Language-Based Security

Contents

1. What is Software Security?
2. Sources of Software Insecurity
3. Pointers and Arrays
4. Type Checking
5. Type safety
6. Call Stacks-Program Function Calls
7. Stack Smashing Attacks
8. Buffer Overflow Attacks

1. What is Software Security?

- **Software security** is distinct from security software like Firewalls, intrusion detection, encryption
- It focuses on protecting the environment in which the software operates

Goal: Software security is to engineer software in a way that ensures it continues to function correctly even when under malicious attacks.

2. Sources of Software Insecurity

a) Complexity, inadequacy, and change.

b) Incorrect or changing assumptions (capabilities, inputs, outputs). Flawed specifications and designs.

c) Poor implementation of software interfaces (input validation, error and exception handling).

d) Inadequate knowledge of secure coding practices.

Sources of Software Insecurity (Cont.)

e) Unintended, unexpected interactions
   i. with other components
   ii. with the software's execution environment

f) Lack of or little consideration of security at all lifecycle stages. Don't think like an attacker

Most of the vulnerabilities in the National Vulnerability Database (https://nvd.nist.gov/) are due to programming errors

Insecure software is prevalent, but:

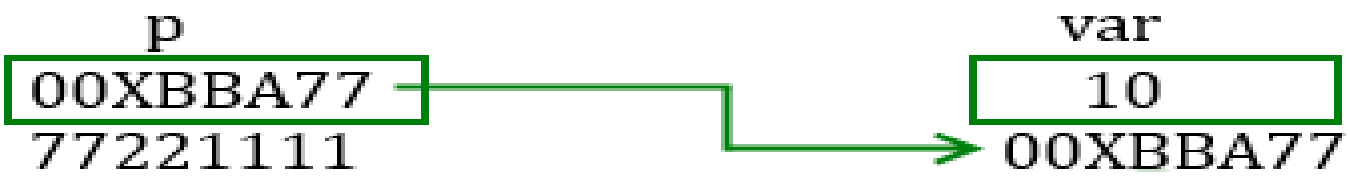
- To what extent are programming languages responsible?
- Are some language features inherently more (or less) secure than others? How can we evaluate the "dangerousness" of a language?
- How can we identify and avoid insecure features?
- How can we enforce software security at the programming level, even when using an insecure language

3. Pointers and Arrays

3.1 Pointer and Reference Types

```
int var =10;
int *p;
p = &var;
```

C - Pointers



P is an pointer here which is pointing to the address of variable var.
Note: Data type for var and p should be the same.

A pointer type variable can hold memory addresses and a special value, NULL

It provides the power of indirect addressing

Pointer Operations

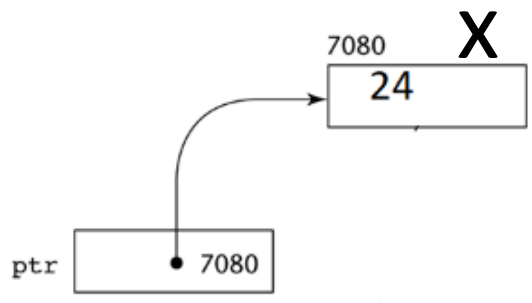
Two fundamental operations:

(1) assignment

(2) dereferencing

The **assignment** is used to set a pointer variable's value to some useful address

```
int x =24;
int *ptr;
ptr = &x;
```



Dereferencing yields the value stored at the location represented by the pointer's value

C++ uses an explicit operation via *

```
int j;
j = *ptr
```

sets j to the value located at `ptr`

Pointers in C: Example

```
main() {
        int x = 10;
        int *p;
        printf("x = %d\n", x);
        p = &x;
        *p = 20;
        printf("x = %d\n", x);
}
```

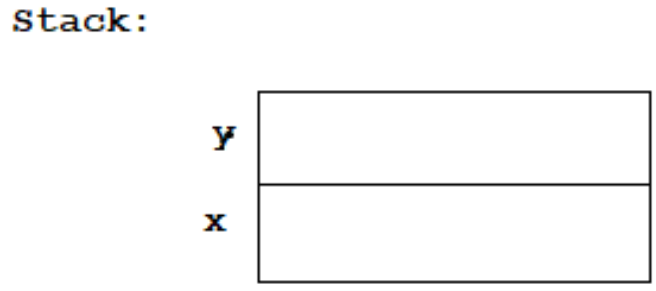
output:

x = 10

x = 20

This leads to pointer arithmetic

```
main() {
        int x, y;
        x = 10;
        printf("x = %d\n", x);
        *((&y) - 1) = 20;
        printf("x = %d\n", x);
}
```



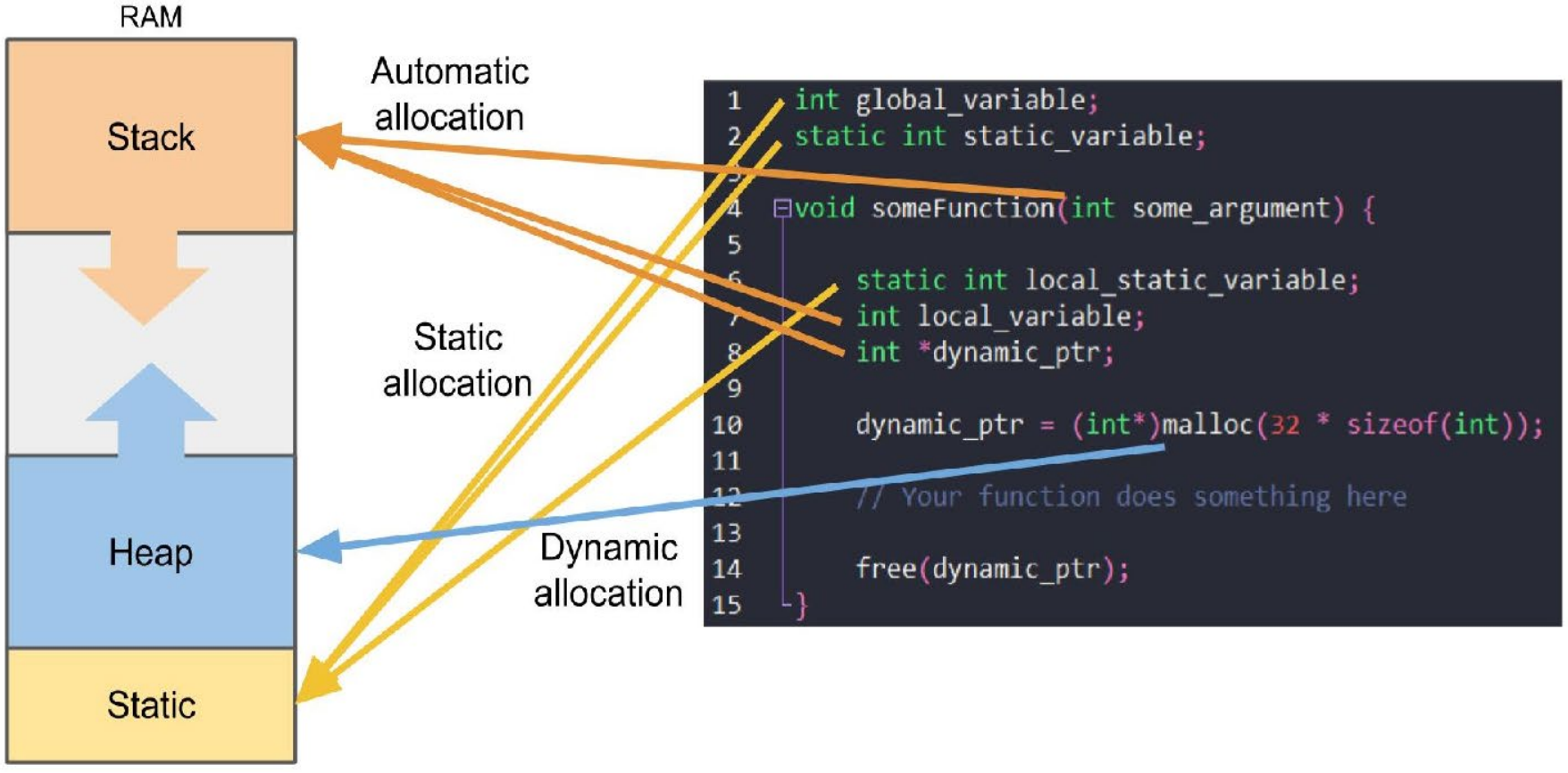
output:

x = 10

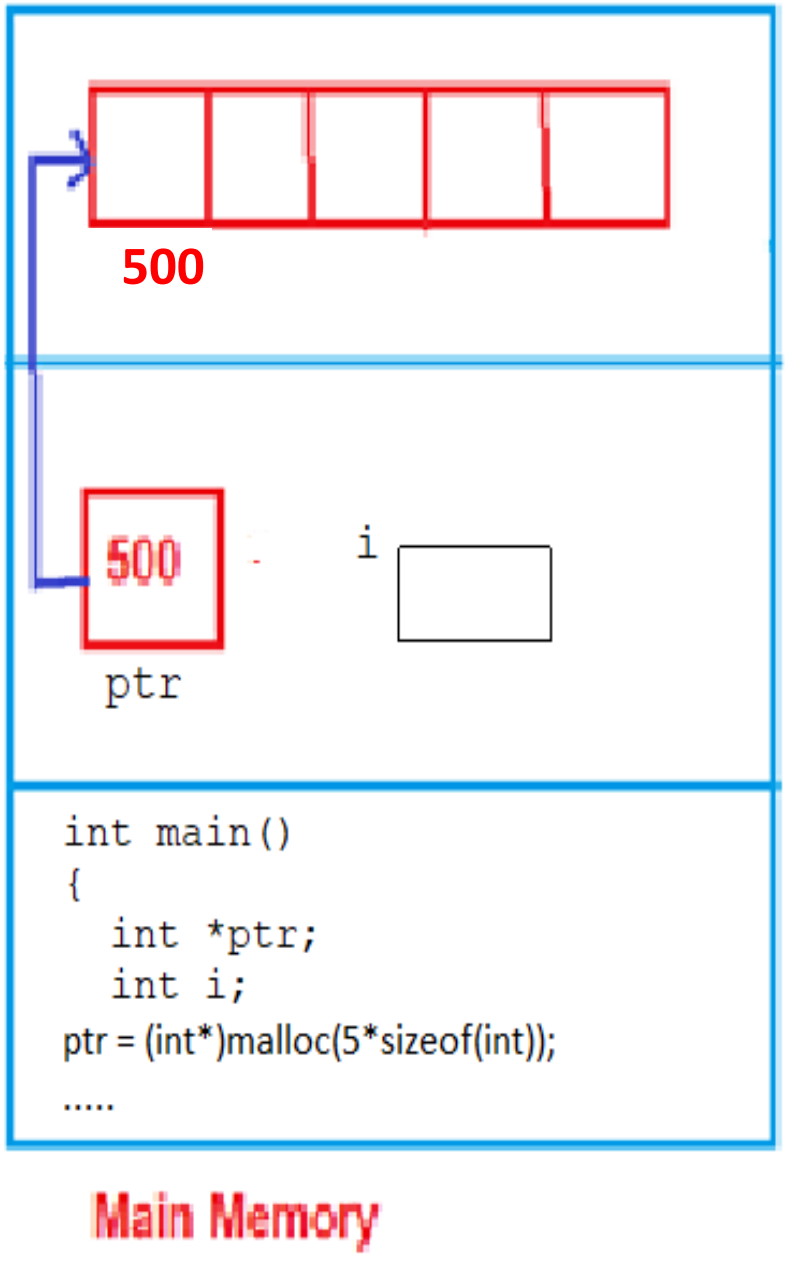
x = 20

C allows access to arbitrary memory locations through pointers

Pointer and Reference Types



- Provide a way to manage dynamic memory, can be used to access a location in the area where storage is dynamically created (usually called a ***heap***)
- Static variables are allocated memory in the data segment, not the stack segment.




```
#include <stdio.h>
int main()
{   int *ptr;
    int i;
    ptr = (int*)malloc(5*sizeof(int));
    for(i=0; i < 5; ++i) {
        *(ptr+i) = 2*i + 1;
    }
    for(i=0; i < 5; ++i) {
        printf("%d\n", *(ptr+i));
    }
}
```

What is the output?

3.2 Problems with Pointers

1. Dangling pointers (dangerous)

```
int *p1 = (int *) malloc(50*sizeof(int));
int *p2;
p2 = p1;
free(p2);
```

- A pointer points to a heap-dynamic variable that has been deallocated

2. Lost heap-dynamic variable

```
int *p1 = (int *) malloc(50*sizeof(int));
p1 = (int *) malloc(100 *sizeof(int));
```

An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

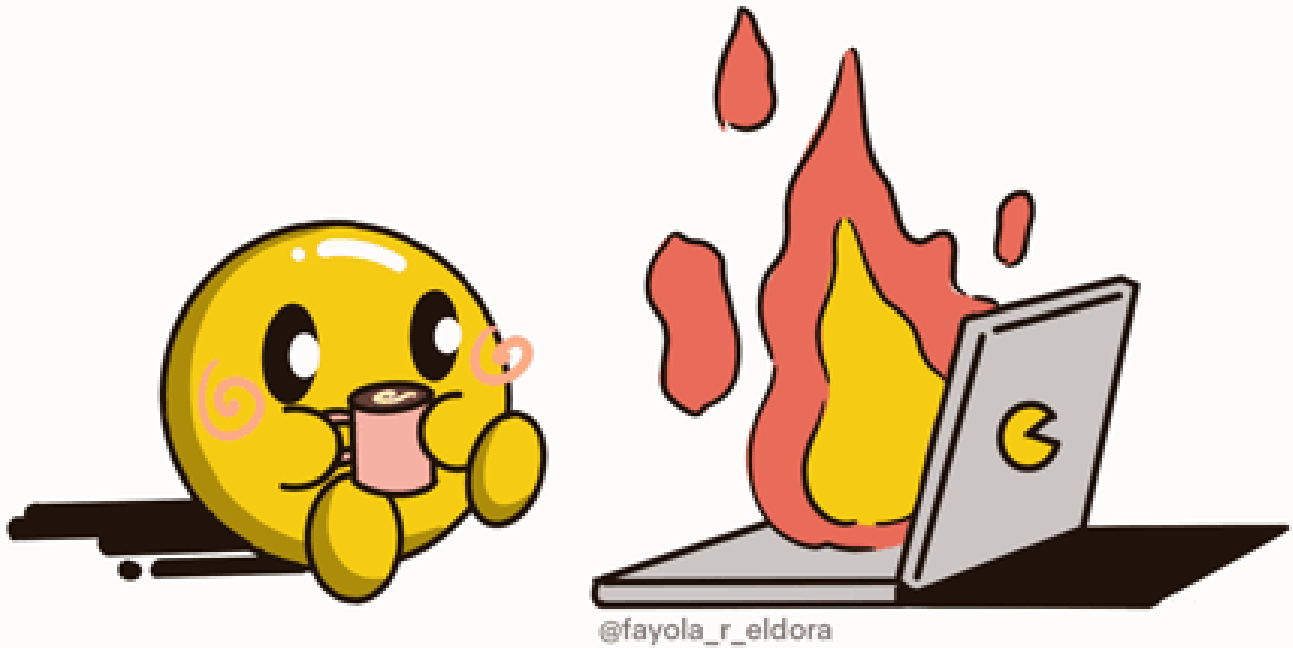
- Pointer `p1` is set to point to a newly created heap-dynamic variable
- Pointer `p1` is later set to point to another newly created heap-dynamic variable
- The process of losing heap-dynamic variables is called ***memory leakage***

3.3 Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`
`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

➔ **C/C++ language allows access to arbitrary memory locations through the improper use of pointers.**


@fayola_r_eldora

Example: Arrays out of bound




```
int main(int argc, char** argv) {
    int a1[1] = {42};          // two 1-element arrays
    int a2[1] = {78};
    a2[1] = 999;               // out of bounds!
    printf("%d\n", a1[0]);     // 999 !!
}
```

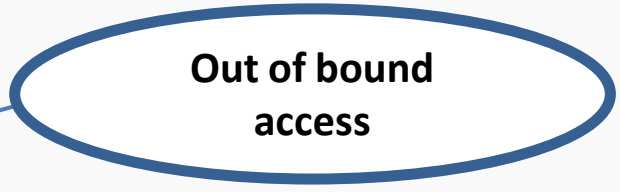
- C does not check array bounds. If you go past the end of the array, you write into the next piece of memory.
  - In this case, that memory happens to refer to a1...
  - can lead to corrupt data elsewhere or crashes ("segfaults")
  - many security bugs, viruses, etc. are from OS/app C code that mistakenly goes past the end of an array on certain input

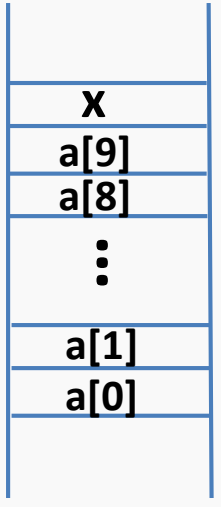
Enough ingredients for errors introduced by careless programmers:

```
main()
{
   int x, a[10], i;

   x = 10;
   printf("x = %d\n", x);
   for (i=0; i <= 15; i++)
          a[i] = 20;
   printf("x = %d\n", x);

}
```





Output:

x = 10

X = 20

Out of bound access in array a, leading to modification of the value of x.

No checks enforced by the C compiler!

Compare with Java → a strongly typed language

```
public class Array1 {
  public static void main (String args []) {
    int x, a [] = new int[10], i;

    x = 10;
    System.out.println ("x=" + x);
    for (i=0; i<=15; i++) a[i]=20;
    System.out.println ("x=" + x);
  }
}
```

x=10

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at Array1.main(Array1.java:7)

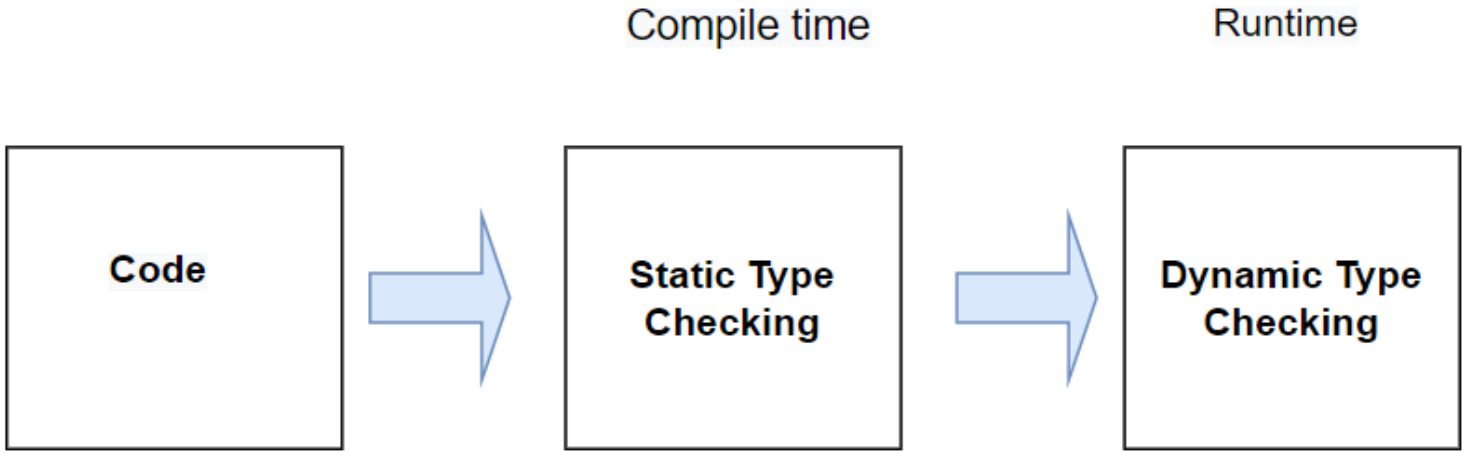
4. Type Checking

4.1 Type Checking

```
double x=12.5;
double y= 8.3;
int z = x + y;
printf("z = %d\n", z);
```

- ***Type checking*** is the activity of ensuring that the operands of an operator are of compatible types
- A ***compatible type*** is one that is either legal for the operator or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type
  - This automatic conversion is called ***coercion***.
- A ***type error*** is the application of an operator to an operand of an inappropriate type

4.2 Classification of type systems



Static type checking: Type checking is done at compile time. Java, C, C++, C#, ML, Haskell, Go, Scala

Dynamic type checking: Type checking is done at run time. Scheme, JavaScript, Ruby, PHP, Perl, Python

Type Binding

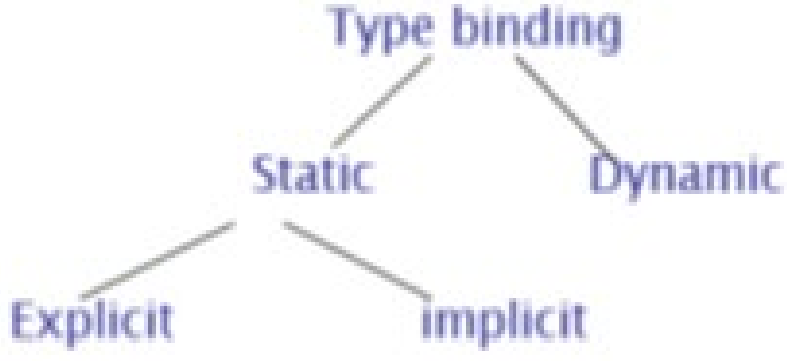
How is a type specified?

When does the binding take place?

a) **Static** type binding, the type can be specified by either

- Explicit declaration, or
- Implicit declaration

b) **Dynamic** type binding



Type Binding (continue)
Static Explicit/Implicit Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables

```
int x;
```

- An *implicit declaration* is a default mechanism for specifying types of variables through default conventions, rather than declaration statements
- Basic, Perl, Ruby, JavaScript, and PHP provide implicit declarations
  - Advantage: writability (a minor convenience)
  - Disadvantage: reliability (less trouble with Perl)

Static Implicit Declaration

Some languages use type inferencing to determine types of variables (context)

- C#: a variable can be declared with var and an initial value. The initial value sets the type

```
var sum=0;
var total=0.0;
var name="Fred";
```

Visual Basic 9.0+, ML, Haskel, and F# use type inferencing. The context of the appearance of a variable determines its type.

Consider a function

function Foo (a, b) = a + b, a and b are declared as integer

The compiler knows that + has type

Integer + Integer → Integer

Which means + takes two Integers and returns another Integer. It can infer that Foo also has type Integer

Dynamic Type Binding

- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited))
- Specified through an assignment statement when it is executed

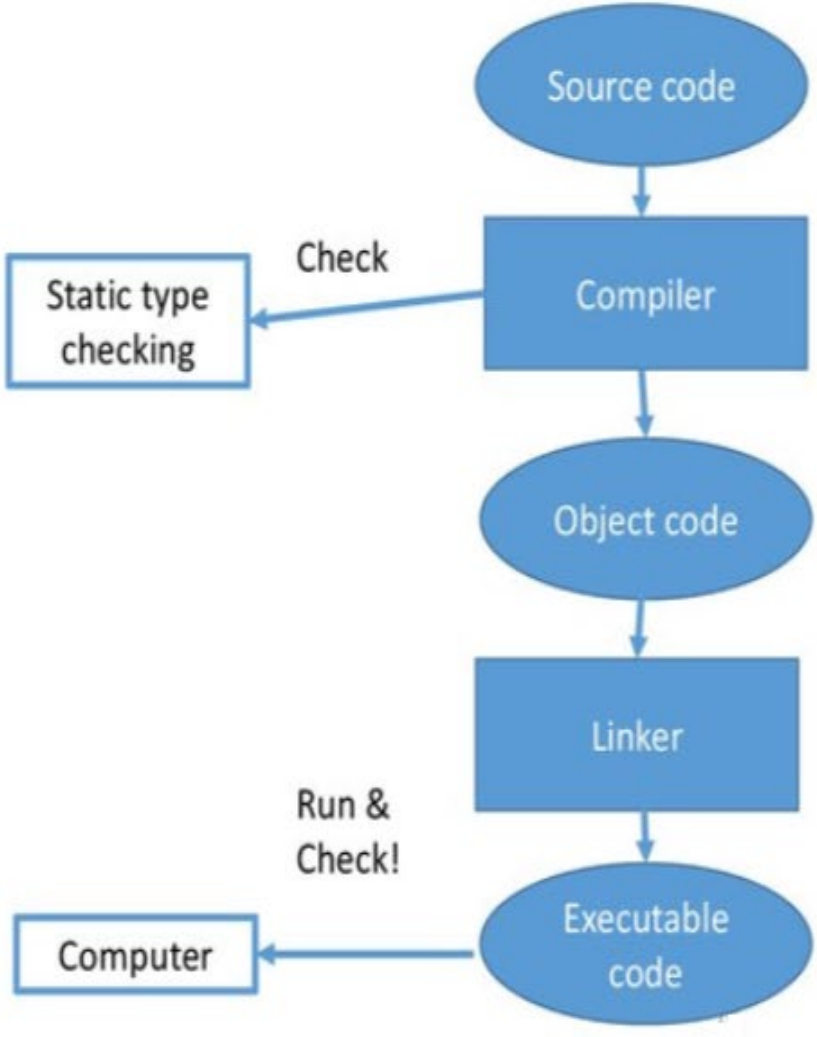
  e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult

Static and dynamic type checking

- JavaScript, Python, … dynamic type checking.
  - "easy" to code
  - Harder to debug
  - Slower code execution speed
- Java, C#, C, C++,Eiffel , … static type checking
  - "harder" to code and compiler
  - Less runtime errors
  - No extra checks at runtime
  - More abilities to optimize the output code



4.3 Strong Typing

- A programming language is **strongly typed** if it enforces strict rules about how types can be used and interchanged.
  - ✓ Strict Type Checking
  - ✓ Compile-Time or Runtime Enforcement
  - ✓ Error Prevention
  - ✓ E.g., Java, Python
- Advantages:
  - ✓ Type safety
  - ✓ Readability and maintainability
- Disadvantages:
  - ✓ Less flexibility
  - ✓ Potentially more boilerplate

Strong Typing (cont.)

Language examples:

- C and C++ are not strongly typed
  - ✓ Function parameter not type checked
  - ✓ Union do not provide type checking
- Java and C# are strongly typed (due to explicit type casting)

Example: Union in C

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data d;

   d.i = 10;
   d.f = 220.5;
   strcpy( d.str, "C Programming");

   printf( "d.i : %d\n", d.i);
   printf( "d.f : %f\n", d.f);
   printf( "d.str : %s\n", d.str);

   return 0;
}
```

When this code is compiled and executed, it produces the following result:

d.i : 1917853763
d.f : 412236058032779486045275994368.000000
d.str : C Programming

5. Type safety

5.1 Type Safety

- A programming language is type safe if the only operations performed on data are those allowed by the data's type.
- Type checking can be done at either compile time or run time.
- C/C++ are not considered type safe.
- Java and Standard ML (SML) are considered type safe.

5.2 Type safety, "strong" vs. "weak"

Type safety: Degree to which a language prevents type errors.

Strong typing, weak typing: Whether the language has severe or relaxed restrictions on what operations allow types to mix (what implicit type conversions are allowed).

string + int ?
int * double ?
string = char ?

Lack of type safety in C

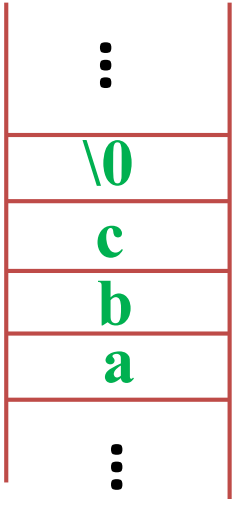
```
int main(int argc, char** argv) {
    char msg[4] = "abc";          // [97, 98, 99, 0]
    int* num = (int*) msg;

    printf("%s\n",  msg);         // abc
    printf("%d\n", *num);         // 6513249
    (*num)++;                     // (97 + 98*2^8 + 99*2^16 + 0*2^24)

    printf("%d\n", *num);         // 6513250
    printf("%s\n",  msg);         // bbc
}
```

- The code allows a string (`char[]`) to be interpreted as though it were an `int`! This is unsafe.
- C is the poster child for unsafe languages...

\0	c	b	a	
0	99	98	97	
00000000	01100011	01100010	01100001	➔ **6513249**



5.3 Safe Languages

There are typed languages that are not safe ("weakly typed languages")

- All safe languages use types (static or dynamic)

	Typed		Untyped
	Static	Dynamic	
Safe	ML, Java, Ada, C#, Haskell, ...	Lisp, Scheme, Ruby, Perl, Smalltalk, PHP, Python, ...	λ-calculus
Unsafe	C, C++, Pascal, ...	?	Assembly

6. Call Stacks

Program Function Calls

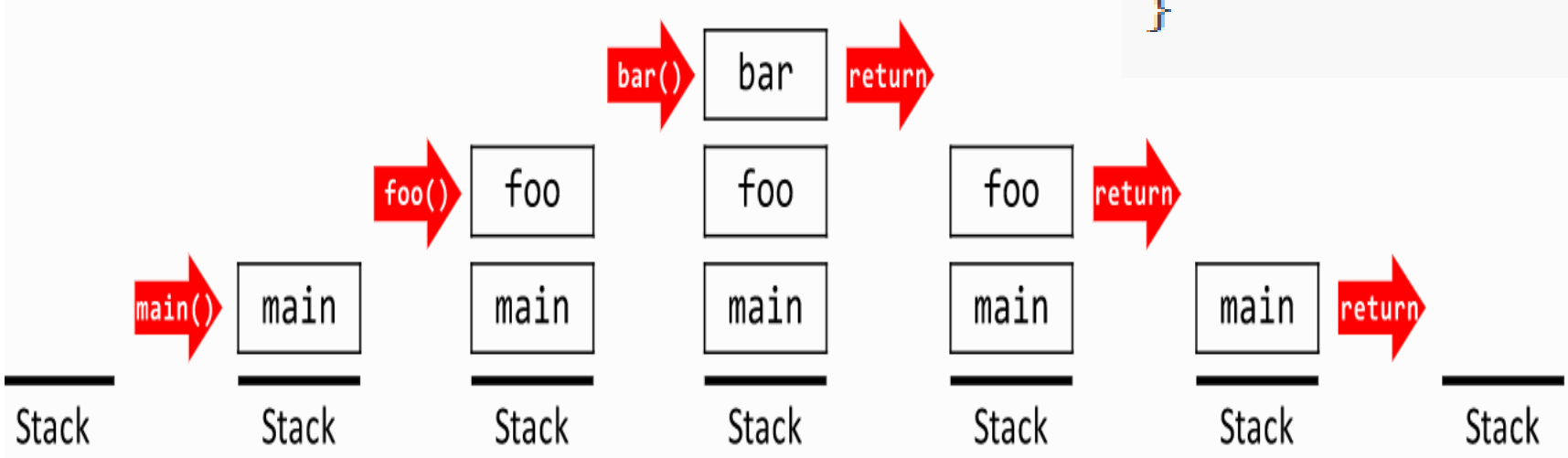
6.1 Implementing "Functions Calls"

1. Save the execution status of the caller
2. Pass parameters
3. Pass the return address to the callee
4. Transfer control to the callee

```
void bar() {
}

void foo() {
  bar();
}

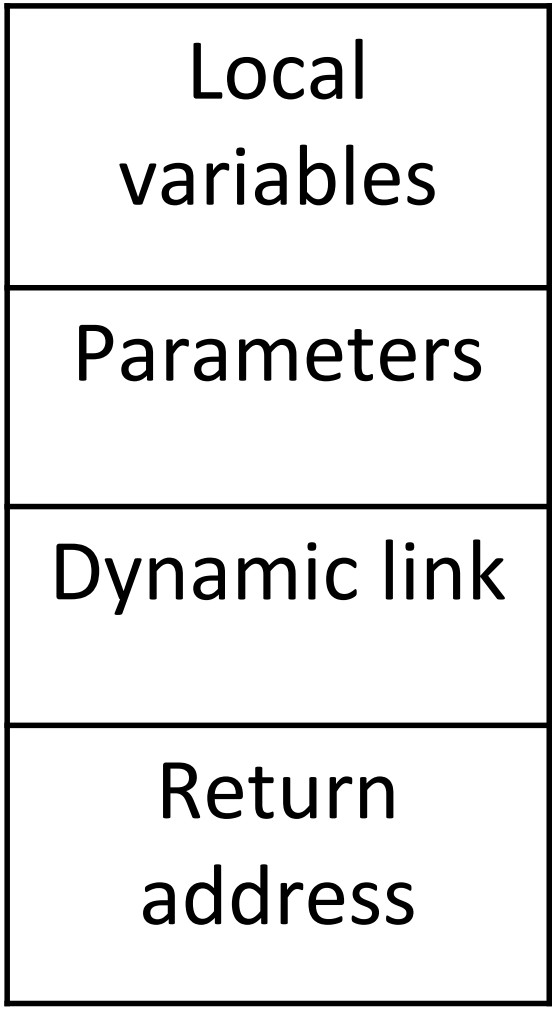
int main() {
  foo();
}
```



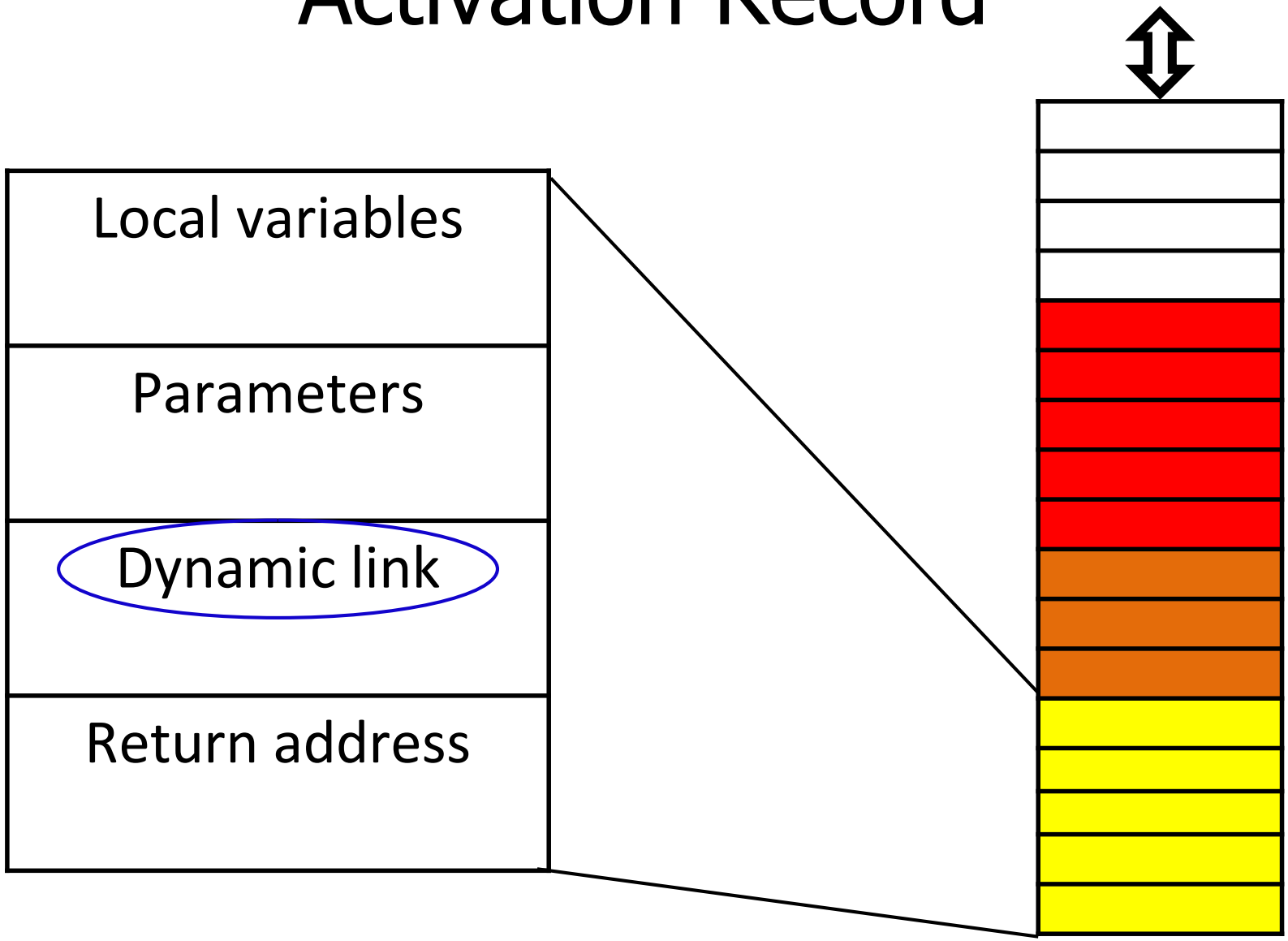
A stack that stores activation records.

Function Activation

- Layout for the required storage
  - **Activation record**
- **Activation record instance (ARI)**
  - activation record for a particular function activation



Activation Record

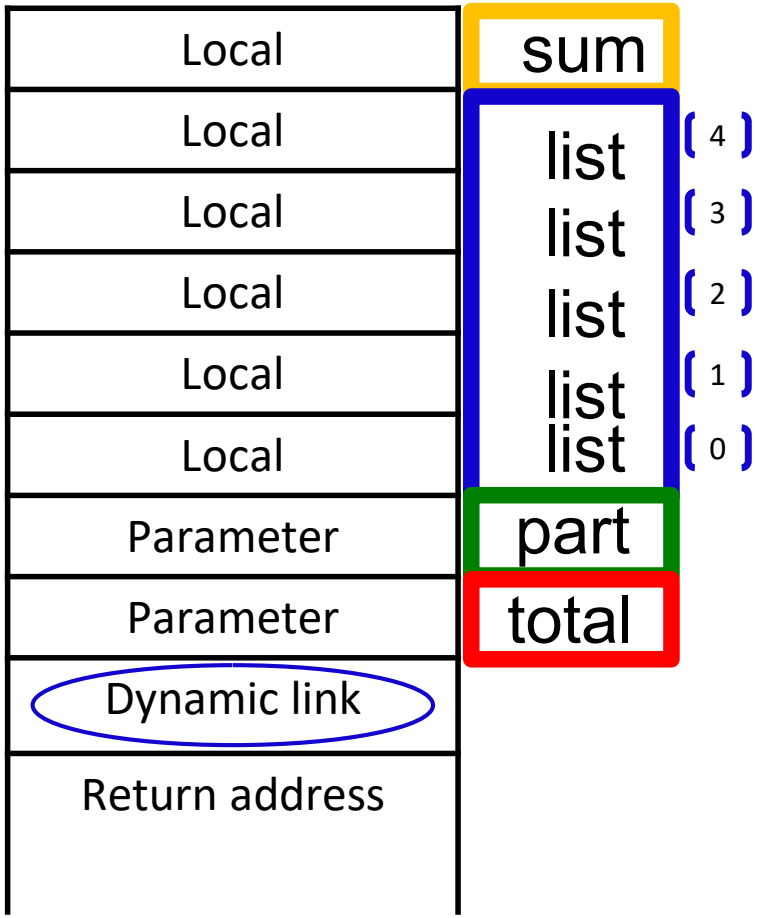


Dynamic link

- points to the top of the ARI of the caller

ARI Example for a C Function

```
void sub(float total, int part) {
        int list[5];
        float sum;
        …
}
```

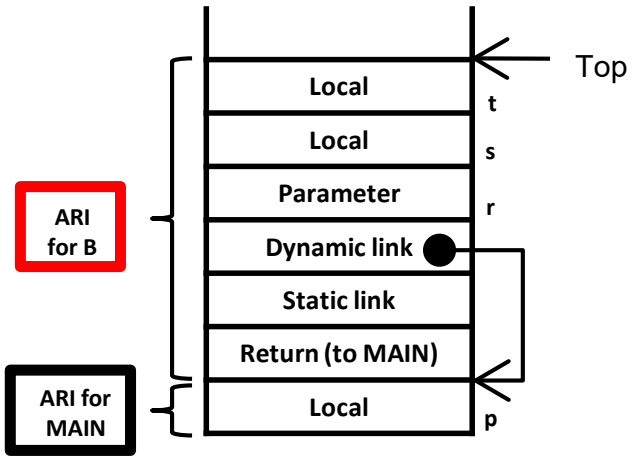


Example 1: Call Stack of C Functions (no recursion)

```
void A(int x) {
        int y;
        ...
        C(y);
        ...
}
void B(float r) {
        int s, t;
        ...
        A(s);
        ...
}
void C(int q) {
        ...
}
void main() {
        float p;
        ...
        B(p);
        ...
}
```

Stack after MAIN calls B





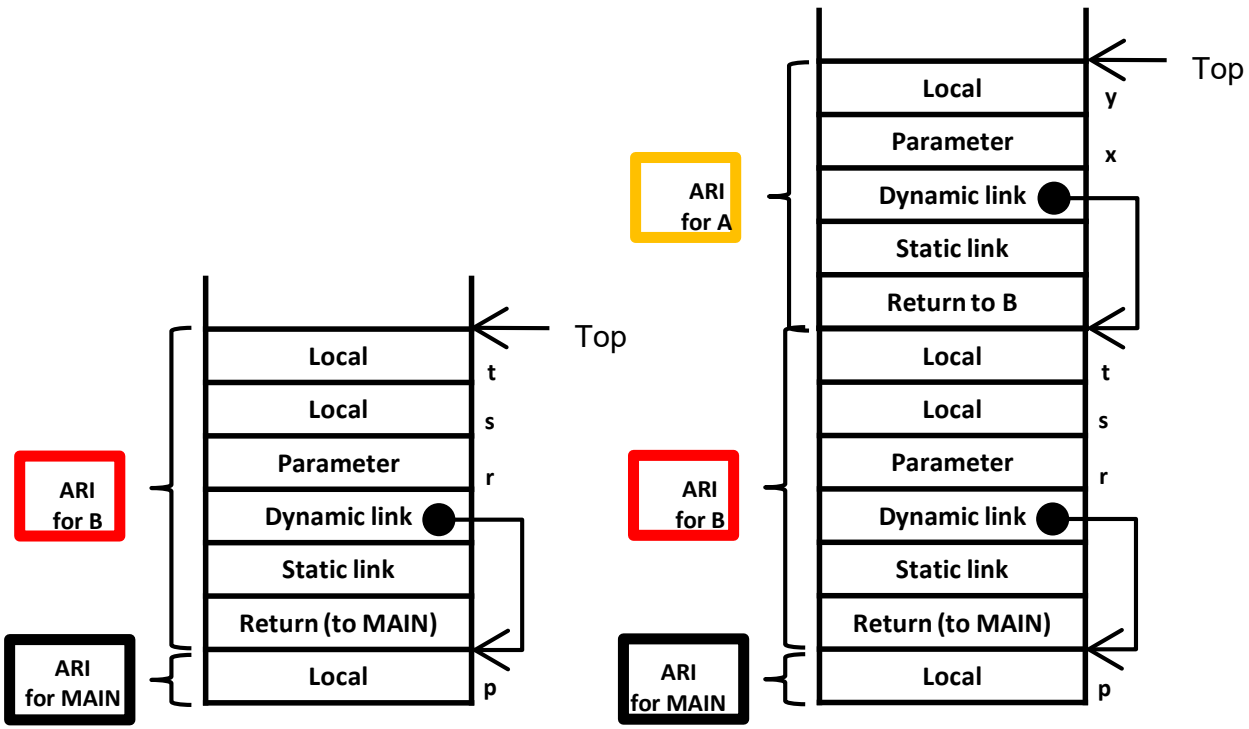

```
void main()
{
  float p;
...
  B(p);...
}
```

```
void B(float r)
{
  int s, t; ...
  A(s); ...
}
```

Static link: In languages that allow nested functions, functions must access the outer function's stack frame.

Stack after B calls A

main calls B calls A




```
void main()
{
  float p;
...
  B(p);...
}
```

```
void B(float r)
{
  int s, t; ...
  A(s); ...
}
```

```
void A(int x) {
  int y; ...
  C(y); ...
}
```

Stack after A calls C




```
void main()
{
  float p;
...
  B(p);...
}
```

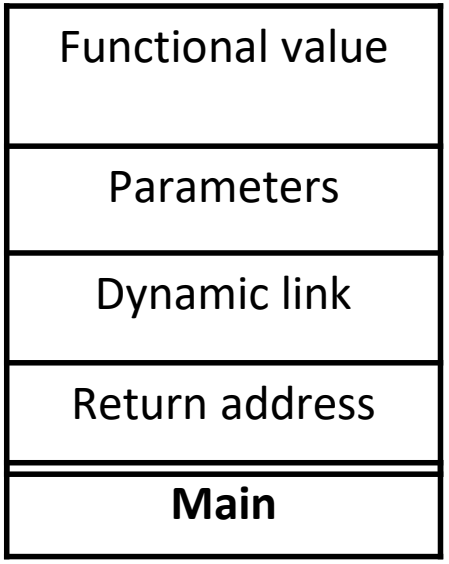
```
void B(float r)
{
  int s, t; ...
  A(s); ...
}
```

```
void A(int x)
{
  int y; ...
  C(y); ...
}
```

```
void C(int q)
{
  ...
}
```

Example 2: Call Stack Activation Records in Recursion

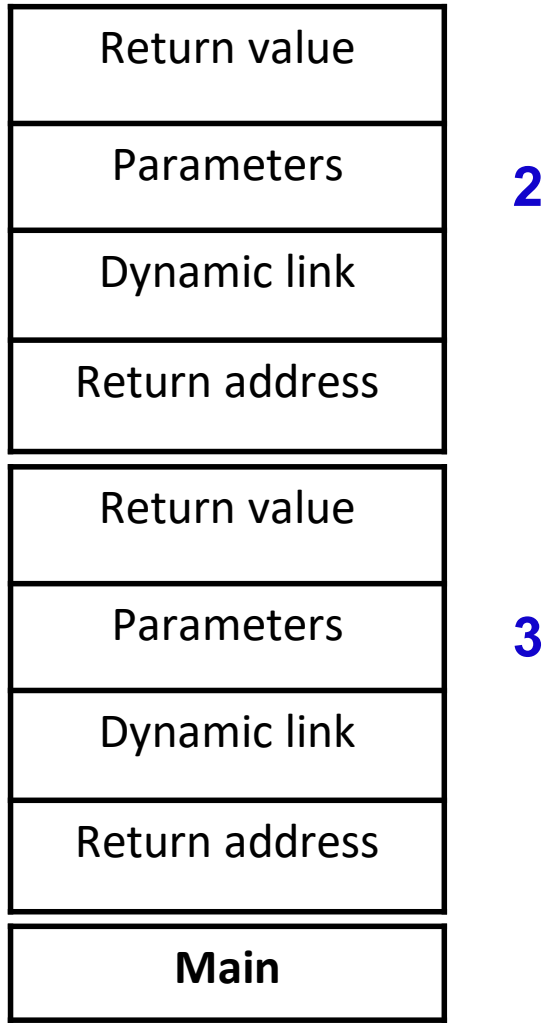
```
int factorial(int n) {
  if (n <= 1)
    return 1
  else
    return (n * factorial(n-1));
}
void main() {
        int value;
        value = factorial(3);
}
```



3

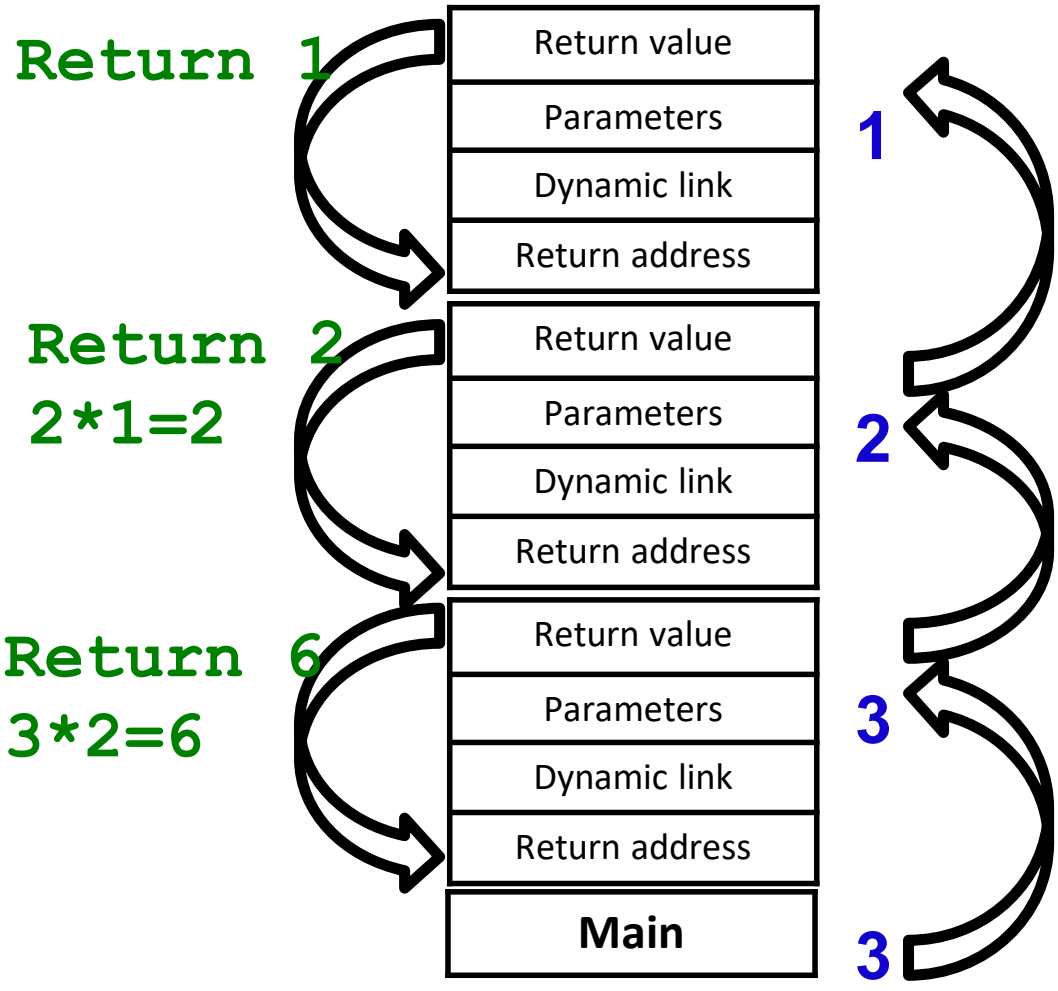
Example 2: Call Stack Activation Records for factorial

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
void main() {
        int value;
        value = factorial(3);
}
```



Example 2: Call Stack Activation Records for factorial

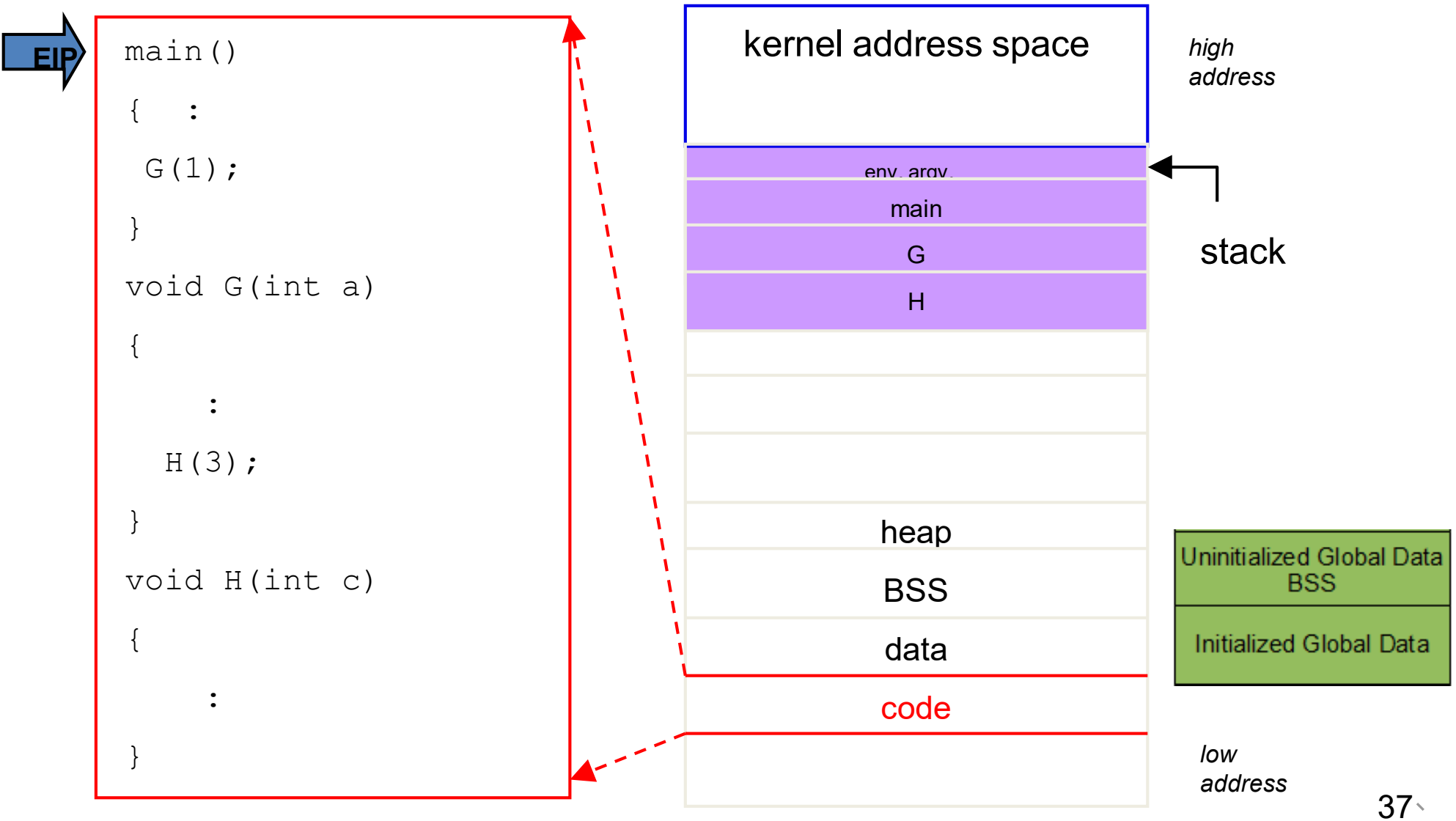
```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
void main() {
          int value;
          value=factorial(3);
}
```



7. Stack Smashing Attacks

- Stack smashing attack: a.k.a. **stack overflow attack** or **buffer overflow attack**, is a type of security vulnerability in computer software or applications.
- It occurs when a program writes more data to a block of memory, or buffer, than it was allocated for.
- In most programming languages, especially C and C++, the stack is a region of memory where functions store local variables, function parameters, and control flow data.

A Linux Process Layout and Stack Operations



How a stack smashing attack typically works

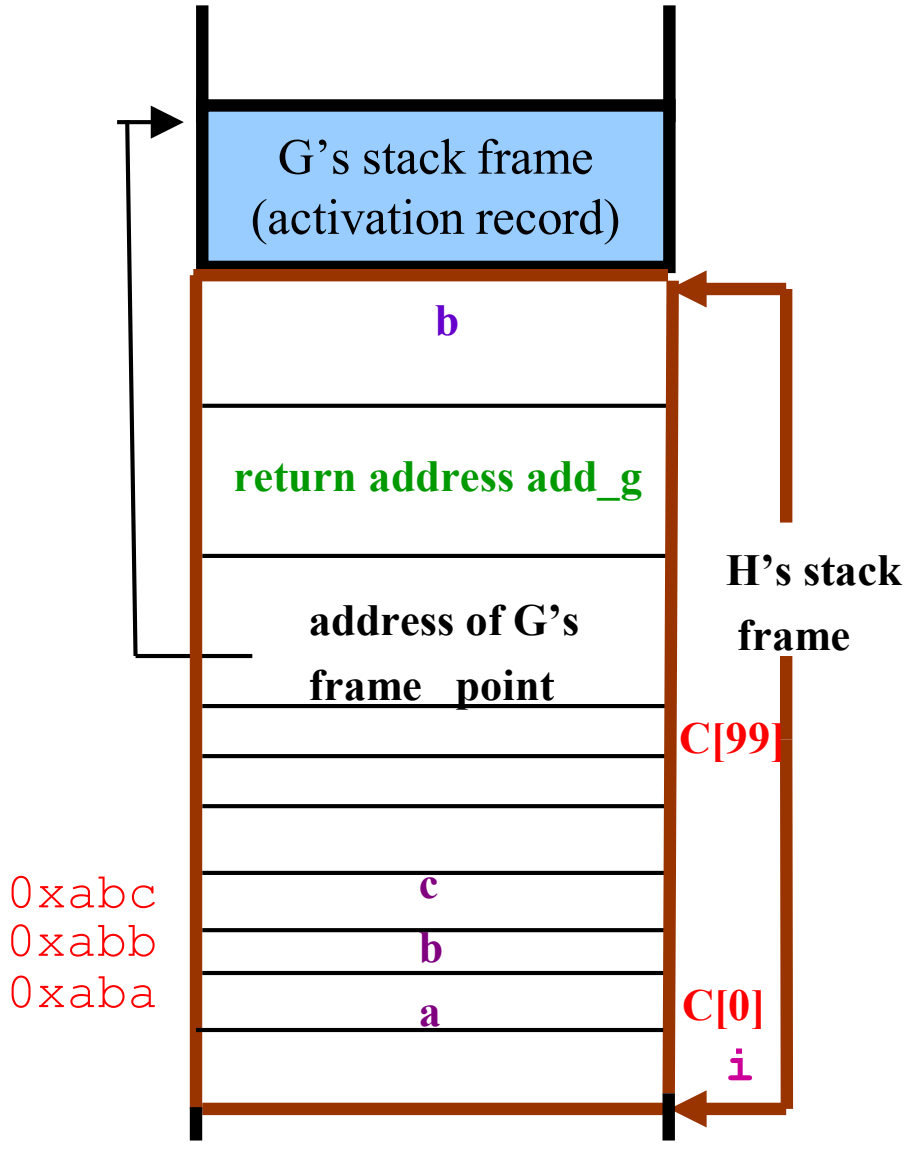
- **Buffer Overflow**: An attacker sends more data than a buffer can hold, causing the excess data to overflow into adjacent memory locations, often including the stack.
- **Overwriting Control Data**: By overflowing a buffer, an attacker can overwrite critical control data, such as function return addresses or instruction pointers stored in the stack frame.
- **Manipulating Program Flow**: By manipulating these control data, an attacker can redirect the program's flow of execution to malicious code injected into the compromised buffer.
- **Execution of Arbitrary Code**: Once the attacker controls the program flow, they can execute arbitrary code, leading to unauthorized access, privilege escalation, or the injection of malware.

An Example

```
G(int a)
{
  H(3);
add_g: …..
}
H( int b)
{ char c[100];
  int i=0;

  while((c[i++]=getch())!=EOF)
  {
     ……
   }
}
```

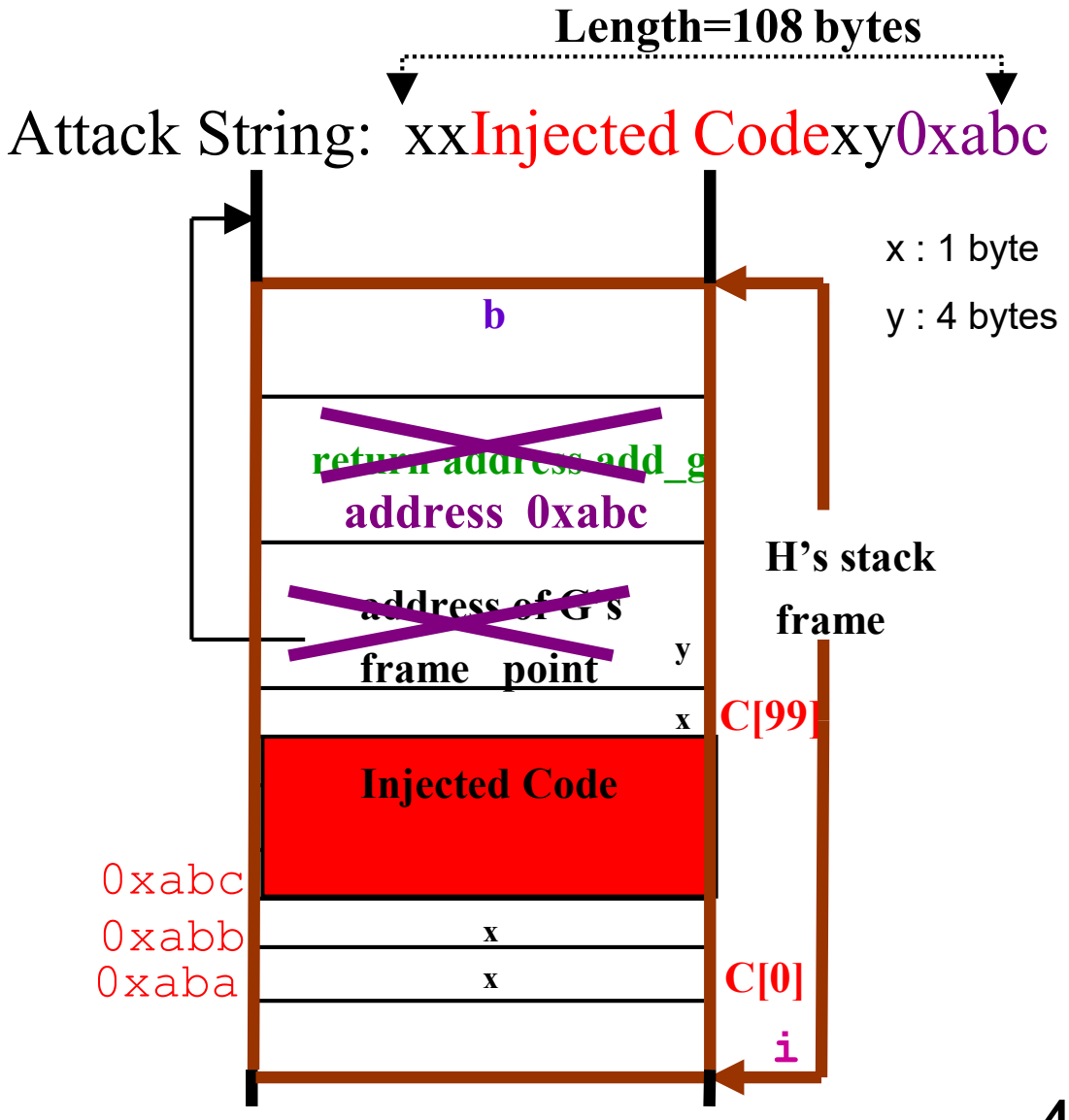
Input String: abc



Example (cont.): Buffer Overflow Attack

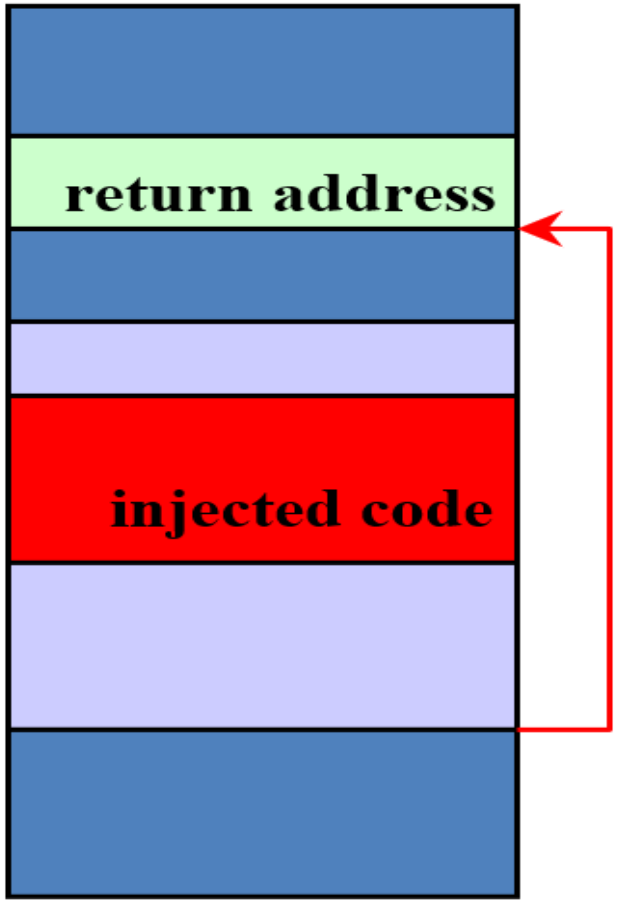
```
G(int a)
{
  H(3);
add_g: …..
}
H( int b)
{ char c[100];
  int i=0;

   while((c[i++]=getch())!=EOF)
   {
      ……
    }
}
```



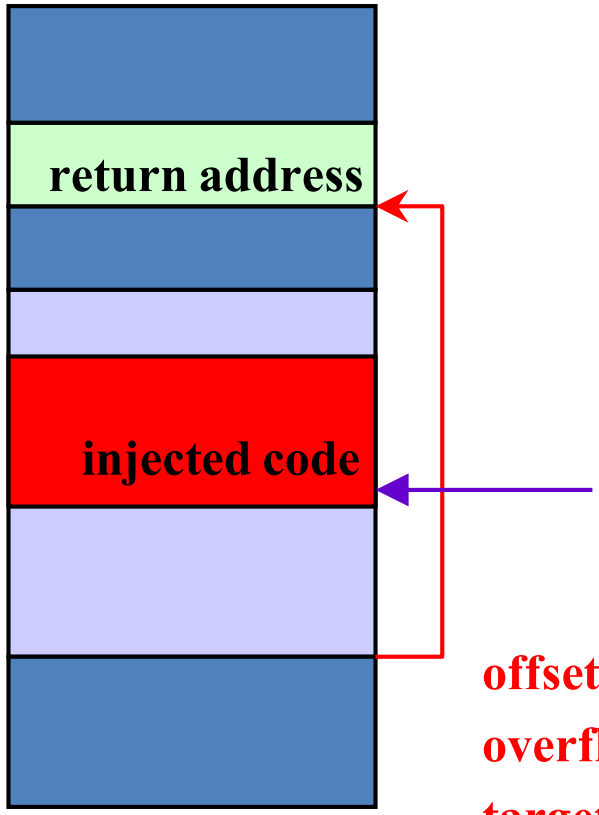
A Successful Buffer Overflow Attack

A successful buffer overflow-style attack should be able to overflow the *right place* (e.g., the place to hold a return address)



A Successful Buffer Overflow Attack (cont.)

The **offset** and the entry point **address** are non-predicable. They can not be decided by just looking the source code or binary code.



address of injected code entry point.

offset between the beginning of the overflowed buffer and the overflow target.

Preventing stack smashing attacks: secure coding practices

- **Bounds Checking:** Ensuring that buffers do not overflow by validating the size of incoming data before copying it into buffers.
- **Safe String Functions:** Using secure versions of string manipulation functions (e.g., *strncpy* instead of *strcpy* in C) that allow developers to specify the size of the destination buffer.
- **Address Space Randomization:** Randomly arranging the memory addresses used by system processes and applications, making it harder for attackers to predict the location of critical data structures.
- **Stack Canaries:** Introducing random values, called canaries, between the buffer and control data. If the canary is modified, it indicates a buffer overflow, triggering an alert or terminating the program.

Preventing stack smashing attacks: secure coding practices

- **Non-Executable Stack:** Marking the stack region of memory as non-executable, preventing injected code from executing on the stack.

It's important for developers to be aware of these vulnerabilities and employ secure coding practices to prevent stack smashing attacks and other forms of buffer overflows.

Strings and Buffer Overflows

public enemy number 1

The good news

C is a small language that is close to the hardware

- you can produce highly efficient code
- compiled code runs on raw hardware with minimal infrastructure

C is typically the programming language of choice

- for highly efficient code
- for embedded systems (which have limited capabilities)
- for system software (operating systems, device drivers,...)

The bad news: using C/C++ is dangerous





The Nature of the problem

Suppose in a C program we have an array of length 4

```
char buffer[4];
```

What happens if we execute the statement below ?

```
buffer[4] = a;
```

This is UNDEFINED! *ANYTHING* can happen !

If the data written (i.e. the "a") is user input that can be controlled by an attacker, this vulnerability can be exploited:

Anything that the attacker wants can happen.

Solution to this problem

- Check array bounds at runtime
  - Algol 60 proposed this back in 1960!
- Unfortunately, C and C++ have not adopted this solution, for efficiency reasons.

  (Perl, Python, Java, C#, and even Visual Basic have the array boundary check)
- Therefore, buffer overflow has always been the number one security issue in software.

Any C/C++ Code Handling Untrusted Input Is Vulnerable

- **Code receiving input over an untrusted network**
  - e.g., sendmail, web browsers, wireless network drivers,...
- **Code accepting input from untrusted users on multi-user systems**
  - Especially critical for services running with high privileges (e.g., ROOT on Unix/Linux, SYSTEM on Windows)
- **Code processing untrusted files**
  - Files that have been downloaded or received via email
- **Embedded software**
  - Examples: Devices with (wireless) network connections such as mobile phones, RFID cards, airplane navigation systems, ...

String Manipulation Functions

Causes buffer overflow!!

Exploitation

- The general idea is to give servers very large strings that will overflow a buffer.
- For a server with sloppy code – it's easy to crash the server by overflowing a buffer.
- It's sometimes possible to actually make the server do whatever you want (instead of crashing).

Example: gets()

```
char buf[20];
gets(buf);  // read user input until
            // first EoL or EoF character
```

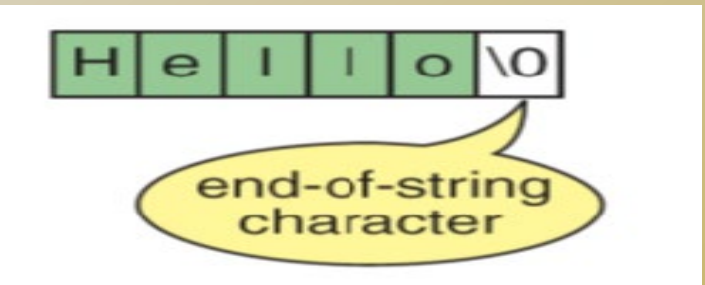
- Never use **gets**
- Use **fgets(buf, size, stdin)** instead

fgets(): reads character data until it either encounters a new line or size – 1 characters have been read.

Example: strcpy()

char dest[20];
strcpy(dest, src); // copies string src to dest

- **strcpy** assumes **dest** is long enough and **src** is null-te



- Use **strncpy(dest, src, size)** instead

Spot the defect! (1)

```
char buf[20];
char prefix[] = “http://”;
char path[]= “www.abc.com/zyz/file1”;
...
strcpy(buf, prefix);
   // copy the string prefix to buf
strncat(buf, path, sizeof(buf));
   // concatenates path to the string buf
```

The strcat() function appends the entire second string to the first, whereas strncat() appends only the specified number of characters in the second string to the first.

It shall be the number of chars that the buffer still can accept, instead tof he buf size

Another common mistake is giving **sizeof(path)** as 3rd argument...

Spot the defect! (2)

```
char src[9];
char dest[6];

char *base_url = ”www.ru.nl”;
strncpy(src, base_url, 9);
    // copies base_url to src
strcpy(dest, src);
    // copies src to dest
```

base_url is 10 chars long, incl. its null terminator ‘\0’, so **src** won’t be null-terminated

so **strcpy** will overrun the buffer **dest**

```
printf("length of dest = %d", strlen(dest));
```

Output:

length of dest = ?

Example: **strcpy** and **strncpy**

- Don't replace **strcpy(dest, src)** by
  - **strncpy(dest, src, sizeof(dest))**
- but by
  - **strncpy(dest, src, sizeof(dest)-1)**
  - **dest[sizeof(dest)-1] = `\0`;**
  - if **dest** should be null-terminated!
- **A strongly typed programming language could of course enforce that strings are always null-terminated...**

Spot the defect! (3)

```
#include <fcntl.h>
char *buf;
int i, len;
int fd = open("foo.txt", O_RDONLY);
read(fd, &len, sizeof(len));
   // read sizeof(int) bytes, ie. an int,
   // and store these in len
buf = malloc(len);
read(fd, buf, len);
```

Spot the defect! (3)

```
#include <fcntl.h>
char *buf;
int i, len;
int fd = open("foo.txt", O_RDONLY);
read(fd, &len, sizeof(len));
   // read sizeof(int) bytes, ie. an int,
   // and store these in len
buf = malloc(len);
read(fd, buf, len);
```

`len` might be negative

`len` cast to unsigned, a huge positive number → overallocated or failed

`read` may read more than expected → buffer overflow

Implicit Casting Bug

- **A signed/unsigned or an implicit casting bug**
  - **Very nasty – hard to spot**
- **C compiler doesn't warn about type mismatch between signed int and unsigned int**
  - **Silently inserts an implicit cast**

Spot the defect! (4)

```
#define MAX_BUF 256


void BadCode (char* input)
{ short len;
  char buf[MAX_BUF];

  len = strlen(input);


  if (len < MAX_BUF) strcpy(buf, input);
}

Void main()
{
  BadCode("...very long string more than 32K of characters..");
}
```

What if input is larger than 32K ?

Short is 16- bits signed int → max is $2^{15} - 1 = 32767$

len will be a negative number, due to integer overflow

hence: potential buffer overflow

Format String Vulnerability

- A format string is an ASCII string that contains text and format parameters.

Example:

```
// A statement with format string
printf("my name is : %s\n", “Smith");
// Output //
My name is : Smith
```

- If the programmer passes an attacker-controlled buffer as an argument to a printf (or any of the related functions, including sprintf, fprintf, etc), **the attacker can perform writing to arbitrary memory addresses**.

Spot the defect! (5)

```
#include <stdio.h>
int main(int argc, char* argv[])
{ if (argc > 1)
        printf(argv[1]);
        return 0;
}
```

This program is vulnerable to **format string** attacks, where calling the program with strings containing special characters can result in security breaches and system compromise.

The following program contains Format String Vulnerability:

```
int main(int argc, char** argv)
{
  char buffer[100];
  strncpy(buffer, argv[1], 100);
  // We are passing command line
  // argument to printf
  printf(buffer);
  return 0;
}
```

The attacker can pass the following string as argv[1]:

"%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"

and fool the printf into thinking it has 15 arguments. It will naively print the next 15 addresses on the stack, thinking they are its arguments:

```
$ ./a.out "%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"
0xffffdddd 0x64 0xf7ec1289 0xffffdbdf 0xffffdbde (nil) 0xffffdcc4 0xffffdc64
(nil) 0x25207025 0x70252070 0x20702520 0x25207025 0x70252070 0x20702520
```

Preventing Format String Vulnerabilities

- **Avoid User-Controlled Format Strings**: Do not allow user input to directly influence format strings.
- **Validate and Sanitize Input**: Validate and sanitize all user inputs before using them in format functions.
- **Use Explicit Format Strings**: Instead of passing user input as the format string, use explicit format strings. For example, printf("%s", userInput) is safer than printf(userInput).
- **Limited Permissions**: Ensure that the process running the code has limited permissions to minimize the impact of a successful attack.
- **Static Code Analysis**: Use static code analysis tools to detect potential format string vulnerabilities in the codebase.

Preventing Format String Vulnerabilities

Safe Code
Replace: `printf(buffer);`
With:

```
printf("%s", buffer);
```

if you compile the program and run it:

```
./a.out "AAAA%p %p %p %p %p %p %p %p %p"
```

The printf will not interpret the `"%p %p %p %p %p %p %p %p %p"` in the input string, and the output will be:

```
"AAAA%p %p %p %p %p %p %p %p %p"
```

Prevention of Buffer Overflow Attacks

Countermeasures of Buffer Overflow Attacks

1. Array bounds checking
2. Non-executable stack/heap
3. Safe **C** library
4. Compiler solutions, e.g.,
   - StackGuard
   - **RAD**
5. Type safe language, e.g., Java
6. Static source code analysis
7. Anomaly Detection
8. Code Randomization
9. Memory Address Obfuscation/**ASLR**

1. Array Bounds Checking

- Fundamental solution for all kinds of buffer overflow attacks.
- High run-time overhead.

```
int     scores[100];
int     score;
int     count = 0;

cout << "Enter a score (-1 to stop): ";

cin >> score;
while (score != -1 && count < 100)
{
        scores[count++] = score;
        cin >> score;
}
```

DO NOT USE:	Instead USE:

void main() {
char buf[4];
char src[8] = "rrrrr";
strcpy(buf,src);
}
	if (src_size >= buf_size) {
	cout<< "error";
	return(1);
	}
	else {
	strcpy(buf,src);
	}
	OR
	strncpy(buf,src,buf_size - 1);
	buf[buf_size - 1] = '\0';

2. Non-executable memory

Memory used by a process (program in execution) consists of different segments.

Program counter should point to the code segment, *not* to heap, stack, or data segments.

Making these segments *non-executable* makes it impossible for an attacker to execute their malicious code that they manage to get into some stack or heap allocated buffer.

Modern architecture introduces the NX bit to help OS manage execution permission.



Non eXecutable Stack

- **Does not block buffer overflows, but prevents the shellcode from being executed**
- **To compile a C program, just use the -z noexecstack option to mark the stack segment non-executable.**

Compile the vulnerable program **myprog.c** as:

```
$ gcc -z noexecstack myprog.c
```

Shellcode is sequence of machine code, or executable instructions, that is injected into a computer's memory with the intent to take control of a running program.

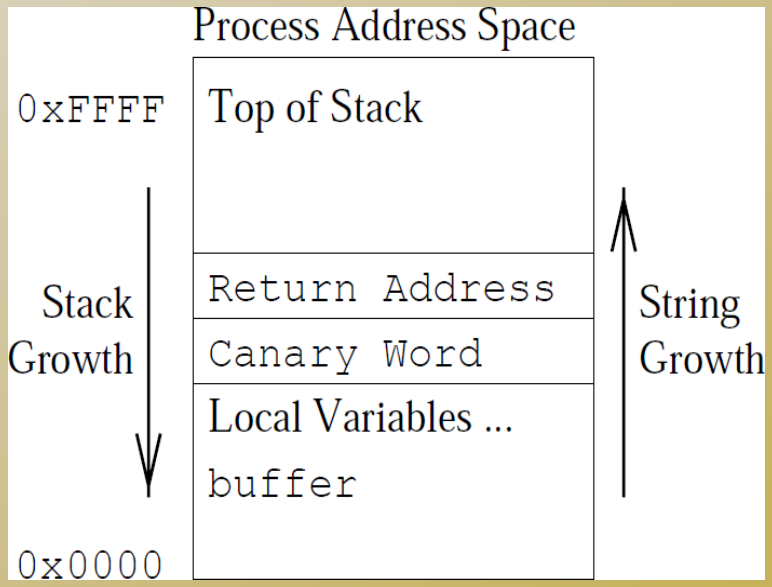
3. Safe C Library

- Some string-related **C** library functions, such as **`strcpy`** and **`strcat`** don't check the buffer boundaries of destination buffers.
- Replace **`strcpy`** with **`strncpy`**, or replace **`strcat`** with **`strncat`**, … and so on.
- Weakness:
  - Plenty of other **C** statements could still results in buffer overflow vulnerabilities.
    - E.g.
```
while ((*(ptr+i)=getchar())!=EOF)
        i++;
```

4. Compiler Solutions:

StackGuard

- Put a ***canary word*** before each return address in each stack frame. Usually, when a buffer overflow attack is launched, not only the return address but also the canary word will be overwritten; thus, by checking the integrity of the canary word, this mechanism can defend against stack smashing attacks.



- **Embed "canaries" in stack frames and verify their integrity prior to function return.**

Compiler Solutions: StackGuard (cont.)

Random canary:

- Choose random string at program startup.
- Insert canary string into every stack frame.
- Verify canary before returning from function.
- To corrupt random canary, attacker must learn current random string.

Low performance overhead

Weakness:

- Change the layout of the stack frame of a function; hence, this mechanism is not compatible with some programs, e.g., debugger.
- Only protect return addresses.

Compiler Solutions: RAD

- When a function is called, a copy of its return address is saved in a well-protected area **RAD (Return Address Defender),** a safe region of memory that would be very difficult to corrupt)
- When the function finishes, before returning to its caller, the callee checks the return address in its stack frame to see whether the **RAD** has a copy of that address. If there is no such address in the **RAD**, then a buffer overflow attack is alarmed.
- RAD does not alter the structure of the stack frame (therefore, it is compatible with unmodified debuggers).
- Low performance overhead.
- Weakness: only protect return addresses.

5. Type-safe Language (e.g., Java)

- Type-safe languages automatically enforce array bounds checking, helping to prevent common errors such as buffer overflows.
- Weakness:

  The majority of existing software is not written in these languages. Rewriting all legacy code using type-safe languages would be an extremely challenging task due to the scale and complexity involved.

6. Static Source Code Analysis

- Analyze source code to find potential program statements that could result in buffer overflow vulnerabilities. E.g., program statements like

```
while((*(buf+i)=getchar())!=EOF)
  i++;
```

  are not safe.

- Weakness:
  - False positives and false negatives.
  - Difficulty to obtain the source code.
  - Limited understanding of runtime behavior.

7. Anomaly Detection

- Anomaly detection is a technique used to identify data patterns that deviate from the norm. In cybersecurity, it helps detect unusual activities or behaviors that could signal potential security threats.

- Weakness:
  - False positives and false negatives.

8. Code Randomization

- This method involves the randomization of the code that is executed in a process.
- This approach encrypts instructions of a process and decrypts instructions when they are prepared to be executed. Because attackers don't know the key to encrypt their code, their injected code can not be decrypted correctly. As a result, their code can not be executed.
- Weakness:
  - The main assumption of this method is that most attacks that attempt to gain control of a system are **code injection attacks**.
  - Performance overhead.

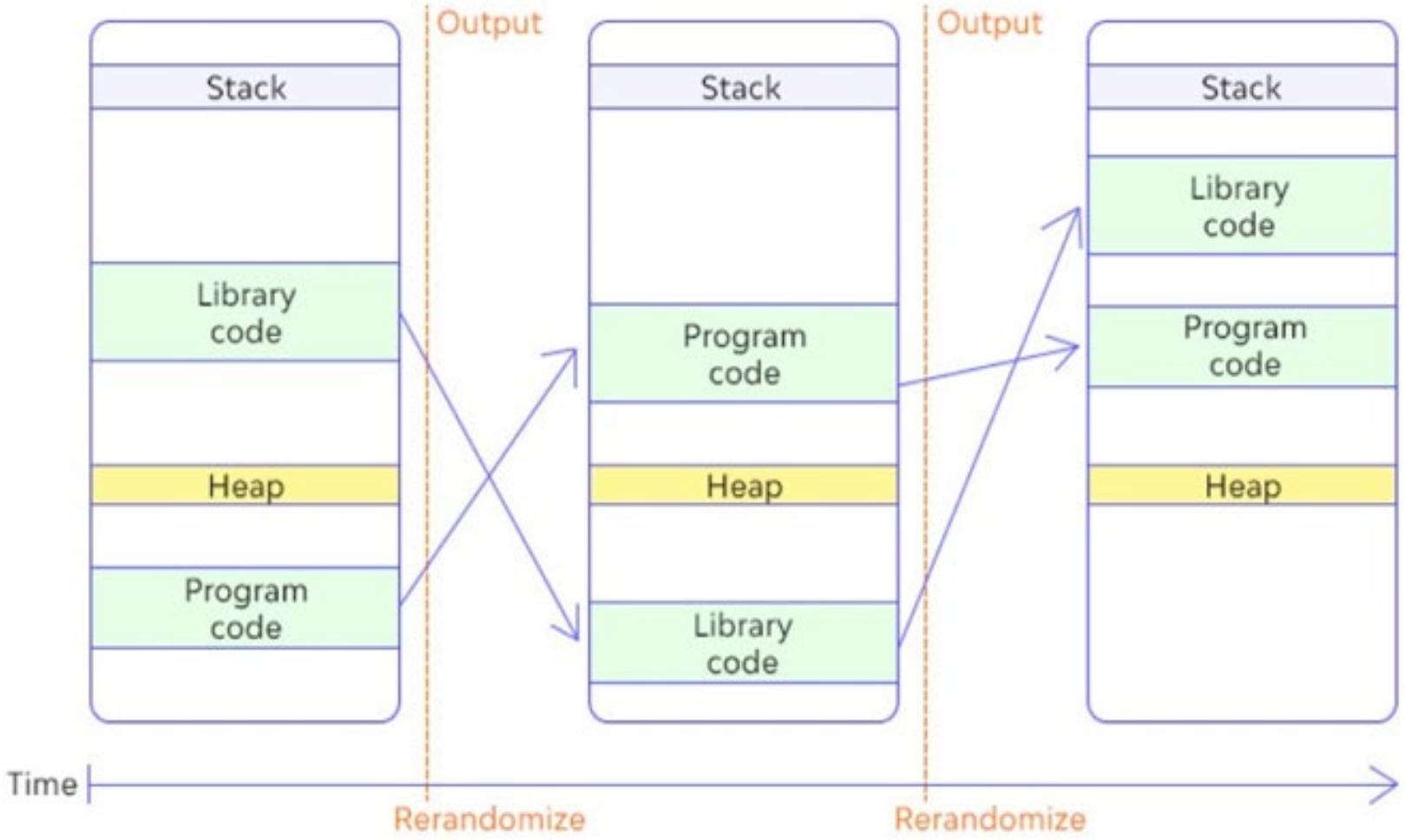
9. Memory Address Obfuscation/ASLR

- **ASLR**: Address Space Layout Randomization
- A security technique that randomizes the memory locations of key process components (e.g., the stack, heap, and libraries) **each time** a program is run.
- It makes it significantly harder for attackers to predict where their malicious code or targeted functions will be located in memory, forcing them to rely on guesswork.
- Weakness:
  - Change the memory layout during runtime → may conflict with certain assumptions made by the original file format..
  - Increase the complexity of debugging, as memory addresses are different on each execution

Aspects of Address Obfuscation (1)

- Address obfuscation involves randomizing the base addresses of various memory regions to make it harder for attackers to predict memory layouts. Key components affected by this randomization include:
  - ✓ The stack
  - ✓ The heap
  - ✓ The starting address of dynamically linked libraries
- Additionally, address obfuscation can rearrange the order of variables and functions in memory.

Address Space Layout Randomization example



Aspects of Address Obfuscation (2)

- Another important aspect is the introduction of random-length gaps, such as:
  - ✓ Padding in stack frames
  - ✓ Padding between `malloc` allocations
  - ✓ Padding between variables and static data structures
  - ✓ Random length gaps in the code segments.

Introduction to Return Oriented Programming

DEP: Data Execution Prevention

- Execute Code, not Data.
- Data areas marked non-executable.
  - Stack marked non-executable.
  - Heap marked non-executable.
- Hardware enforced (NX).
- You can load your shellcode in the stack or the heap...
- ...but you can't jump to it.

EIP control

- EIP movement restricted.
  - Can't jump to Heap.
  - Can't jump to Stack.
- Can only land in executable regions.



A **DLL** is a dynamic link library. It is a collection of code and/or data, which may be used by several applications (or other libraries/modules).

EIP: Extended Instruction Pointer

How can an Attacker Defeat DEP?

- Borrow bits and pieces of code that already exist in executable regions of the process!
- Then create a SEQUENCE of operations to carry out an attack

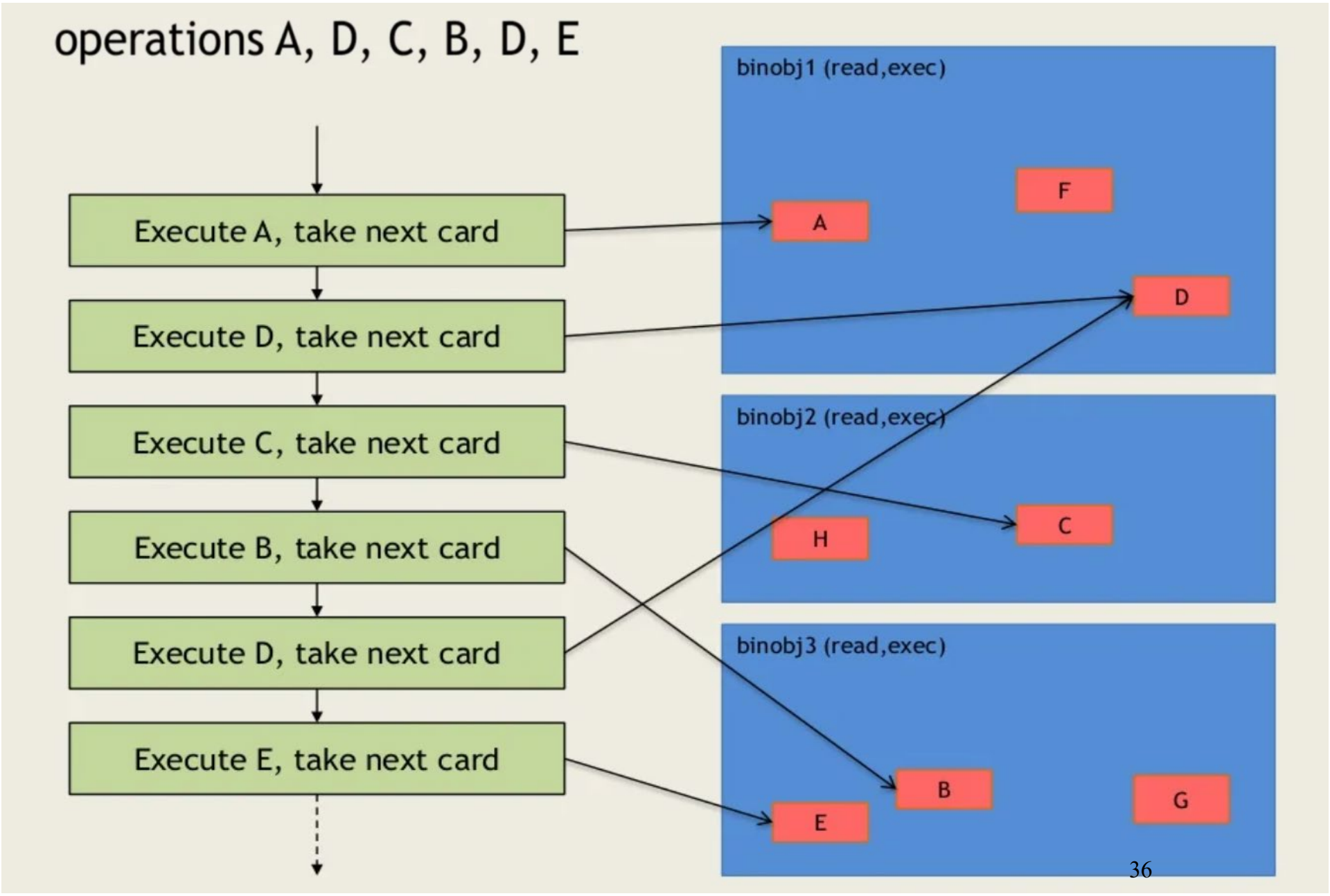
Orchestrating Code Execution

- Assume a series of cards where each card indicates an operation to be performed.
- The code for that operation is present in the executable regions of the process memory.
  - binary or shared library.

Orchestrating Code Execution

- If we can create a chain of cards, each corresponding to a basic operation...
- ...we can perform complex operations.

- Example: Assume a complex operation broken down into a sequence of basic operations A, D, C, B, D, E.


operations A, D, C, B, D, E
Execute A, take next card
Execute D, take next card
Execute C, take next card
Execute B, take next card
Execute D, take next card
Execute E, take next card
binobj1 (read,exec)
F
A
D
binobj2 (read,exec)
H
C
binobj3 (read,exec)
B
G
E

Orchestrating Code Execution

- Complex code can therefore be transformed into a sequence of primitive operations...
- ...which are already present in the process' executable regions.

- These operations are called GADGETS.

Such an Idea is Called **Return Oriented Programming (ROP)**

- The first idea of executing **existing** code was discussed in 1997
- Solar Designer's **Ret2LibC** technique
- Designed to defeat non executable stack

How Ret2LibC works?

- **Find Gadgets**: The attacker analyzes the target program's binary to identify useful gadgets. Gadgets are short sequences of instructions ending with a ret instruction. These gadgets are used to perform specific actions, such as modifying registers or the stack.
- **Controlled Stack**: The attacker exploits a vulnerability (like a buffer overflow) to overwrite the program's stack with controlled data, including the return addresses.
- **Redirect Control Flow**: Instead of pointing the return addresses to the start of malicious shellcode (which might be blocked due to security measures like non-executable stacks), the attacker points them to specific gadgets in LibC.

How Ret2LibC works? (cont.)

- **Chain Gadgets**: The attacker constructs a ROP chain, which is a sequence of return addresses pointing to different gadgets. Each gadget performs a specific task, such as loading values into registers or invoking LibC functions.
- **Execute ROP Chain:** When the vulnerable function returns, it starts executing the ROP chain. The gadgets within the chain perform the desired operations using the attacker-controlled data, effectively executing the attacker's commands.
- **Achieve Arbitrary Code Execution**: By chaining gadgets effectively, attackers can achieve arbitrary code execution without injecting new code, making it difficult to detect and prevent these attacks.

Ret2LibC (Return-to-libC) Attack

- **Exploit buffer overflow** to overwrite the return address of the current function.
- **Set the return address (EIP)** to point to a function in the LibC (e.g., system()).
- **Prepare the fake stack frame** with arguments for the function (e.g., “/bin/sh”)
- The program **returns to the LibC function** and executes it using the attacker’s crafted arguments.
- The function call (e.g., system(“/bin/sh/”)) results in the attacker's desired outcome, such as spawning a shell.

Midterm Exam Preparation

Overview

- Exams are tests of your understanding of the content.
- Attend class, read textbooks, review class notes, complete quizzes and homework ➔ fully understand the contents
- This is a close-book and close-notes exam.
- The exam format: MCQs, essays, etc.
- **90 minutes**; around 46 questions with points ranging from 1 to 10
- The exam covers Chapters 1 – 4

Ret2LibC – how does it work?

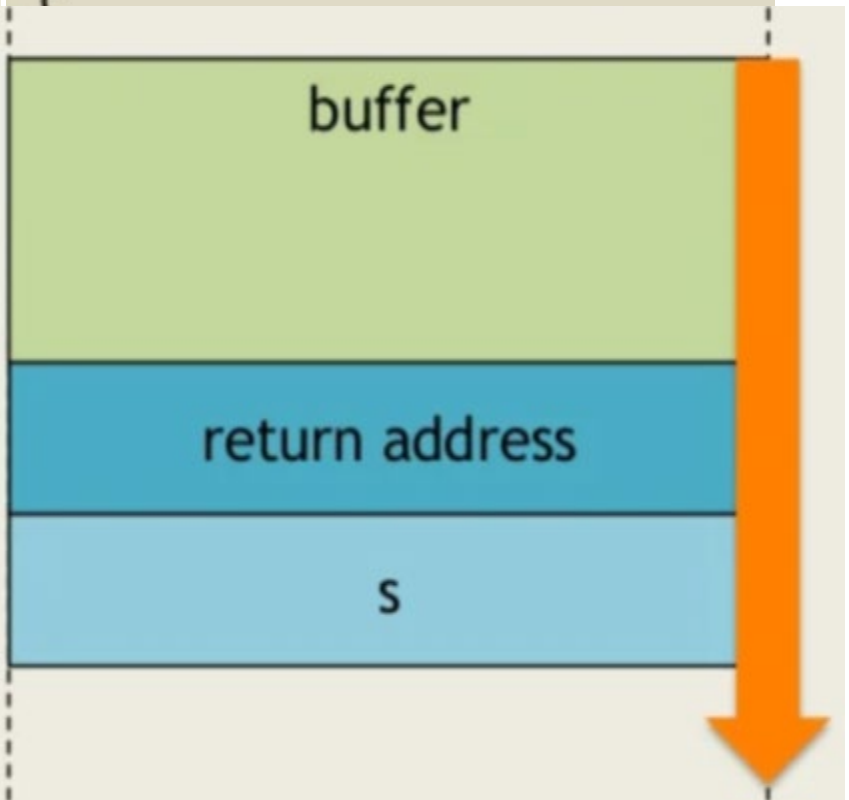
- The following function is vulnerable to a stack overflow:

```
void func1(char *s)
{
    char buffer[80];
    strcpy(buffer, s);
    :
    :
}
```

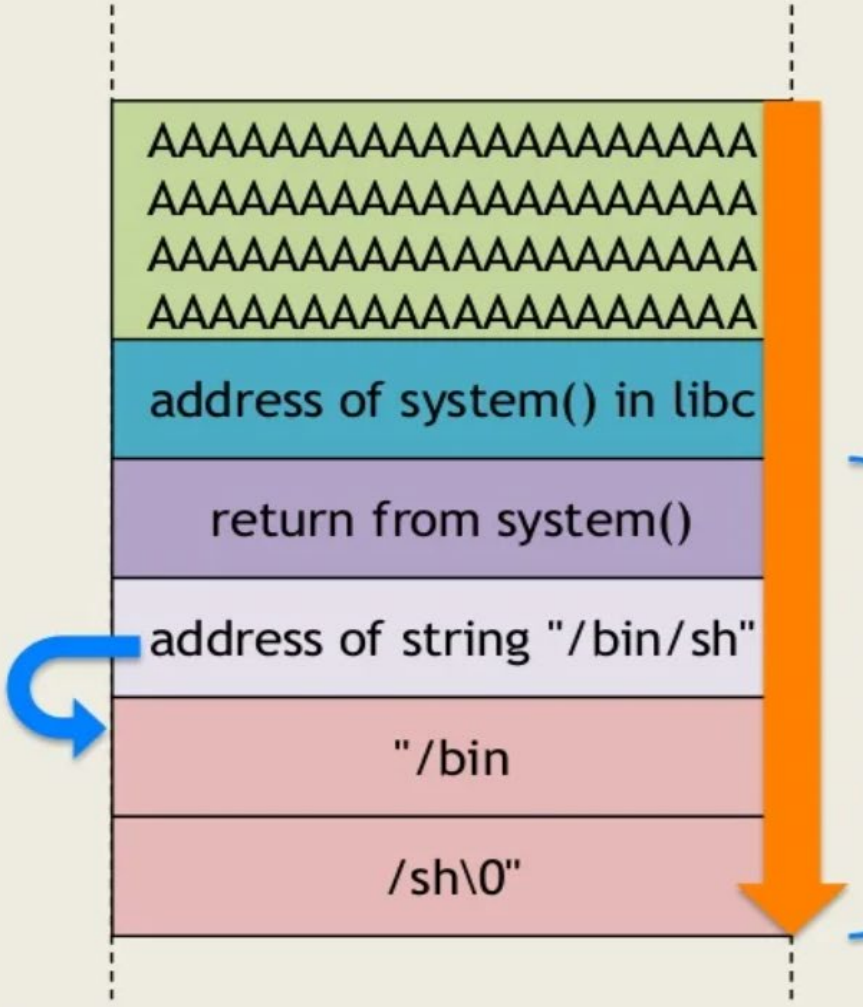
Stack frame for func1()

- If s points to a string greater than 80 characters, it will result in a stack overflow.
- The return address will be overwritten with an attacker controlled value.
- When func1() returns, EIP can be set to an arbitrary address, resulting in process execution hijacking.

```
void func1(char *s)
{
    char buffer[80];
    strcpy(buffer, s);
    :
    :
}
```



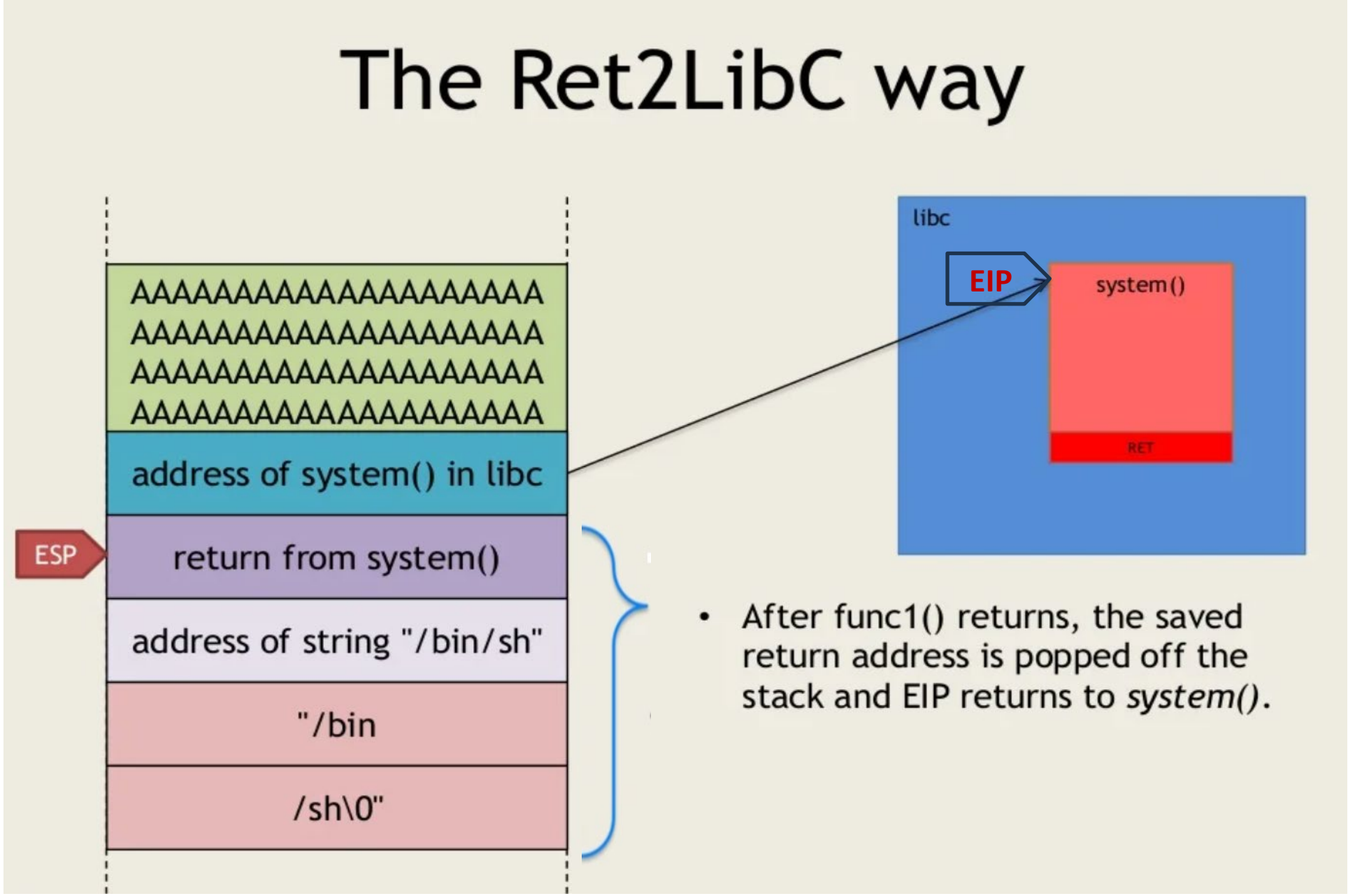
The Ret2LibC way



- func1() is made to "return" to system() by overwriting its return address with the address of system() in libc.
- A fake *calling frame* for system() is appended after the return address for func1()
- Return address for system() and parameter (*pointer to "/bin/sh"*) are provided in the fake calling frame.

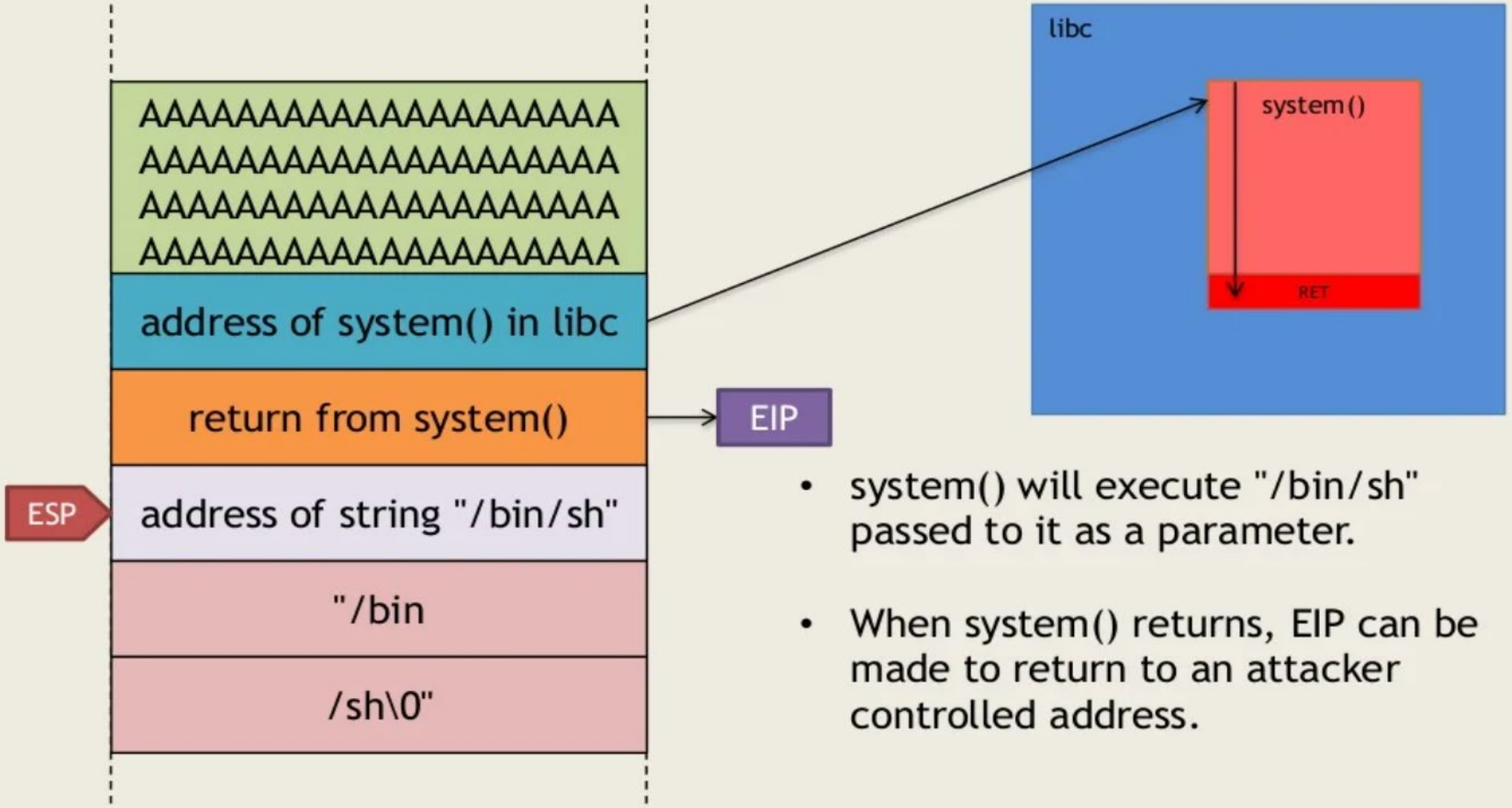
Function prototype of system

int system(const char *command);



ESP is **used to point to the next item on the stack and is referred to as the 'stack pointer'**.

The Ret2LibC way



- system() will execute "/bin/sh" passed to it as a parameter.
- When system() returns, EIP can be made to return to an attacker controlled address.

Ret2LibC Conclusions

- It is possible to call an arbitrary function by placing a crafted (fake) stack frame in memory and overwriting the return address (EIP) to point to a desired function, such as one in LibC.
- After the arbitrary function is executed, it is also possible to retain control of the EIP, allowing the attacker to chain further function calls or redirect execution as needed.
- Ret2LibC serves as a foundation for Return-Oriented Programming (ROP), a more advanced exploitation technique that chains multiple small code fragments (called "gadgets") to achieve complex behaviors without injecting custom code.

Summary

- **ROP** requires a different mindset for code execution, as it doesn't follow the traditional flow of executing instructions sequentially.
- Instead of executing a series of opcodes and operands, ROP relies on chaining small **gadgets**, each ending in a return instruction.
- These gadgets are arranged on the stack in a specific order to perform the desired operations without injecting new code.
- ROP requires substantial hands-on practice to master, given the complexity of identifying gadgets and constructing reliable chains.
- There are sophisticated tools available to assist in assembling ROP chains, such as: **Skyrack**, **RoPeWe**,and **ropshell.com**

Supplementary Materials with Complex Examples

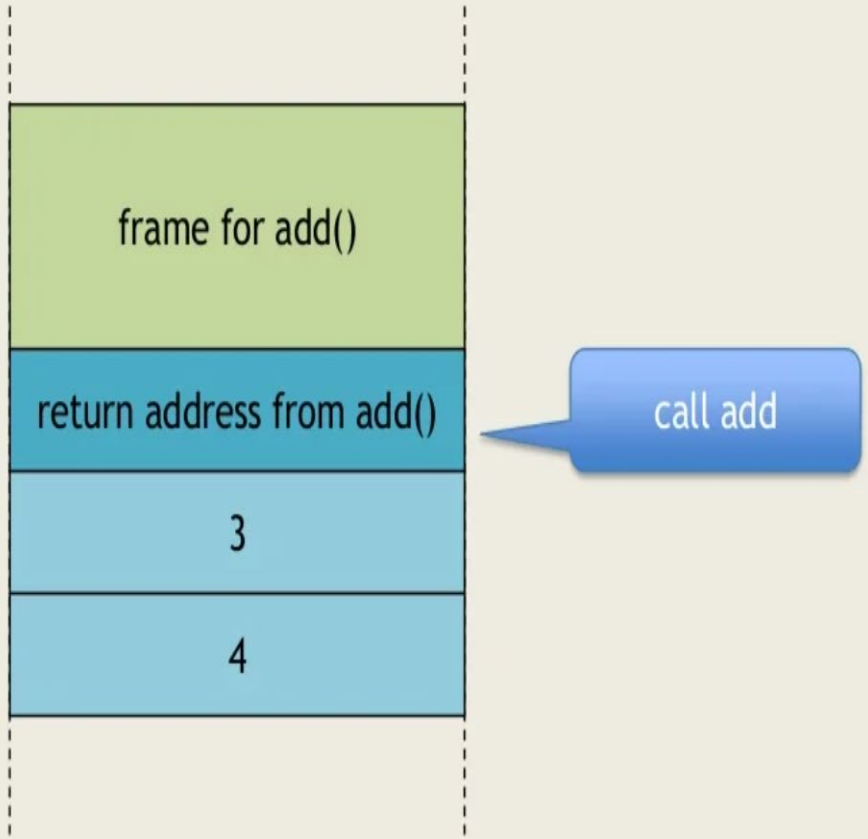
Calling a function

- Add two ints x, y.
- add(3,4)
- What does the calling frame look like?

```
void add(int x, int y)
{
   int sum;
   sum = x + y;
   printf("%d\n", sum);
}

int main()
{
   add(3, 4);
}
```

Stack frame for add(3,4)



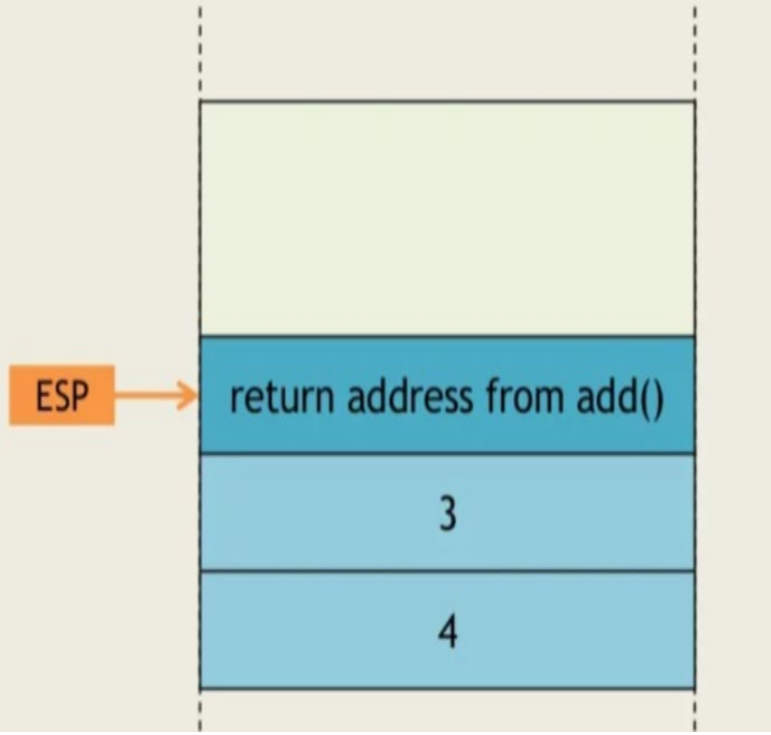

```
void add(int x, int y)
{
   int sum;
   sum = x + y;
   printf("%d\n", sum);
}

int main()
{
   add(3, 4);
}
```

Return from add(3,4)

- add() is about to return.
- RET after epilogue of add().
- Where does ESP point to?
  - immediately before the RET

Before the RET



Another function

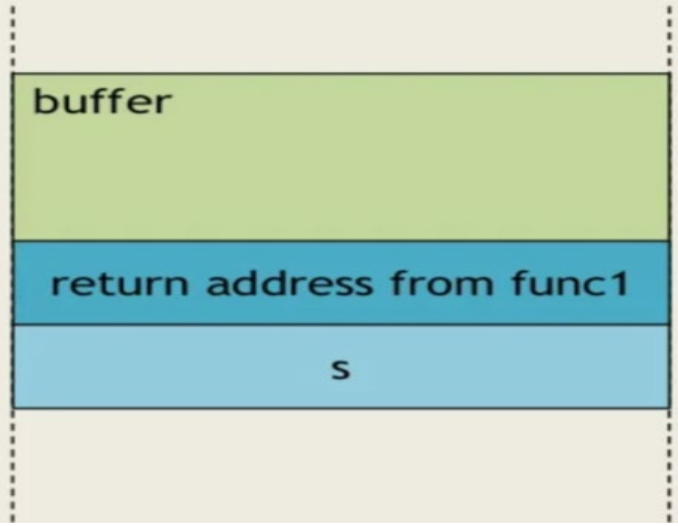
- Stack overflow in func1.
- Can we call add(5, 6) after returning from func1?

```
void func1(char *s)
{
   char buffer[128];
   strcpy(buffer, s);
}

int main()
{
   func1(argv[1]);
}
```

```
void add(int x, int y)
{
   int sum;
   sum = x + y;
   printf("%d\n", sum);
}
```

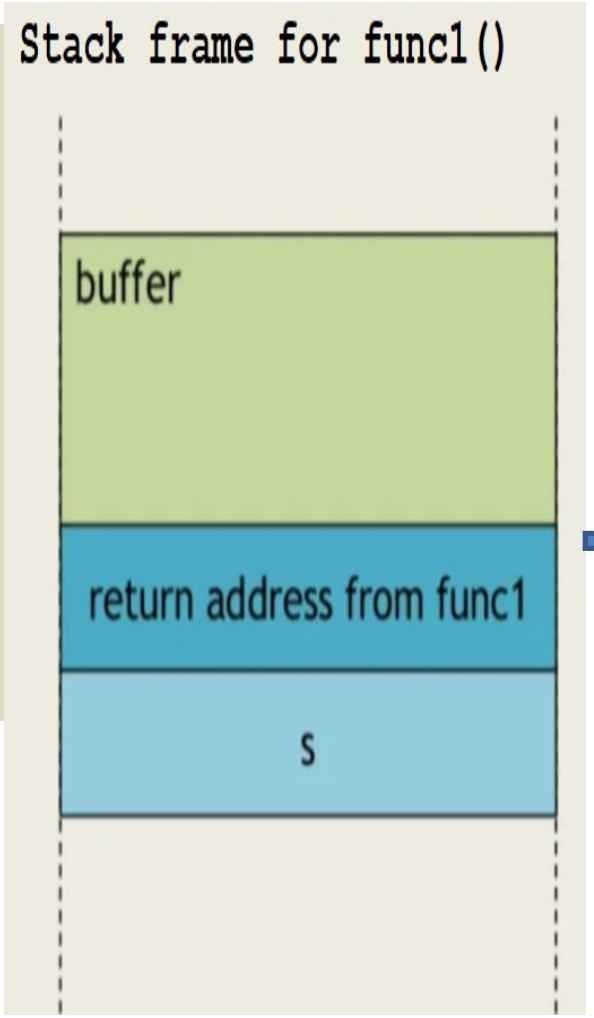
Stack frame for func1()

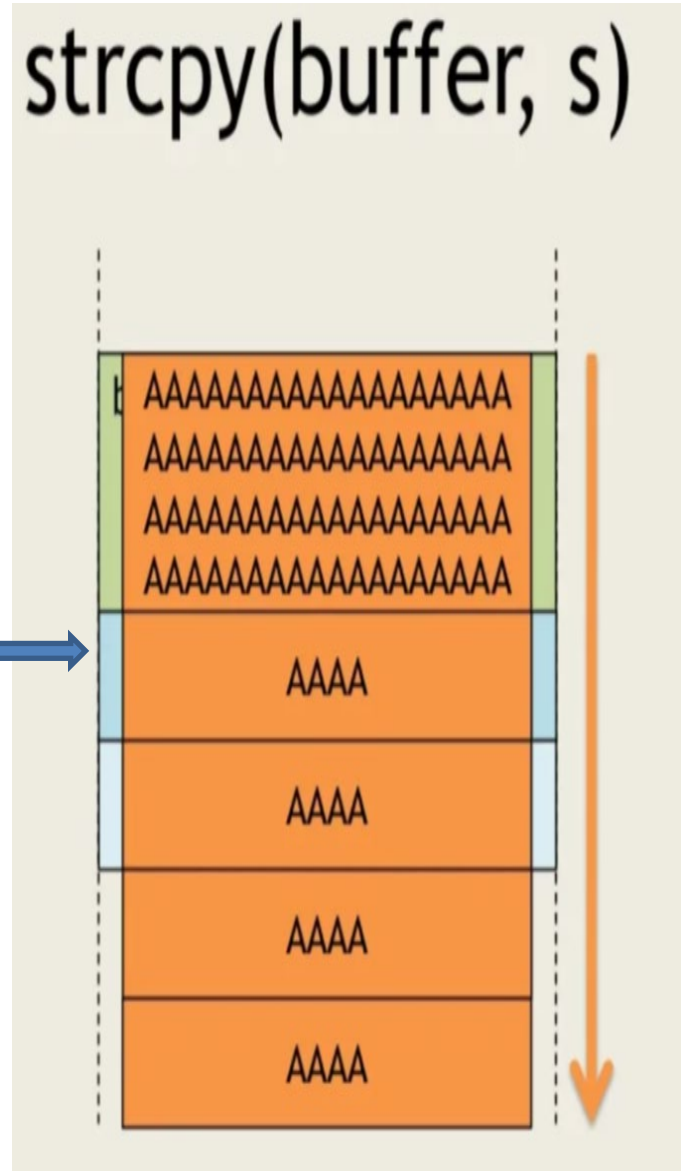


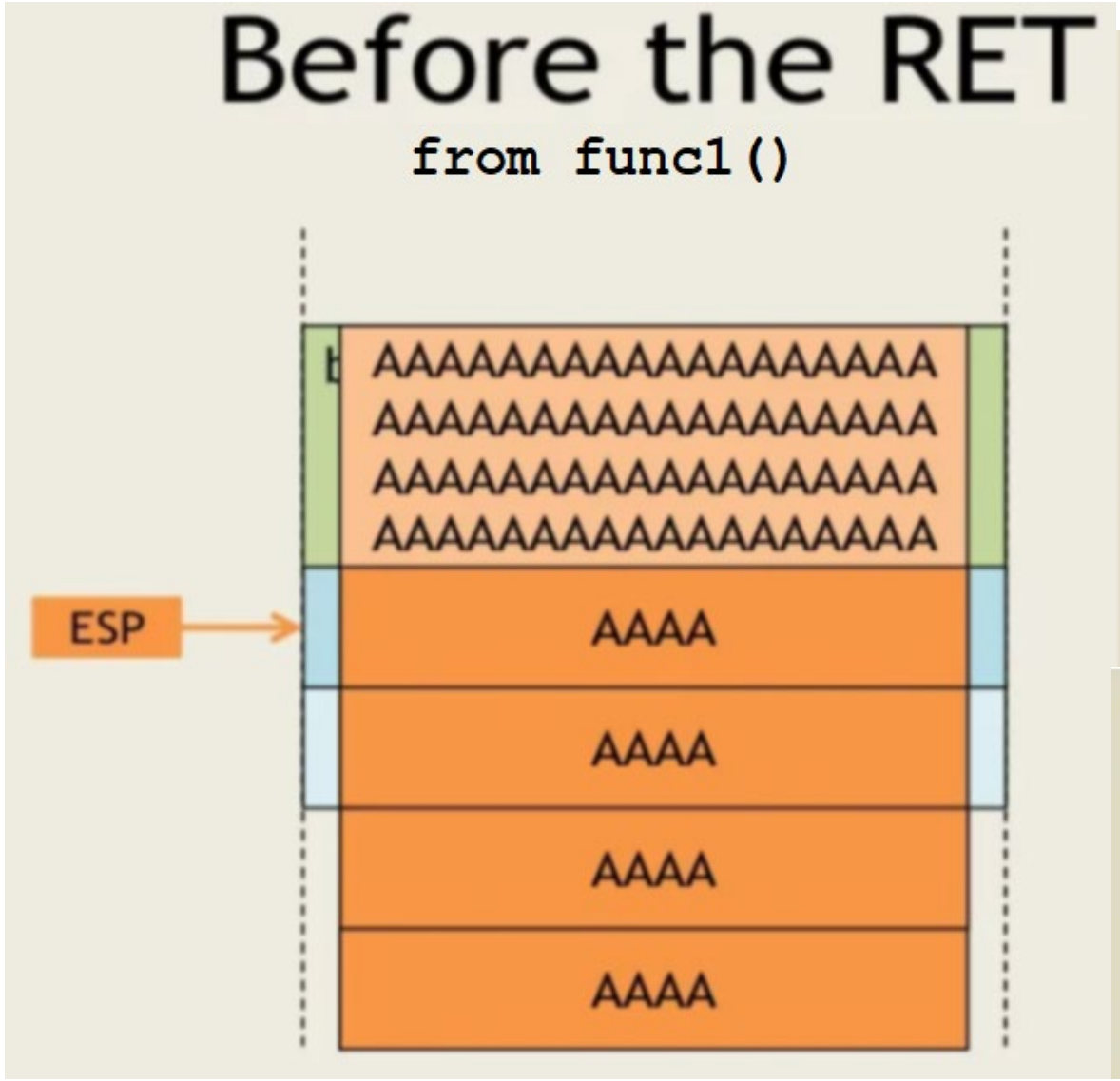
Overflow the buffer

```
void func1(char *s)
{
   char buffer[128];
   strcpy(buffer, s);
}

int main()
{
   func1(argv[1]);
}
```





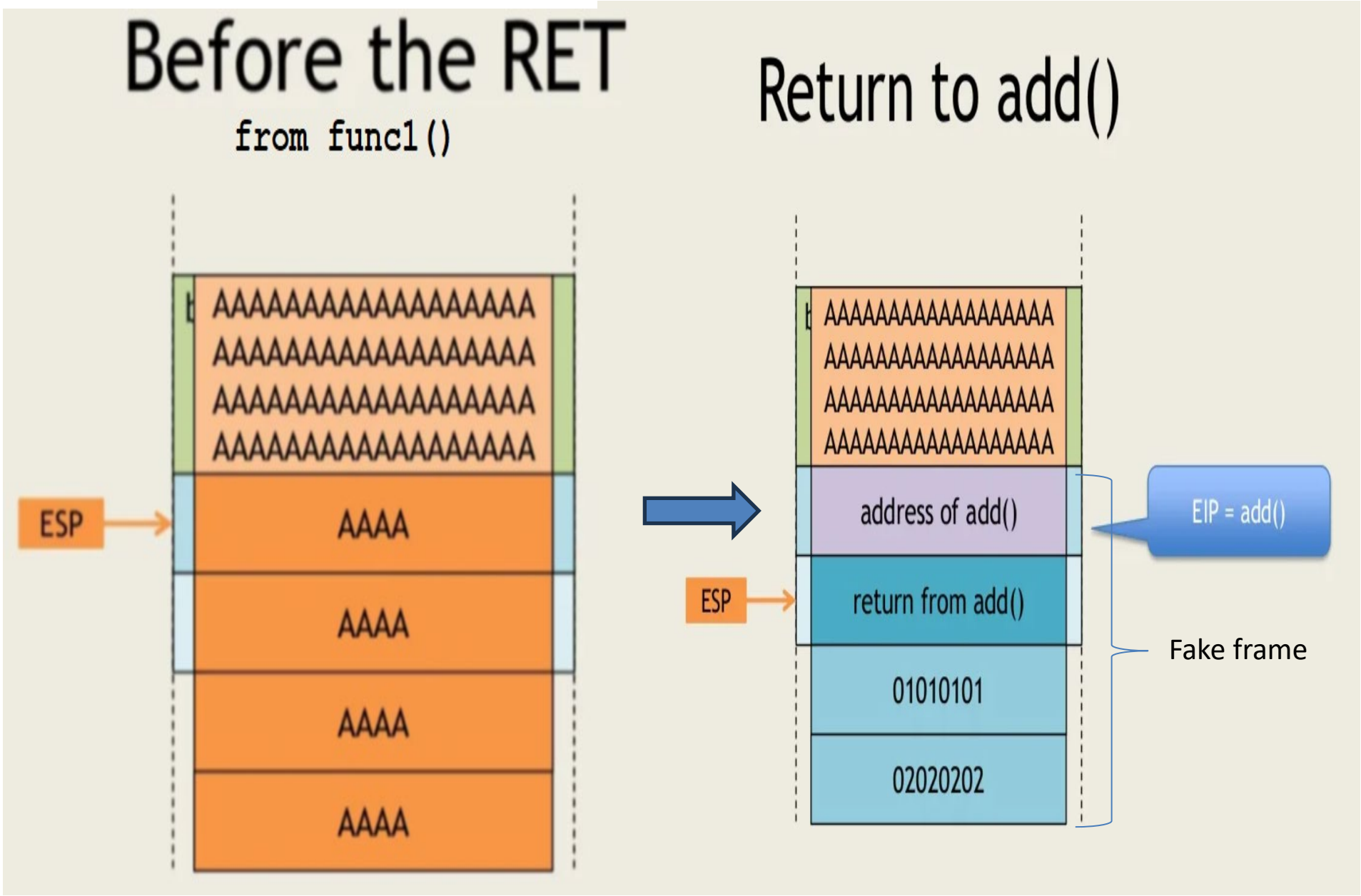



```
void func1(char *s)
{
   char buffer[128];
   strcpy(buffer, s);
}

int main()
{
   func1(argv[1]);
}
```

```
void add(int x, int y)
{
   int sum;
   sum = x + y;
   printf("%d\n", sum);
}
```

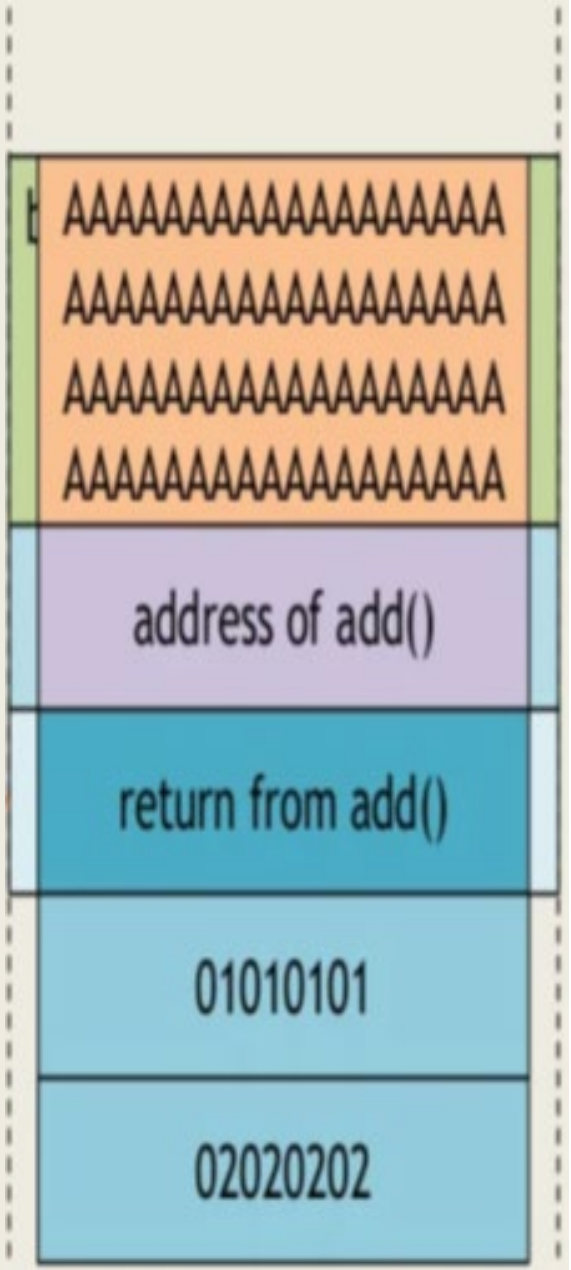
Buffer is already overflown



The attack string S contains 128 chars ‘A’s, address of add(), the return address from add(), and the two parameters (01010101 and 02020202)

Return to add()

- By carefully creating a frame...
- ...we can make the program "return to our function". **the add( ) function**
- We control the parameters.
- We also control where to jump to after our function returns.



rop_victim.c

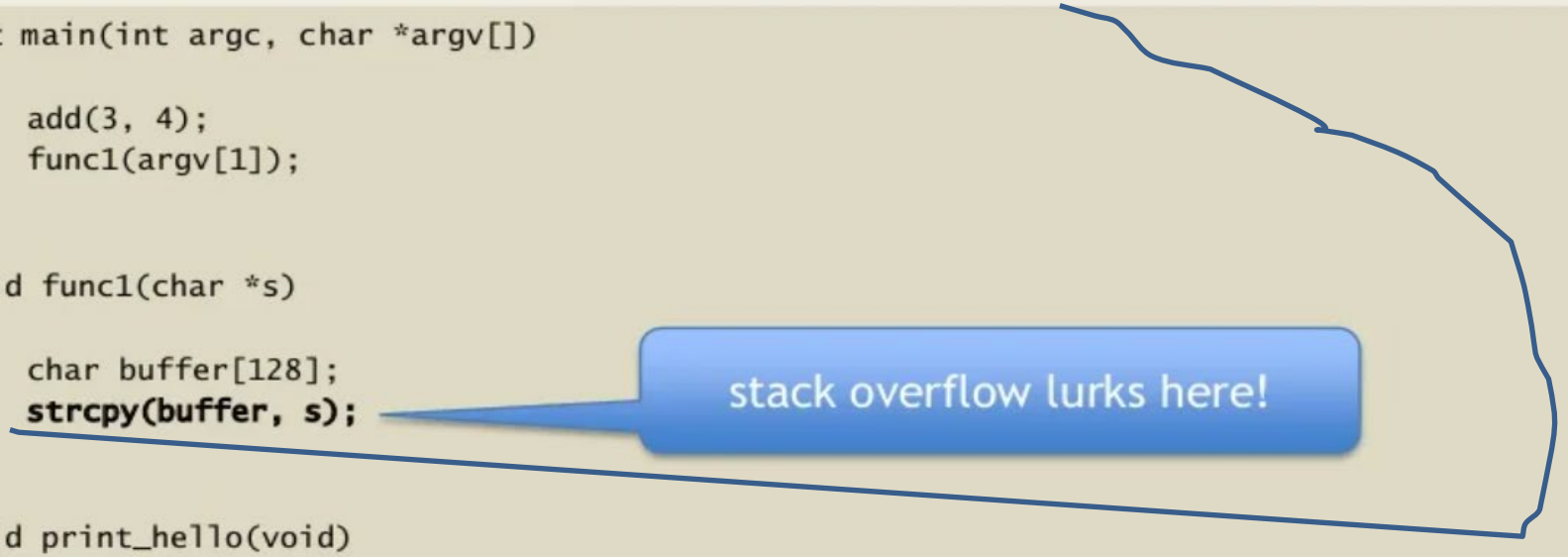
ROP example: We want to call add() from func1()

```
int main(int argc, char *argv[])
{
    add(3, 4);
    func1(argv[1]);
}

void func1(char *s)
{
    char buffer[128];
    strcpy(buffer, s);
}

void print_hello(void)
{
    printf("Hello World\n");
}

void add(int x, int y)
{
    int sum;
    sum = x + y;
    printf("%d + %d = %d\n", x, y, sum);
}
```



Building rop_victim

- Create a file called "rop_victim.c" with the contents as shown in the previous slide.
- Compile "rop_victim.c" as follows

```
$ gcc rop_victim.c -o rop_victim
```

- And run it normally:

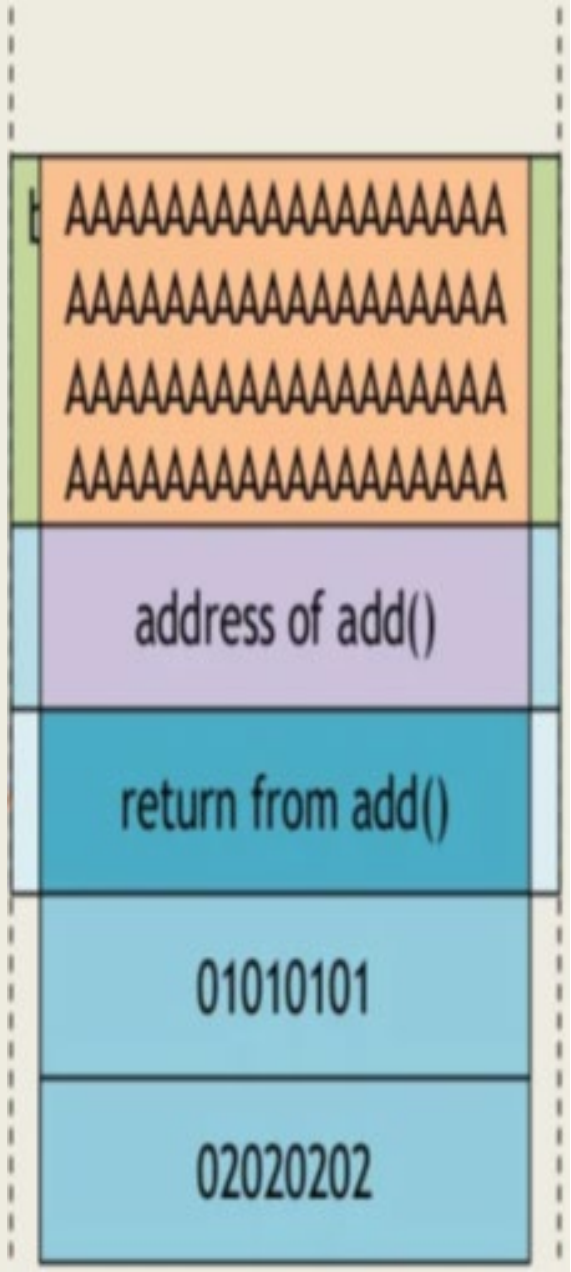
```
$ ./rop_victim HELLOWORLD
```

3 + 4 = 7

No buffer overflow – S is a small string “HELLOWORLD”

Making func1() return to add()

- We shall exploit the stack overflow in func1() and make it *return* to add().
- For this, we must inspect the calling frame for add()...
- ...and eventually place a fake frame for add() on the stack when overflowing func1().



Inspecting add()'s calling frame

- Open "rop_victim" in gdb, and set a breakpoint just before the call to add() in main().
- We want to inspect the stack memory before *call add()*...
- ...and immediately after *call add()* as well.

```
$ gdb rop_victim
```

gdb: a GNU debugger that can step instruction by instruction.

gdb'ing add

- Set a breakpoint before *call add()*

```
$ gdb rop_victim
(gdb) disassemble main
0x08048454 <+0>:    push   %ebp
0x08048455 <+1>:    mov    %esp,%ebp
0x08048457 <+3>:    and    $0xfffffff0,%esp
0x0804845a <+6>:    sub    $0x10,%esp
0x0804845d <+9>:    movl   $0x4,0x4(%esp)
0x08048465 <+17>:   movl   $0x3,(%esp)
0x0804846c <+24>:   call   0x80484bd <add>
0x08048471 <+29>:   cmpl   $0x1,0x8(%ebp)
0x08048475 <+33>:   jle    0x8048487 <main+51>
0x08048477 <+35>:   mov    0xc(%ebp),%eax

(gdb) break *0x0804846c
```

```
int main(int argc, char *argv[])
{
    add(3, 4);
    func1(argv[1]);
}

void func1(char *s)
{
    char buffer[128];
    strcpy(buffer, s);
}

void print_hello(void)
{
    printf("Hello World\n");
}

void add(int x, int y)
{
    int sum;
    sum = x + y;
    printf("%d + %d = %d\n", x, y, sum);
}
```

Before add

- Run rop_victim and single step into add()

```
(gdb) run HELLOWORLD
Starting program: /home/krafty/rop_intro/rop_victim HELLOWORLD
Breakpoint 1, 0x0804846c in main ()

(gdb) stepi
0x080484bd in add ()

(gdb) where
#0  0x080484bd in add ()
#1  0x08048471 in main ()
```

- We are inside add() now.
- Look at the stack frame.

Next slide

```
int main(int argc, char *argv[])
{
    add(3, 4);
    func1(argv[1]);
}

void func1(char *s)
{
    char buffer[128];
    strcpy(buffer, s);
}

void print_hello(void)
{
    printf("Hello World\n");
}

void add(int x, int y)
{
    int sum;
    sum = x + y;
    printf("%d + %d = %d\n", x, y, sum);
}
```

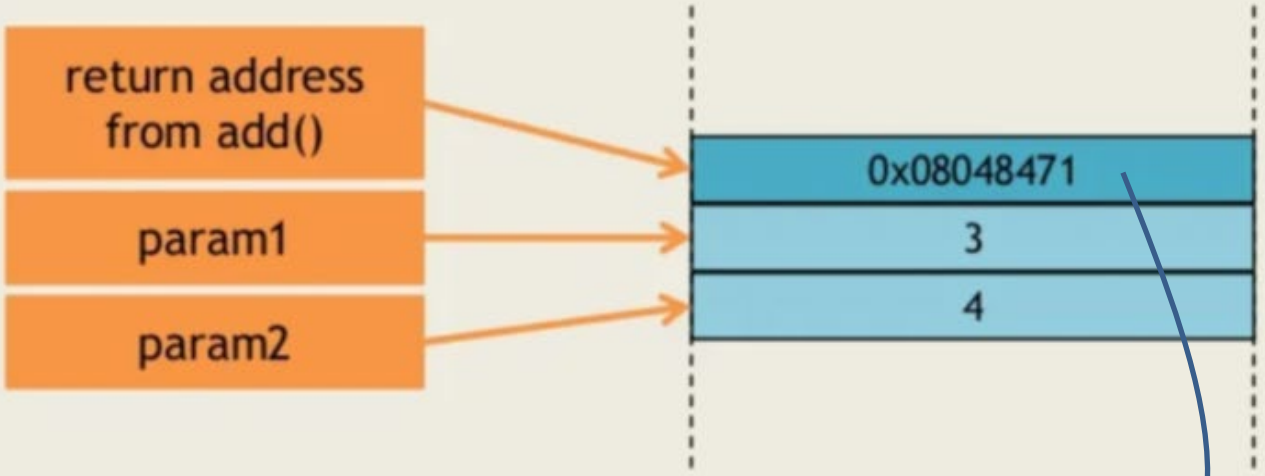
Stack Frame for add(3,4)

- Dump the stack:

print out the contents of the first 10 words (aka 40 bytes) of memory

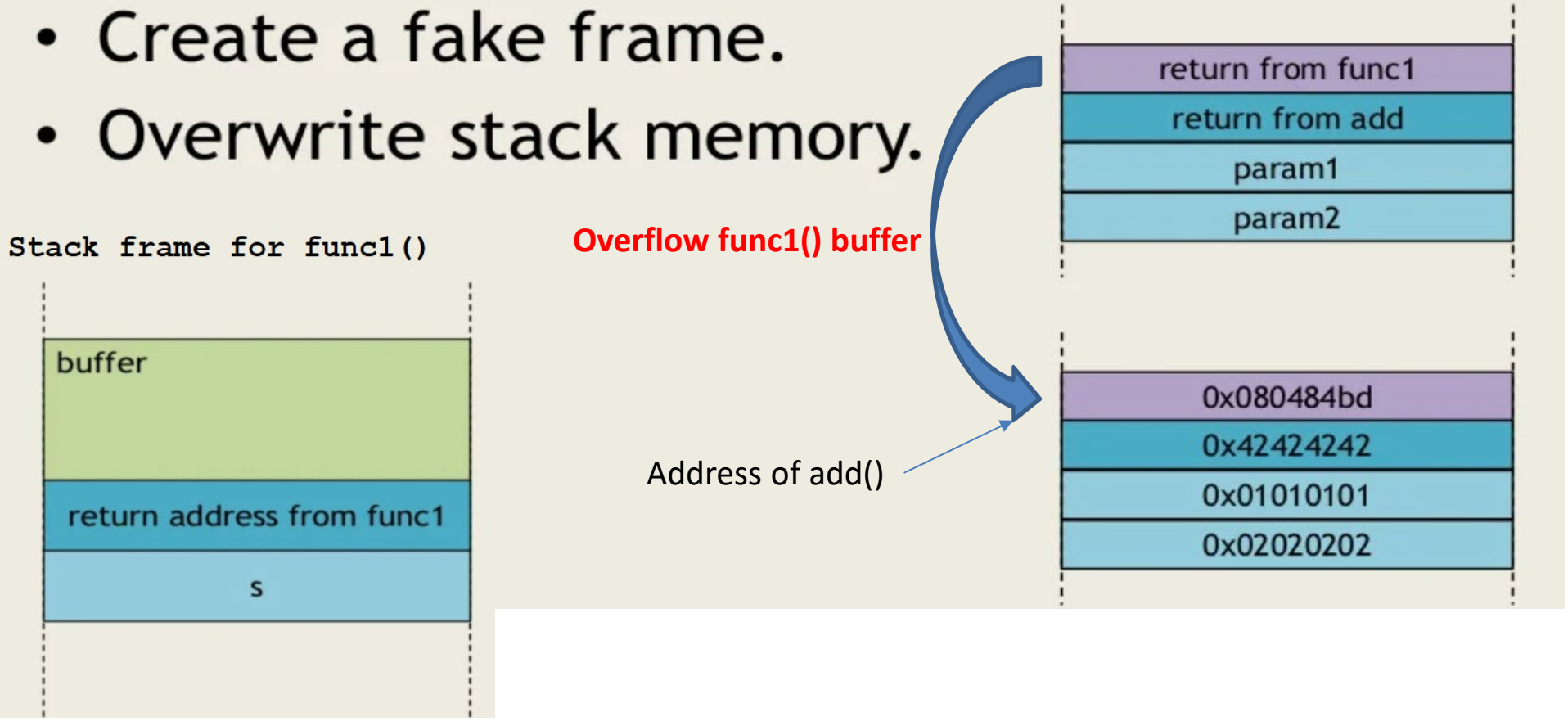
```
(gdb) x/10x $esp
0xbffff41c: 0x08048471 0x00000003 0x00000004 0x0804851b
0xbffff42c: 0x00292ff4 0x08048510 0x00000000 0xbffff4b8
```

```
(gdb) disassemble main
0x08048454 <+0>:   push   %ebp
0x08048455 <+1>:   mov    %esp,%ebp
0x08048457 <+3>:   and    $0xfffffff0,%esp
0x0804845a <+6>:   sub    $0x10,%esp
0x0804845d <+9>:   movl   $0x4,0x4(%esp)
0x08048465 <+17>:  movl   $0x3,(%esp)
0x0804846c <+24>:  call   0x80484bd <add>
0x08048471 <+29>:  cmpl   $0x1,0x8(%ebp)
0x08048475 <+33>:  jle    0x8048487 <main+51>
0x08048477 <+35>:  mov    0xc(%ebp),%eax
```



Overflowing func1()

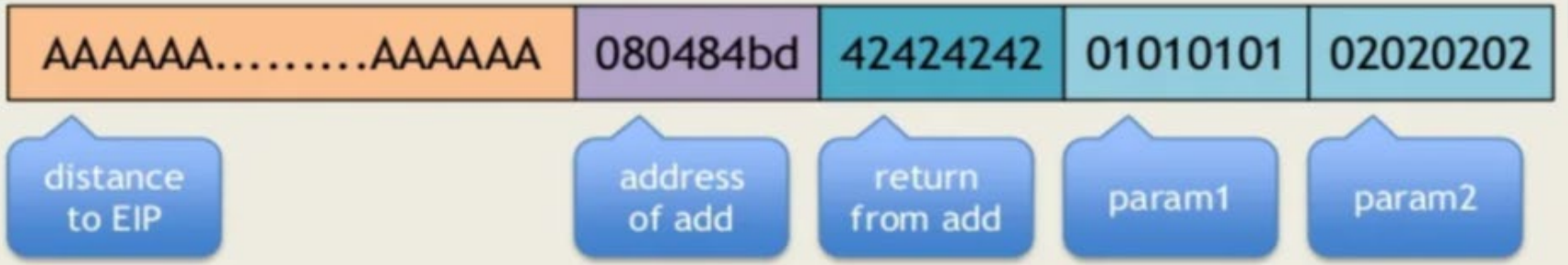
- Overflow func1 and…
  …return to add(01010101, 02020202)
- Create a fake frame.
- Overwrite stack memory.



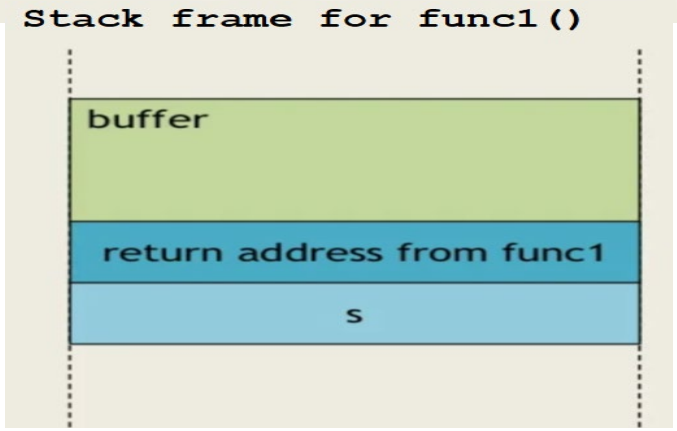
Creating a fake frame

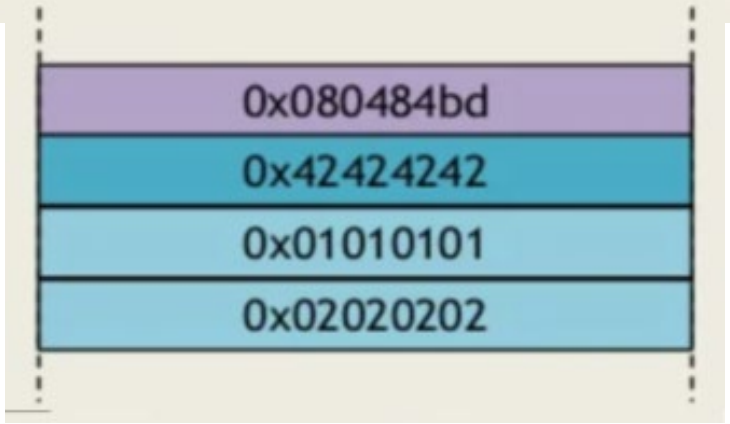
- Create the buffer overflow payload as follows:

S = argv[1] = fake frame



- Write a program to create such a buffer and use it as an input to rop_victim.c

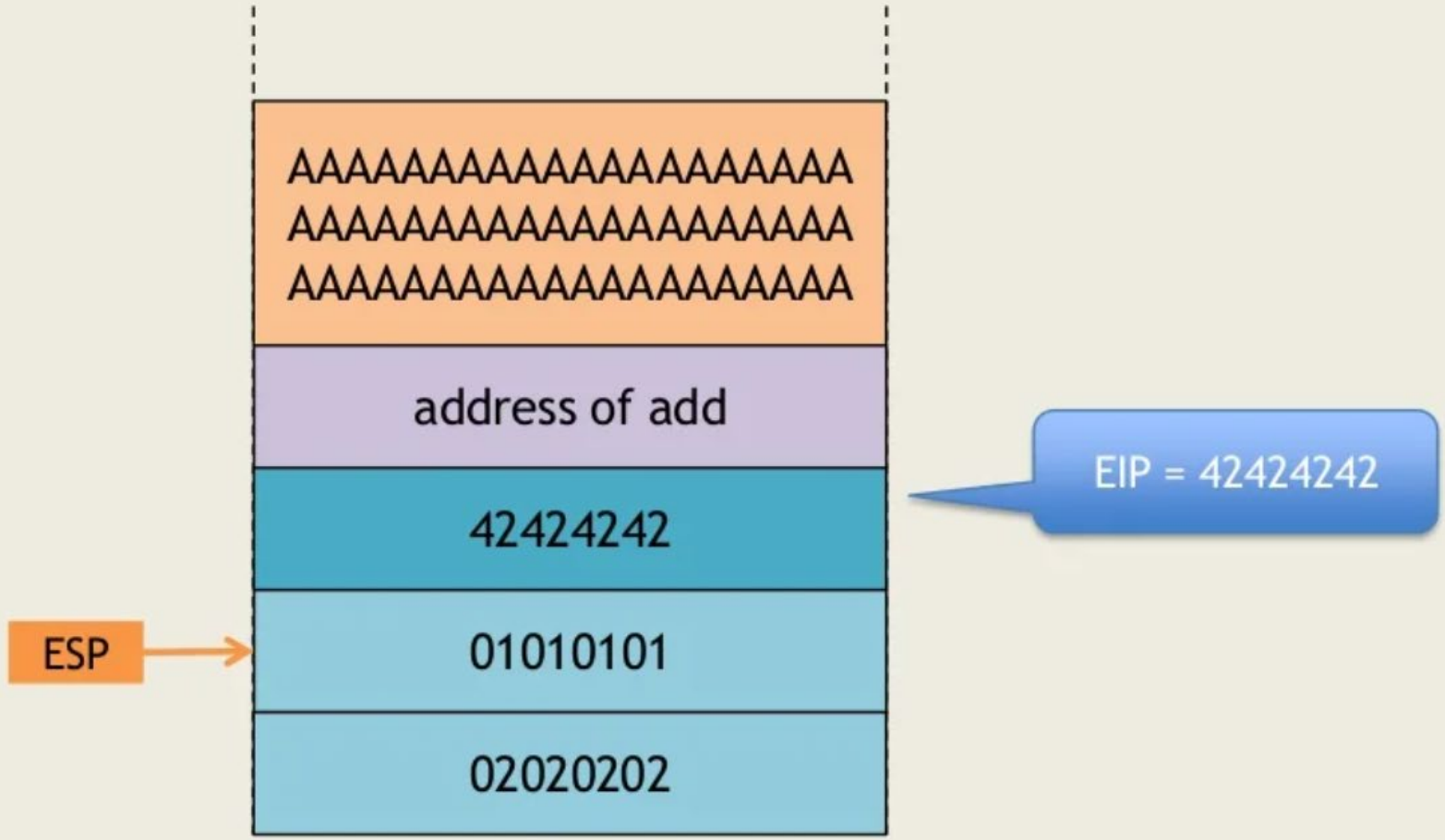




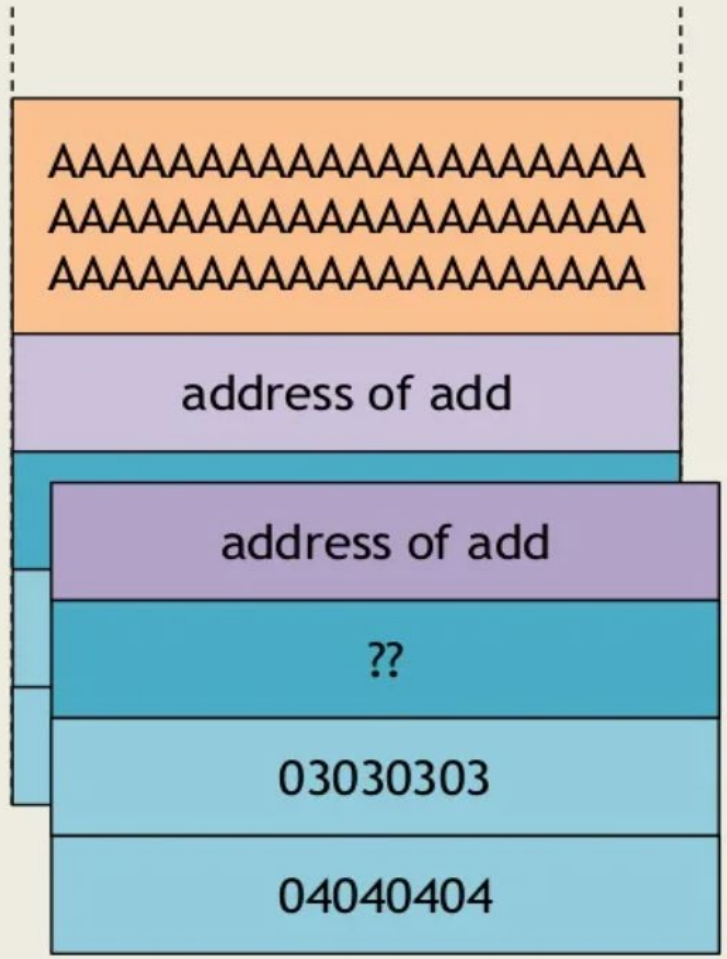
Chaining functions

- After add(01010101, 02020202), we want to run add(03030303, 04040404).
- How should we set up the frames?
- First, study the frame after add() returns.

After add() returns

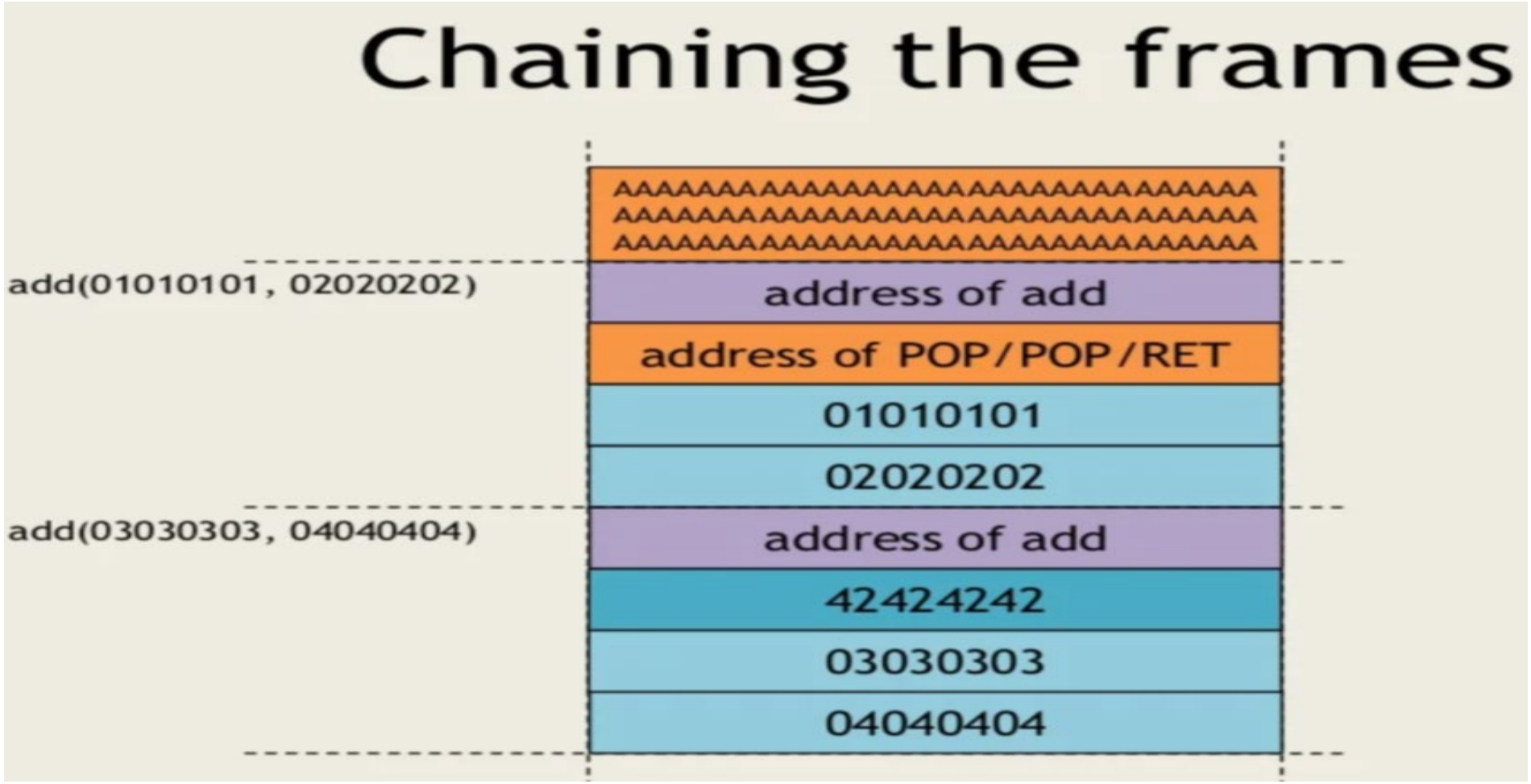


Where does the new frame go?



Where does the new frame go?

- We get only ONE chance at strcpy.
- How do we preserver params 01010101 and 02020202?
- We can only APPEND the second frame below our first frame.

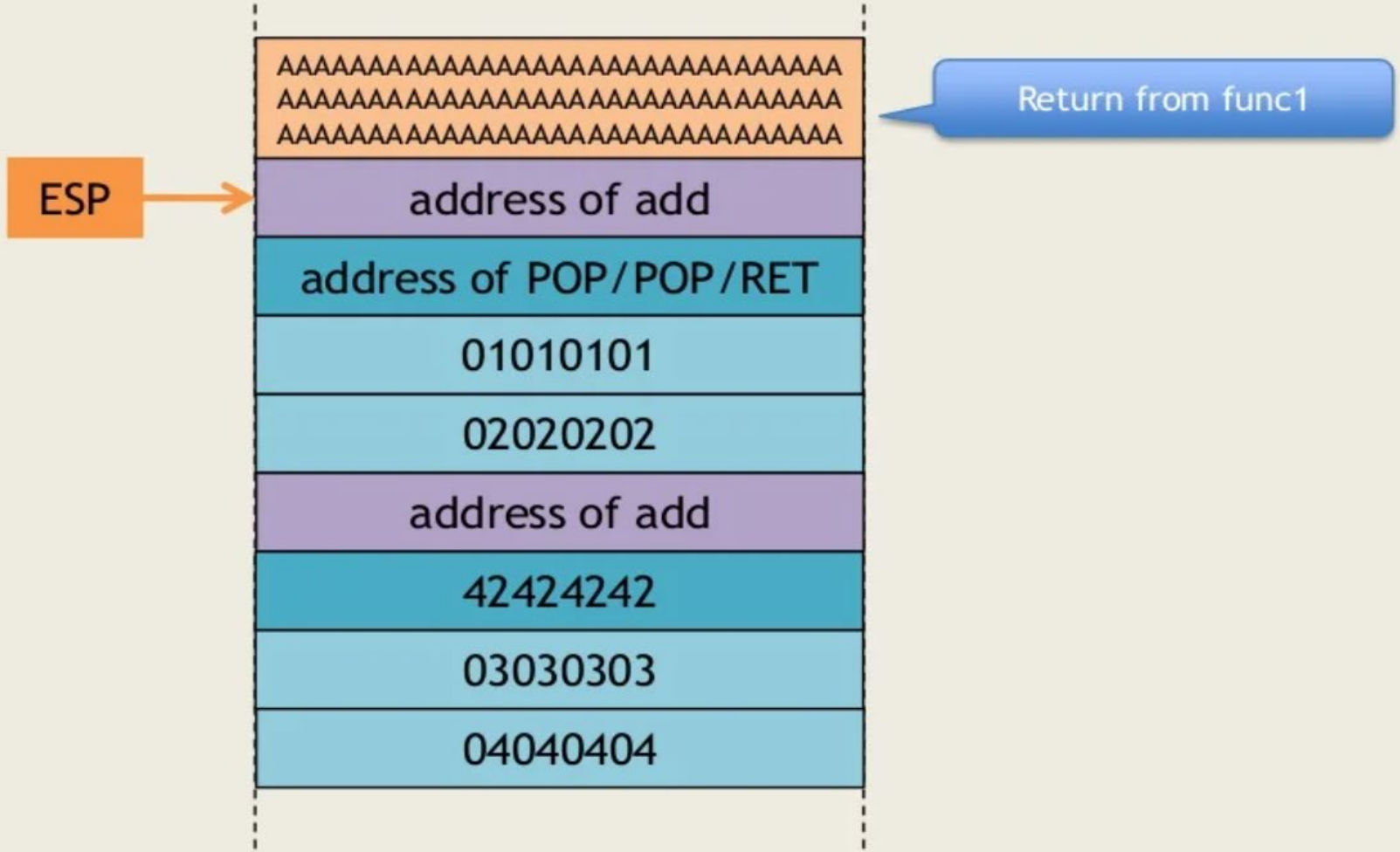


POP/POP/RET:

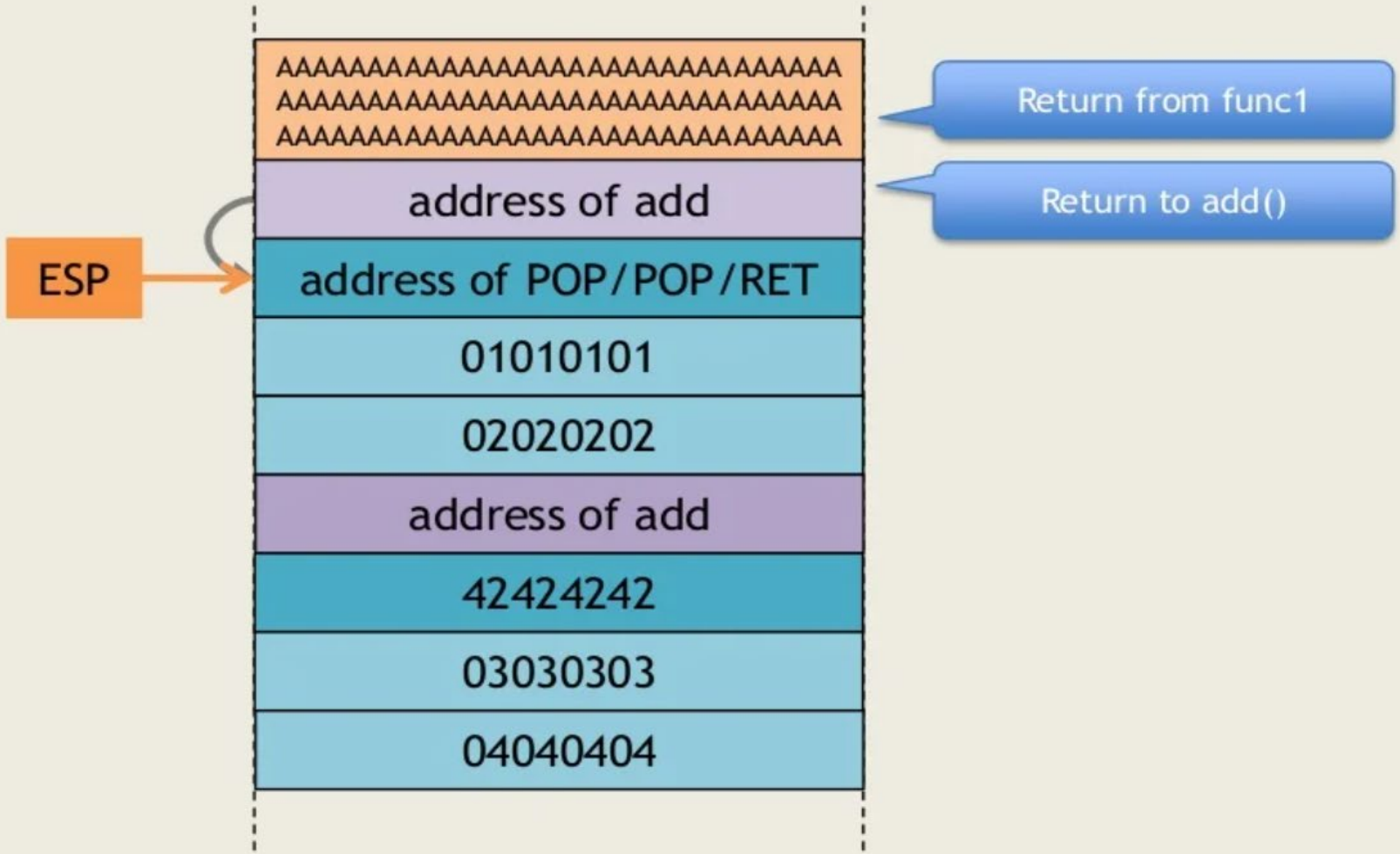
Move ESP two places towards higher addresses (with the two POPs) and we continue execution (with the RET) at the contents of the address ESP points to at this moment.

The buffer overflow set the Address of the POP POP RET instruction sequence (which is 08048422 in our example) at another part of the memory.

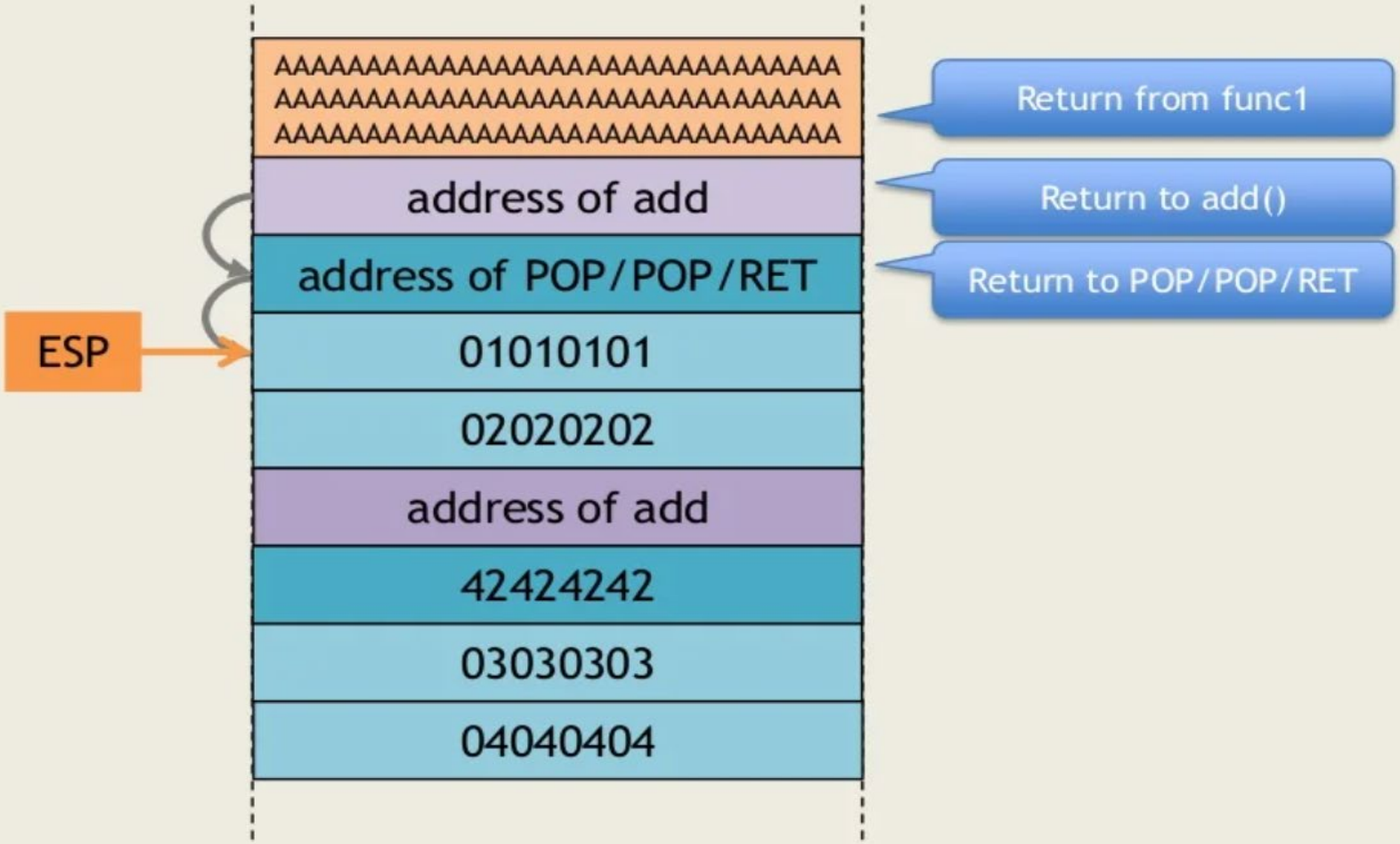
Keeping ESP in control



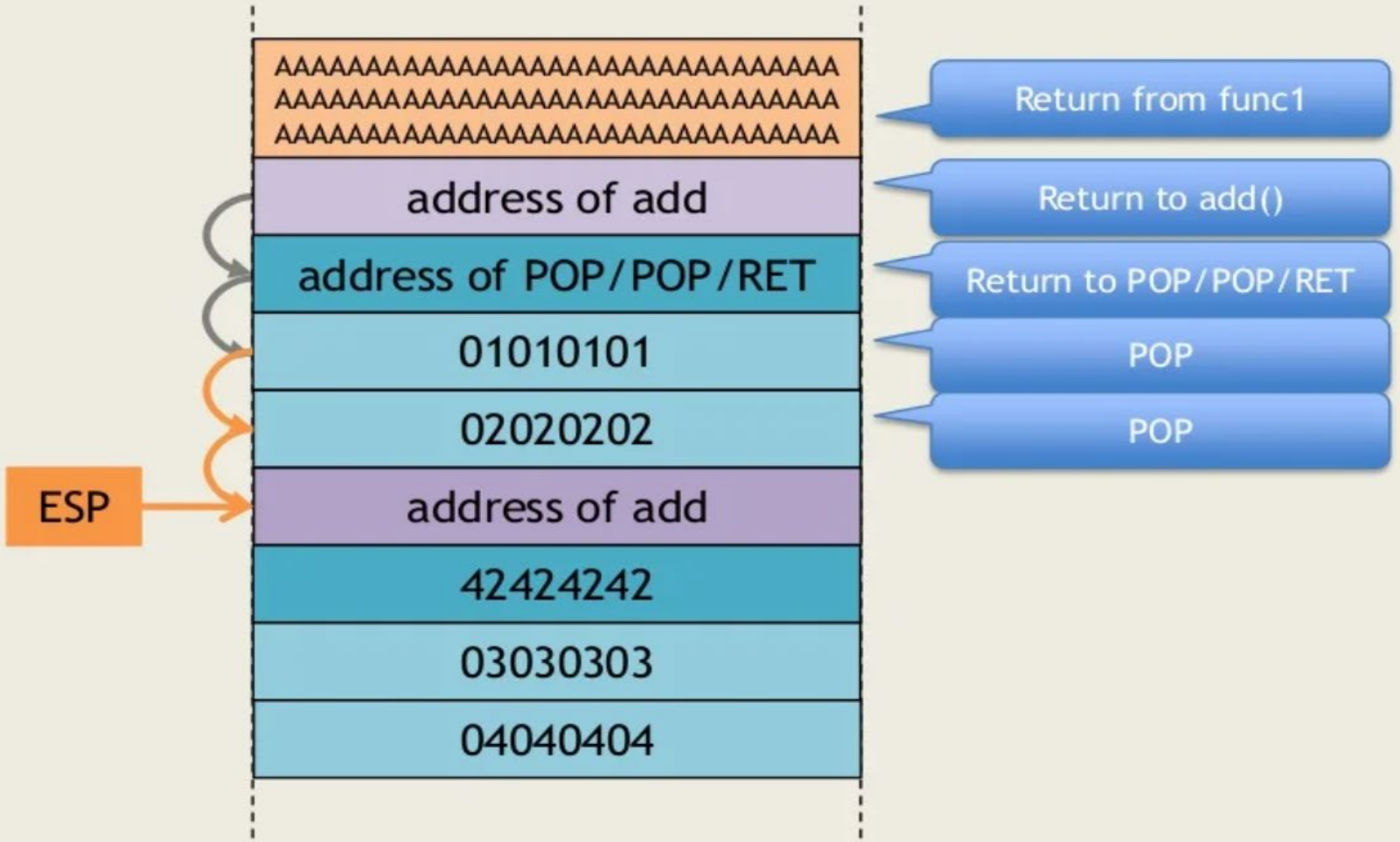
Keeping ESP in control



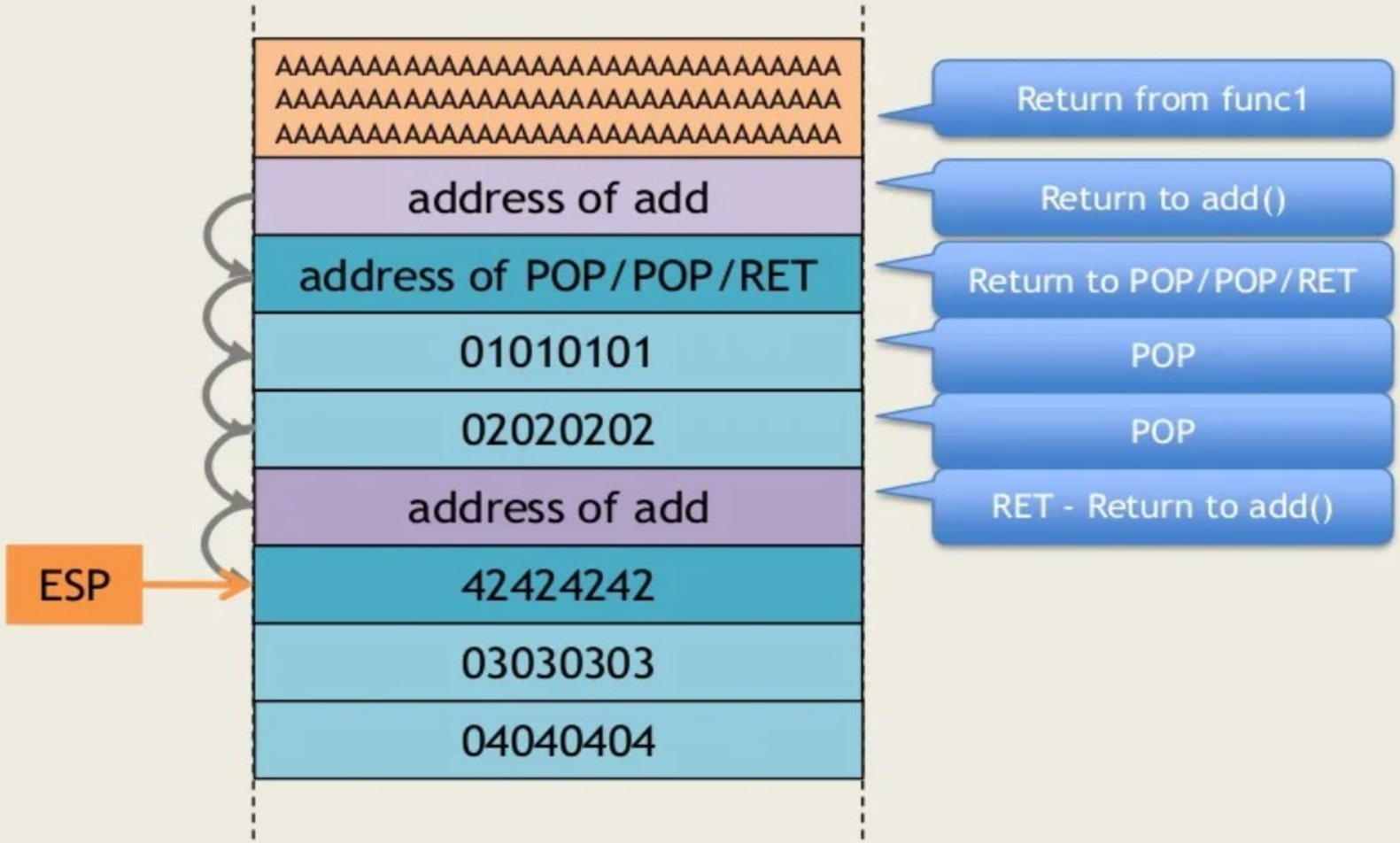
Keeping ESP in control



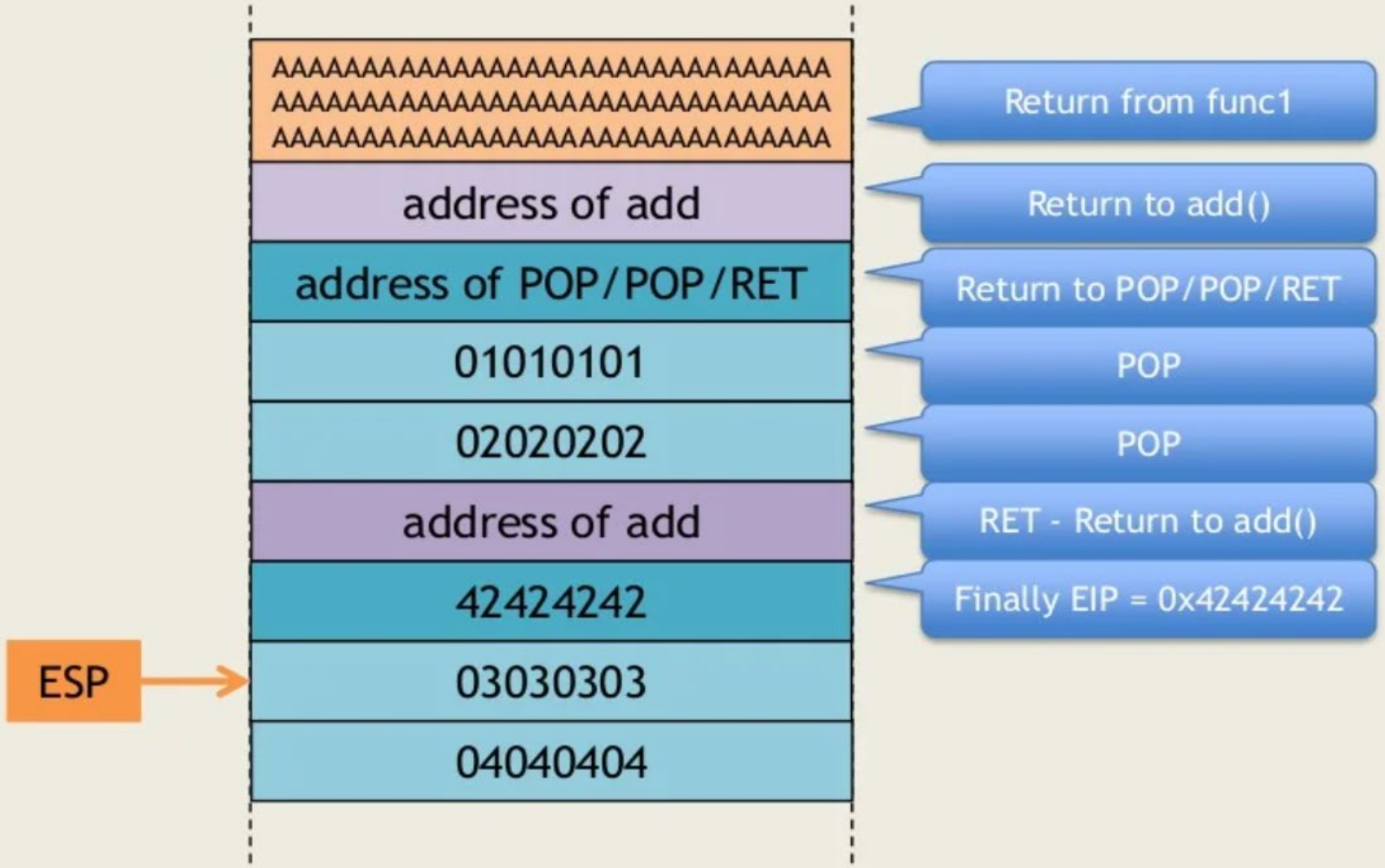
Keeping ESP in control



Keeping ESP in control

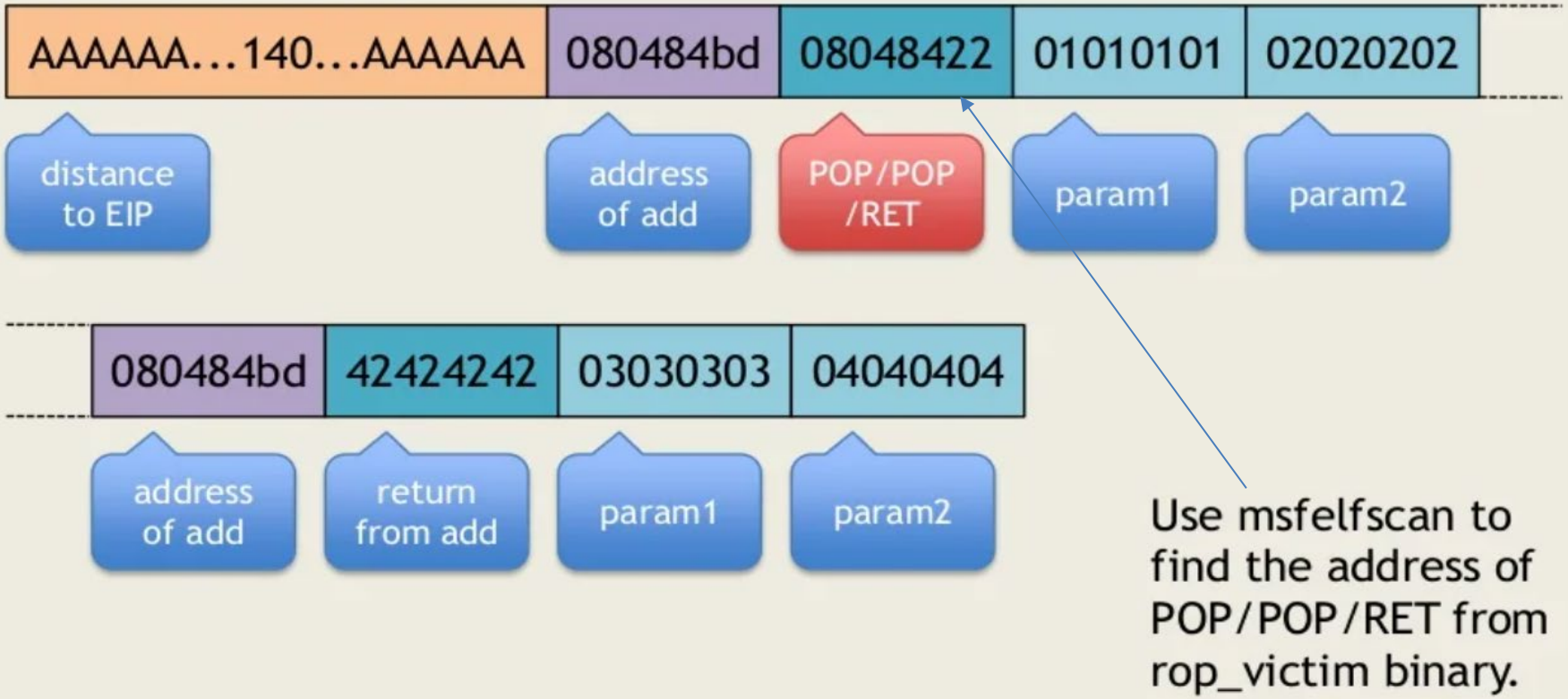


Keeping ESP in control



Chained Frames

- Create another buffer overflow payload:



Msfelfscan, is used to locate interesting addresses within executable and linkable format programs

Code Execution – The ROP Way

- Piece together snippets of code
- Gadgets – primitive operations, to be searched for within the process binary or shared libraries.

- Every gadget must end with a RET.

Defense Against Hijacking Program Control

What Makes a Process (a running Program) Safe?

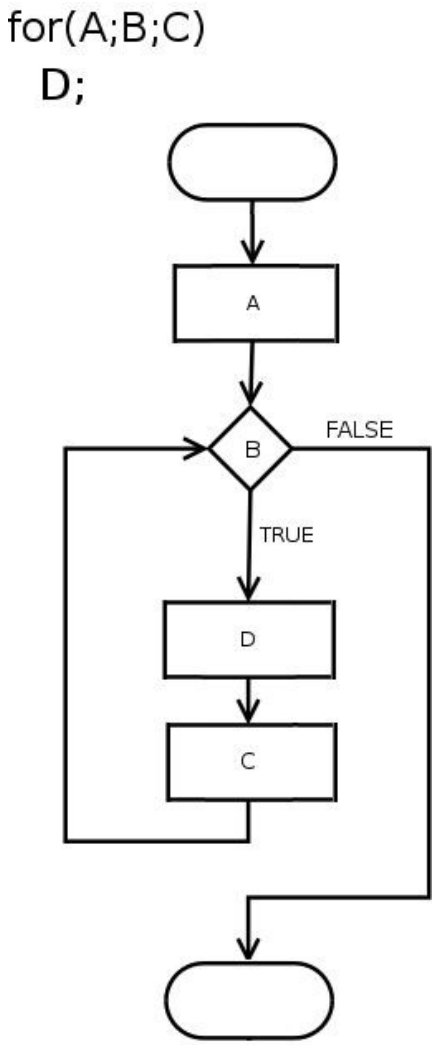
- Type safety: all function calls and operations have arguments of correct type
- Memory safety: all memory accesses are “correct”
  - Ensure array bounds are respected.
  - Avoid overwriting memory belonging to other processes.
  - Do not execute data as if it were code.
- Control-flow safety: ensure all control transfers are as intended by the original program
  - Avoid arbitrary jumps and refrain from calling library routines not used by the original program

Behavior Based Detection

Control Flow Integrity

Program Control Flow

- Unconditional jumps
- Conditional jumps
- Loops
- Subroutines
- Unconditional halt



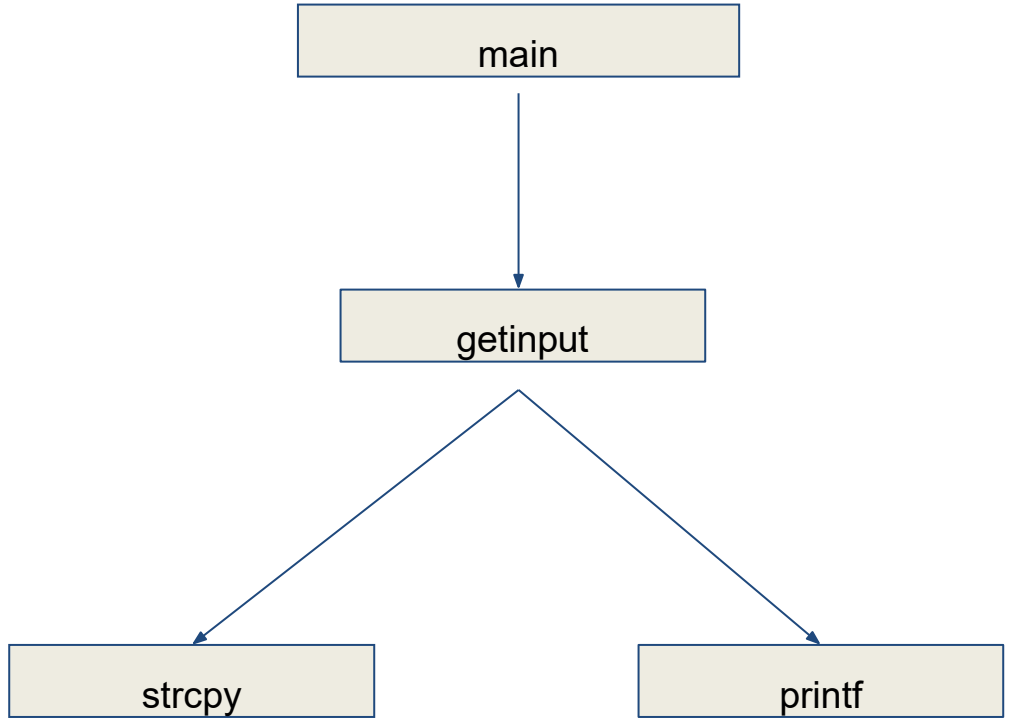
Simple Call Graph

```c
#include <stdio.h>
#include <string.h>

void getinput(char *input) {
     char  buffer[32];

     strcpy(buffer, input);
     printf("You entered: %s\n", buffer);
}

int main(int argc, char **argv) {
     getinput(argv[1]);
     return(0);
}
```



The fundamental problem:

Code is not executed in the intended way!

Adversary may get control over the program's flow.

How can we make sure that the program is executed in the intended way?

i.e., how to prevent an adversary from getting control over the program's flow.

Control-Flow Integrity (CFI)

Prevents a wide variety of malware attacks from redirecting the flow of execution (the control flow) of a program.

Why CFI?

- **CFI enforces integrity of a program's execution flow path**
- An undefined change in the path of execution by an adversary can be detected using CFI
- CFI efficiently detects and mitigates buffer overflow, Return Oriented Programming (ROP), return-to-libc attacks, etc.

CFI Overview

Execution must follow a path in a control flow graph (CFG) created **ahead of run time**.

"static"

Method:

- build CFG statically, e.g., at compile time
- instrument (rewrite) binary, e.g., at install time
  - add IDs and ID checks; maintain ID uniqueness
- perform ID checks at run time
  - indirect jumps shall have matching IDs

Control-Flow Integrity

- Execution must follow a path of a Control-Flow Graph (**CFG**)
- **CFG can be pre-computed from:**
  - source-code analysis
  - binary analysis
  - execution profiling
- But how can we enforce this extracted control-flow?

Behavior-based Detection

- Is there an efficient way to check if a program is performing the way we expect?
- This approach shall identify when a program is compromised and deviates from our expectations.

→ **we must define the expected program behavior.**

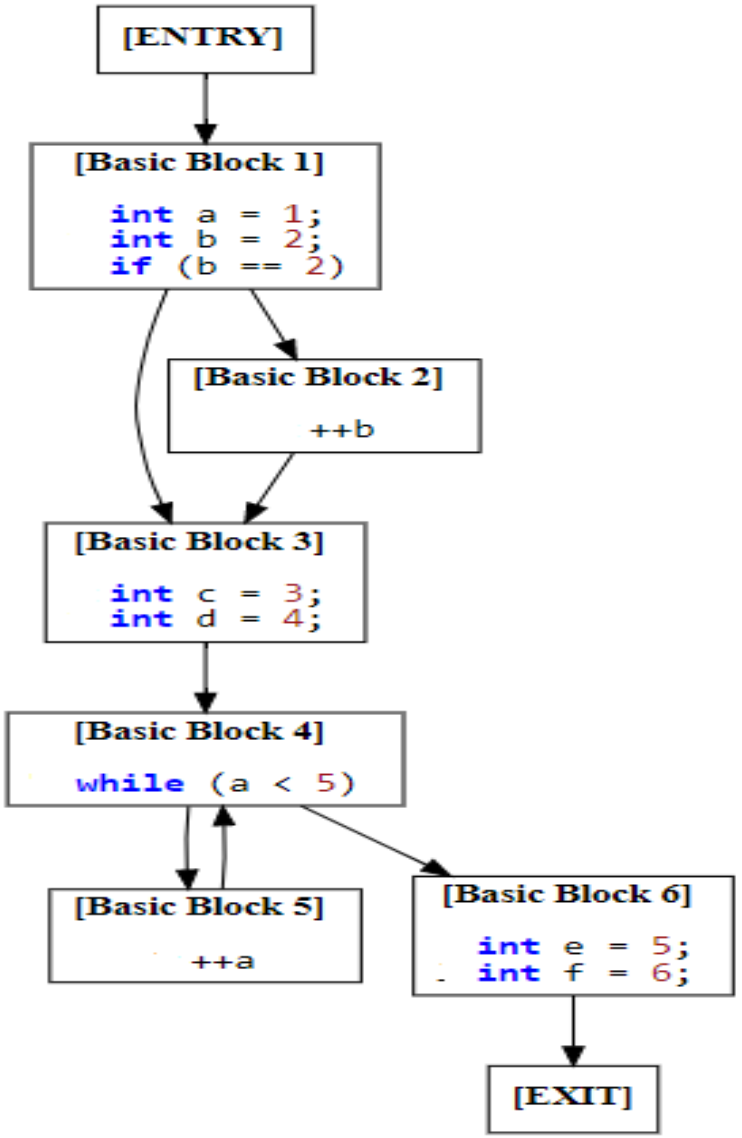
- Challenges:
  - ✓ Define “expected behavior”
  - ✓ Detect deviations from expectation efficiently
  - ✓ Avoid compromise of the detector

Behavior-Based Detection Through Control Flow Graphs from Basic Blocks

Basic Block:

A consecutive sequence of instructions, control is "straight" (no jump targets except at the beginning, no jumps except at the end)

```
int a = 1;
int b = 2;
if (b == 2)
    ++b;
int c = 3;
int d = 4;
while (a < 5)
    ++a;
int e = 5;
int f = 6;
```



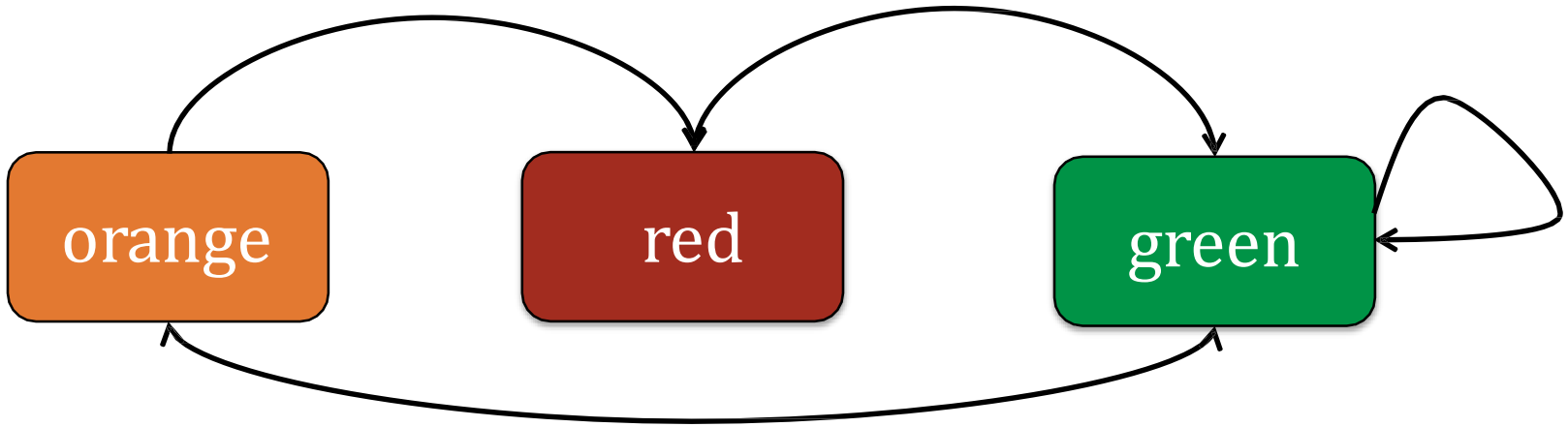
Call Graph: Example 1

Nodes are functions. There is an edge (v_i, v_j) if function v_i calls function v_j.

```
void orange()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
green();
...
}
```

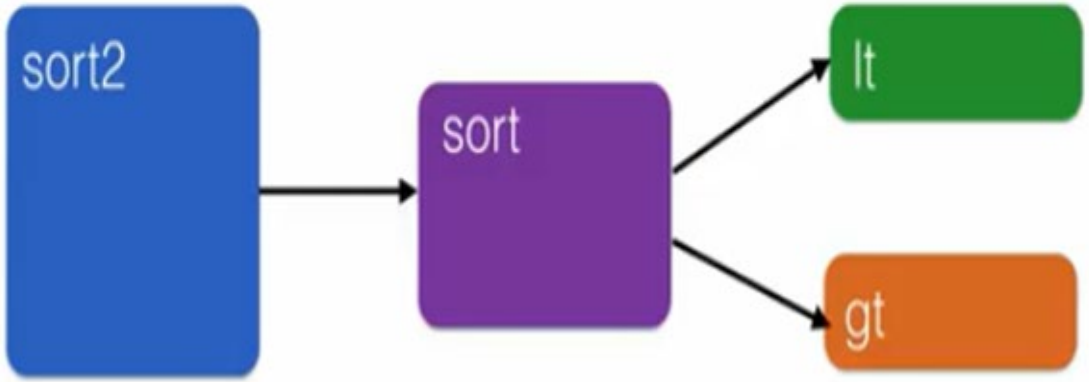
```
void green()
{
  green();
  orange();
}
```



Call Graph: Example 2

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



Which functions call other functions

Function sort2 takes two arrays, a and b, and a length.
It sorts one in ascending order, and the other in descending order.

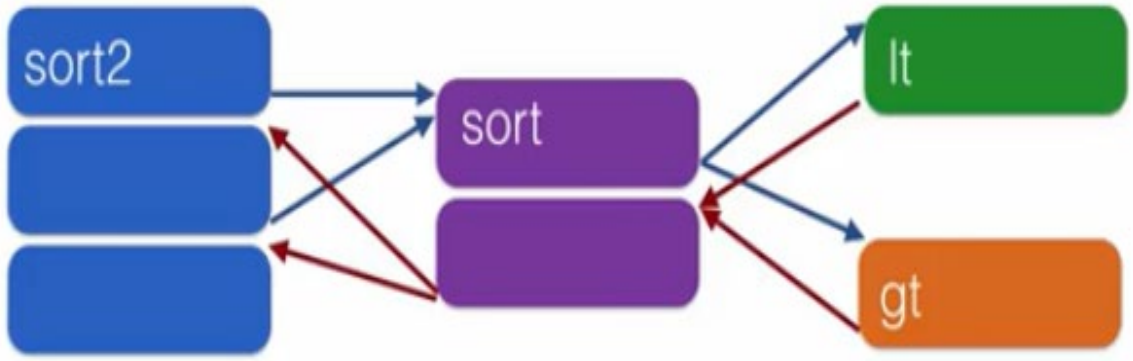
The program's call graph depicts the relationships between functions, indicating which functions invoke others.

sort2 calls sort, and sort will call lt and gt,

Control Flow Graph

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



*Break into **basic blocks***
*Distinguish **calls** from **returns***

CFG (Control Flow Graph) can be derived from a Call Graph

- The CFG is defined by distinguishing calls and returns in the program.
- Each function is broken into basic blocks
- Sort2 function calls sort. Then when it returns to sort2, it calls the second sort and then returns to sort2 again until it completes.

- Therefore, we can control the flow as follows:

  sort2 calls **sort** → **sort** calls **lt** → returns back to **sort** → returns back to **sort2**.

  Then **sort2** calls **sort** again → **sort** calls **gt** → returns back to **sort** → finally returns back to **sort2**.

How Control Flow Integrity (CFI) Works

- *Define "expected behavior":*

Control flow graph (CFG)

- *Detect deviations from expectation efficiently*

In-line reference monitor (IRM)

- *Avoid compromise of the detector*

Sufficient randomness, immutability

The expected behavior is defined by the control flow graph or CFG.

It does this by using an **In-line Reference Monitor or IRM**.

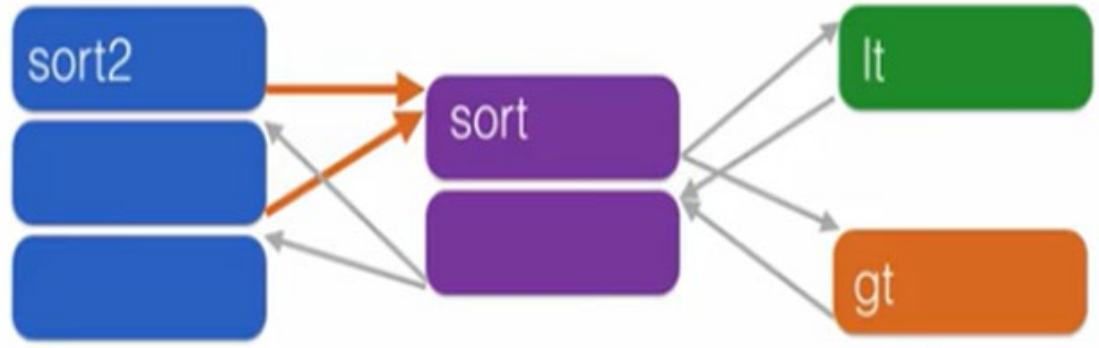
An **IRM** rewrites a program by inserting instructions to **check whether CFI attributes are maintained.**

To avoid the detector being compromised, we rely on sufficient randomness of the secrets used by CFI and on the immutability of the code.

Monitoring the Control of a CFI

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



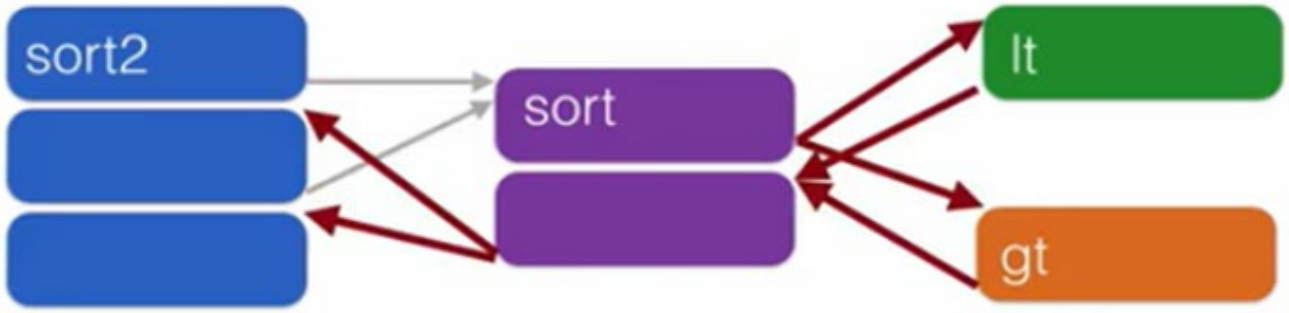
Direct calls (always the same target)

Direct calls to sort do not need to be monitored. Because they're just in the control, in the program text.

Monitoring the Control of a CFI (cont.)

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```



Indirect transfer (`call` *via register, or* `ret`)

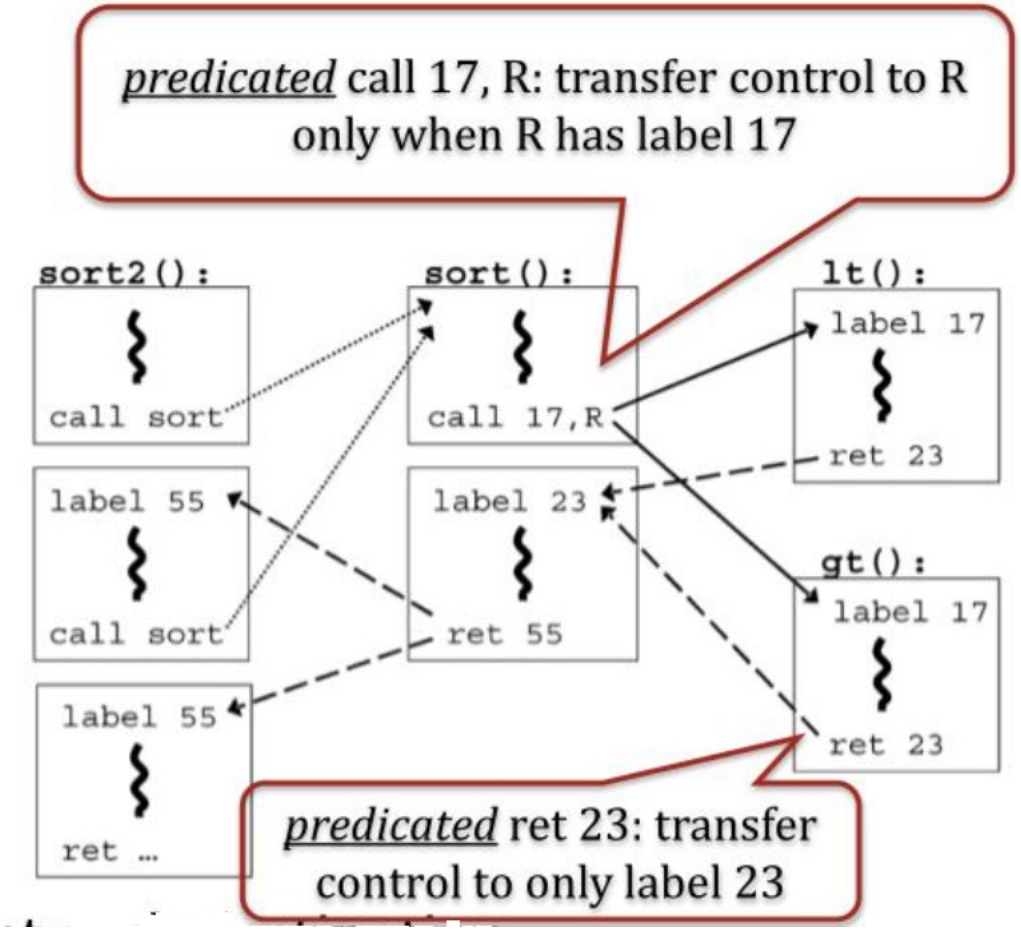
The indirect calls to lt and gt, and all of the returns need to be monitored because they are done through dynamic data on the stack.

- The returns are going to pop values off of the stack which is dynamically alterable.
- And the jumps, the calls of lt and gt, are through function pointers which are again pushed onto the stack.

How Does In-Line Monitor Work?

Implement the monitor in-line as a **program transformation (instrumentation)**

- Insert a label just before the target address of an indirect transfer.
- Insert binary code that, when run, checks whether the label of the target instruction matches the label of the possible destination
- If the label is not what we expect, then abort.



Source

Opcode bytes	Instructions		
FF E1	jmp	ecx	; computed jump

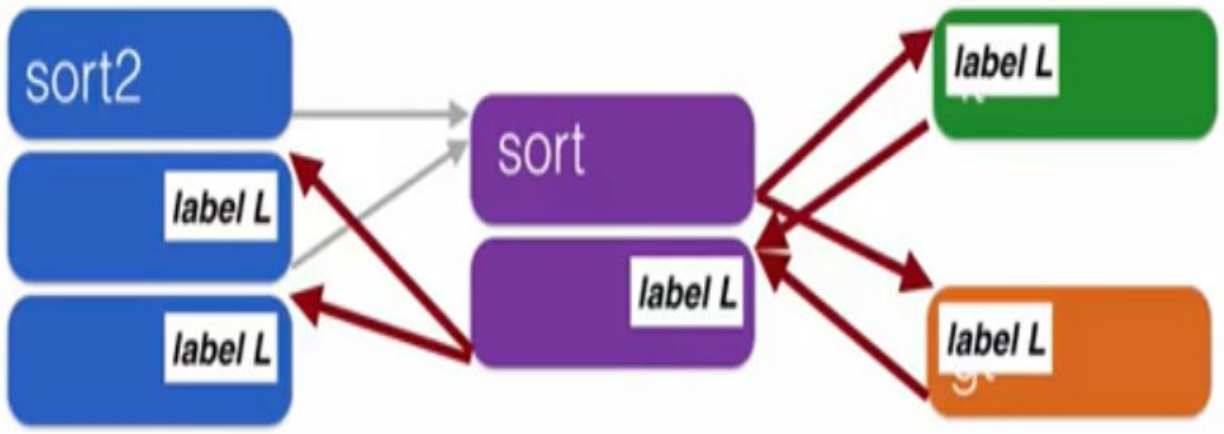
can be instrumented as

Opcode bytes	Instructions		
81 39 78 56 34 12	cmp	[ecx], 12345678h	; comp ID & dst
75 13	jne	error_label	; if != fail
8D 49 04	lea	ecx, [ecx+4]	; skip ID at dst
FF E1	jmp	ecx	; jump to dst

Destination

Opcode bytes	Instructions		
8B 44 24 04	mov	eax, [esp+4]	; dst
...			
78 56 34 12	; data 12345678h		; ID
8B 44 24 04	mov	eax, [esp+4]	; dst
...			

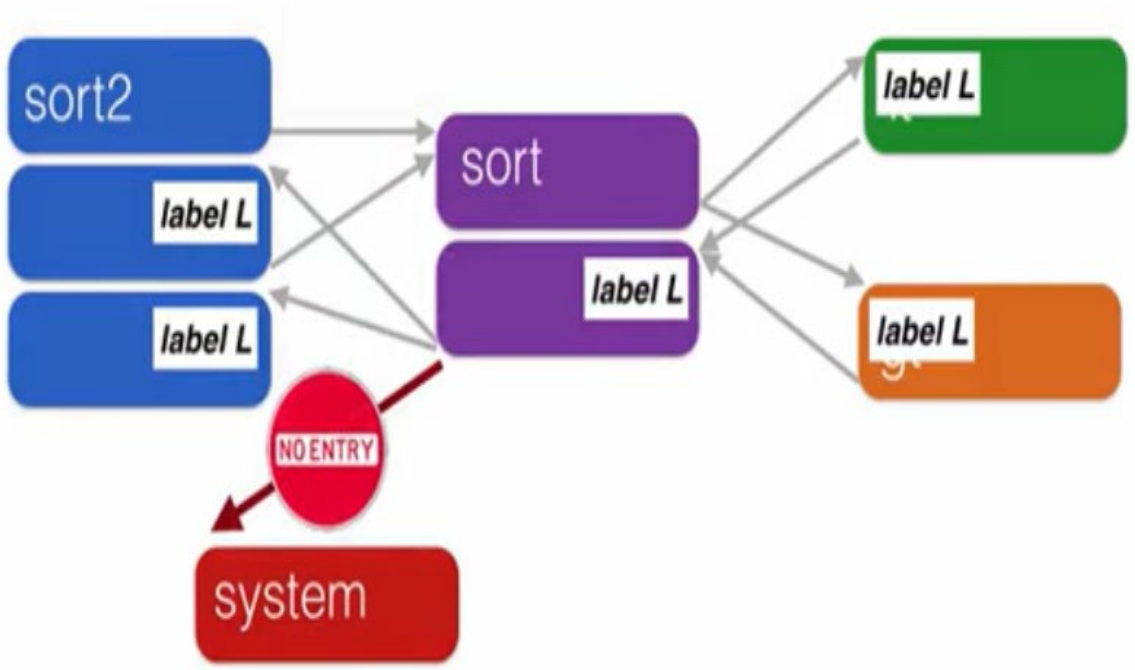
Simplest labeling



Use the same label at all targets

- The simplest possible labeling is to just pick a random value, label L.
  - ✓ Stick label L at all possible targets of indirect transfers in the entire program.
- The end of every arrow is the same label L.
- Every time returning from a function or calling through a function pointer, you need to check whether the target has a legal label L

Simplest labeling

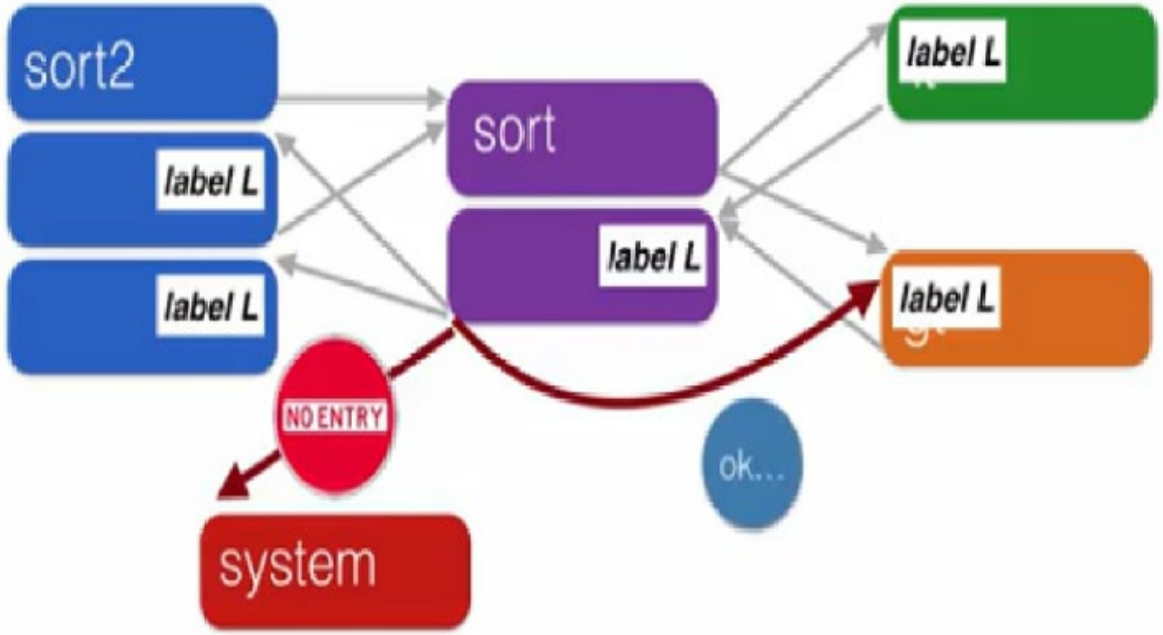


Use the same label at all targets

Blocks return to the start of direct-only call targets

- Prevent overriding a return address to go to a system command somewhere in the library.
- Because the system command does not have the label L preceding it.

Simplest labeling



Use the same label at all targets

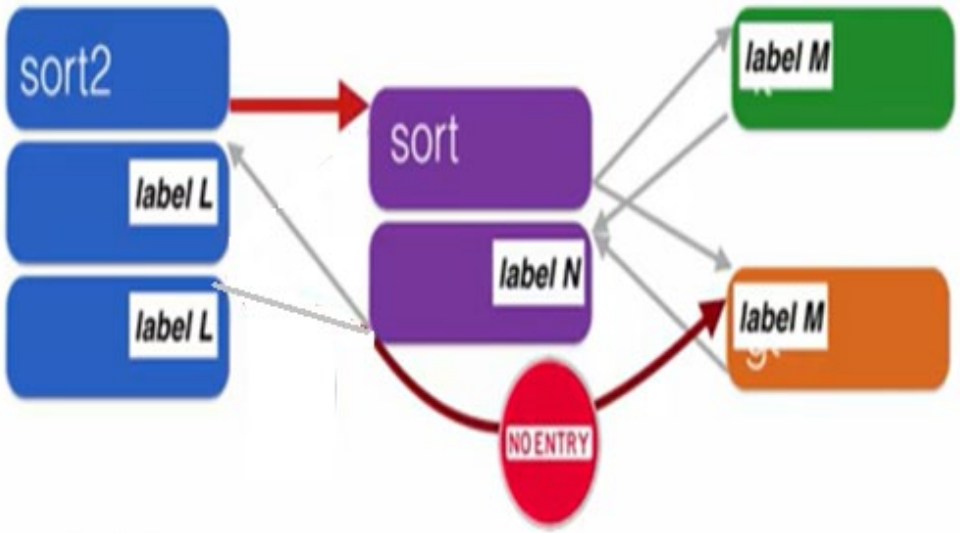
Blocks return to the start of direct-only call targets but not incorrect ones

- **On the other hand, this simplest labeling cannot prevent returns to other unintended locations with label L.**

  Example: the sort function should have returned back to the caller sort2. But instead, for example, the adversary modifies the return address to point to the beginning of the greater than function instead.

- This is legal according to CFI because it has the expected label.
- This is not the expected control flow in the program but is allowed according to the control flow graph.

Detailed labeling



Constraints:

- return sites from calls to `sort` must share a label (*L*)
- call targets `gt` and `lt` must share a label (*M*)
- remaining label unconstrained (*N*)

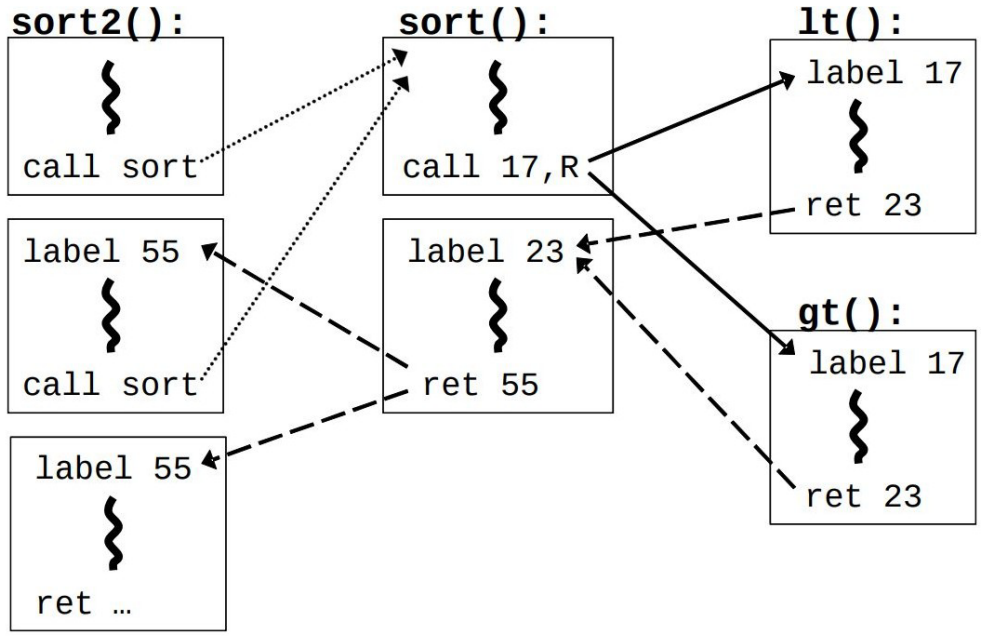
- To overcome this problem, a more detailed labeling of all of the targets of indirect transfers is needed.
- Solution:
  - ✓ The return calls from sort must all share the same shared label L. When sort returns, it doesn't know which is its caller block in sort2, and both caller blocks have the same label.
  - ✓ The call targets of gt and lt must share the same shared label M. The reason is that the sorting function does not know which function pointer will be given, so all these function pointers must have the same target label, in this case M.
  - ✓ The last label is not constrained, so it can be given a new label N.
  - ✓ The weird transfer you saw earlier will be rejected because label L is indeed not equal to label M.

Example: Enforcing CFI by In-line Reference Monitor w/ Detail Labeling

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



For Example:

L = 55

N = 23

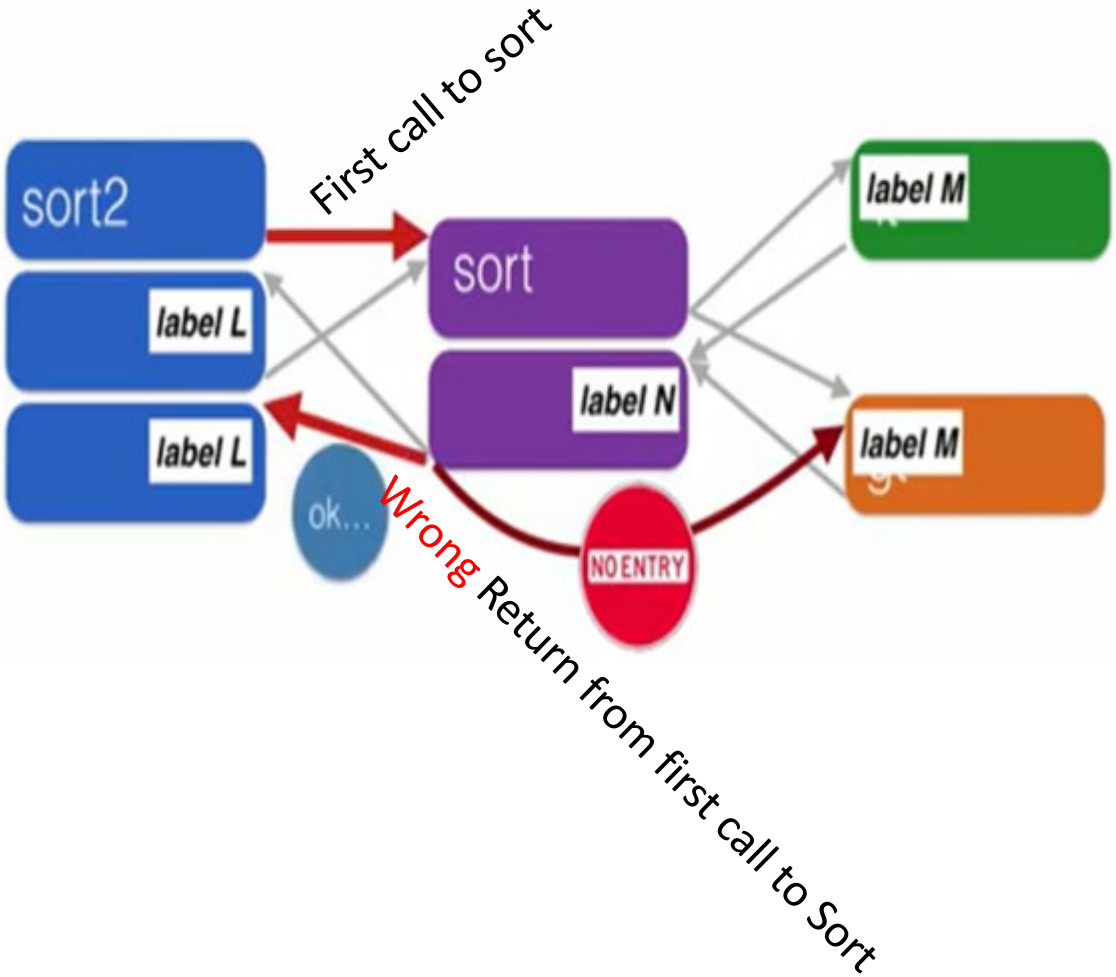
M = 17

……> direct call, no need to be monitor

——→ call to function pointer or function argument need to be monitor

- -- - -> return from a function, need to be monitor

Detailed labeling



Another problem with Detail Labeling:

It can't prevent the weird case of returning from the first call of sort function to the second label L, instead of returning to the first label L

It is allowed by the control flow graph of the program. But it's not clear why the adversary would gain anything from this, at least in this example.

Assumptions

- **Unique IDs**: ID conflicts can cause problems during ID checking, so IDs used in CFI must be unique.
- **Non writable code**: If the code memory area is writable, the CFI implementation code may become corrupted, so the code memory must not be writable.
- **Non executable DATA**: If the data portion is executable, an attacker can add executable malicious code to the data portion.

Can we defeat CFI?

- **Inject code** that has a **legal label**

  Won't work because we assume **non-executable data**

  i.e. injecting code to the overflow buffer will not work - nonexecutable data section

- **Modify code labels** to allow the desired control flow

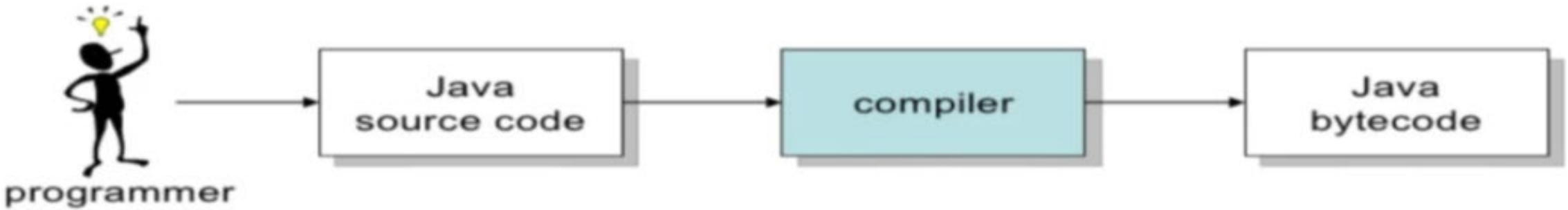
  Won't work because the **code is immutable**

Java Security

Outline

- components of Java
- Java security models
- main components of the Java security architecture
  - class loaders
  - byte code verification
  - the Security Manager

Development lifecycle



notes

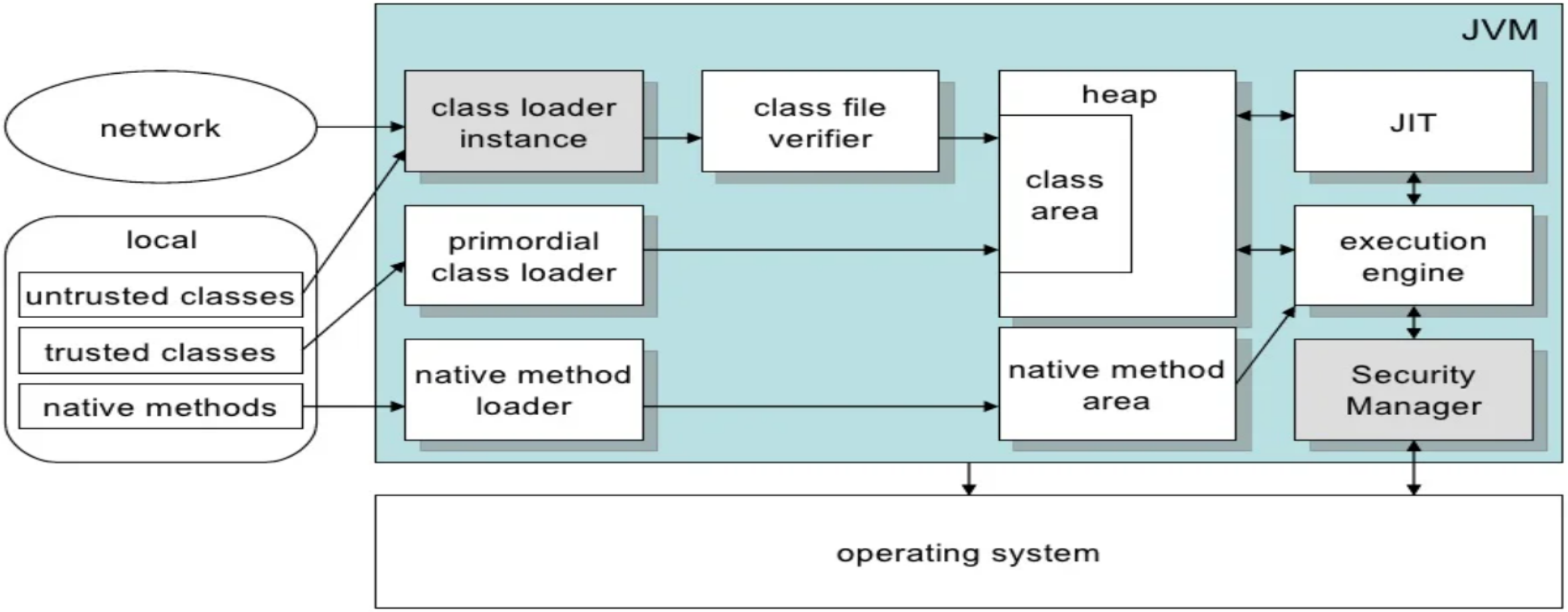
- Java is *compiled* and *interpreted*
  - source code is compiled into bytecode (low-level, platform independent code)
  - bytecode is interpreted (real machine code produced at run time)

- dynamic linking (no link phase at compile time)
  - program consists of class definitions
  - each class is compiled into a separate class file
  - classes may refer to each other, references are resolved at run-time

Java language features

- object-oriented
- multi-threaded
- strongly typed
- exception handling
- very similar to C/C++, but *cleaner* and *simpler*
  - no more struct and union
  - no more (stand alone) functions
  - no more multiple inheritance
  - no more operator overloading
  - no more pointers
- garbage collection
  - objects no longer in use are removed automatically from memory

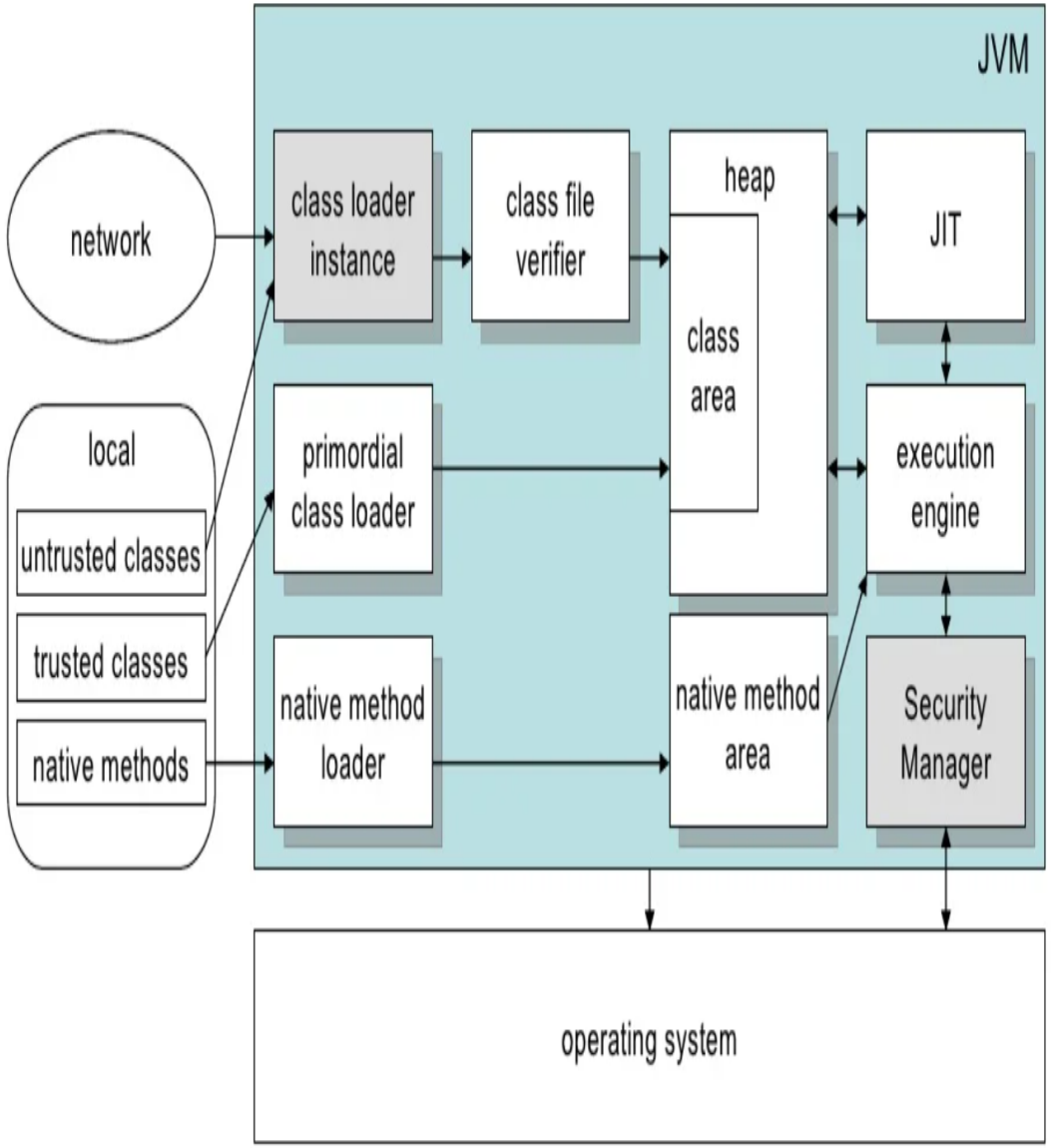
The Java Virtual Machine (JVM)



A native method is a Java method (either an instance method or a class method) whose implementation is also written in another programming language such as C/C++.

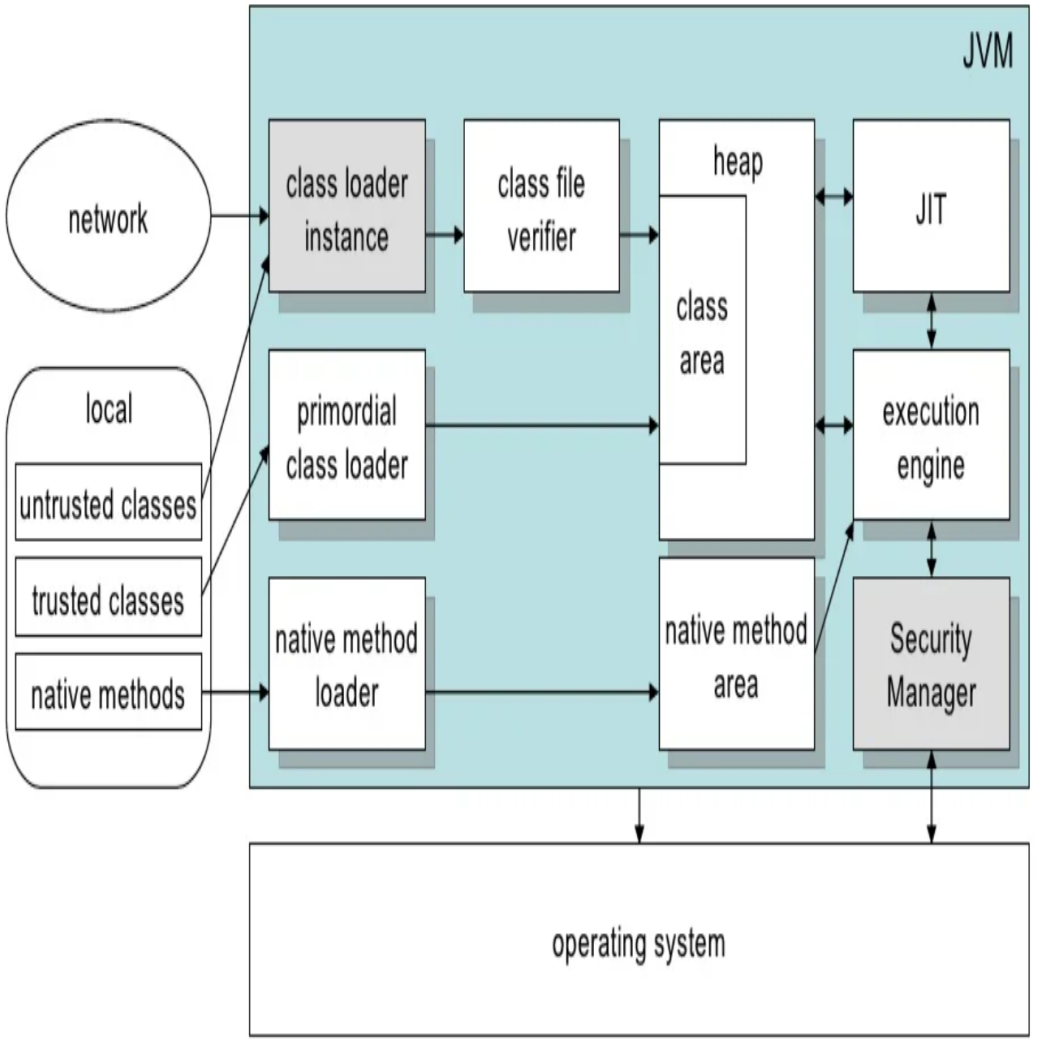
JVM cont'd

- class loaders
  - locate and load classes into the JVM
  - primordial class loader
    - loads trusted classes (system classes found on the boot class path)
    - integral part of the JVM
  - class loader instances
    - instances of java.net.URLClassLoader (which extends SecureClassLoader)
    - load untrusted classes from the local file system or from the network and passes them to the class file verifier



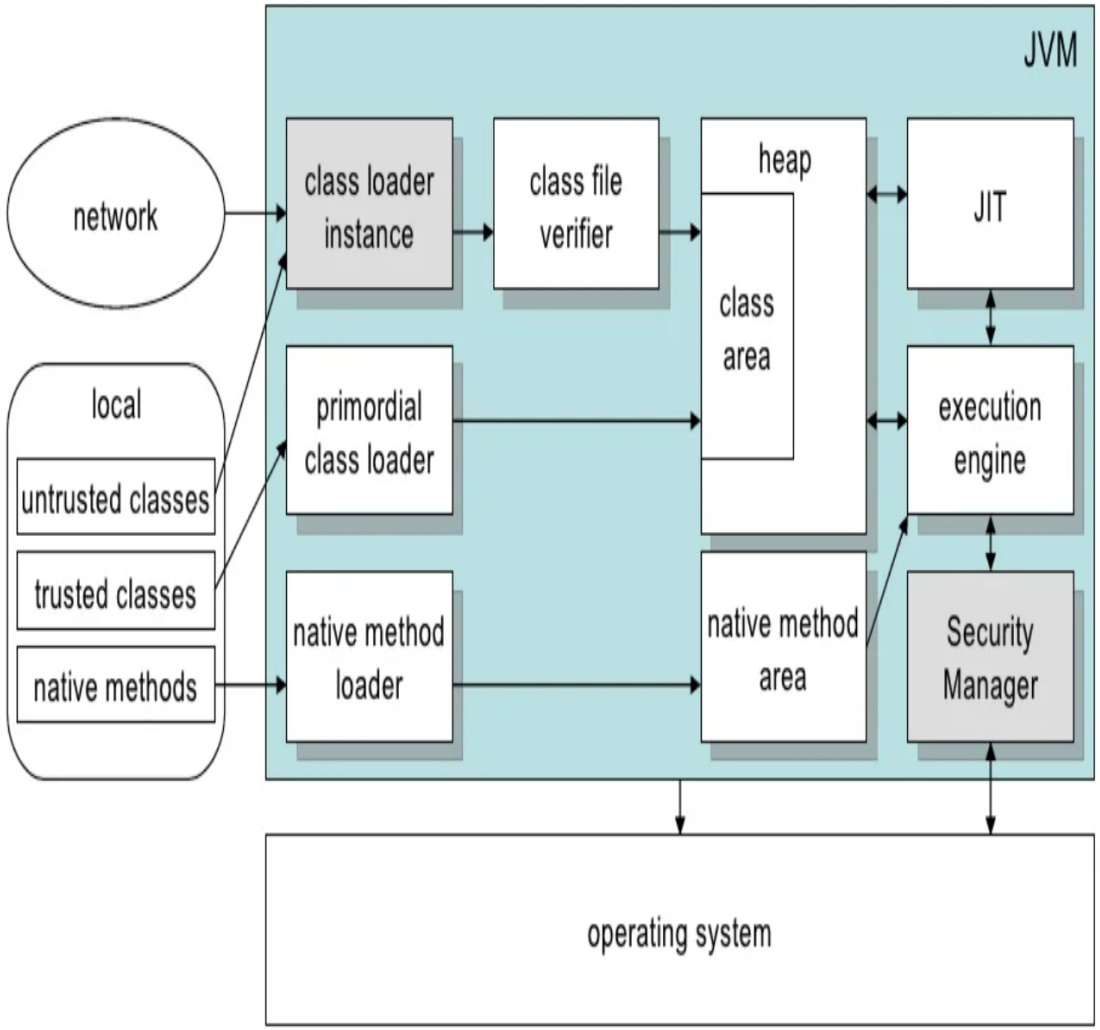
JVM cont'd

- native method loader
  - native methods are needed to access some of the underlying operating system functions (e.g., graphics and networking features)
  - once loaded, native code is stored in the native method area for easy access
- the heap
  - memory used to store objects during execution
  - how objects are stored is implementation specific
- execution engine
  - a virtual processor that executes bytecode
  - has virtual registers, stack, etc.
  - performs memory management, thread management, calls to native methods, etc.



JVM cont'd

- **Security Manager**
  - enforces access control at run-time (e.g., prevents applets from reading or writing to the file system, accessing the network, printing, ...)
- **JIT – Just In Time compiler**
  - performance overhead due to interpreting bytecode
  - translates bytecode into native code on-the-fly
    - works on a method-by-method basis
    - the first time a method is called, it is translated into native code and stored in the class area
    - future calls to the same method run the native code
  - all this happens after the class has been loaded and verified



Java security models

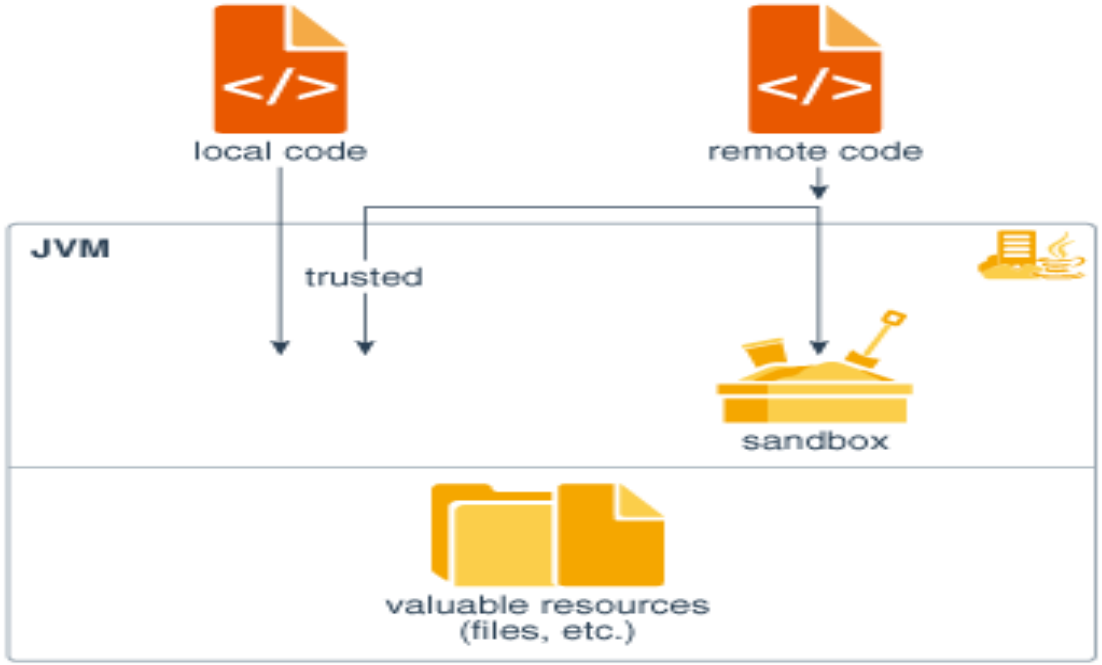
- the need for Java security
- the sandbox (Java 1.0)
- the concept of trusted code (Java 1.1)
- fine grained access control (Java 2)

The need for Java security

- code mobility can be useful
  - may reduce bandwidth requirements
  - improve functionality of web services
- but downloaded executable content is dangerous
  - the source may be unknown hence untrusted
  - hostile applets may modify or destroy data in your file system
  - hostile applets may read private data from your file system
  - hostile applets may install other hostile code on your system (e.g., virus, back-door, keyboard sniffer, ...)
  - hostile applets may try to attack someone else from your system (making you appear as the responsible for the attack)
  - hostile applets may use (up) the resources of your system (DoS)
  - all this may happen without you knowing about it

The sandbox

The Java security model is based on a customizable "sandbox" in which Java programs can run safely, without potential risk to systems or users





- introduced in Java 1.0
- local code had unrestricted access to resources
- downloaded code (applet) was restricted to the sandbox
  - cannot access the local file system
  - cannot access system resources,
  - can establish a network connection only with its originating web server

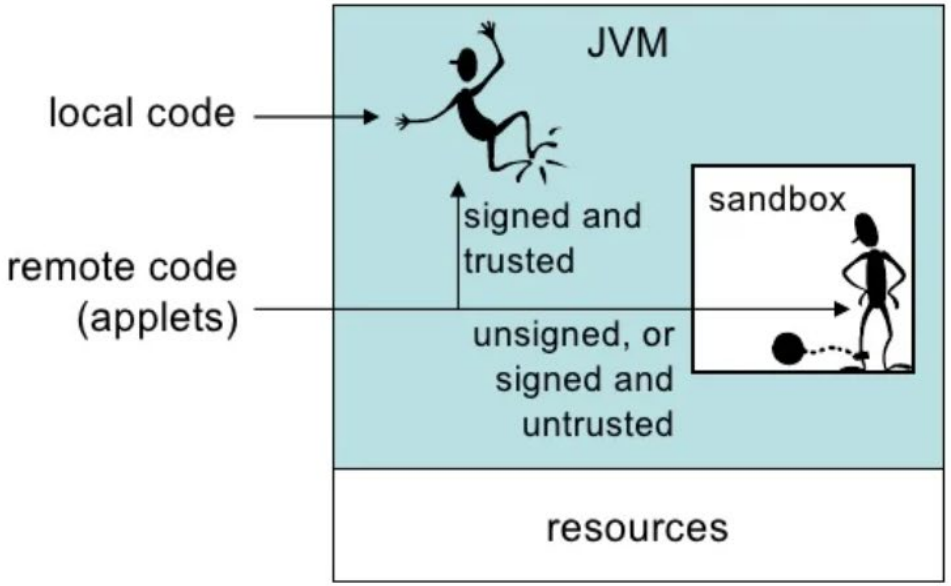
JDK 1.0 Sandbox

- Trusted code can read, write and delete any file, start and kill threads and open, use and close socket connections without any restrictions.

- Untrusted code cannot access any files, threads are hidden and only socket connections to the applet's origin server are allowed.

The concept of trusted code

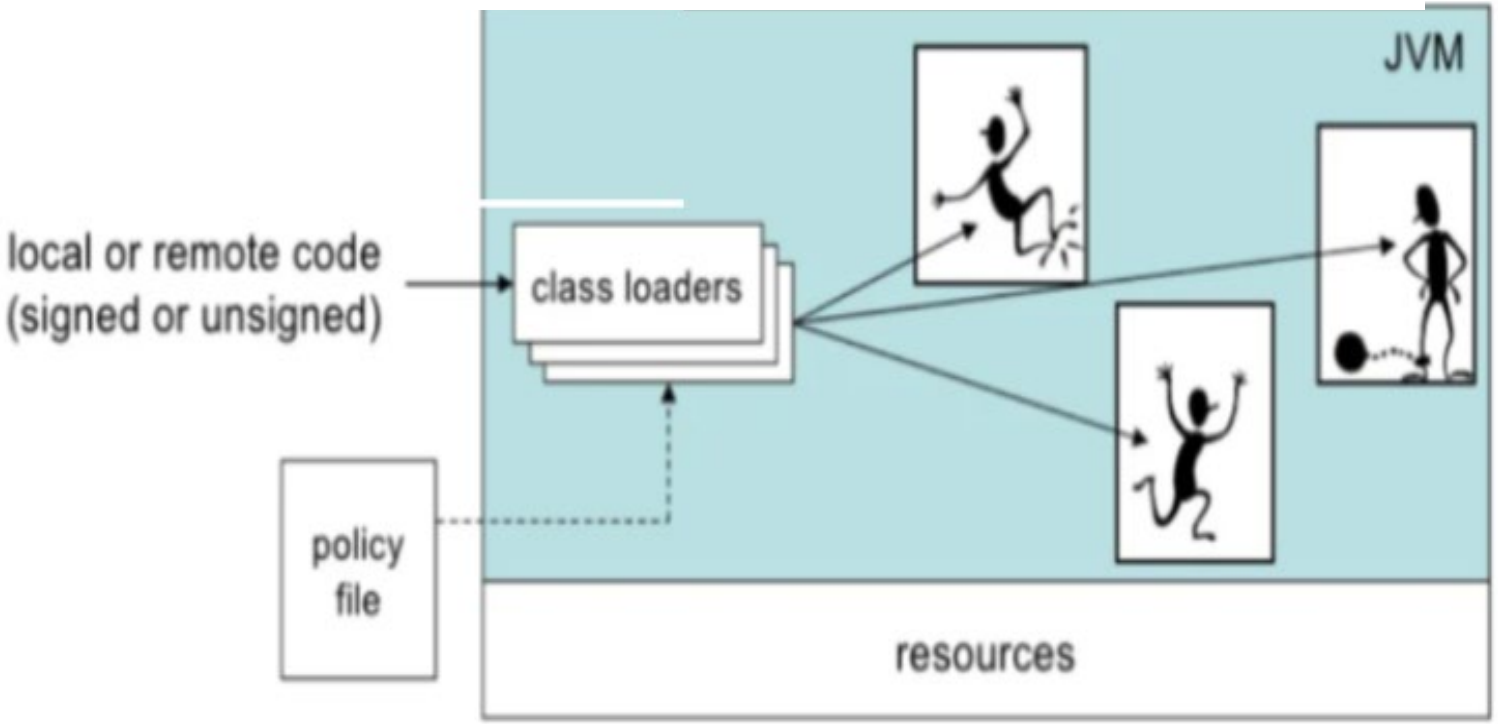
idea: applets that originate from a trusted source could be trusted



Signed applets: containing a signature that the browser should verify through a remotely running, independent certificate authority server (e.g. Thawte, Symantec, GoDaddy, …)

- **introduced in Java 1.1**
- **applets could be digitally signed**
- **unsigned applets and applets signed by an untrusted principal were restricted to the sandbox**
- **local applications and applets signed by a trusted principal had unrestricted access to resources**

Fine grained access control



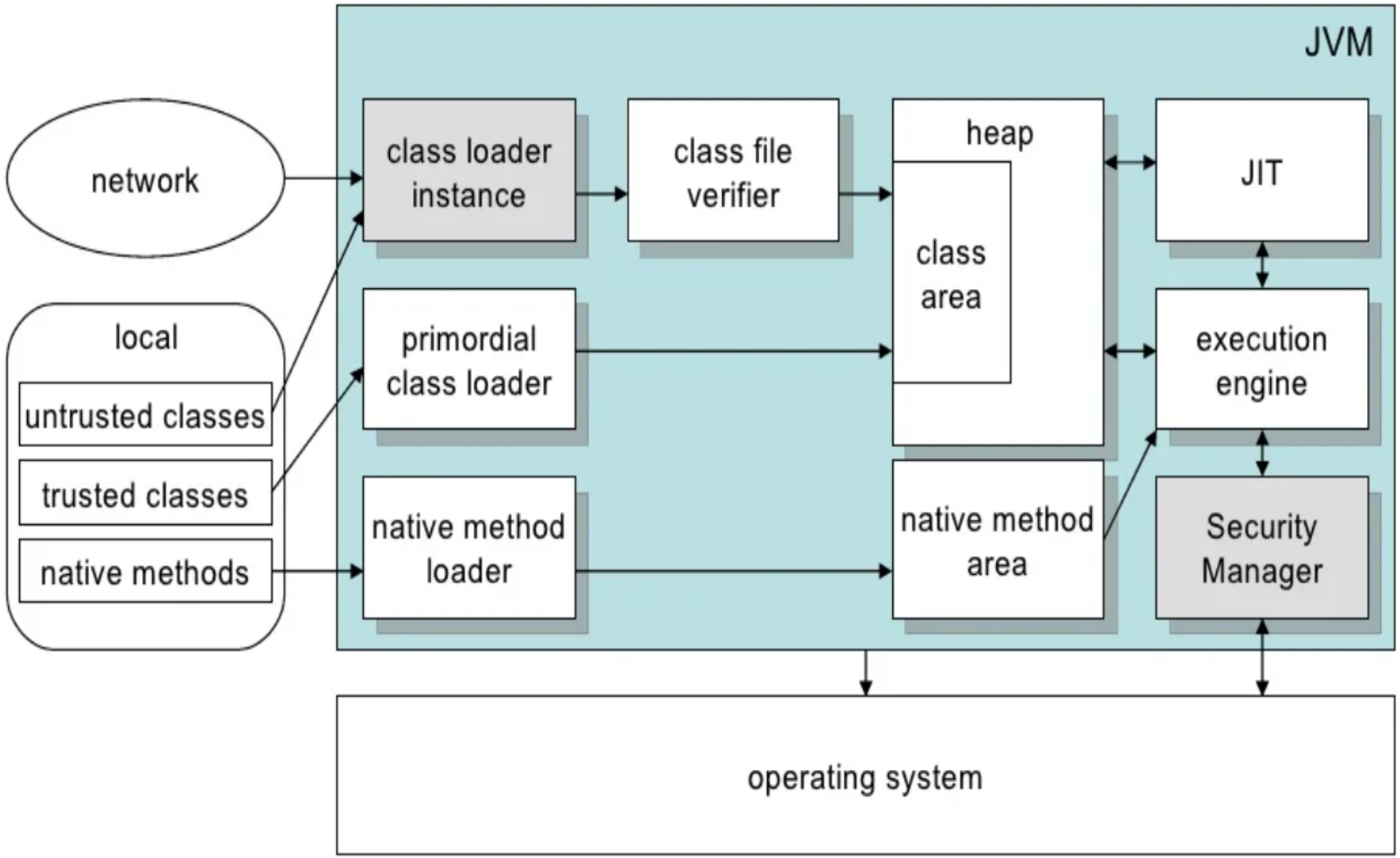
What is a POLICY file?

- A **POLICY file** is a configuration file used by **Java** Runtime Environment (**JRE**) to determine the granted permissions for each Java program.

- Introduce in Java 2
- Idea: Every code (remote or local) has access to the system resources based on what is defined in the Policy file.
- A protection domain is an association of a code source and the permissions granted
- Protection domains are established (determined) by the class loader
- Permissions granted to a code source are specified in the policy file

The three pillars of Java security

- the Security Manager
- class loaders
- the bytecode verifier



The Security Manager

- ensures that the permissions specified in the policy file are not overridden
- implements a checkPermission() method, which
  - returns a yes or a no (based on the code source and the permissions granted for that code source in the policy file)

Class loaders

Java packages are namespaces.

- separate name spaces
  - classes loaded by a class loader instance belong to the same name space
  - a class in one name space cannot access a class in another name space

  → classes from different Web sites cannot access each other
- establish the protection domain (set of permissions) for a loaded class
- enforce a search order that prevents trusted system classes from being replaced by classes from less trusted sources

Bytecode Verifier

It checks the code to ensure that:

- **Variables are initialized before they are used.**
- **Rules for accessing private data and methods are not violated.**
- **Local variable accesses fall within the runtime stack.**
- **The runtime stack does not overflow.**
- **No "illegal" data conversions will occur (i.e., the verifier will not allow integers to serve as pointers).**
  - ➢ **This ensures that variables will not be granted access to restricted memory areas.**
- **Bytecode instructions will have appropriately-typed parameters**

Java Security APIs

- Separate packages that are now included as part of JDK 1.4
  - JCE - Java Cryptography classes
  - JSSE - Java Secure Sockets Extension
  - JAAS - Java Authentication and Authorization Services
  - More..

Web Scripting Security

Readings for This Lecture

- **Http Cookie**
  https://en.wikipedia.org/wiki/HTTP_cookie
- **Same Origin Policy**
  https://en.wikipedia.org/wiki/Same-origin_policy
- **Cross-site Scription**
  https://portswigger.net/web-security/cross-site-scripting

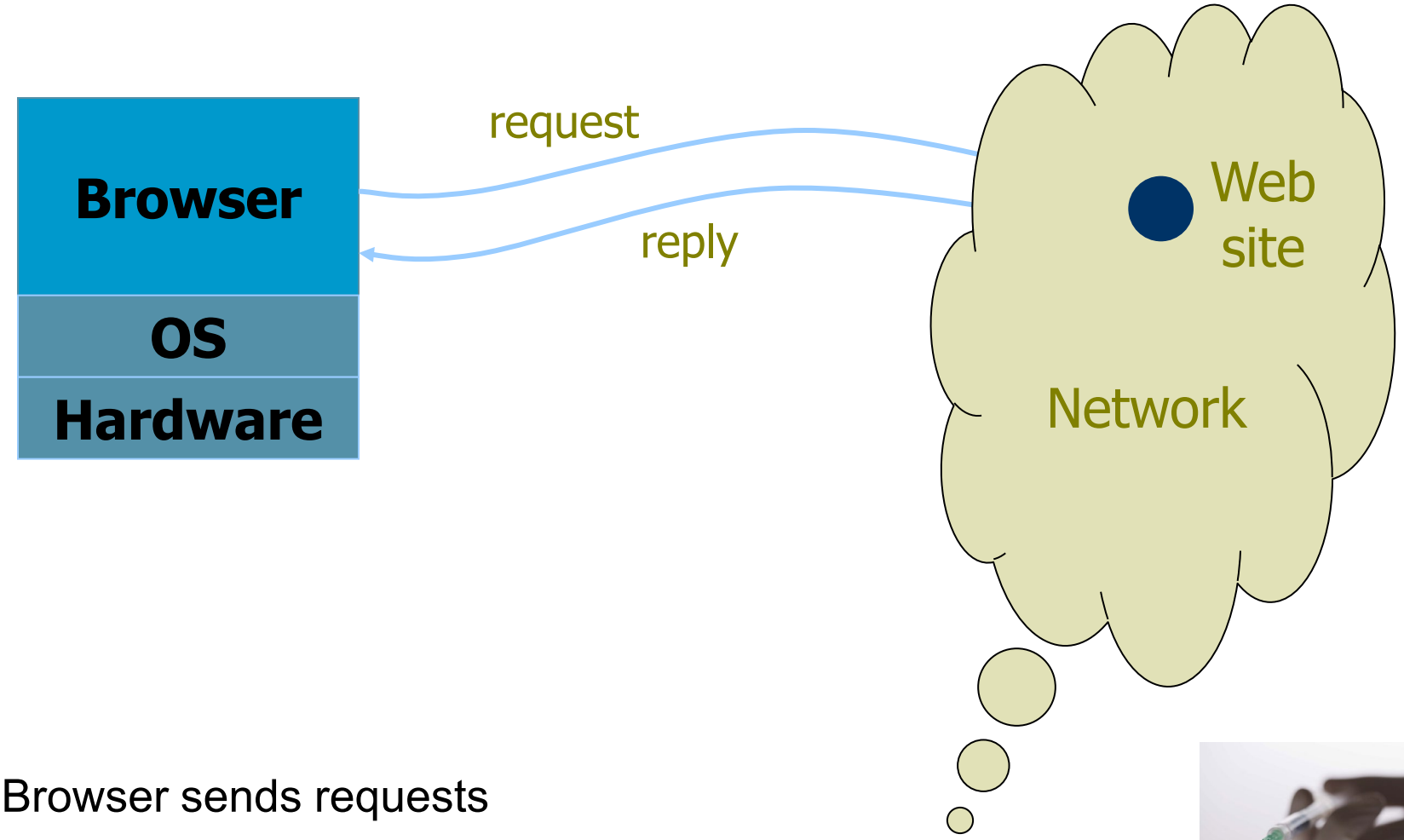
Agenda

- Web Security/Privacy Issues
- HTTP: HyperText Transfer Protocol
- Cookies to Store State Info
- Web Authentication via Cookies
- Cross Site Scripting (XSS)
- Reflected XSS
- Stored XSS
- DOM-based XSS
- Preventing XSS

Web Security/Privacy Issues

- Many sensitive tasks are done through web
  - Online banking, online shopping
  - Database access
  - System administration
- Web applications and web users are targets of many attacks
  - Cross Site Scripting (XSS)
  - SQL injection
  - Cross Site Request Forgery (CSRF)
  - Information leakage
  - Session hijacking

Web Browser and Network



- Browser sends requests
- Web site sends response pages, **which may include code**
- Interaction is subject to network attacks

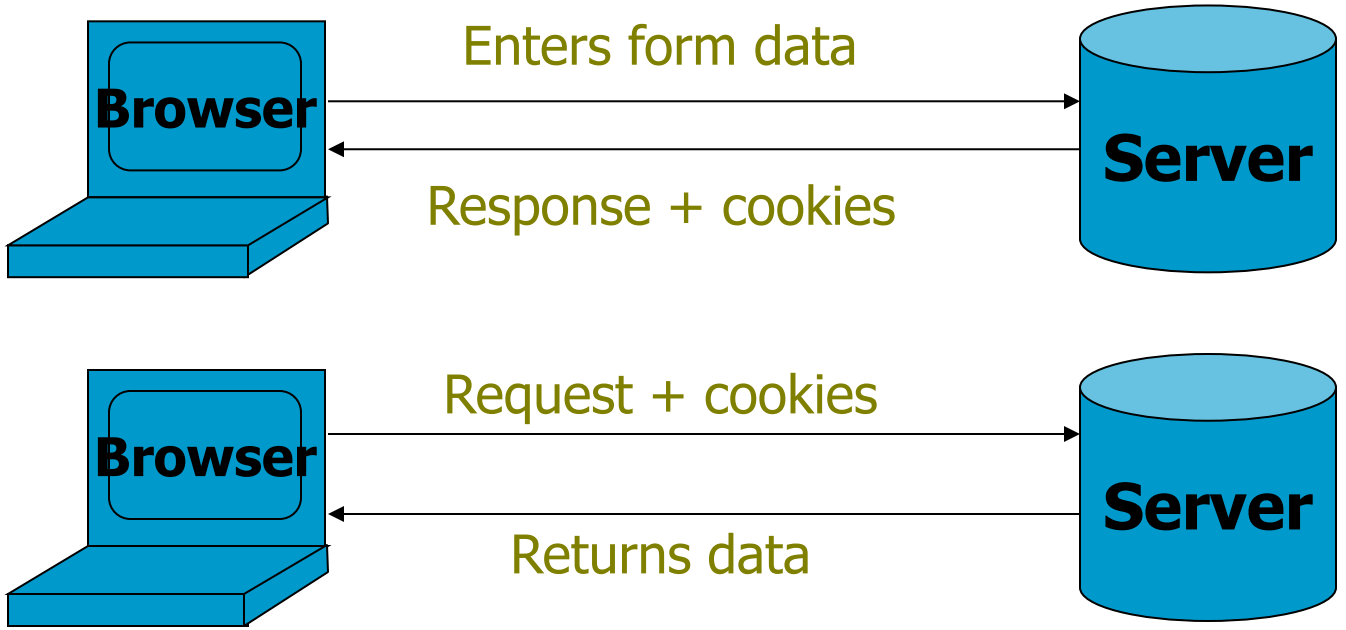


HTTP: HyperText Transfer Protocol

- Browser sends HTTP requests to the server
  - Methods: GET, POST, …
  - GET: to retrieve a resource (html, image, script, CSS,…)
  - POST: to submit a form (login, register, …)
- Server replies with an HTTP response
- **Stateless** request/response protocol
  - Each request is independent of previous requests

Use Cookies to Store State Info

- Cookies
  - A cookie is a name/value pair created by a website to store information on your computer



Http is stateless protocol; cookies add state

Web Authentication via Cookies

- Cookies are stored by the browser
- Used by the web applications
  - used for authenticating, tracking, and maintaining specific information about users
    - e.g., site preferences, contents of shopping carts
  - data may be sensitive
  - may be used to gather information about specific users

- Cookie ownership
  - Once a cookie is saved on your computer, only the website that created the cookie can read it

Cross Site Scripting

Client Side Scripting

- Web pages (HTML) can embed dynamic contents (code) that can be executed on the browser
- JavaScript
  - embedded in web pages and executed inside browser

```
<html>
      …
    <P>
<script>
      var num1, num2, sum
      num1 = prompt("Enter first number")
      num2 = prompt("Enter second number")
      sum = parseInt(num1) + parseInt(num2)
      alert("Sum = " + sum)
</script>
      …
</html>
```

Browser receives content, displays HTML and executes scripts

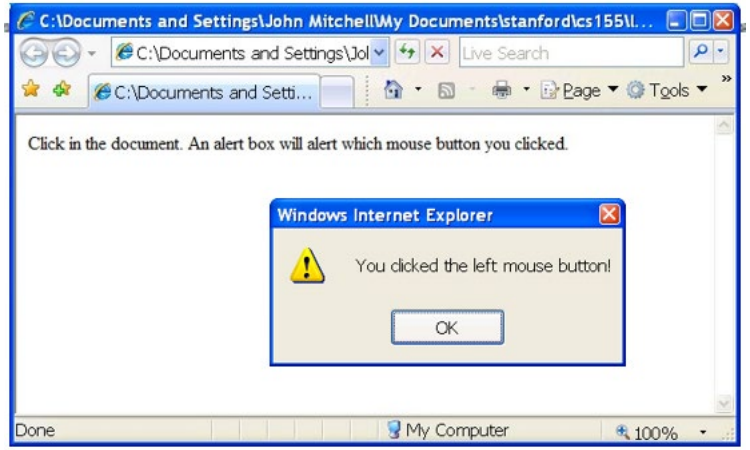
Client Side Scripting: Event-Driven Script Execution

```
<script type="text/javascript">
    function whichButton(event) {
    if (event.button==1) {
            alert("You clicked the left mouse button!") }
    else {
            alert("You clicked the right mouse button!")
    }}
</script>
...
<body onmousedown="whichButton(event)">
...
</body>
```

Script defines a page-specific function

Function gets executed when some event happens

Other events:
onLoad, onMouseMove, onKeyPress, onUnLoad



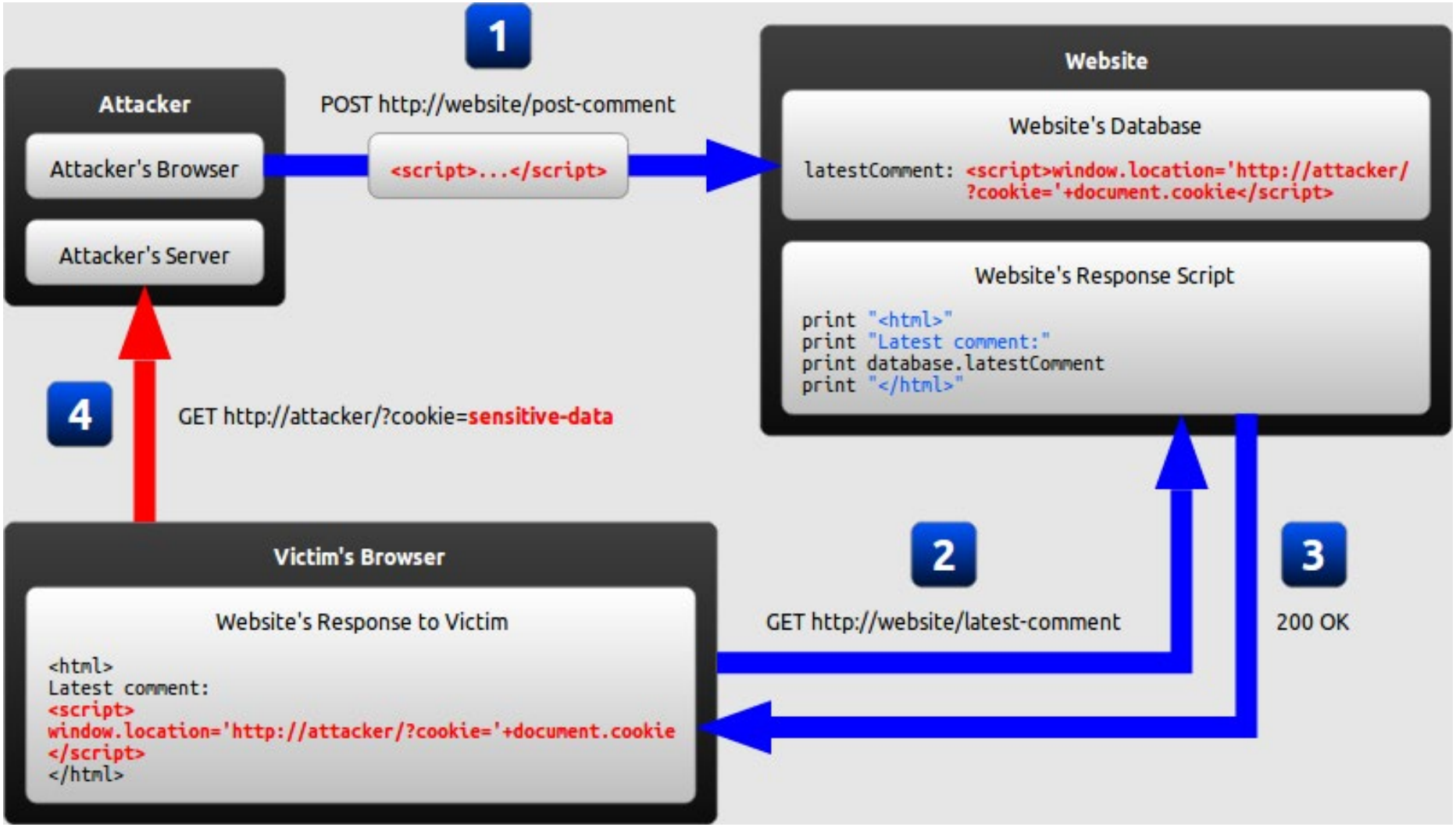
Cross-Site Scripting (XSS)

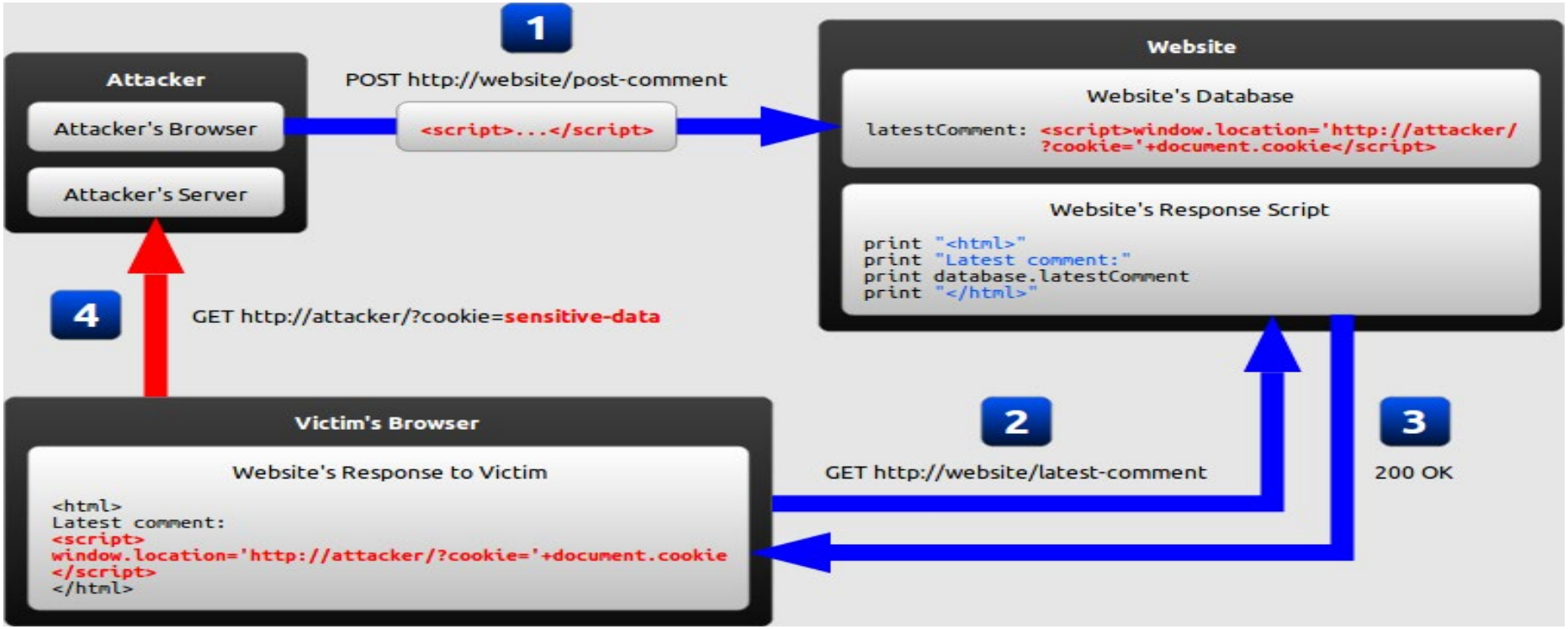
- ***Cross-Site Scripting (XSS)*** is a web vulnerability that lets a malicious hacker introduce (inject) undesired commands into legitimate **client-side code** (usually **JavaScript**) executed by a browser on behalf of the web application.

- The actual attack occurs when the victim visits the web page or web application that executes the malicious code.

- Vulnerable vehicles that are commonly used for Cross-Site Scripting attacks are forums, message boards, and web pages that allow comments.

XSS: Overview and Contexts





1. The attacker injects a payload into the website's database by submitting a vulnerable form with malicious JavaScript content.
2. The victim requests the web page from the web server.
3. The web server serves the victim's browser the page with attacker's payload as part of the HTML body.
4. The victim's browser executes the malicious script contained in the HTML body. In this case, it sends the victim's cookie to the attacker's server.
5. The attacker now simply needs to extract the victim's cookie when the HTTP request arrives at the server.
6. The attacker can now use the victim's stolen cookie for impersonation.

How Cross-Site Scripting (XSS) works?

There are two stages to a typical XSS attack:

1. To run malicious JavaScript code in a victim's browser, an attacker must first find a way to inject malicious code (payload) into a web page that the victim visits.
2. After that, the victim must visit the web page with the malicious code. If the attack is directed at particular victims, the attacker can use social engineering and/or phishing to send a malicious URL to the victim.

For step one to be possible, the vulnerable website needs to directly include user input in its pages. An attacker can then insert a malicious string that will be used within the web page and treated as source code by the victim's browser.

Cross-Site Scripting (XSS) Attack Tags

The most common tags that could be used to compromise the security of a website or web application through an XSS attack are:

<script> tag

The <script> tag is the most straightforward XSS payload. A script tag can reference either:

- external JavaScript code

```
<!-- External script -->
<script> src=http://evil.com/xss.js></script>
```

- or can embed the code within the script tag itself.

```
<!-- Embedded script -->
<script> alert("XSS"); </script>
```

Cross-Site Scripting Attack Tags

<body> events

JavaScript event attributes such as onload and onerror can be used in many different tags. This is a very popular XSS attack vector.

```
<!-- onload attribute in the <body> tag -->
<body onload=alert("XSS")>
```

event

An XSS payload can be delivered inside the <body> by using event attributes (see above) or other more obscure attributes such as the background attribute.

```
<!-- background attribute -->
<body background="javascript:alert("XSS")">
```

attribute

The background **attribute specifies a background image for a document**

Scripts are Powerful

- Client-side scripting is powerful and flexible, and can access the following resources
  - Local files on the client-side host
    - read / write local files
  - Webpage resources maintained by the browser
    - Cookies
    - Document Object Model (DOM) objects
      - steal private information
      - control what users see
      - impersonate the user
  - Communicating with websites

The Types of XSS Attacks

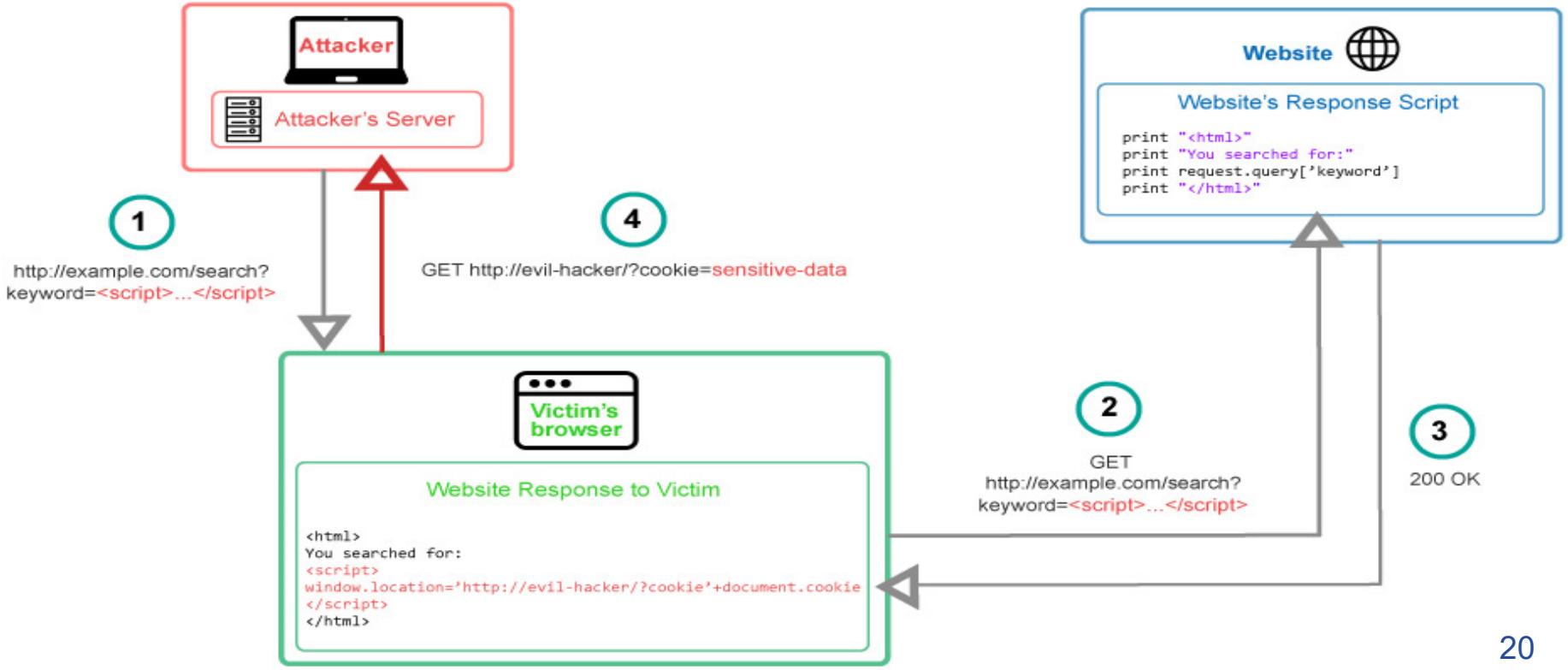
There are three main types of XSS attacks:

1. Reflected XSS, where the malicious script comes from the current HTTP request.
2. Stored XSS, where the malicious script comes from the website's database.
3. DOM-based XSS, where the vulnerability exists in client-side code rather than server-side code.

Reflected XSS

A reflected XSS attack occurs when the malicious JavaScript (string) is sent as a part of the victim's request to the website.

- Is not going to be saved permanently, but the malicious code will be reflected back to user.
- This type of Cross-site scripting (XSS) is commonly found in phishing attacks.



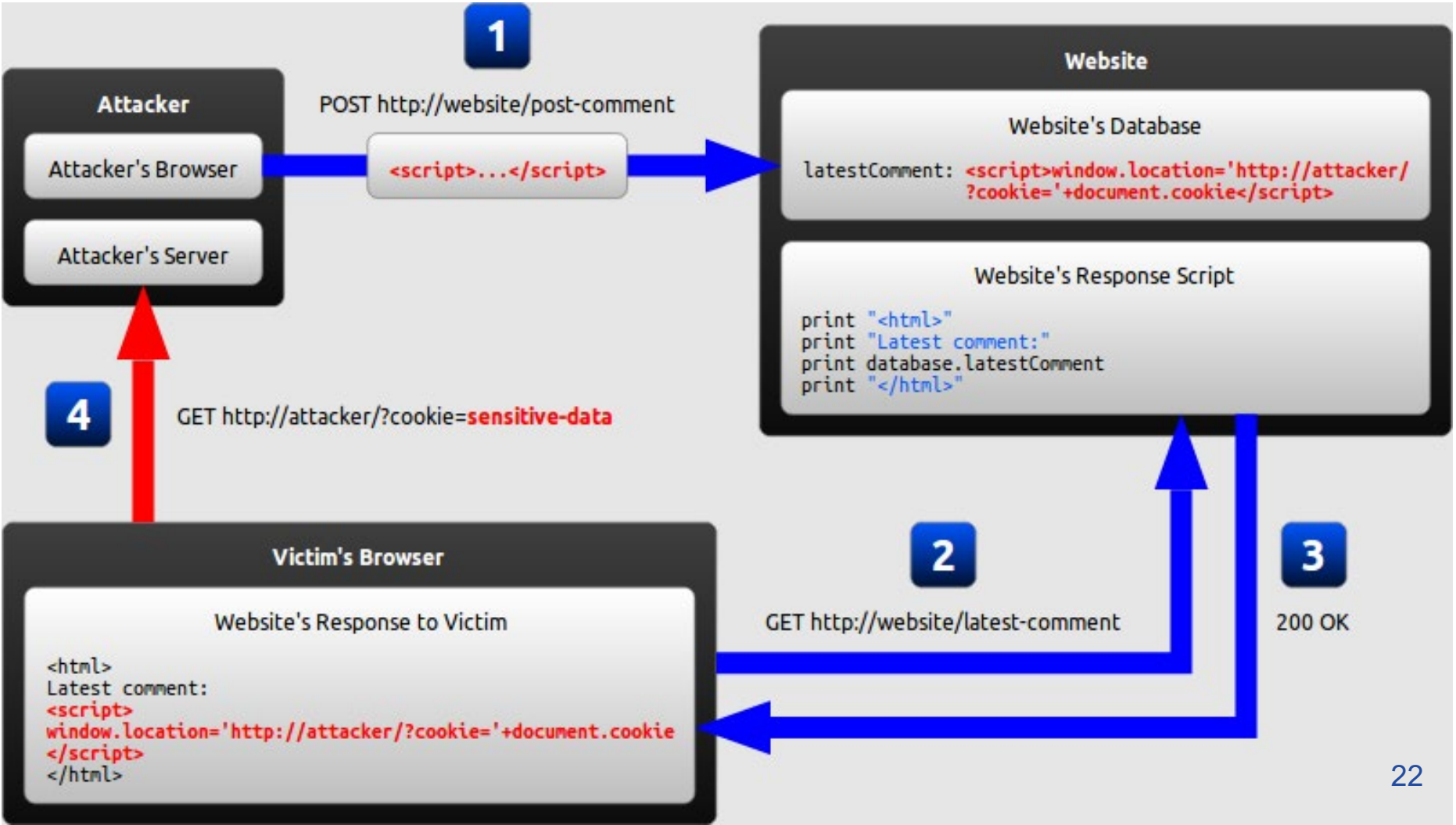
Impact of Reflected XSS Attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. Amongst other things, the attacker can:

- View any information that the user is able to view.
- Modify any information that the user is able to modify.
- Initiate interactions with other application users, including malicious attacks, that will appear to originate from the initial victim user.

Stored Cross-Site Scripting

Stored Cross-Site Scripting (a.k.a. persistent XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.



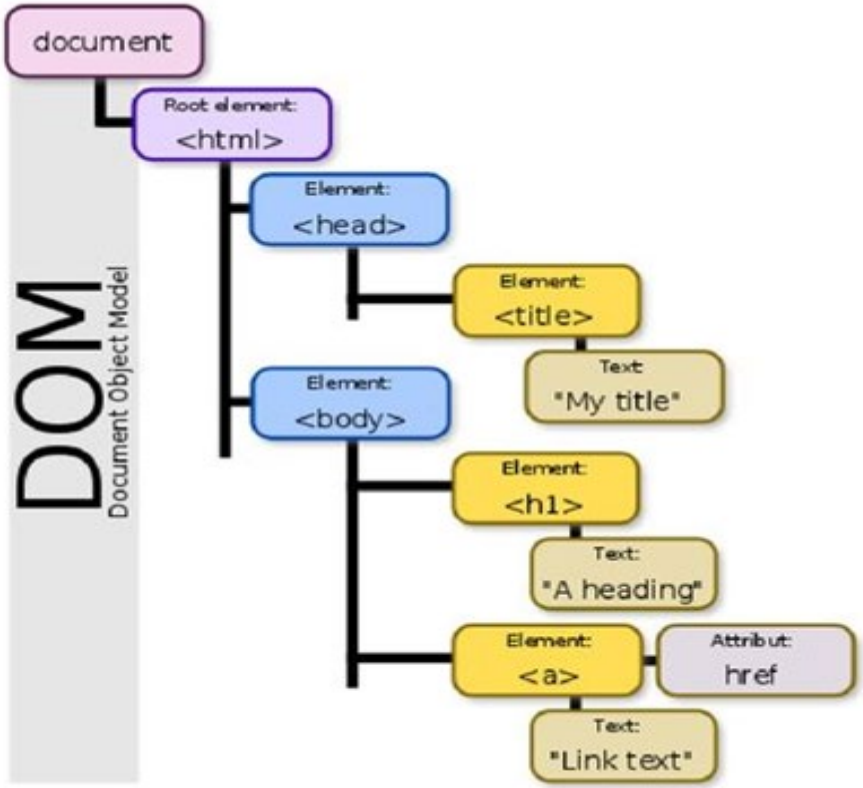
Key Difference Between Reflected XSS and Stored XSS

Aspect	**Reflected XSS**	**Stored XSS**
Persistence	Non-persistent; script only lives in the request/response cycle.	Persistent; script is stored on the server and affects multiple users.
Trigger	Requires a user to interact with a crafted link or request.	Automatically triggers when a user visits a page with stored malicious content.
Target	Affects only the user who interacts with the crafted payload.	Affects all users who visit the page with the stored script.
Attack Vector	Delivered via URLs, query strings, or form submissions.	Delivered via stored data (comments, user profiles, etc.).
Risk Level	Typically lower as it requires user interaction.	Typically higher as it can affect multiple users without user interaction.

DOM-based XSS

What is DOM

The Document Object Model is a convention for representing and working with objects in an HTML or XML document.



Browsers and DOM

- The DOM is the browser's internal representation of a webpage.
- Browsers allow to embed scripts in HTML code, using the tag:

<script type="text/javascript"> ... </script>

- The browser gives a script an access to the "DOM" (Document Object Model) for the current webpage, which allows the script to change any and every aspect of the page.

DOM-Based XSS Attack

DOM-based cross-site scripting is a type of cross-site scripting (XSS) where the attacker takes advantage of the Document Object Model (DOM).

Is a web application vulnerability that allows attackers to manipulate the DOM environment in a user's browser by injecting malicious client-side code, resulting in:

1. Steal the cookies of the user or
2. Change the page's behavior as the attacker like.

Example of a DOM-Based XSS Attack

Consider the URL:

http://www.example.com/userdashboard.html

The HTML source of the URL would look like this:

```
<html>
<head>
<title>Custom Dashboard </title>
...
</head>
Main Dashboard for
<script>
    var pos=document.URL.indexOf("context=")+8;
    document.write(document.URL.substring(pos,document.URL.length));
</script>
...
</html>
```

The inline script looks for `context=` in the URL (`document.URL.indexOf("context=")`), takes all the text to the right of it (`+8` means 8 characters to the right of the beginning of `context=`), and uses `document.write` to insert that text directly into HTML to be interpreted by the browser.

Example (Cont.)

If you call the following URL:

```
http://www.example.com/dashboard.html?context=Thomas
```

the page will say:

```
Dashboard for Thomas
```

If an attacker crafts a URL:
http://www.example.com/userdashboard.html?context=<script> …… </script>
The malicious scripts will be executed.

Sandbox

- Browsers must confine (sandbox) these scripts so that they cannot access arbitrary local resources

- A security mechanism for separating/limiting running programs
  - Running untrusted programs.
    - E.g., javascripts in webpages, mobile apps
  - Running programs that are likely to be exploited.

Same Origin Policy (SoP)

- Browser must have a security policy to **provide separation among mutually untrusted scripts**
- SoP isolates the scripts and resources downloaded from different origins
  - E.g., evil.org scripts cannot access bank.com resources
- Origin = domain name + protocol + port
  - all three must be equal for origin to be considered the same



Same Original Policy: What it Controls

- **Same-origin policy applies to the following accesses:**
  - manipulating browser windows
  - URLs requested via the XmlHttpRequest
  - manipulating frames
  - manipulating documents
  - manipulating cookies

XMLHttpRequest (XHR) is an API in the form of an object whose methods transfer data between a web browser and a web server.

Problems with SOP

- Poorly enforced on some browsers
  - Particularly older browsers
- Limitations if site hosts unrelated pages
  - Example: Web server often hosts sites for unrelated parties
    - http://www.example.com/account/
    - http://www.example.com/otheraccount/
  - Same-origin policy allows script on one page to access properties of document from another
- Can be bypassed in Cross-Site-Scripting attacks

Preventing Cross-Site Scripting

- An XSS attack is a type of code injection: **user input** is mistakenly interpreted as malicious program code.
- To prevent this type of code injection, **secure input handling** is needed.
- Two fundamentally different ways of perform secure input handling:
  - ✓ **Encoding**: escape user input so that the browser only interprets it as data, not code.
  - ✓ **Validation**: sanitize the user input so that the browser interprets it as code without malicious commands.

Preventing XSS - Encoding

- Involve transforming special characters in user input into their respective HTML entities. E.g., the less-than sign (**<**) becomes **<**, and the greater-than sign (**>**) becomes **>**.
- Example:

  User input: **<**script**>**alert('XSS')**<**/script**>**

  Encoded input:
  <script**>**alert('XSS')**<**/script**>**
- Popular encoding libraries:
  - ✓ **OWASP** (Open Web Application Security Project) **ESAPI** (Enterprise Security API)
  - ✓ Microsoft's **AntiXSS** and PHP's **htmlspecialchars()** function.

Preventing XSS - Validation

- The act of filtering user input so that all malicious parts of it are removed without necessarily removing all the code within it.
- One of the easiest types of validation is the one that allows certain HTML elements (e.g., ,) but not others (e.g., <script>).
- Classification strategies:
  - ✓ **Blacklisting**: specify what is not allowed
  - ✓ **Whitelisting**: specify what is allowed
- Validate outcome: user input identified as malicious can be either **rejected** or **sanitized.**

Functional Programming

What is functional programming?

Wikipedia: Functional programming is a **programming paradigm** that treats computation as the evaluation of mathematical functions.

Pure functional programming

- No side effects
  - output of a function depends **only** on its inputs
  - function does not change anything in the evaluation
  - can be evaluated in any order (many times, never)

- More complex function based on recursion
  - no for/while cycles
  - natural problem decomposition
    - mathematical induction

Brief History

- Lambda calculus (1930s)
  - formal theory of computation, older than TM (Turing Machine)
- Lisp = List Processing (1950s)
  - early practical programming language
  - second oldest higher-level language after Fortran
- ML = Meta language (1970s)
  - Lisp with types, used in compilers
- Haskell = first name of Curry (1990s)
  - standard for functional programming research
- Python, Scala, Java8, C++ 11, ….

A Brief List of Functional Languages

- LISP
  - Common LISP
  - Scheme
  - PLT Scheme
  - Racket
  - Clojure
- ML (Meta Language) is a general-purpose functional programming language.
  - Haskell
  - Scala
  - Caml
  - OCaml
- F#
- Erlang
- Miranda

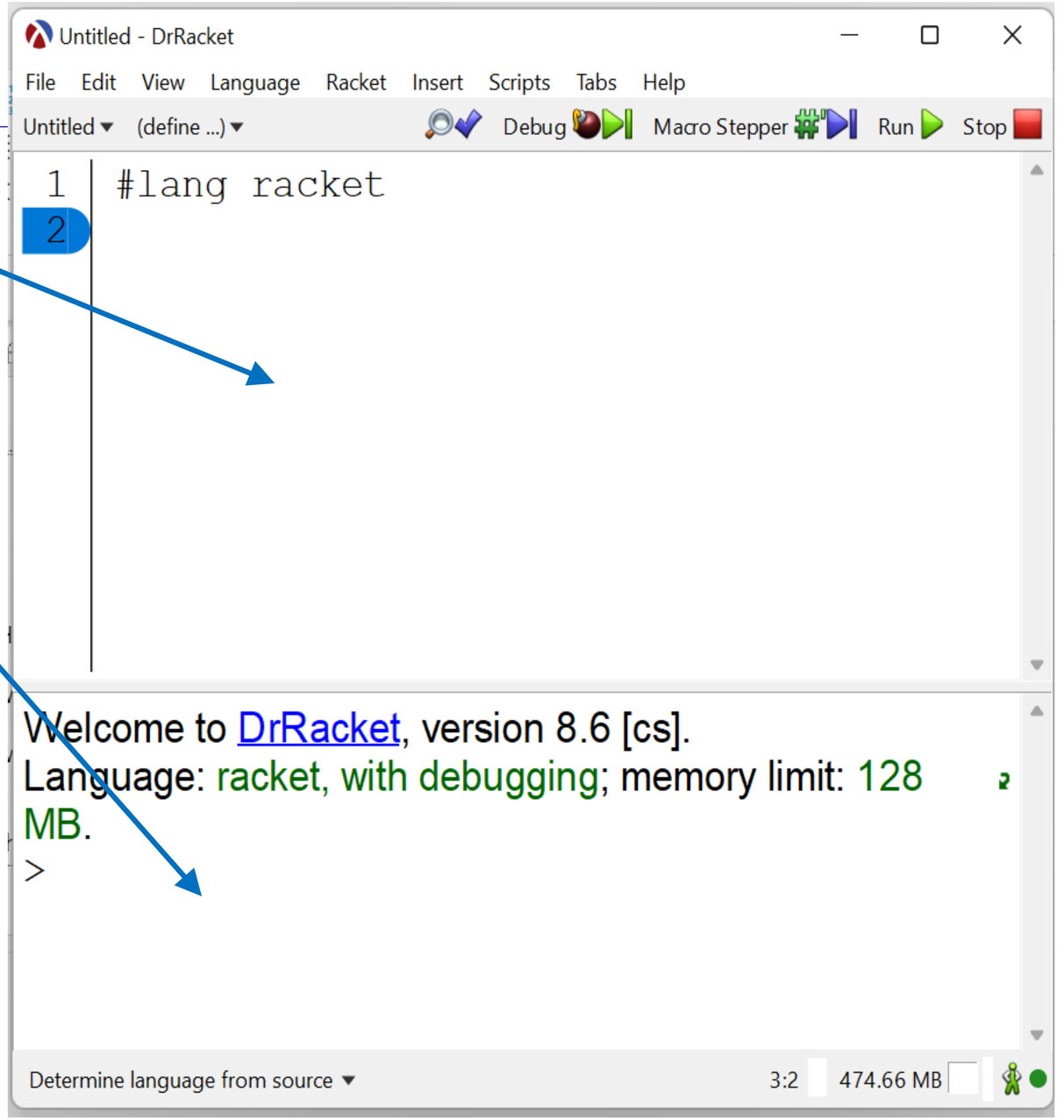
Racket

- Functional Programming Language
- http://racket-lang.org
- IDE: Dr. Racket
- Loading, Running, and Saving code
- Submitting source files (.rkt)

Download the most recent version from https://download.racket-lang.org/.
Note that it comes with a visual IDE called DrRacket that you should be able to use in assignments for this course.

Running Racket

- Once it's installed, you can run Racket by launching the DrRacket IDE. The IDE shows a text window (or definitions window) and an interaction window. The idea is that you write your program in the text window, and use the interaction window to test it.



Here are a few useful keyboard shortcuts:

- ctrl-E will open/close the interaction window
- ctrl-D will open/close the definitions window
- ctrl-S will save the current definitions
- ctrl-R will run the current definitions in the interaction window

Running Racket (cont.)

Be aware that Racket supports multiple languages, and in these notes we will always be using the base Racket language unless we say otherwise. To ensure you are using the correct language, make sure that all your Racket programs have this at the top:

```
#lang racket
```

Using Racket's Interactive Interpreter

- We'll mainly be using Racket via its interactive interpreter. This is sometimes referred to as a **REPL**, which stands for **read-eval-print loop**.

  It lets us evaluate expressions one at a time. For instance:

```
> (+ 3 2)
5

> (* 10 4)
40

> (- 5 8)
-3

> (/ 6 2)
3

> (/ 5 2)
2 1/2
```

Polish Notation (a.k.a. Prefix Notation)

- The Polish logician Jan Łukasiewicz invented this notation in 1924 in order to simplify sentential logic. Polish notation
  - 3 + 2
  - `(+ 3 2)`
  - Evaluates to “5”
- Precedence by parentheses
  - 2 + 3x
  - `(+ 2 (* 3 x))`
- Don’t forget spaces!
  - `(+3 2)  ; wrong!`
  - `(+ 3 2) ; correct!`

Prefix Notation

- All functions in Racket are called using **prefix** notation.
- One advantage of prefix notation is that you can pass multiple arguments to many operations, e.g.,

```
> (+ 3 2 3 5)
13

> (* 1 6 2 2)
24

> (/ 100 10 5)
2

> (- 1 2 3)
-4
```

Racket Built-In Datatypes

- **Booleans**: Represented by ‘#t’ for true and ‘#f’ for false.
- **Numbers**: Including integers, decimals, fractions, and complex numbers.
- **Characters**: Single characters, written as ‘#\a’, ‘#\b’, etc.
- **Strings**: Unicode strings, written within double quotes, like "Hello, World!".
- **Bytes and Byte Strings**: Represented by ‘#u8(0 1 2 3)’ for a byte string containing bytes ‘0’, ‘1’, ‘2’, and ‘3’.

Booleans

- boolean values: #t for true and #f for false.
  Uppercase #T and #F are parsed as the same values
- the result of a test expression for if, cond, and, or, etc.,
- however, any value other than #f counts as true.

```
> (= 2 (+ 1 1))
#t
> (boolean? #t)
#t
> (boolean? #f)
#t
> (boolean? "no")
#f
> (if "no" 1 0)
1
```

Numbers

- Integer (e.g., 1, 4, -3, 0)
- float (e.g., 0.0, 3.5, 1.23E+10)
- rational (e.g. 2/3, 5/2)

```
> (integer? 1)
#t
> (real? 1)
#t
> (exact-integer? 6.0)
#f
```

```
> (integer? 5.0)
#t
> (real? 6.4)
#t
> (rational? 5.0)
#t
```

```
> (complex? 5.0)
#t
> (complex? 1+2i)
#t
> (integer? 1+2i)
#f
> (rational? 3/4)
#t
```

```
> (sin -5+2i)
3.607660774213156 3+1.0288031496599335i
```

Basic Functions

- Arithmetic functions (+, -, *, /, abs, sqrt)
- Relational (=, <, >, <=, >=) (for numbers) e.g., (<= 3 3.1) → #t
- Relational (eq?, eqv?, equal?) for arbitrary data (more about these later)
  - ✓ **eq?** for checking if two values are **the same object** in memory

```
(define a '(1 2 3))        ; Define a list
(define b '(1 2 3))        ; Define another list with the same elements
(define c a)               ; Define c to be the same as a
```

(eq? a b) → #f **(eq? a c)** → #t

  - ✓ **eqv?** for checking if two values are **equivalent in value and type**.

(eqv? 5 5) → #t **(eqv? 'xy 'xy)** → #t

  - ✓ **equal?** for checking if two values are **structurally equivalent**, especially for complex data structures.

(equal? a b) → #t **(equal? a c)** → #t

- Logical (and, or, not): and or are short circuit logical functions.

Logic Functions

```
> (and)
#t
> (and (= 2 3))
#f
> (and (= 2 2) (< 4 5))
#t
> (and (= 2 2) (< 4 5) (> 4 10))
#f
```

```
> (or)
#f
> (or (= 2 3))
#f
> (or (= 2 3) (< 4 5))
#t
> (or (= 2 3) (> 4 5) (> 6 10))
#f
```

```
> (or (= 2 2) (error "oops"))
#t
```

Binding: Racket `define` keyword

`(define <identifier> <expression>)`

- Assigning a name (alias) to a value, function, or construct—*not assigning a value to a variable*
  - `(define x 5)`
  - `(define y 3)`
  - `(+ x y)` `; evaluates to 8`

Bindings

(**define** x e)

define is a keyword, x can be any *identifier* and e can be any *expression*.

```
>(define x 5)
> X
5
>(define scale 4.5)
>(define title "Dr. Racket")
>(* scale 5)
22.5

>(string-append title "!!!")
"Dr. Racket!!!"
```

Bindings

It is possible to override almost *any* name

```
>(define * +)
>(*   4   5)
9


>(+   4   5)
9
```

Bindings and Functions

```
(define x 2)
(define y 3)
(if (< x y) y x)
3
(- (if (< x y) y x) (if (> x y) y x))
1
```

Function definitions usually have a different form:

```
>(define (add1 n) (+ 1 n))

>(add1 5)
6
```

```
(define (abs-diff x y)
  (if (< x y)
      (- y x)
      (- x y)))

> (abs-diff x y)
1
```

Example: Define a function

```
(define (greet name)
(printf "Welcome to Racket ~a!" name)
(newline)
(printf "I hope you learn a lot.") )
```

It is called like this:

```
(greet "Alan")
Welcome to Racket Alan!
I hope you learn a lot.
```

In general, a `cond` form looks like this:

```
(cond [test1 result1]
      [test2 result2]
      ...
      [else result_else]
)
```

Either use []- in cond
Or ( ) - in cond

```
(define (sign n)
  (cond [(not (number? n)) "not a number"]
        [(< n 0) "negative"]
        [(> n 0) "positive"]
        [else "zero"]
))

> (sign -5)
"negative"
> (sign 3)
"positive"
> (sign 0)
"zero"
> (sign "three")
"not a number"
```

Input

Input Function:
Read-line: read strings

```
#lang racket
(display "Please enter your name: ")
(define name (read-line))
(display "Hello, ")
(display name)
(newline)
```

When we run this program,

```
Please enter your name: Sam
Hello, Sam
```

Input Function:
Read: read an expression

```
#lang racket
(display "Please enter a number: ")
(define val (read))
(display val)
(display " cubed is ")
(display (* val val val))
(newline)
```

When we run this program,

```
Please enter a number: 5
5 cubed is 125
```

Racket Comments

- Single line comment

```
; this is a single line Racket comment
```

- Multi-line comments

```
#|
   This is a multi-line
   Racket comment
  |#
```

- Single expression comment (ignores the entire expression)

```
#; (define succ
       (lambda (n)
         (+ n 1)))
```

Exercises

- Write a Racket function to compute the triple of a value,

```
> (triple 4)
12
```

(define (triple X) (* 3 X))

- Write a Racket function to compute a Fibonacci number:

```
1                          if n = 0 or n =1
fib(n -1) + fib(n -2)      if n > 1
> (fib 5)
8
```

(define (fib N)
(if (or (zero? N) (= N 1))
1
(+ (fib (- N 1)) (fib (- N 2)))))

- Write a Racket function to check if a number is even

```
> (even 8)
#t
```

(define (even num) (= (remainder num 2) 0))

Racket Syntax

- Almost everything in Racket is an expression
- **All expressions are represented as lists**
- Lists are enclosed in parentheses and are separated by spaces, not commas
  - `'(1 2 3 4)`
- Lists may be nested—in fact, every Racket program is a single large nested list
  - `'(1 (2 3) 4)`

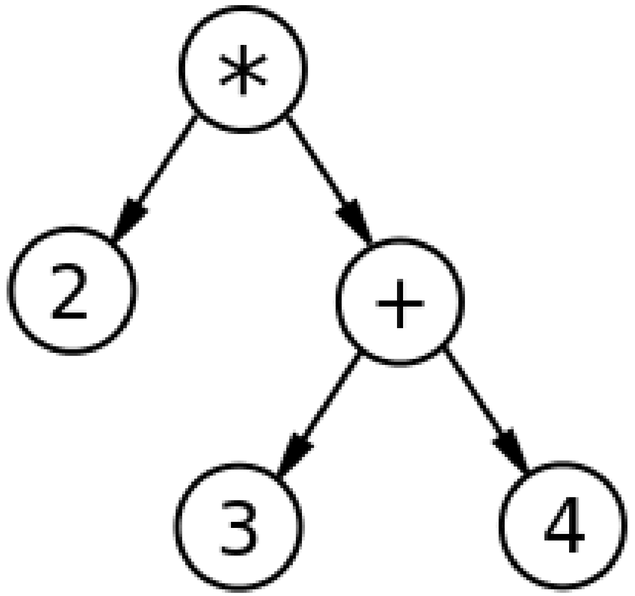
Racket Syntax (cont.)

- Some examples
  - `(sqr 3)`
  - `(+ 5 3)`
  - `(list 1 2 3)`
  - `'(1 2 3)`
- The first element of every list is assumed by the compiler to be a function
- Lists may or may not be “evaluated”
- The leading tick mark suppresses the default automatic evaluation of an expression

S-Expression

- S-Expressions (for "**S**ymbolic **E**xpression") are a notation for nested list (tree-structured) data, invented for and popularized by the programming language Lisp, which uses them for source code as well as data.

```
(*  2 (+  3  4) )
```



S-Expression

- In the usual parenthesized syntax of Lisp, an S-expression is classically defined recursively as

1. A primitive value, such as 12, true, or "tacos“

The leaves of an s-expression are atomic (indivisible) and are called atoms.

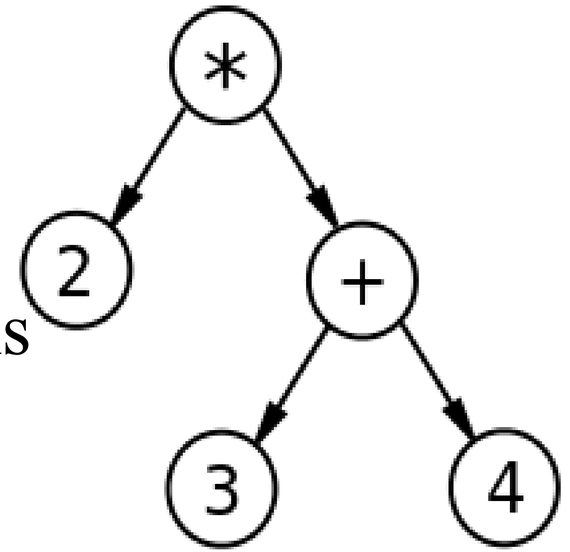
OR

2. A parenthesized list containing zero or more s-expressions separated by whitespace: (s-expression*)

( )

(1 2)

(1 (3 (4) 5) 6)



(* 2 (+ 3 4))

S-Expression

- Operation ⇒ Value
  - `(+ 4 3)`
  - `(/ 15 3)`
  - `(expt 4 2)`
- Evaluation ⇒ Boolean
  - `(= 3 3)`
  - `(eq? 4 4)`
  - Boolean literal (`#t` and `#f`)

Naming Conventions in Racket

- Conventions for user defined names are less strict than traditional imperative languages, such as those in the C-family
- The lexical parser is able to recognize names that contain characters other than just alpha-numeric and underscores.

```
> (define &R@ *)
> (define %$# 7)
> (define $#% 3)
> (&R@ %$# $#%)
21
```

Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*

- A *lambda expression (function)* specifies
  - the parameter(s) and
  - the mapping (body) of a function in the following form

cube: `λ(x) .x * x * x`

32

Lambda Calculus

- Based on mathematical recursive function theory
- **Lambda (λ) calculus**

Lambda calculus (also written as **λ-calculus or called "the lambda calculus"**) is a formal system in mathematical logic and computer science for expressing computation based on functions by way of variable binding and substitution.

- Anonymous functions
- Immutability
- First-order Functions – Functions are “first class citizens”

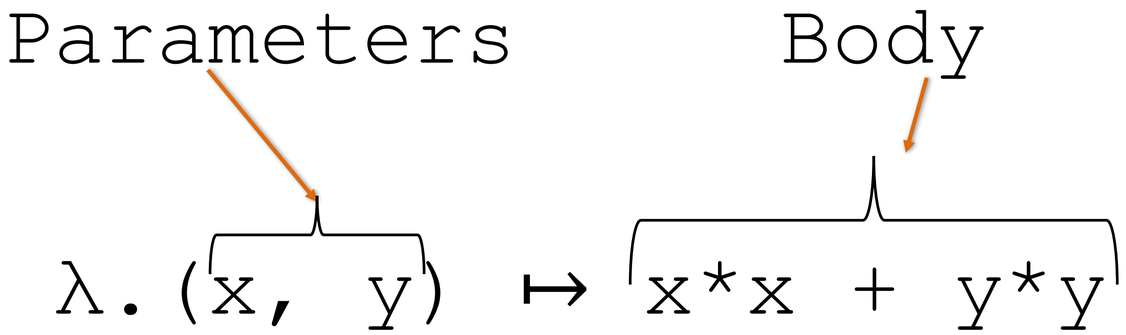
  First-class functions: functions are treated like other variables. The functions can be assigned to any other variable or passed as an argument or can be returned by another function.

Anonymous Form

- The identity function

```
λ.(x) ↦ x
```

- "Square Sum" (sqsum) Function



Lambda Function

- Racket Syntax

  `(λ (args) (body))`

- Racket Examples
  - `(λ (x) (x))  ; identity function`
  - `(λ (x) (* x x)) ; square`

- Here in Racket with a passed value
  - `((λ (x) (* x x)) 3)`

Lambda Function (cont.)

square(x) function

- ((λ (x) (* x x)) 3)

- ```
  (define square (lambda (x)
                   (* x x)
                 )
  )
  (square 3)
  ```

- ```
  (define (square x)  ; no lambda
            (* x x)
  )
  (square 3)
  ```

Lambda Function (cont.)

Form	Named?	Anonymous?	Preferred When...
(λ (x) (* x x))	❌	✅	One-time use in expressions, map, etc.
(define square (λ (x) (* x x)))	✅	✅	Function needs to be treated as a value
(define (square x) (* x x))	✅	❌	General use — most readable and idiomatic

Lambda Function (cont.)

- `Lambda` functions can have any number of parameters

```
 >(lambda (x) (* x x))
#<procedure>

>(lambda (a b x) (+ (* a x x) (* b x)))
#<procedure>
```

- Lambda functions can be applied to parameters

```
>((lambda (x) (* x x)) 7)
 49
>((lambda (a b x) (+ (* a x x) (* b x))) 2 4 6))
96
```

- Bind the procedure object with identifier **square**:

```
(define square (lambda (x) (* x x)))

(square 5) ; This will square 5 and return 25
```

square holds the procedure object of the lambda function

Higher-order functions

- A higher-order function, is one that either takes functions as parameters or yields a function as its result, or both
- Functions are treated as **first-class citizens**

Functions as Parameters

```
def sort(numbers, compare_function):
    # sorting logic using compare_function
    pass
```

Functions as Return Values:

```
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply
```

```
double = multiplier(2)  # double is a function that multiplies its input by 2
```

(define multiplier (λ (n) (λ (x) (* n x))))
(define double (multiplier 2))
(define 5-times (multiplier 5))

- **(double 20)**
- **40**
- **(5-times 20)**
- **100**

Function Composition

- Function composition involves combining two or more functions to produce a new function. In mathematical notation, if you have two functions $f(x)$ and $g(x)$, their composition, denoted as $(f \circ g)(x)$, is defined as $f(g(x))$.

  $h \equiv f \circ g \rightarrow h(x) \equiv f(g(x))$

  e.g., $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x \rightarrow h \equiv f \circ g$ yields $(3 * x) + 2$

- Associativity: Function composition is associative, which means that the order of composition doesn't matter, $(f \circ g) \circ h$ is equivalent to $f \circ (g \circ h)$.
- Identity Function: The identity function $I(x) = x$ serves as the identity element for function composition. When any function f is composed with the identity function, $f \circ I = I \circ f = f$.

41

Apply-to-all

- A higher-order function applies a given function to all elements of a collection, such as a list, array, or set. The result is a new collection containing the transformed elements.
- **map** function applies a given function to each element of a list and returns a new list with the transformed values

```
> (define numbers '(1 2 3 4))
> (define squared-numbers (map (lambda (x) (* x x)) numbers))
> squared-numbers
'(1 4 9 16)
```


42

Apply-to-all (cont.)

- **filter** function applies a given predicate function to each element of a list and returns a new list containing only the elements for which the predicate is true

```
> (define numbers '(1 2 3 4))
> (define even-numbers (filter even? numbers))
> even-numbers
'(2 4)
```

- **foldl (reduce)** function applies a given binary function to the elements of a list in a cumulative way, reducing the list to a single value.

```
> (define numbers '(1 2 3 4))
> (define product (foldl * 1 numbers))
> product
24
```


43

Lists and List Processing

Lists

- Any length

```
> (list 2 3)
'(2 3)
> (list 2 3 4 5 8)
'(2 3 4 5 8)
```

- Heterogeneity

```
>(list 3.14 "hello" 42)
'(3.14 “hello” 42)
```

- Can be created directly using this shorthand

```
'(2 3)
'("one" "two" 3.14)
```

Nested Lists

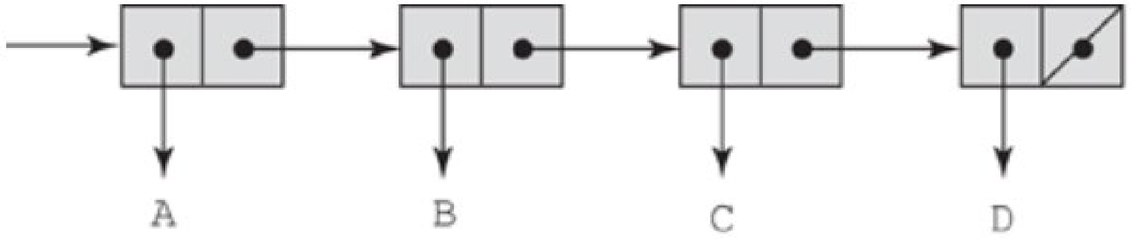
- Direct nesting

```
>(list 4 6 (list 8 3) 9)
   '(4 6 (8 3) 9)
>(list 1 2 3 (list (list 'a 'bd 'c) 4 5 6))
   '(1 2 3 ((a bd c) 4 5 6)) ; quote is applied to each element
>(flatten '(1 (2 3 (4 5 (6) 7) 8 9))) ;
'(1 2 3 4 5 6 7 8 9)
```

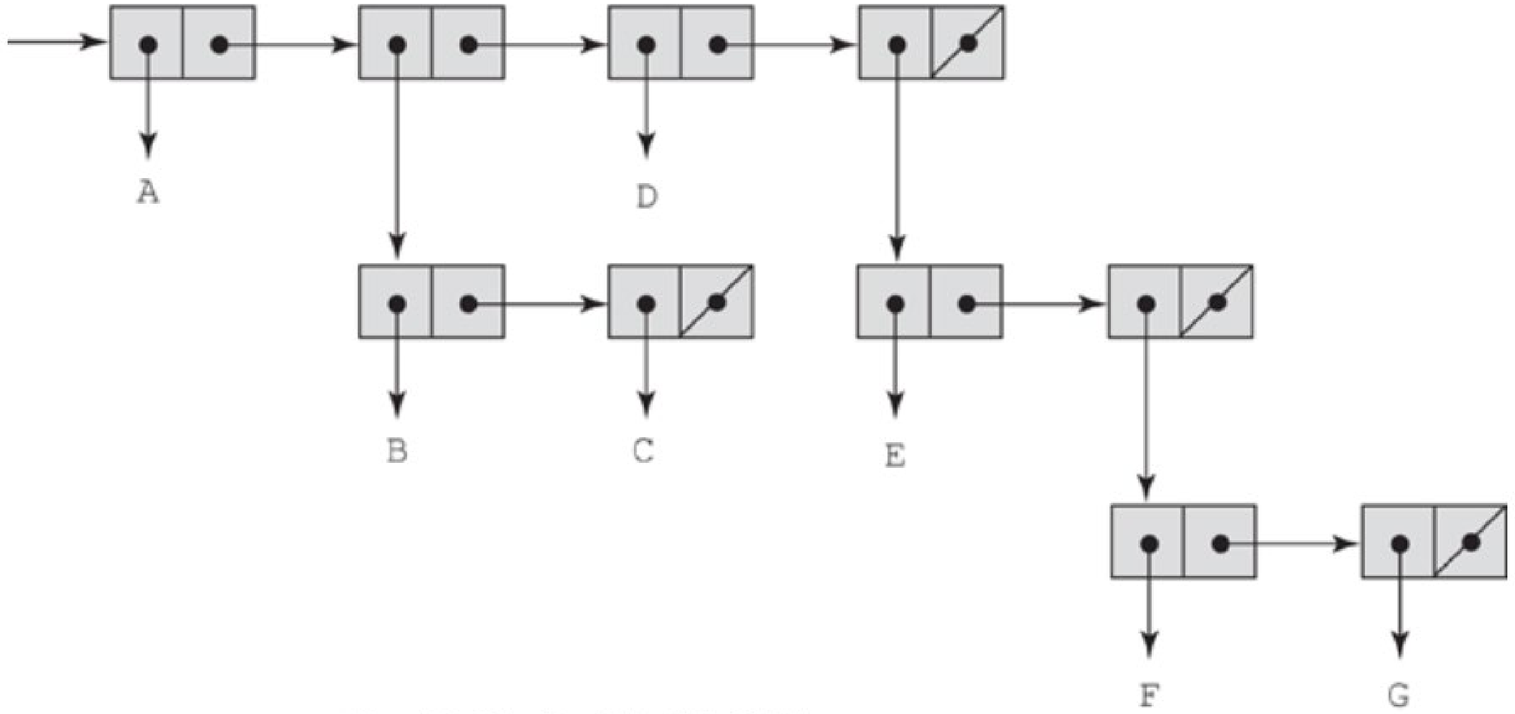
- Indirect nesting

```
>(define x (list 5 7 3))
> (list 9 8 x 1)
 '(9 8 (5 7 3) 1)
```

Internal Representations of Lists



(A B C D)



(A (B C) D (E (F G)))

`cons` Operator / Keyword

The **cons** function takes an element and a list and “adds” the element to the front of the list, creating a new list with all of the elements:

```
> (cons 1 '(2 3))
'(1 2 3)
> (cons "apple" '("banana"))
'("apple" "banana")
> (define lst '(2 3 4))
> (define new-list (cons 1 lst))
> new-list
‘(1 2 3 4)
```

cons Keyword

- Lists are actually recursive binary constructions (cons)
- Terminated with a **`null`**

```
>(cons 1 '())
  '(1)
>(list 1 2 3)
'(1 2 3)
>(cons 1 (cons 2 (cons 3 '())))
'(1 2 3)
```

Cons and Empty List

- Synonyms

```
'()
empty
null
```

- In practice, these are all equivalent

```
>(cons 4 '())
'(4)
>(cons 4 empty)
'(4)
>(cons 4 null)
'(4)
```

Examples

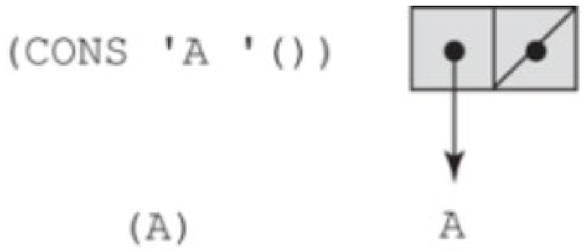
```
> (cons 'a ' (b c))
‘(a b c)

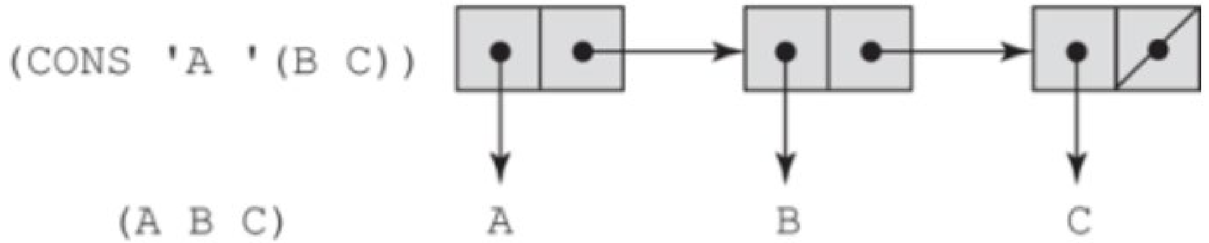
> (cons '(a b) (cons 'a '()))
‘((a b) a)

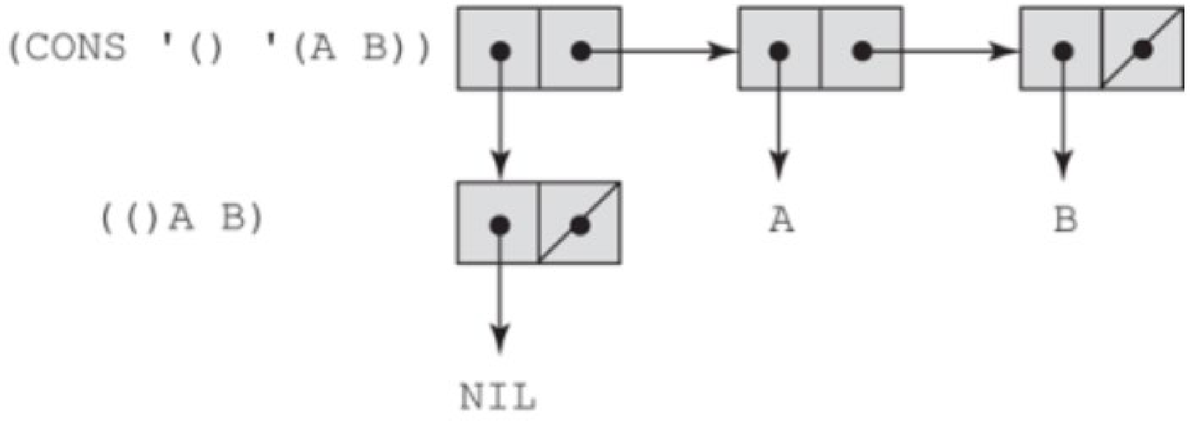
> (list '(a b) 'a)
'((a b) a)

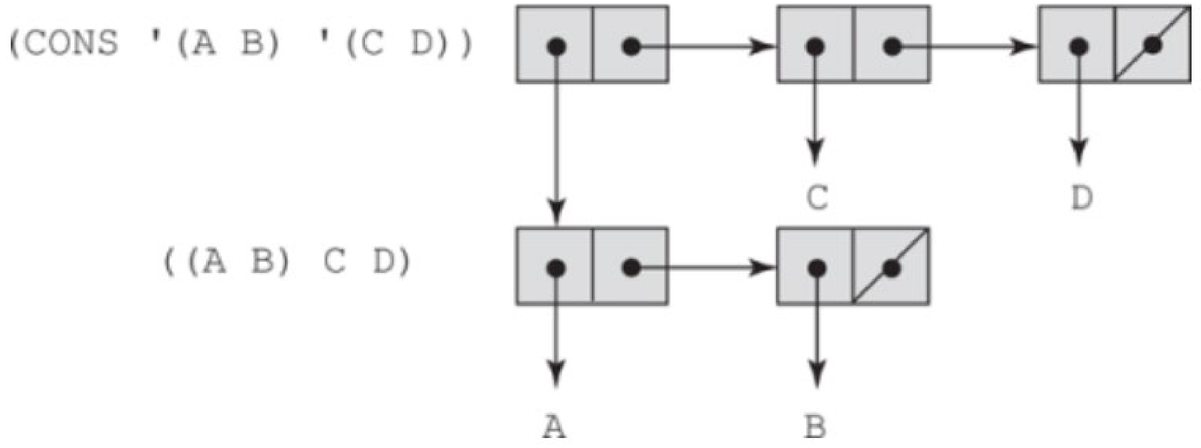
> (cons '(a b) ‘(c d))
‘((a b) c d)
```

Internal Representation of cons Operations









`cons` Operator / Keyword

- What about applying **cons** to two atoms? It returns a dotted pair (**cons cell**),

  > (cons 1 2)

  '(1 . 2)

- It is not a list ‘(1 2), instead the result is a dotted pair → constructs a memory object holding two values or pointers to values.

  > (equal? (list 1 2) (cons 1 2))

  #f

  why? because (list 1 2) == (cons 1 (cons (2 ‘())) == ‘(1 2), but (cons 1 2) == ‘(1 . 2)

`car` and `cdr` Operators / Keywords

- **car** and **cdr** take a list apart, in a sense, the inverse of **cons**
- **car**: retrieves the first element of a list or pair. E.g.,

```
> (car '(1 2 3))
1
```

```
> (car (cons 42 "hello"))
42
```

- **cdr**: retrieves the rest of the elements of a list or pair after the first element.

```
> (cdr '(1 2 3))
'(2 3)
```

```
> (cdr (cons 42 "hello"))
“hello”
```

The origin of names **car** and **cdr** lies in the first implementation of Lisp, which was on an IBM 704 computer.

append Keyword

- **append** function is used to concatenate lists together. It takes one or more lists as arguments and returns a new list containing all the elements of the input lists concatenated together.

```
> (append '(a b c) '(1 2 3 4))
 ‘(a b c 1 2 3 4)

> (append '(1 2 3) '(4 5 6) '(7 8 9))
 '(1 2 3 4 5 6 7 8 9)
```

Additional List Functions

- What are these aliases for?

```
>(first '(1 2 3 4))
1
>(last '(1 2 3 4))
4
>(rest '(1 2 3 4))
'(2 3 4)
```

- Combinations

```
>(cadr '(1 2 3 4))
2
   (car (cdr '(1 2 3 4)))
>(cddr '(4 7 5 6))
'(5 6)
   (cdr (cdr '(4 7 5 6)))
>(cdar '(4 7 5 6))
error, why?
   (cdr (car '(4 7 5 6)))
```

Exercise

```
>(sum-list '(1 2 3 4 5))
15
```

```
(define (sum-list lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst) (sum-list (rest lst)))]
  )
)
```

Exercise (cont.)

Write a Racket function called censor-word, which takes a symbol and list. It returns a list in which all occurrences of the symbol have been replaced by YYYYY.

Example:

> (censor-word 'FORTRAN '(I AM A FORTRAN PROGRAMMER))

(I AM A YYYYY PROGRAMMER)

Solution

```
(define (censor-word S L)
    (if (empty? L)
        '()
        (if (equal? (car L) S)
            (append ' (YYYYY) (censor-word S (cdr L)))
            (append (list (car L)) (censor-word S (cdr L)))
        )
    )
)
```

Additional List Functions

```
>(reverse '(4 9 8 5 12 6 7))
'(7 6 12 5 8 9 4)


>(length '(4 9 8))
3


>(list-ref '(4 9 8) 0)
4
```

Additional List Functions

```
>(take '(4 9 8 5 12 6 7) 2)
'(4 9)

>(take-right '(4 9 8 5 12 6 7) 3)
'(12 6 7)

>(drop '(1 2 3 4 5 6 7) 2)
'(3 4 5 6 7)

>(drop-right '(1 2 3 4 5 6 7) 3)
'(1 2 3 4)
```

map operation

```
> (map abs (list -1 2 -3 4)) ;
'(1 2 3 4)

> (map + '(1 2 3 4) '(10 20 30 40)) ;
 '(11 22 33 44)

> (define (square x)
            (expt x 2)
 )
> (square 3)
 9
> (map square '(1 2 3 4 5))
'(1 4 9 16 25)
```

String Comparison

- `(string=? "hi" "hi")`
- `(string>? "hi" "hit")`
- `(string<? "hi" "hit")`
- `(string>=? "hi" "hit")`
- `(string<=? "hi" "hit")`
- Case Insensitivity
  - `(string-ci=? "HELLO" "hello")`
  - `(string-ci<? "HI" "hi")`

String Functions

```
> (string-upcase “hello")
“HELLO"
> (string-downcase "aBC!")
"abc!"
> (string-length "Apple")
5
> (string-append "Apple" "Banana")
"AppleBanana"
> (substring "Apple" 1 3)
"pp"
> (substring "Apple" 1)
"pple"
> (string-replace "foo bar baz" "bar" "blah")
"foo blah baz"
> (string-ref "Apple" 0)
#\A
```

A printable character normally prints as #\ followed by the represented character.

64

String Functions

```
> (string-prefix? "Racket" "R")
#t
> (string-prefix? "Racket" "Rac")
#t
> (string-prefix? "Jacket" "R")
#f
> (string-suffix? "Racket" "et")
#t
> (string-contains? "Racket" "ack")
#t
```

Exercise

Write a Racket function that takes a string and a list of strings as arguments Then return true if the string exists in the list, otherwise return false.

```
(define (contains-str? str L)
  (cond
    [(empty? L) #f]
    [else
     (or
       (string=? (first L)  str)
       (contains-str?  str  (rest L)))
    ]
  )
)
```

```
> (contains-str? "Hi" '("Hello" "Hi" "No"))
#t
> (contains-str? "YES" '("Hello" "Hi" "No"))
#f
```

Exercise (cont.)

Write a Racket function that takes a string and a list of strings as arguments, then return the frequency of this string in the given list of strings. E.g.,

```
> (freq "hi"   '("Hello" "hi" "Hi" "No" "no" "hi" ))
2
```

```
(define (freq str L)
      (cond ((empty? L) 0)
            ((equal? str (car L)) (+ 1 (freq str (cdr L))))
            (else (freq str (cdr L)))
      )
)
```

More Racket:
Control Structures

Iterations in Racket

A **For** loop iterates through the sequence produced by the sequence-expression. For each element of the sequence:

- **For** loop binds the element to id,
- and then it evaluates the body.

```
(for ( [id   sequence-expression]  ...  )
        body ...)
```

Examples:

```
> (for ([i '(1 2 3)])
    (display i))
123
```

Iterating Over a Range of Numbers

Syntax:

(for ([identifier range-expression]) body)

- The **in-range** function generates a sequence of numbers:
  - given an **optional** starting number (which defaults to 0),
  - a number before which the sequence ends,
  - and an **optional** step (which defaults to 1).

```
> (for ([i (in-range 3)])
       (display i) (display " "))
0  1  2
> (for ([i (in-range 1 4)])
       (display i))
123
> (for ([i (in-range 1 4 2)])
       (display i))
13
> (for ([i (in-range 4 1 -1)])
       (display i))
432
> (for ([i (in-range 1 4 1/2)])
     (printf " ~a " i))
 1  3/2  2  5/2  3  7/2
```

Iterating Over a List

- The **in-list** function iterate over elements of a list:

```
> (for ([i (in-list ’(1 2 3 4 5))]) (display i) (display “ “))
1 2 3 4 5
```

- Iterating over nested lists using multiple in-list functions:

```
> (for ([row (in-list '((1 2) (3 4) (5 6)))])
    (for ([element (in-list row)])
      (display element) (display " "))
    (newline))
1 2
3 4
5 6
```

Iterating Over a String

- The **in-string** function iterate over characters of a string:

```
> (for ([char (in-string "Hello World!")])
    (display char) (display " "))
H e l l o  W o r l d !
```

Note:

For loop in Racket can work without explicitly specifying the data source (in-list or in-string) if you're iterating over a list or string.

```
> (for ([i '(1 2 3 4 5)]) (display i) (display " "))
1 2 3 4 5
```

for/list

- The **for/list** variant combines a For Loop with the List comprehension to make a choice for list generation in Racket.
- It accumulates *body* results into a list

```
> (for/list ([i '(1 2 3)])
      (*  i  i))
'(1 4 9)
> (for/list ([i "abc"]) i)
 '(#\a #\b #\c)
> (for/list ([i 4]) i)
   '(0 1 2 3)
```

A printable character normally prints as #\ followed by the represented character.

For Loop with Parallel Iteration

- When multiple [*id sequence-expr*] clauses are provided in a **for** form, **the corresponding sequences are traversed in parallel:**

```
> (for ([i (in-range 1 4)]
        [chapter '("Intro" "Details" "Conclusion")])
  (printf "Chapter ~a. ~a\n" i chapter))
```

```
Chapter 1. Intro
Chapter 2. Details
Chapter 3. Conclusion
```

for*: nested for loop

- The for* form, which has the same syntax as for, nests multiple clauses instead of running them in parallel:

```
> (for* ([book '("Guide" "Reference")]
         [chapter '("Intro" "Details" "Conclusion")])
   (printf "~a   ~a\n" book chapter))
```

```
Guide   Intro
Guide   Details
Guide   Conclusion
Reference   Intro
Reference   Details
Reference   Conclusion
```

Control Structures

- if
- when
- cond

if

- (if <bool> <expr1> <expr2>)

```
> (if (< 2 3) 0 1) => 0
> (if (< 3 2) 0 1) => 1
> (if (= 3 (+ 2 1)) 0 1) => 0
> (if (or (= 2 3) (= 3 3))
      (+2 3)
      (+ 3 3)) => 5
```

- Coding Style

```
(if <bool>
    <expr1>
    <expr2>)
```

(if (positive? -5) (error "doesn't get here") 2) ➔ 2

- **Must take three arguments, i.e. cannot omit "else"**
- else-if

```
(if <bool> <expr1>
    (if <bool>
        <expr2>
        <expr3>))
```

The true part of the top if

The else (false) part of the top if

When

- `(when <bool> <expr1>)`
- Coding Style

```
(when <bool>
      <expr1>)
```

- Must take two arguments, i.e. **cannot have an "else"**

  Like C-style if without an else.

```
> (define x 10)
> (define y 20)
> (when (< x y)
        (+ x y))
    → 30
```

Output

Output

- display
  - (**display** 1)
  - (**display** one) ??
  - (**display** "one")
- Also…
  - **print**
  - **printf**

```
>(printf "~a" 3)
3
>(printf "~a~n~a" 3 9)
3
9
> (printf "The next two squares are ~a and ~a." (* 3 3) (* 4 4))
The next two squares are 9 and 16.
```

```
> (print (* 3 4))
12

> (print "Chris Marker")
"Chris Marker"
```

  - **write**

```
> (write '(1 2 3 4))
(1 2 3 4)
```

Output (cont.)

Function	Output Style	Example Output	Notes
display	Human-readable	(1 2 3)	No quotes
print	Racket-readable	'(1 2 3)	Adds '
printf "~a"	Human-readable	(1 2 3)	Like display
write	Racket-readable	'(1 2 3)	Like print, suitable for reading back

Format Specifiers in printf

- **~a - Human-Readable Output**

  Outputs the value in a simple, human-readable format. It prints values in a way that hides extra information, so quotes are omitted for strings, and no special symbols are added for complex structures.

  e.g., (printf "~a" "Hello") ; Outputs: Hello

- **~s - Scheme-Readable Output**

  Outputs the value in a Scheme-readable (code-like) format. It retains quotes for strings and outputs lists or other data structures in a way that Racket can read back as code.

  e.g., (printf "~s" "Hello") ; Outputs: "Hello" (with quotes)

- **~v - Verbose Output**

  A more verbose output format that's often similar to ~s but adds details for certain complex structures.

Function Behavior

eval

- In Racket, by default the first list element is evaluated as a function and the remainder of elements are arguments to that function

```
> (define my-list (+ 1 2))
> my-list
3
```

- The leading “tick character” surpasses the automatic evaluation

```
> (define my-list '(+ 1 2))
> my-list
'(+ 1 2)
```

- **The `eval` function strips the leading tick of a list.**

```
> (define my-quoted-list '(+ 1 2))
> (define my-evaluated-list (eval my-quoted-list))
> my-quoted-list
'(+ 1 2)
> my-evaluated-list
3
```

eval

```
> (eval '(+ 6 6))
12
> (define fn '*)
> fn
‘*
> (define x 3)
> x
3
> (define y (list '+ x 5))
> y
'(+ 3 5)
> (eval y)
8
```

```
> (define z (list fn 10 y))
> z
'(* 10 (+ 3 5))
> (eval z)
80
```

apply

- (**apply** procedure arg1 arg2 ... args)
  - procedure: The procedure (function) to apply.
  - arg1, arg2, …: individual arguments to be passed to the procedure.
  - args: a list containing additional arguments to be passed to the procedure.

```
> (apply + '(1 2 3 4 5 6 7 8 9))
45
> (apply + 1 2 3 4 5 '(6 7 8 9))
45

> (+ 1 2 3 4 5 6 7 8 9)
45
```

A useful programming trick is to use apply in a function that takes a list of arguments, Example:

```
(define (sum s)
        (apply + s)
)

> (sum '(1 2 3))
6
```

Racket: Math

Division Operators

- (/ 10 3)

```
> (/ 10 3)
3 1/3
```

- (**quotient** 10 3)

```
> (quotient 10 3)
3
```

- (**remainder** 10 3)

```
> (remainder 10 3)
1
```

- (**modulo** 10 3)

```
> (modulo 10 3)
1
```

Other Math

- sqrt
- expt e.g., (expt 2 3) → 8
- exp e.g., (exp 3) → e^3 = 20.085536923187668
- sin, cos, tan, asin, acos, atan

Additional List Functions

```
>(member 4 '(6 3 4 8))
'(4 8)
>(member 5 '(6 3 4 8))
#f
>(null? '())
#t
>(remove-duplicates '(1 2 3 3 4 5 5 6 6 7 8 9
9 10 10 11))
'(1 2 3 4 5 6 7 8 9 10 11)
```

let

- **let** expression is used to create local bindings within a specified scope → bind variables to values and use those bindings within a specific block of code
- **Syntax**: (**let** ([variable1 value1] [variable2 value2] ... [variableN valueN]) body)
- **Variable Bindings**: a list of variable-value pairs. Each variable is bound to its corresponding value. These bindings are only accessible within the **let** body
- **Body**: the code that is executed within the scope of the **let** bindings

```
> (let ((a 2) (b 3) ) ( + a b))
5
> a
a: undefined
```

begin

- **begin** is a special form that allows you to group multiple expressions into a single body.
- The begin form evaluates each expression in sequence and returns the value of the last expression.

```
> (begin
    (display "addition")
    (display (+ 2 4))
    (newline)
    (display "multiplication")
    (display  (* 4 6))
    (newline)
)
addition6
multiplication24
```

```
> (let ([curly 0])
    (begin
      (define moe (+ 1 curly))
      (define larry (+ 1 moe))
      (list larry curly moe)
    )
)
'(2 0 1)
```

begin (cont.)

```
> (define (test x)
    (begin
      (display "the value of x is ")
      (display x)
      (newline)
    )
)
> (test 45)
the value of x is 45

> (define x 2)
(define result
  (begin
    (* x 3)
    (display "Hello, ")
    (display "World!")
    (+ x 10)))
Hello, World!
> result
12
```

```
> (define (monitor_and_add x y)
  (+
     (begin
       (display "the value of x is ")
       (display x)
       (newline)
       y
       x
     )
     y
  )
)
> (monitor_and_add 2 3)
the value of x is 2
5
```

Revisit let Example




```
> (define x 55)
(begin
     (display "before let x= ")
     (display x)
     (newline)
     (let ((x 2))
        (begin
          (display "in let, x= ")
          (display x)
          (newline)
        )
     )
     (display "after let, x= ")
     (display x)
)
before let x= 55
in let, x= 2
after let, x= 55
```

Racket:
Recursion

Recursion Examples

```
> (define (find-length lst)
  (if (null? lst)
    0
    (+ 1 (find-length (cdr lst)))
  )
)
> (find-length '(1 2 3 4))
4
```

```
> (define (factorial n)
    (if (= n 1)
      1
      (* n (factorial (- n 1)))
    )
)
> (factorial 30)
265252859812191058636308480000000
```

```
> (define (squarer xs)
    (if (empty? xs)
       empty
     (cons (sqr (first xs)) (squarer (rest xs)))
    )
)
> (squarer '(1 2 3 4 5))
'(1 4 9 16 25)
```

Determine whether a given integer number is prime

```
> (define (is-prime? n)
  (cond
    ((<= n 1) #f) ; Numbers less than or equal to 1 are not prime
    ((= n 2) #t) ; 2 is prime
    ((even? n) #f) ; Even numbers (except 2) are not prime
    (else (let loop ((divisor 3))
      (if (> (* divisor divisor) n) ; If divisor squared is greater than n, n is prime
        #t
        (if (zero? (remainder n divisor)) ; If n is divisible by divisor, it's not prime
          #f
          (loop (+ divisor 2))))))))) ; Increment divisor by 2 to check only odd divisors
```

```
> (is-prime? 7)
#t
> (is-prime? 9)
#f
> (is-prime? 131)
#t
> (is-prime? 435)
#f
```

Syntax of **named let**

```
(let name ((param1 val1)
           (param2 val2)
           ...)
  body)
```

Local Function

Retrieve all prime numbers from a List

```
> (define (getprimes List)
    (cond ((empty? List)  '())
          ((is-prime? (car List)) (cons (car List) (getprimes (cdr List))))
          (else (getprimes (cdr List)))
   )
)
> (getprimes '(1 3 5 8 11 13 345))
'(3 5 11 13)
```

Insert an element at a given position into a list.

```
> (define (insert-at elem org-list pos)
    (if (or (eq? pos 1) (eq? org-list '()))
      (cons elem org-list)
      (cons (car org-list) (insert-at elem (cdr org-list) (- pos 1)))
     )
)
> (insert-at 'alfa '(a b c d) 2)
'(a alfa b c d)
```

Logic Programming Languages

Introduction

- Logic programming is a programming paradigm that is based on the principles of formal logic.
- In logic programming, programs are written in the form of logical assertions or rules, and the execution of a program involves making logical inferences to derive new information or reach conclusions.
  - ✓**Logical Rules and Facts**
  - ✓**Declarative Nature**
  - ✓**Inference and Backtracking**
  - ✓**Pattern Matching**
  - ✓**Non-Procedural**
  - ✓**Symbolic Reasoning and AI**
- The most well-known logic programming language is **Prolog** (Programming in Logic).

Prolog (Programming in Logic)

- **Facts**: statements about the world. They are used to define relationships between objects.

  e.g., likes(john, pizza).

  likes(mary, ice_cream).

- **Rules**: Rules define relationships based on existing facts. They use logical implications to make deductions.

  e.g., happy(X) :- likes(X, sunshine).

- **Queries**: Prolog programs can be queried to ask questions based on the defined facts and rules.

  e.g., ?- likes(john, pizza).

Prolog (Cont.)

- **Variables**: Prolog allows the use of variables. Variables are denoted with a capital letter or an underscore. They can be used to find values that satisfy certain conditions.

```
?- likes(X, pizza).
```

- **Recursion**: Prolog supports recursion. Predicates can be defined in terms of themselves. This feature is used for tasks like searching through complex data structures.

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Prolog Example

Below is student-professor relations that show facts, rules, and goals.

```
studies(charlie, csc135).
studies(olivia, csc135).
studies(jack, csc131).
studies(arthur, csc134).

teaches(kirke, csc135).
teaches(collins, csc131).
teaches(collins, csc171).
teaches(juniper, csc134).
```

facts

```
professor(P, S) :- teaches(P, C), studies(S, C).
```

A rule

```
?- studies(jack, csc131).
true.

?- studies(jack, X).
X = csc131.

?- professor(kirke, Student).
Student = charlie ;
Student = olivia.
```

goals

Prolog Applications

- **Knowledge Representation:** Its ability to express relationships, facts, and rules
- **State-Space Searching**: backtracking mechanism and rule-based programming
- **Logic Problems**: Prolog is designed for expressing and solving logical problems.
- **Theorem Provers**: It can be employed to express and prove mathematical theorems using logical inference.
- **Expert Systems**: computer programs designed to emulate the decision-making ability of a human expert in a particular domain.
- **Deductive Databases**: where data is stored and queried using logical rules and relationships

Prolog Applications (cont.)

- **Agents:** implementing intelligent agents in artificial intelligence applications.
- **Symbolic Manipulations:** such as pattern matching and unification.
- **Natural Language Processing:** pattern matching and symbolic manipulation capabilities make it suitable for tasks related to natural language processing.
- **Constraint Logic Programming:** Prolog supports constraint logic programming, where constraints on variables are expressed as logical relationships.

SWI-Prolog

A widely used and actively maintained Prolog implementation that provides a robust environment for developing and running Prolog programs.

- **Open Source**: It's licensed under GPL (General Public License)
- **Cross-Platform**: designed to run on multiple platforms, including Windows, macOS, and various Unix-like OS.
- **Interactive Development Environment** (**IDE**): includes an IDE with a command-line interface → it provides a **REPL**.
- **Graphical Debugger**
- **Module System**: supports a module system that allows developers to organize their Prolog code into separate modules.
- **Predicate Indexing**: efficient indexing mechanisms for predicates, which can significantly improve the performance of queries, especially in programs with large datasets.

SWI-Prolog (cont.)

- **Built-In Libraries**: a comprehensive set of built-in libraries, including libraries for file handling, networking, constraint logic programming (CLP), and more.
- **Foreign Language Interface**: provides facilities for interfacing with code written in other programming languages, such as C.
- **Unicode Support**: supports Unicode characters, allowing developers to work with a wide range of character sets and languages.
- **Extensive Documentation**: thorough documentation, including manuals, tutorials, and reference materials.
- **Active Community**: an active and responsive community of users and developers.

Download SWI-Prolog today at:

http://www.swi-prolog.org/

Fundamentals of Logic Programming

- **Axioms (Facts and Rules)**:
  - **Facts**: Statements that are assumed to be true → represent basic knowledge.
  - **Rules**: Logical implications or relationships between facts. → define how new information can be inferred.
- **Goal Statements**:
  - Statements or queries that the programmer wants to prove or satisfy → the desired outcomes or information sought.
- **Axioms as a Theory**:
  - The collection of facts and rules forms the knowledge base or theory of the logic program → the initial set of assumptions
- **Goal Statements as Theorems**:
  - The theorems or propositions that the programmer aims to prove or derive from the given axioms.

Fundamentals of Logic Paradigm (cont.)

- **Computation as Deduction**:
  - Computation involves deducing new information from the axioms to prove the goal statements. → The process is often based on logical inference.
- **Interpreter and Inference**:
  - The interpreter is responsible for finding a set of axioms and inference steps that lead to the proof of the goal statements. → deduce conclusions by performing logical inference
- **Proof**:
  - The successful determination of a collection of axioms and inference steps that imply the goal statements constitutes a proof → The interpreter's goal is to find a valid proof.

GOAL: find a set of facts and rules that logically entail the desired outcomes.

Relational Programming

In Prolog, a relation is specified as a predicate, and the programming paradigm is often referred to as relational programming.

- **A predicate is a tuple:** a relationship that defines a logical connection between its arguments.
- **Contrast to Functional Programming:**
  - In functional programming, functions map input values to output values, and the result is deterministic.
  - In Prolog, relations express facts or logical connections without emphasizing the idea of computation as a mapping of inputs to outputs.
  - E.g., brother(sam, bill). brother(sam, bob). → **brother** maps "sam" to two different range elements which is a relation (not a function)
- **N-ary Relations:**
  - Relations in Prolog can be n-ary → involve more than two entities.
  - E.g., **family**(jane, sam, [ann, tim, sean]) → **family** relation involves three entities: Jane, Sam, and a list of children.

Defining Relations (predicates) by Facts

- parent(tom, bob)
- parent is the name of a relation (fact)

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
goodperson(bob).
likes(tom, apples, oranges).
```

Directionality in Parameters

- In traditional programming languages like C, Java, or Python, parameters have a specific directionality → they serve as inputs (in) or outputs (out). For example,

```
int add(int a, int b, int *result) {
    *result = a + b;
}
```

a and b are in parameters
result is out parameters

- In Prolog, parameters are not strictly categorized as "in" or "out."
- Variables can be used in both input and output positions.
- When a query is executed, the Prolog engine tries to bind variables to values in a way that satisfies the given rules and facts.

```
square(X, Result) :- Result is X * X.
```

Both X and Result can be considered as both in and out parameters.

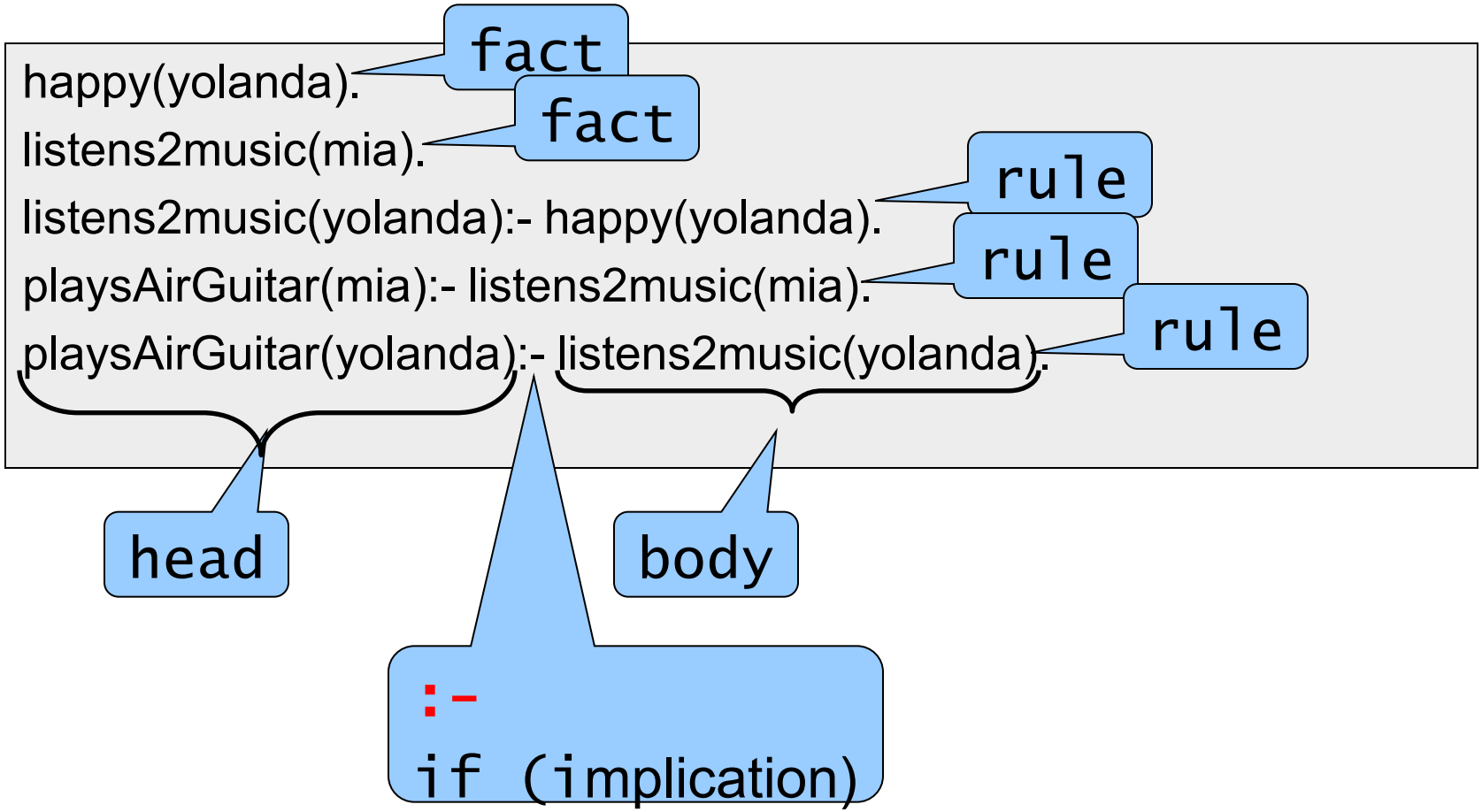
Given the fact square (2, 4), you can ask square (X, 4) and Prolog will bind X to 2, or you can ask square (2, X) and Prolog will bind X to 4.

Example: Knowledge Base 1

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

```
?- woman(mia).
true
?- playsAirGuitar(jody).
true
?- playsAirGuitar(mia).
false
?- tattoed(jody).
ERROR: Unknown procedure: tattoed/1 (DWIM could not correct goal).
?- party.
true
```

Example: Knowledge Base 2



Example: Knowledge Base 2

```
happy(yolanda).
listens2music(mia).
listens2music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2music(mia).
playsAirGuitar(yolanda):- listens2music(yolanda).
```

```
?- playsAirGuitar(mia).
true
?- playsAirGuitar(yolanda).
true
```

Clauses

In Prolog, a clause is a fundamental unit of logic programming that consists of facts and rules. Clauses define relationships and conditions in the form of logical statements.

- **Facts**: simple statements about the relationships between entities. They are often used to represent base-level knowledge or assertions.
- **Rules**: express relationships based on conditions. They consist of a head and a body: the head specifies the relationship, and the body contains conditions that must be satisfied for the relationship to hold.

```
happy(yolanda).
listens2music(mia).
listens2music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2music(mia).
playsAirGuitar(yolanda):- listens2music(yolanda).
```

- There are five clauses in this knowledge base: two facts and three rules

Predicates

- In Prolog, a predicate is a logical statement or relationship that defines a condition or property.
- Predicates are used to express knowledge about the relationships among different entities or the satisfaction of certain conditions.
- Predicates consist of terms, which can be atoms, variables, or complex structures.
- The key components of predicates in Prolog:
  - **Predicate Name**: identifies the relationship or condition being described. It usually starts with a lowercase letter.
  - **Arguments:** Predicates take one or more arguments, which can be constants, variables, or complex terms.
  - **Arity**: the number of arguments it takes
- Predicates are defined by clauses, which can be either facts or rules.
- Prolog interpreter uses these predicates to answer queries and derive conclusions based on the defined logic.

Predicates (cont.)

```
happy(yolanda).
listens2music(mia).
listens2music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2music(mia).
playsAirGuitar(yolanda):- listens2music(yolanda).
```

- There are three predicates in this knowledge base: **happy**, **listens2music**, and **playAirGuitar**.

Expressing Conjunction (and)

- In Prolog, conjunction (logical "and") is expressed using a comma ‘,’ . When multiple goals (conditions) are separated by commas within a clause, all of those goals must be satisfied for the clause to be true.

```
happy(vincent).
listens2music(butch).

playsAirGuitar(vincent):- listens2music(vincent) , happy(vincent).
playsAirGuitar(butch):- happy(butch).
playsAirGuitar(butch):- listens2music(butch).
```

- Vincent plays Air Guitar if Vincent listens to music **and** feels happy.

Example: Knowledge Base 3

```
happy(vincent).
listens2music(butch).
playsAirGuitar(vincent):- listens2music(vincent), happy(vincent).
playsAirGuitar(butch):- happy(butch).
playsAirGuitar(butch):- listens2music(butch).
```

```
?- playsAirGuitar(vincent).
false
?- playsAirGuitar(butch).
true
```

Expressing Disjunction (or)

- In Prolog, disjunction (logical "or") is expressed using a semicolon ‘;’ . When multiple goals (conditions) are separated by semicolons within a clause, Prolog will try to satisfy **at least one** of those goals for the clause to be true.

```
happy(vincent).
listens2music(butch).
playsAirGuitar(vincent):- listens2music(vincent), happy(vincent).
playsAirGuitar(butch):- happy(butch).
playsAirGuitar(butch):- listens2music(butch).
```



```
happy(vincent).
listens2music(butch).
playsAirGuitar(vincent):- listens2music(vincent), happy(vincent).
playsAirGuitar(butch):- happy(butch); listens2music(butch).
```

Logic Operators in Prolog

In Prolog, logical operators are used to express relationships and conditions.

- Operators
  - Implication :-
  - Conjunction ,
  - Disjunction ;
- Negation

```
playsAirGuitar(vincent):- not(listens2music(vincent)).
```

Variables in Prolog

In Prolog, variables are used to represent unknown values or placeholders within predicates.

- **Format of Variables**: Variables in Prolog are represented as strings of letters, digits, and underscores. They must begin with a capital letter or an underscore. Examples:

  X, Sister, _, _thing, _y47, First_name, Z2

- **Don't-Care Variable** ('_'): when the specific value of the variable is not relevant or used in the clause. → It signifies that we are not interested in the value of the variable, and it is often used as a placeholder. Examples:

  married(john, _).

  This suggests that John is married, but we're not providing information about his spouse.

Comments in Prolog

- **Single-Line Comment**:

  Anything following the `**%**' symbol on the same line is treated as a comment.

```
% This is a single-line comment in Prolog
predicate(X) :- clause(X).
```

- **Multi-Line Comment**:

  Anything between `**/***' and `***/**' is considered a comment, and it can span multiple lines.

```
/*
   This is a multi-line comment in Prolog.
   It can span multiple lines and is enclosed
   between /* and */.
*/
predicate(X) :- clause(X).
```

Atoms

- In Prolog, an atom is a basic data type that represents a constant.
- An atom is used to represent a name, label, or identifier and consists of a sequence of letters, numbers, and underscores, starting with a lowercase letter or an arbitrary sequence of characters enclosed in single quotes.
- Examples:

butch, big_kahuna_burger, playGuitar,

'Vincent', 'Five dollar shake', '@$%'

Numbers

- In Prolog, numbers can be expressed as integers or floating-point numbers.
- Integer numbers are sequences of digits, optionally preceded by a sign (‘+’ or ‘-”).

  12, -34, 22342

- Floating-point numbers consist of an integer part, a decimal point, a fractional part, and an exponent.
- The exponent is indicated by the letter ‘e’ followed by an integer.

  34573.3234, -0.001, 3.1415e-3

Arity

- In Prolog, the term "arity" refers to the number of arguments or operands that a predicate takes.

  woman(mia). — a unary predicate (an arity of 1)

  loves(vincent, mia). — a binary predicate (an arity of 2)

- A predicate is usually indicated with the suffix "/" followed by a number to indicate the arity.
- A predicate with different arities are considered distinct predicates.

  loves(vincent, mia).

  loves(john, mia, judy).

  loves/2 and loves/3 are treated as different predicates due to their distinct arities.

- Prolog uses the predicate's name and arity to uniquely identify predicates, allowing for different behaviors based on the number of arguments provided.

Example of Arity

```
loves(vincent, mia)
happy(yolanda).
listens2music(mia).
listens2music(yolanda):- happy(yolanda).
playsAirGuitar(mia):- listens2music(mia).
playsAirGuitar(yolanda):- listens2music(yolanda).
```

- This knowledge base defines
  - loves/2
  - happy/1
  - listens2music/1
  - playsAirGuitar/1

Prolog Variables

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- woman(X).
```

Variable Instantiation

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- woman(X).
X=mia
```

Asking Alternatives

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- woman(X).
X=mia;
```

Asking Alternatives

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- woman(X).
X=mia;
X=jody
```

Asking Alternatives

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- woman(X).
X=mia;
X=jody;
X=yolanda
```

Asking Alternatives

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- woman(X).
X=mia;
X=jody;
X=yolanda.
```

Conjunctive query

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- loves(marsellus,X), woman(X).
```

Conjunctive query

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- loves(marsellus,X), woman(X).
X=mia.
?-
```

Conjunctive query

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- loves(pumpkin,X), woman(X).
```

Conjunctive query

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- loves(pumpkin,X), woman(X).
false
?-
```

Disjunctive query

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

```
?- loves(pumpkin,X); woman(X).
X = honey_bunny ;
X = mia ;
X = jody ;
X = yolanda.
?-
```

Variables in Prolog Rules

- Rules allow the interpreter to infer things
- Rules generally contain variables

```
employed(X) :- employs(Y,X).
```

can be read as:

```
for all X, X is employed if there
exists a Y such that Y employs X
```

Variables in Prolog Rules

```
grandmother(A, C):- mother(A, B),mother(B, C).
```

can be read as:

```
for all A, C [A is the grandmother of C if
there exists a B such that A is the mother of B
and B is the mother of C].
```

We probably want another rule that says:

```
grandmother(A, C) :- mother(A, B),father(B, C).
```

Variables in Prolog Rules

- Facts:

```
rainy(dallas).
cold(dallas).
rainy(houston).
```

- Rule:

```
snowy(X):- rainy(X), cold(X).
```

- Query: ?- snowy(A)
- Given these facts and rules along with this query, variable A would be instantiated (or unified with) to dallas
- Prolog returns,

?- snowy(A).

A = dallas.

Knowledge Base 5

```
loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).

jealous(X,Y):- loves(X,Z), loves(Y,Z).
```

```
?- jealous(marsellus,W).
W=vincent;
W= marsellus
?-
```

Horn Clauses

- Horn clauses are a specific form of logical expressions commonly used in logic programming.
- A Horn clause defines a rule in Prolog,

$H{:-}B_1,B_2,\ldots,B_n$ or $H{:-}\ B_1;B_2;\ldots;B_n$

- Here: H is the head of the clause, a single goal or predicate.
- $B_1,B_2,\ldots,B_n$ (or $B_1;B_2;\ldots;B_n$) are the body of the clause, a conjunction (or disjunction) of goals or predicates.
- :- can be read as "if" or "is implied by.“
- If there are no body predicates on the right side, the Horn clause becomes a fact.

Prolog Programming Essentials

- **Prolog Programming Model**: Facts and Rules, Queries
- **Principle of Resolution:**
  - ✓ The process of resolving a query by searching for applicable rules and facts in order to prove the query true or false.
  - ✓ Applying rules and facts through a process of pattern matching and unification.
- **Unification:**
  - ✓ The process of finding substitutions for variables in order to make two terms identical.
  - ✓ When a query is posed, Prolog attempts to unify the terms in the query with the terms in the program's facts and rules.
- **Execution Order:**
  - ✓ Follow a depth-first search strategy.
  - ✓ The order of clauses in the program can be crucial and affects the behavior of the program, particularly when it comes to backtracking.

Prolog Programming Essentials (cont.)

- **Backtracking:**
  - ✓ A key feature of Prolog that allows exploring alternative paths when a particular search path fails to satisfy a goal.
- **Other important aspects:**
  - ✓ Arithmetic Operations: Allowing the formulation of rules based on numerical calculations.
  - ✓ Lists: A fundamental data structure in Prolog, allowing the representation of sequences of elements.
  - ✓ Cut: A powerful but complex feature used to control backtracking and commit to choices made during execution.
  - ✓ Negation: Provides a way to express negation and state that a goal should not be true.

A Prolog Program about Food

```
indian(curry).
indian(dahl).
indian(tandoori).
indian(kurma).
chinese(chow_mein).
chinese(chop_suey).
chinese(sweet_and_sour).
italian(pizza).
italian(spaghetti).
mild(dahl).
mild(tandoori).
mild(kurma).

likes(sam,Food) :-  indian(Food),  mild(Food).
likes(sam,Food) :-  chinese(Food).
likes(sam,Food) :-  italian(Food).
likes(sam,chips).
```

Principle of Resolution

- The principle of resolution governs the execution process.
  - ✓ If C_1 and C_2 are Horn clauses, and the head of C_2 matches one of the terms in the body of C_1, then the term in C_1 can be replaced with C_2.
- For example, given

  C_1: likes(sam,Food) :- indian(Food), mild(Food).

  C_2: indian(dahl).

  C_3: mild(dahl).

  by applying the principle of resolution and instantiating the variable 'Food' to 'dahl'
  - ✓ we can replace the first and second terms in C_1 with C_2 and C_3, respectively.
  - ✓ Consequently, the goal 'likes(sam, dahl)' can be successfully proven.

Unification

- The process of associating (binding) variables and values is accomplished through unification.
- Variables that receive a value during this process are said to be instantiated.
- The rules of unification in Prolog are as follows:

Unification rules

- A constant unifies only with itself.
- Two structures unify if and only if they share the same predicate and the same number of arguments, and the corresponding arguments **unify recursively**.
- A variable can unify with anything, e.g., other variables, constants, or complex structures.

Unify Recursively

- The unification process extends beyond the top-level terms to encompass the sub-components or arguments within compound terms.
- It allows Prolog to thoroughly check and establish equality not only at the surface level but also within the internal structure of terms.
- Example:

```
father(john, jim).
father(john, ann).
siblings(X, Y) :- father(Z, X), father(Z, Y).

?- siblings(jim, ann).
```

- Prolog unifies the variables X and Y with jim and ann, respectively. Additionally, the variable Z is unified with the common father, creating a recursive unification process within the structure of the rule.

Unification - Equality

- In Prolog, equality is fundamentally tied to unification.
- When two terms can be unified, they are considered equal in the context of Prolog.
- An equality goal is often used to express equality in Prolog, which employs an infix predicate **=**, the equality operator
- For instance,

```
?- dahl = dahl.
true.
?- dahl = curry.
false.
?- likes(Person, dahl) = likes(sam, Food).
Person = sam,
Food = dahl.
?- likes(Person, Food) = likes(sam, Food).
Person = sam.
```

Unification Process

- **Constant Unification**:

```
?- apple = apple. % constant apple unifies w/ itself
true.
```

- **Variable Binding**:

```
?- X = 42.   % the variable X is instantiated and bound
X = 42.      % to the constant 42
```

- **Variable Unification**:

```
?- Y = apple. % a variable Y can unify with a constant apple,
Y = apple.    % binding the variable to that value.
```

- **Structural Unification**:

```
?- color(red, Color) = color(red, crimson). % same predicate
Color = crimson. % w/ same number of arguments; Unification
                 % binds the variable Color to crimson
```

- **Failure in Unification**:

```
?- red = blue.    % Mismatched constants result in a failed
false.            % unification.
```

- **Variable Identification**:

```
parent(X, Y) :- mother(X, Y); father(X, Y).
% recognizing the variable as a unique, unbound placeholder in
% Prolog's initial parsing and understanding of a clause.
% X and Y are identified as variables when Prolog parses the
% rule → they can later be bound to specific values.
```

Unification Example

```
likes(sue,fondue).
likes(bob,fondue).

friends(X,Y) :- likes(X, Something), likes(Y, Something).
```

Query: Find out who is friends with Sue?

```
   ?- friends(sue,Y).

% X can be unified with sue in the above rule

friends(sue,Y) :- likes(sue, Something), likes(Y, Something).

% Something can be unified with fondue
% Thus, replace Something in friends body with fondue:

   friends(sue,Y) :- likes(sue, fondue), likes(Y, fondue).

% remove the first clause from the body of this rule:

   friends(sue, Y) :- likes(Y, fondue).

% now we try to satisfy the second goal: likes(Y, fondue).
% Y can  be unified with sue and bob: likes(sue, fondue), and
   bob: likes(bob, fondue)
Finally the answer to the original query,
       Y = sue ;
       Y = bob.
```

Execution Order

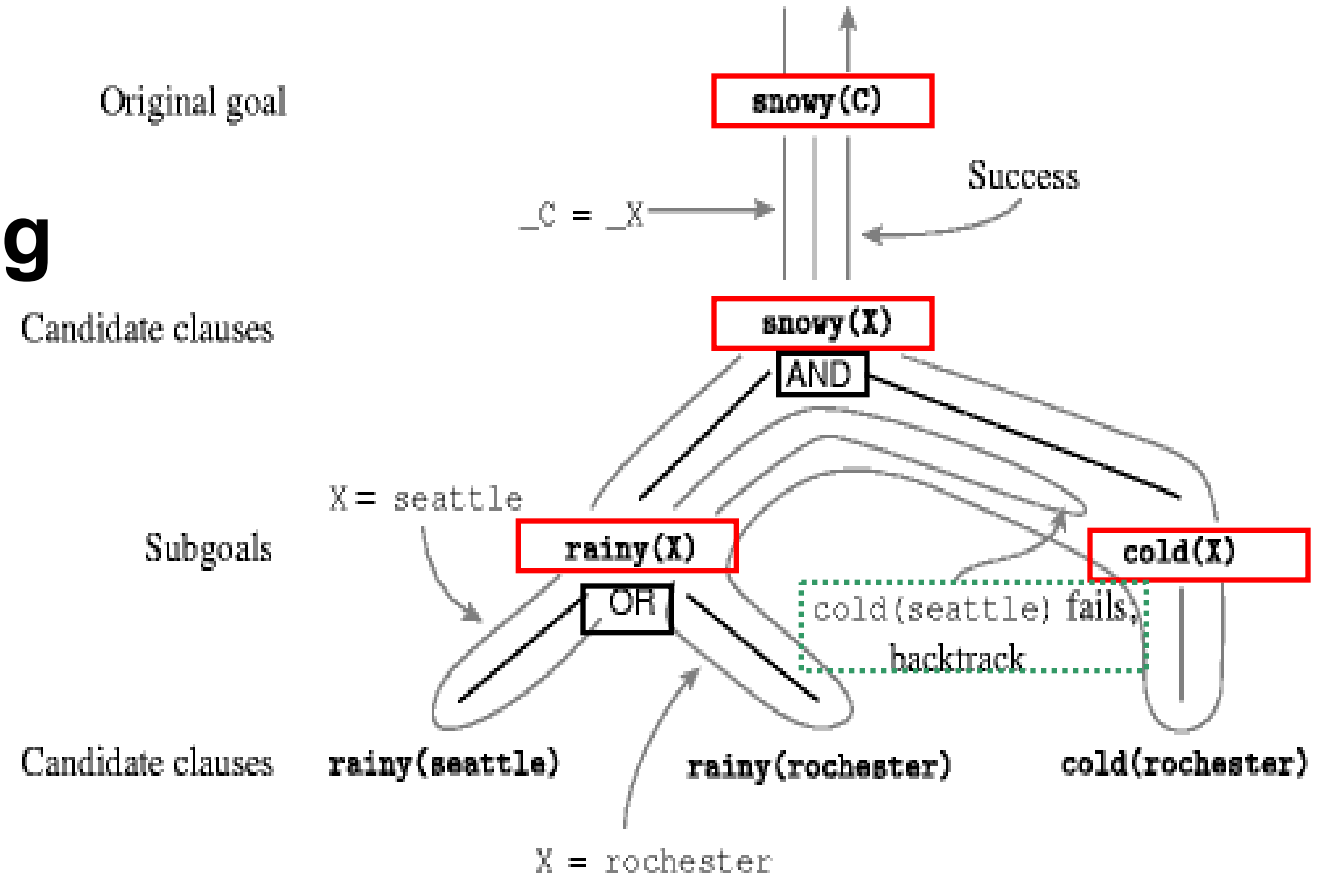
- Prolog employs a resolution sequence search, automatically conducted by the Prolog interpreter, to find a solution that satisfies the given goal.
- Search Strategy: **Forward Chaining**
  - ✓ Derives the goal from the axioms or facts in the knowledge base.
  - ✓ More efficient when the number of facts is small, and the number of rules is extensive.
- Search Strategy: **Backward Chaining**
  - ✓ Starts with the goal and attempts to resolve it by working backward through rules and facts.
  - ✓ Generally more efficient, serving as the primary mechanism in the execution of Prolog programs.
- If rules are complex and involve many conditions, backward chaining might be more effective.
- Some problems naturally lend themselves to forward chaining, while others may benefit more from backward chaining.

Backward Chaining in Prolog

- Backward chaining follows a classic **depth-first backtracking** algorithm
- Example
  - Goal:

  Snowy(C)

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X)
```



Depth-First Backtracking

- Prolog explores the search space in a depth-first manner.
- If the system encounters a point where it cannot proceed, it backtracks to the most recent choice point and explores alternative choices.
- **The order of rules and facts is crucial**. Prolog considers rules and facts in the order they are defined → starts with the first rule that matches the query and then proceeds to subsequent rules if needed.
- Different orderings may lead to different solutions, affect the efficiency of the search, or result in non-terminating resolutions.
- Example,

```
parent(john, mary).
parent(john, bob).
parent(mary, alice).
parent(bob, carol).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

?- ancestor(X,carol).
X = bob;
X = john.
```

```
parent(john, mary).
parent(john, bob).
parent(mary, alice).
parent(bob, carol).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
ancestor(X, Y) :- parent(X, Y).

?- ancestor(X,carol).
X = john;
X = bob.
```

Rules order changed

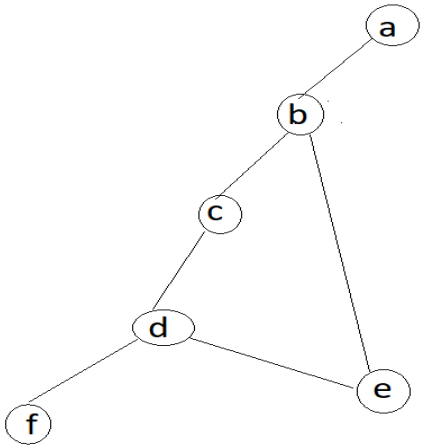
Depth-First Backtracking (example)

```
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(b, e).
edge(d, f).

path(X, X).
path(X, Y) :- path(X, Z), edge(Z, Y).
```

?- path(b,b).
true .

?- path(a,f).
true .



?- path(a,c).

Call: (10) path(a, c)	**Goal path(a,c) is called: find if there is a path from 'a' to 'c'.**
Call: (11) path(a, _30004)	**Look for a general path from 'a' to an unknown node _30004**
Exit: (11) path(a, a)	**Find that there is a direct path from 'a' to 'a'.**
Call: (11) edge(a, c)	**Check if there is an edge from 'a' to 'c'.**
Fail: (11) edge(a, c)	**No such edge, so this goal fails.**
Redo: (11) path(a, _30004)	**Backtracks to explore alternative paths.**
Call: (12) path(a, _34056)	**Look for a general path from 'a' to an unknown node _34056**
Exit: (12) path(a, a)	**Find that there is a direct path from 'a' to 'a'.**
Call: (12) edge(a, _30067)	**Checks for an edge from 'a' to an unknown node _30067**
Exit: (12) edge(a, b)	**Find an edge from ‘a’ to ‘b’.**
Exit: (11) path(a, b)	**Find a path from 'a' to 'b'.**
Call: (11) edge(b, c)	**Check for an edge from 'b' to 'c'**
Exit: (11) edge(b, c)	**Find an edge from ‘b’ to ‘c’.**
Exit: (10) path(a, c)	**Find a path from 'a' to 'c’.**
true.	**Result**

Depth-First Backtracking (example)

Different order of the recursive rules results in non-terminating resolutions

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```

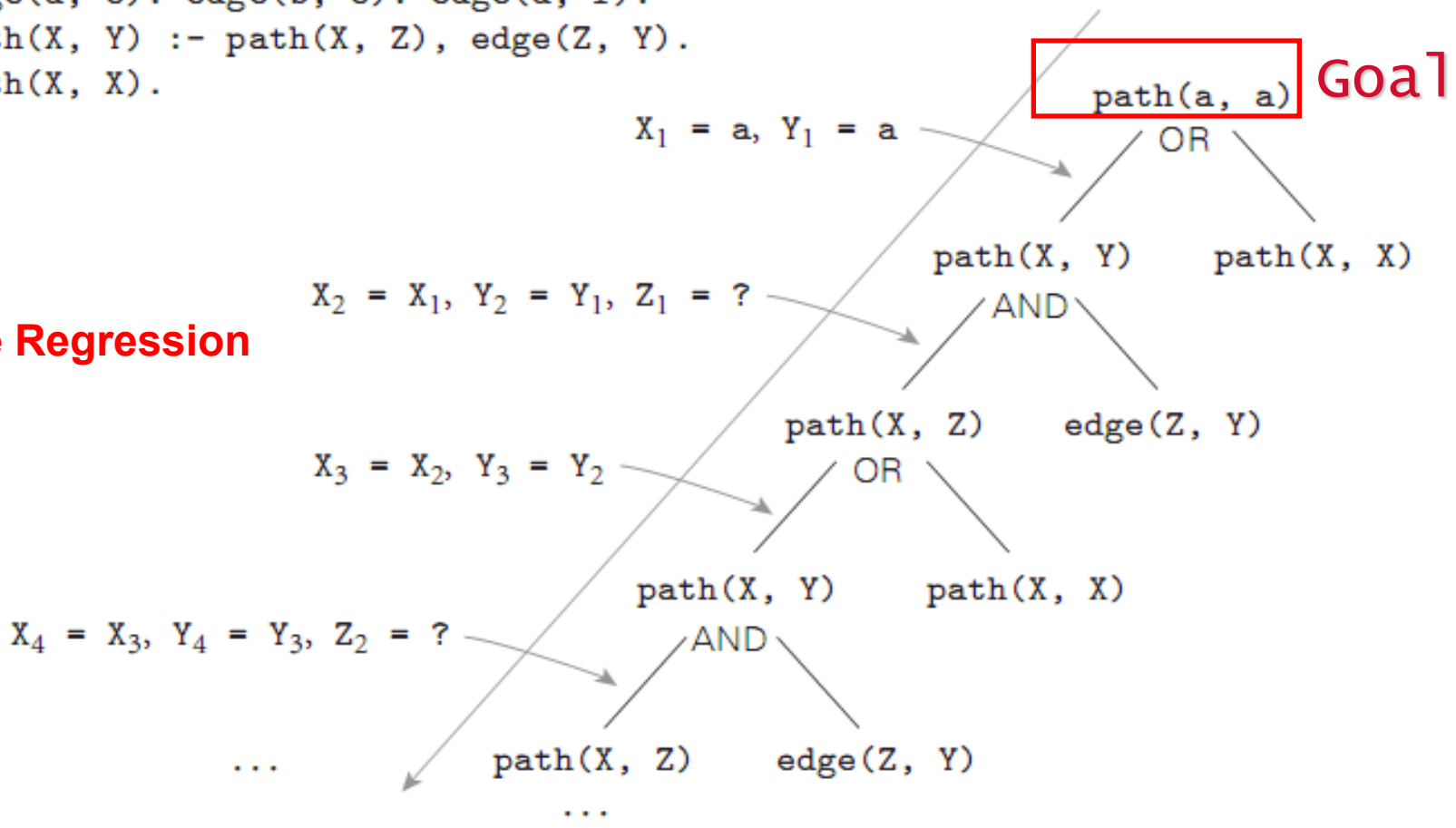


Figure 11.2 **Infinite regression in Prolog.** In this figure even a simple query like `?- path(a, a)` will never terminate: the interpreter will never find the trivial branch.

Prolog is Not Purely Declarative

- The order in which facts and rules are defined matters.
- A left-to-right pursuit of sub-goals: part of the backtracking mechanism. → it can be viewed as imperative.
- Changing the order of clauses can indeed alter the behavior of programs, which may lead to different results.
- Incorrect ordering of clauses or the definition of recursive rules without proper termination conditions can lead to infinite loops during the execution of queries.
- The program efficiency can be impacted by the ordering of clauses. Suboptimal orderings might result in less efficient searches.

Prolog programmers need to be mindful of how they structure their knowledgebase and rules to achieve desired outcomes.

Arithmetic in Prolog

- While Prolog is not typically the language of choice for heavy-duty mathematics due to its primary focus on logic programming, it does include arithmetic capabilities.
- The **is** operator is used for evaluating arithmetic expressions in Prolog.
- The pattern for evaluating arithmetic expressions:

X is Expression, where X is instantiated to the value of Expression

```
?- X is 10 + 5.
X = 15.
?- X is 10 - 5.
X = 5.
?- X is 10 * 5.
X = 50.
?- X is 10 / 5.
X = 2.
?- X is 10 + 5 * 6 / 3.
X = 20.
?- is(X, 1+2).
X = 3.
```

```
?- A = 2, B = 3, X = A + B.
A = 2,
B = 3,
X = 2+3.
?- A = 2, B = 3, X is A + B.
A = 2,
B = 3,
X = 5.
?- X is Y.
ERROR: Arguments are not
sufficiently instantiated
```

Arithmetic in Prolog (cont.)

- Prolog also supports comparison operators in **arithmetic expressions, =:=, =\=, <, >**

```
?- X is 5, Y is 5, X =:= Y.
X = Y, Y = 5.
?-  X is 5, Y is 7, X =:= Y.
false.

?- Y is 5, Y =\= 3.
Y = 5.
?- Y is 3, Y =\= 3.
false.
```

```
?- Z is 12, Z > 10.
Z = 12.
?- Z is 10, Y is 10, Z > Y.
false.

?- W is 7, X is 8, W < X.
W = 7.
?- W is 10, W < 8.
false.
```

- '**=:=**' ('**=\=**') is different from '**==**' ('**\=**') in Prolog.
- '**==**' ('**\=**') is used for equality (inequality) comparison. It succeeds if the two terms on either side can (cannot) be unified.

```
?- X = 5, Y = 5, X == Y.
X = Y, Y = 5.
?-  X = [1,2,3],Y = [1,2,3], X == Y.
X = Y, Y = [1, 2, 3].
?- X = [1,2,3],Y = [1,3,2], X == Y.
false.
```

Example: Factorial

fact(0,1).

fact(A, B) :- A > 0, C is A-1, fact(C, D), B is A*D.

```
?- fact(3,X).
X = 6 .
```

1. **1st clause**, fact(0,1)., does not match, move to the next clause.
2. **2nd clause**:
   - Unify 'A' with '3', 'B' is the variable TBD
   - Evaluate 'A > 0': true
   - Calculate 'C is A-1': C=2
   - Recursive call 'fact(2,D)'
3. **Recursive call with A=2**
   - Repeat the same process for fact(2,D)': 'A>0' true, C=1, recursive call 'fact(1,E)'
4. **Recursive call with A=1**
   - Repeat the same process for 'fact(1,E)': 'A>0' true, C=0, recursive call 'fact(0,F)'
5. **Base case with A=0**
   - 'F' is unified w/ '1' (result of 'fact(0,F)')
6. **Backtracking to Previous Recursive Call:**
   - The value of 'E' (result of 'fact(1,E)') is '1*1=1'
7. **Backtracking to Previous Recursive Call:**
   - The value of 'D' (result of 'fact(2,D)') is '2*1=2'
8. **Backtracking to Original Query:**
   - The value of 'B' (result of 'fact(3,B)') is '3*2=6'
9. **Result:** X = 6.

```
?- fact(5,2).
false.
```

Lists in Prolog

- In Prolog, lists are a fundamental data structure that represents sequences of elements.
- A list is written in square brackets, and elements are separated by commas. The empty list is denoted by '[]'.

```
EmptyList = [].
NumericList = [1, 2, 3, 4, 5].
NestedList = [a, [b, c], d].
```

- Lists in Prolog can be constructed using the '[Head | Tail]' notation where 'Head' is the first element of the list, and 'Tail' is the rest of the list

```
?- List = [1, 2, 3, 4, 5],[First|Rest] = List.
List = [1, 2, 3, 4, 5],
First = 1,
Rest = [2, 3, 4, 5].
```

Decomposing lists: head

User defined "**first**" predicate (similar to **car** in LISP)

- **Option 1:**

  first(X, List) :- List=[X | Xs].

- **Option 2:**

  first(X, [X | Xs]).

- **Option 3:**

  first(X, [X | _]).

```
?- first(X,[1,2,3,4,5]).
X = 1.
?- first(2,[1,2,3,4,5]).
false.
?- first(1,[1,2,3,4,5]).
true.

?- first(6,[6,7,8,9,10]).
true.
?- first(9,[6,7,8,9,10]).
false.
?- first(X,[6,7,8,9,10]).
X = 6.

?- first(X,[2,1,7,11,0]).
X = 2.
?- first(2,[2,1,7,11,0]).
true.
?- first(11,[2,1,7,11,0]).
false.
```

Decomposing lists: tail

- User defined '**tail/2**' predicate (similar to **cdr** in LISP)

  tail([X|Y], Y).

```
?- tail([1,2,3,4,5],Y).
Y = [2, 3, 4, 5].
?- tail([1,[2,3],4,5],Y).
Y = [[2, 3], 4, 5].
?- tail([1,2,3,4,5],[2,3,4,5]).
true.
?- tail([1,2,3,4,5],[3,4,5]).
false.
```

- User defined '**second/2'** predicate

  second([_ , Second | _], Second).

```
?- second([a, b, c, d, e], Sec).
Sec = b.
?- second([a, [b, c], d, e], Sec).
Sec = [b, c].
?- second([a, b], Sec).
Sec = b.
?- second([a], Sec).
false.
```

Decomposing lists: n^{th} element

- Built-in predicate '**nth0/3**' (the index starts from 0)

  nth0(N, Lst, X).

```
?- nth0(3, [1,2,3,4,5], X).
X = 4.
```

- Built-in predicate '**nth1/3**' (the index starts from 1)

  nth1(N, Lst, X).

```
?- nth1(3, [1,2,3,4,5], X).
X = 3.
```

Decomposing lists: Examples

- Accessing Individual Elements:

  element_at_n(N, List, Element) :- nth0(N, List, Element).

```
?- element_at_n(2,[1,2,3,4,5],X).
X = 3.
?- element_at_n(0,[1,2,3,4,5],X).
X = 1.
?- element_at_n(5,[1,2,3,4,5],X).
false.
```

- Extracting even numbers from a list:

  extract_evens([], []).

  extract_evens([H|T], [H|Result]) :- 0 is H mod 2, extract_evens(T, Result).

  extract_evens([_|T], Result) :- extract_evens(T, Result).

```
?- extract_evens([1,2,3,4,5,6], X).
X = [2, 4, 6].
```

List Manipulation

Prolog provides built-in predicates for working with lists. Some common predicates include:

- **length/2**: Calculates the length of a list.

```
?- length([1, 2, 3], Len).
Len = 3.
```

- It can also be user defined as,

```
list_length([], 0).
list_length([H|T], N) :- list_length(T, N1), N is N1+1.
```

```
?- list_length([1,2,3,4], N).
N = 4.

?- list_length([1,[2,3,4],5],Len).
Len = 3.
```

List Manipulation (cont.)

- **member/2**: Checks if an element is a member of a list.

```
?- member(2, [1, 2, 3]).
true .
?- member(5, [1, 2, 3]).
false.
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3.
```

- It can also be user defined as,

```
is_member(X, [X|T]).
is_member(X, [H|T]) :- is_member(X, T).
```

```
?- is_member(3, [1,2,3]).
true .
?- is_member(5, [1,2,3]).
false.
?- is_member(X,[[1,2],3]).
X = [1, 2] ;
X = 3 .
```

List Manipulation (cont.)

- **append/3:** Concatenates two lists

```
?- append([1, 2], [3, 4], Result).
Result = [1, 2, 3, 4].
?- append([1,[2,3,4]],[5,6],Result).
Result = [1, [2, 3, 4], 5, 6].
```

- It can also be user defined as,

```
list_append([], L, L).
list_append([H|T], L, [H|L2]) :- list_append(T, L, L2).
```

```
?- list_append([1,2], [3,4,5], X).
X = [1, 2, 3, 4, 5].
?- list_append([1,2], Y, [1,2,3,4,5,6]).
Y = [3, 4, 5, 6].
?- list_append(A,B,[1,2,3]).
A = [],
B = [1, 2, 3] ;
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
A = [1, 2, 3],
B = [] .
```

List Predicates Examples

Prolog allows users to define predicates that operate on lists.

- A predicate to calculate the sum of a list:

```
sum_list([], 0).
sum_list([Head|Tail], Sum) :-  sum_list(Tail, RestSum), Sum is Head + RestSum.
```

```
?- sum_list([1,2,3,4,5],Sum).
Sum = 15.
```

- A predicate to verify if a list is sorted:

```
sorted([]).
sorted([_]).
sorted([A, B | T]) :- A =< B, sorted([B | T]).
```

```
?-  sorted([1,2,3,4,5]).
true .
?- sorted([1,2,3,3,4,5]).
true .
?- sorted([2,5,7,4,8,9]).
false.
```

List Predicates Examples (cont.)

- Find the last element of a list:

```
last_element(X, [X]).
last_element(X, [ _ | L]) :- last_element(X, L).
```

```
?- last_element(X,[a,b,c,d,e]).
X = e .
?- last_element(X,[a,b,c,[d,e]]).
X = [d, e]
```

- Create a list containing all integers within a given range, e.g., range(4,9,L).→ L = [4,5,6,7,8,9]

```
range(I, I, [I]).
range(I, K, [I|L]) :- I < K, I1 is I + 1, range(I1,K,L).
```

```
?- range(5,8,L).
L = [5, 6, 7, 8].
?- range(8,4,L).
false.
```

List Predicates Examples (cont.)

- Duplicate the elements of a list: e.g., duplicate([a, b, b, c], X). → X = [a, a, b, b, b, b, c, c]

```
duplicate([], []).
duplicate([X|Xs], [X, X|Ys]) :- duplicate(Xs, Ys).
```

```
?- duplicate([a, b, b, c], X).
X = [a, a, b, b, b, b, c, c].
```

- Splitting a list into two parts: elements less than X and elements greater than or equal to X

```
split_list(_, [], [], []).
split_list(X, [H|T], [H|LT], GE) :- H < X, split_list(X, T, LT, GE).
split_list(X, [H|T], LT, [H|GE]) :- H >= X, split_list(X, T, LT, GE).
```

```
?- split_list(8,[3,10,5,2,19,15,20,7,8,11,5],Less,GreaterEqual).
Less = [3, 5, 2, 7, 5],
GreaterEqual = [10, 19, 15, 20, 8, 11]
```

Control the Control Flow?

- Revisit the list predicate **is_member**,

  is_member(X, [X | _]).

  is_member(X, [_ | T]):- is_member(X, T).

```
?- is_member(e,[a,e,r,d,e,k,e]).
true ;
true ;
true .
```

- If a given atom a appears in the list n times, the goal

```
?- is_member(a, L)
```

  can succeed n times.

- This behavior is a result of Prolog's backtracking mechanism, allowing it to explore multiple solutions for a goal.
- A way to control backtracking and limit the number of solutions returned?

The Cut Operator: !

- The cut operator **!** is used to control backtracking and prune the search space. It is a way to prevent backtracking over specific branches.

```
is_member(X, [X | _]):- !.
is_member(X, [_ | T]):- is_member(X, T).
```

```
?- is_member(e,[a,e,r,d,e,k,e]).
true .
```

- The cut operator is used after the first clause to prevent backtracking into the other clause **once a match is found**.
- The cut operator can be a powerful tool for controlling the control flow and preventing unnecessary backtracking in Prolog programs.
- Another option is to force the first element of the list not to be equal to X in the second rule,

```
is_member(X, [X | _]).
is_member(X, [H | T]):- X \= H, is_member(X, T).
```

Cut Operator: Examples

- **Factorial number**:

```
fact(N,1) :- N = 0, !.
fact(N,F) :- N > 0, M is N -1, fact(M,F1), F is N * F1.
```

```
?- fact(0,F).
F = 1 .
?- fact(1,F).
F = 1 .
?- fact(10,F).
F = 3628800 .
```

If N = 0 in the first clause we do not need to consider the second clause because the second will fail → we CUT to prune unnecessary consideration of the second clause.

With or without the cut, the program produces the same solutions. Its intended meaning is intact.

Cut Operator: Examples (cont.)

- **Minimum of two numbers**:

  Without Cut('**!**'):

```
min(N1,N2,N1) :- N1 =< N2.
min(N1,N2,N2) :- N2 < N1.
```

  With Cut('**!**'):

```
min(N1,N2,N1) :- N1 =< N2, !.
min(_,N2,N2).
```

```
?- min(7,11,M).
M = 7.
?- min(11,7,M).
M = 7.
?- min(7,7,M).
M = 7.
```

Cut '!' and if-then-else Construct

- The use of the cut '!' in Prolog can serve a role similar to an if-then-else construct in other programming languages.

  statement :- condition, !, then_part.
  statement :- else_part.

- Here, the cut commits us to the then_part if the condition is true → we will never go to the second rule.
- However, if the condition comes back as false (i.e. no derivation is found to make it true) → move on to the second rule and try to find if it can be satisfied.
- Example:

```
compare_numbers(X, Y) :- X =:= Y, !,
write('Numbers are equal.'), nl.
compare_numbers(X, Y) :- X > Y, !,
write('First number is greater.'), nl.
compare_numbers(_, _) :- write('Second
number is greater.'), nl.
```

```
?- compare_numbers(7,7).
Numbers are equal.
true.
?- compare_numbers(7,11).
Second number is greater.
true.
?- compare_numbers(11,7).
First number is greater.
true.
```

Negation Operator: '\+'

- In Prolog, '\+' operator is used for negation. It's often referred to as "not" or "logical negation."
- The main purpose of '\+' is to express the negation of the goal. Success if the goal after "\+" fails, fail if the goal succeeds.
- Example:

```
even_number(X) :- 0 is X mod 2.
odd_number(X) :- \+ even_number(X).

?- odd_number(7).
true.

?- odd_number(8).
false.
```

```
w(X) :- q(X), \+ r(X).
q(a).
r(a).
q(e).

?- w(a).
false.
?- w(e).
true.
?- q(c).
false.
?- w(c).
false.
```

Negation Operator '\+' Examples

- Hotel is full if there are no vacant rooms. Room 13 is vacant, and room 113 is vacant.

```
hotelFull :- \+ vacantRoom(X).
vacantRoom(13).
vacantRoom(113).


?- hotelFull.
false.
```

Negation Operator '\+' Examples

X is at home if X is not out. Sue is out. John is Sue's husband.

```
home(X):- \+ out(X).
out(sue).
husband(john, sue).
```

?- home(john).
true.

?- home(X).
false.

home(john). checks whether "John" is at home. It succeeds because there is no fact stating that "John" is out. → '\+ out(john)' is true → home(john) is true.

home(X). This query checks whether there is anyone at home. It fails because the definition of 'home/1' explicitly checks if the argument is not out, and there are no other facts about individuals being at home.

Why this happens?

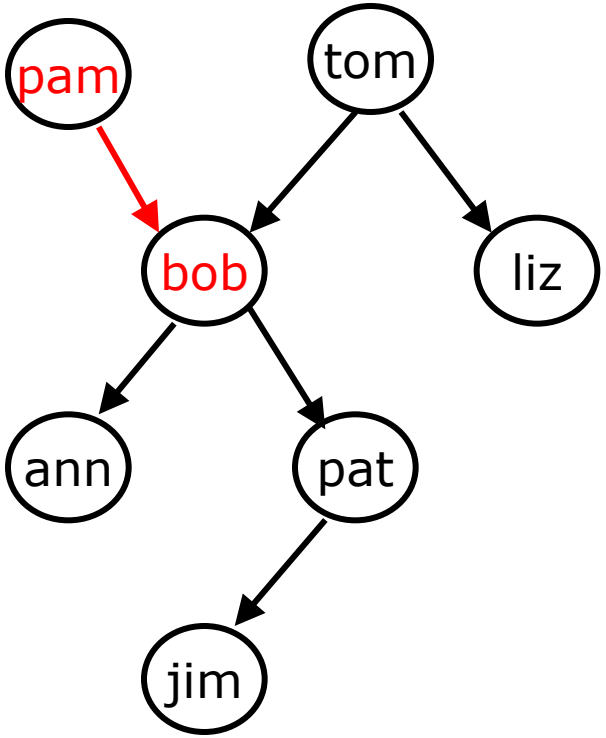
- Prolog assumes that if something is not explicitly defined in the database, it is false.
- **\+** works by checking whether out(X) fails → no binding for X is found
- **How to address this?**

```
person(john).
person(sue).
home(X) :- person(X),\+ out(X).
```

?- home(X).
X=john

Example: Family Tree - 1

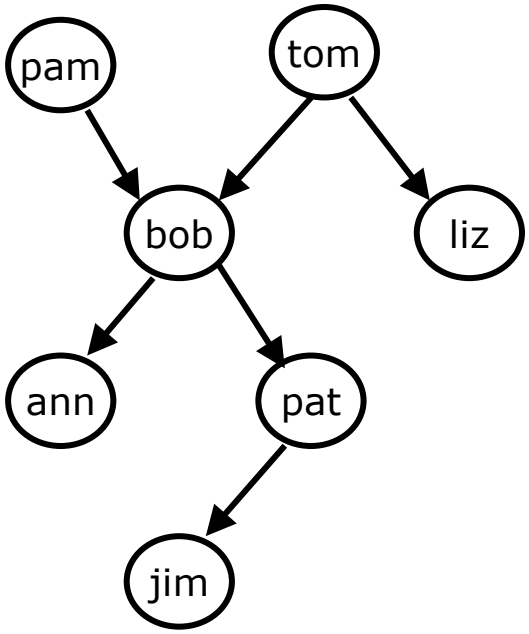
- Given a whole family tree



Defined by the Prolog program:

```
parent( pam, bob).  % Pam is a parent of Bob
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).
```

Example: Family Tree - 2



parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).

With the initial knowledge base (6 facts), the program can respond to the following simple questions

- **Is bob a parent of pat?**

```
?- parent(bob,pat).
true.
```

- **Is liz a parent of pat?**

```
?- parent(liz,pat).
false.
```

- **Who is jim's parent?**

```
?- parent(X,jim).
X = pat.
```

- **Who is bob's parent?**

```
?- parent(X,bob).
X = pam ;
X = tom.
```

- **Who is tom's parent?**

```
?- parent(X,tom).
false.
```

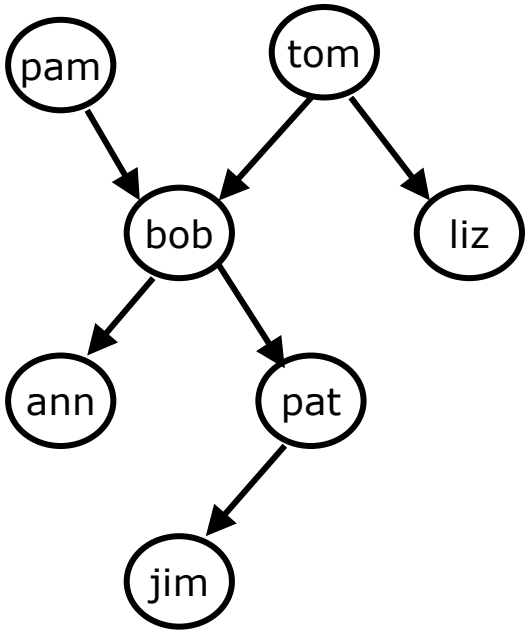
- **Who are Bob's children?**

```
?- parent(bob,X).
X = ann ;
X = pat.
```

- **Who are ann's children?**

```
?- parent(ann,X).
false.
```

Example: Family Tree - 3



parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).

- **Who is a parent of whom?**

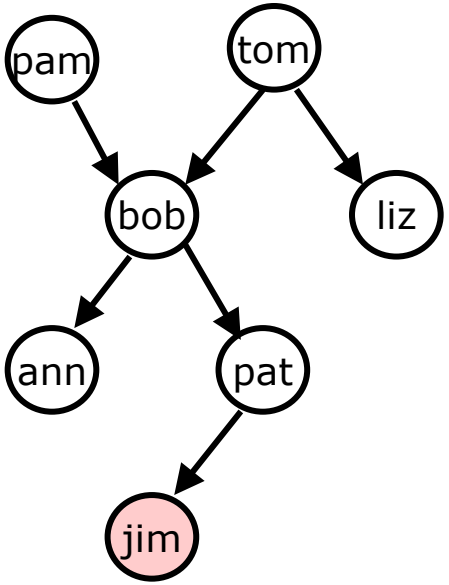
```
?- parent(X,Y).
X = pam,
Y = bob ;
X = tom,
Y = bob ;
X = tom,
Y = liz ;
X = bob,
Y = ann ;
X = bob,
Y = pat ;
X = pat,
Y = jim.
```

Note: the order of parent/child pairs generated by the program.

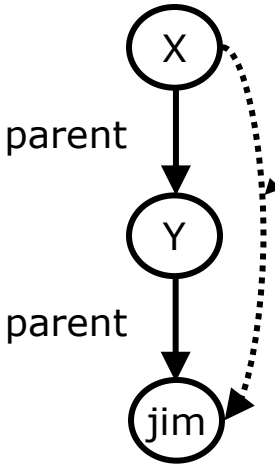
- **Do ann and pat have a common parent?**

```
?- parent(X,ann),parent(X,pat).
X = bob.
```

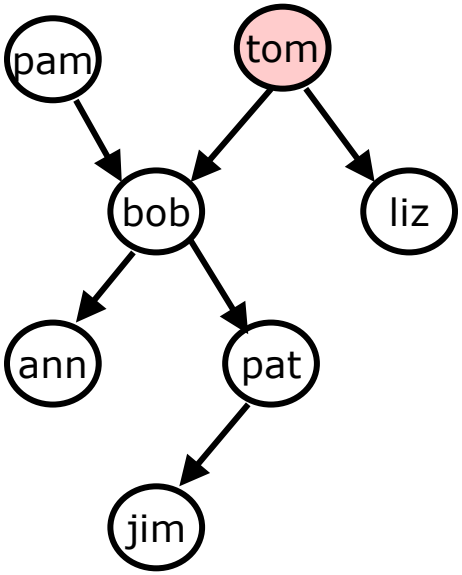
Example: Family Tree - 4



- **Who is a grandparent of jim?**




```
?- parent( Y, jim), parent( X, Y).
Y = pat,
X = bob.
```



- **Who are tom's grandchildren?**

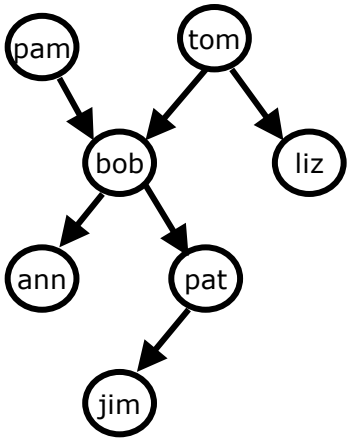
```
?- parent(tom,X),parent(X,Y).
X = bob,
Y = ann ;
X = bob,
Y = pat .
```

Example: Family Tree - 5

Family Tree KB:

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).

grandparent(X, Z) :-
parent(X, Y), parent(Y, Z).



- **Add rule for "grandparent"**
  - **Who is a grandparent of jim?**

```
?- grandparent(X,jim).
X = bob .
```

  - **Who is a grandparent of pat?**

```
?- grandparent(X,pat).
X = pam ;
X = tom .
```

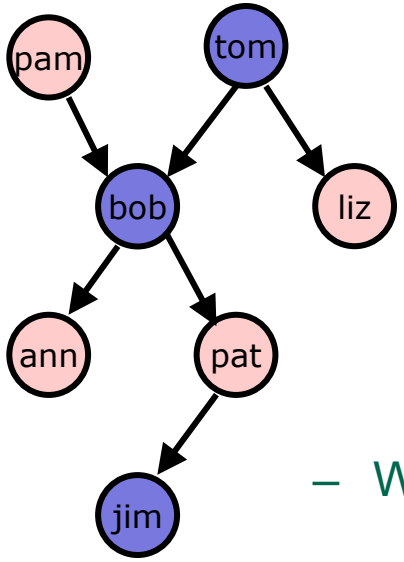
  - **Who are tom's grandchildren?**

```
?- grandparent(tom,X).
X = ann ;
X = pat .
```

Example: Family Tree - 6

- Add facts on gender:

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).



- Define the "offspring" relation:

offspring(Y, X) :- parent(X, Y).

– liz is tom's offspring?

```
?- offspring(liz,tom).
true.
```

– Who are bob's offsprings?

```
?- offspring(X,bob).
X = ann ;
X = pat.
```

– Who is the offspring of whom?

```
?- offspring(X,Y).
X = bob,
Y = pam ;
X = bob,
Y = tom ;
X = liz,
Y = tom ;
X = ann,
Y = bob ;
X = pat,
Y = bob ;
X = jim,
Y = pat.
```

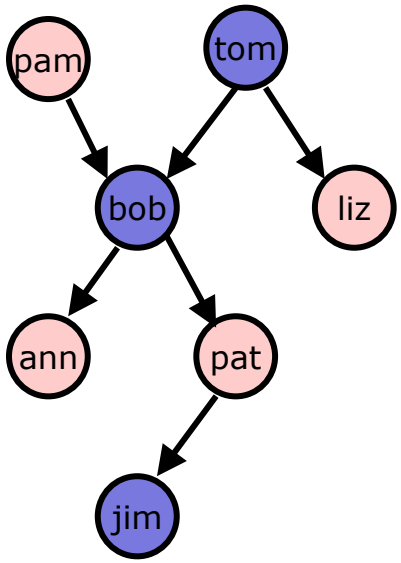
Example: Family Tree - 7

- Add the “mother” relation:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).

grandparent( X, Z) :-   parent( X, Y), parent( Y, Z).
offspring( Y, X) :- parent( X, Y).
mother( X, Y) :- parent( X, Y), female( X).
```




```
?- mother(pat,jim).
true.

?-  mother(tom,liz).
false.

?- mother(pam,pat).
false.

?- mother(X,bob).
X = pam .

?- mother(X,pat).
false.
```

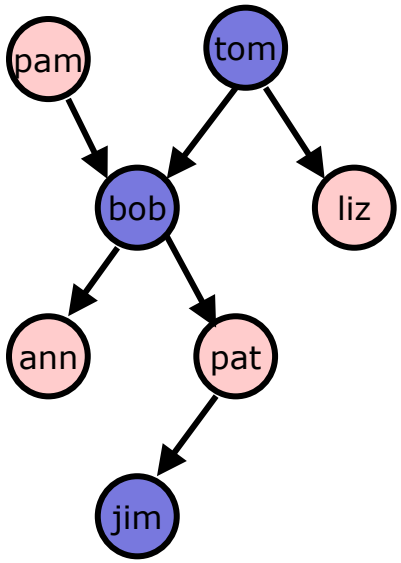
Example: Family Tree - 8

- Add the “sister” relation:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).

grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
offspring( Y, X) :- parent( X, Y).
mother( X, Y) :- parent( X, Y), female( X).
sister( X, Y) :- parent( Z, X),  parent( Z, Y), female(X).
```




```
?- sister(ann,pat).
true.

?- sister(bob,liz).
false.

?- sister(liz,bob).
true.

?- sister(pam,tom).
false.

?- sister(X,bob).
X = liz .

?- sister(ann,X).
X = ann ;
X = pat.

ann is sister of ann? 😳
```

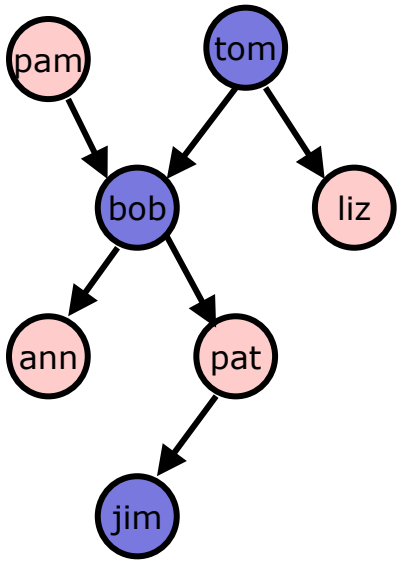
Need to correct the rule for sister

Example: Family Tree - 9

- Correct the “sister” relation:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).
```




```
?- sister(ann,X).
X = pat.

?- sister(pam,X).
false.

?- sister(X,Y).
X = liz,
Y = bob ;
X = ann,
Y = pat ;
X = pat,
Y = ann .
```

```
grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
offspring( Y, X) :- parent( X, Y).
mother( X, Y) :- parent( X, Y), female( X).
sister( X, Y) :- parent( Z, X),  parent( Z, Y), female(X), X \= Y.
```

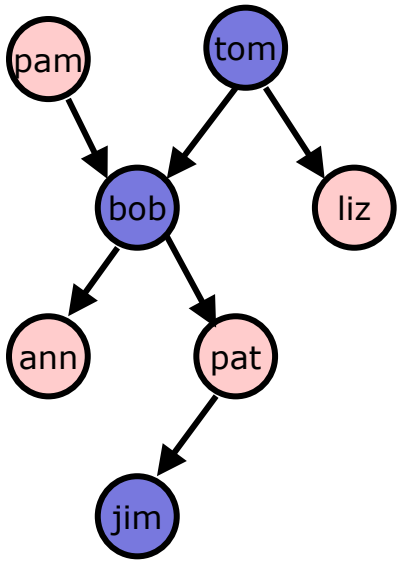
Example: Family Tree - 10

- Add the “has_a_child” relation:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).

grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
offspring( Y, X) :- parent( X, Y).
mother( X, Y) :- parent( X, Y), female( X).
sister( X, Y) :- parent( Z, X),  parent( Z, Y), female(X), X \= Y.
has_a_child(X) :- parent(X, Y).
```




```
?- has_a_child(pam).
true.

?- has_a_child(bob).
true .

?- has_a_child(ann).
false.

?- has_a_child(X).
X = pam ;
X = tom ;
X = tom ;
X = bob ;
X = bob ;
X = pat.
```

Duplicated tom and bob are returned!

Example: Family Tree - 11

- Revised has_a child(X):

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).

grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
offspring( Y, X) :- parent( X, Y).
mother( X, Y) :- parent( X, Y), female( X).
sister( X, Y) :- parent( Z, X),  parent( Z, Y), female(X), X \= Y.
has_a_child(X) :- setof(Parent, Child^(parent(Parent, Child)), Parents),
                  member(X, Parents).
```

Built-in predicate **setof/3: setof(Variable, Goal, Set).**

- **Variables** that are present in the Goal.
- The **Goal** or query for which we want to find solutions.
- **Set**: all the solutions to the Goal sorted and with duplicates removed.

'**Child^**': 'Child' is scoped only within the goal following the '^' symbol

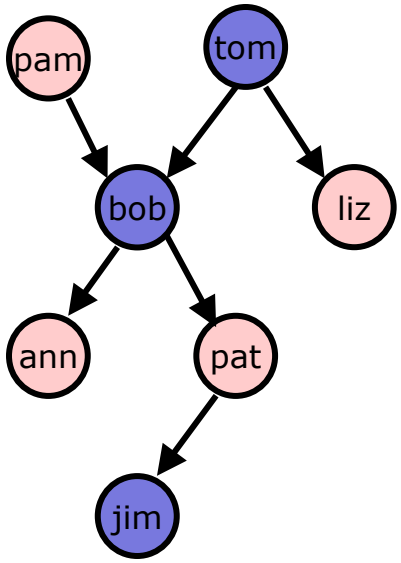
```
?- has_a_child(X).
X = bob ;
X = pam ;
X = pat ;
X = tom .
```

Example: Family Tree - 12

- Add the “predecessor” relation (a recursive rule):

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).
```




```
?- predecessor(pam,bob).
true .
?- predecessor(tom,jim).
true .
?- predecessor(pat,pam).
false.
?- predecessor(ann,jim).
false.
?- predecessor(X,bob).
X = pam ;
X = tom .
?- predecessor(pam,X).
X = bob ;
X = ann ;
X = pat ;
X = jim .
```

```
grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
offspring( Y, X) :- parent( X, Y).
mother( X, Y) :- parent( X, Y), female( X).
sister( X, Y) :- parent( Z, X),  parent( Z, Y), female(X), X \= Y.
has_a_child(X) :- setof(Parent, Child^(parent(Parent, Child)), Parents), member(X, Parents).
predecessor( X, Z):- parent( X, Z).
predecessor( X, Z):- parent( X, Y), predecessor( Y, Z).
```

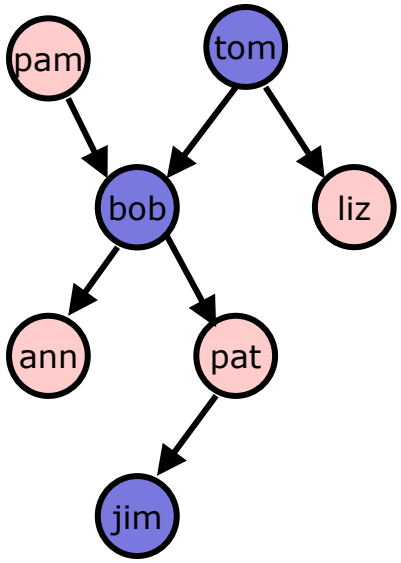
Example: Family Tree - 13

- Family tree knowledge base:

```
parent( pam, bob).
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
parent( pat, jim).

female(pam).
female(liz).
female(ann).
female(pat).
male(tom).
male(bob).
male(jim).

grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
offspring( Y, X) :- parent( X, Y).
mother( X, Y) :- parent( X, Y), female( X).
sister( X, Y) :- parent( Z, X),  parent( Z, Y), female(X), X \= Y.
has_a_child(X) :- setof(Parent, Child^(parent(Parent, Child)), Parents),
                  member(X, Parents).
predecessor( X, Z):- parent( X, Z).
predecessor( X, Z):- parent( X, Y), predecessor( Y, Z).
```




```
?- predecessor(X,Y).
X = pam,
Y = bob ;
X = tom,
Y = bob ;
X = tom,
Y = liz ;
X = bob,
Y = ann ;
X = bob,
Y = pat ;
X = pat,
Y = jim ;
X = pam,
Y = ann ;
X = pam,
Y = pat ;
X = pam,
Y = jim ;
X = tom,
Y = ann ;
X = tom,
Y = pat ;
X = tom,
Y = jim ;
X = bob,
Y = jim .
```

Software Testing: Basic Fuzzing

- Enhancing Security and Reliability

Introduction

- **Software Testing**
  - Purposes: Ensuring the functionality, reliability, and security of software
  - Key Objectives: Identifying defects, preventing issues, and improving overall quality
- **Importance of Detecting Vulnerabilities**
  - Software Vulnerabilities: weaknesses or flaws in a software system's design, implementation, or configuration, e.g., buffer overflows, injection flaws, etc.
  - Consequences: Security breaches, data loss, reputation damage
- **Fuzz Testing (Fuzzing)**
  - A dynamic software testing technique that involves providing invalid, unexpected, or random input to a program to uncover vulnerabilities
  - Role in Software Security: Complementary to traditional testing methods

What is Fuzz Testing?

- Identify software bugs, security vulnerabilities, and potential points of failure by subjecting the program to a variety of inputs
- Generate test inputs systematically or randomly, including malformed data, unexpected user inputs, or boundary cases, to explore the program's response to diverse and unpredictable scenarios.
- Fuzzing Techniques: Mutation-Based Fuzzing and Generation-Based Fuzzing
- Fuzz testing is often automated, allowing for the efficient and systematic execution of a large number of test cases.
- Monitor code coverage analysis to access the effectiveness of the testing process.

Historical Context of Fuzz Testing

- It was initially used to test Unix utilities in 1980s
- Gained prominence in the 1990s with focus on network protocols and file formats.
- Security researchers played a crucial role in popularizing fuzz testing as a powerful technique for finding security flaws.
- An integral part of security practices, especially in identifying vulnerabilities in complex software; an adoption in both open-source and commercial software development.
- Expanded beyond its initial use in operating systems and network protocols; embraced by industries such as web applications, browsers, IoT devices, and industrial control systems.
- Advancements in mutation-based and generation-based fuzzing techniques; integration of artificial intelligence and machine learning for more intelligent test case generation.
- Influence secure coding practices and the development of more resilient software; an integration into continuous integration and continuous deployment (CI/CD) pipelines.

Basic Fuzzing Concepts and Components

- **Test Input Generation: Random vs. Smart Fuzzing**
  - Random Fuzzing: generating inputs randomly without specific patterns or constraints → provide broad coverage of input space but may miss specific scenarios.
  - Smart Fuzzing: involve more targeted and controlled input generation; follow predefined specifications or models to explore specific aspects of the program's behavior.
- **Mutation-Based Fuzzing**
  - Modify existing valid inputs to create variations, e.g., bit flips, insertions, deletions, or permutations.
  - Systematically alters input data to explore different execution paths → identify vulnerabilities related to input processing, parsing, and validation.
- **Generation-Based Fuzzing**
  - Create entirely new inputs based on predefined specifications or models to allow for more controlled and targeted exploration of input space.
  - Define rules or patterns for generating input data which is useful for testing specific features or scenarios in the program.

Basic Fuzzing Concepts and Components (cont.)

- **Test Execution: Automation in Fuzzing**
  - Fuzz testing is often automated for efficient and systematic execution of a large number of test cases → ensures comprehensive coverage across different parts of the software.
- **Code Coverage Analysis**
  - Monitor code coverage to track which parts of the program are exercised during testing → evaluate the thoroughness of the testing process.
- **Dynamic Analysis of Behavior**
  - Dynamically analyze the behavior of the program in response to diverse inputs.
  - Unexpected behaviors, crashes, or security vulnerabilities are identified during the testing process.
- **Feedback Loop**
  - The results of fuzz testing, identified vulnerabilities or unexpected behaviors, create a feedback loop for developers to improve the software and address detected issues.
- **Continuous Fuzzing Integration:** integration into development lifecycle

Fuzz Testing Use Cases

- **Microsoft Security Response Center:**
  - Apply fuzz testing extensively in its Security Response Center to identify vulnerabilities.
- **Google's OSS-Fuzz Project:**
  - An open-source initiative that applies fuzz testing to various open-source software projects.
- **Mozilla Firefox:**
  - Fuzz testing has been instrumental in identifying and fixing security vulnerabilities in Firefox.
- **ClamAV Antivirus Software:**
  - ClamAV, an open-source antivirus software, employs fuzz testing to ensure robust detection of malware and vulnerabilities.

Fuzz Testing Impact on Software Security

- **Early Detection of Vulnerabilities:** Identify vulnerabilities in the development phase allows for timely remediation.
- **Preventing Exploitation:** Proactively identifying and fixing vulnerabilities before software deployment
- **Enhancing Resilience:** regular testing helps uncover potential weaknesses.
- **Open Source Software Security:** Identify and address vulnerabilities in open-source software benefits the entire community of users.
- **Comprehensive Security Testing**: Complement other security testing methodologies, providing a comprehensive approach to identifying vulnerabilities.
- **Regulatory Compliance:** Align with regulatory requirements for robust security practices and meet compliance standards.

Fuzz Testing Tools and Frameworks

- **AFL (American Fuzzy Lop):** https://github.com/google/AFL
  - A widely used open-source fuzz testing tool known for its effectiveness in discovering vulnerabilities.
  - It employs genetic algorithms to mutate test inputs systematically.
- **LibFuzzer:** https://llvm.org/docs/LibFuzzer.html
  - Part of the LLVM compiler infrastructure and is designed for in-process fuzzing.
  - It integrates seamlessly with codebases and is commonly used for testing C and C++ projects.
- **honggfuzz:** https://github.com/google/honggfuzz
  - A fast and customizable fuzz testing tool that supports various instrumentation methods.
  - It is suitable for testing applications written in C, C++, and other languages.
- **AddressSanitizer (ASan):** https://github.com/google/sanitizers/wiki/AddressSanitizer
  - A standalone fuzzing tool but is often used in conjunction with other tools.
  - It detects memory-related issues and can be integrated into the LLVM compiler.

Fuzz Testing Tools and Frameworks (cont.)

- **WinAFL:** https://github.com/googleprojectzero/winafl
  - An AFL variant designed for fuzz testing Windows applications.
  - It extends AFL's capabilities to the Windows platform, facilitating efficient testing.
- **OSS-Fuzz:** https://github.com/google/oss-fuzz
  - An open-source project by Google that applies continuous fuzz testing to various open-source software projects.
  - It leverages the infrastructure provided by Google to identify and fix vulnerabilities.

Choosing the Right Tool for Your Needs

Choosing the right fuzz testing tool involves a thoughtful assessment of factors. By aligning the tool with specific project needs, organizations can enhance their fuzz testing practices and improve overall software security.

- **Consideration Factors:** Language Support, Instrumentation, Integration, Platform Compatibility.
- **Customization and Configuration:** tailoring fuzz testing to the specific needs and characteristics of your software.
- **Community Support and Documentation:** Comprehensive documentation and an active community contribute to the tool's reliability and effectiveness.
- **Automation and CI/CD Integration:** supporting automation and seamless integration into continuous integration and deployment pipelines.
- **Security Coverage and Feedback:** capabilities in providing security coverage and meaningful feedback.
- **Scalability:** scalability to handle large codebases and complex applications.

Future Trends in Fuzz Testing

- **Evolving Landscape of Fuzz Testing**
  - **Diversity in Fuzzing Techniques:** Continued exploration of advanced fuzzing techniques, including coverage-guided, mutation-based, and generation-based fuzzing.
  - **Machine Learning Integration:** enhance test case generation and improve the efficiency of fuzz testing.
  - **Smart Fuzzing Platforms:** intelligent fuzzing platforms that adapt and prioritize test cases based on code coverage and potential security impact.
- **Integration with DevOps and CI/CD**
  - **Shift-Left Security:** an integral part of the early stages of software development, ensuring security is addressed from the beginning.
  - **Automated Pipelines:** Seamless integration of fuzz testing into CI/CD pipelines for automated and continuous security assessments.
  - **Collaboration with DevSecOps:** Strengthening collaboration between development, security, and operations teams to embed security practices throughout the software development lifecycle.

Future Trends in Fuzz Testing (cont.)

- **Continuous Improvement and Feedback Loop**
  - **Feedback-Driven Fuzzing:** Insights from real-world exploits and vulnerabilities are used to refine and enhance fuzz testing strategies.
  - **Community Collaboration:** Increased collaboration within the security community to share fuzzing techniques, tools, and findings, fostering a collective approach to software security.
- **Advanced Target Scoping and Prioritization**
  - **Intelligent Target Selection:** Utilization of intelligent algorithms for target selection, focusing on high-risk components and potential attack vectors.
  - **Risk-Based Prioritization:** ensure that critical vulnerabilities are identified and addressed promptly.

Fuzz testing continues to evolve as a fundamental practice in uncovering software vulnerabilities and improving overall security posture

Final Exam Preparation

Overview

- Exams are tests of your understanding of the content.
- Attend class, read textbooks, review class notes, complete quizzes and homework ➔ fully understand the contents
- This is a closed-book and closed-notes exam.
- The exam format: MCQs, essays, etc.
- **90 minutes**; 47 questions and 2 bonus questions with points ranging from 1 to 10
- The exam covers topics in programming language-based security, functional programming, logic programming, and Fuzz Testing.