# RECITATION 4 – THE STACK

15-213-m12

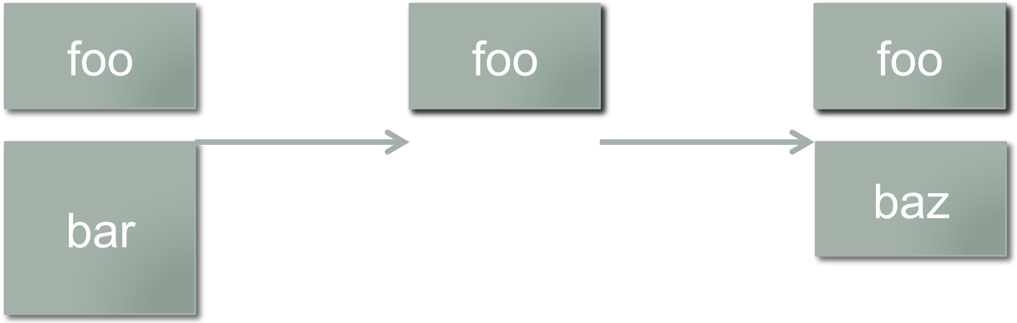
Rick Benua

# The Stack

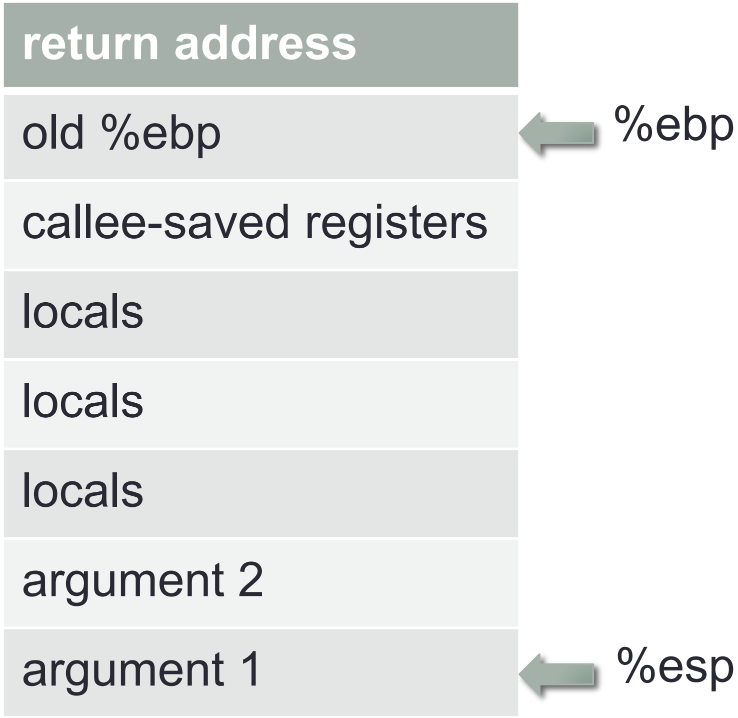
- Region of memory dedicated to local variables and arguments for **all** functions currently being executed
- Maintained using registers %esp and %ebp (on IA32)
  - %esp points to the top of the stack **(actually the lowest address)**
  - %ebp points to the base of the current “frame” – section of data associated with current function
- Modern compilers don’t need %ebp for this
  - Omitted by default on x86-64
  - %rbp can be another GPR
  - Can pass compiler flags to omit it on IA32

# The Stack

- Memory on the stack can be accessed without checks
  - Callee reaches into caller's frame to find arguments
  - Caller may pass a pointer into its stack frame to callee
    - (read as input, or write result, or both!)
- Callee may **NOT** return pointers into its stack
  - Stack space is "freed" upon return
  - Reused for next function call



# Anatomy of a Stack Frame – IA32



**return address**

old %ebp

callee-saved registers

locals

locals

locals

argument 2

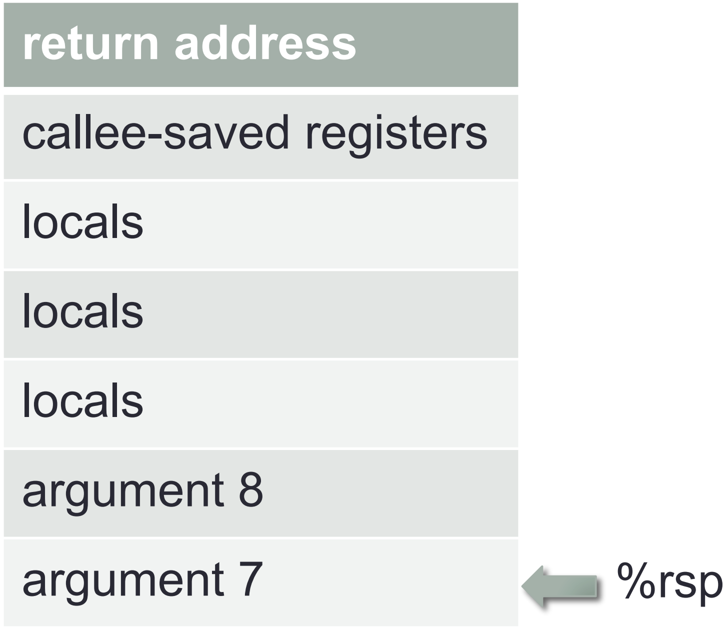
argument 1

%ebp

%esp

- Just before calling a function
- arguments to next call pushed on stack in reverse order

# Anatomy of a Stack Frame – x86-64



**return address**

callee-saved registers

locals

locals

locals

argument 8

argument 7

%rsp

- No base pointer – compiler uses offset from %rsp to find return value
- Arguments passed in registers, but can spill over onto the stack

# Buffer Lab

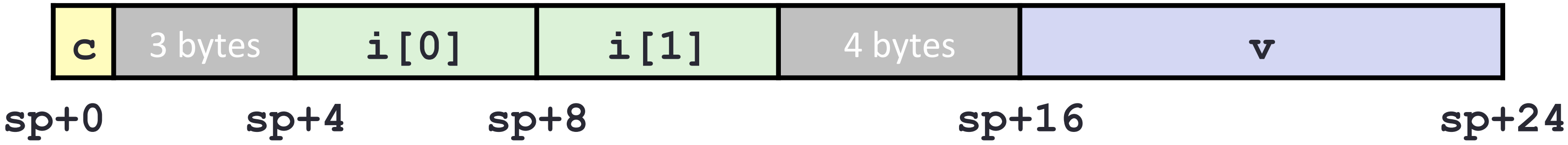
- Out now!
- Due Tuesday
- More examination of programs
  - Create buffer overflow exploits for a known program
- READ THE HANDOUT
  - FOR THE LOVE OF GOD, READ THE ENTIRE HANDOUT
- Series of incrementally more complex exploits

# Buffer Overflow

- Common idiom in code: Copy input from user into buffer, then process it
- Copy may not check length of input
  - Part of the point of this lab is to teach you to **not** do that
- Can reach beyond buffer into other parts of stack
- Strings generally written in from low – high addresses
- “up” the stack, including into saved %ebp or return address!
- This is very bad.

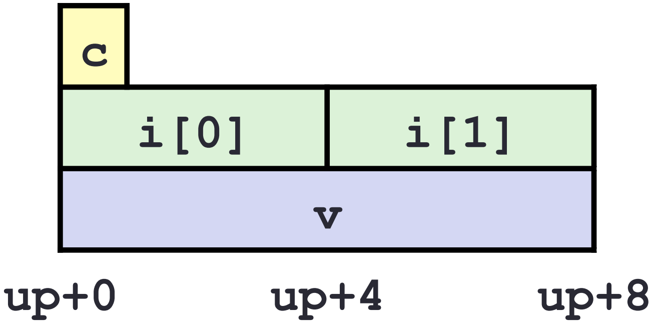
# Structures

- Structures combine sets of related values that can be passed around together
- Values not necessarily contiguous in memory
    - Each value must be aligned to its size
    - Entire struct must be aligned to the largest constraint of any member
- Each member is at a constant offset from the beginning of the struct



# Unions

- Structures store values "next to" each other
- Unions store values "on top of" each other
- Casting between types does conversion
- Union access does not



```
0x00001f30 <main+0>:   push   %ebp
0x00001f31 <main+1>:   mov    %esp,%ebp
0x00001f33 <main+3>:   sub    $0x10,%esp
0x00001f36 <main+6>:   movl   $0x39,-0xc(%ebp)
0x00001f3d <main+13>: movl   $0x0,-0x10(%ebp)
0x00001f44 <main+20>: jmp    0x1f62 <main+50>
0x00001f46 <main+22>: mov    -0xc(%ebp),%eax
0x00001f49 <main+25>: add    $0x1,%eax
0x00001f4c <main+28>: mov    %eax,%ecx
0x00001f4e <main+30>: shr    $0x1f,%ecx
0x00001f51 <main+33>: lea    (%eax,%ecx,1),%eax
0x00001f54 <main+36>: sar    %eax
0x00001f56 <main+38>: mov    %eax,-0xc(%ebp)
0x00001f59 <main+41>: mov    -0x10(%ebp),%eax
0x00001f5c <main+44>: lea    0x1(%eax),%eax
0x00001f5f <main+47>:  mov    %eax,-0x10(%ebp)
0x00001f62 <main+50>: mov    -0x10(%ebp),%eax
0x00001f65 <main+53>: cmp    $0x7,%eax
0x00001f68 <main+56>: jle    0x1f46 <main+22>
0x00001f6a <main+58>: mov    -0x10(%ebp),%eax
0x00001f6d <main+61>: cmp    $0x1,%eax
0x00001f70 <main+64>: je     0x1f7b <main+75>
0x00001f72 <main+66>: movl   $0x1,-0x8(%ebp)
0x00001f79 <main+73>: jmp    0x1f82 <main+82>
0x00001f7b <main+75>: movl   $0x0,-0x8(%ebp)
0x00001f82 <main+82>: mov    -0x8(%ebp),%eax
0x00001f85 <main+85>: mov    %eax,-0x4(%ebp)
0x00001f88 <main+88>: mov    -0x4(%ebp),%eax
0x00001f8b <main+91>: add    $0x10,%esp
0x00001f8e <main+94>: pop    %ebp
0x00001f8f <main+95>:  ret
```

```
int main(){
  int x = 57;
  int y = 0;
  for(; y < 8; y++){
    x = (x + 1) / 2;
  }
  if(y != 1){
    return 1;
  }
  else{
    return 0;
  }
}
```

```
struct{
  int i;
  char c[3];
  struct s *n;
  double d;
  short s;
} s;
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **0x00** | i | i | i | i | c[0] | c[1] | c[2] | **--** |
| **0x08** | n | n | n | n | **--** | **--** | **--** | **--** |
| **0x10** | d | d | d | d | d | d | d | d |
| **0x18** | s | s | **--** | **--** | **--** | **--** | **--** | **--** |