CS527 Software Security

Reverse Engineering

Mathias Payer

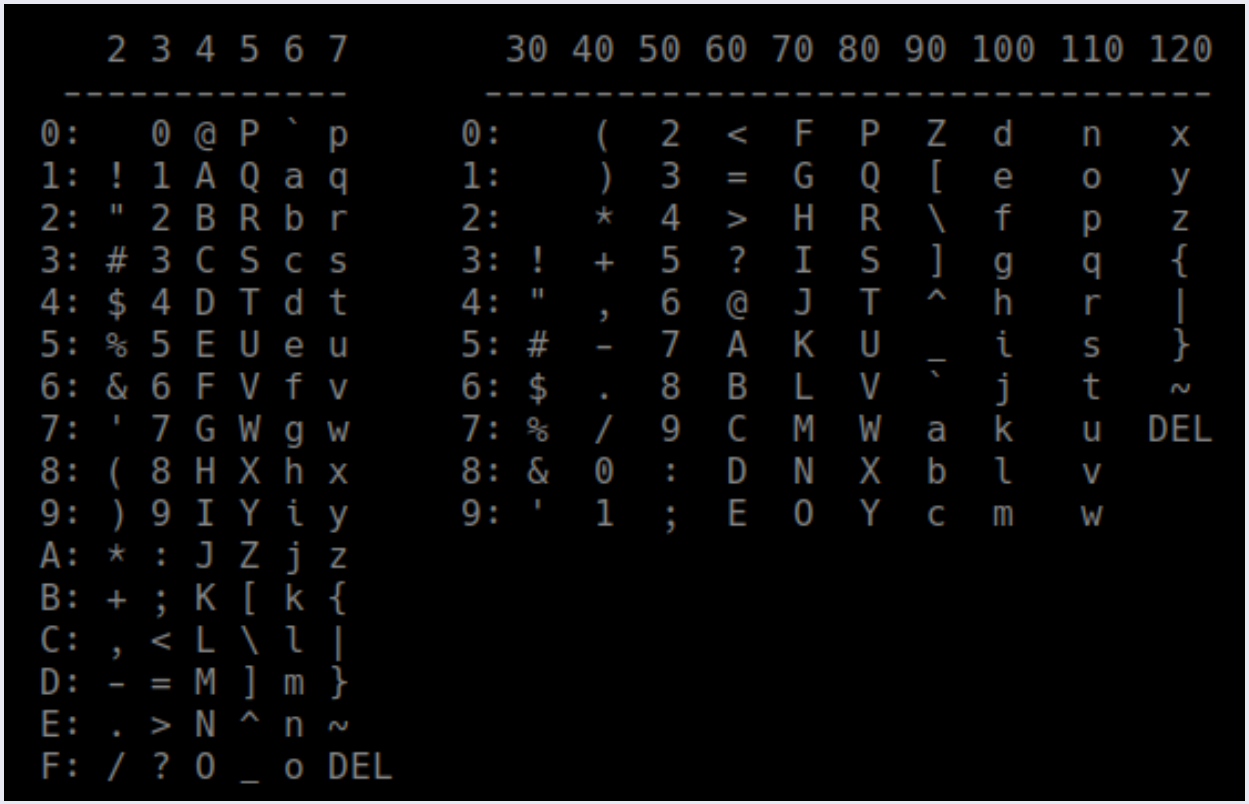
Purdue University, Spring 2018

Basics: encodings

- Code is data is code is data
- Learn to read hex numbers: `0x38 == 0011'1000`
- Python: `hex(int('00111000', 2))`
- Remember common ASCII characters and instructions

Basics: characters

- ASCII - American Standard Code for Information Interchange
- 1 byte per character (7bit, extended to 8)
- How to go from upper to lower? What are numbers?



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

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

Figure 1

Endianness

- To break the egg at the big or at the small end?
  - See Gulliver's travels for the literatal answer

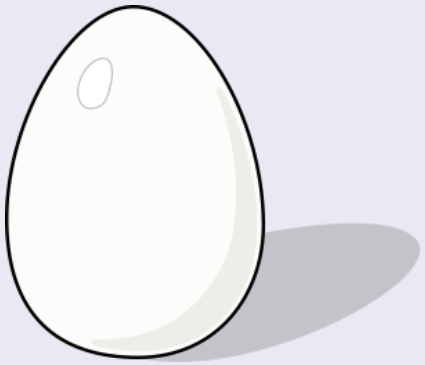


Figure 2

Endianness (CS)

How do we store multi-byte integers, such as 0x12345678 in memory?

Byte	00	01	02	03
Big endian	12	34	56	78
Little endian	78	56	34	12
Middle endian	56	34	78	12

Little endian architectures: x86, ARM.
Big endian architectures: MIPS, RISC.

Compilation: C source

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  if (argc == 2)
    printf("Hello %s\n", argv[1]);
  return 0;
}
// gcc -W -Wall -Wextra -Wpedantic -O3 -S hello.c
```

Generated code

```
  .file "foo.c"
  .section  .rodata.str1.1,"aMS",@progbits,1
.LC0:
  .string "Hello %s\n"
  .section  .text.unlikely,"ax",@progbits
.LCOLDB1:
  .section  .text.startup,"ax",@progbits
.LHOTB1:
  .p2align 4,,15
  .globl  main
  .type main, @function
main:
.LFB0:
  .cfi_startproc
  cmpl  $2, %edi
  je  .L6
  xorl  %eax, %eax
  ret
```

Assembly instructions

Instructions are encoded as bytes in memory: `code == data`.
Some architectures require that pages are mapped executable to execute them.

- Different architectures map bytes to instruction differently.
- Common architectures: x86, ARM, MIPS

Assembly mnemonics

Decoding machine code into assembly code makes code *readable*.

- AT&T syntax: `mov src, dst` (gcc, gdb)
- Intel syntax: `mov dst, src` (radare2, IDA)

Pick and choose your favorite, get comfortable with either.

Data storage

Data can be stored in

- Registers: fast, directly accessible
- Memory: load and store to registers
- Disk/network: slow access through operating system

Registers

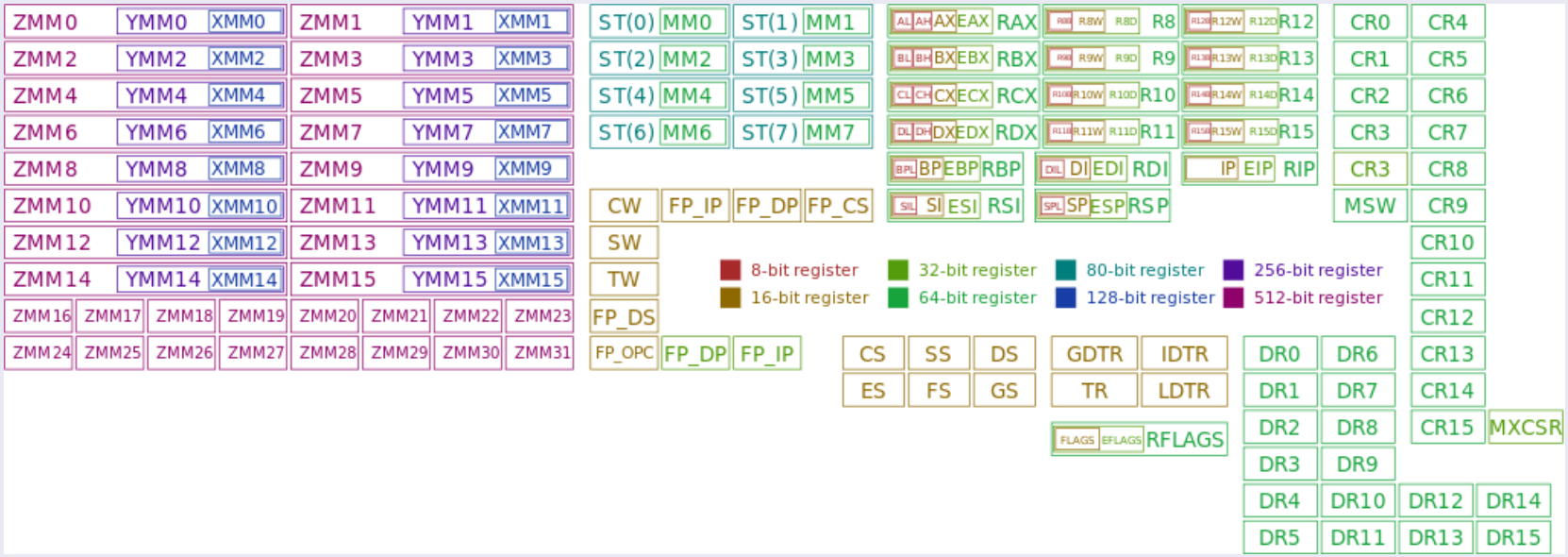


Figure 3

Data movement

- Constant to register: `movq $0x20, %rdx (rdx = 0x20;)`
- Register to register: `movl %ebx, %ecx (ecx = ebx;)`
- Memory to register: `movq -0x4(%rdi, %rdx, $0x4), %rbx (rbx = *(rdi+rdx*0x4 - 0x4);)`

Arithmetic

- `addl %eax, %ebx (ebx += eax;)`
- `addl $0x23, (%rbx) (*rbx += 0x23;)`
- Many fun instructions, check out floating point

Control flow

- Unconditional: `jmp target` (direct/indirect)
- Function call: `call target` (direct/indirect)
- Update flag register:
  - `cmp t1, t2` (`sub t1, t2`)
  - `test t1 t2` (`and t1, t2`)
  - Updates flag, result is discarded
- Conditional jump: `jcond target` (direct)

x86 caveat

x86 is an incredibly complex and rich instruction set.
Rely on the Intel Instruction Manual, the AMD Instruction Set Description, or one of the online resources.
Reference `http://ref.x86asm.net/`

x86 instruction decoding

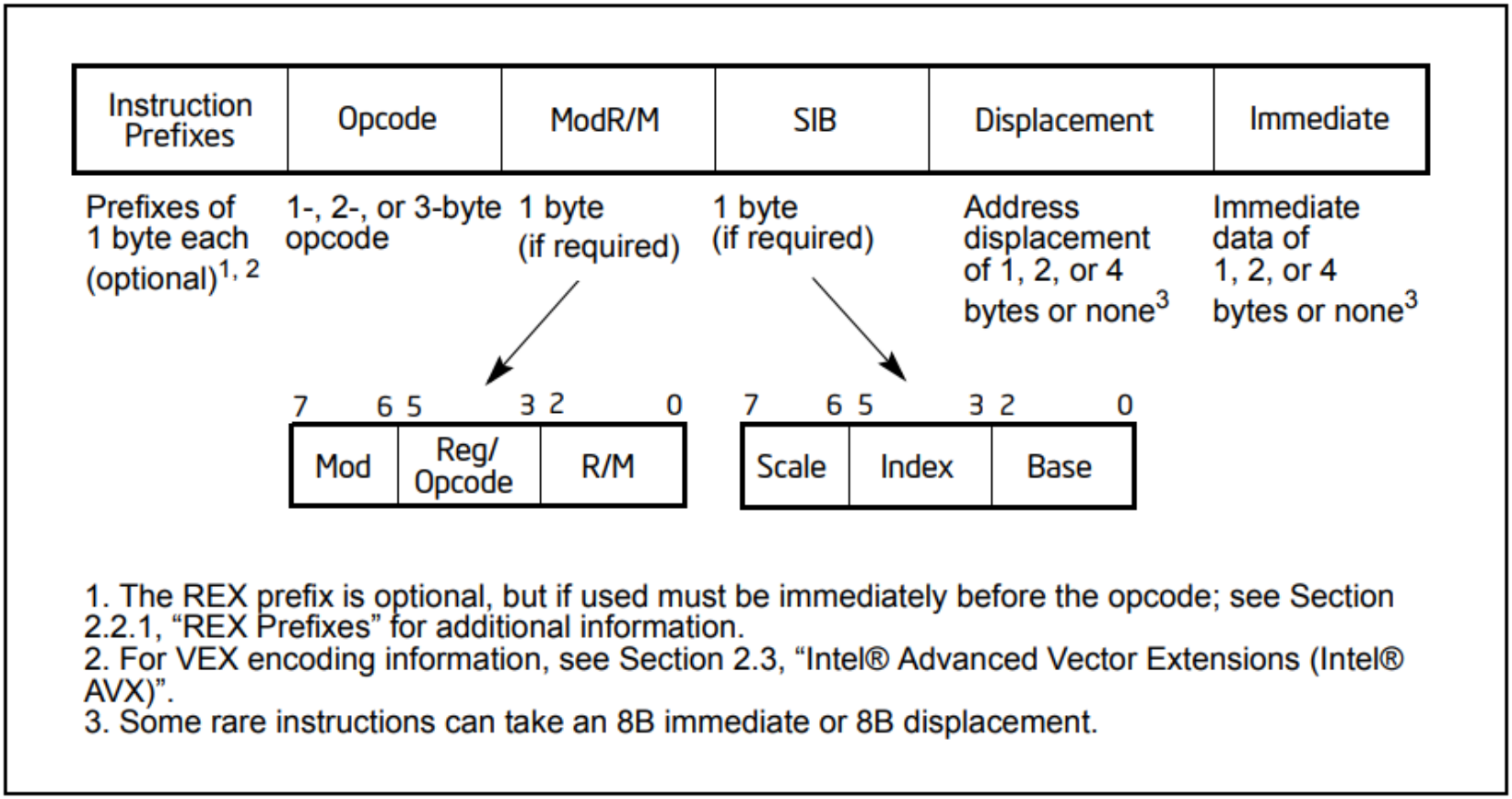


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Figure 4

Process address space

- Programs get "full" virtual address space
  - 32, 48, or 64 bit
- Where to place
  - program,
  - libraries,
  - global data,
  - heap,
  - stack(s)?

Process address space

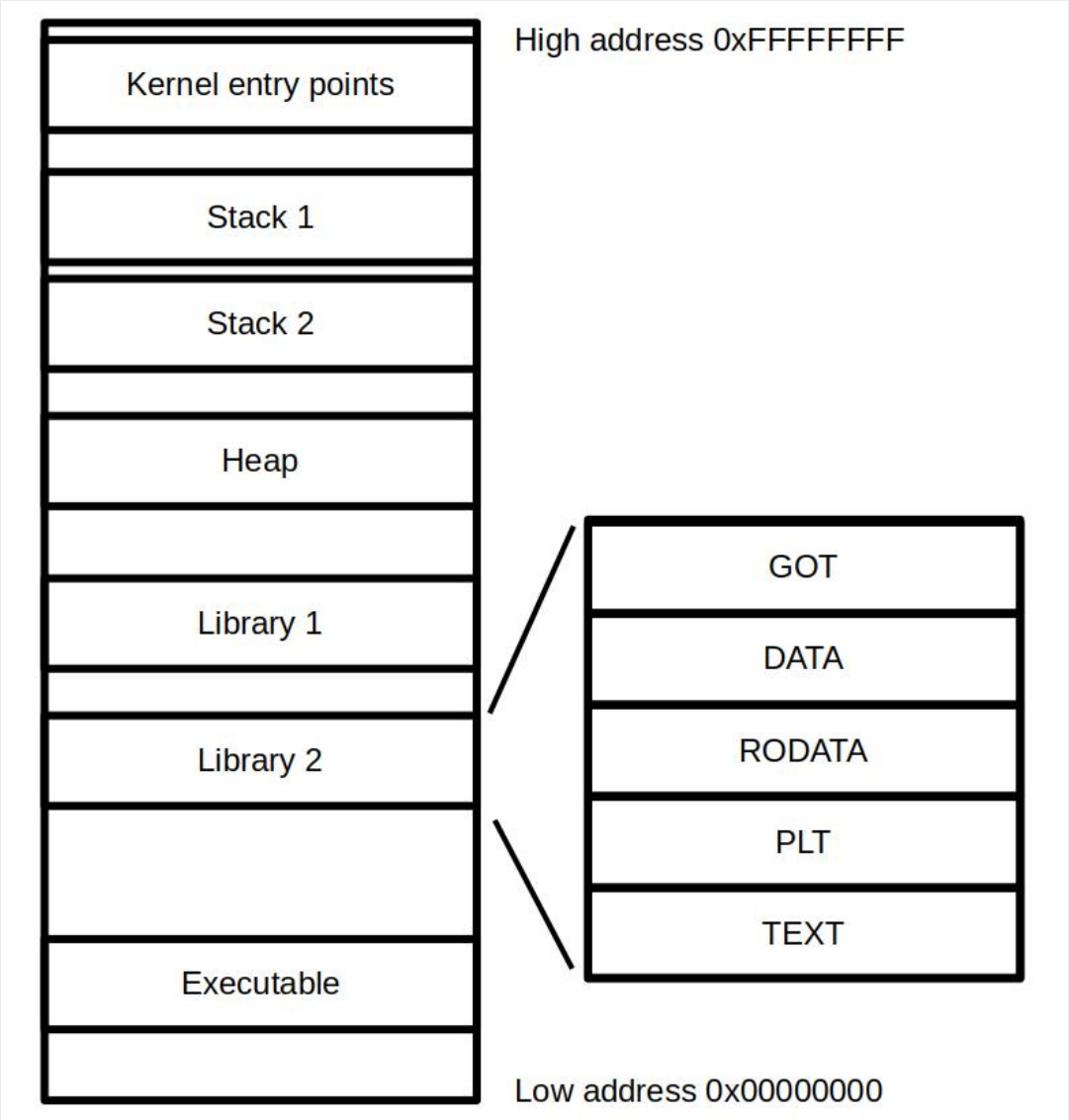


Figure 5

Process address space

Questions to ponder:

- What are the permissions of the individual sections?
- What are the requirements for placing code/data?
- How much flexibility is there?

The loader

Programs are either statically linked (self-contained) or use a dynamic loader for bootstrapping.

- Loader is the first program to run
- Loads and relocates program
- Loads and relocates all libraries
- Resolves all references
- Stiches programs together
- Call initialization functions
- Handles control to program
- Programs call into libc to initialize

Linking

```
0000400470 <main>:
70: 83 ff 02                    cmp    $0x2,%edi
73: 74 03                       je     400478 <main+0x8>
75: 31 c0                       xor    %eax,%eax
77: c3                          retq
78: 50                          push   %rax
79: 48 8b 56 08                 mov    0x8(%rsi),%rdx
7d: 40 b7 01                    mov    $0x1,%dil
80: be 04 06 40 00              mov    $0x400604,%esi
85: 31 c0                       xor    %eax,%eax
87: e8 d4 ff ff ff              callq  400460
                              <__printf_chk@plt>
8c: 31 c0                       xor    %eax,%eax
8e: 5a                          pop    %rdx
8f: c3                          retq
```

What about all the other code in `objdump -d a.out`?

Start files

```
0000000000400470 <main>: ...
0000000000400490 <_start>:
90: 31 ed                       xor    %ebp,%ebp
92: 49 89 d1                    mov    %rdx,%r9
95: 5e                          pop    %rsi
96: 48 89 e2                    mov    %rsp,%rdx
99: 48 83 e4 f0                 and
                              $0xfffffffffffffff0,%rsp
9d: 50                          push   %rax
9e: 54                          push   %rsp
9f: 49 c7 c0 f0 05 40 00        mov    $0x4005f0,%r8
a6: 48 c7 c1 80 05 40 00        mov    $0x400580,%rcx
ad: 48 c7 c7 70 04 40 00        mov    $0x400470,%rdi
b4: e8 87 ff ff ff              callq  400440
                              <__libc_start_main@plt>
b9: f4                          hlt
ba: 66 0f 1f 44 00 00           nopw   0x0(%rax,%rax,1)
```

ELF format

- ELF allows two interpretations of each file: sections and segments
- Segments contain permissions and mapped regions. Sections enable linking and relocation
- OS checks/reads the ELF header and maps individual segments into a new virtual address space, resolves relocations, then starts executing from the start address
- If `.interp` section is present, the interpreter loads the executable (and resolves relocations)
- More: `http://www.skyfree.org/linux/references/ELF_Format.pdf`

ELF format

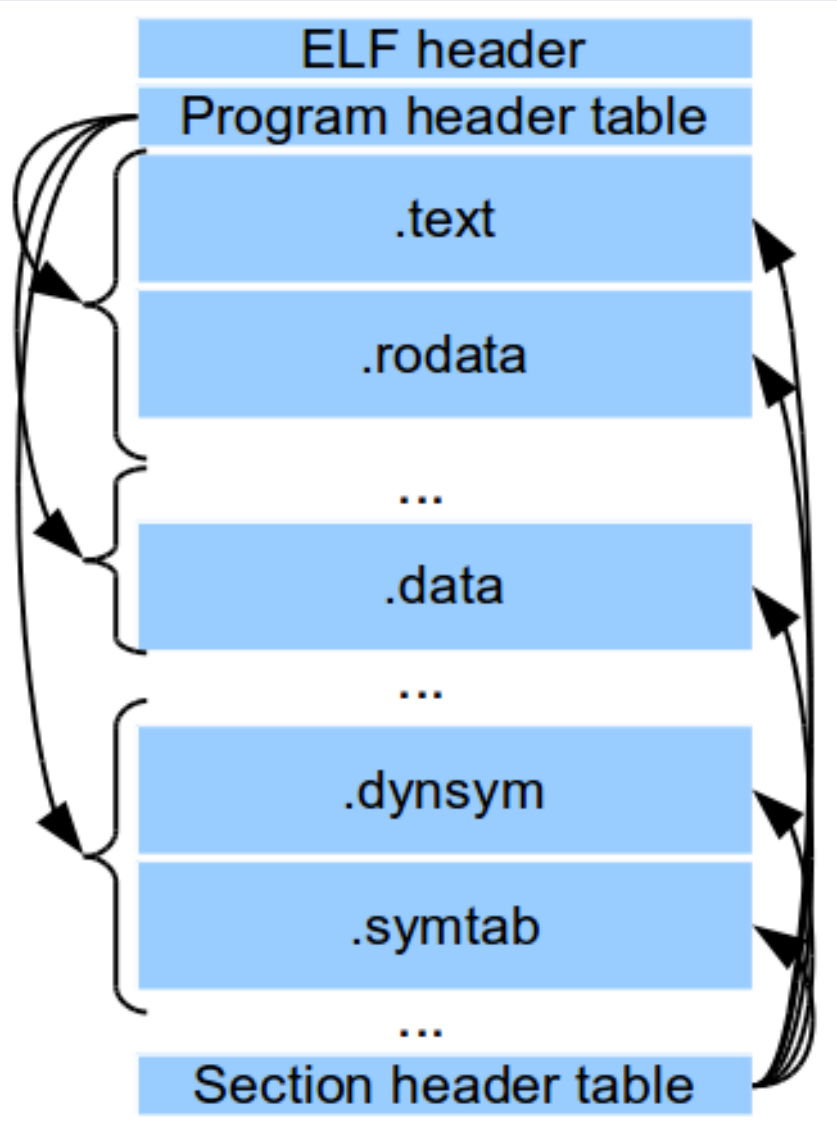


Figure 6

ELF tools

- `readelf` and `objdump` to display information
- `readelf -h a.out` for basic information
- `readelf -l a.out` program headers
- `readelf -S a.out` sections to relocate executable

Stack and heap layout

- Loader maps runtime sections of shared objects into virtual address space
- Loader calls global init functions
- libc initializes heap through `sbrk`, enables malloc

Stack and heap

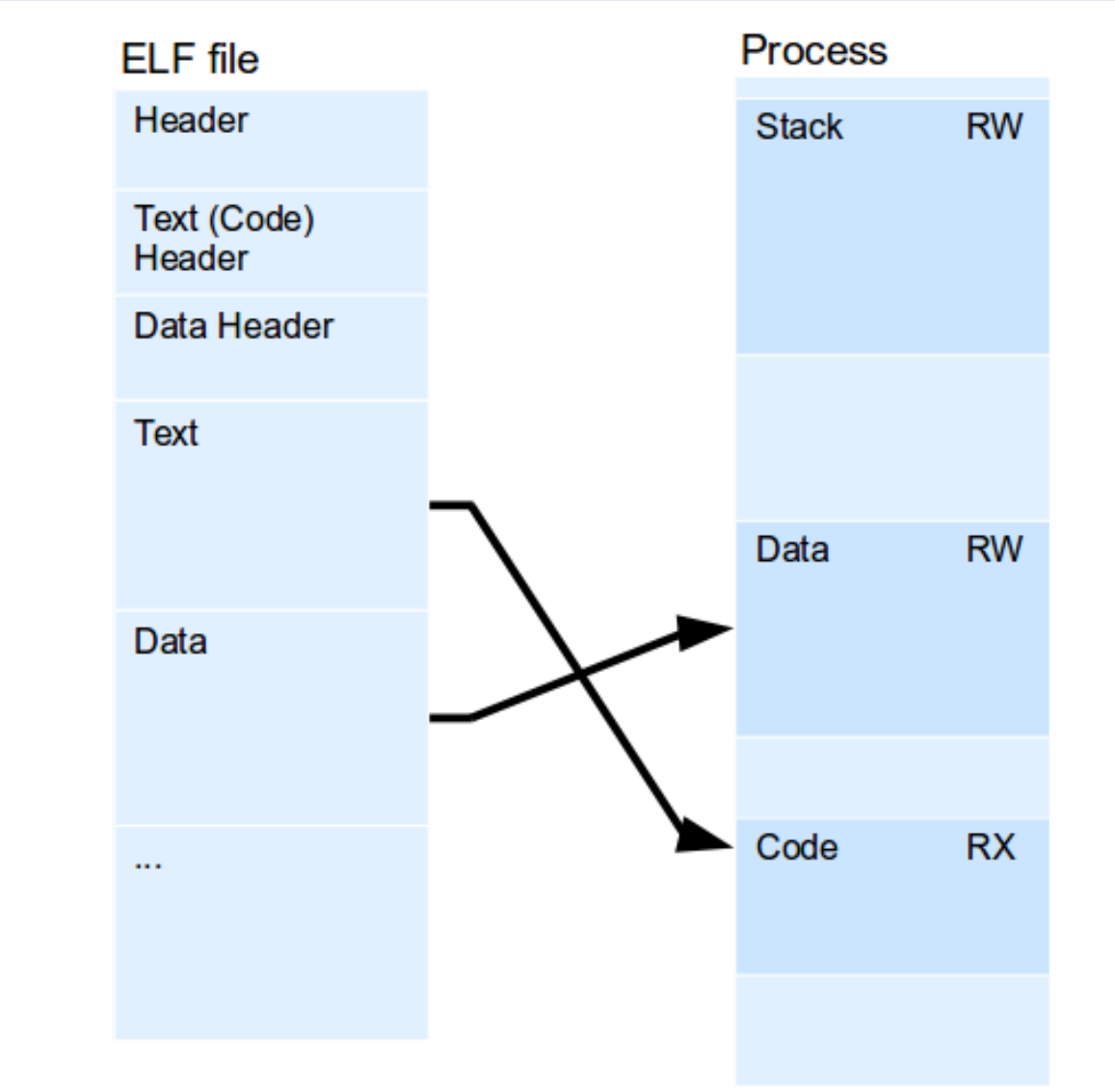


Figure 7

Stack layout

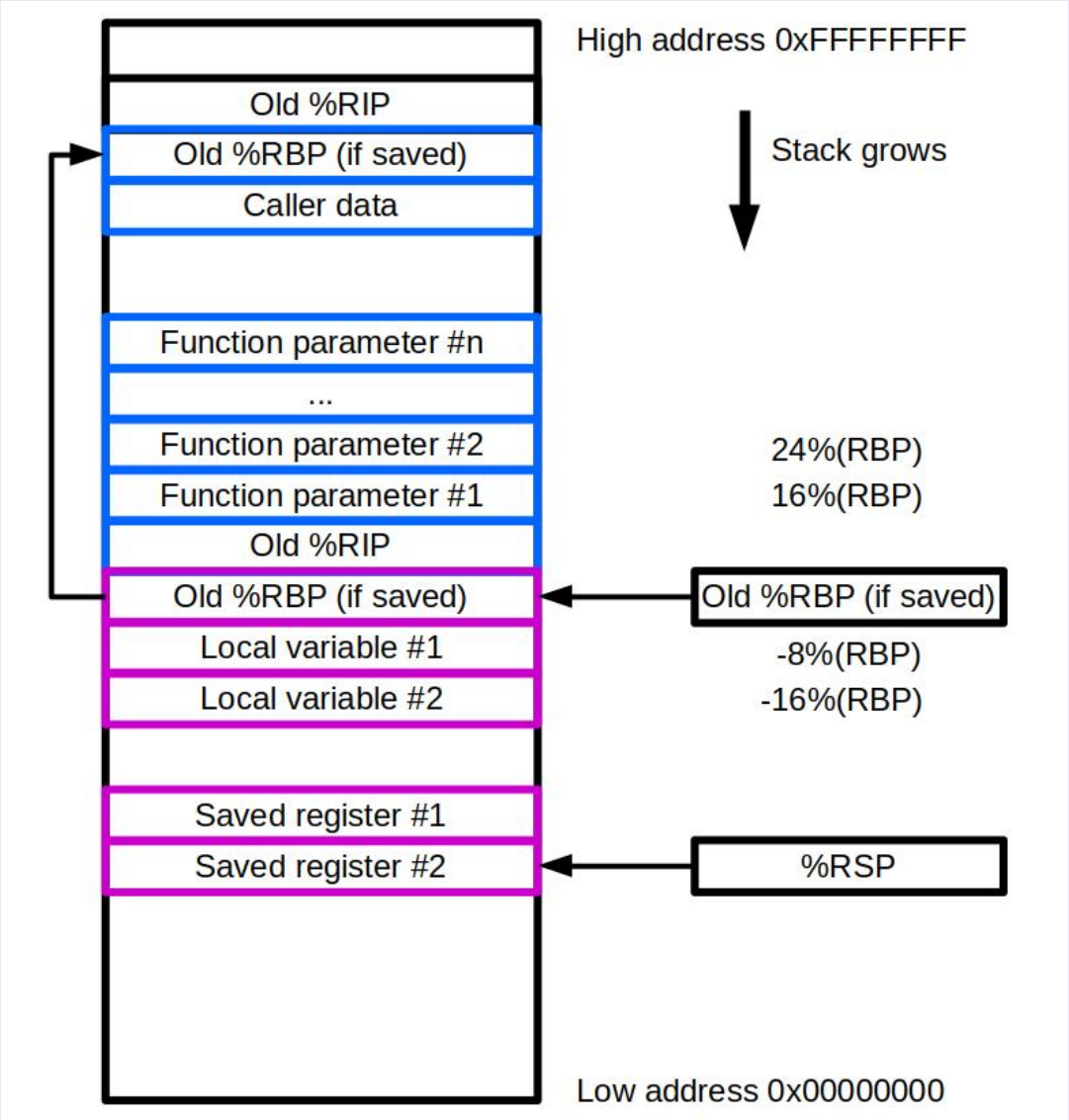


Figure 8

Calling convention

- How are arguments passed between functions
- In which order are arguments passed
    - Left to right or right to left?
- Register ownership (caller or callee saved)
- Registers used to pass arguments
- Handling of variadic functions, pass by value, corner cases

Calling convetion examples

- x86, cdecl: right to left
- x64, cdecl: right to left, plus rdi, rsi, rdx, rcx, r8, r9 for first 6 arguments

Shared libraries

- Global Offset Table contains pointers to symbols in other shared objects
- Procedure Linkage Table contains code that transfers control through the GOT to a symbol in another shared object
- The entries in the GOT that point to functions are initialized with the loader's address to resolve it on-the-fly

Shared libraries

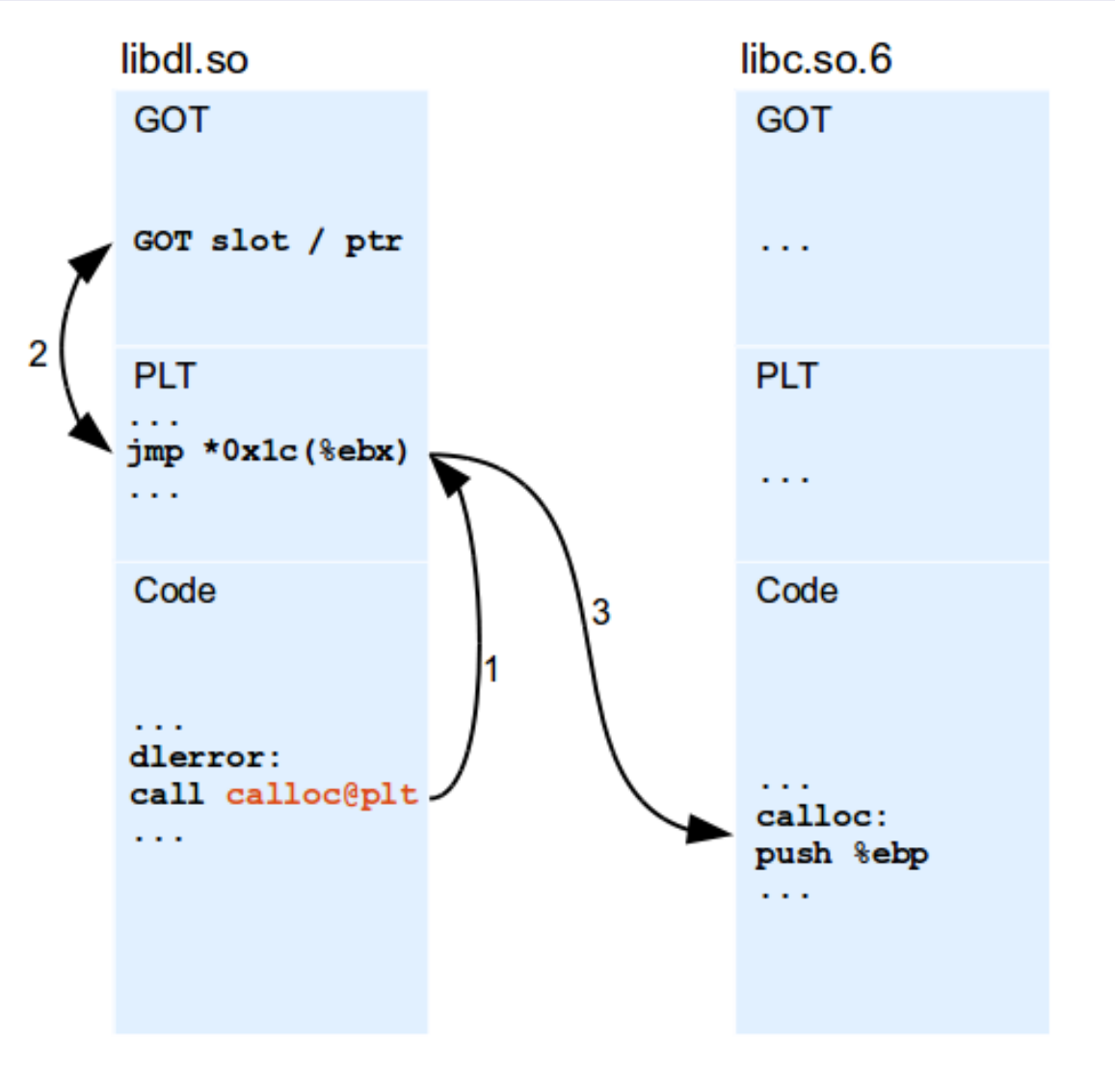


Figure 9

Interaction with the Operating System

- Processes interact with the operating system through system calls
- . . . or faults (such as protection fault, segmentation fault, or FP exception)

System calls

- `int 0x80`: x86, old way, Linux specific
- `sysenter`: x86, only saves subset of state, requires "call gate"
- `syscall`: x64 way
- `int 0x21`: x86, DOS
- `int 3`: debug interrupt
- `svc`: "supervisor call", ARM way

System calling convention

Special calling convention, pack all request data into registers.
Information is Linux specific.

- `rax/eax` contains system call number
- Parameters are passed in registers
  - x86: `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`
  - x64: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
  - More than 6 arguments: pass on stack

Assembly example

```
  .section  .rodata
.LC0:
  .string "Hello world"
  .text
  .globl  main
main:
  pushq %rbp
  movq  %rsp, %rbp
  leaq  .LC0(%rip), %rdi
  call  puts@PLT
  movl  $0, %eax
  popq  %rbp
  ret
```

Run: `gcc hello.s`

System call example

```
.text
.global _start
_start: movl    $len,%edx   # 3: message length.
        movl    $msg,%ecx   # 2: pointer to message.
        movl    $1,%ebx     # 1: file handle (stdout).
        movl    $4,%eax     # syscall nr: sys_write.
        int     $0x80
        movl    $0,%ebx     # 1: exit code.
        movl    $1,%eax     # syscall nr: sys_exit
        int     $0x80
.data
msg:    .ascii  "Hello, world!\n"
        len = . - msg       # length
```

Linking and running

```
as hello.s -o hello.o
ld -s -o hello hello.o
./hello
```

More details:
https://web.archive.org/web/20120822144129/http://www.cin.ufpe.br/~if817/arquivos/asmtut/index.html

Reverse engineering

- Understand what the program/a function is doing
- Be aware of architecture/environment
- What does the function expect, where to focus?

Static binary analysis

- Binaries are truthful modulo obfuscation
- Quick glance: `file binary`, `checksec binary`
- Look at the code: `objdump`, `r2`, `ida6q`

Understanding binaries

```
for (int b = 0; b < a; b++) { x; }

  movl  $0, -4(%rbp)
  jmp .L6
.L7:
  movl  -4(%rbp), %eax
  movl  %eax, %esi
  leaq  .LC2(%rip), %rdi
  movl  $0, %eax
  call  printf@PLT
  addl  $1, -4(%rbp)
.L6:
  movl  -4(%rbp), %eax
  cmpl  -20(%rbp), %eax
  jl  .L7
```

Understanding binaries

```
if (a) { x; } else { y; }

  cmpl  $2, -4(%rbp)
  jne .L2
  leaq  .LC0(%rip), %rdi
  movl  $0, %eax
  call  printf@PLT
  jmp .L4
.L2:
  leaq  .LC1(%rip), %rdi
  movl  $0, %eax
  call  printf@PLT
.L4:
```

Testing compilation

- `gcc -S test.c`
- Play with different optimization settings

Understanding binaries

```
while (it != NULL) {
  if (it->val == i) return it;
  it = it->next;
}

  movq  root(%rip), %rax
  testq %rax, %rax
  je  .L3
  cmpl  8(%rax), %edi
  jne .L7; jmp .L3
.L8:
  cmpl  %edi, 8(%rax)
  je  .L3
.L7:
  movq  (%rax), %rax
  testq %rax, %rax
  jne .L8
.L3:
```

Dynamic binary analysis

- Inspect the program at runtime
- `ltrace` lists library functions
- `strace` lists system calls
- `gdb` allows fine-grained introspection

gdb

- Breakpoints: stop execution if hit
- Watchpoint: stop if an address is read/written
- Inspect memory, registers
- Inspect code
- gdb is scriptable!

`http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf`

We may have a core dump to look at

Figure 10

gdb scripting (based on example)

```
python

p = gdb.lookup_type('long').pointer()

def deref(addr):
  val = gdb.Value(addr).cast(p).dereference()
  return int(val) & 0xffffffff

start = gdb.Value(0x601060).cast(p).dereference()

while start != 0x0:
  #print(start)
  char = deref(start + 8) ^ 0x23
  sys.stdout.write(chr(char))
  start = deref(start)
CTRL-D
```

Summary

- Processes execute programs in virtual address spaces
- Programs are encoded as data on a given ISA
- Binaries can be inspected statically or dynamically
- Get a basic understanding first, then focus on details
- Don't try to understand everything, leverage abstractions