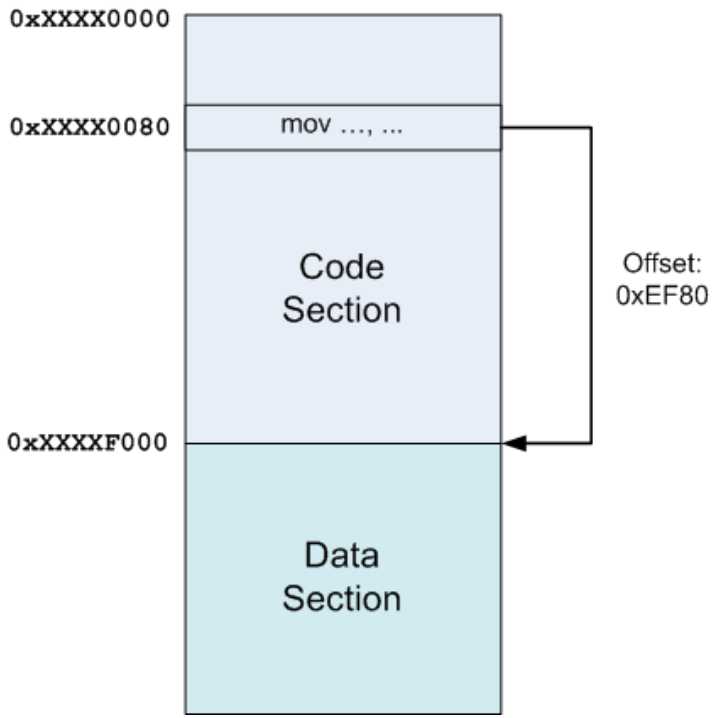
Be consisted (for predictable results)

`-f...` are for the compiler, `-pie`/`-shared`/`-static` for the linker

- `-static` static linking; shared libraries are ignored
- `-shared -fpic` produce a shared object
- `-pie -fpie` produce an *"executable" shared object* $\rightarrow$ better ASLR
- *(only in recent versions)* `-static-pie`
  `https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html`

PIC: key insights (1/4)

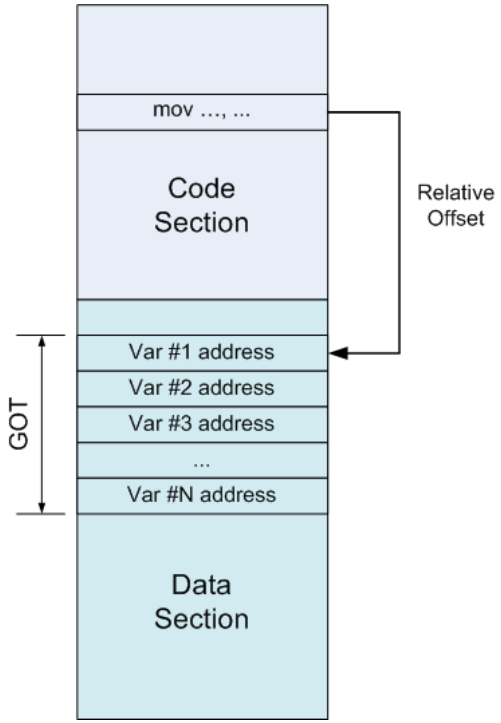
1. offsets between `text` and `data` sections statically known
2. we can use IP-relative offsets to access (statically linked) data
   - requires thunking on 32 bits; x64 can use RIP-relative offsets
     → `pic-test.c`



`https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/`

PIC: key insights (2/4)

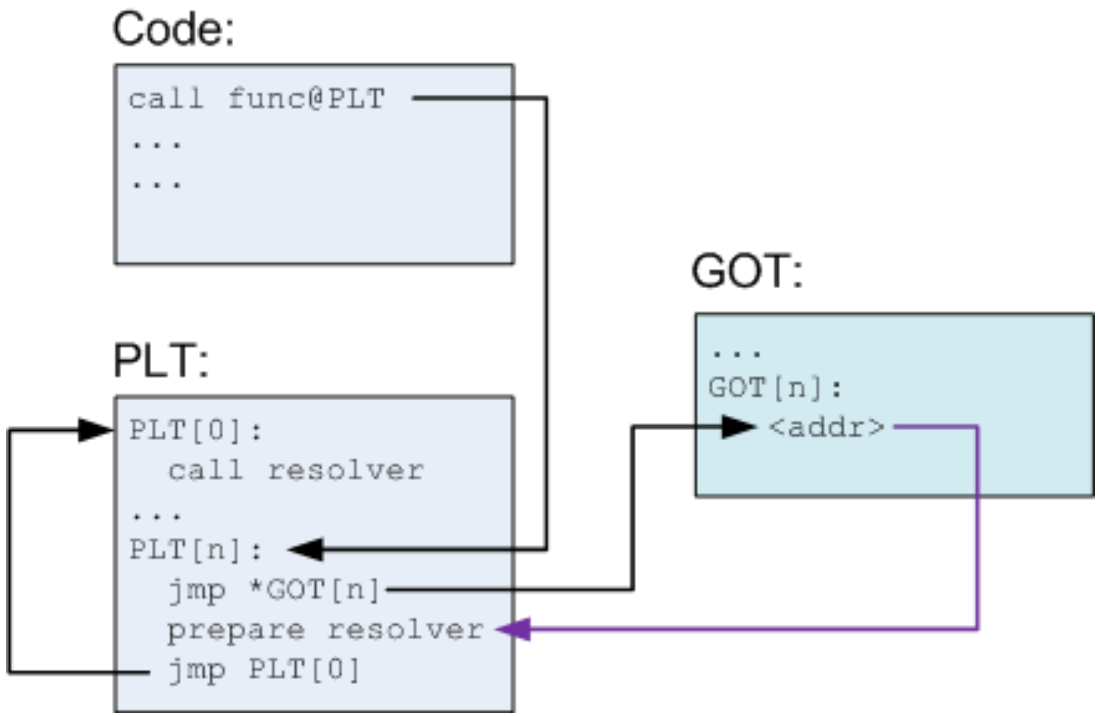
3. we use an indirection, through the GOT – Global Offset Table, for dynamically linking external references
   - GOT resides in the `data` section and the (static) linker generates (dynamic) relocation entries for it
   - one relocation entry for each variable v, regardless the number of times v is accessed



`https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/`

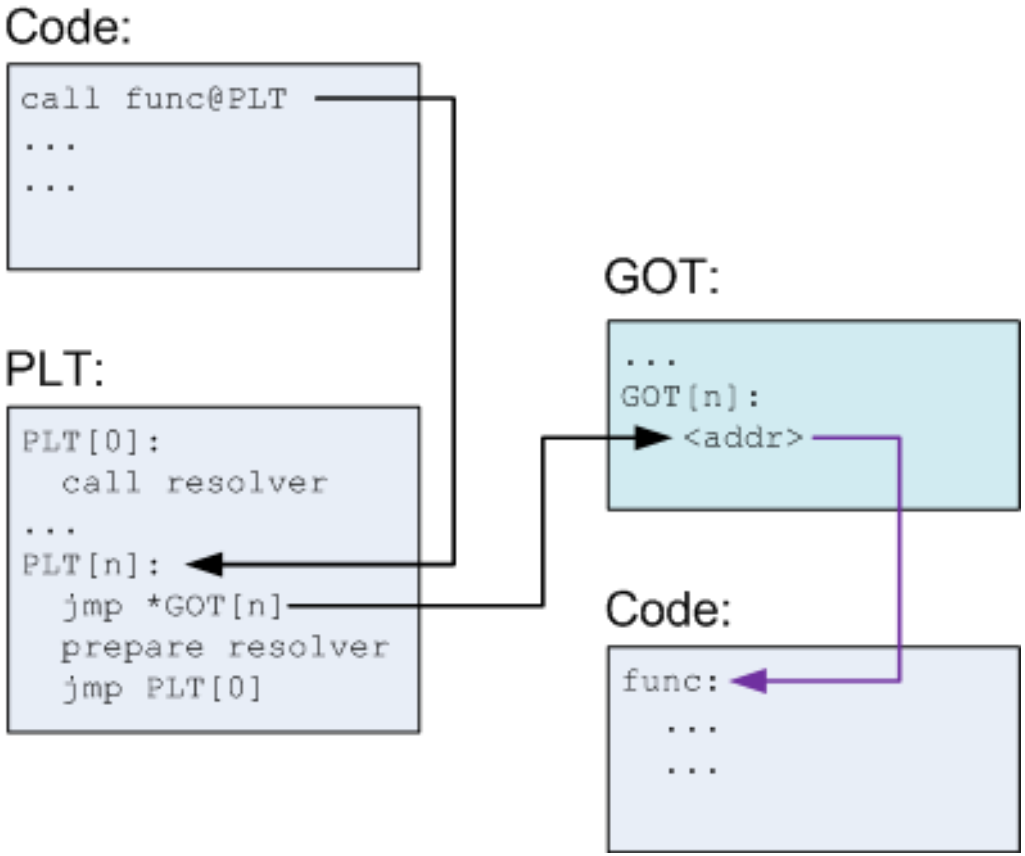
PIC: key insights (3/4)

4. to allow lazy binding, calls to dynamic symbols are translated into calls into the PLT – Procedure Linkage Table
   - PLT = array of stubs that call the "real" functions using GOT
   - only functions can be lazily bound, variables are always eagerly bound
5. first entries of PLT/GOT are "special"; PLT[0]/GOT[0] is the resolver



PIC: key insights (4/4)

When the symbol has been resolved:



- `https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/`
- more details (and exploitation): [DFCS+15]

Exploiting GOT/PLT

GOT is an interesting data-structure, which *needs* to be writable

- unless full RELRO is activated ($\rightarrow$ next-slide)

Food for thought:

- GOT overwrite = calling `system` when you want to call `printf` ☺
- leaking the address of, say `puts`, may help to find the address of `system`

RELocationReadOnly

RELRO is a memory corruption mitigation
Related `ld` options:

- `-z norelro` (default) – don't create `PT_GNU_RELRO`
- `-z relro` create `PT_GNU_RELRO` segment header
- `-z now` – tell the dynamic linker to resolve all symbols when the program is started, or when the shared object is linked to using `dlopen`

Full relro in gcc: `-Wl,-z,relro,-z,now`

To see the mapping section → segment

```
readelf --wide --program-headers
```

Library interposition

Linux linkers support library interpositioning

- i.e. to intercept calls to library functions and execute your own code
- basic idea: calls to a *target function* are replaced with calls to a wrapper function, with the same signature

Three kinds:

- compile-time: using macros of C preprocessor. . . boring ☺
- link-time: using `--wrap` flag of `ld` (typically through `-Wl,--wrap,`*func-name* from `gcc`)
  - any undefined reference to *symbol* will be resolved to `__wrap_`*symbol*. Any undefined reference to `__real_`*symbol* will be resolved to *symbol*
- run-time, using the linker API ($\rightarrow$ next slide)
  - like `ltrace`, but we can *change/fake* values

Dynamic Linker API

```
void *dlopen(const char *filename, int flags);
```

- `LD_LIBRARY_PATH` may contain a colon-separated list of directories to search for libraries, before checking `/lib` and `/usr/lib`
  - as a security measure, ignored for set-user/group-ID programs

```
void *dlsym(void *handle, const char *symbol);
```

two special pseudo-handles:

- `RTLD_DEFAULT`: find the first occurrence of the desired symbol using the default shared object search order
- `RTLD_NEXT`: find the next occurrence of the desired symbol in the search order after the current object. This allows to provide a wrapper around a function in another shared object

To intercept, say `strcmp`, from bash: `LD_PRELOAD=./my_strcmp.so ...`

Outline

Process tracing

ptrace

`ptrace(2)` provides a means by which one process (the *tracer*) may observe and control the execution of another process (the *tracee*), and examine and change the tracee's memory and registers.

Used by `strace`, debuggers, ...
While being traced

- tracee will stop each time a signal (except for SIGKILL) is delivered
- the tracer will be notified at its next call to waitpid(2)
- while tracee is stopped, tracer can inspect and modify the tracee
- tracer then causes tracee to continue, optionally ignoring the delivered signal (or delivering a different signal)

Breakpoints

On x86 (from Intel's documentation):

> *The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code).*

in the *tracee* executing an `INT 3` raises a signal `SIGTRAP`, which yields the control to the *tracer*

- `https://eli.thegreenplace.net/2011/01/23/how-debuggers-work-part-1`
- `https://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints`
- `https://eli.thegreenplace.net/2011/02/07/how-debuggers-work-part-3-debugging-information`

Poor's man anti-debugging technique

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/ptrace.h>

int main()
{
        if (ptrace(PTRACE_TRACEME, 0, 0, 0) == -1) {
                printf("Hello world!\n");
                return EXIT_SUCCESS;
        }
        /* evil behaviour */
        printf("I'm evil!!!\n");
}
```

More info:

https://seblau.github.io/posts/linux-anti-debugging

References

[DFCS+15] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.
How the elf ruined christmas.
In *USENIX Security Symposium*, pages 643–658, 2015.