Language Based Security

Kumar Neeraj Verma

TU München

Winter Semester 2008

Organization

Lectures:	Wednesday,	10:15 - 11:45,	room MI 02.07.014
	Starting 15.10.08		
Tutorials:	Thursday,	10:00 - 11:45,	room MI 02.07.014
	Starting 30.10.08		
Exam:	Oral		

Planned contents

- Buffer overflow attacks
  $\longrightarrow$ Prevention using program analysis
- Security issues in Java
- Type systems for safety
- Bytecode verification and proof carrying code
- Access control
- Information flow analysis

Computer Security

Some goals

- Confidentiality of information
- Authenticity
- Preventing other improper behavior like not paying for services
- Ensuring availability of services
- Preventing damage of information

Challenges

- Increasing complexity of software; frequent updates
- Untrusted programs
- Computer systems are not isolated
- Numerous possibilities for attacks: webpages with executables, emails, cookies, . . .
- Financial cost of an insecurity could be huge
- Traditional OS kernel based security not sufficient to prevent attacks like viruses in emails.

  $\longrightarrow$ Use lanuguage based security: based on program analysis and program rewriting.

The Morris Worm, 1988

- One of the first known internet worms.
- Among others it exploited a buffer overflow vulnerability in fingerd.
- A worm at an infected host copied itself to other hosts by exploiting vulnerabilities. The number of copies running at a host slowed it down to the point of being unusable.
- An estimated 6000 machines (10 % of hosts at that time) were infected.
- Cost of the damage estimated to be $10M-100M.

New buffer overflow vulnerabilities still continue to be found.

The MS-SQL Slammer worm, 2003

- Exploited a buffer overflow vulnerability in Micorsoft SQL server announced in 2002.
- Affected more than 75000 hosts, most of them within the first 10 minutes.

The Code Red worm, 2001

- Exploited a buffer overflow vulnerability in Microsoft's IIS web server.

Buffer overflows

- The C language allows access to arbitrary memory locations through improper use of pointers.
- This leads to a typical programming error of accessing a buffer (array) beyond the space allocated for it.
- Typically exploited by stack smashing attacks involving overflowing buffers on the stack to overwrite the return address.
- Data extracted from CERT advisories show that buffer overflows are responsible for nearly half of todays vulnerabilities.

Pointers and arrays in C

For any variable we can obtain the corresponding memory location using the `&` operator. The `*` operator gives the value stored at a memory location.

```
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
```

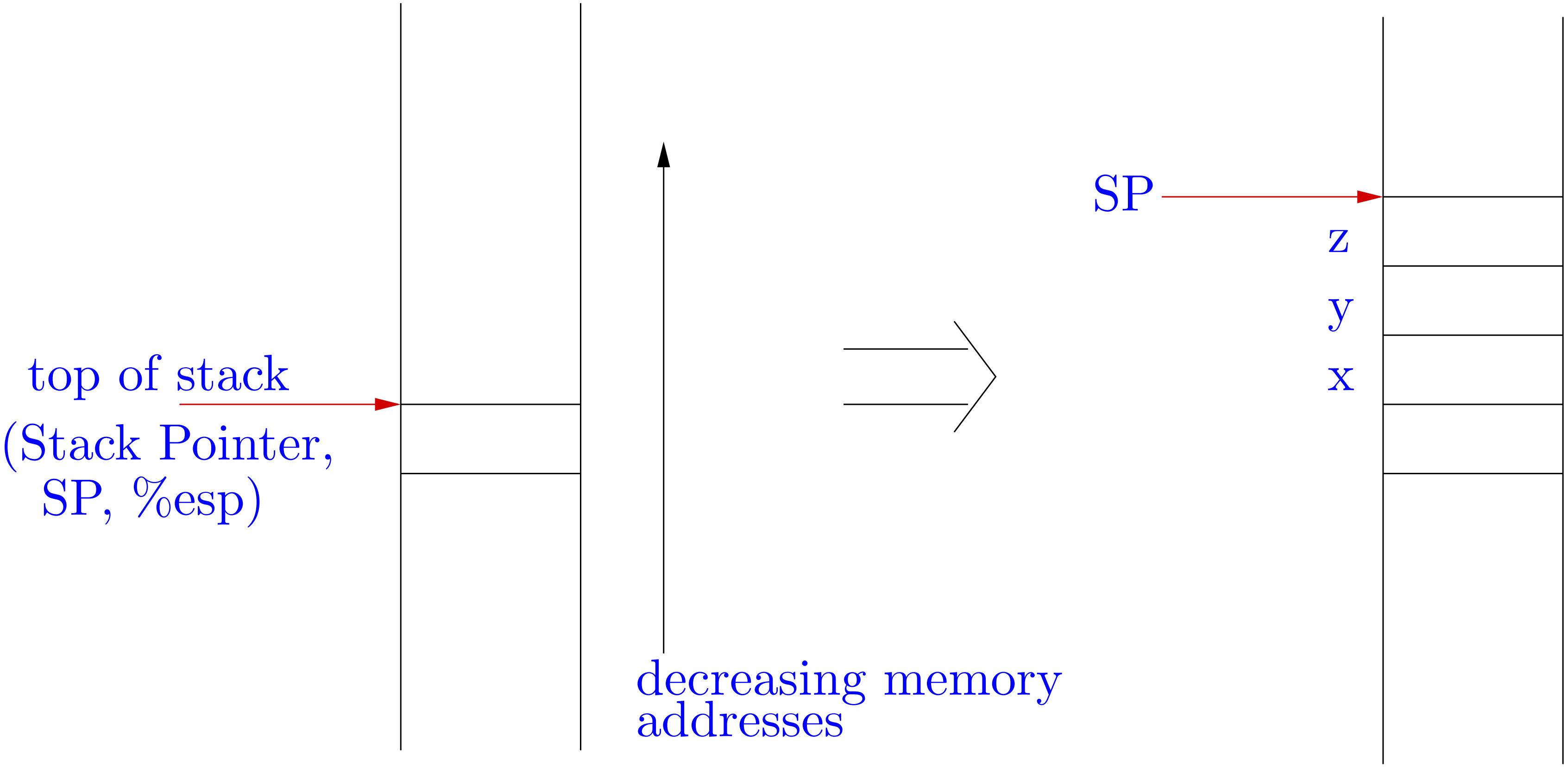
C allows access to arbitrary memory locations through pointers.

Here we need to know that `x` and `y` are allocated space on consecutive locations.

The declaration

```
int x,y,z;
```

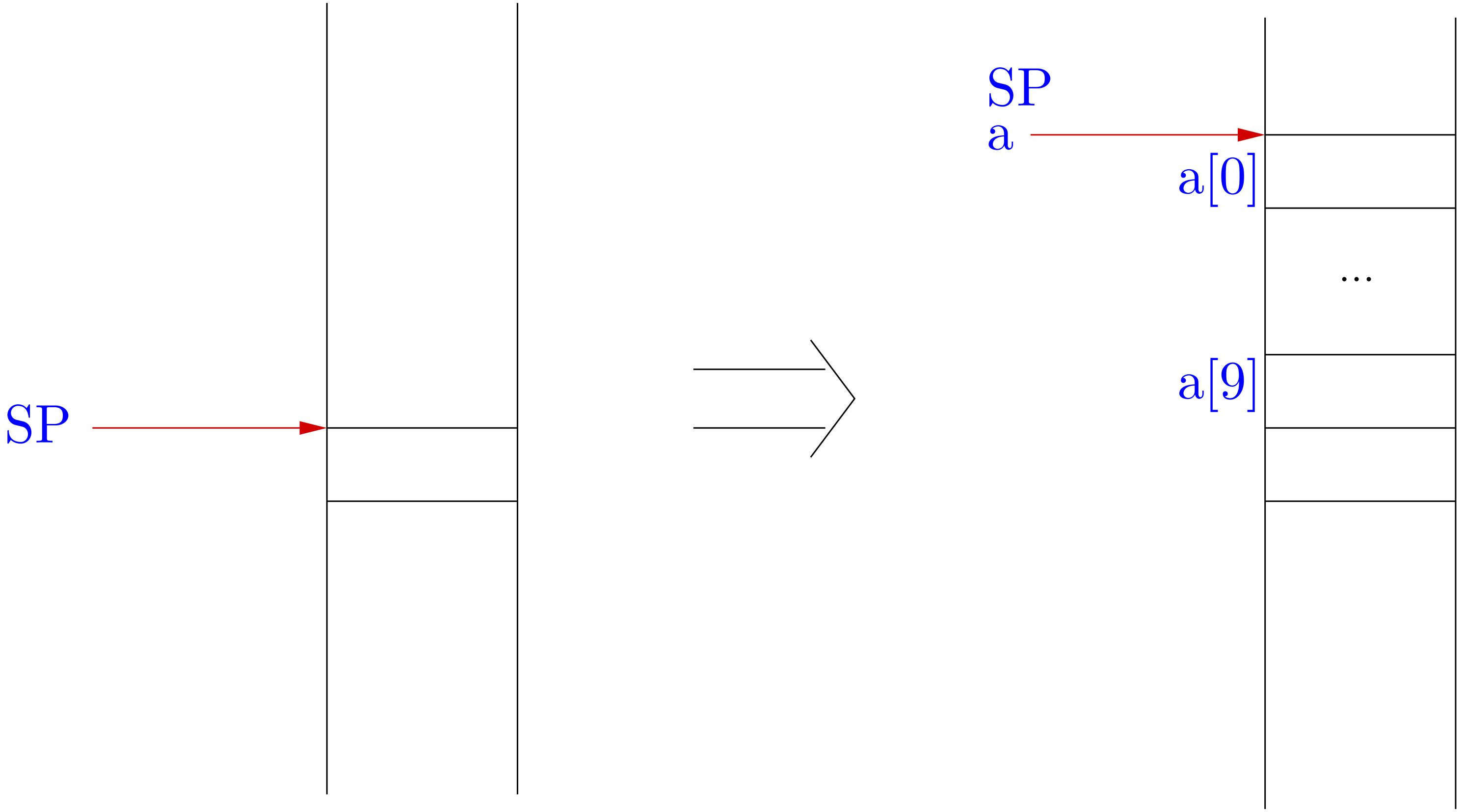
leads to allocation of space on the stack as follows.



Allocating space for arrays on the stack:

```
int a[10];
```

`a` is also the address where `a[0]` is stored. `a[5]=10` is same as `*(a+5)=10`.



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);

  /* Note: code may require adjustment to
       machine and compiler */
}
```

```
x = 10
x = 20
```

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

No checks enforced by the C language!

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)
```

Exceptions may then be caught and some other action taken.

```
public class Array2 {
  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++)
      try { a[i]=20; } catch (Exception e) { }
    System.out.println (”x=” + x);
  }
}
x=10
x=10
```

Function calls and stack frames

- Each time a function is called, space must be allocated for the local variables of the function. This region of the stack is called the stack frame for this function call.

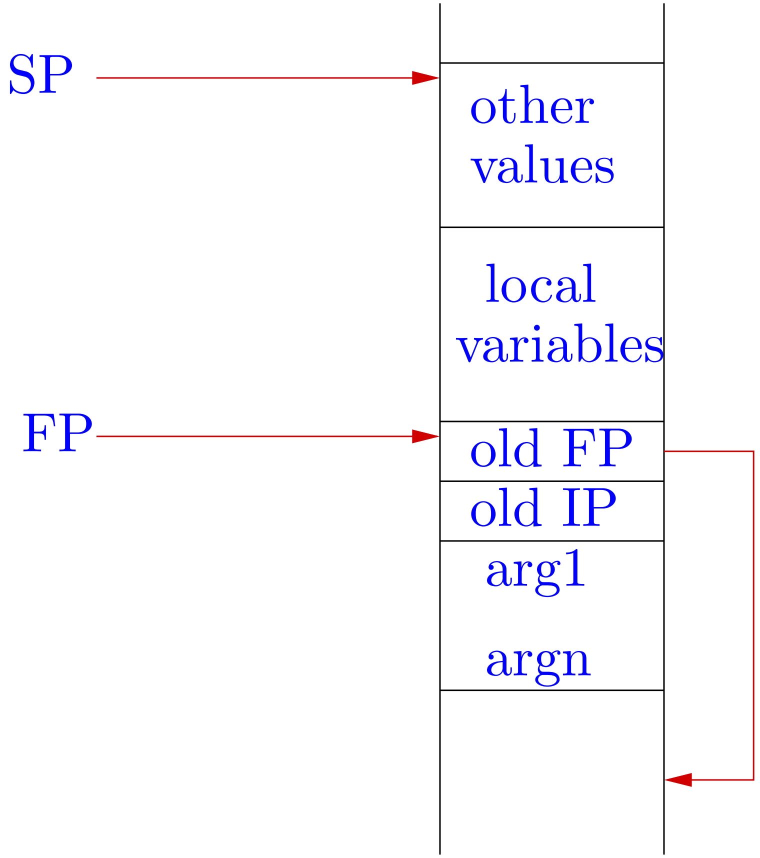
  ⇒ Use a Frame Pointer (FP, %ebp) to indicate the location of the current frame. This allows easy access to the local variables at runtime.

- On return from a function call, execution must continue from the next instruction after the function call.

  ⇒ Store the old instruction pointer (PC) in the stack frame.

- On return from a function, the current stack frame is popped out and execution continues with the previous stack frame.

  ⇒ Store the old FP on the stack.



A simple example of function call.

```
/* function.c */
void f (int x, int y) {
  int a,b,c;
}

int main () {
  f (10, 20);
}
```

Let's see the compiled code produced.

```
$ gdb function
```

. . .

The caller:

```
(gdb) disassemble main
 ...
0x804832f <main+19>: push   $0x14
0x8048331 <main+21>: push   $0xa
0x8048333 <main+23>: call   0x8048314 <f>
 ...
```

The arguments are pushed on to the stack and the function is called.

The caller:

```
(gdb) disassemble main
 ...
0x804832f <main+19>: push   $0x14
0x8048331 <main+21>: push   $0xa
0x8048333 <main+23>: call   0x8048314 <f>
 ...
```

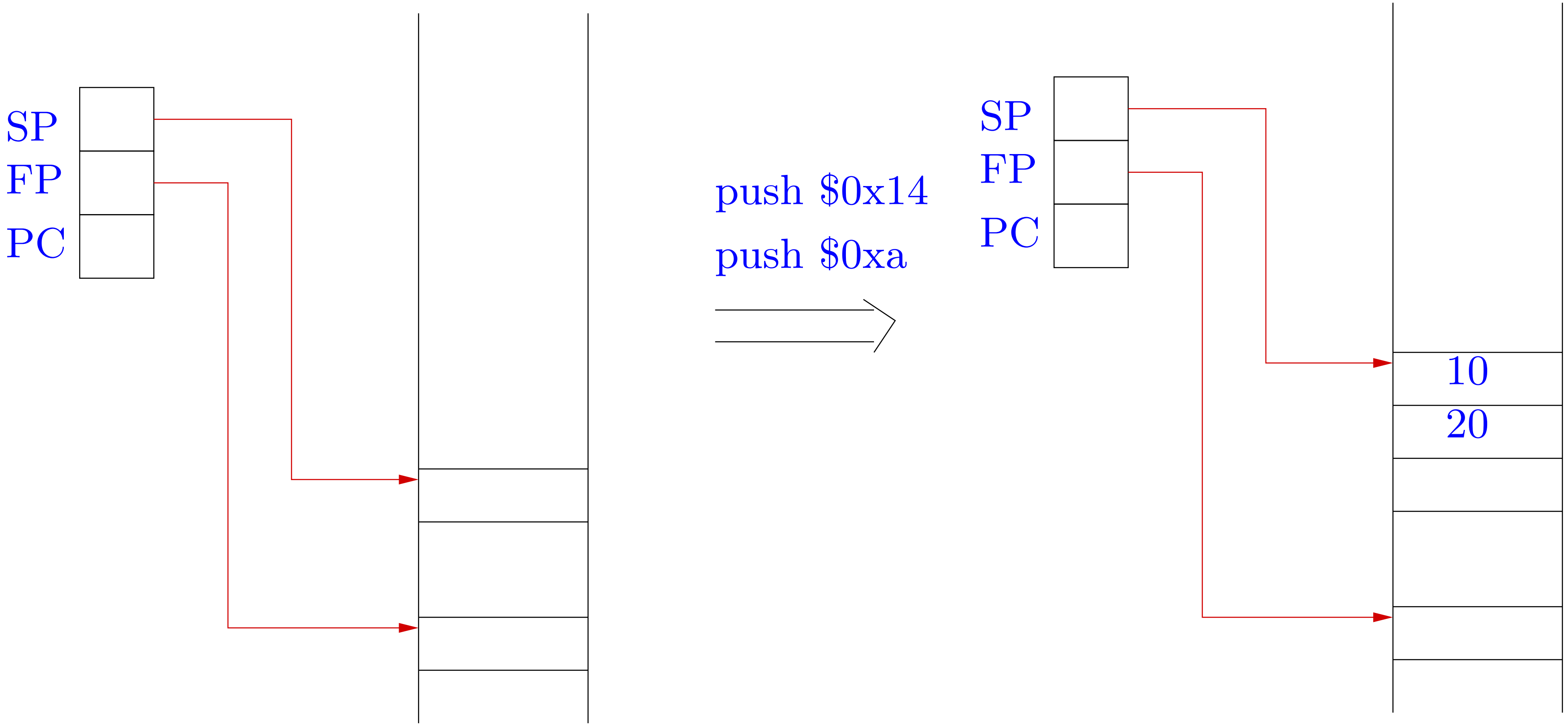
The arguments are pushed on to the stack and the function is called.

And the callee. . .

```
0x8048314 <f>:      push  %ebp
0x8048315 <f+1>:    mov   %esp,%ebp
0x8048317 <f+3>:    sub   $0xc,%esp
0x804831a <f+6>:    leave
0x804831b <f+7>:    ret
```

- Save old FP, update FP
- Allocate space for local variables, do computations
- Restore FP, pop saved FP from stack
- Return (restore PC, pop saved PC from stack)

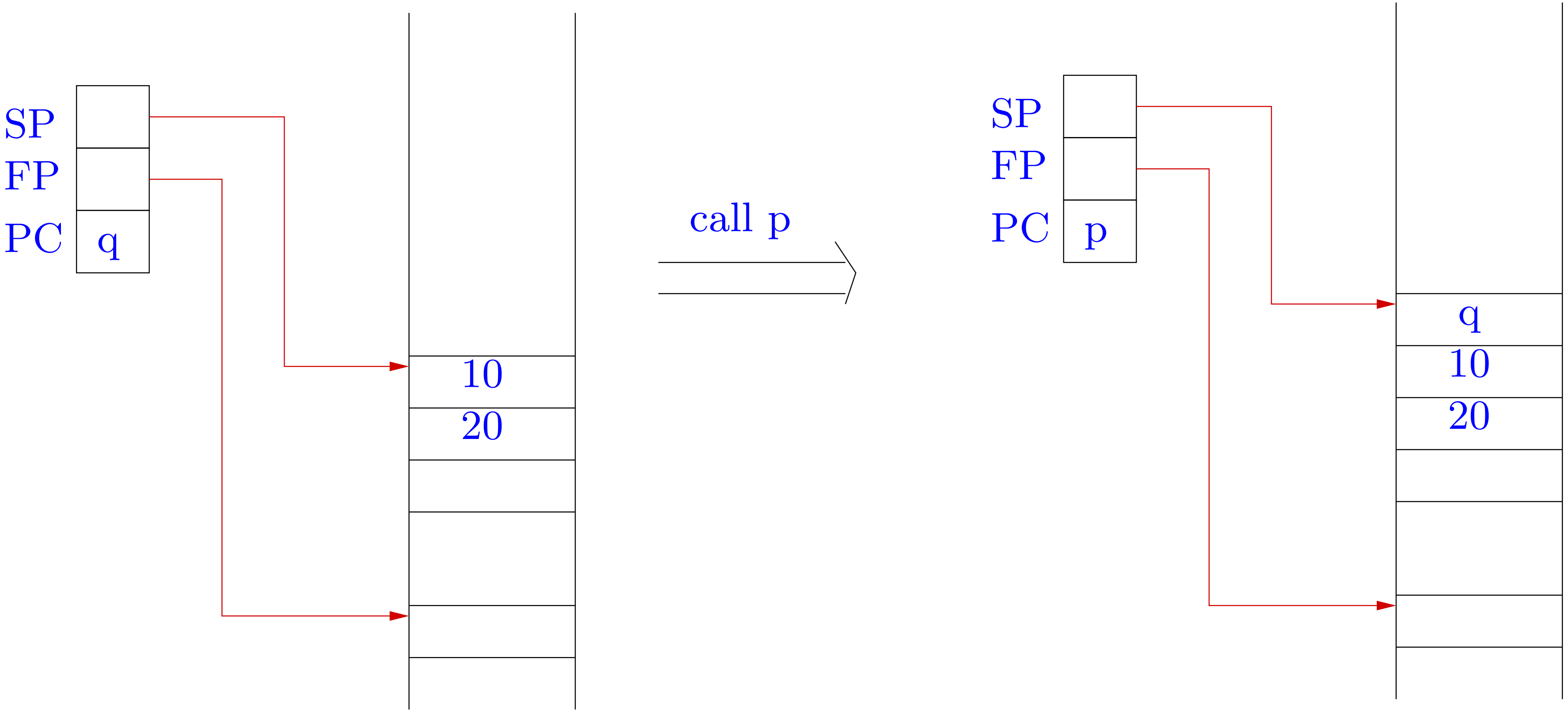
At run time: pushing arguments



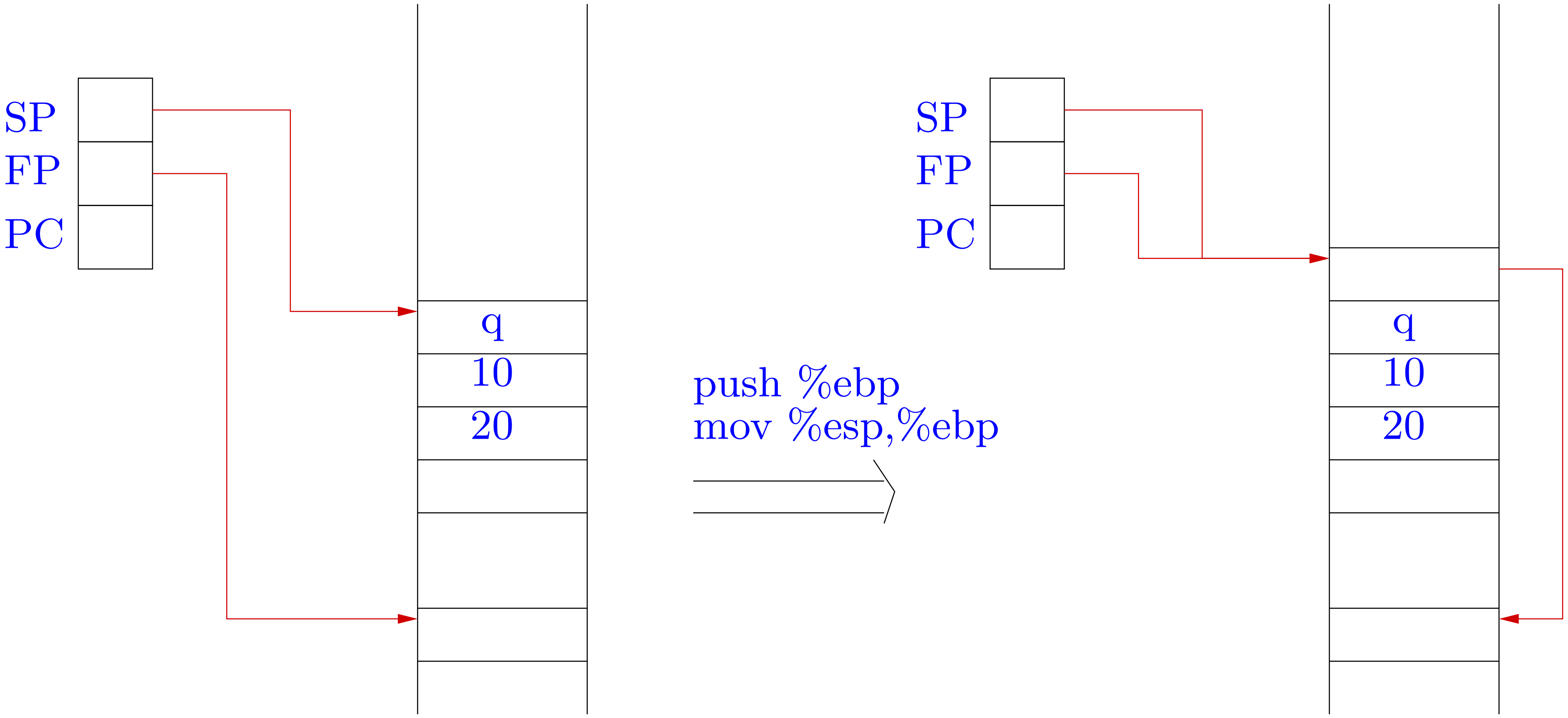
push $0x14

push $0xa

Calling function: saving PC and updating PC

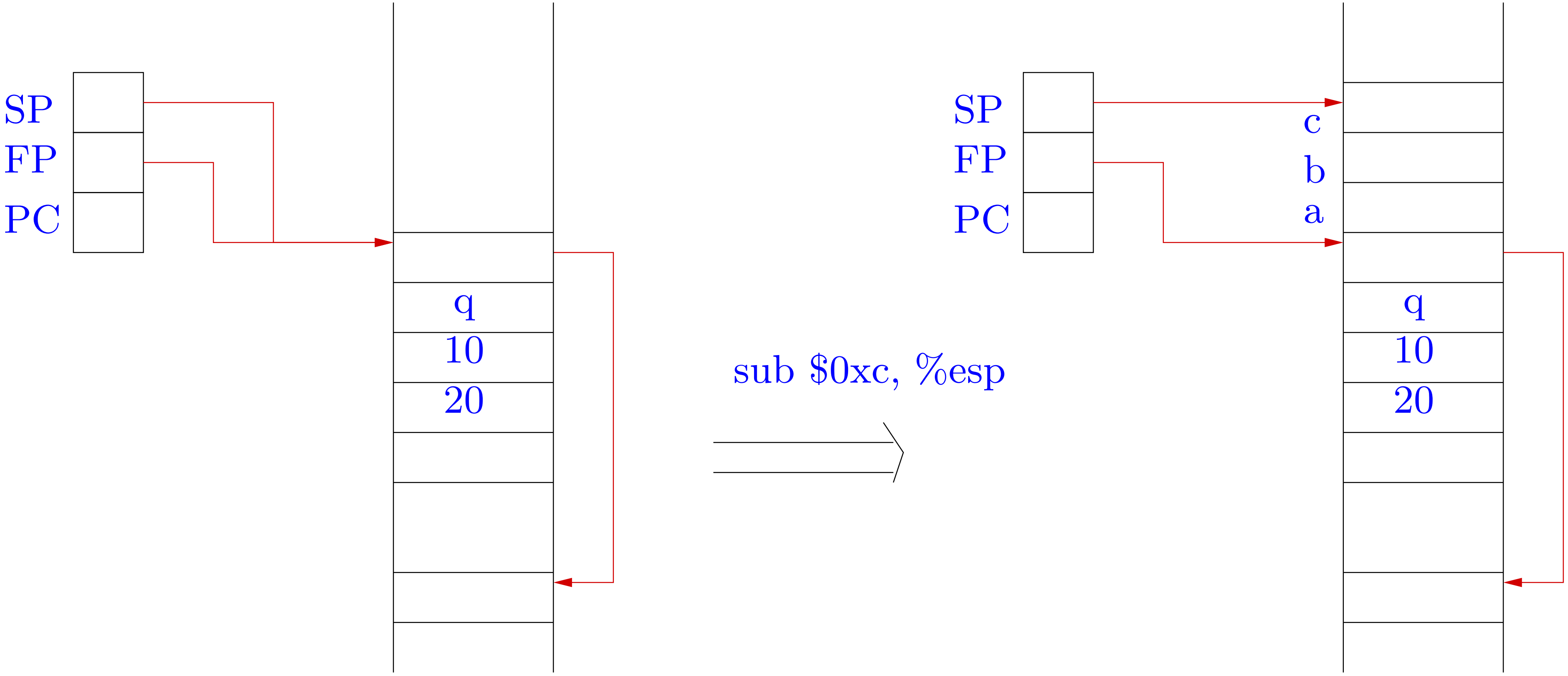


Inside callee: saving FP and updating FP

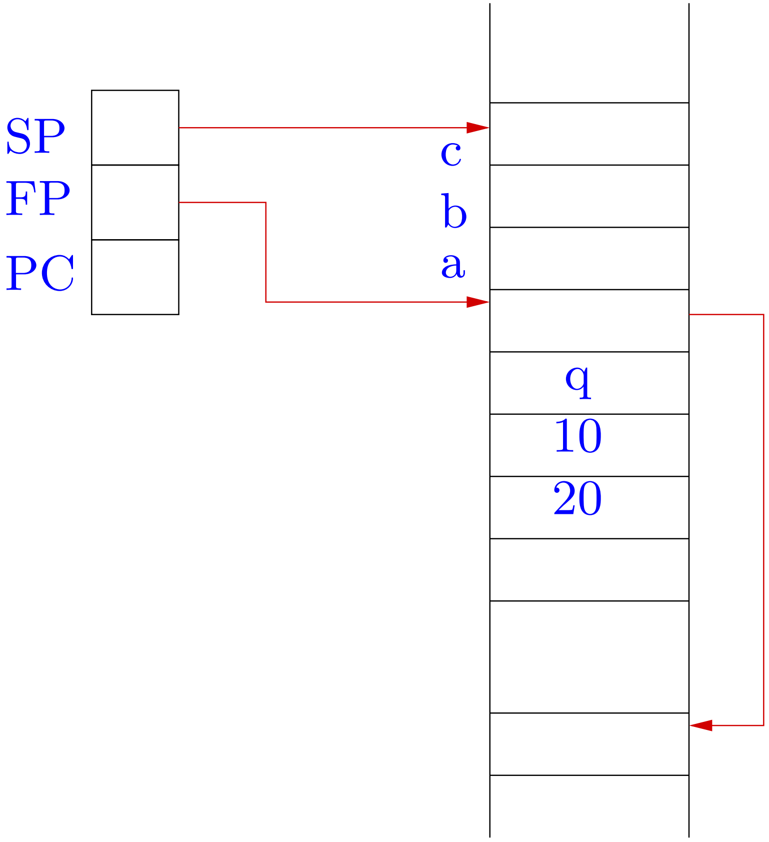


```
push %ebp
mov %esp,%ebp
```

Allocating space for local variables



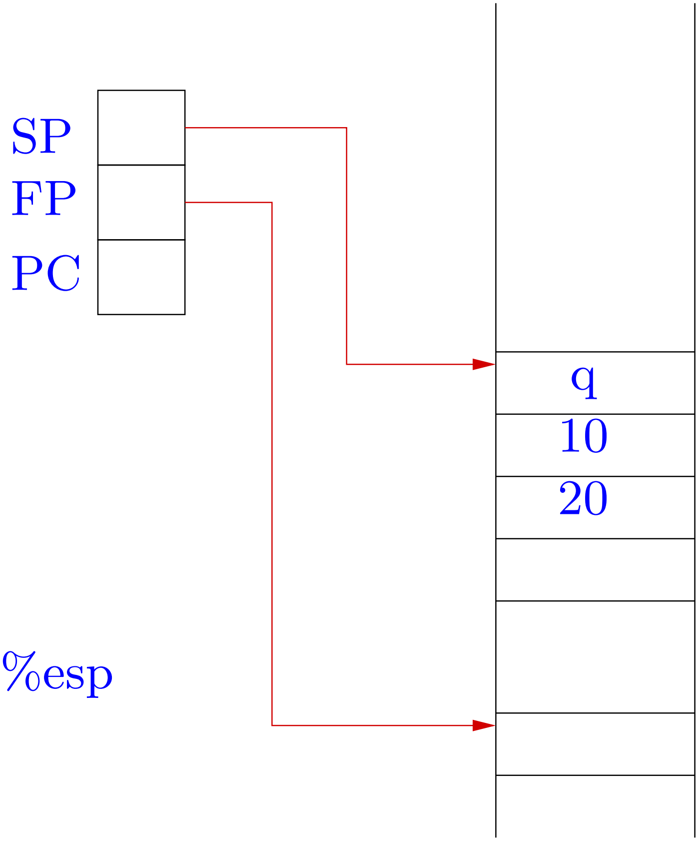
End of callee: restoring FP and popping saved FP



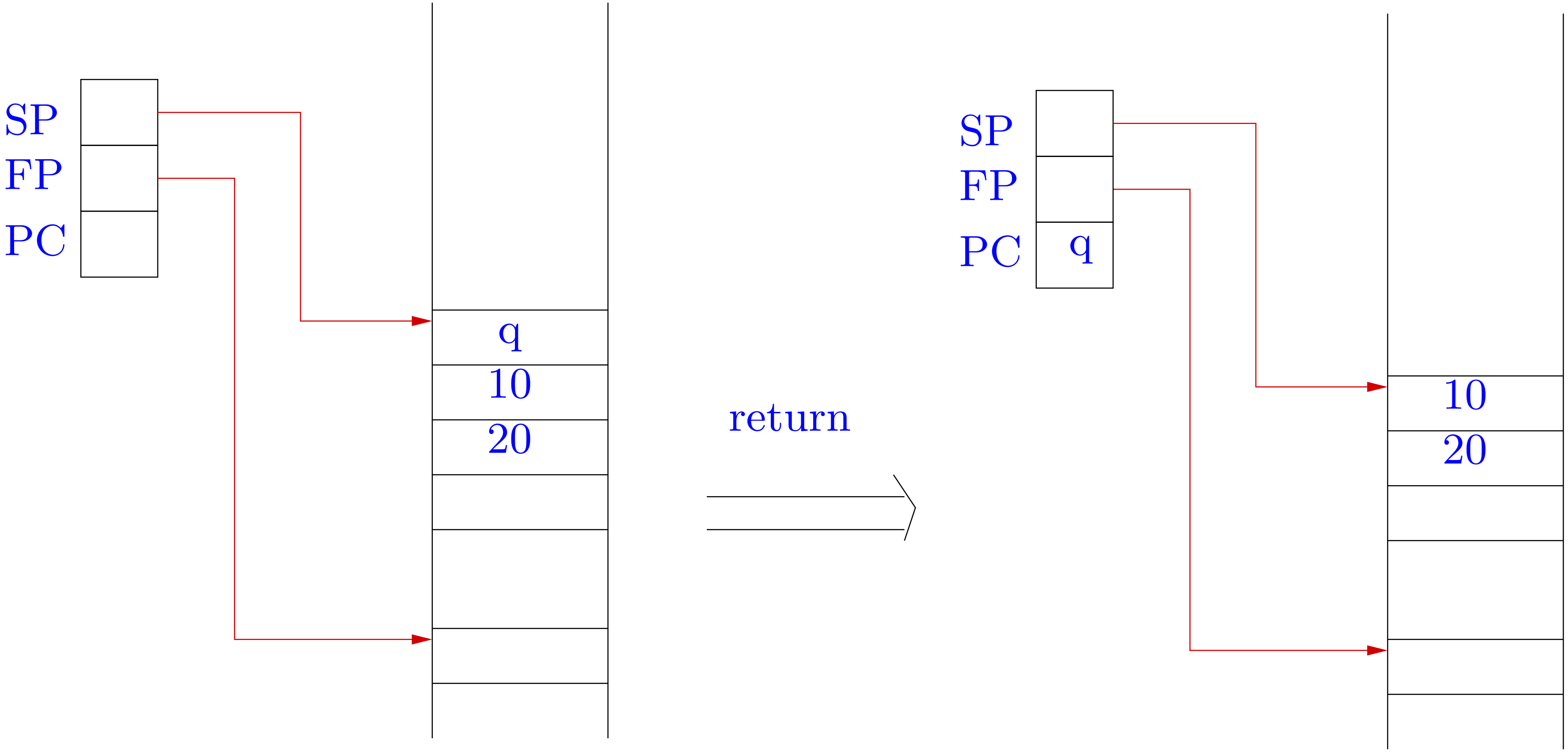
leave ⟹

equivalently:

```
mov %ebp, %esp
pop %ebp
```



Returning: restoring PC and popping saved PC



The return address is stored on the stack.

⇒ it can also be overwritten to point to arbitrary code!!!

```
void f () {
int a[10];
a[15] += 7;
}
```

```
main () {
   int x = 10;
   f ();
   x = 20;
   printf (”x=%d!\n”,x);
}
```

Output:

```
x=10!
```

We have skipped the instruction x = 20; !

- Where is the return address stored (a[15])?
- What should be the new return address (increment by 7)?

Organization of the stack: a[0], . . . , a[9], old FP, old PC

Hence the return address is at the location a[11].

Organization of the stack: a[0], . . . , a[9], old FP, old PC

Hence the return address is at the location a[11].

Not always: compiler optimizations may create blank spaces around array a.

⇒ Look at the compiled code.

Organization of the stack: a[0], . . . , a[9], old FP, old PC

Hence the return address is at the location a[11].

Not always: compiler optimizations may create blank spaces around array a.

⇒ Look at the compiled code.

```
0x8048344 <f>:         push   %ebp
0x8048345 <f+1>:       mov    %esp,%ebp
0x8048347 <f+3>:       sub    $0x38,%esp
 ...
```

Space allocated after old FP is 0x38 = 56 = 4*14 bytes.

Hence return address is at address a[15]

```
 ...
0x8048369 <main+23>: call    0x8048344 <f>
0x804836e <main+28>: movl   $0x14,0 xfffffffc (%ebp)
0x8048375 <main+35>: sub     $0x8,%esp
 ...
```

Instruction x = 20; requires 35 - 28 = 7 bytes.

Hence we put a[15] +=7 in the function f in order to skip execution of this instruction.

⇒ Besides modifying data, we may cause arbitrary code to be executed!

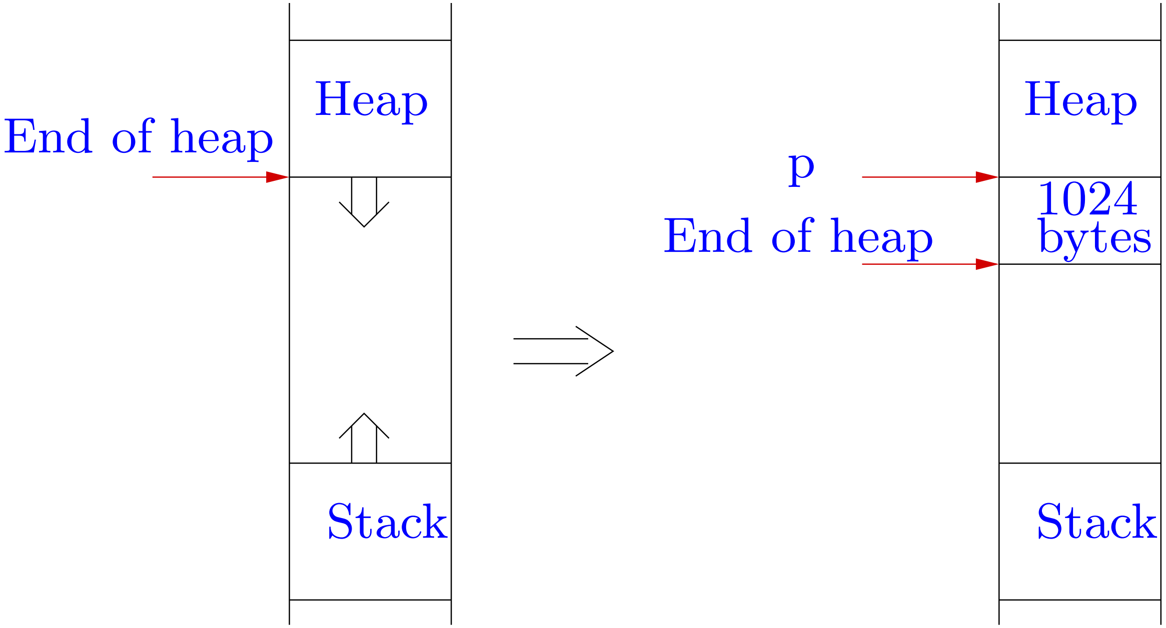
Weaknesses can be exploited by users by supplying appropriate inputs.

```
int main (int argc, char *argv[]) {
  char s[1024];
  strcpy(s,argv[1]);
   ...
}
```

- An appropriate input is given to overwrite the return address,
- At the minimum, the program may abort abruptly.
- An ingenious attacker may get some desired code to be executed (shellcode) by providing it as a part of the input string!

Heap based overflows: buffer overflows in the heap instead of the stack.

char *p = (char *) malloc (1024);



Instead of overwriting return addresses, an attacker may overwrite important variables.

Further errors arise because of improper use of string library functions.

In C, the end of a string is indicated by the null character.

The statement

```
strcpy (s,t);
```

will keep copying characters starting from t till a null character is found, irrespective of space allocated for s and t.

```
i = strlen (s);
```

tries to find the first null charachter beyond s.

Some techniques for preventing buffer overflow attacks.

- Careful programming: e.g. use strncpy instead of strcpy.

- Make the stack region non-executable: however some applications make use of an executable stack.

- Compiler tools: save the return address at a safe place (data region).

- Run time checks: use a preloaded library which provides safer versions of standard unsafe functions.

Detecting buffer overflow vulnerabilities

- Static program analysis: automated analysis of programs without running them.
- an exact analysis of buffer overflow vulnerabilities is theoretically impossible.

$\Longrightarrow$ do approximate analysis:

  - we fail to detect some vulnerabilities: unsafe approximation.
  - or we declare certain good programs as vulnerable: safe approximation (our approach).
  - or both.

- tradeoff between efficiency of analysis and precision of analysis.

Use of integer analysis

Most vulnerabilities are caused due to improper string manipulation.

Modify the program to include

- integer variables representing lengths of strings, overlaps between strings, etc.
- safety conditions before all string manipulation instructions.

Use well-known integer analysis algorithms to verify the safety conditions.

$\Longrightarrow$ we reduce string analysis problem to the simpler integer analysis problem.

Analyse instrumented C code

Dor, Rodeh and Sagiv

Original C code

```
char s [10];
s [15]  = 'a';
```

Instrumented C code

```
char s [10];  int  sAlloc  = 10;
                   assert  (15 < sAlloc);
s [15]  = 'a';
```

The integer variable sAlloc remembers the space allocated for string s.

The statement assert(15 $<$ sAlloc); says that the program should abort here if sAlloc $\leq$ 15.

We use an integer analysis algorithm to check that the assert conditions are satisfied.

Handling pointer arithmetic.

Original C code

```
char s [10];
char *p;
p = s + 7;
p[5] = ’a’;
```

Handling pointer arithmetic.

Instrumented C code

Original C code

```
char s[10];
char *p;
p = s + 7;
p[5] = 'a';
```

```
char s[10];   int sAlloc = 10;
char *p;      int pAlloc = 0;
              assert (7 <= sAlloc);
p = s + 7;    pAlloc = sAlloc - 7;
              assert (5 < pAlloc);
p[5] = 'a';
```

The second assert condition does not hold, as desired.