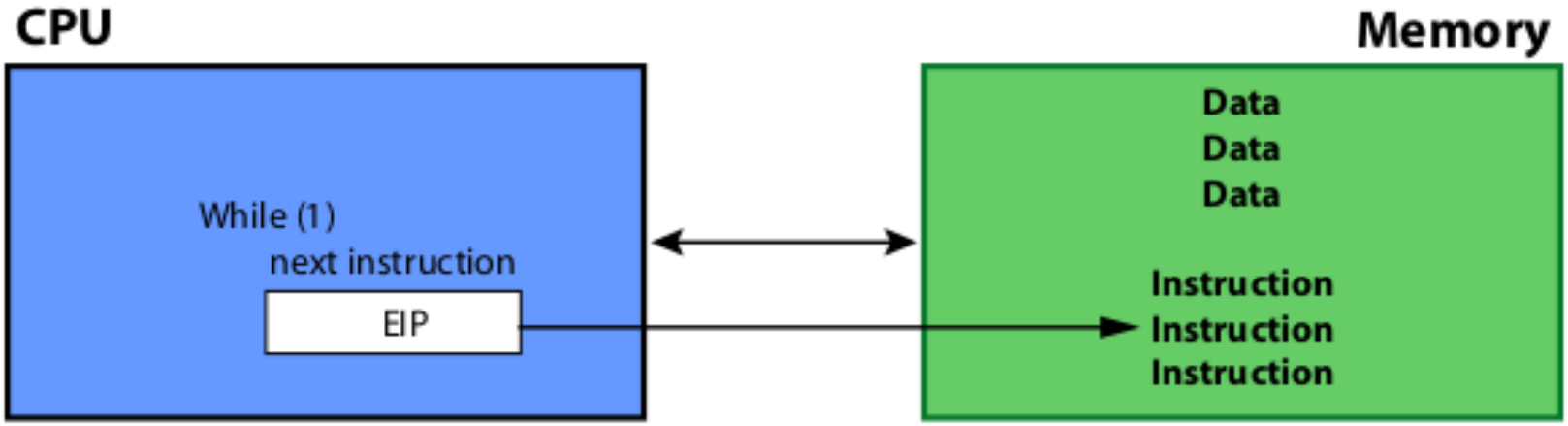
Buffer overflows

Myrto Arapinis
School of Informatics
University of Edinburgh

March 08, 2019

x86 CPU/Memory

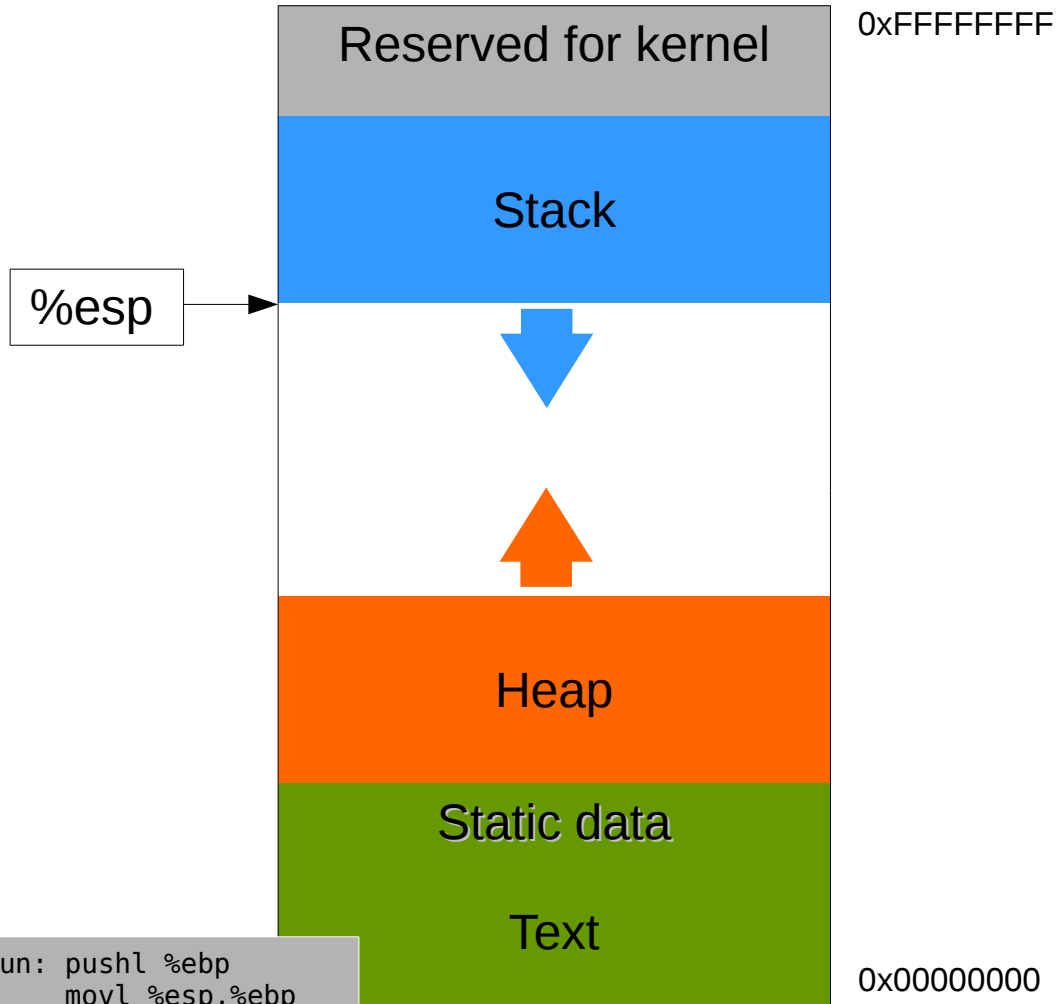


- Memory stores instructions and data
- CPU interprets instructions
- `%eip` points to next instruction
- `%eip` incremented after each instruction
- `%eip` modified by `call`, `ret`, `jmp`, and conditional `jmp`

x86 registers

- Temporary registers: `%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`
- Extended stack pointer: `%esp`
- Extended base pointer: `%ebp`

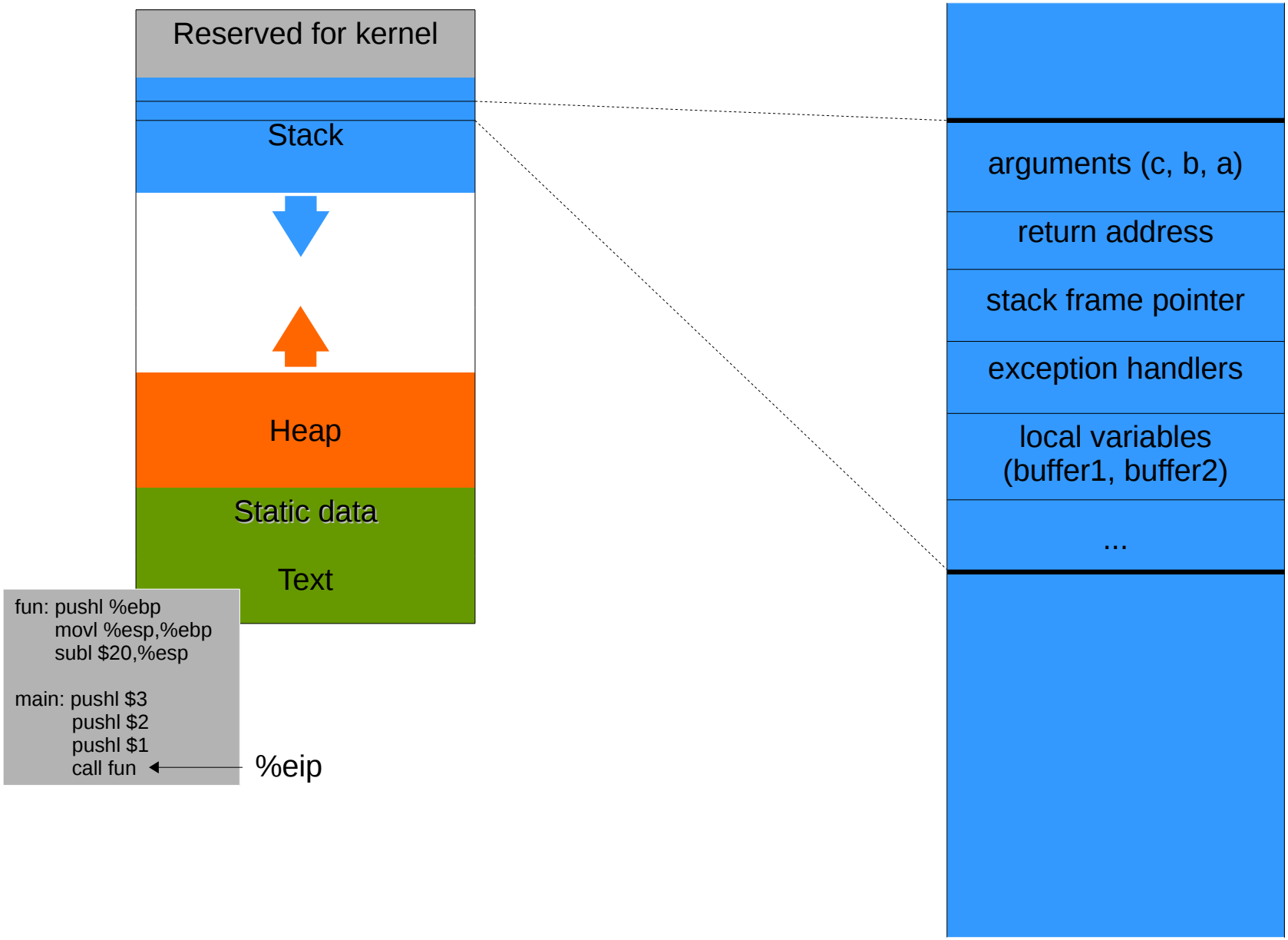
x86 process memory layout (simplified)




```
void function(int a, int b, int c) {
     char buffer1[5];
     char buffer2[10];
}
void main() {
     function(1,2,3);
}
```

```
fun: pushl %ebp
     movl %esp,%ebp
     subl $20,%esp

main: pushl $3
      pushl $2
      pushl $1
      call fun
```

Stack frame


Reserved for kernel
Stack
Heap
Static data
Text
fun: pushl %ebp
movl %esp,%ebp
subl $20,%esp
main: pushl $3
pushl $2
pushl $1
call fun
%eip
arguments (c, b, a)
return address
stack frame pointer
exception handlers
local variables
(buffer1, buffer2)
...

Stack and functions: Summary

Calling function

1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction to run after control returns
3. Jump to the function's address

Called function

4. Push the old frame pointer onto the stack (%ebp)
5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
6. Push local variables onto the stack

Returning function

7. Reset the previous stack frame: %esp = %ebp, %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%esp)

x86 assembly

emacs@myrto-thinkpad

File Edit Options Buffers Tools C Help

Save Undo

```
#include<stdio.h>

int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void){
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d\n", x);
  return 0;
}
```

-:--- **fact.c** All L1 (C/l Abbrev)

emacs@myrto-thinkpad

File Edit Options Buffers Tools Asm Help

Save Undo

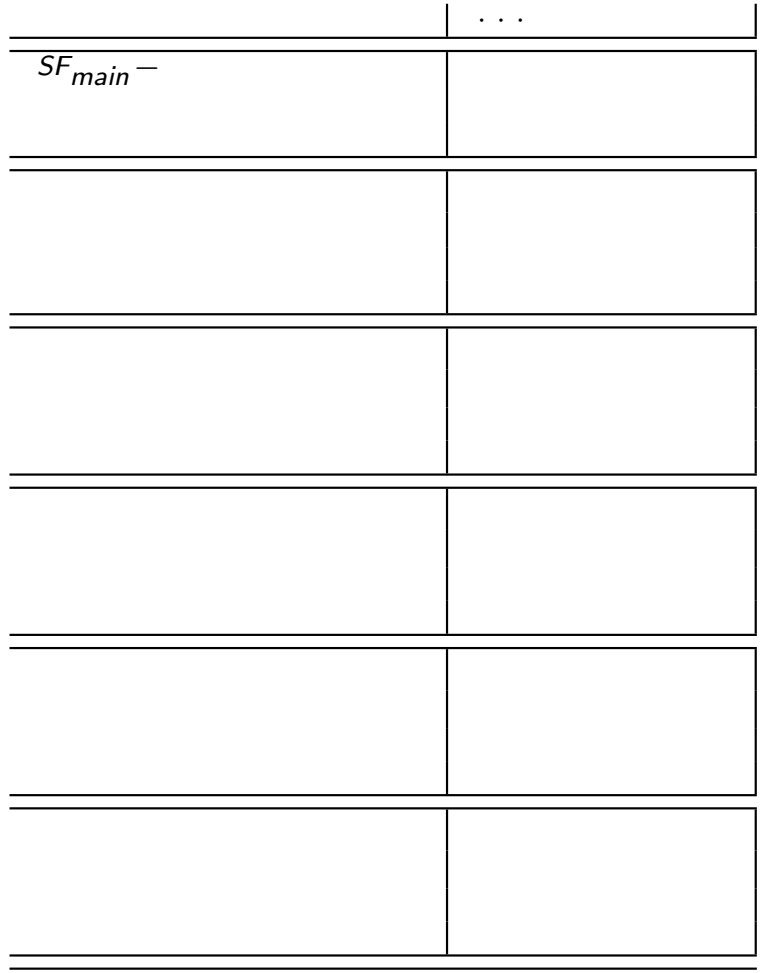
```
fact:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    $1, -12(%ebp)
        cmpl    $0, 8(%ebp)
        jne     .L2
        movl    -12(%ebp), %eax
        jmp     .L1
.L2:
        jle     .L1
        movl    8(%ebp), %eax
        subl    $1, %eax
        movl    %eax, (%esp)
        call    fact
        movl    %eax, -12(%ebp)
        movl    8(%ebp), %eax
        imull   -12(%ebp), %eax
        jmp     .L1
.L1:
        leave
        ret
.LC0:
        .string "Factorial 4 is %d\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 28(%esp)
        movl    $4, (%esp)
        call    fact
        movl    %eax, 28(%esp)
        movl    28(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

-:--- **fact.s** All L26 (Assembler)

Wrote /home/marapini/Documents/Work/Teaching/INFR10067-ComputerSecurity/1819/Lectures/Lec18_OS_Intro/GDB_demo/fact.s

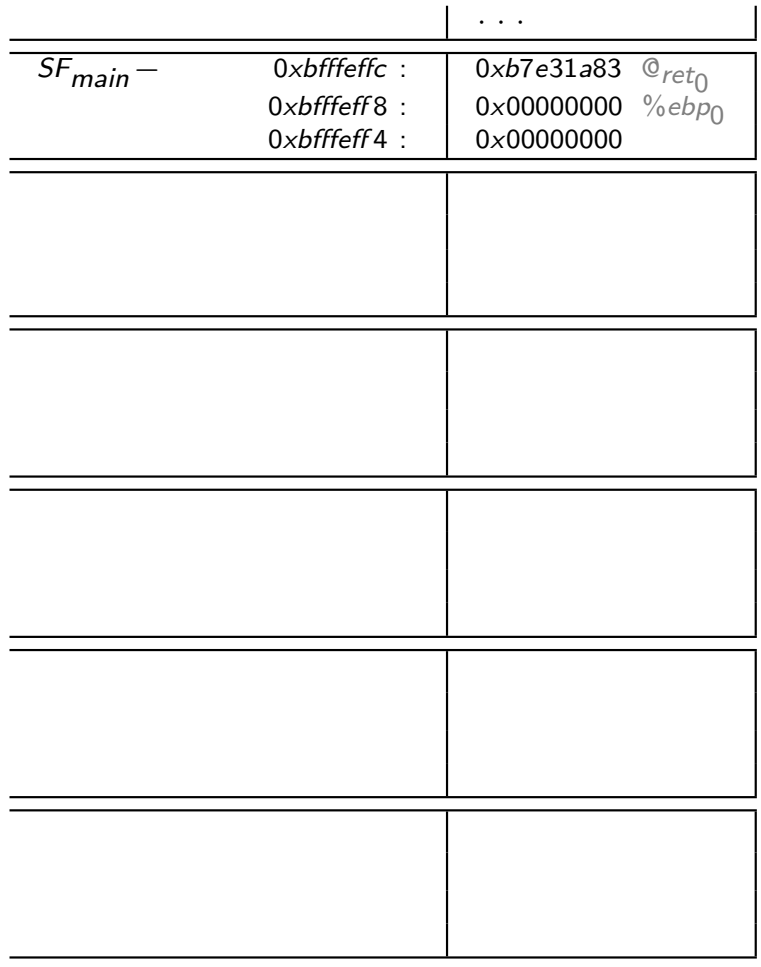
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



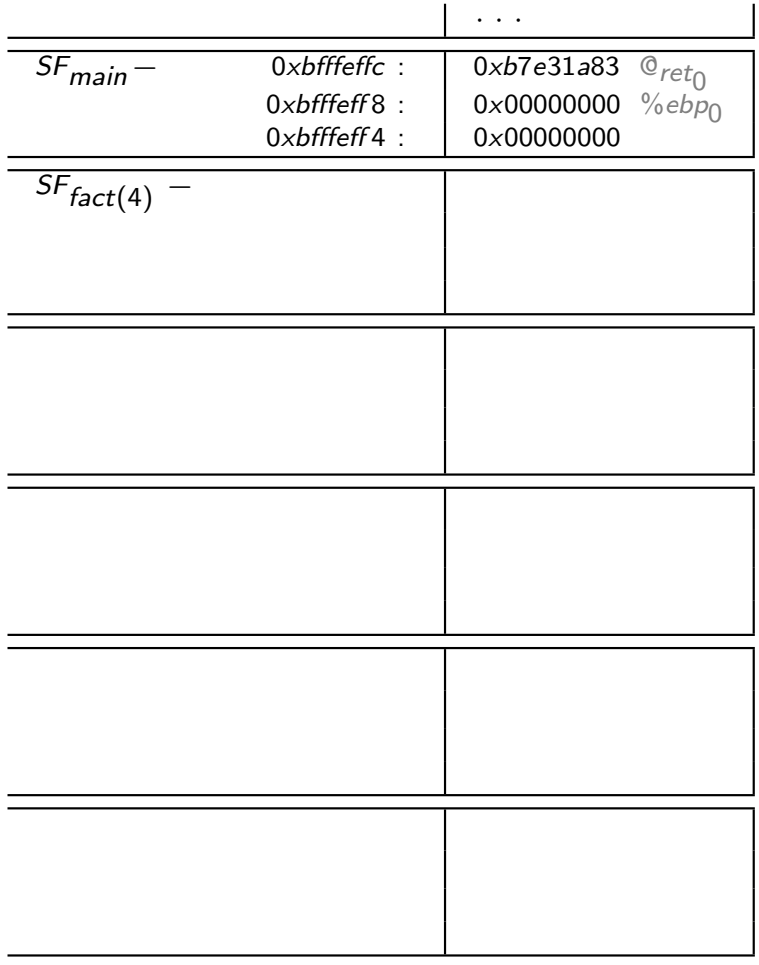
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



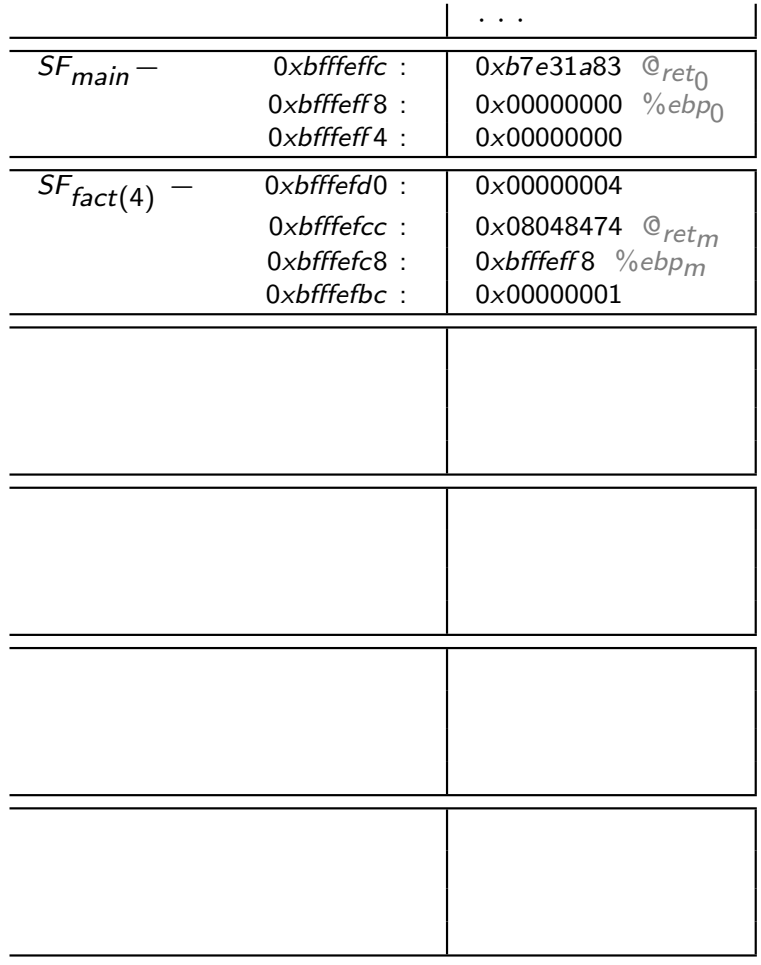
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



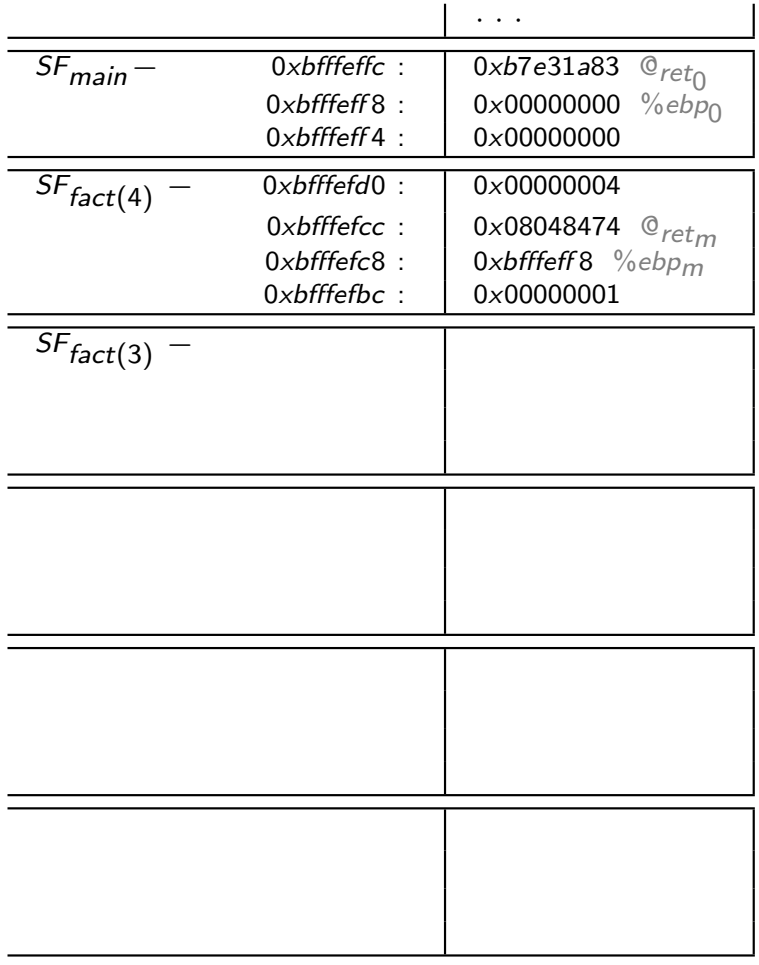
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @$_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @$_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @$_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ $@_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ $\%ebp_0$
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ $@_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ $\%ebp_m$
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ $@_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ $\%ebp_4$
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —		

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @$_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @$_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @$_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ @$_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ %ebp_3
	$0xbfffef5c$:	$0x00000001$

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @$_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @$_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @$_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ @$_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ %ebp_3
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —		

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ $@_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ $\%ebp_0$
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ $@_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ $\%ebp_m$
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ $@_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ $\%ebp_4$
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ $@_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ $\%ebp_3$
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —	$0xbfffef40$:	$0x00000001$
	$0xbfffef3c$:	$0x08048449$ $@_{ret_2}$
	$0xbfffef38$:	$0xbfffef68$ $\%ebp_2$
	$0xbfffef2c$:	$0x00000001$

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @ret_0
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @ret_m
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @ret_4
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ @ret_3
	$0xbfffef68$:	$0xbfffef98$ %ebp_3
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —	$0xbfffef40$:	$0x00000001$
	$0xbfffef3c$:	$0x08048449$ @ret_2
	$0xbfffef38$:	$0xbfffef68$ %ebp_2
	$0xbfffef2c$:	$0x00000001$
$SF_{fact(0)}$ —		

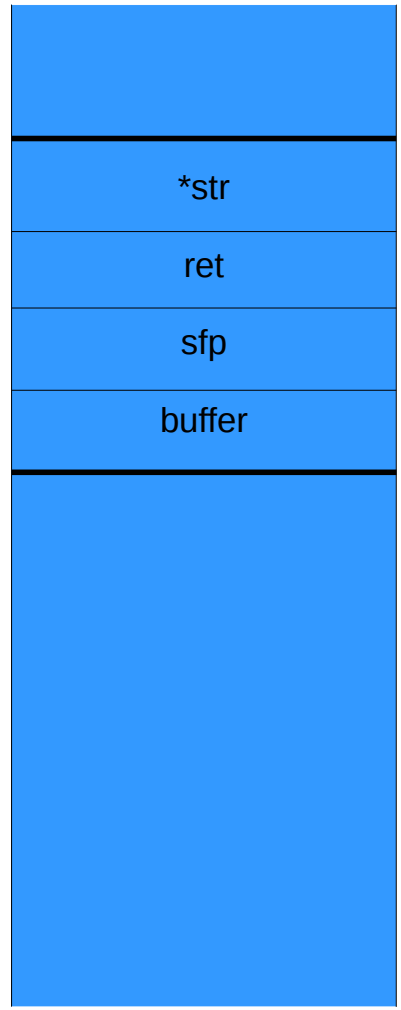
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ $@_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ $\%ebp_0$
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ $@_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ $\%ebp_m$
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ $@_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ $\%ebp_4$
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ $@_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ $\%ebp_3$
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —	$0xbfffef40$:	$0x00000001$
	$0xbfffef3c$:	$0x08048449$ $@_{ret_2}$
	$0xbfffef38$:	$0xbfffef68$ $\%ebp_2$
	$0xbfffef2c$:	$0x00000001$
$SF_{fact(0)}$ —	$0xbfffef10$:	$0x00000000$
	$0xbfffef0c$:	$0x08048449$ $@_{ret_1}$
	$0xbfffef08$:	$0xbfffef38$ $\%ebp_1$
	$0xbfffeefc$:	$0x00000001$

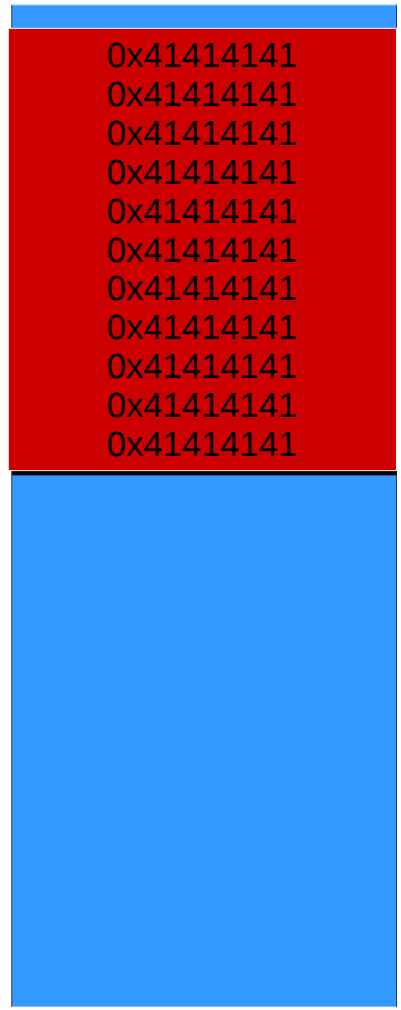
Buffer overflows

```
void function(char *str) {
     char buffer[16];
     strcpy(buffer,str);
}
void main() {
     char large_string[256];
     int i;
     for( i = 0; i < 255; i++)
          large_string[i] = 'A';
function(large_string);
}
```



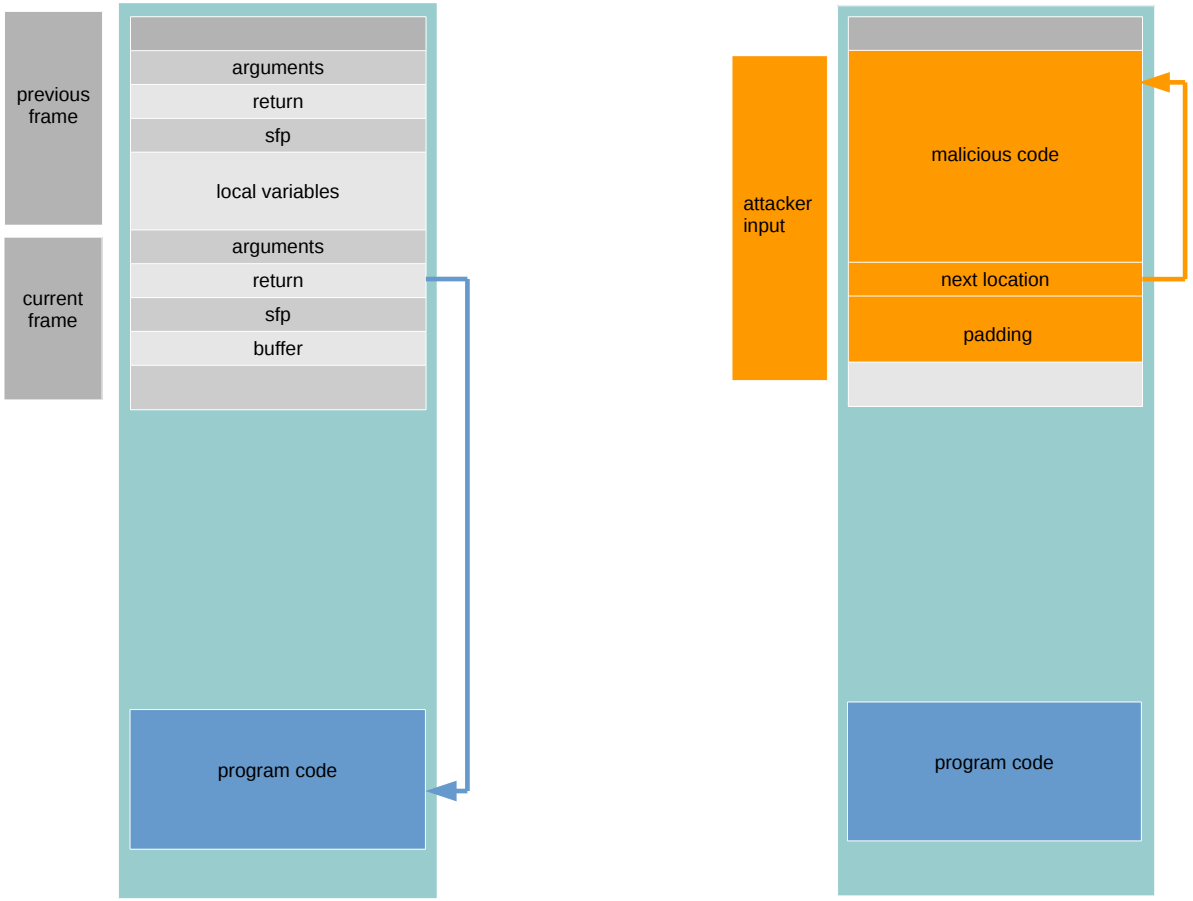
Buffer overflows

```
void function(char *str) {
     char buffer[16];
     strcpy(buffer,str);
}
void main() {
     char large_string[256];
     int i;
     for( i = 0; i < 255; i++)
          large_string[i] = 'A';
function(large_string);
}
```



`strcpy(src,dest)` does not check that `dest` is bigger than `src`
The return address is now `0x41414141`

Control hijacking



A buffer overflow can change the flow of execution of the program:

- load malicious code into memory
- make %eip point to it

Shellcode injection

Goal: "spawn a shell" - will give the attacker general access to the system

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

C code

```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

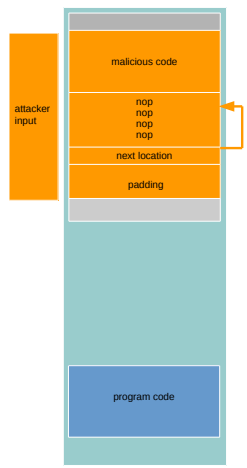
Machine code
(part of attacker's input)

- must inject the machine code instructions (code ready to run)
- the code cannot contain any zero bytes (`printf`, `gets`, `strcpy` will stop copying)
- can't use the loader (we're injecting)

The return address

Challenge: find the address of the injected malicious code?

- If code accessible: we know how far is the overflowed variable from the saved %ebp
- If code not accessible: try different possibilities!
  In a 32 bits memory space, there are 2^{32} possibilities
- NOP sled
  - guess approximate stack state when the function is called
  - insert many NOPs before Shell Code



Reference

Aleph One. Smashing The Stack For Fun And Profit.
http://phrack.org/issues/49/14.html#article