Memory corruption: overflows, exploitation & defenses

Sistemi di Calcolo (Part II – Spring 2018)

Instructors: Daniele Cono D'Elia, Riccardo Lazzeretti

The present material has been adapted mainly from:

- *Secure Programming lectures by David Aspinall*
- *The 10K Students Challenge by SysSec Consortium*

Memory corruption vulnerabilities

- Memory corruption vulnerabilities let the attacker cause the program to write to areas of memory (or write certain values) that the programmer did not intend.
- In the worst cases, these can lead to **arbitrary command execution** under the attacker's control.

Reasons for memory corruption

- Memory corruption vulnerabilities arise from possible:
  - buffer overflows, in different places
    - stack overflows
    - heap overflows
  - other programming mistakes
    - out-by-one errors
    - type confusion errors
    - format strings
    - use-after-free

Role of programming languages

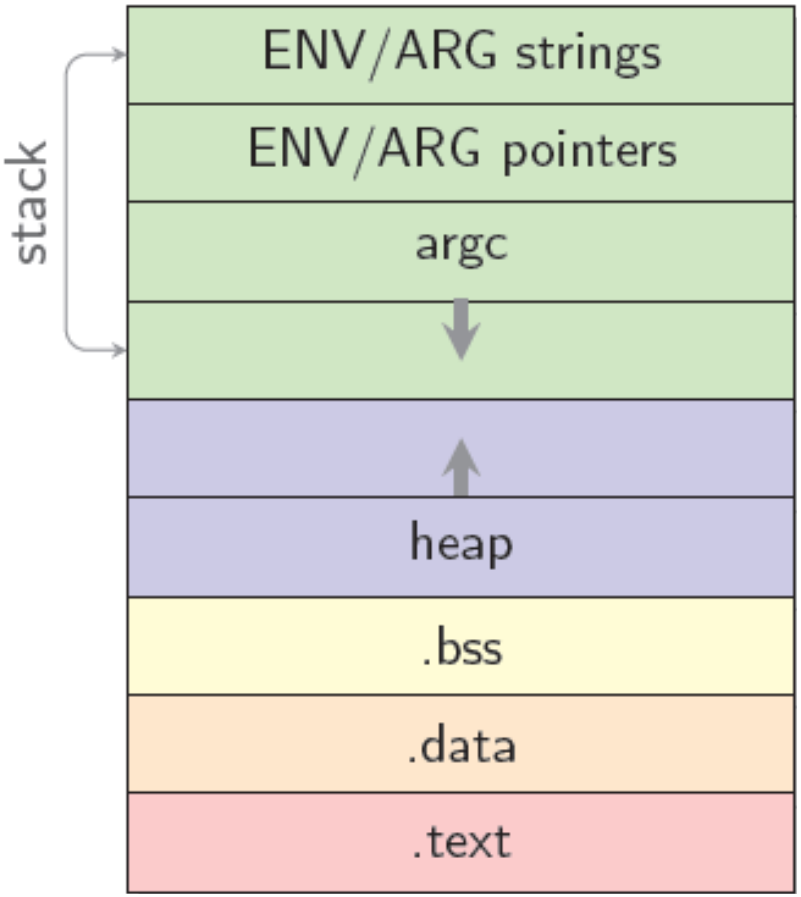
- Low-level languages manipulate memory directly
- Advantage: efficient, precise
- Disadvantage: easy to violate data abstractions
  - arbitrary access to memory
  - pointers and **pointer arithmetic**
  - mistakes violate *memory safety*

A programming language or analysis tool is said to enforce **memory safety** if it ensures that reads and writes stay within clearly defined memory areas, belonging to different parts of the program. Memory areas are often delineated with *types* and a *typing discipline*.

Instant x86 ASM programming

- Hundreds of instructions from several families:
  - Data movement: MOV ...
  - Arithmetic: ADD, FDIV, IDIV, MUL, ...
  - Logic: AND, OR, XOR, ...
  - **Control**: JMP, CALL, LEAVE, RET, ...
- General-purpose registers are split into pieces:
  - 32 bits: EAX (extended A)
  - 16 bits: AX
  - 8 bits: AH, AL (high and low bytes of A)
- Some registers are used as pointers to segments

Memory Layout of a Process



How the stack works

The **program stack** (aka function stack, runtime stack) holds *stack frames* (aka *activation records*) for each function that is invoked.

- Very common mechanism for high-level language implementation
  - Exact mechanisms vary by CPU, OS, language, compiler, compiler flags.
- So has special CPU support
  - Stack pointer register: ESP
  - Frame pointer register: EBP
  - push and pop machine instructions

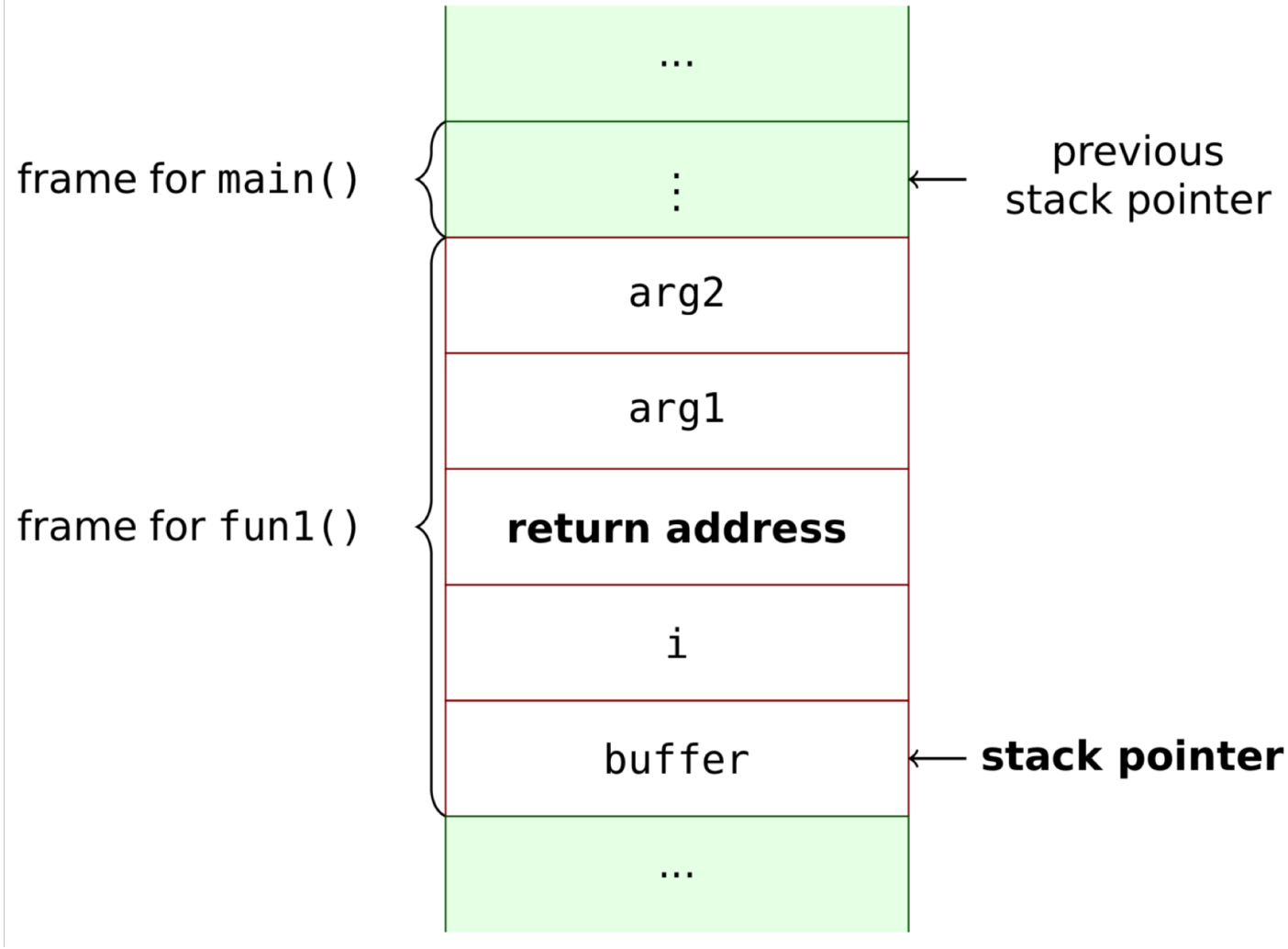
Stack usage with function calls

- Parameters may be passed to the function body on the stack or in registers; the precise mechanism is called the **calling convention**.
- Local variables are allocated space on the stack.
- A **frame pointer** may be used to help locate arguments and local variables.

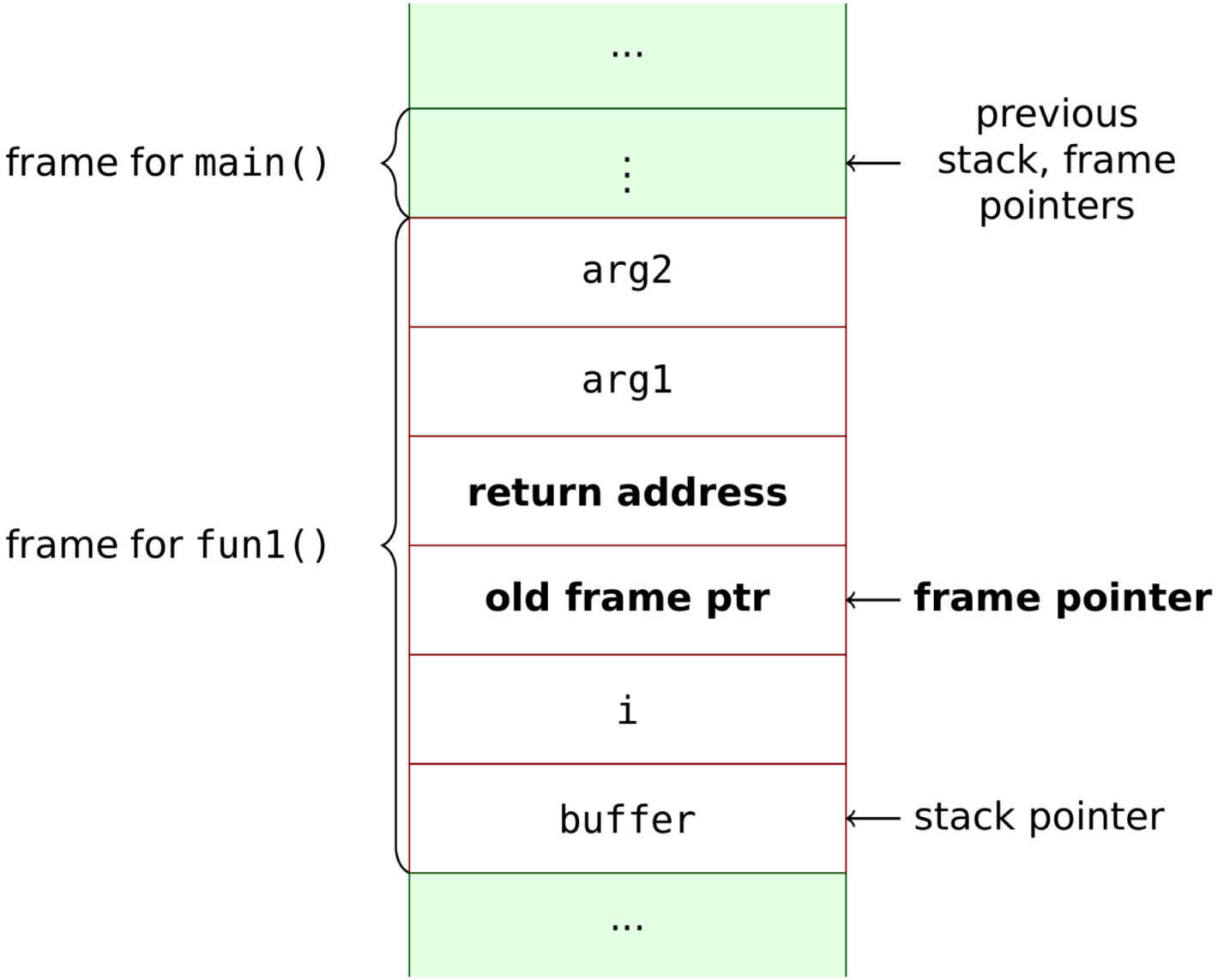
An example

```
void fun1(char arg1, int arg2) {
    char *buffer[5];
    int i;
    *buffer[0] = (char)i;
}
void main() {
  fun1('a', 77);
}
```

Stack usage with function calls



Using frame pointers



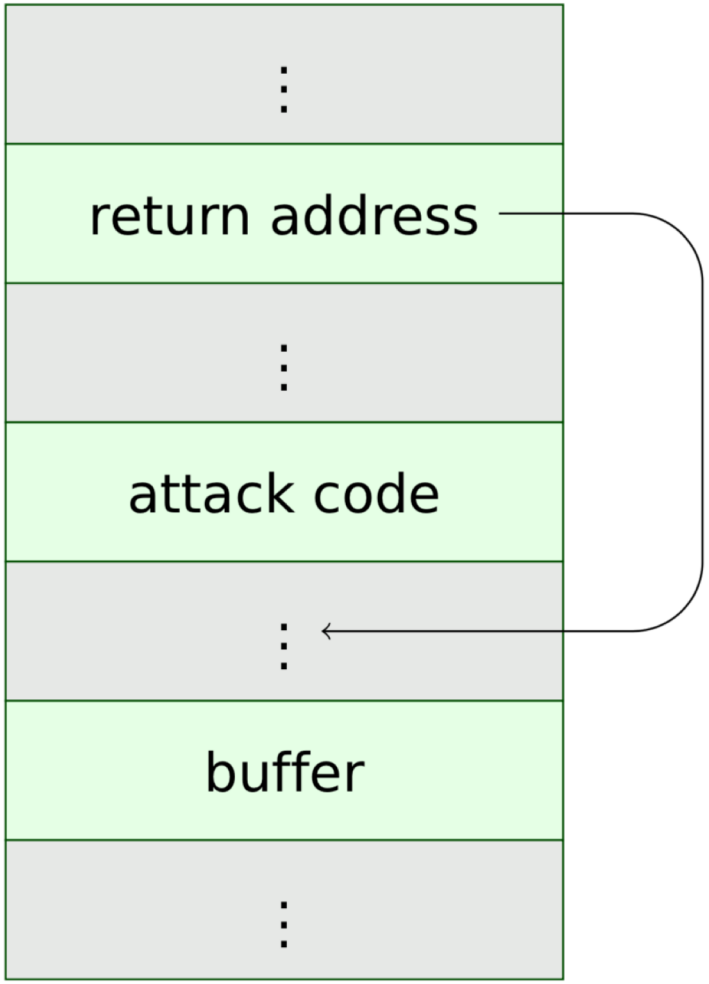
Assembly code for the example

```
fun1:
  pushl   %ebp              ; save previous frame pointer
  movl    %esp, %ebp        ; set new frame pointer
  subl    $36, %esp         ; allocate enough space for locals
  movl    -24(%ebp), %eax   ; EAX = address of buffer[0]
  movl    -4(%ebp), %edx    ; EDX = i
  movb    %dl, (%eax)       ; set buffer[0] to be low byte of i
  leave                     ; drop frame
  ret                       ; return

main:
  pushl   %ebp              ; save previous frame pointer
  movl    %esp, %ebp        ; set new frame pointer
  subl    $8, %esp          ; allocate space for fun1 arguments
  movl    $77, 4(%esp)      ; store arg2
  movl    $97, (%esp)       ; store arg1 (ASCII 'a')
  call    fun1              ; invoke fun1
  leave                     ; drop frame
  ret                       ; return
```

Stack overflow

- Stack overflow attacks were first carefully explained in *Smashing the stack for fun and profit*, a paper written by Aleph One for the hacker's magazine **Phrack**, issue 49, in 1996.
- Stack overflows are mainly relevant for C, C++ and other unsafe languages with raw memory access (e.g., pointers and **pointer arithmetic**).
- Languages with built-in **memory safety** such as Java, C#, Python, etc, are immune to the worst attacks — *providing* their language runtimes and native libraries have no such exploitable flaws.

Stack overflow: high-level view

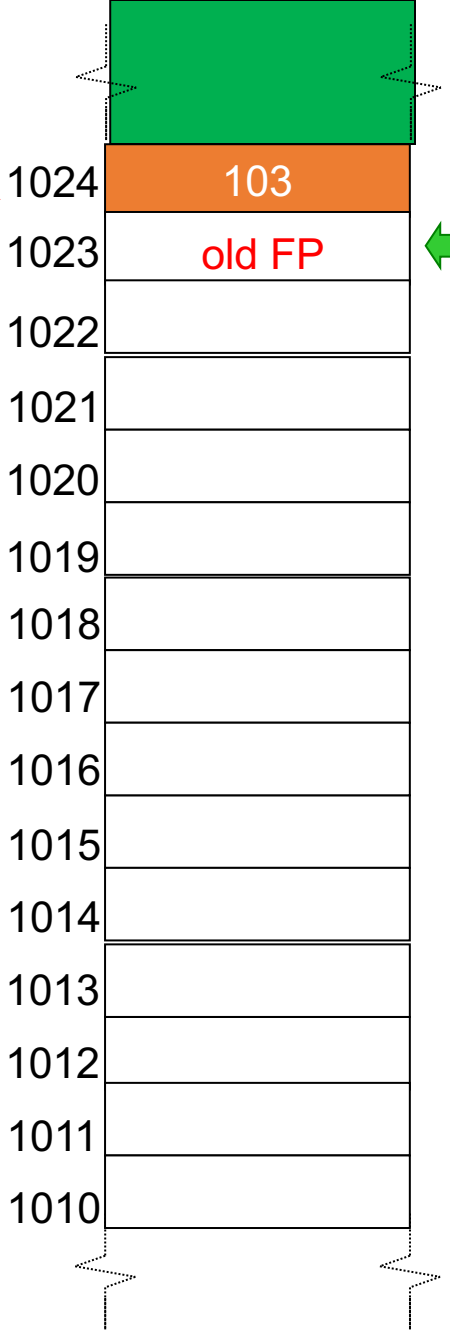


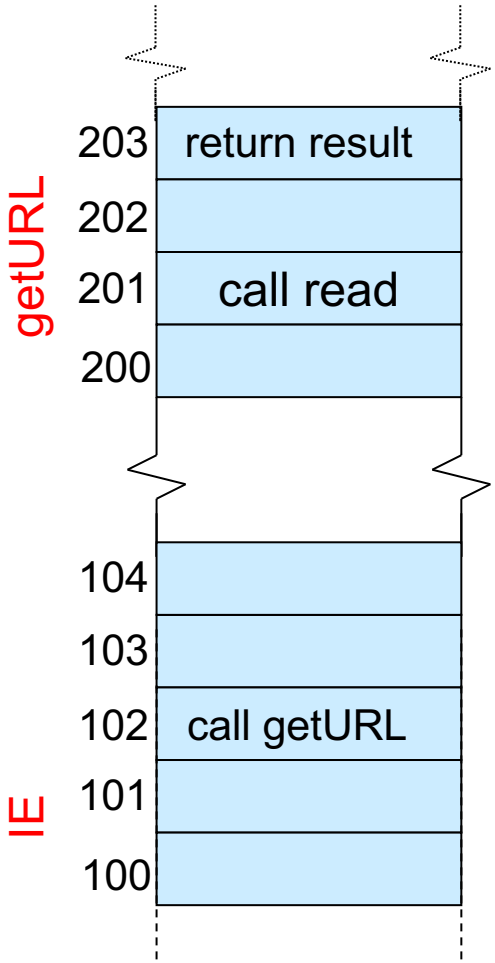
- The malicious payload overwrites all of the space allocated for the buffer, all the way to the return address location.
- The return address is altered to point back into the stack, somewhere before the attack code.
- Typically, the attack code executes a shell.

Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```

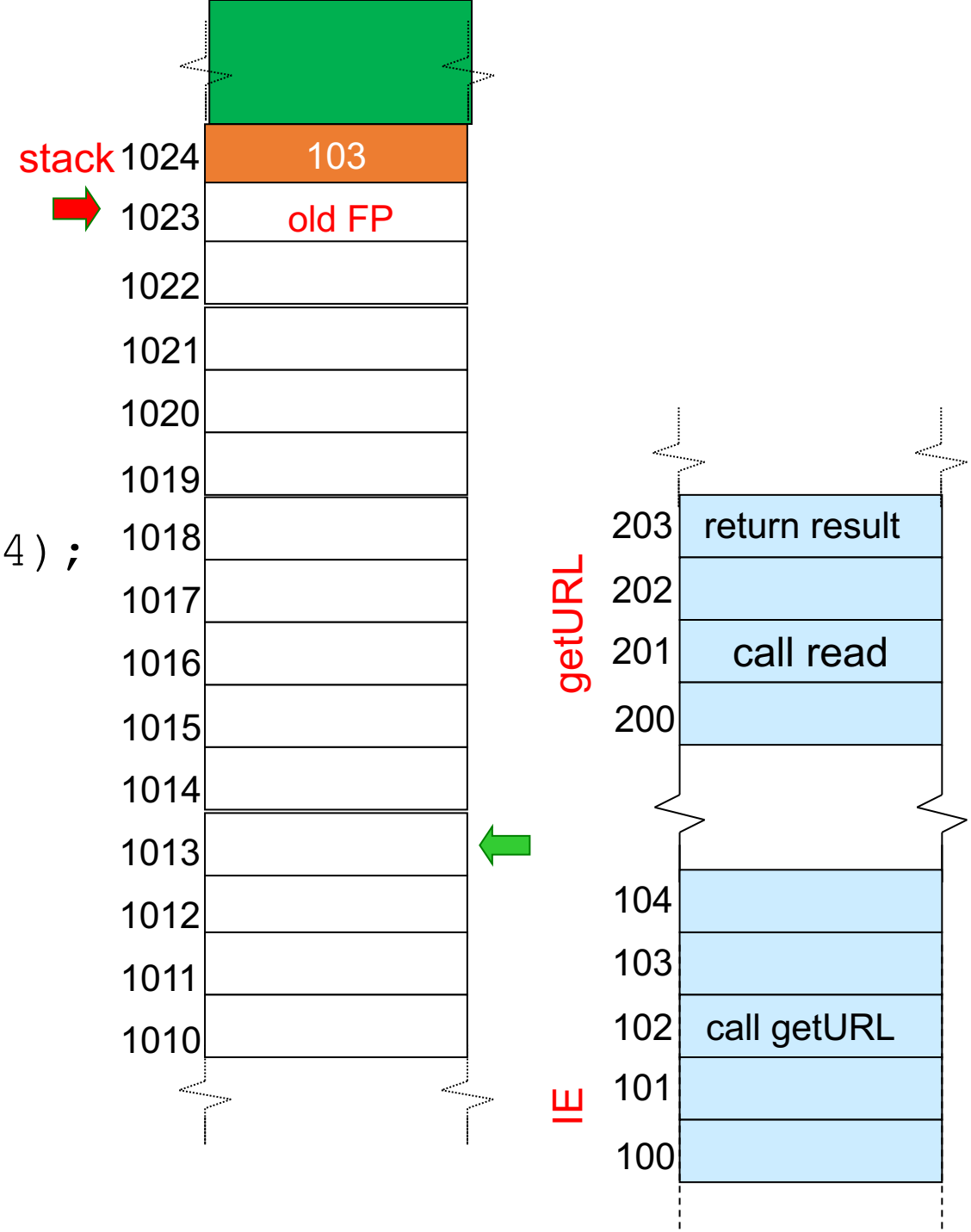
frame pointer (FP)
(points to start of this function's stack frame)





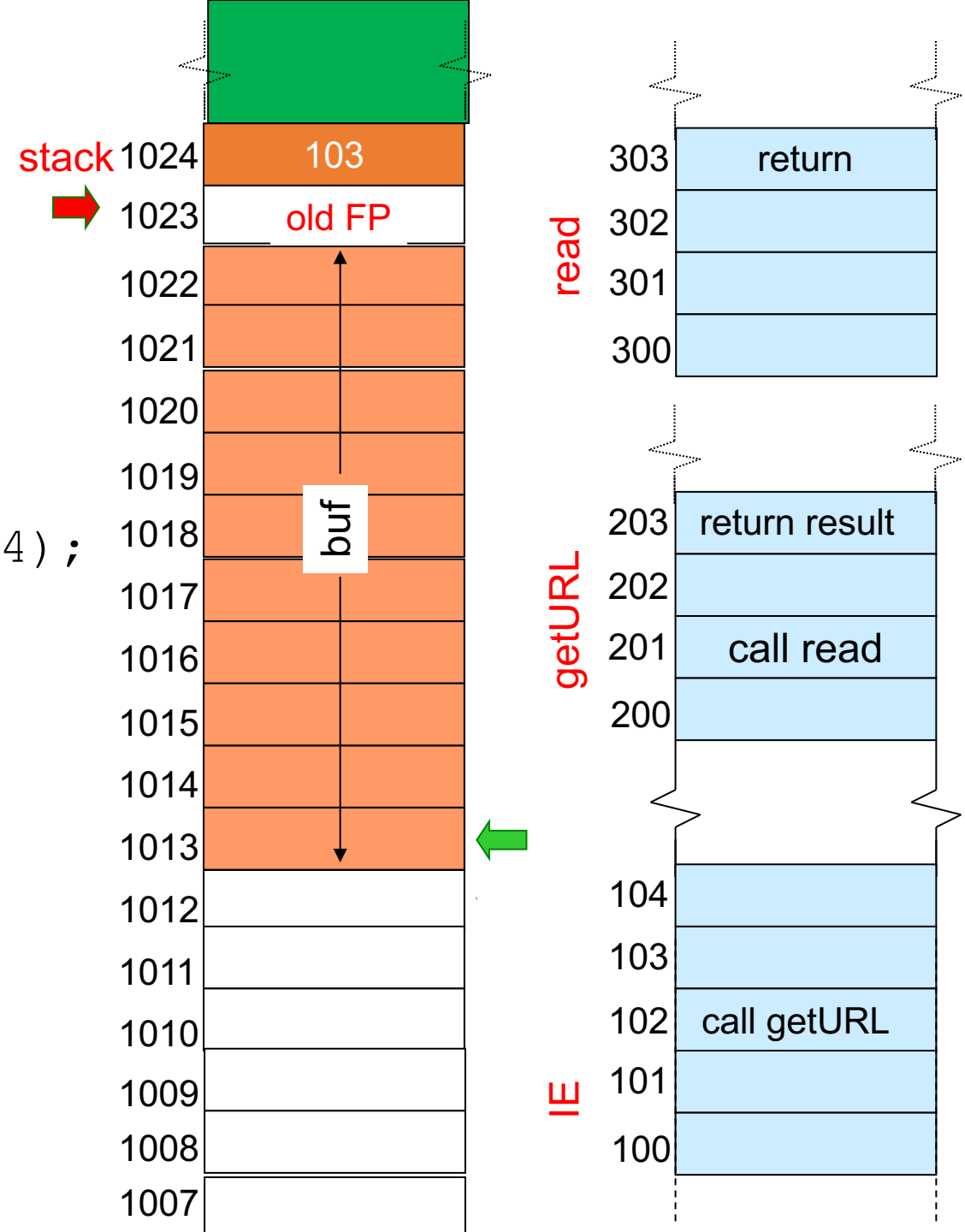
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



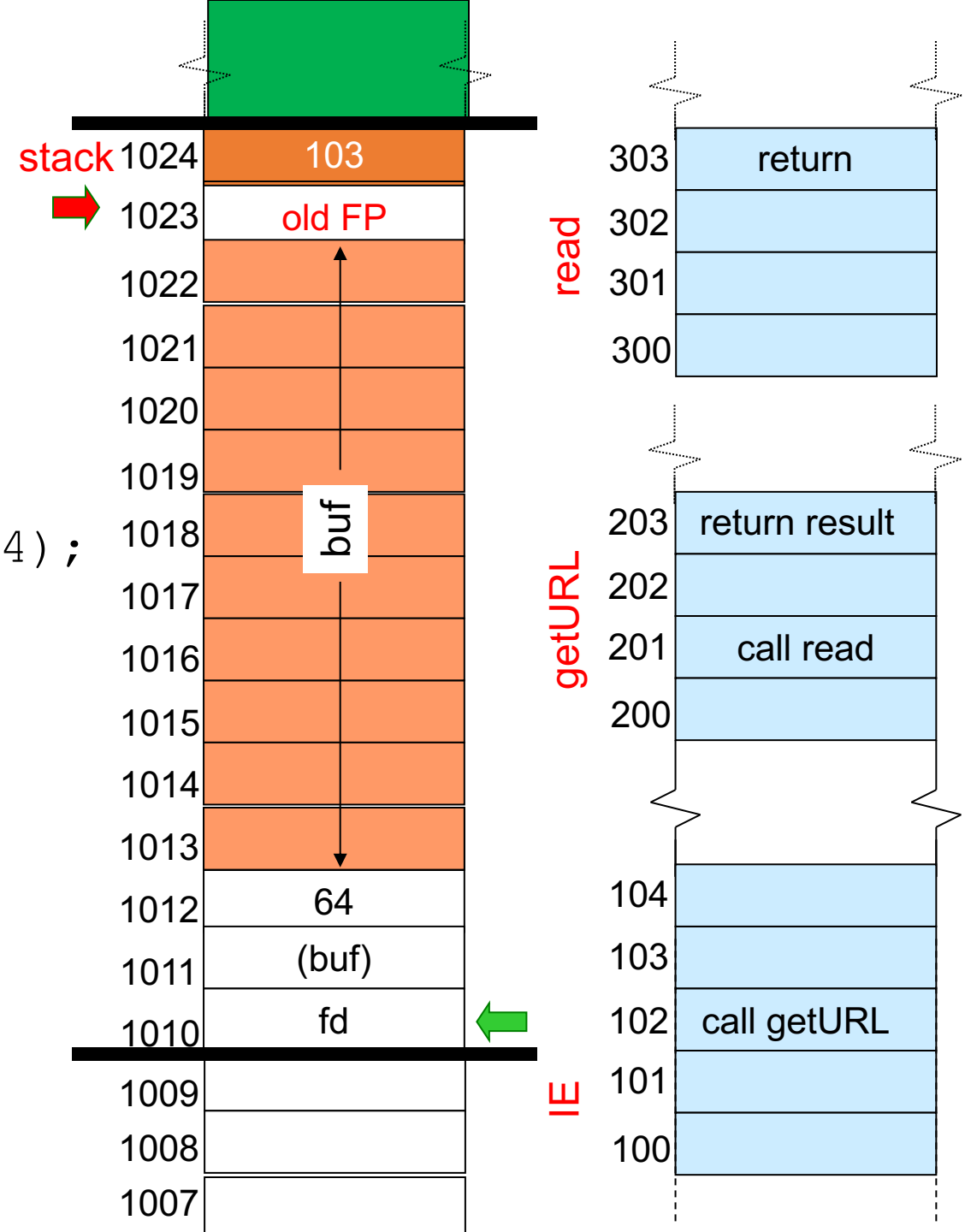
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



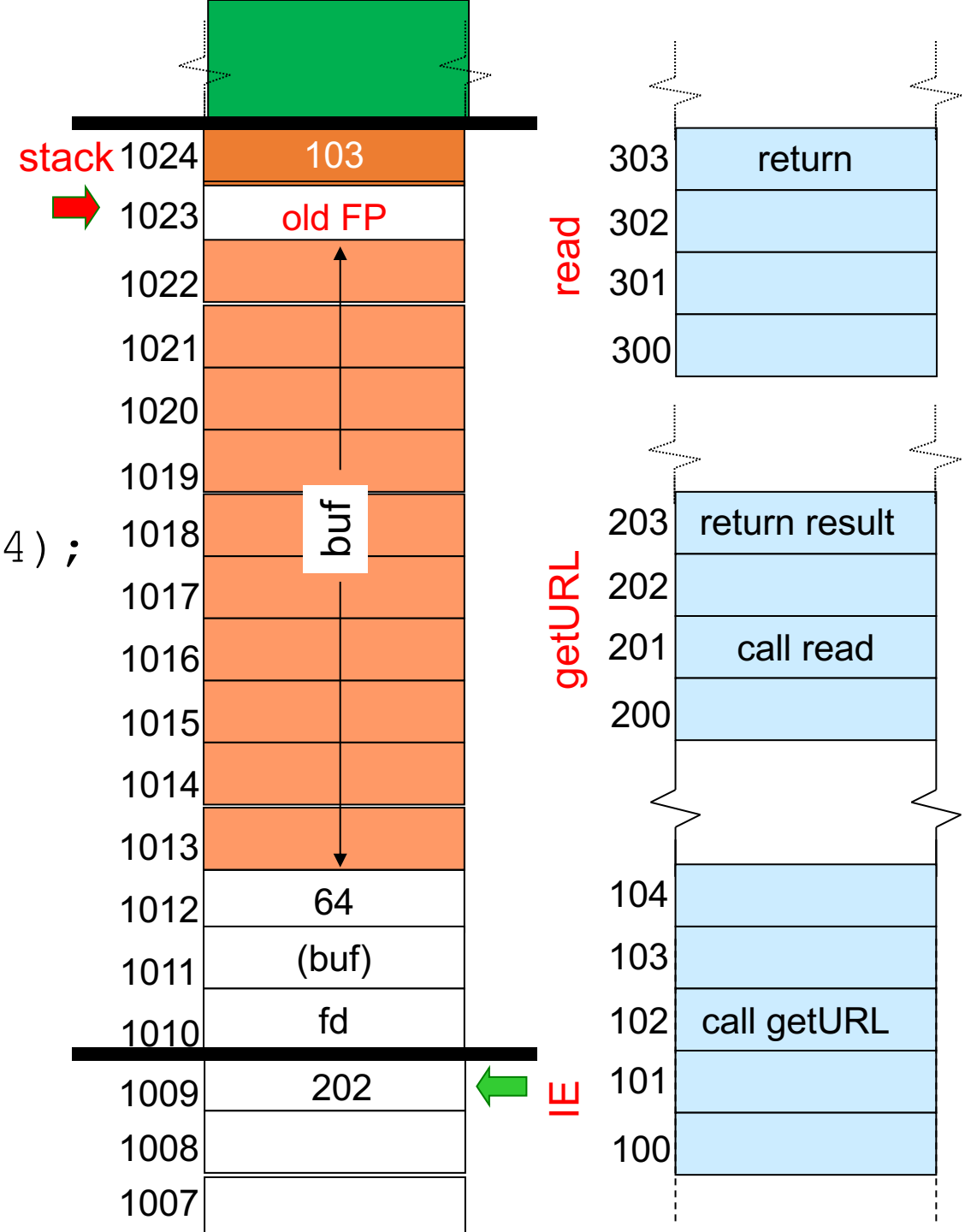
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



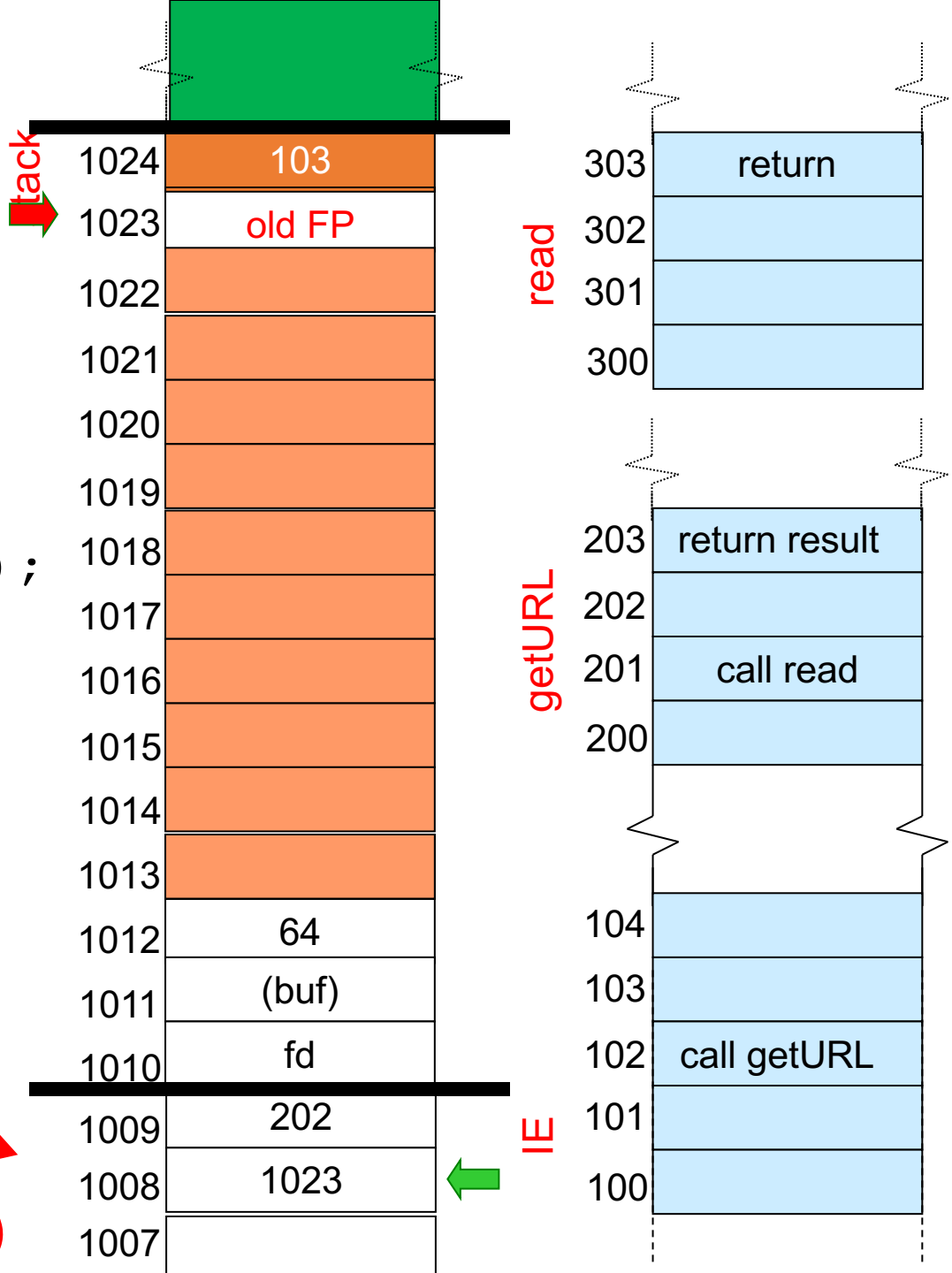
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



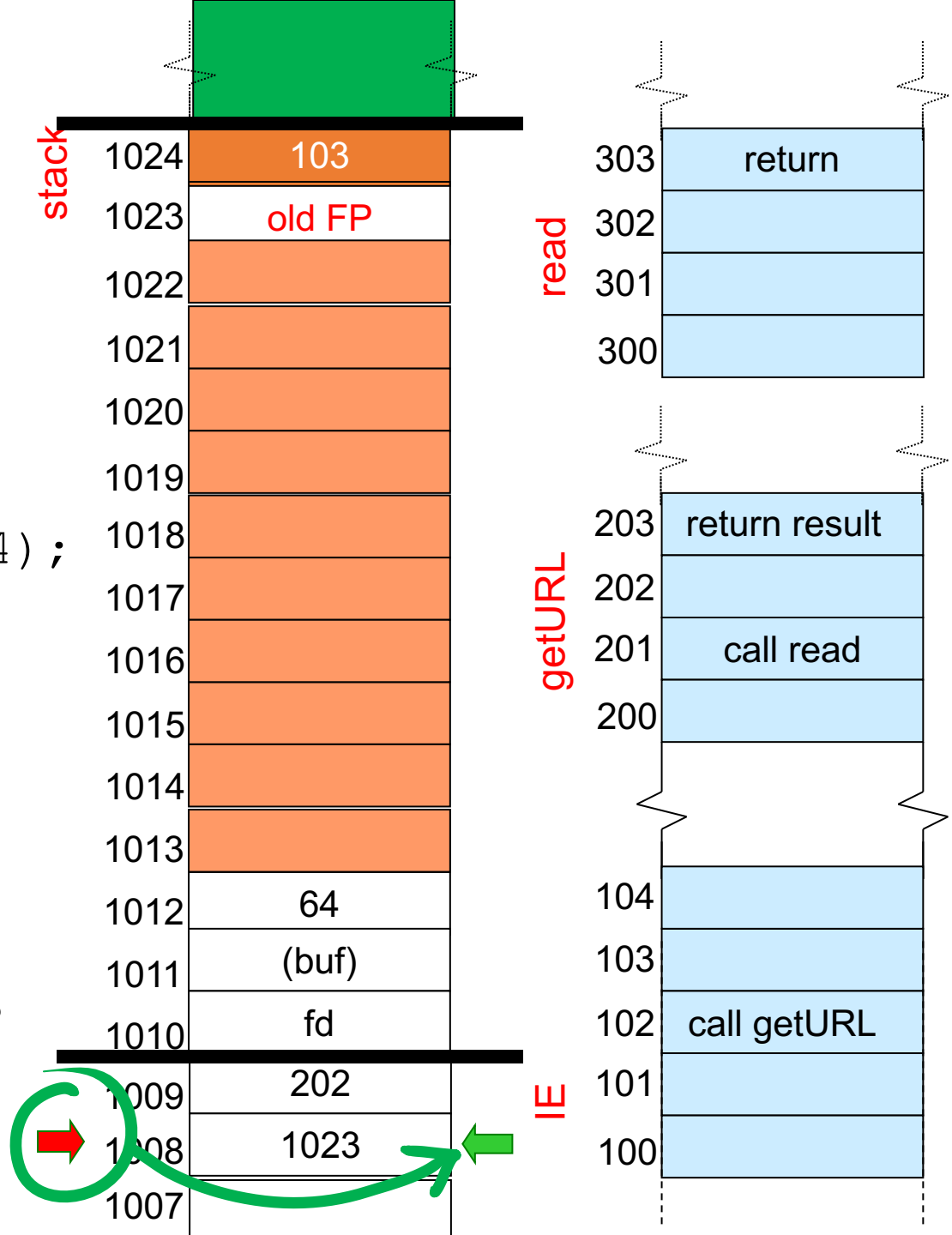
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



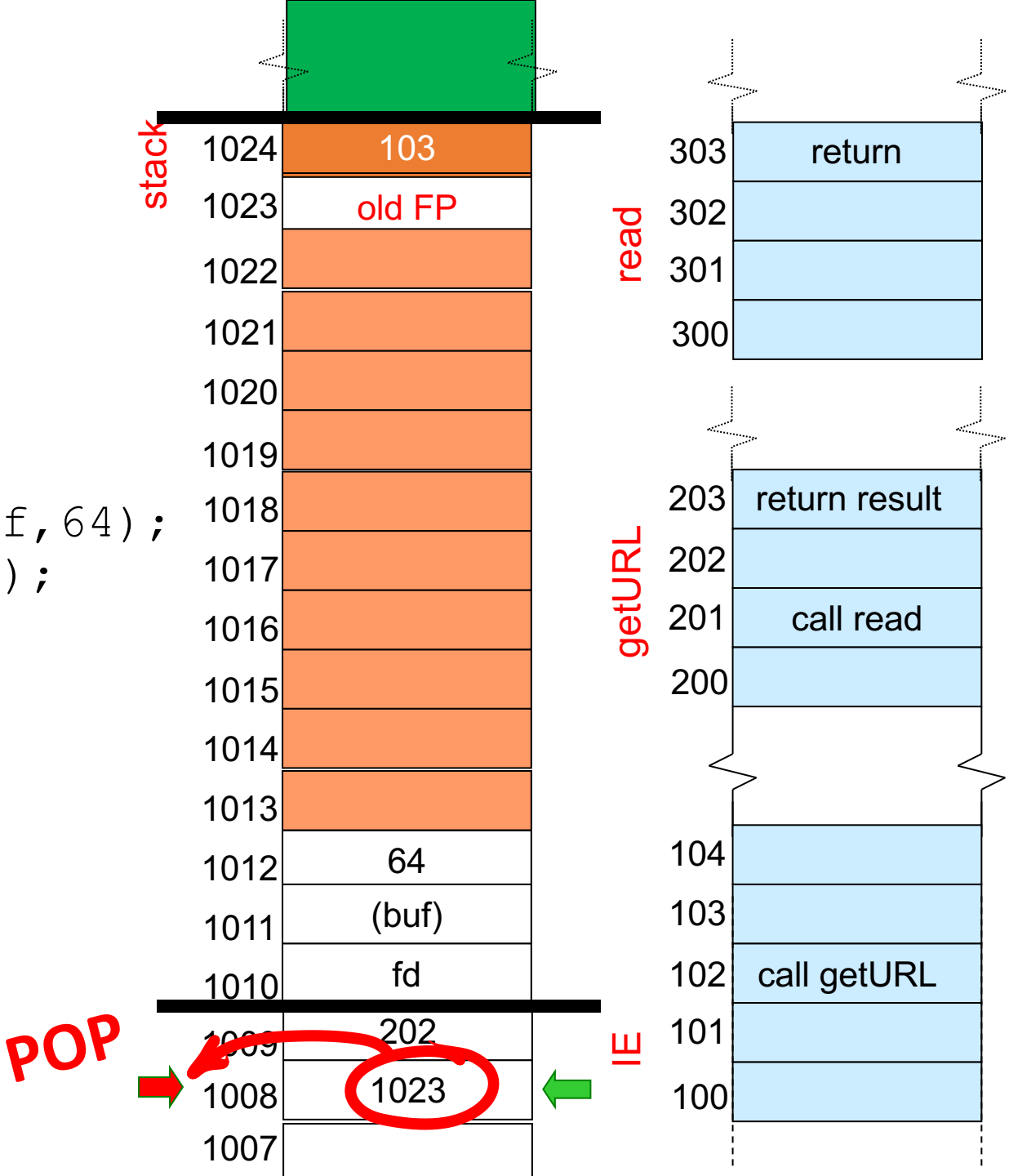
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



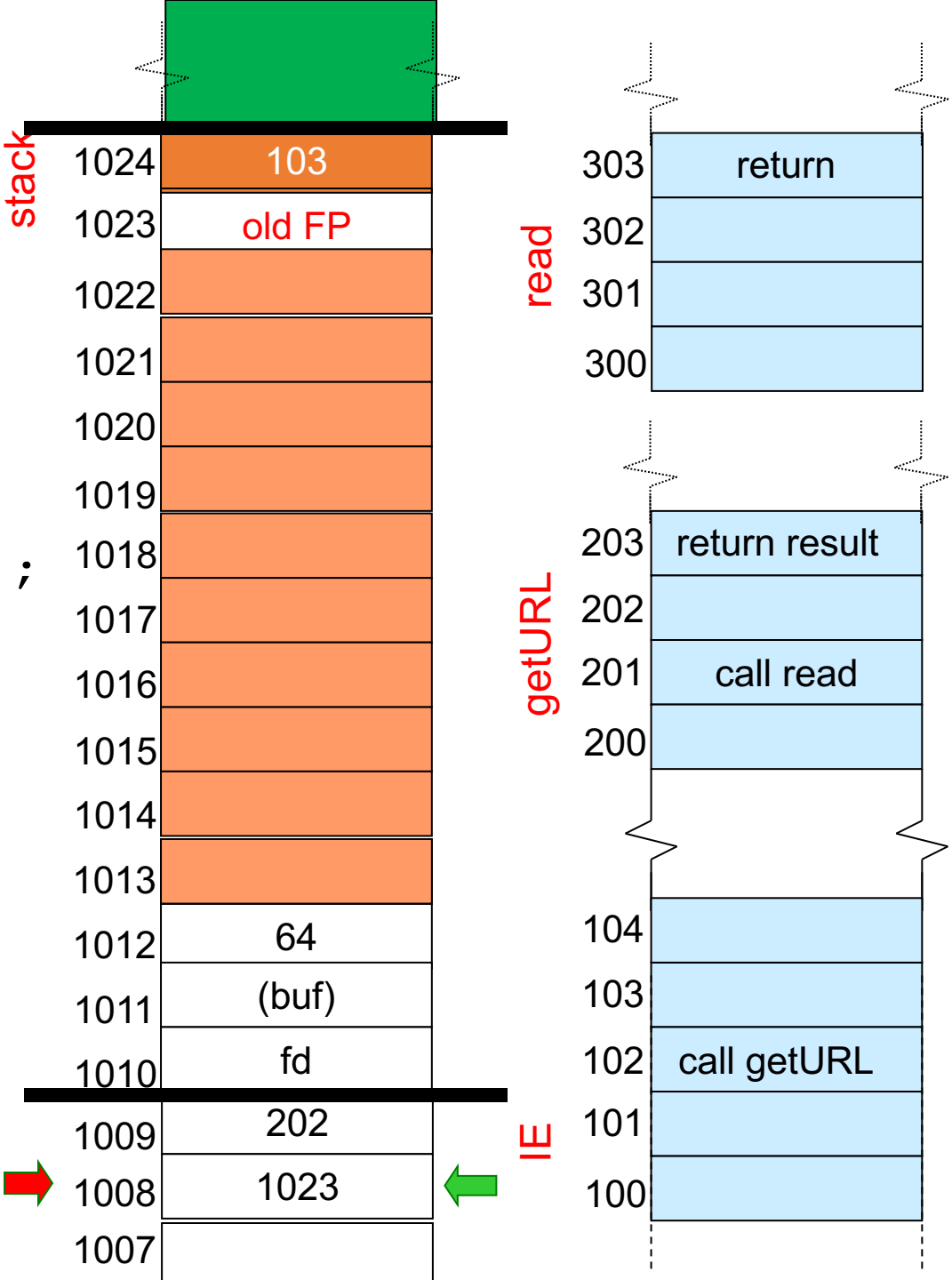
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



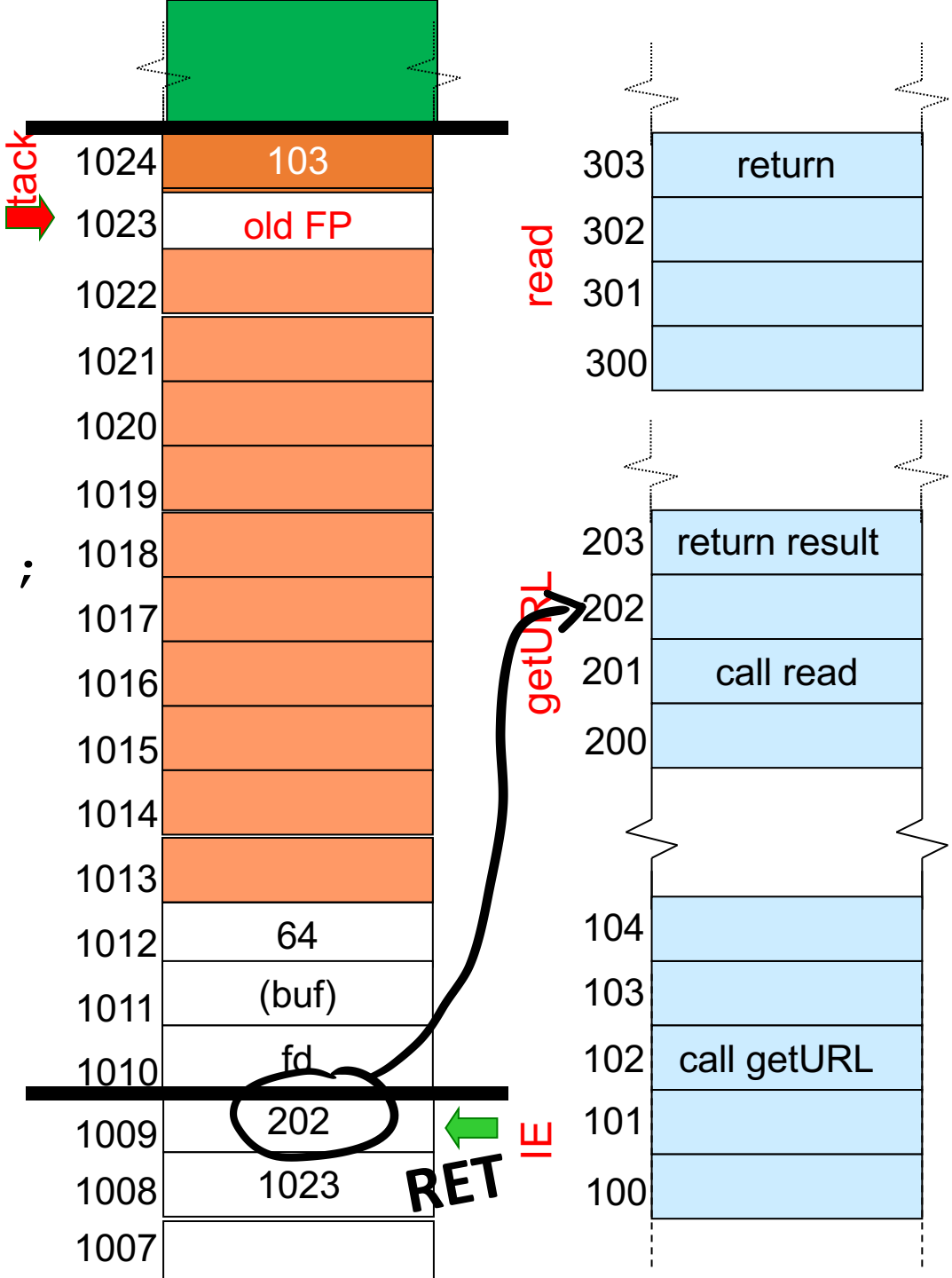
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



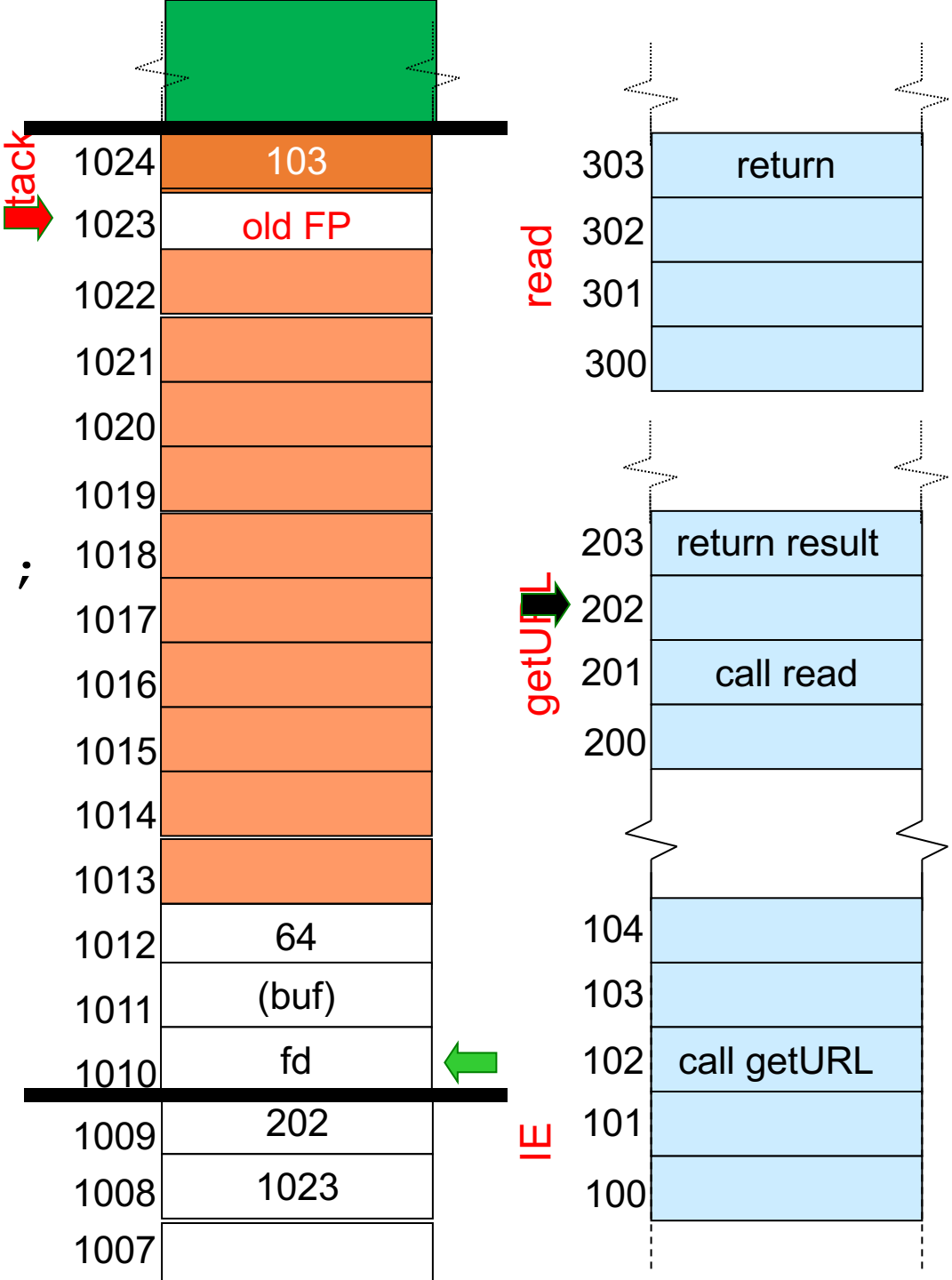
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



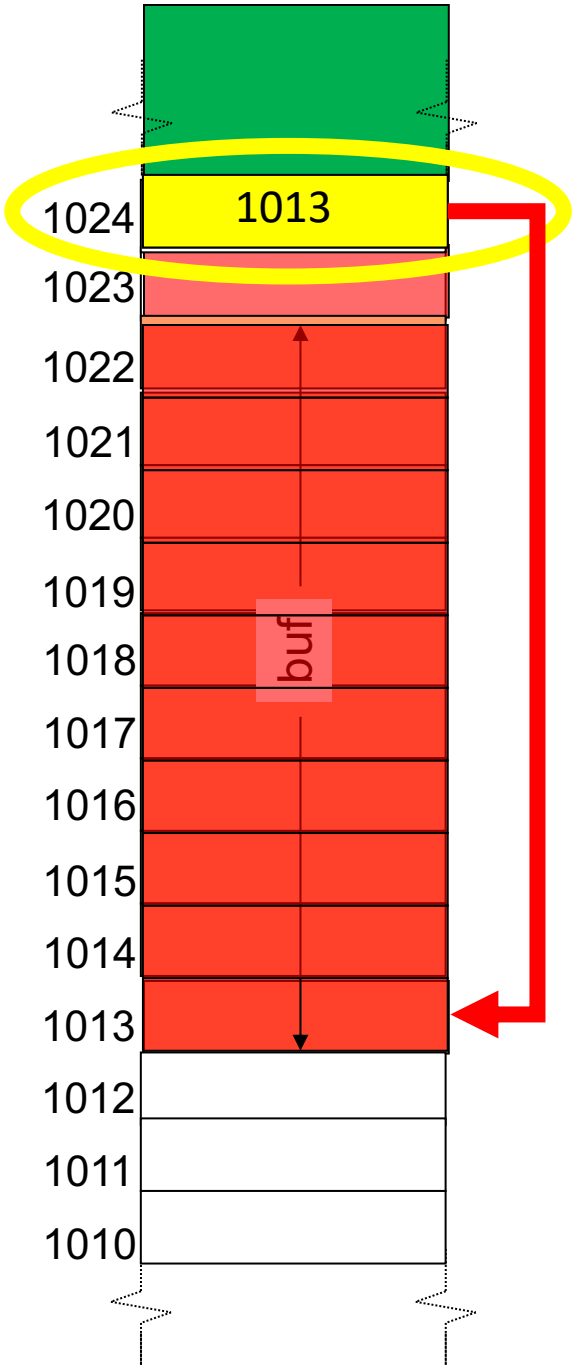
Warm-up example

```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Where is the vulnerability?

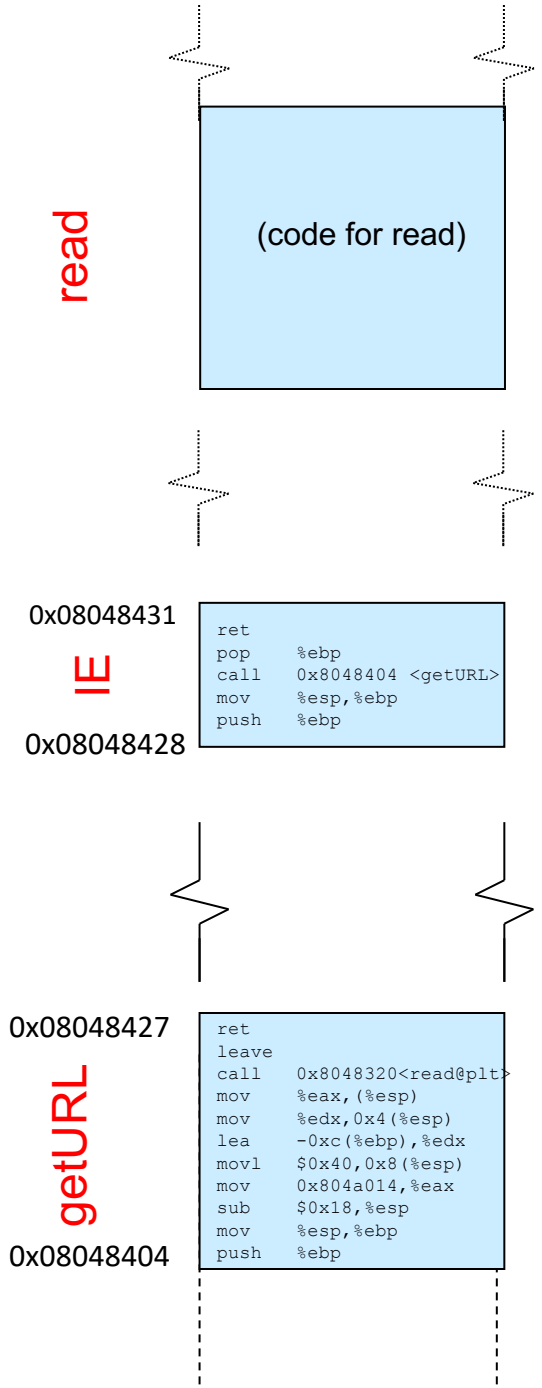
```
getURL ()
{
   char buf[10];
   read(keyboard, buf, 64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Let’s make it more realistic!

So we have:

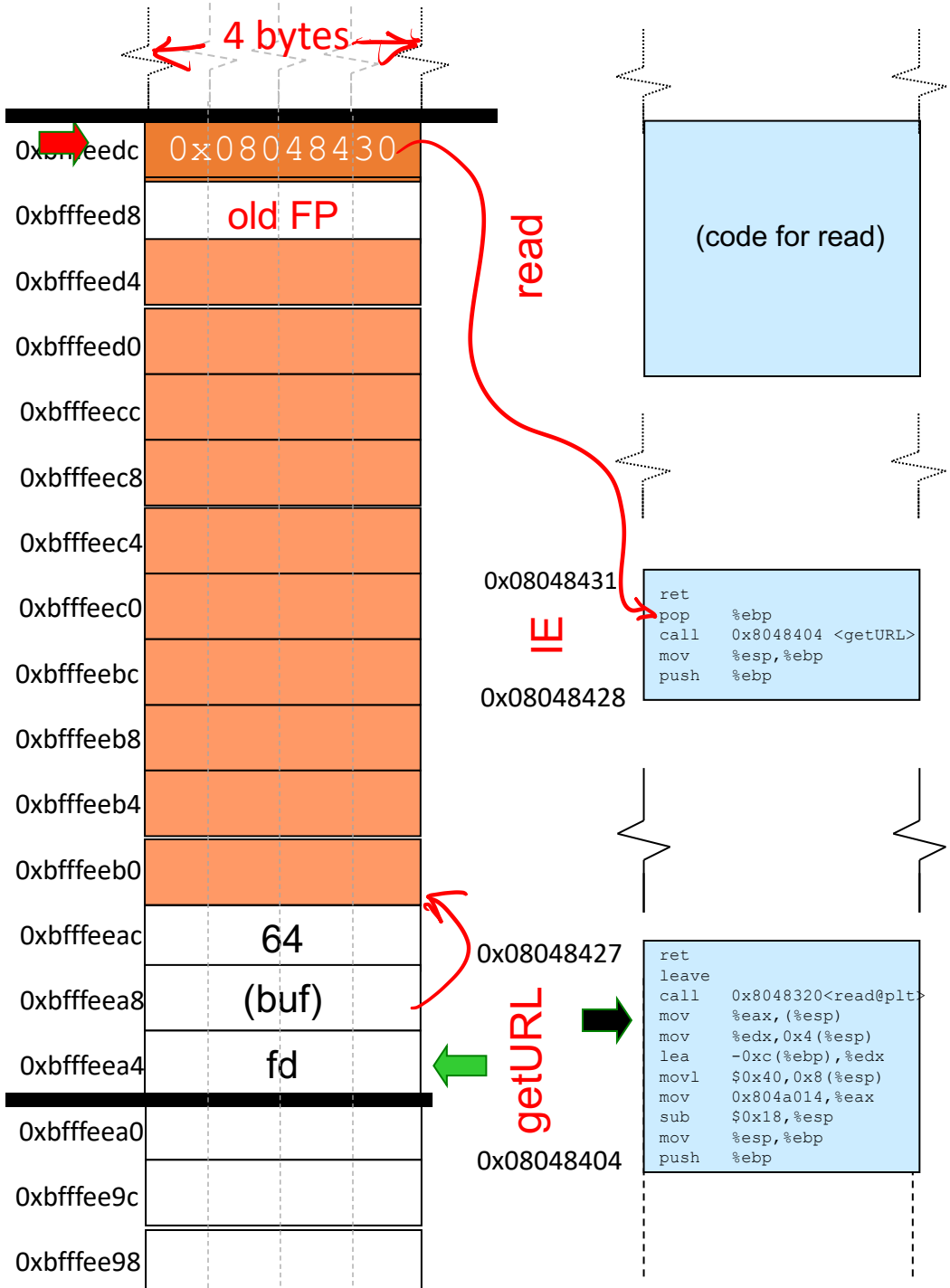
```
getURL ()
{
   char buf[40];
   read(stdin,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



What about the stack?

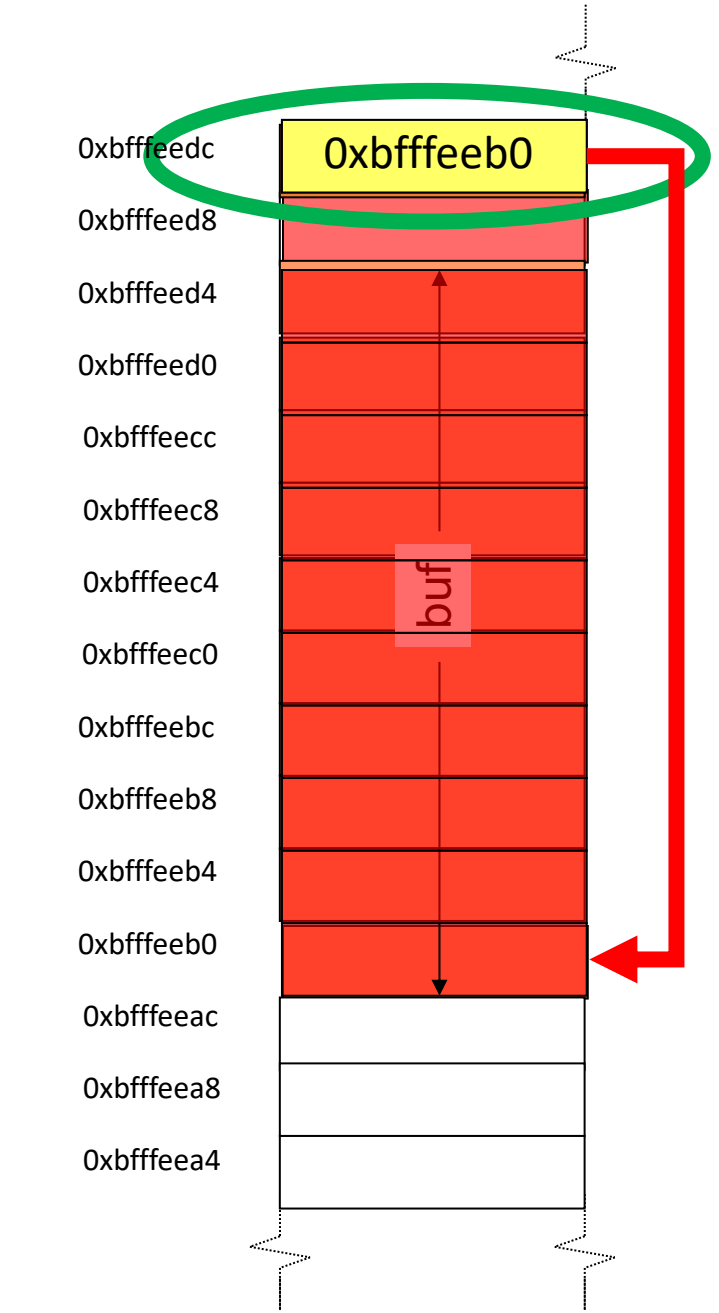
When getURL is about to call 'read'

```
getURL ()
{
   char buf[40];
   read(stdin,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Exploit

```
getURL ()
{
   char buf[10];
   read(fd, buf, 64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



0xbfffee98

That is it, really. Or not?

All we need to do is stick our program in the buffer

- Easy to do: attacker controls what goes in the buffer!
  - and that program simply consists of a few instructions (not unlike what we saw before)

But sometimes...

- We don't even need to change the return address
- Or execute any of our code

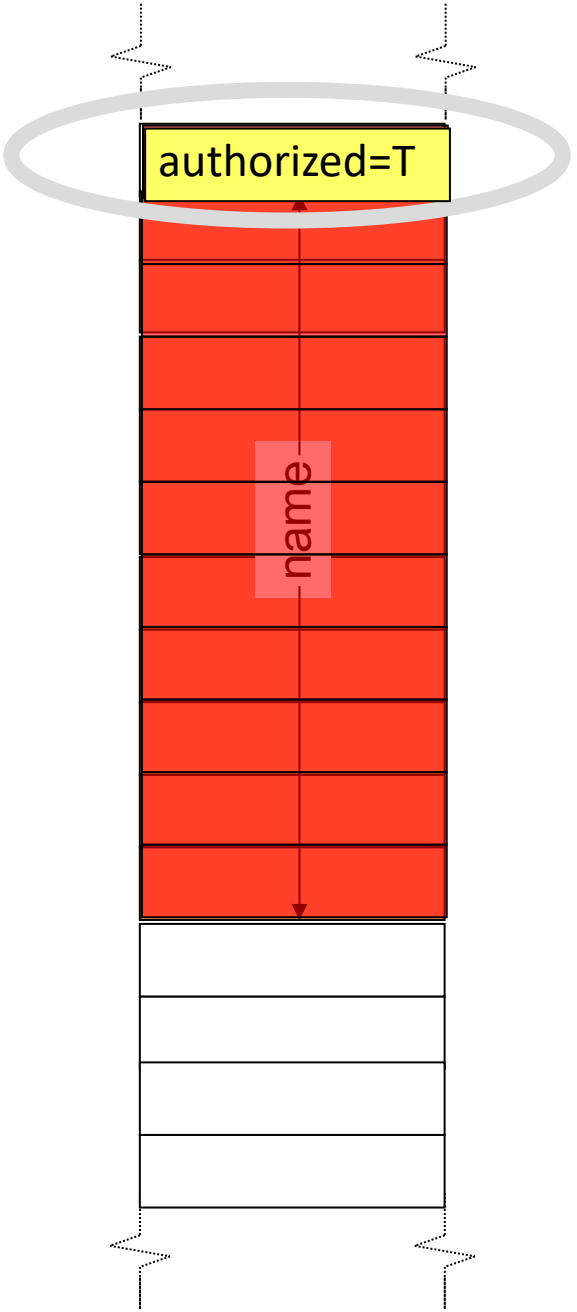
  Let's have a look at an example, where the buffer overflow changes only data…

Exploit against non-control data

```
get_medical_info()
{
   boolean authorized = false;
   char name [10];
   authorized = check();
   read_from_network (name);

   if (authorized)
       show_medical_info (name);
   else
       printf ("sorry, not allowed");
}
```

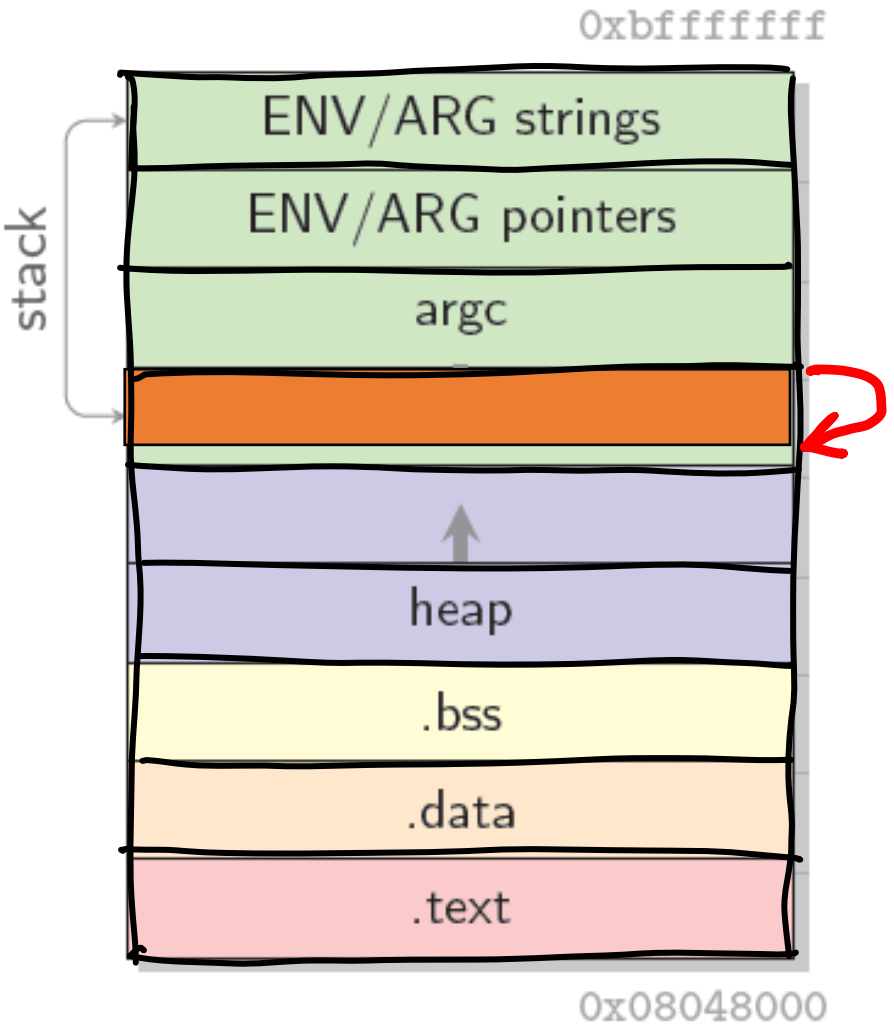
Exploit against non-control data

```
get_medical_info()
{
   boolean authorized = false;
   char name [10];
   authorized = check();
   read_from_network (name);

   if (authorized)
       show_medical_info (name);
   else
       printf ("sorry, not allowed");
}
```



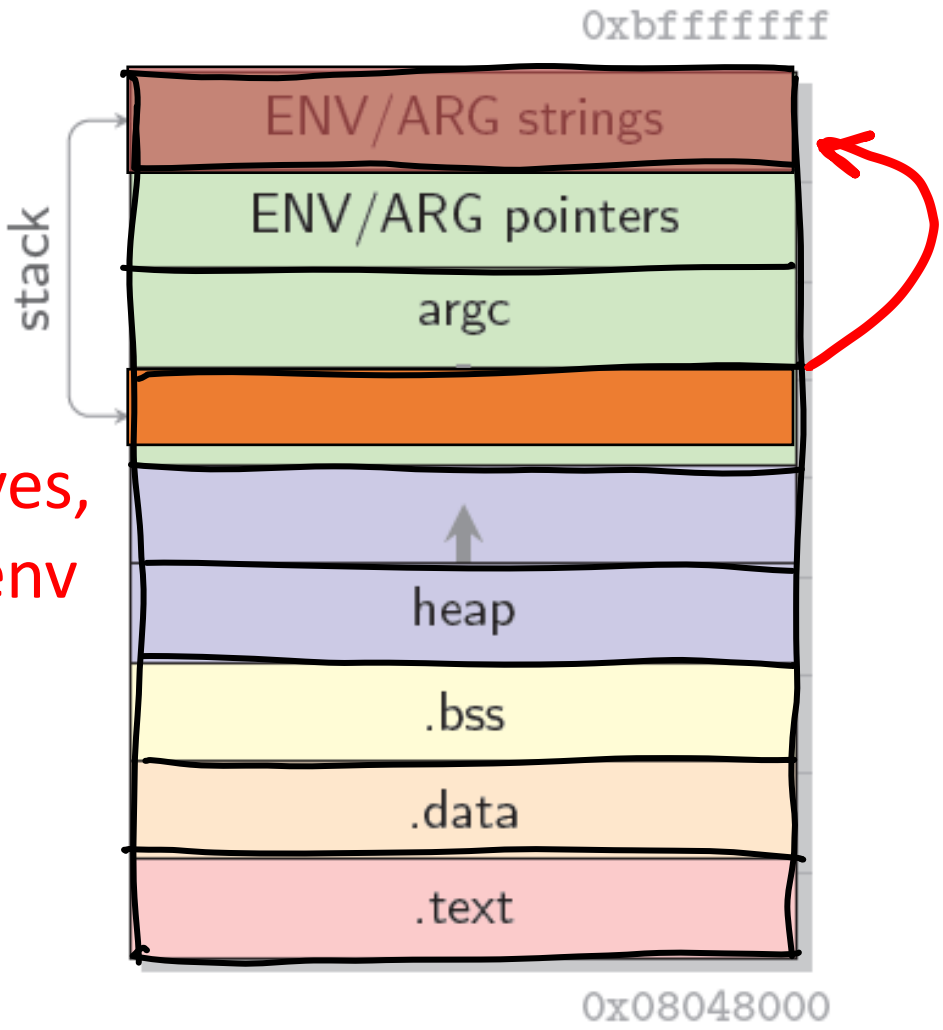
Other return targets also possible!

This is what we did before



But other locations also possible

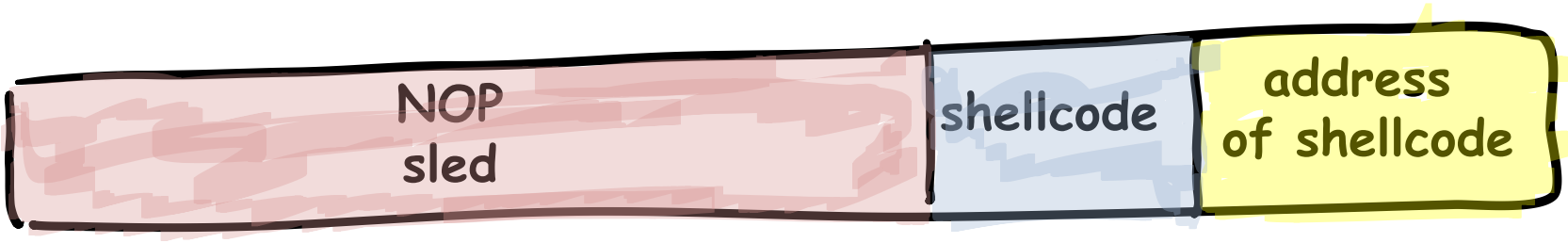
If we start the program ourselves, we control the env



So all the attacker needs to do…

- ... is stick a program in the buffer or environment!
  - Easy: attacker controls what goes in the buffer!
  - For instance, the payload starts a command shell from which the attacker can control the compromised machine
    - hence the name “shellcode”
  - What does such code look like?

Typical injection vector



- Shellcode address:
  - the address of the memory region that contains the shellcode
- Shellcode:
  - a sequence of machine instructions to be executed (e.g. execve("/bin/sh"))
- NOP sled:
  - a sequence of do-nothing instructions (nop). It is used to ease the exploitation: attacker can jump anywhere inside, and will eventually reach the shellcode (optional)

How do you create the vector?

1. Create the shellcode
2. Prepend the NOP sled:

```
perl -e 'print "\x90"' | ndisasm -b 32 -
00000000    90          nop
```

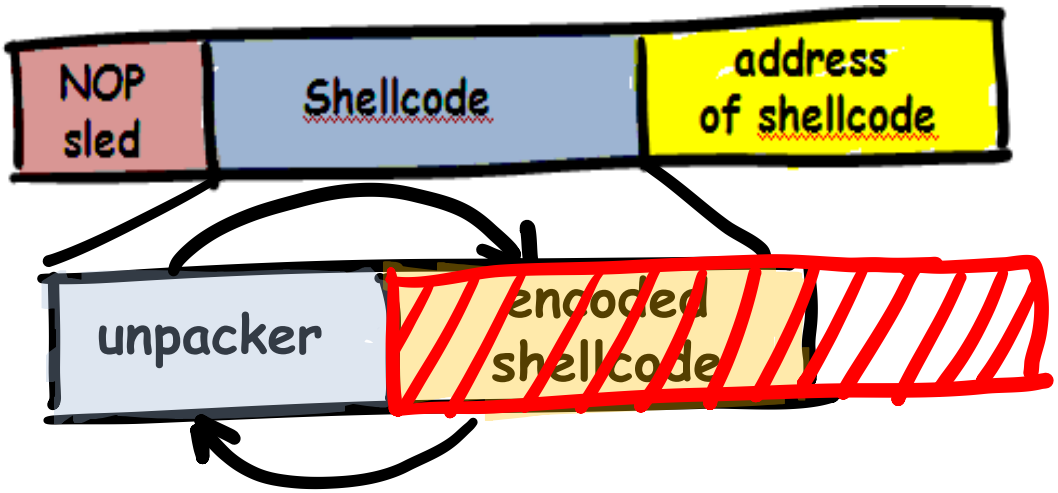
3. Add the address

0xbfffeeb0

```
00000000    31 C0 B0 46   31 DB 31 C9   1..F1.1.
00000008    CD 80 EB 16   5B 31 C0 88   ....[1..
00000010    43 07 89 5B   08 89 43 0C   C..[..C.
00000018    B0 0B 8D 4B   08 8D 53 0C   ...K..S.
00000020    CD 80 E8 E5   FF FF FF 2F   ......./
00000028    62 69 6E 2F   73 68 4E 41   bin/shNA
00000030    41 41 41 42   42 42 42 00   AAABBBB.
```

```
_start:
        xor %eax, %eax
        movb $70,%al        setreuid
        xor %ebx,%ebx
        xor %ecx,%ecx
        int $0x80

        jmp string_addr

mystart:
        pop %ebx
        xor %eax,%eax

        movb %al, 7(%ebx)
        movl %ebx, 8(%ebx)

        movl %eax, 12(%ebx)

        movb $11,%al        execve

        leal 8(%ebx), %ecx
        leal 12(%ebx), %edx

        int $0x80

string_addr:                why this?
        call mystart
        .asciz "/bin/shNAAAABBBB"
```

In reality, things are more complicated



- why do you think encoding is so frequently used?
  - think strcpy(), etc.

A: if strcpy() is used to overflow the buffer, it will stop when it encounters the null byte. So if the shellcode contains a null byte, the attacker has a problem. So the attacker may have to encode the shellcode to remove null bytes and then generate them dynamically

Off-by-one attack (frame pointer overwrite)

- In some cases only one overflow byte is left at the mercy of the attacker
  - we aim for the least significant byte of the saved frame pointer that a function's prologue pushes to the stack
- Corrupting the saved frame pointer allows the attacker to eventually corrupt the stack pointer
  - we can make it point to some location inside the buffer!
- Once the attacker controls ESP, they can cause the next ret instruction to jump to their shellcode

Off-by-one attack: example

```
void func(char *str)
{
  char buf[256];
  int i;

  for (i=0;i<=256;i++)
     buf[i]=str[i];
}

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

Credits: Introduction to Software Exploitation by Corey K.

Off-by-one overflow: example

```
0x08048236 <func+76>:   mov    ecx,DWORD PTR [ebp+8]
0x08048239 <func+79>:   mov    edx,DWORD PTR [ebp-0x104]
0x0804823f <func+85>:   add    ecx,edx
0x08048241 <func+87>:   movsx  edx,BYTE PTR [ecx]
0x08048244 <func+90>:   mov    BYTE PTR [eax],dl
0x08048246 <func+92>:   jmp    0x8048215 <func+43>
0x08048248 <func+94>:   leave
0x08048249 <func+95>:   ret
End of assembler dump.
```

Effects of leave@<func+94>

```
mov ebp, esp
pop ebp ; ebp := corrupted ebp
```

```
disas main
of assembler code for function main:
4824a <main+0>:    push   ebp
4824b <main+1>:    mov    ebp,esp
4824d <main+3>:    sub    esp,0x0
48253 <main+9>:    mov    eax,DWORD PTR [ebp+12]
48256 <main+12>:   add    eax,0x4
48259 <main+15>:   mov    ecx,DWORD PTR [eax]
4825b <main+17>:   push   ecx
4825c <main+18>:   call   0x80481ea <func>
48261 <main+23>:   add    esp,0x4
48264 <main+26>:   leave
48265 <main+27>:   ret
f assembler dump.
```

Effects of leave@<main+26>

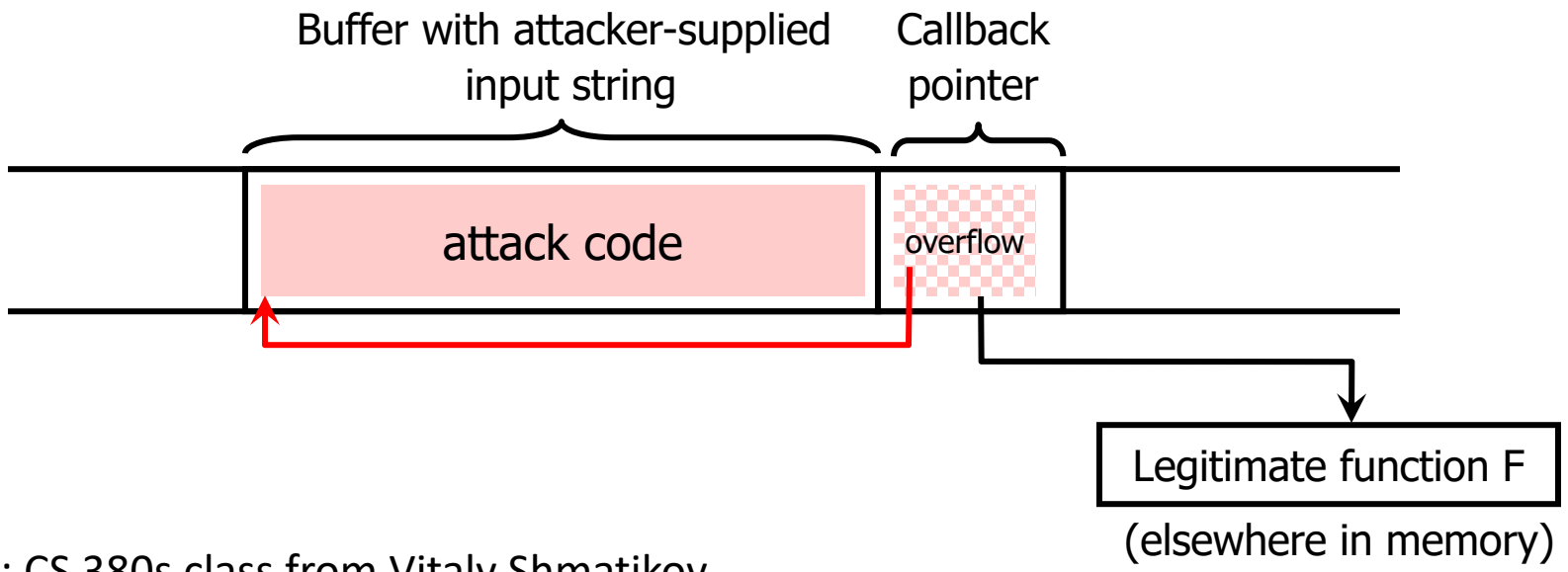
```
mov ebp, esp ; esp := corrupted ebp
pop ebp
```

Effects of ret@<main+27>

```
mov <corrupted ebp>, eip
```

Function pointer overwrite

- C uses function pointers for callbacks: if a pointer to some function F is stored in memory location P, then another function G can call F as (*P)(…)
  - example: providing a comparator for sorting



Credits: CS 380s class from Vitaly Shmatikov

From last year's exams... :-)

```
unsigned sleep(unsigned seconds); // <unistd.h>

void bar(int arg) {
    char buf[64];
    foo(buf, arg, &sleep);
    printf("%s\n", buf);
}

int foo(char* buf, int arg, unsigned (*funcp)(unsigned)) {
    char tmp[128];
    gets(tmp);
    strncpy(buf, tmp, 64);
    (*funcp)(arg);
    return 0;
}
```

How do we stop the attacks?

- Code more carefully!
- The best defense is proper **bounds checking**
  - sometimes it is not obvious how to do it
  - programmers are bound to forget it
- None of the existing countermeasures fully stop the problem of buffer overflows

➔ Are there any *system* defenses that can help?

Ok, what about avoiding some functions?

- Many C library functions do not check input size
  - gets is inherently unsafe and was recently removed
  - strcpy is safe as long as you're an educated C programmer ☺
    - If you pass a string of unknown length to it, or some data chunk that does not end with \0, you clearly made a mistake
    - Similar considerations apply for instance to strcat
- Variants of such functions explicitly check input size
  - strncpy is often suggested as replacement for strcpy...
    - However, did you know that no \0 character is appended at the end of destination if source is longer than the specified number of bytes?
    - Things like strlcpy and strlcat are better still!
  - snprintf is typically better than sprintf (at least in C99)
  - fgets is surely better than gets, but consider getline as well

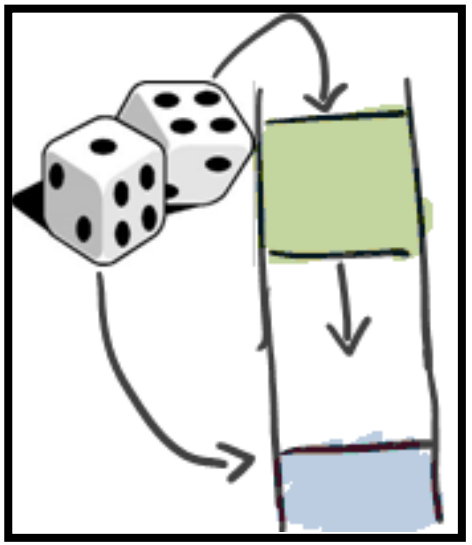
How do we stop the attacks?

A variety of tricks in combination:

NX bit

Canaries

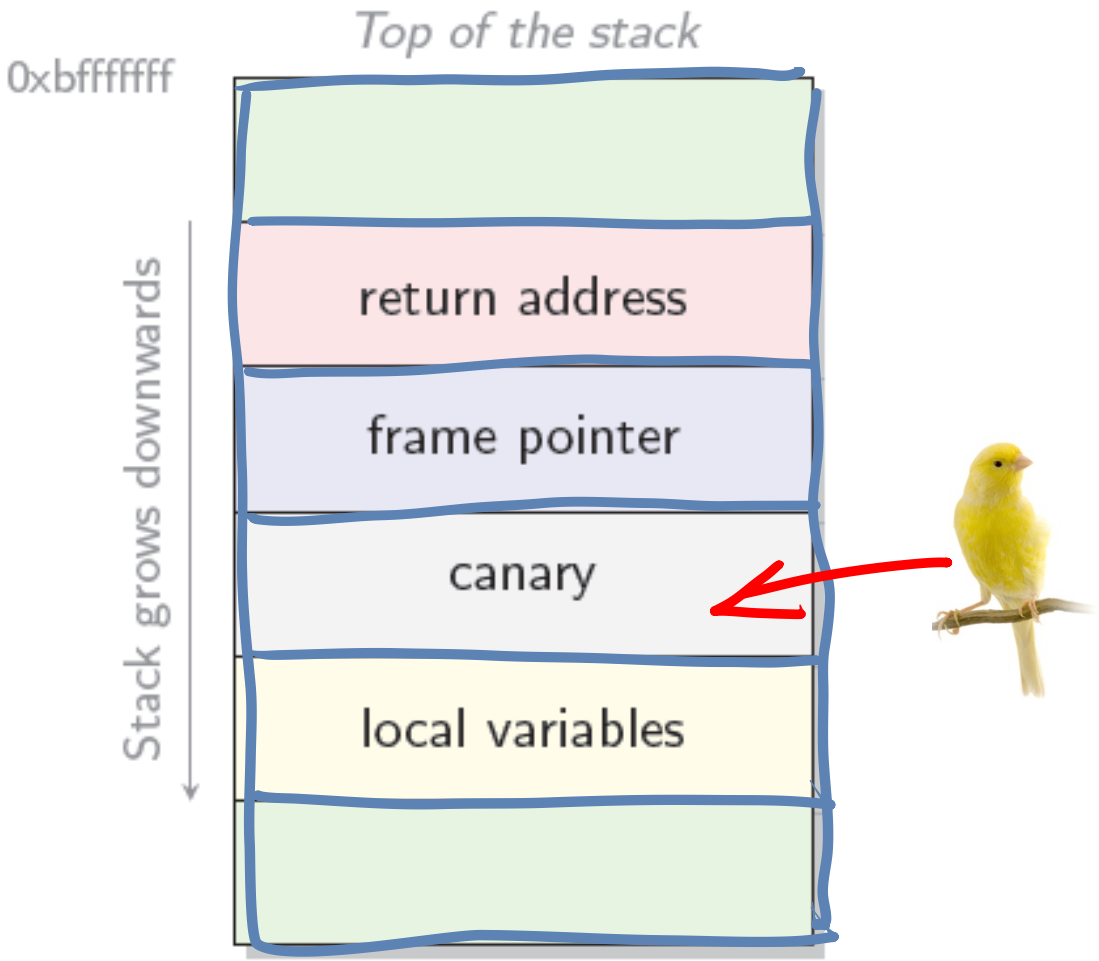
ASLR



Compiler-level techniques
Canaries

- Goal: make sure we detect overflow of return address
  - The functions' prologues insert a *canary* on the stack
  - The canary is a 32-bit value inserted between the return address and local variables
- Types of canaries:
  1. Terminator
  2. Random
  3. Random XOR
- The epilogue checks if the canary has been altered
- Drawback: requires recompilation

Canaries

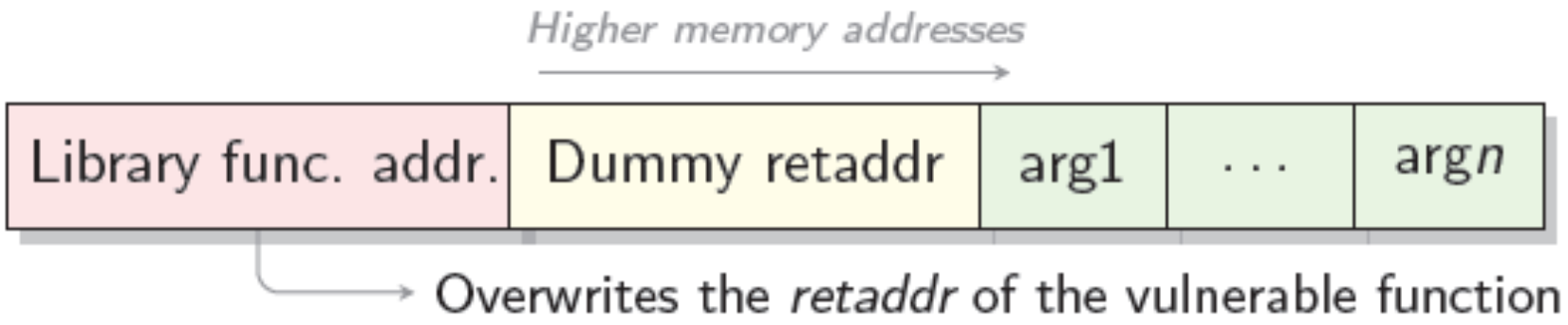

Top of the stack
0xbfffffff
Stack grows downwards
return address
frame pointer
canary
local variables

System-level techniques
DEP / NX bit / W⊕X

- Idea: separate executable memory locations from writable ones
  - A memory page cannot be both writable and executable at the same time
- “Data Execution Prevention (DEP)”

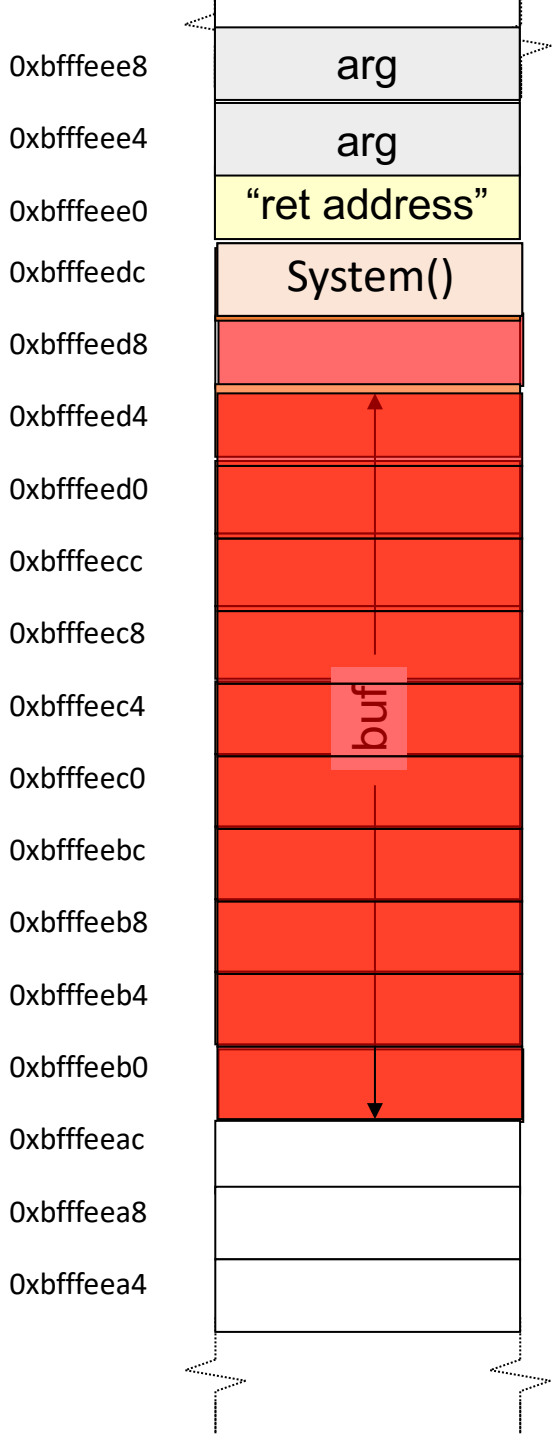
Bypassing W⊕X

- Return into libc
- Three assumptions:
  - We can manipulate a code pointer (e.g., return address)
  - The stack is writable
  - We know the address of a “suitable” library function (e.g., system() can spawn a shell to run a command)



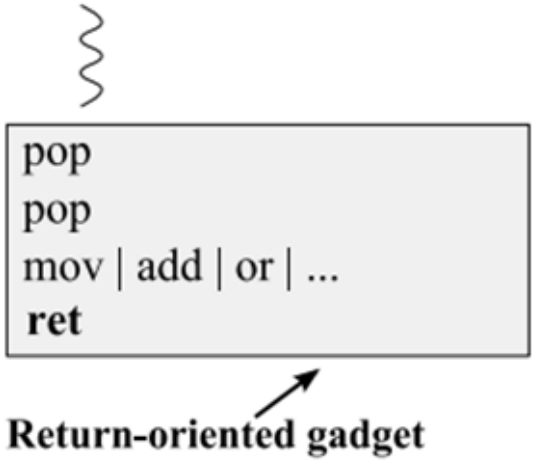
Stack

- Why the “ret address”?
- What could we do with it?



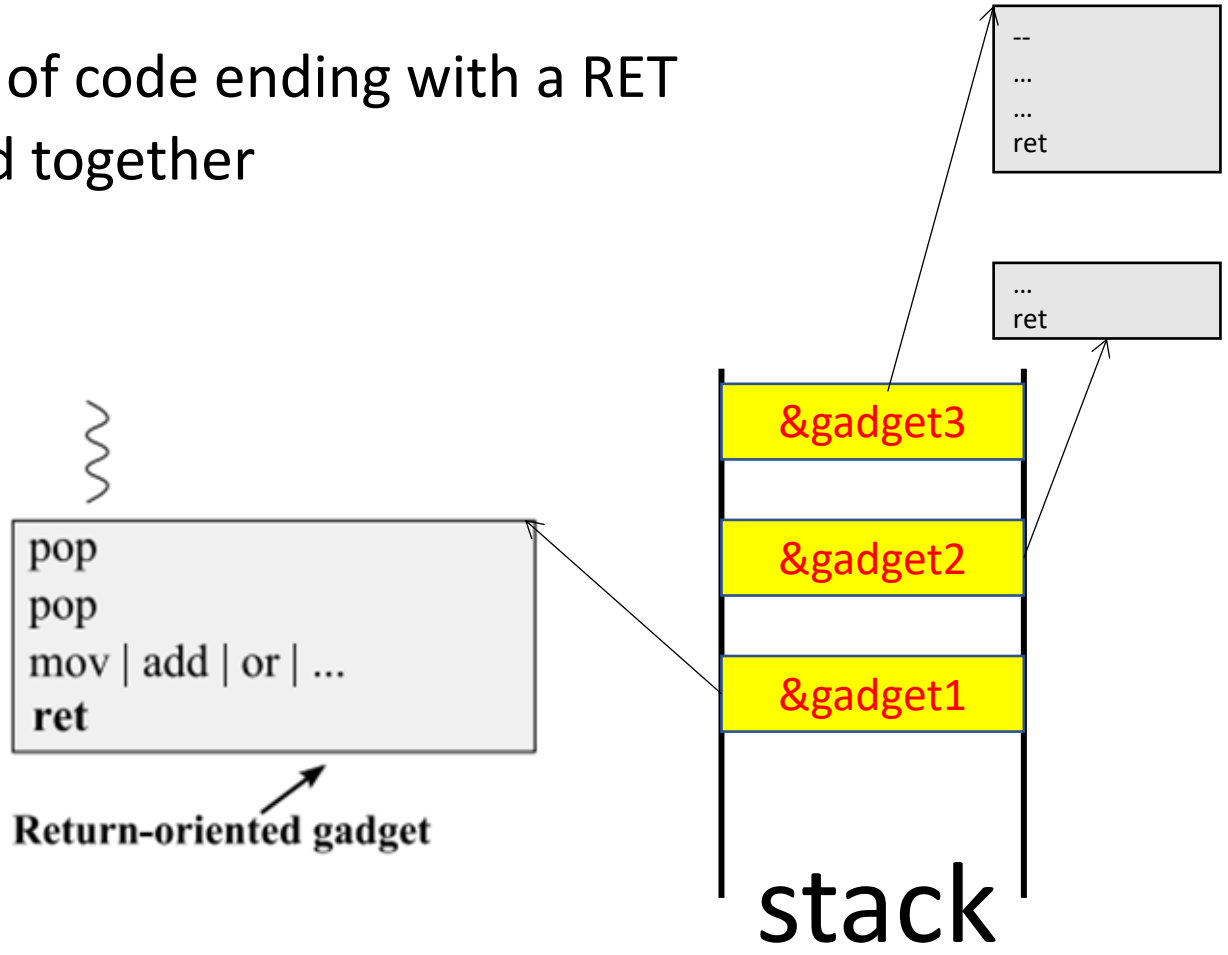
Return Oriented Programming

- ROP chains:
  - Small snippets of code ending with a RET
  - Can be chained together



Return Oriented Programming

- ROP chains:
  - Small snippets of code ending with a RET
  - Can be chained together



System-level techniques
Address Space Layout Randomization

Idea behind ASLR:

- Re-arrange the position of key data areas randomly (stack, .data, .text, shared libraries, . . .)
- Buffer overflow: the attacker does not know the address of the shellcode
- Return-into-libc: the attacker can't predict the address of the library function
- Implementations: Linux kernel > 2.6.11, Windows Vista, . . .

Problems with ASLR

- 32-bit implementations use few randomization bits
- ASLR typically applied to library code only
  - Position-independent code could be used for .text
- An attacker can still exploit non-randomized areas
- Information leaks (e.g., format bug)
  - Memory disclosure bugs defeat ASLR!
  - In current ASLR implementations, knowing the address of one function pointer is enough to determine the addresses of other functions, as they all are still at the same relative offset from the start address of code

Extra resources

- The 1OKStudents Challenge (slides + videos)
- Secure Programming classes by David Aspinall
- «Buffer Overflow Attack» chapter from «Computer Security: A Hands-on Approach» by Wenliang Du
- ROP Emporium (aka learn advanced ROP from the basics)
- Secure Programming HOWTO by David Wheeler
- Smashing the stack for fun and profit by Aleph One
- Buffer overflows by Erik Poll
- The Frame Pointer Overwrite by klog
- C and C++ vulnerability exploits and countermeasures by Frank Piessens