Software Security

Prof. Leonardo B. Oliveira

Agenda

- **C language**
- Memory Management
- Attacks
- Countermeasures
- Final Remarks

Look at this code

```c
#include <stdio.h>

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
    return 0;
}
```

- What is this function doing?

Look at this code

```c
#include <stdio.h>

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
    return 0;
}
```

- What is this function doing?
- What's going to happen if we compile it?

Look at this code

```c
#include <stdio.h>

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
    return 0;
}
```

- What is this function doing?
- What's going to happen if we compile it?
- What's going to happen if we run it?

Demo

Access past the end of an array

Look at this code

```c
#include <stdio.h>

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
    return 0;
}
```

```
security_course$cc test.cpp
security_course$./a.out
32767
security_course$
```

Look at this code

```c
#include <stdio.h>

int main() {
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
    return 0;
}
```

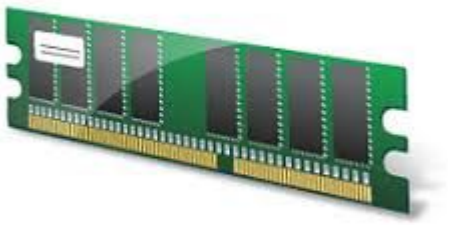
- ➔ In C there's no array bound-checking
- ➔ An illegal array access can be performed
- ➔ Like accessing the 6th element of a 5-element array
- ➔ One can read (or write) past the end of the array!
- ➔ **And then bad things might happen**

Agenda

- C language
- **Memory Management**
- Attacks
- Countermeasures
- Final Remarks

Memory Management: Concepts

- If you know how something works, you can protect yourself against it
  - This holds true for computer stuff as well!
- You must be familiar with computer architecture, object files, and the runtime stack



Memory Organization

For each program, the OS allocates 3 different memory segments

Memory Organization

For each program, the OS allocates 3 different memory segments

Q: What are they?

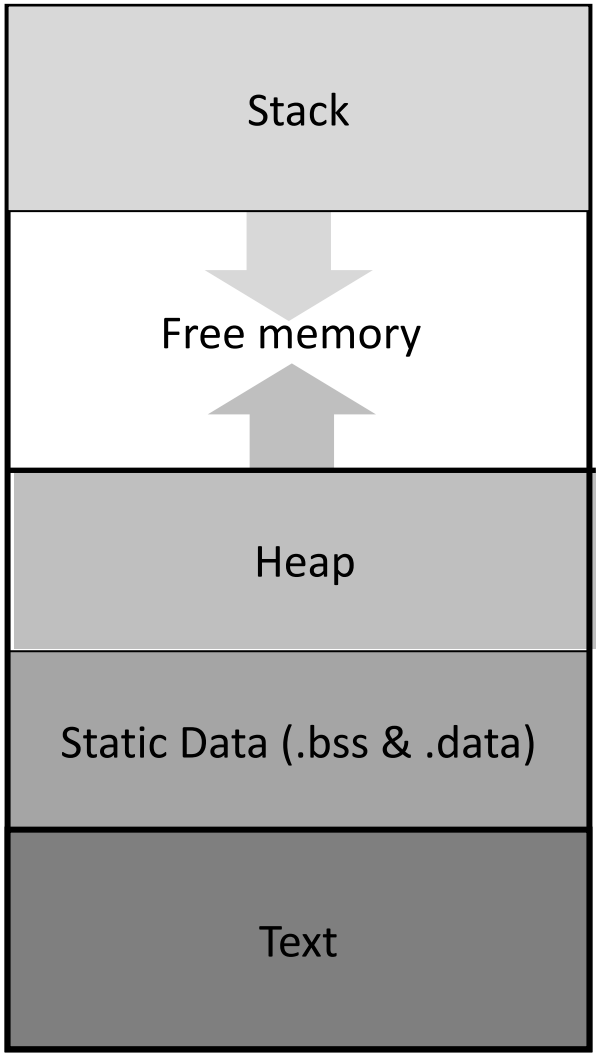
Memory Organization

Stack: stores function's local variables and its own control information

For each program, the OS allocates 3 different memory segments
Q: What are they?

Data segment: stores both dynamic (heap) and also static (.bss and .data) data

Text segment: program instructions in the form of machine code



Byte Ordering

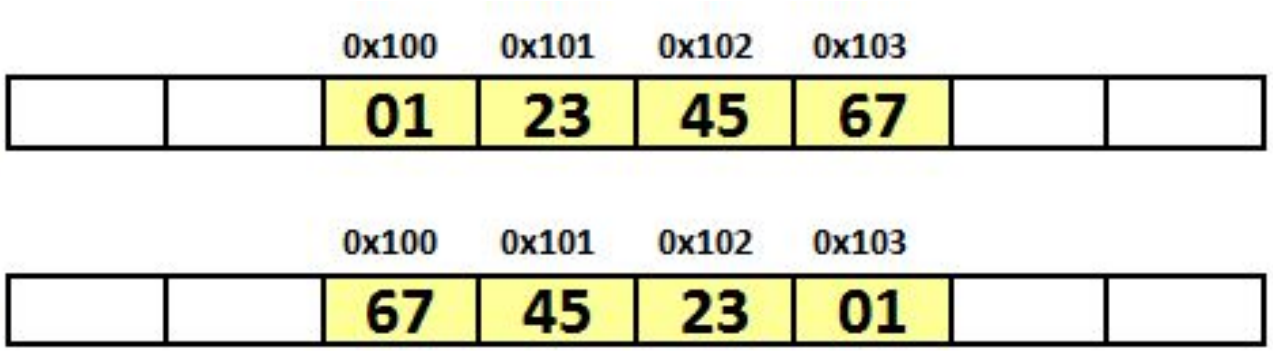
- Big endian – most significant (big) byte in smallest address
  - E.g. Sun
- Little endian – least significant (little) byte in smallest address
  - E.g. IA32

Byte Ordering

- Big endian – most significant (big) byte in smallest address
  - E.g. Sun
- Little endian – least significant (little) byte in smallest address
  - E.g. IA32
- How would the word 0x01234567 be placed at address 0x100?

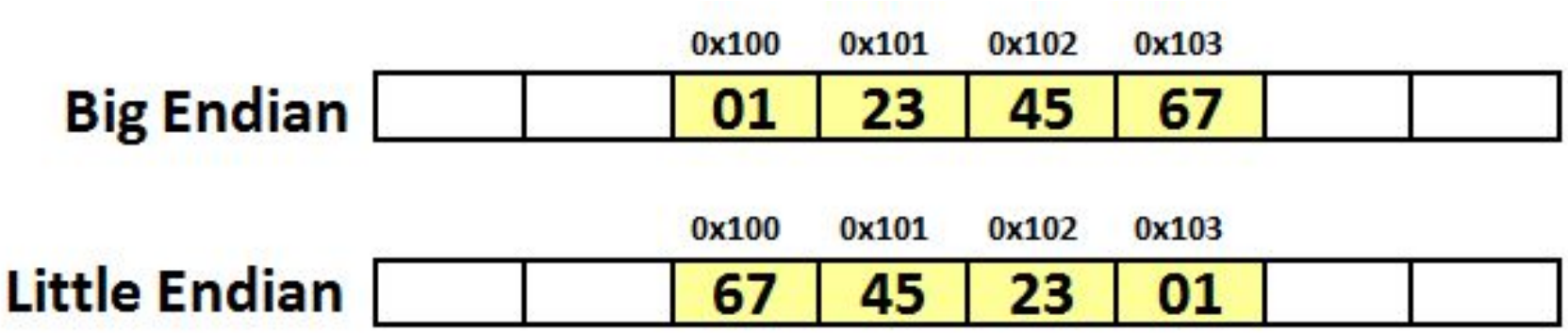
Byte Ordering

- Big endian – most significant (big) byte in smallest address
  - E.g. Sun
- Little endian – least significant (little) byte in smallest address
  - E.g. IA32
- Q: How would the word 0x01234567 be placed at address 0x100?
  - Q: Which is big or little endian?



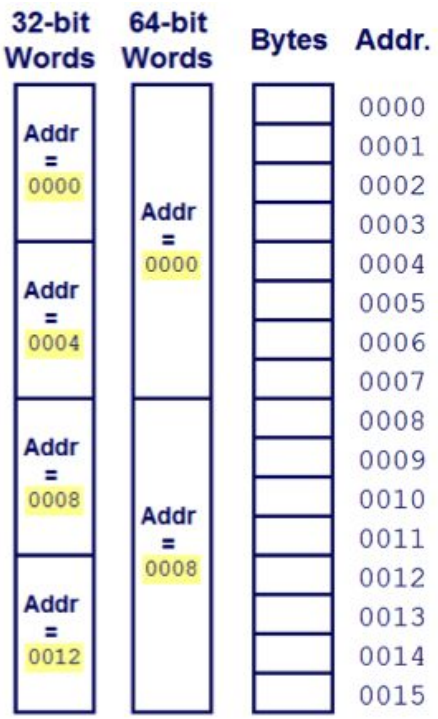
Byte Ordering

- Big endian – most significant (big) byte in smallest address
  - E.g. Sun
- Little endian – least significant (little) byte in smallest address
  - E.g. IA32
- Q: How would the word 0x01234567 be placed at address 0x100?
  - Q: Which is big or little endian?



Stack Alignment

- Easy concept, but fairly important
- Word alignment requires that words begin in multiple of *n*, e.g.:
  - n = 4 for 32-bit architecture
  - n = 8 for 64-bit architecture



Stack Alignment

```
struct {
    char x; // 1 byte
    int  y; // 4 bytes
    char z; // 1 byte
    int  w; // 4 bytes
} foo;
```

- Q: What's the size of foo?

Stack Alignment

```
struct {
    char x; // 1 byte
    int  y; // 4 bytes
    char z; // 1 byte
    int  w; // 4 bytes
} foo;
```

- Q: What's the size of foo?
- A: 10 bytes?

Stack Alignment

```c
#include <stdio.h>

struct {
    char x; // 1 byte
    int  y; // 4 bytes
    char z; // 1 byte
    int  w; // 4 bytes
} foo;

int main() {
    printf("sizeof(foo) == %lu\n", sizeof(foo));
    return 0;
}
```

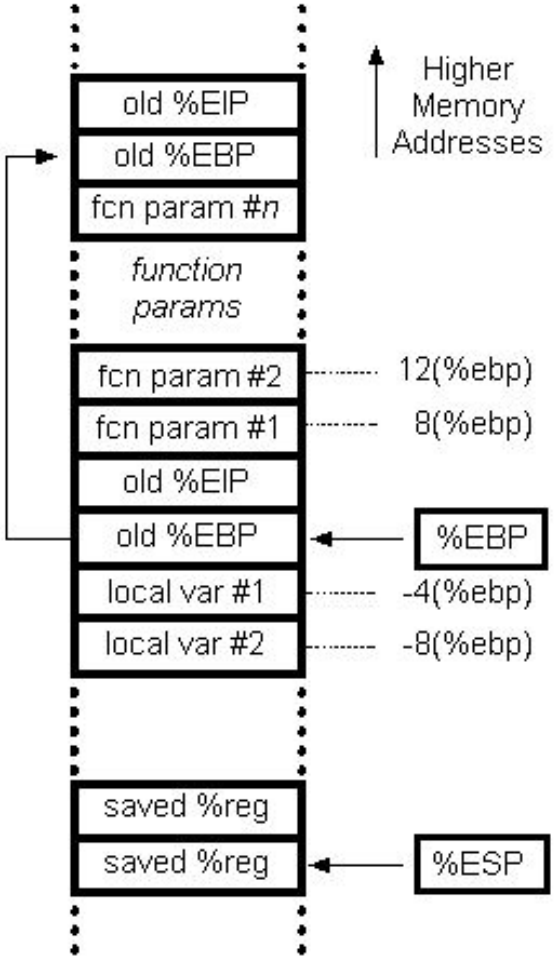
```
security_course$clang test.c
security_course$./a.out
sizeof(foo) == 16
```

Demo

struct size

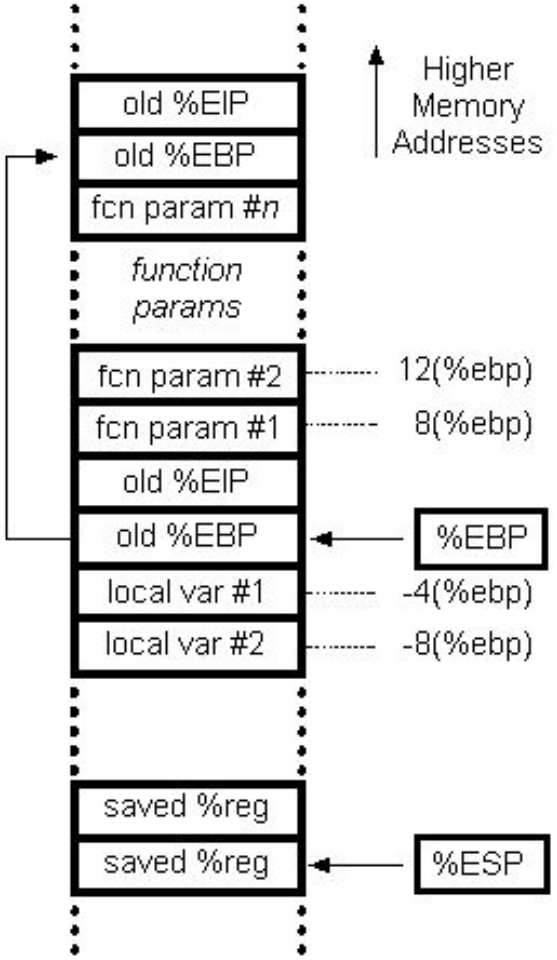
Functions & The Stack

- Q: What's the purpose of a stack?



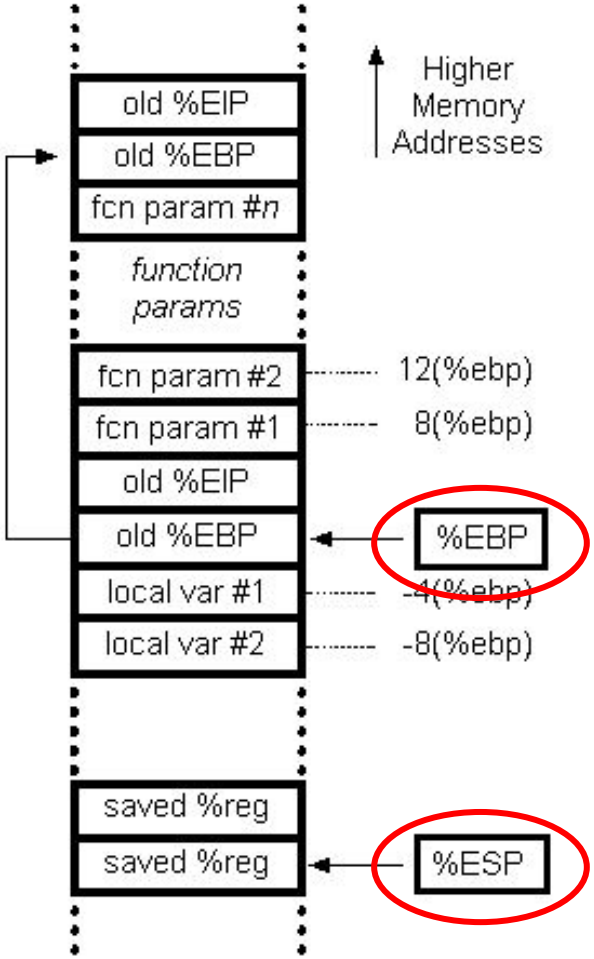
Functions & The Stack

- Q: What's the purpose of a stack?
- A: To implement the dynamics of a function efficiently
  - Change the execution flow of a program
  - Execute some instructions
  - Return control to the point where the function was called



Functions & The Stack

- Q: What's the purpose of a stack?
- A: To implement the dynamics of a function efficiently
  - Change the execution flow of a program
  - Execute some instructions
  - Return control to the point where the function was called
- Especial registers
  - ESP
  - EBP



Special Registers

- ESP
  - ESP points to the top of the stack
  - In the IA32 architecture, it points to the latest used address

Special Registers

- ESP
  - ESP points to the top of the stack
  - In the IA32 architecture, it points to the latest used address
- EBP
  - AKA, *frame* or *base pointer*
  - It is often used to calculate and address based on another address, i.e., EBP plus an offset

Function Dynamics

- Consider the following code

```c
#include <stdio.h>

void foo(int x) {
    printf("Hello World: %d\n", x);
}

int main() {
    foo(1);
    return 0;
}
```

Function Dynamics

- Consider the following code

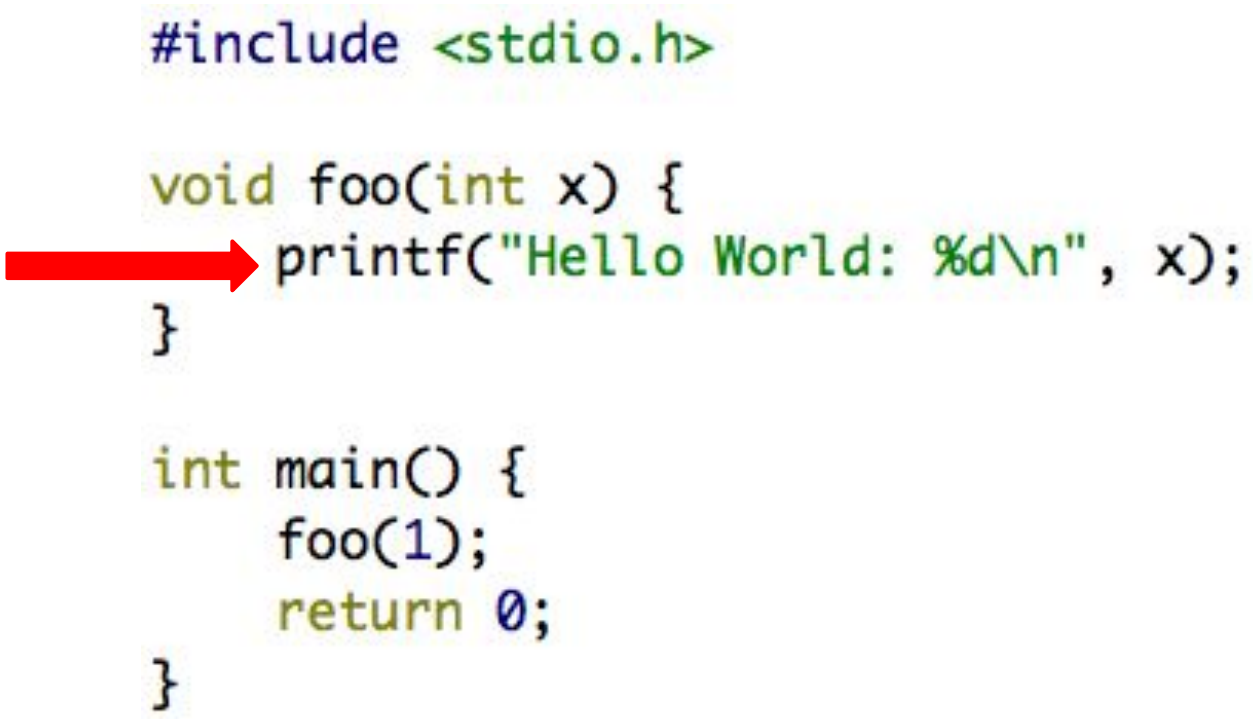
```c
#include <stdio.h>

void foo(int x) {
    printf("Hello World: %d\n", x);
}

int main() {
    foo(1);
    return 0;
}
```

Function Dynamics

- Consider the following code



```c
#include <stdio.h>

void foo(int x) {
    printf("Hello World: %d\n", x);
}

int main() {
    foo(1);
    return 0;
}
```

Function Dynamics

- Consider the following code

```c
#include <stdio.h>

void foo(int x) {
    printf("Hello World: %d\n", x);
}

int main() {
    foo(1);
    return 0;
}
```

If you wanna get software security right you really need to understand this function dynamics!

Right before a function call

- Q: What happens in between a function call and its actual execution?

Right before a function call

- Q: What happens in between a function call and its actual execution?
- A: Whenever a function call is made, the following instructions are carried out
  1. Push function's arguments, in reverse order, onto the stack
  2. Push the return address (RET) onto the stack

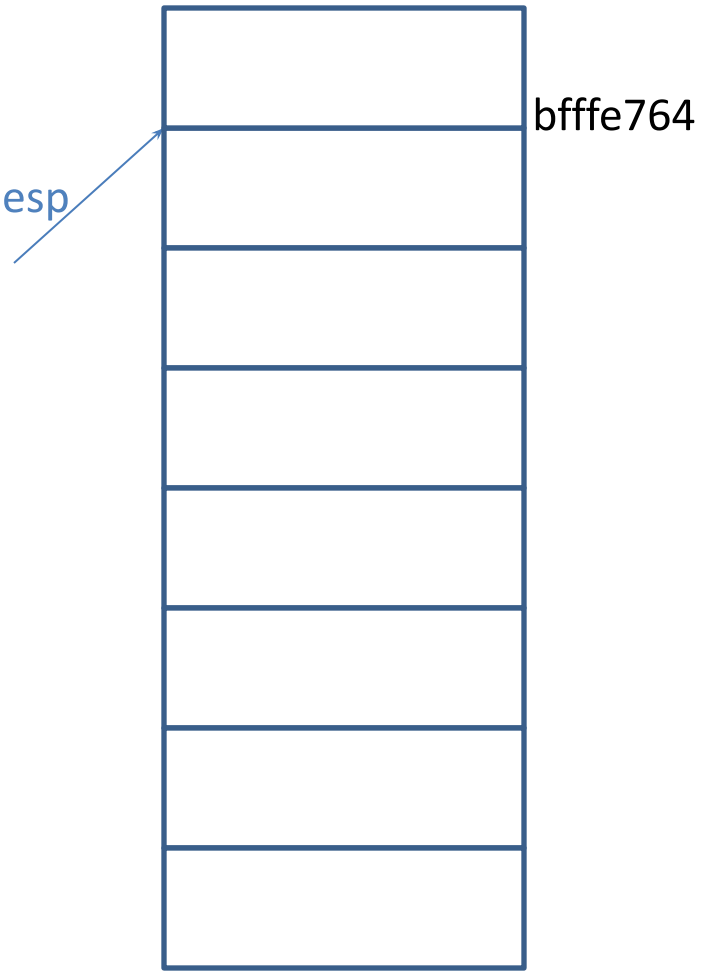
Right before a function call

- Q: What happens in between a function call and its actual execution?
- A: Whenever a function call is made, the following instructions are carried out
  1. Push function's arguments, in reverse order, onto the stack
  2. Push the return address (RET) onto the stack

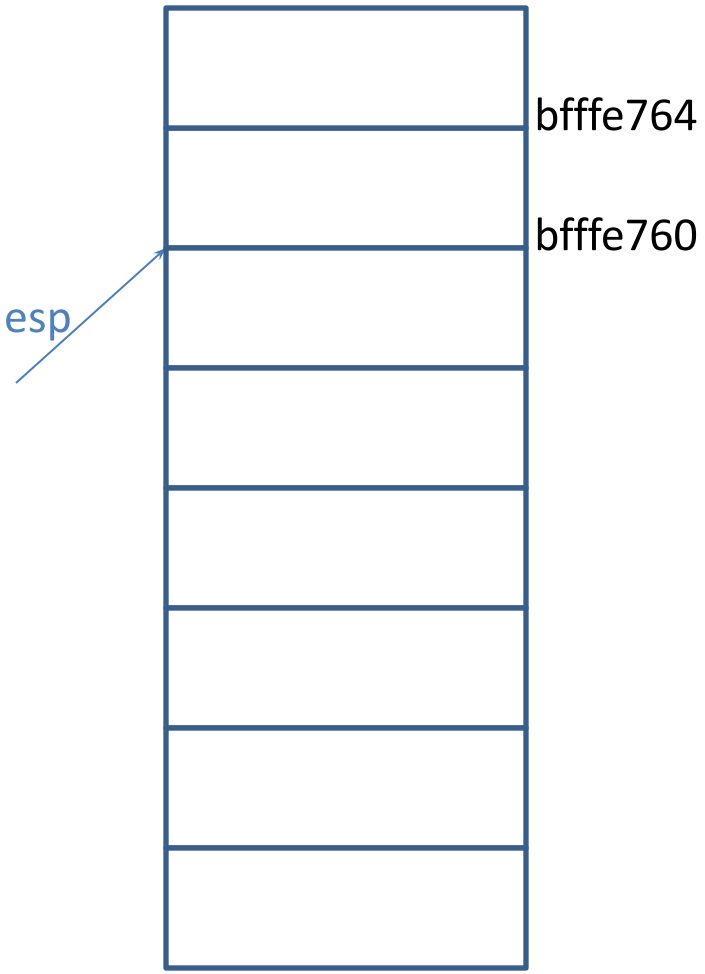
  Q: What is RET and why are we pushing it?

  A: Yes, so we can come back to where we came from

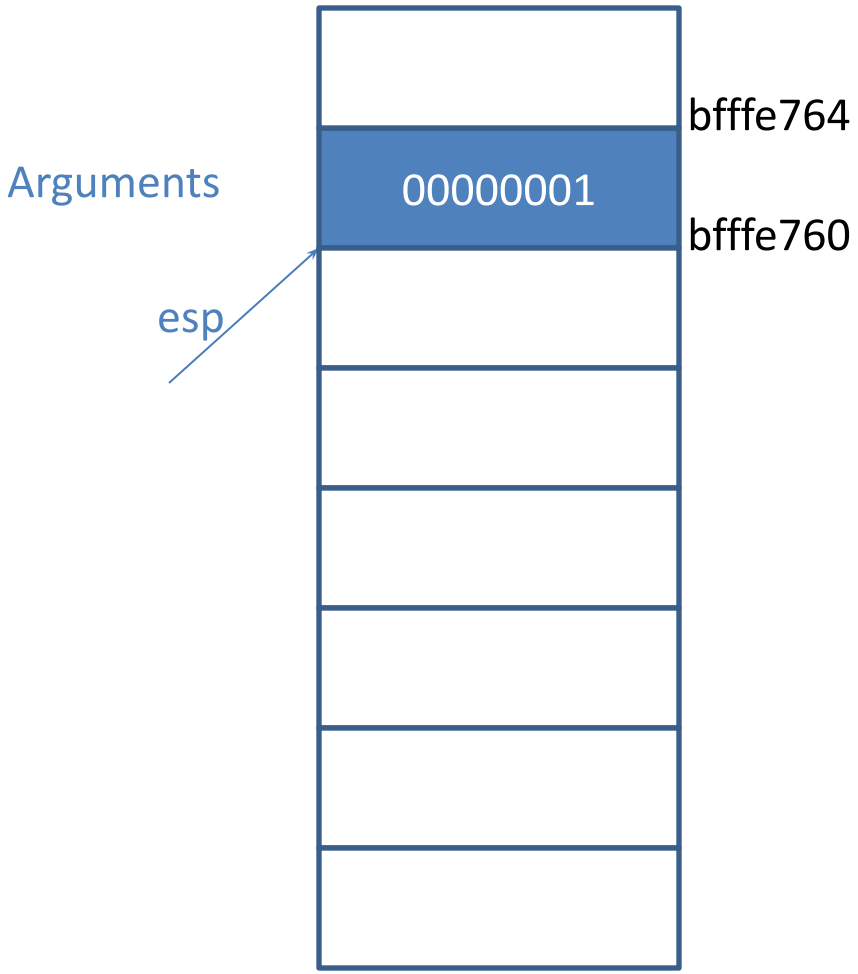
```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



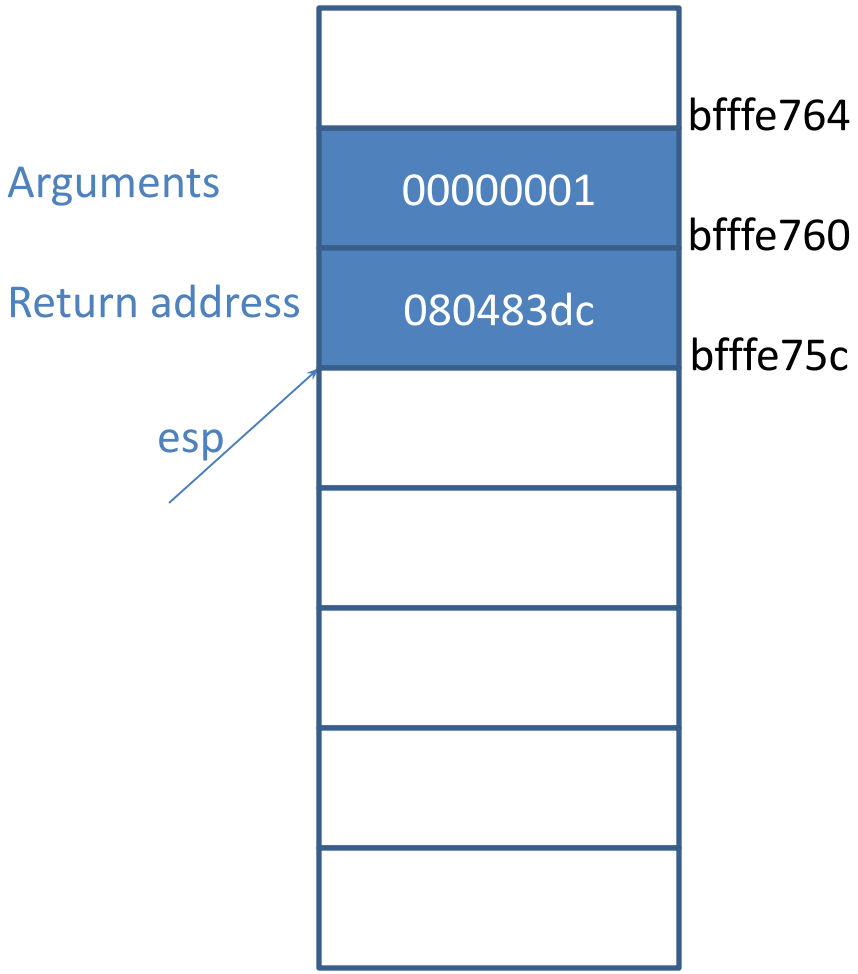
```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



Prologue

- A number of instructions are executed in the beginning of a function
- The idea is to prepare the stack and register for use within the function
- This process is given the name of *prologue*

Prologue

- A number of instructions are executed in the beginning of a function
- The idea is to prepare the stack and register for use within the function
- This process is given the name of *prologue*
- Q: What happens during prologue?

Prologue steps

1. Backs the old EBP up (e.g. function *main* EBP)
   - We push the old EBP onto the stack so we can restore its value later on after the function returns

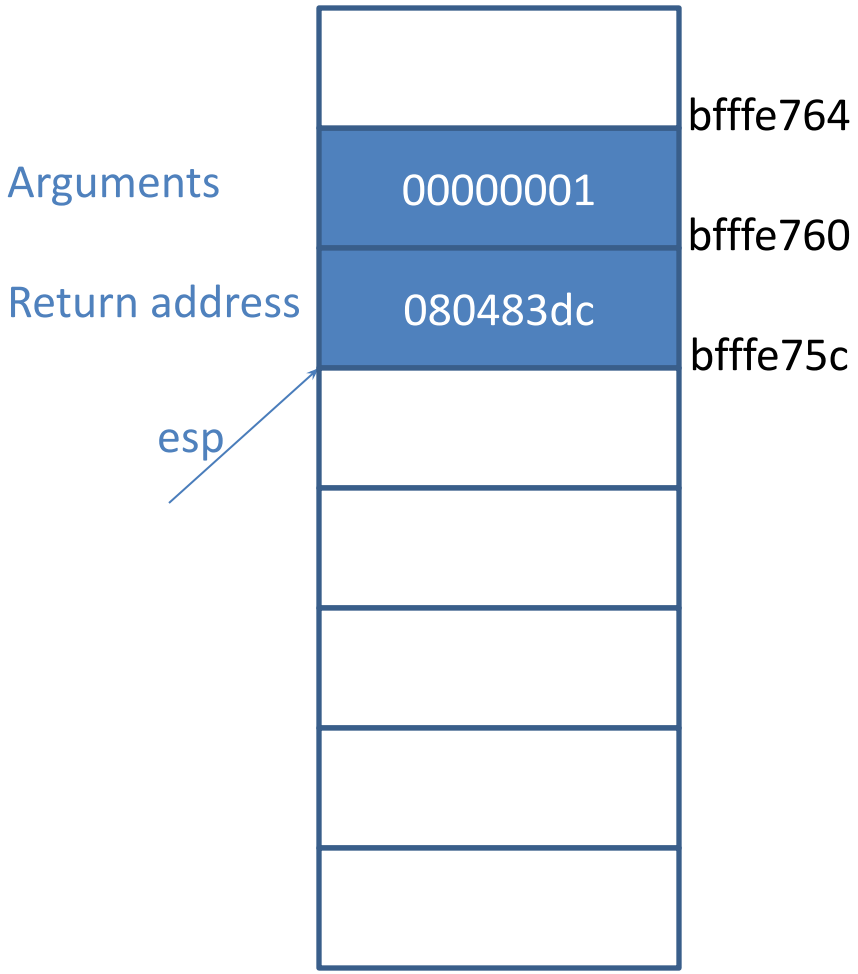
Prologue steps

1. Backs the old EBP up (e.g. function *main* EBP)
   - We push the old EBP onto the stack so we can restore its value later on after the function returns
2. Create the new EBP
   - We do this by making EBP point to the memory location of the current ESP

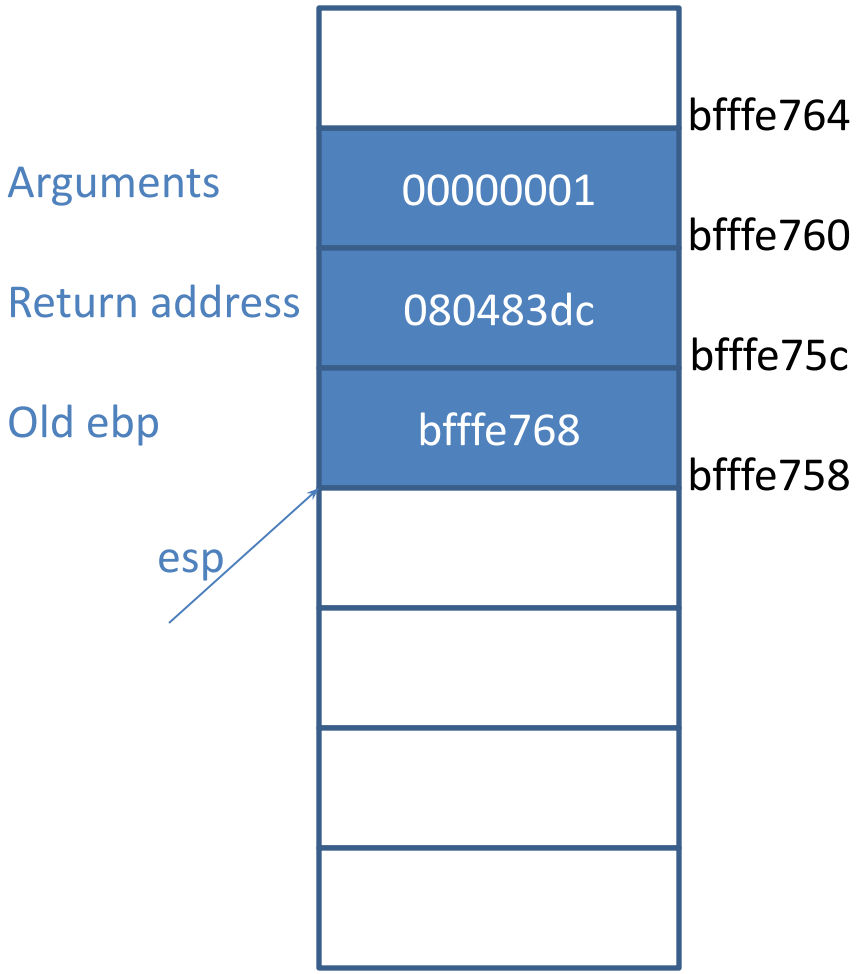
Prologue steps

1. Backs the old EBP up (e.g. function *main* EBP)
   - We push the old EBP onto the stack so we can restore its value later on after the function returns
2. Create the new EBP
   - We do this by making EBP point to the memory location of the current ESP
3. Allocate the space needed for the function's local variables
   - By e.g. decrementing variables' sizes from ESP

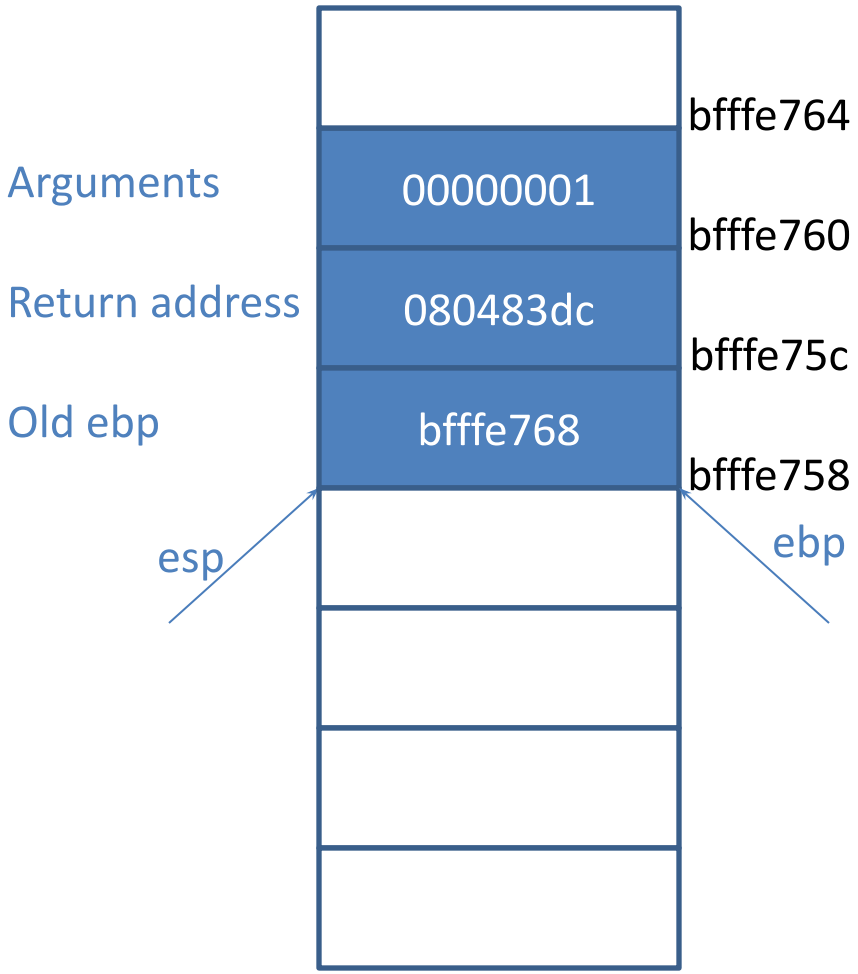
```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



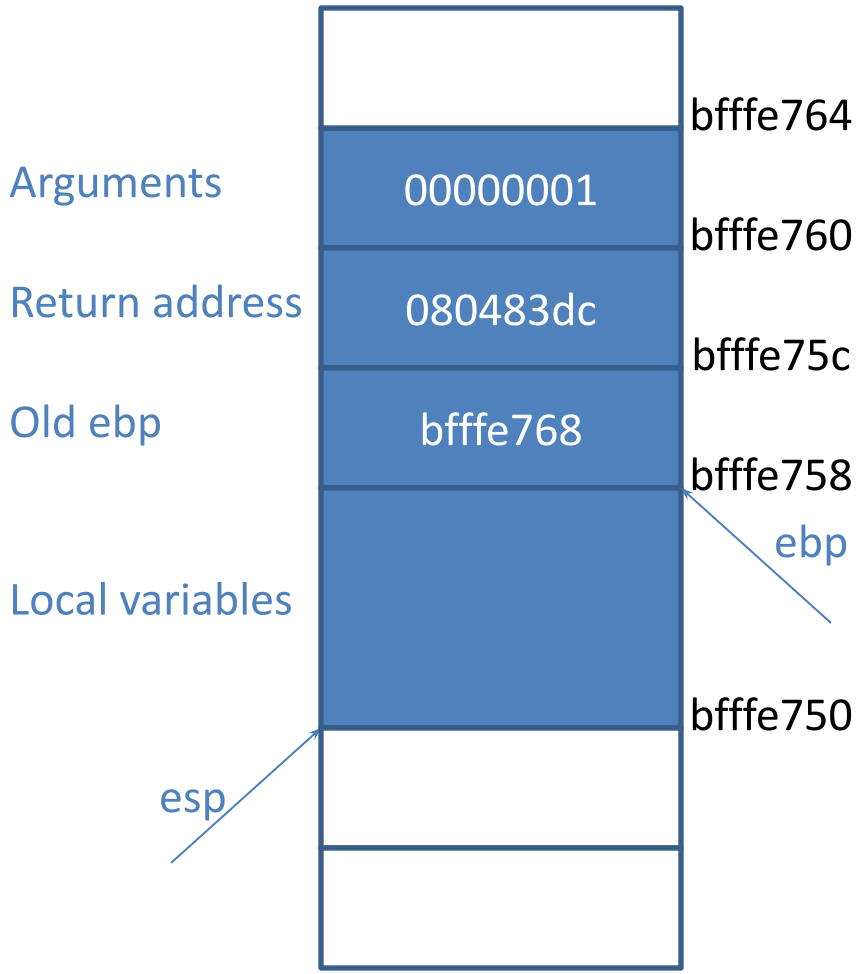
```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



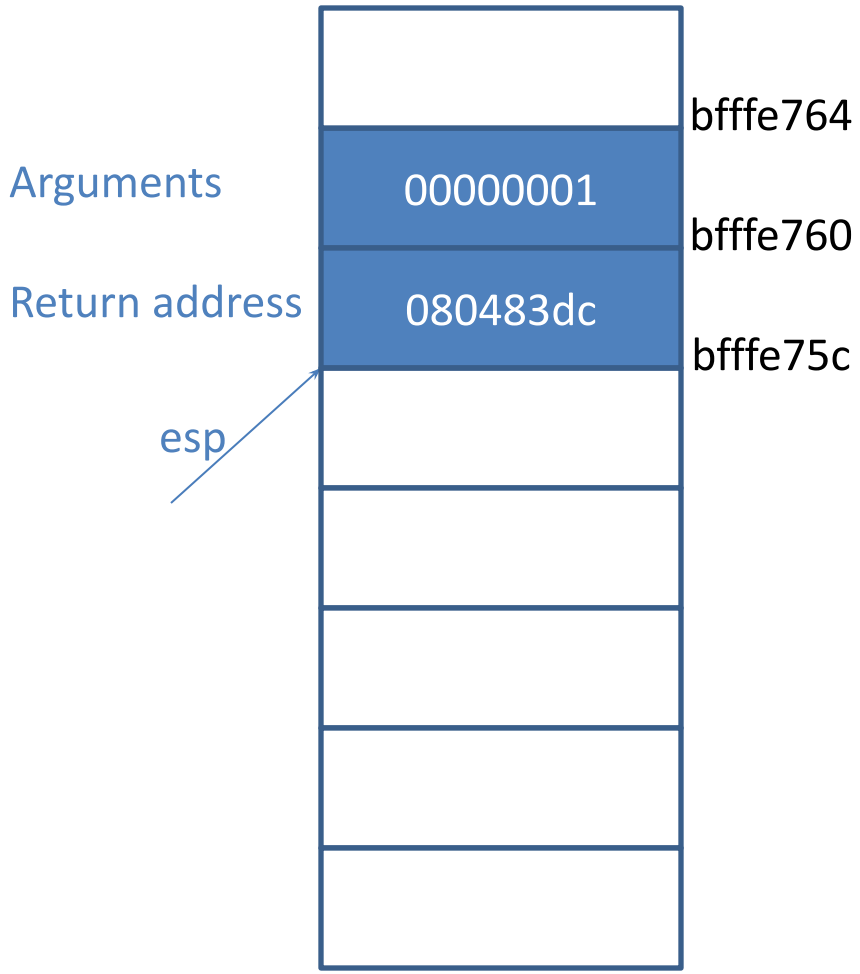
```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



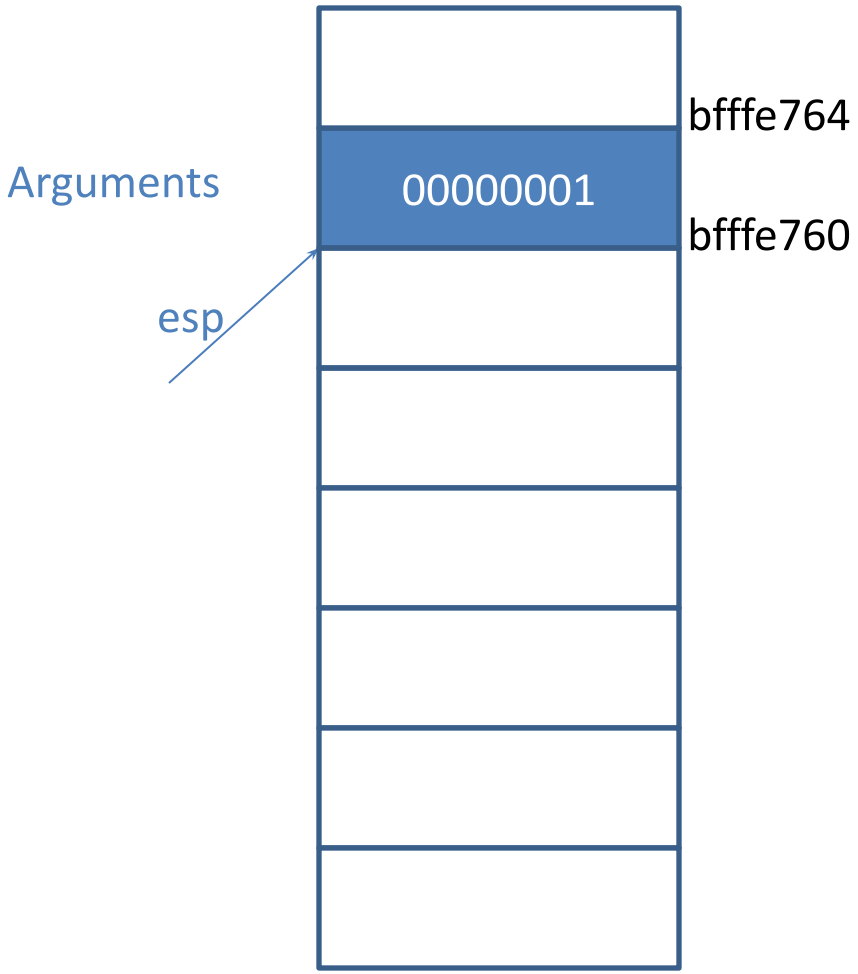
```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



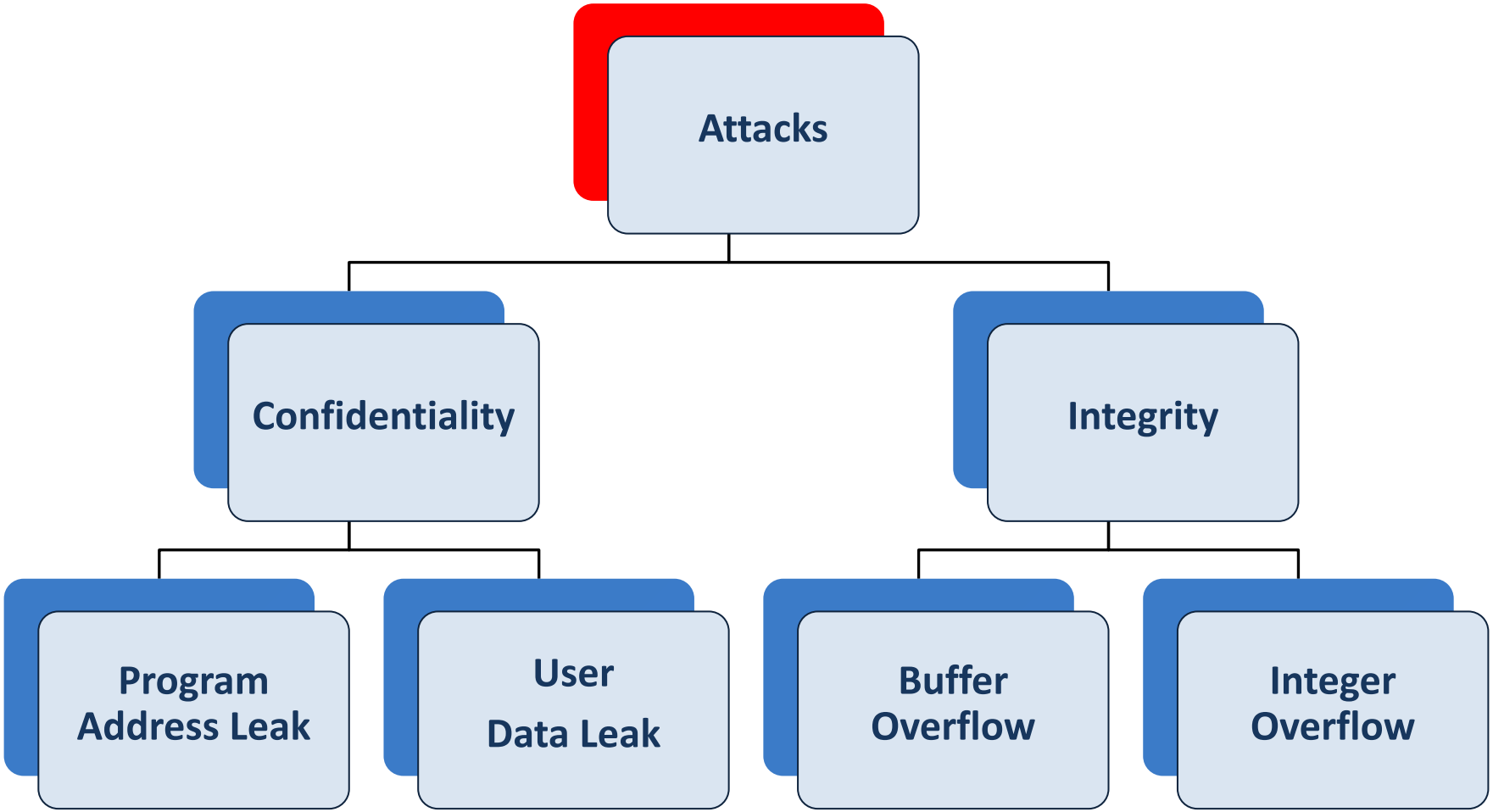
```
        ...
foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        ...
        leave
        ret
        ...
main:
        ...
        subl $4, %esp
        movl $1, (%esp)
        call foo
        ...
```



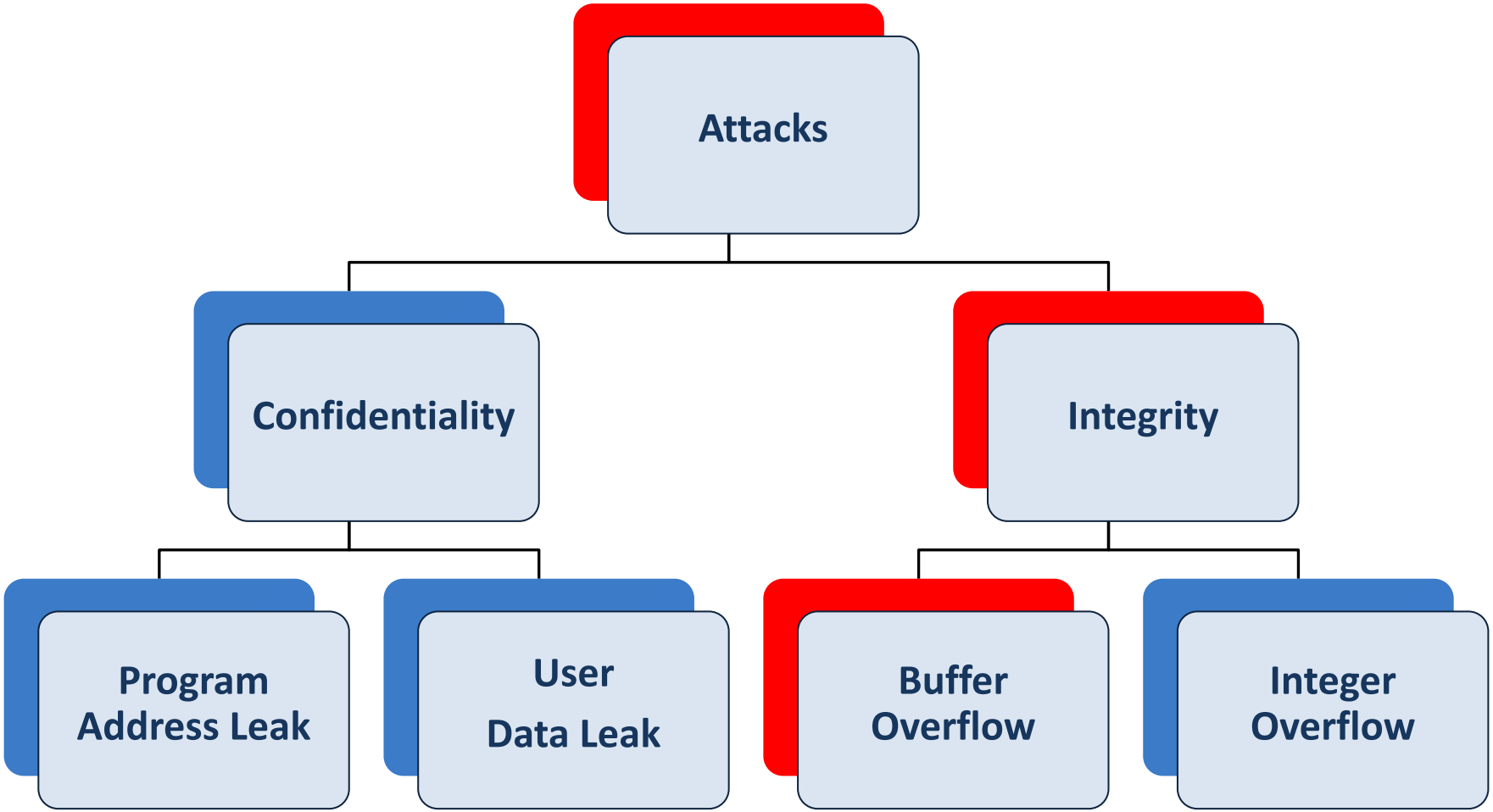
Agenda

- C language
- Memory Management
- **Attacks**
- Countermeasures
- Final Remarks

Attacks Taxonomy



Attacks Taxonomy

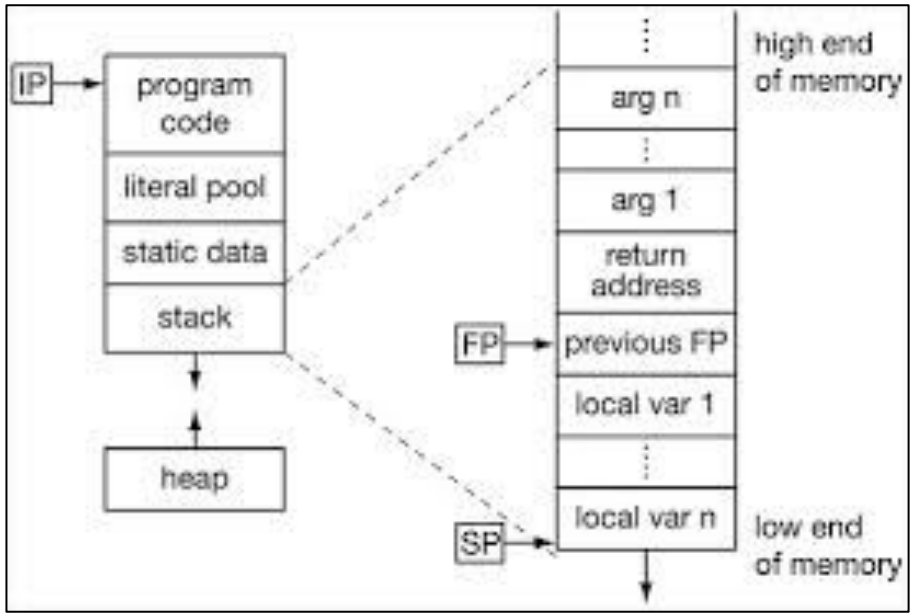


Buffer Overflow

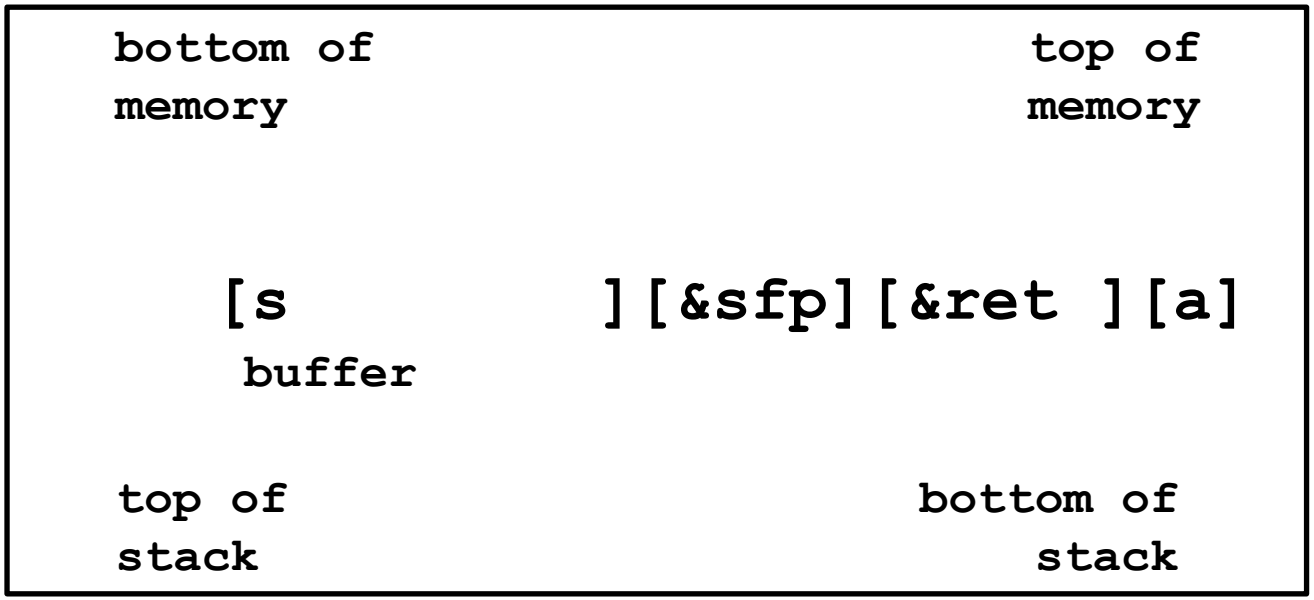
- A buffer overflow happens if, when writing to a buffer, access beyond its boundaries is made and adjacent memory addresses are modified
- The idea is to manipulate data through arrays that are not bound-checked
  - E.g. `strcpy(buffer,str);`

Stack Smashing

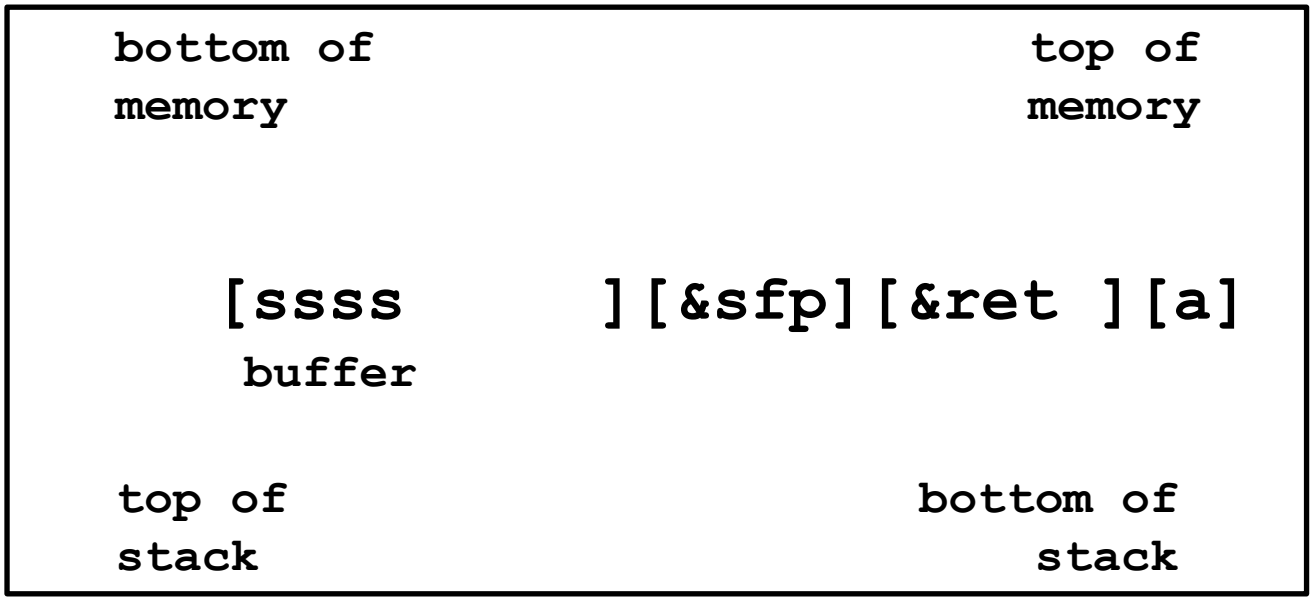
- *Stack smashing* is called when a buffer overflow happens inside a stack



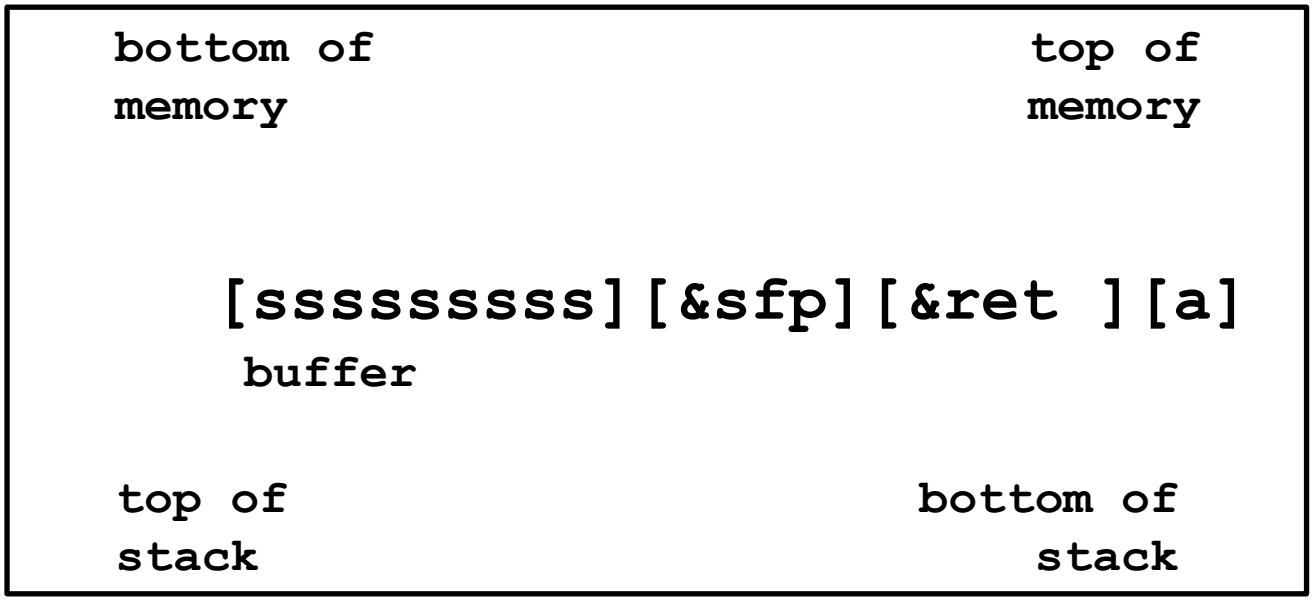
Buffer Overflow Attack (Cont.)



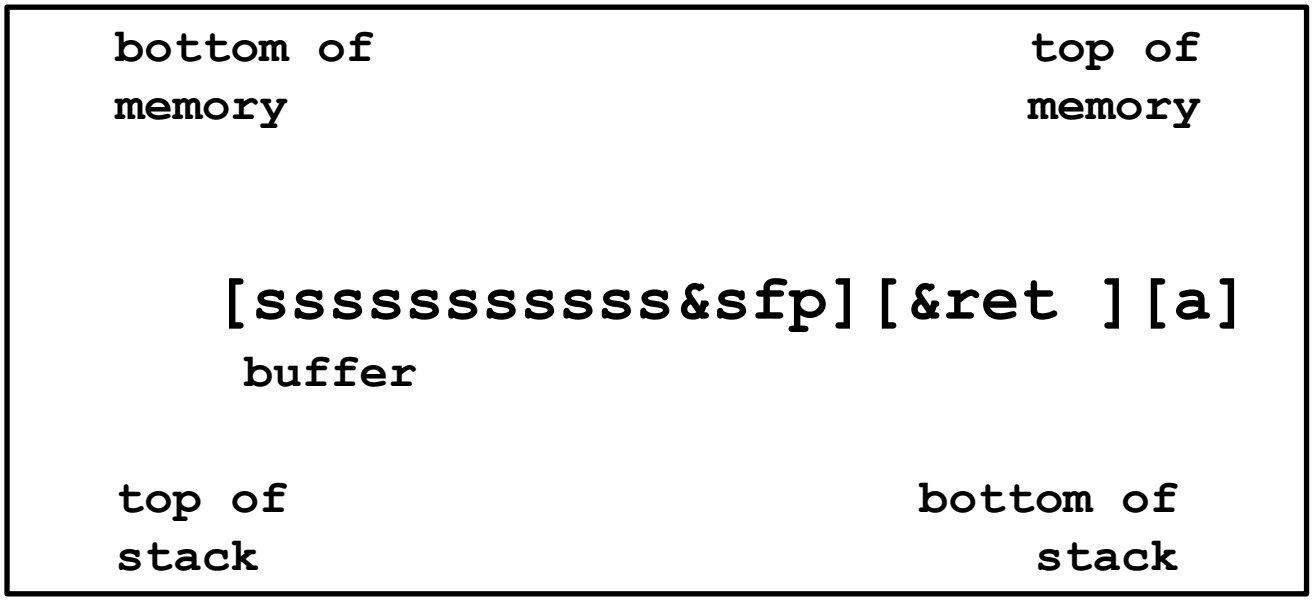
Buffer Overflow Attack (Cont.)



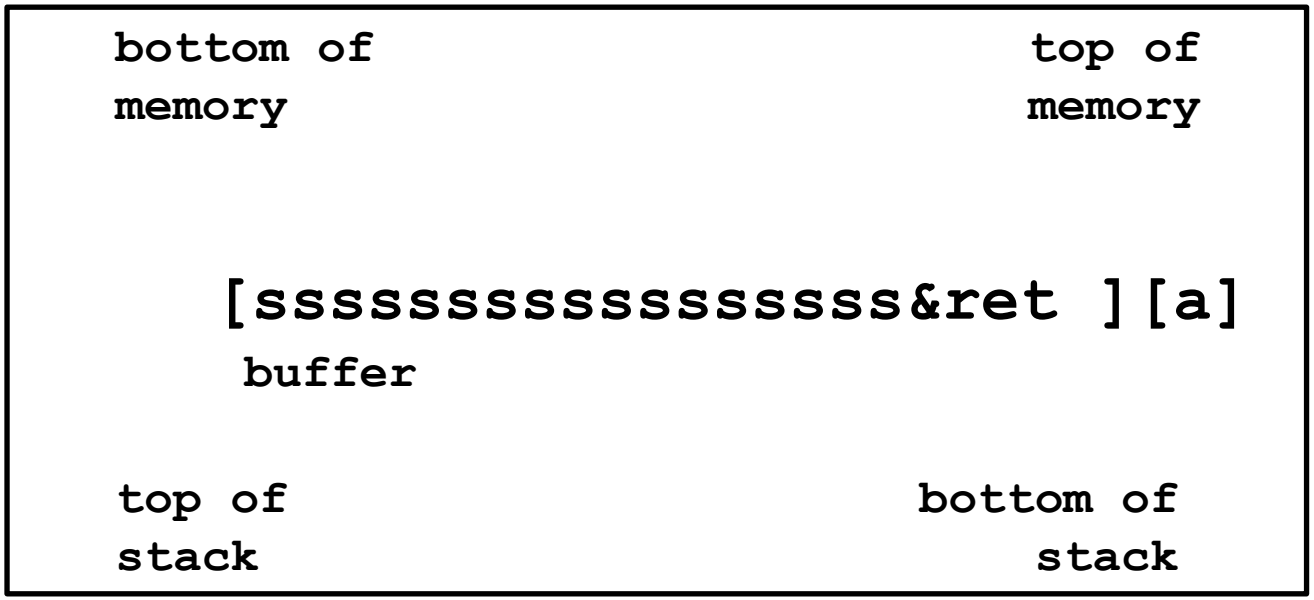
Buffer Overflow Attack (Cont.)



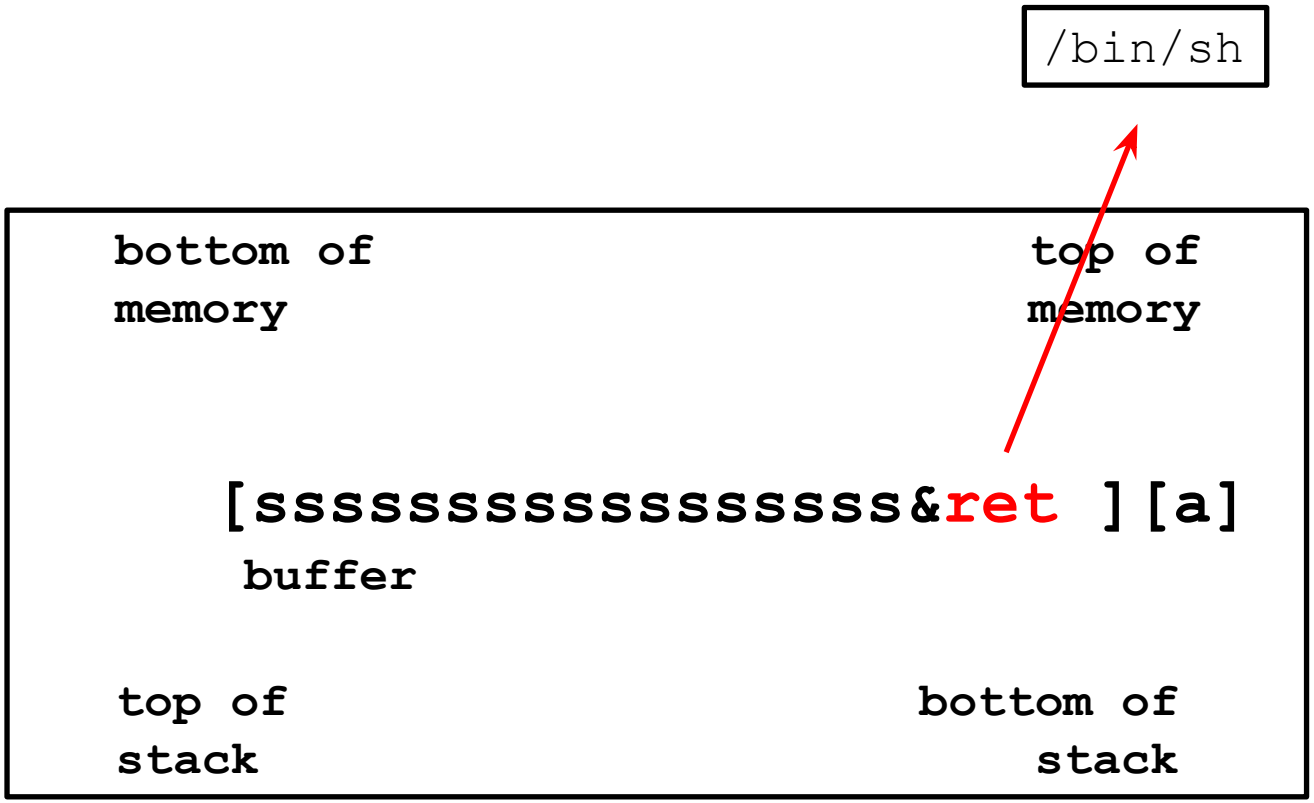
Buffer Overflow Attack (Cont.)



Buffer Overflow Attack (Cont.)



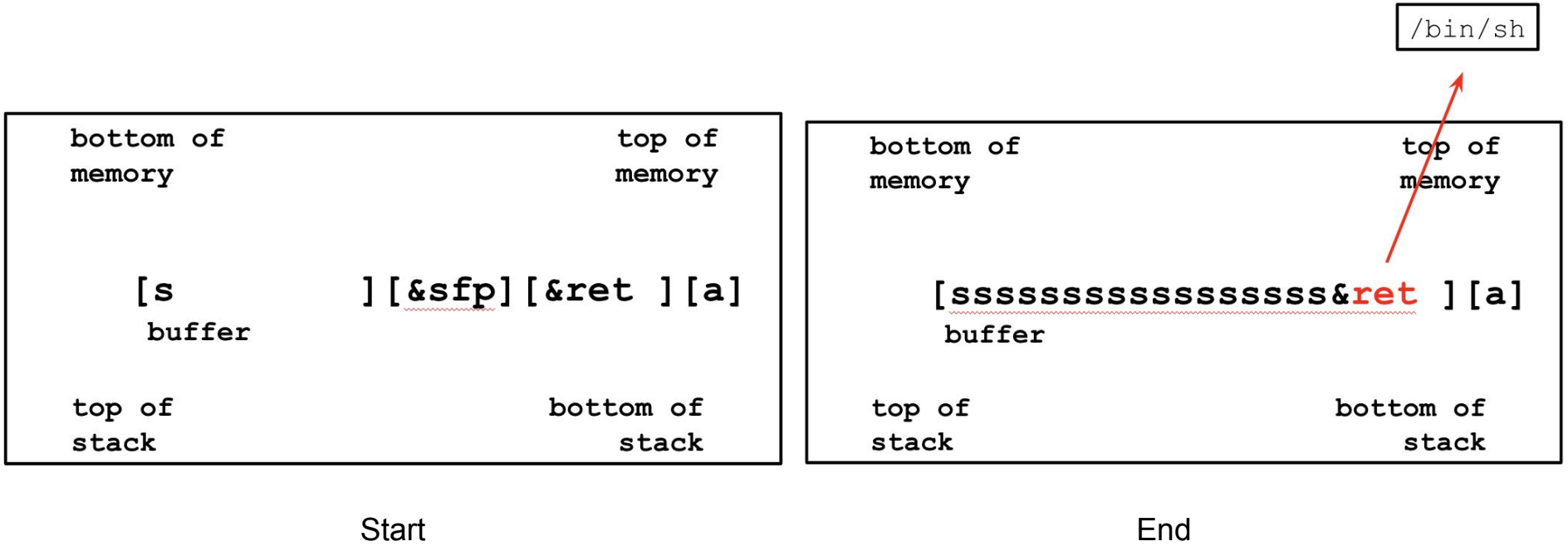
Buffer Overflow Attack (Cont.)



Agenda

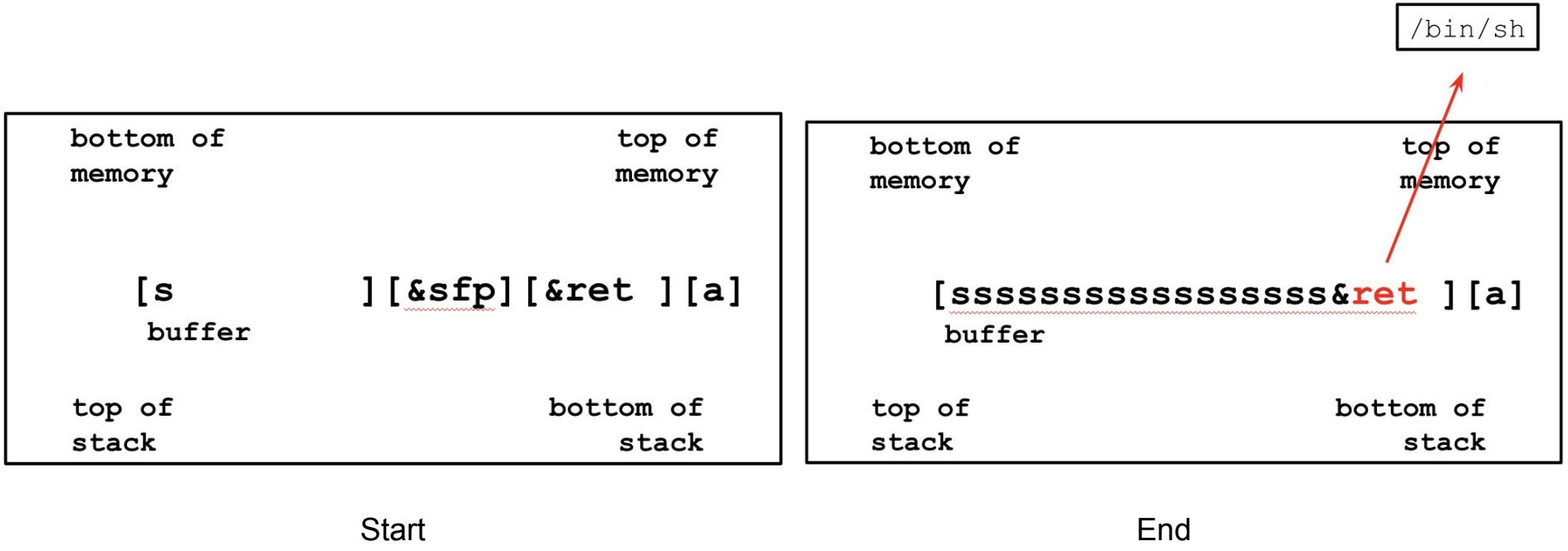
- C language
- Memory Management
- Attacks
- **Countermeasures**
- Final Remarks

Buffer Overflow Attack



Q: Given the layouts before and after a BOF attack, do you have any ideas on how to solve this attack?

Buffer Overflow Attack

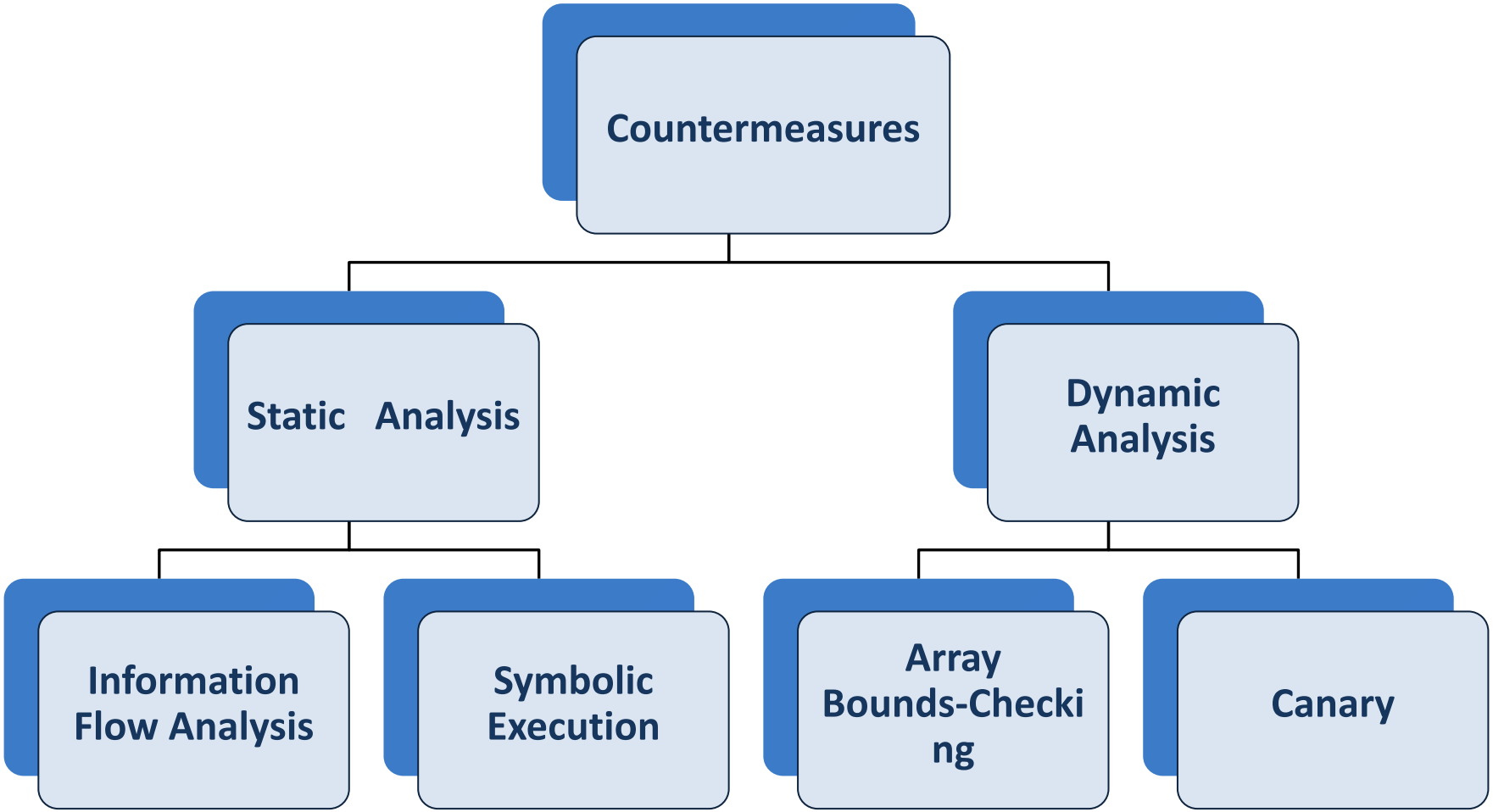


Start

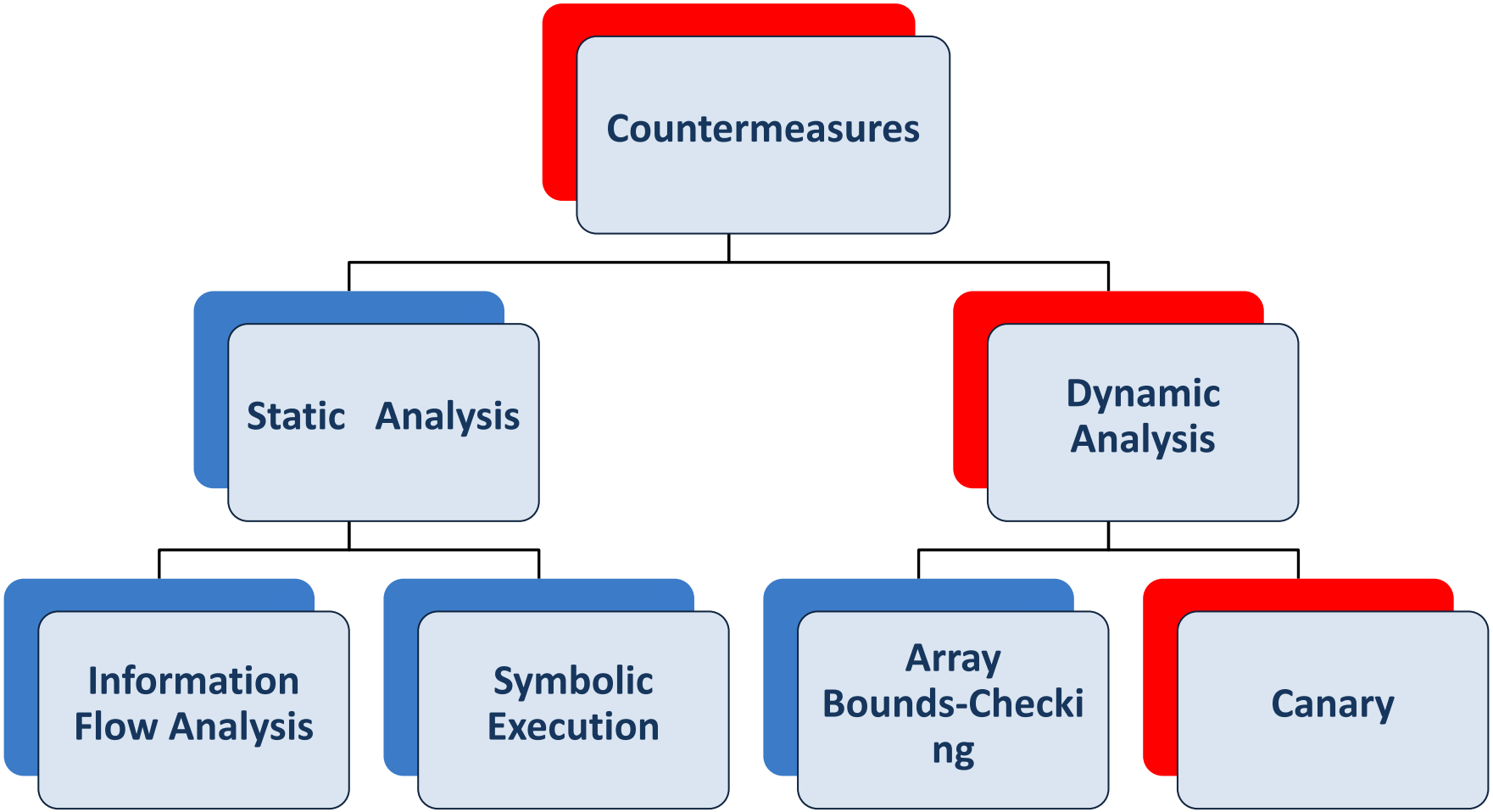
End

Q: Given the layouts before and after a BOF attack, do you have any ideas on how to solve this attack?
A: There are many, but we will look in to Canaries

Countermeasures Taxonomy



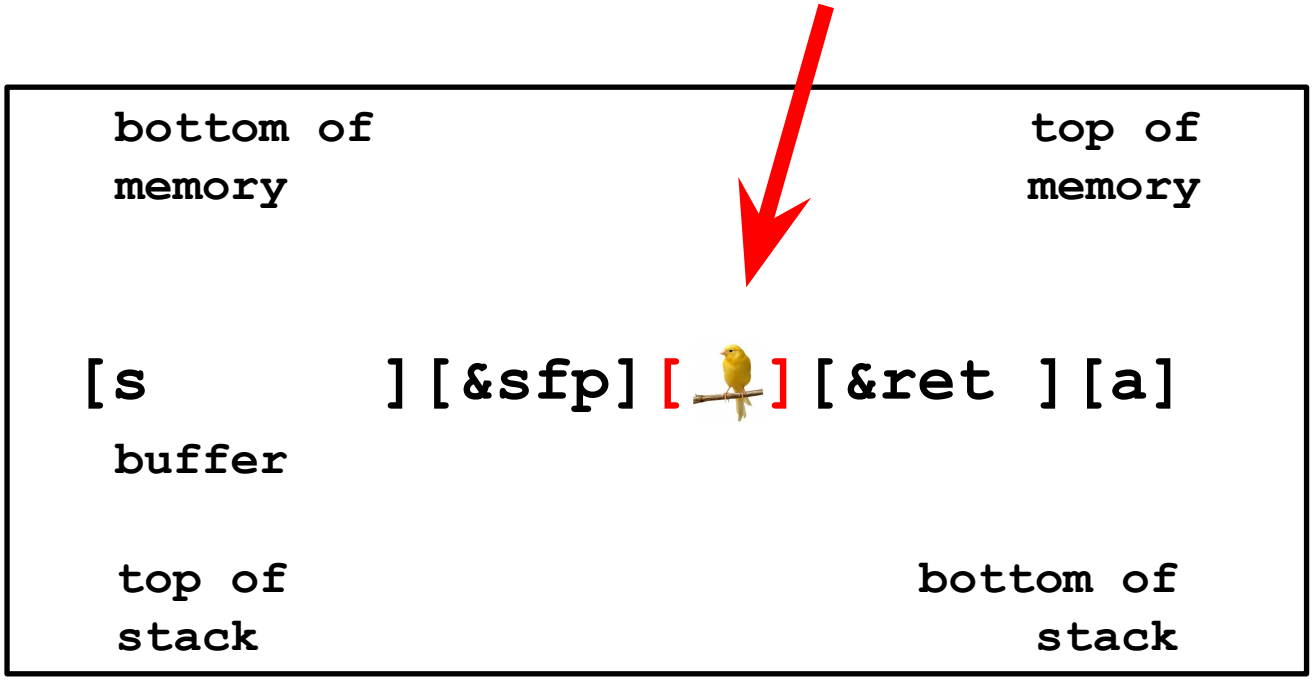
Countermeasures Taxonomy



Canary-based Protection

- Canary-based protection is one of the main protections against stack smashing
  - Proposed by Cowan et al. 1998
  - Like real canaries that detect dangerous gases in coal mines, canaries detects stack-smashing
- The idea is to place a *canary* just below the return address (RET)
  - Therefore, if RET is overwritten so is the canary
  - An an alarm will then go off

Canary-based Protection



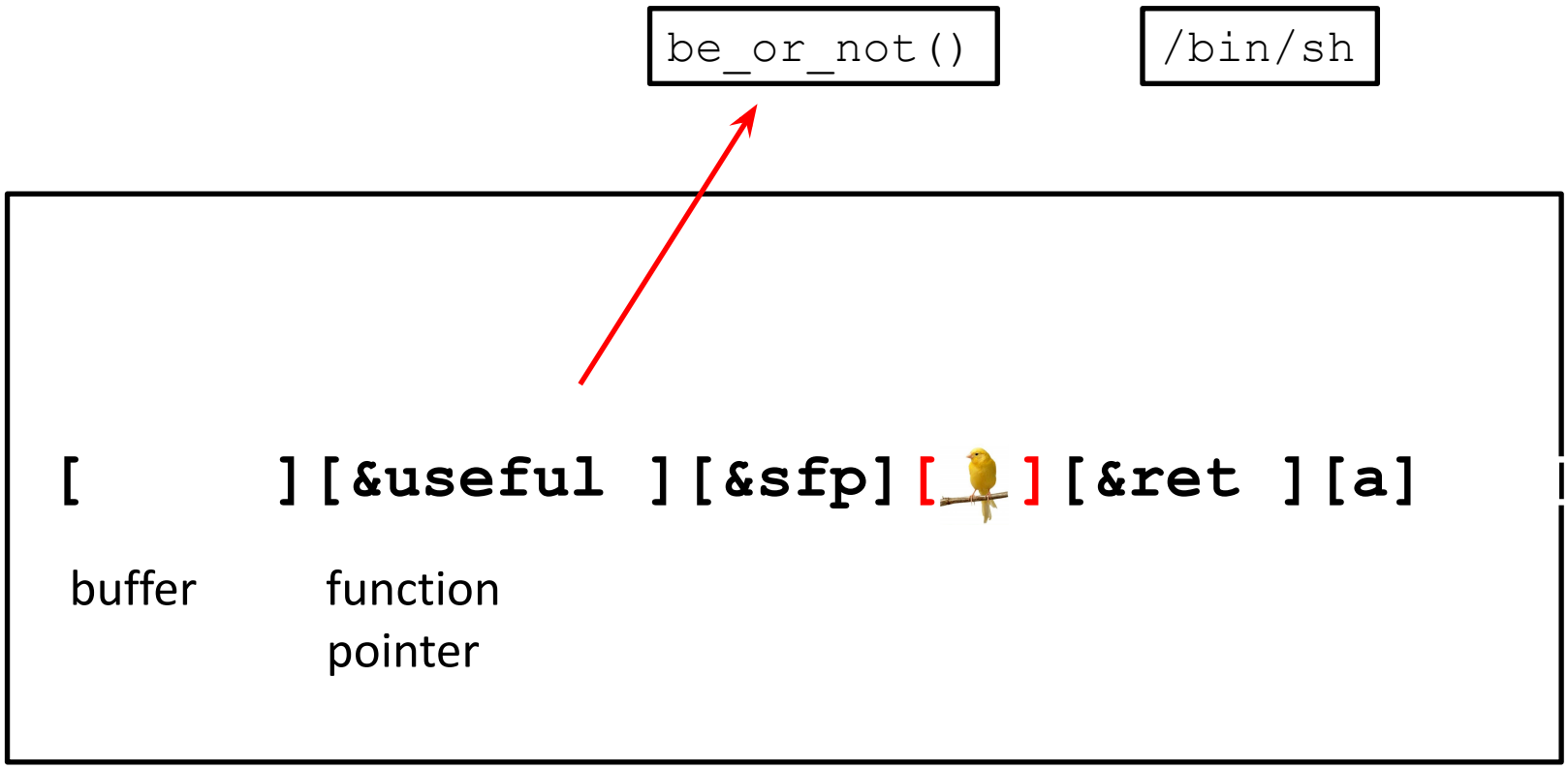
Canary: Pros & Cons

- Computationally cheap

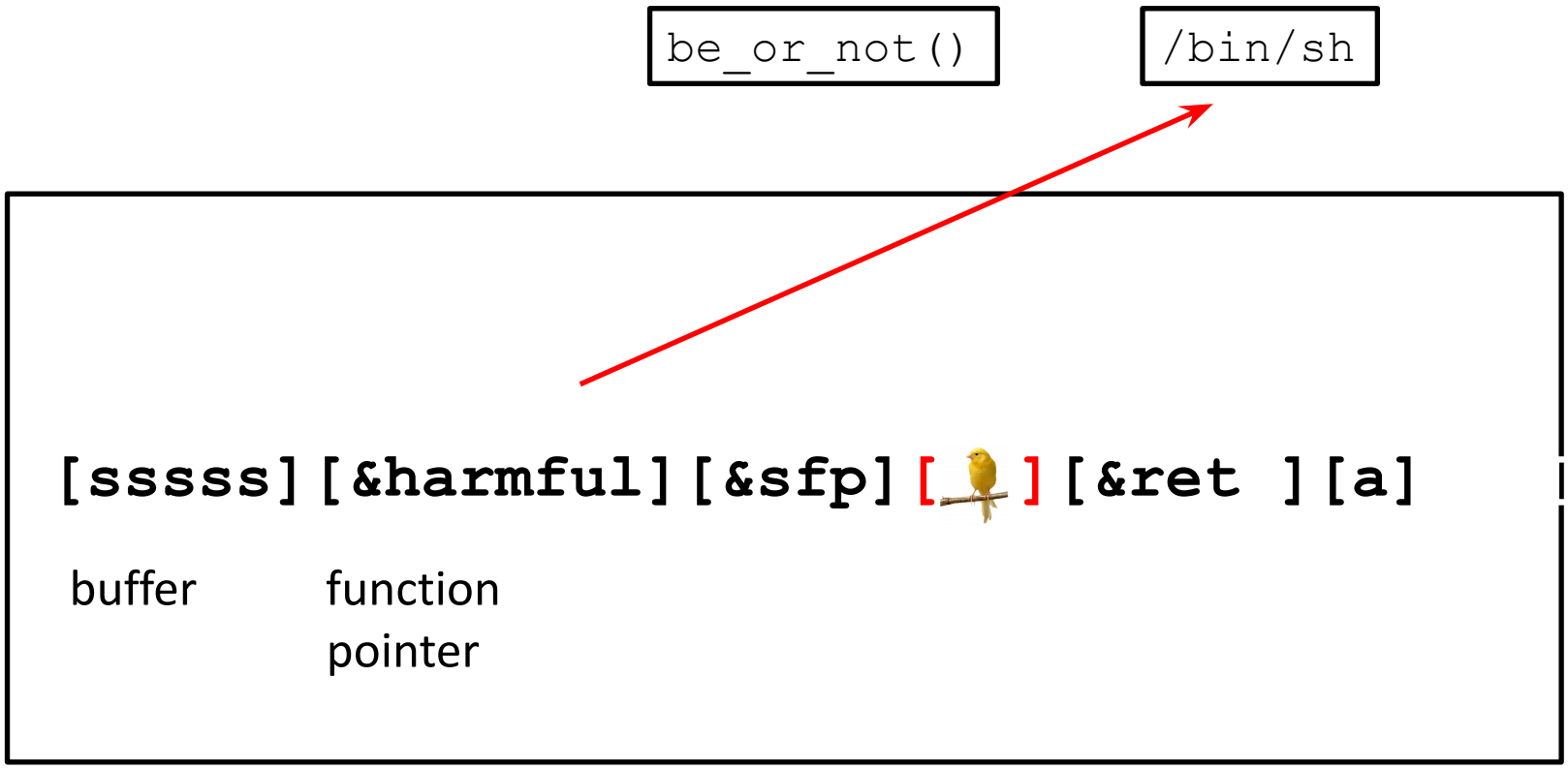
Canary: Pros & Cons

- Computationally cheap
- Problem
  - Current canary-based protection proposals do not guard every kind of stack vulnerability

Canary-based Protected Code Exploitation



Canary-based Protected Code Exploitation



Agenda

- C language
- Memory Management
- Attacks
- Countermeasures
- **Final Remarks**

Final Remarks

- Security is all about getting right the dynamics of a computer
  - It has nothing to do with teenagers sitting in front of their screens and playing games
- Understating the security properties, attacks, and countermeasures will turn you into a security expert
- Security is wide field and the class today aimed to give you only an overview

Final Remarks

- To convince you further about the relevance of this field, I want you students to watch this video
  - Stuxnet

Questions?