February 22, 2016

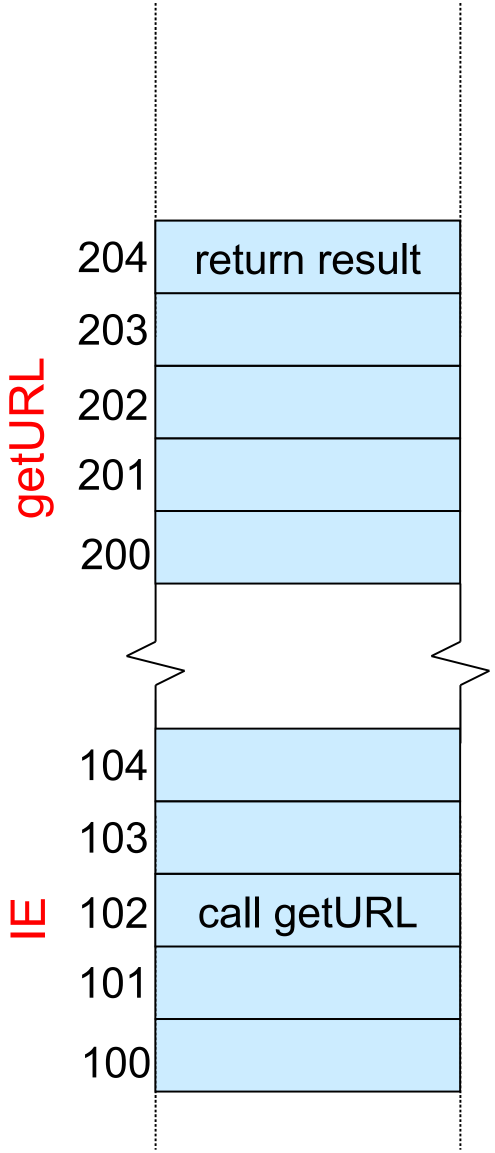
Buffer Overflows

Computer Security 2 (GA02)

Emiliano De Cristofaro, Gianluca Stringhini

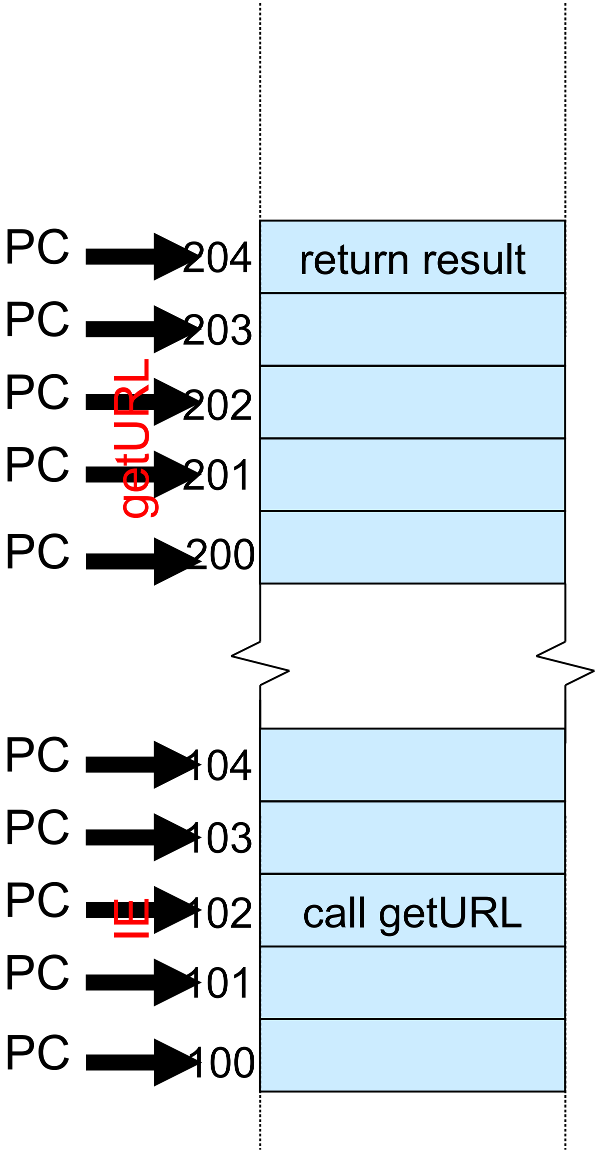
Software

- sequence of instructions in memory
- logically divided in functions that call each other
  - function 'IE' calls function 'getURL' to read the corresponding page
- in CPU, the program counter contains the address in memory of the next instruction to execute
  - normally this is the next address (instruction 100 is followed by instruction 101, etc)
  - ***not so with function call***



Software

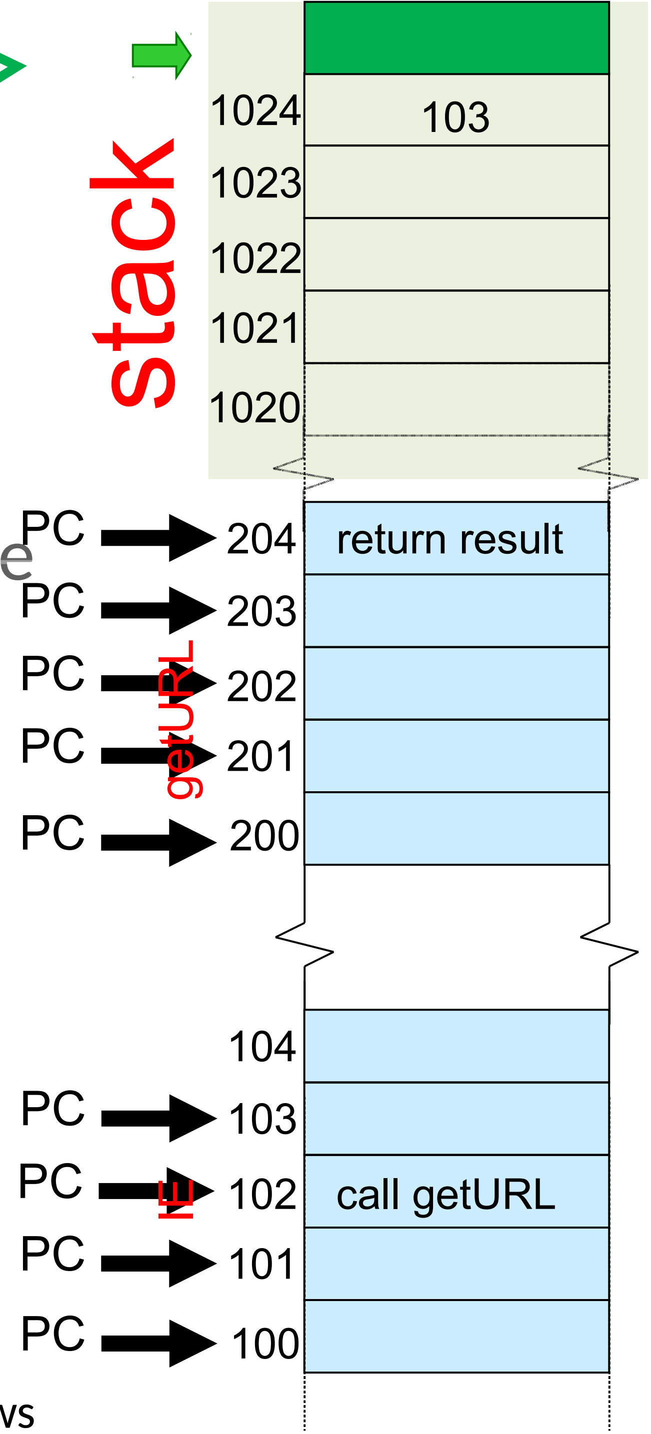
- sequence of instructions in memory
- logically divided in functions that call each other
  - function ‘IE’ calls function ‘getURL’ to read the corresponding page
- in CPU, the program counter contains the address in memory of the next instruction to execute
  - normally this is the next address (instruction 100 is followed by instruction 101, etc)
  - ***not so with function call***



Software

- so how does our CPU know where to return?
  - it keeps administration
  - on a ‘stack’

stack pointer (SP)
➔Points to last added entry on the stack

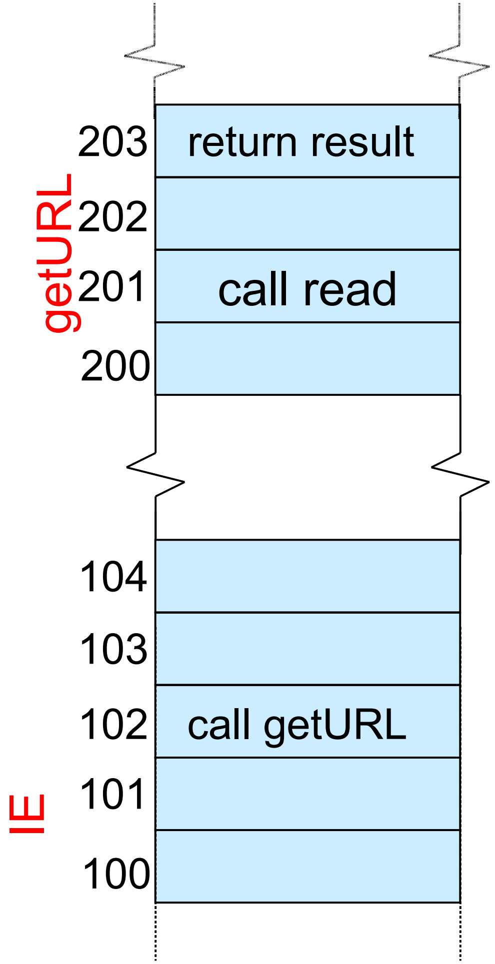


Real Functions

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

Real Functions

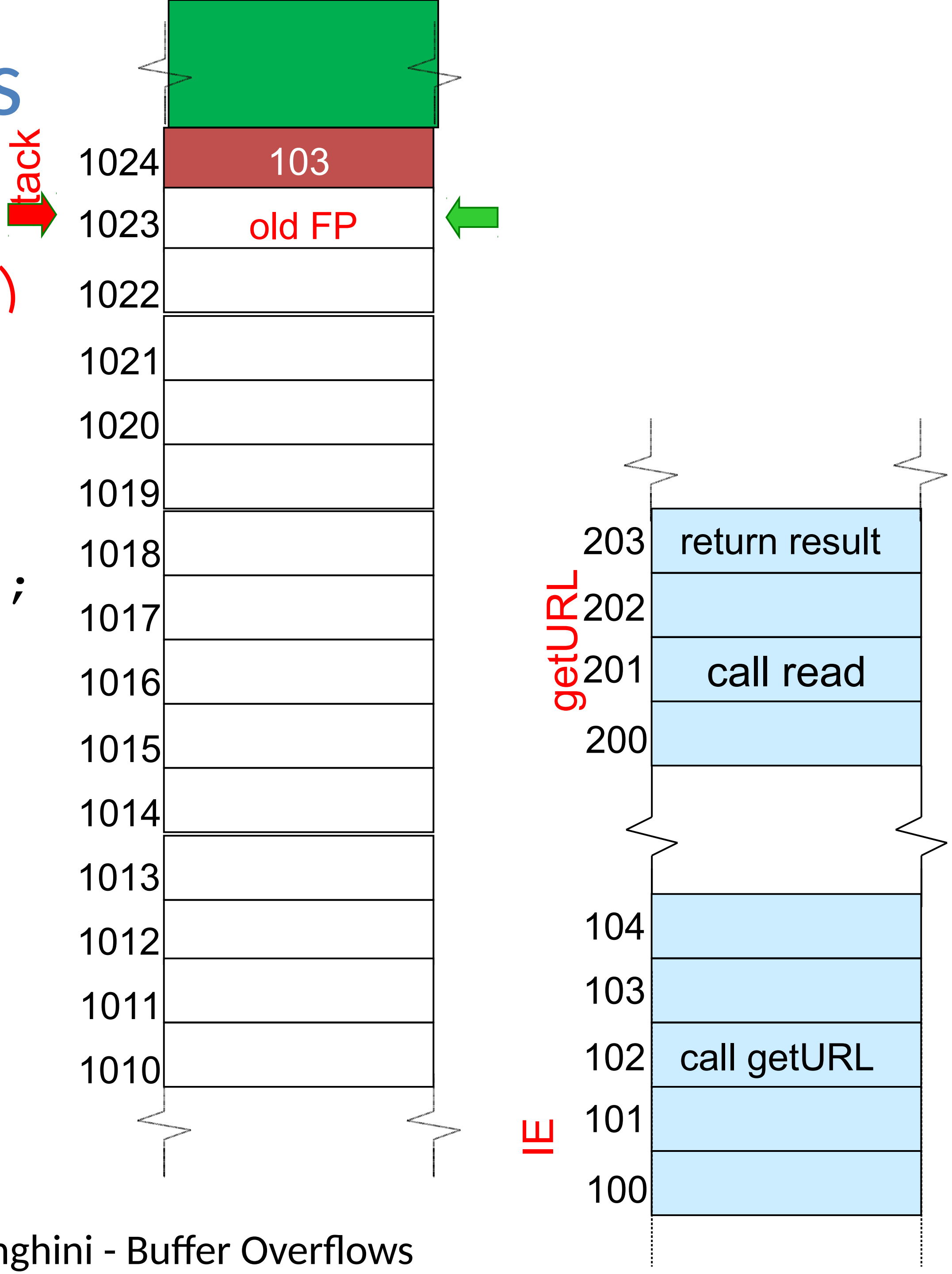
```
getURL ()
{
  char buf[10];
  read(keyboard,buf,64);
  get_webpage (buf);
}
IE ()
{
  getURL ();
}
```



Real Functions

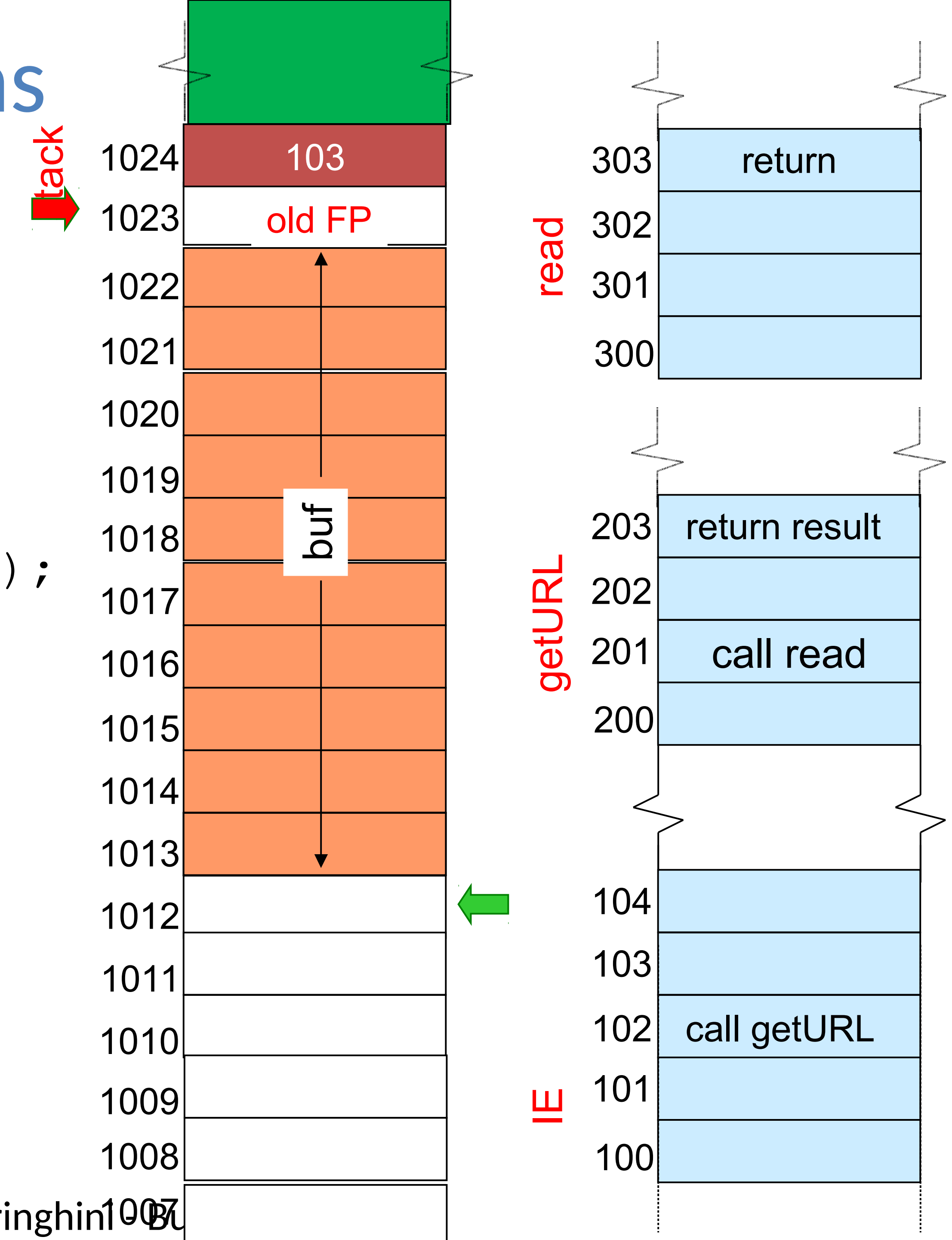
frame pointer (FP)
→Points to start of this function's stack frame

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



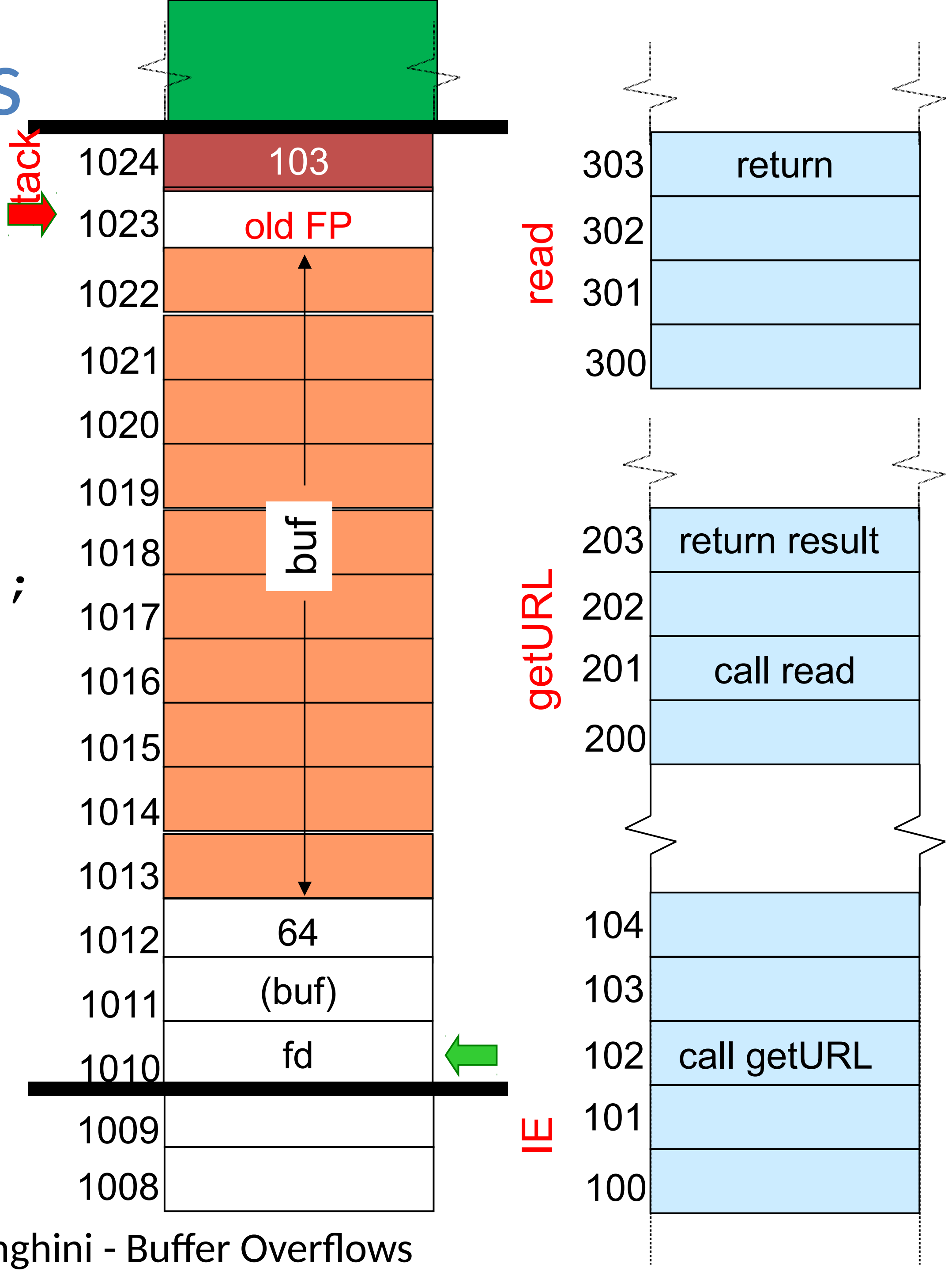
Real Functions

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



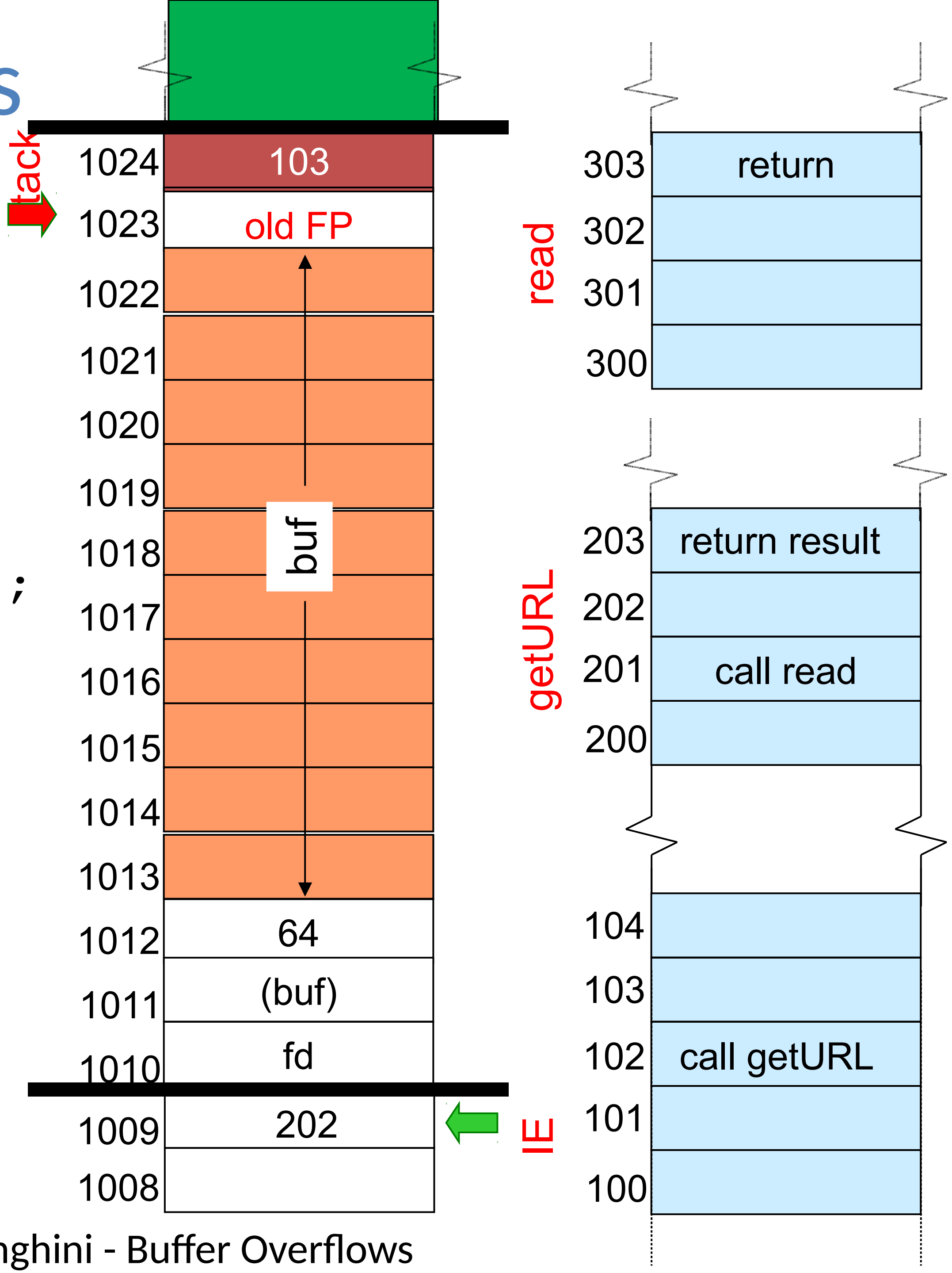
Real Functions

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



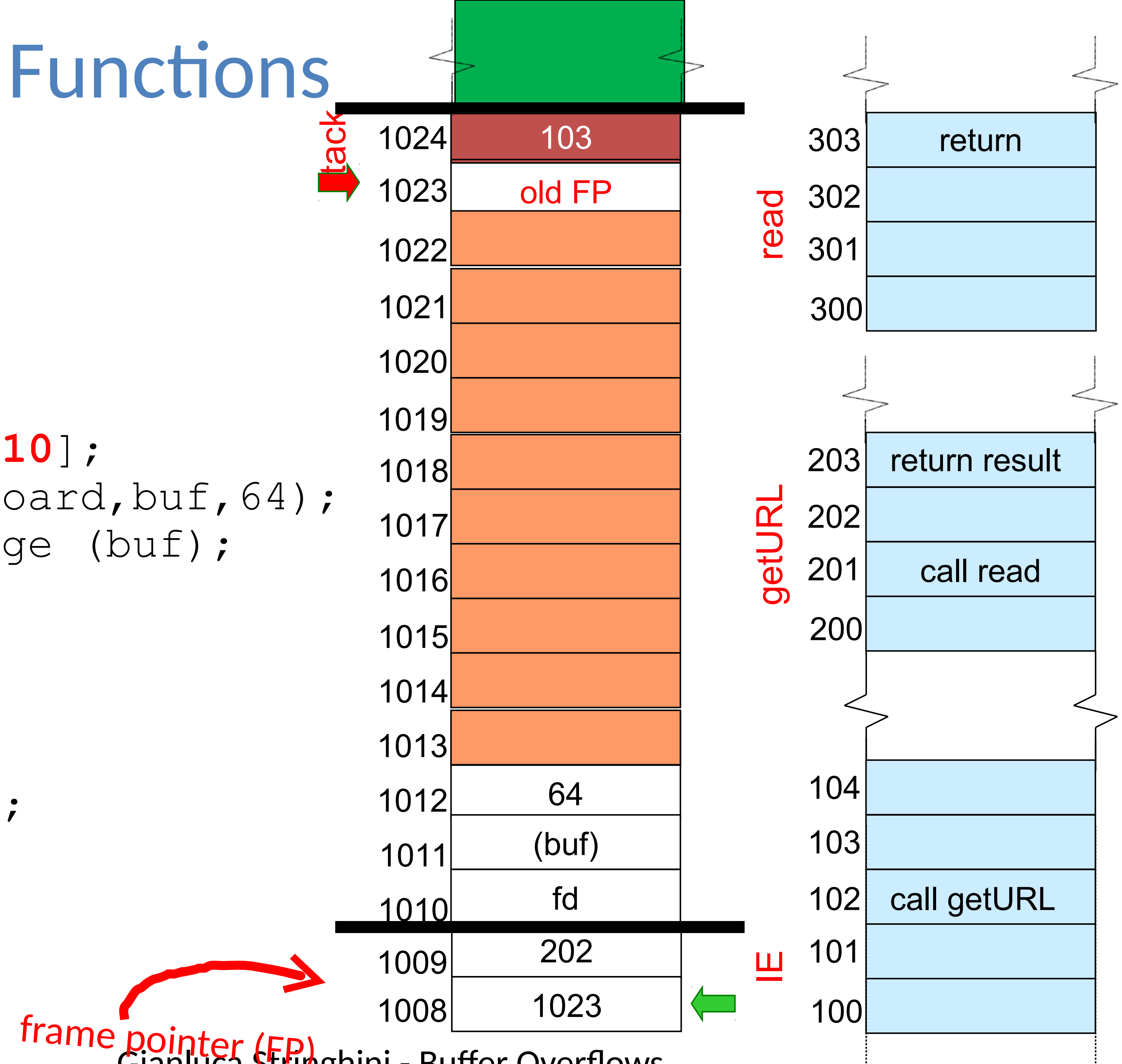
Real Functions

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



Real Functions

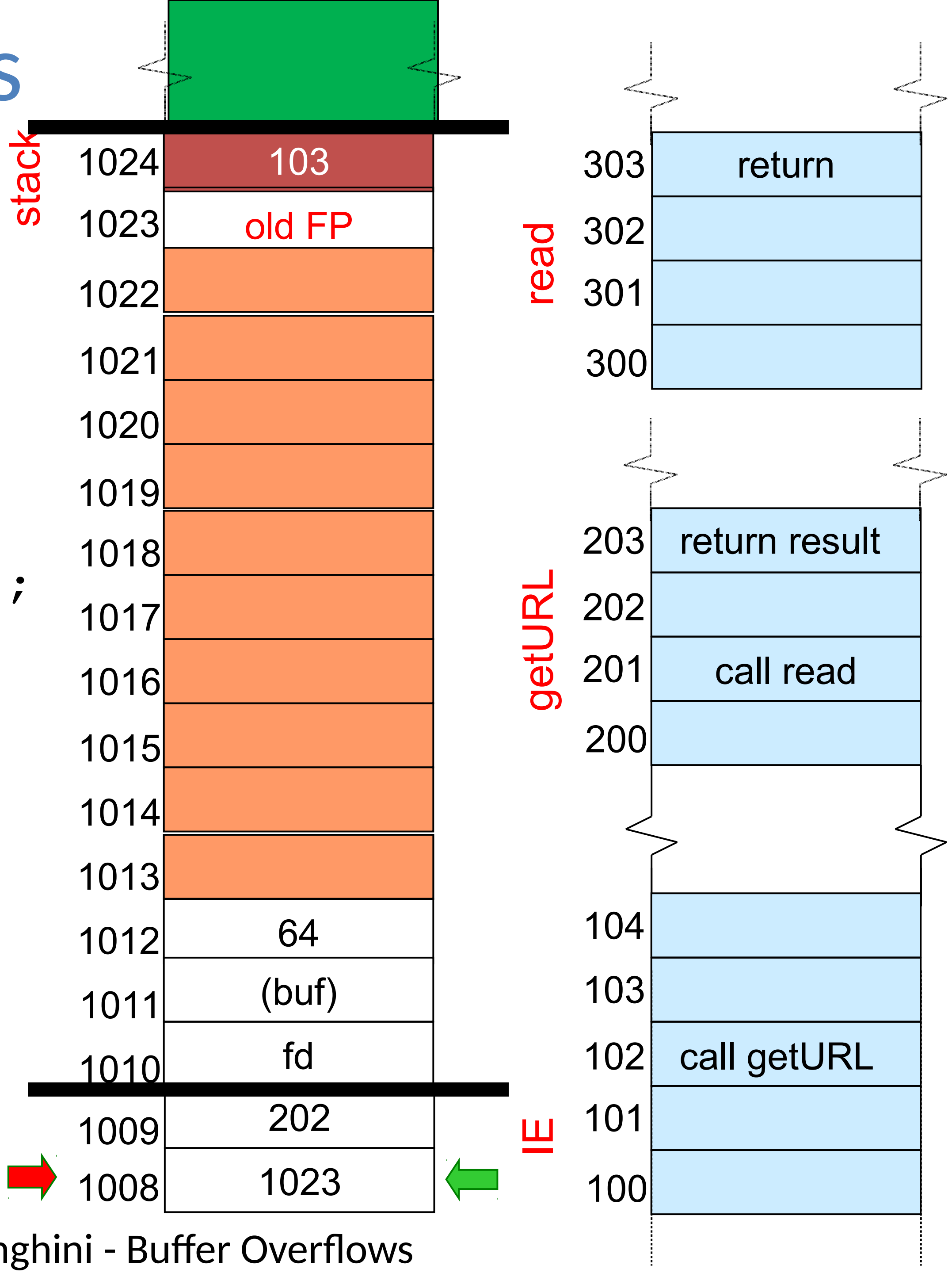
```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Real Functions

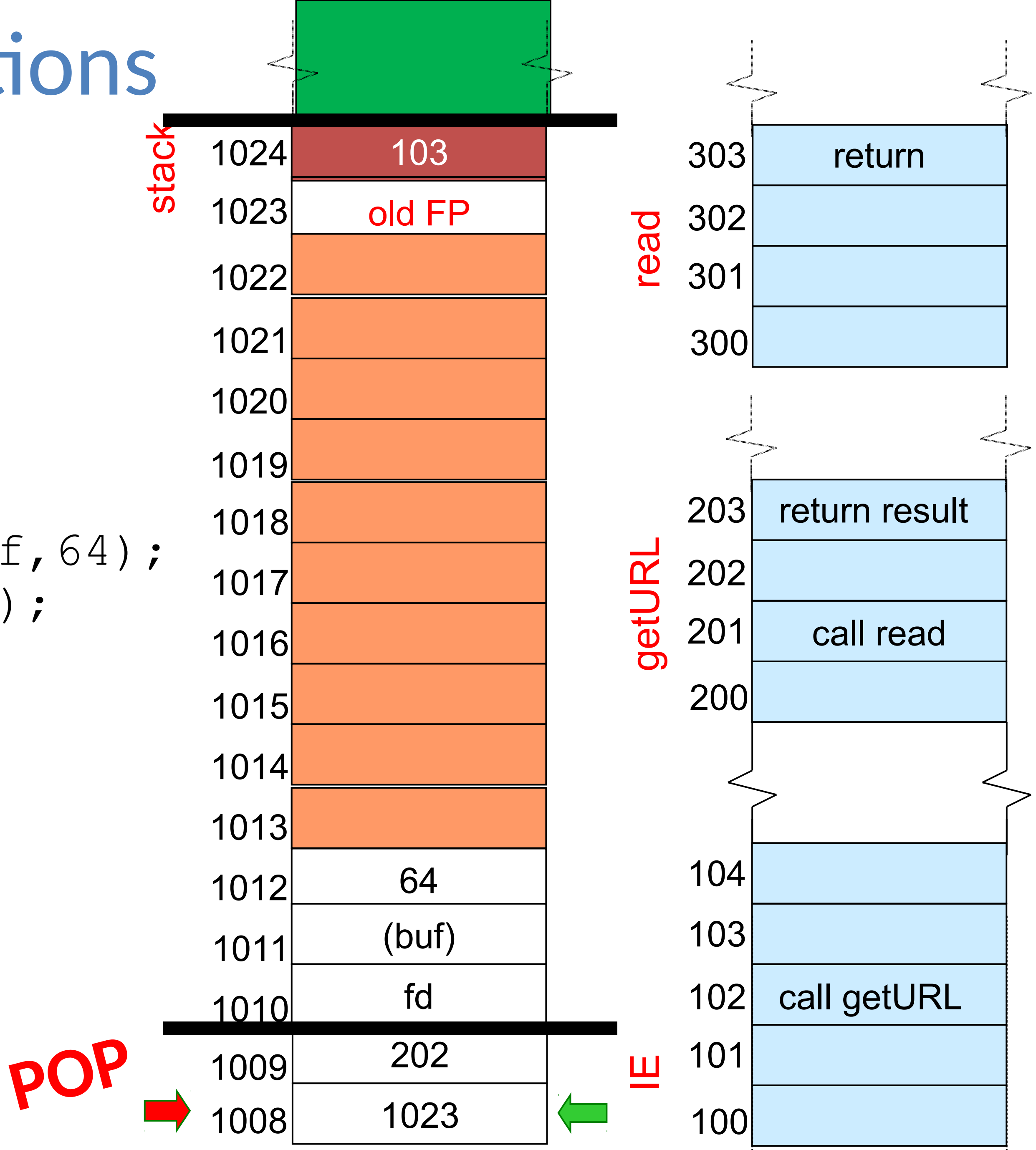
```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```

on "return from read"



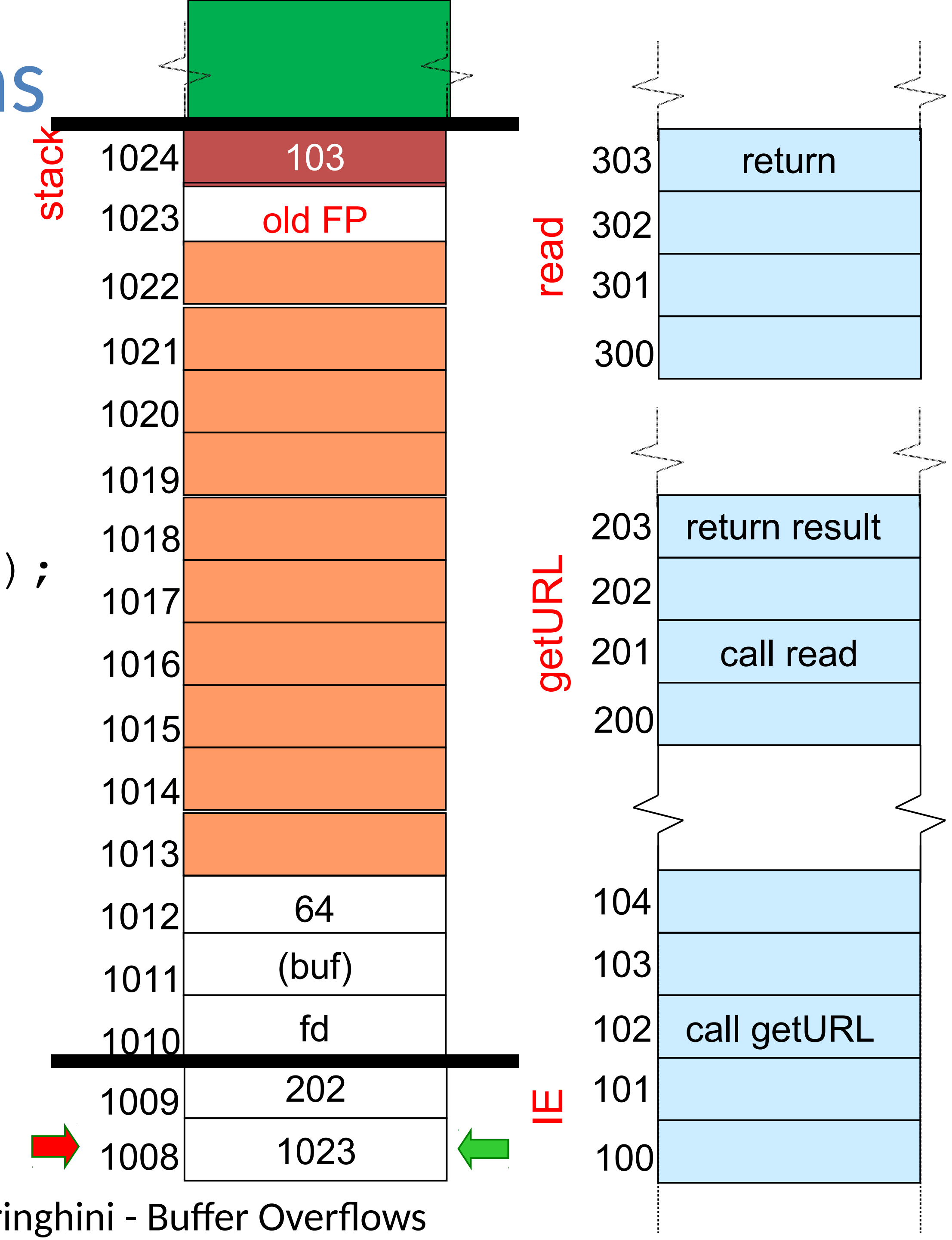
Real Functions

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



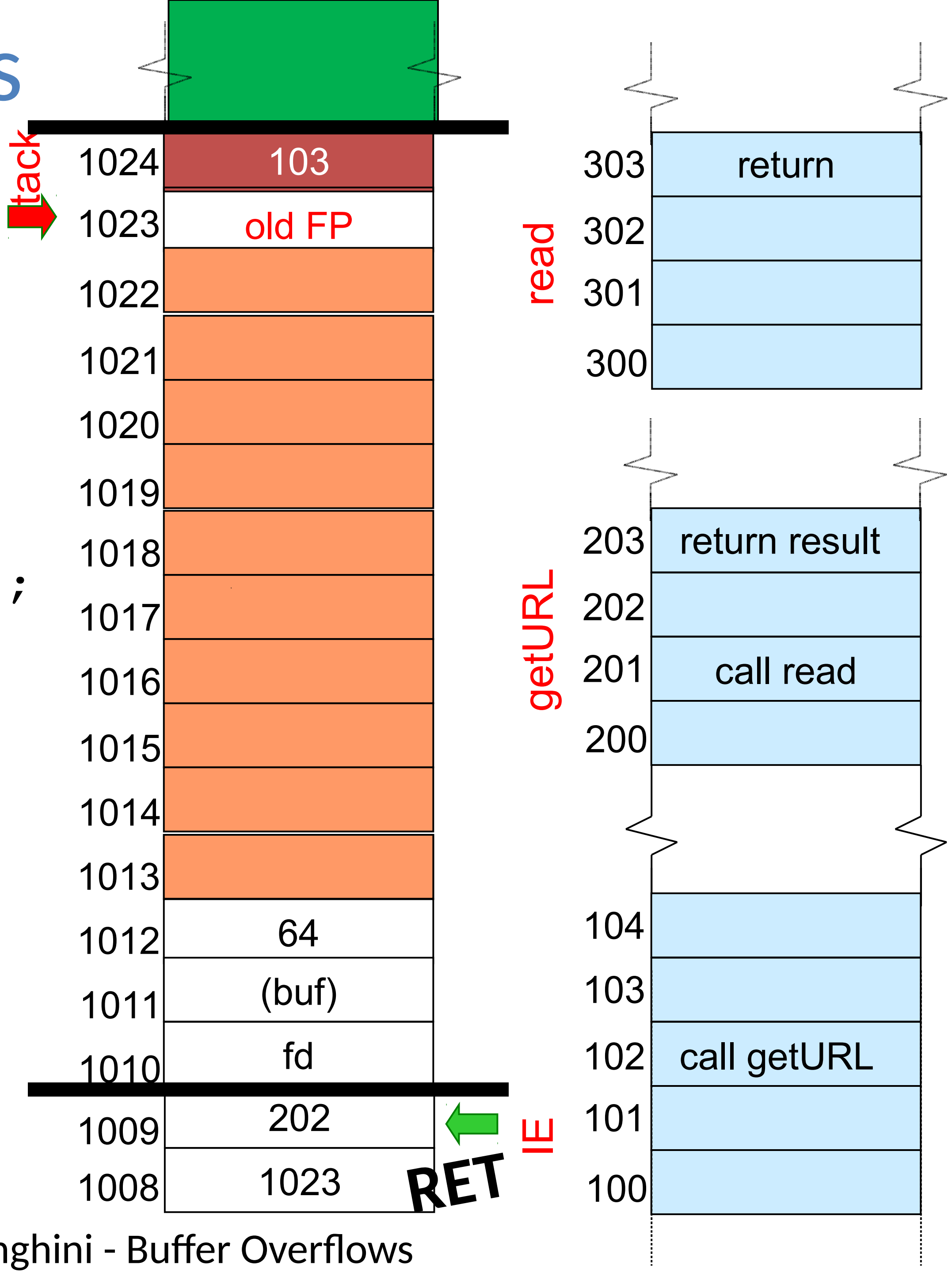
Real Functions

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



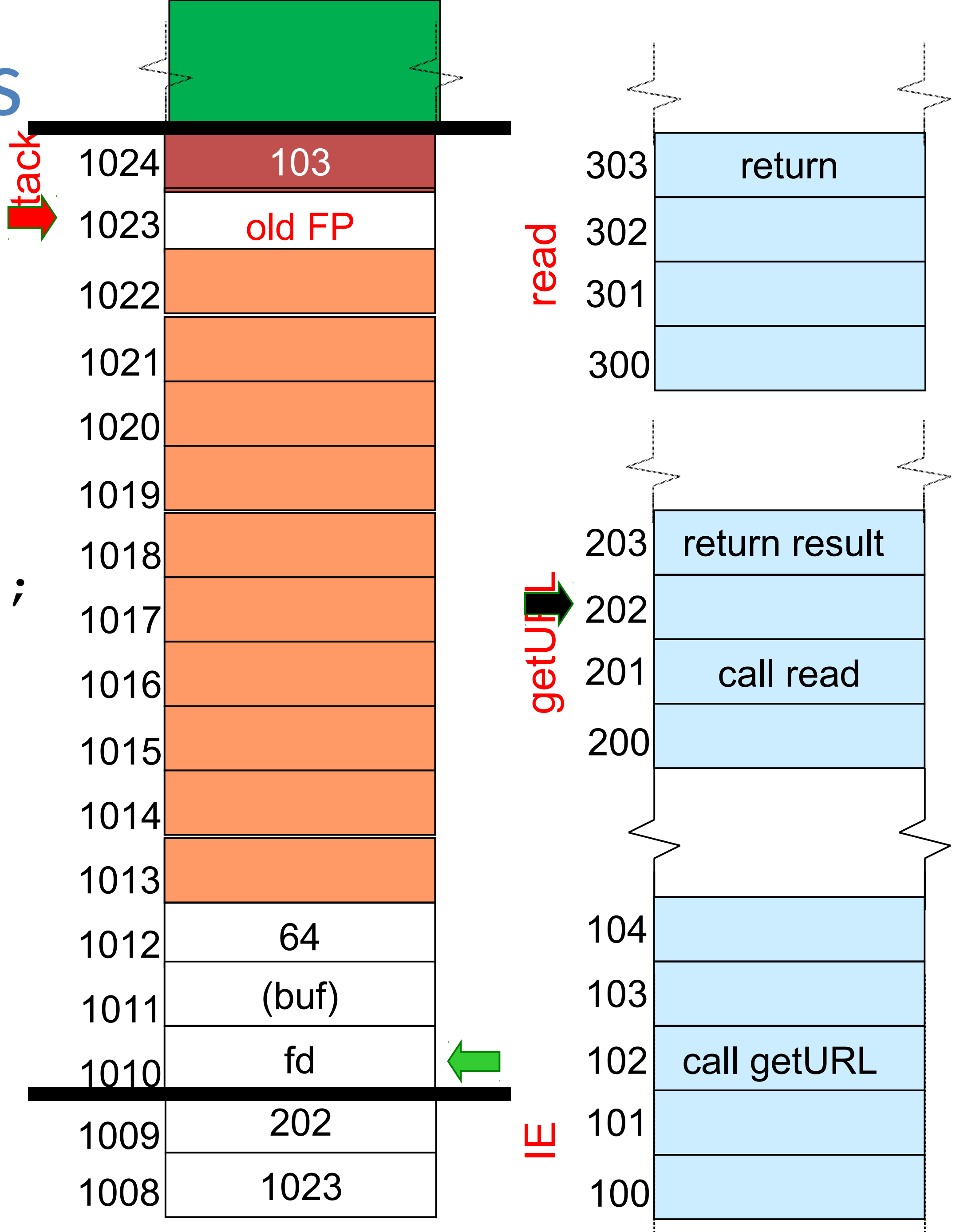
Real Functions

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



Real Functions

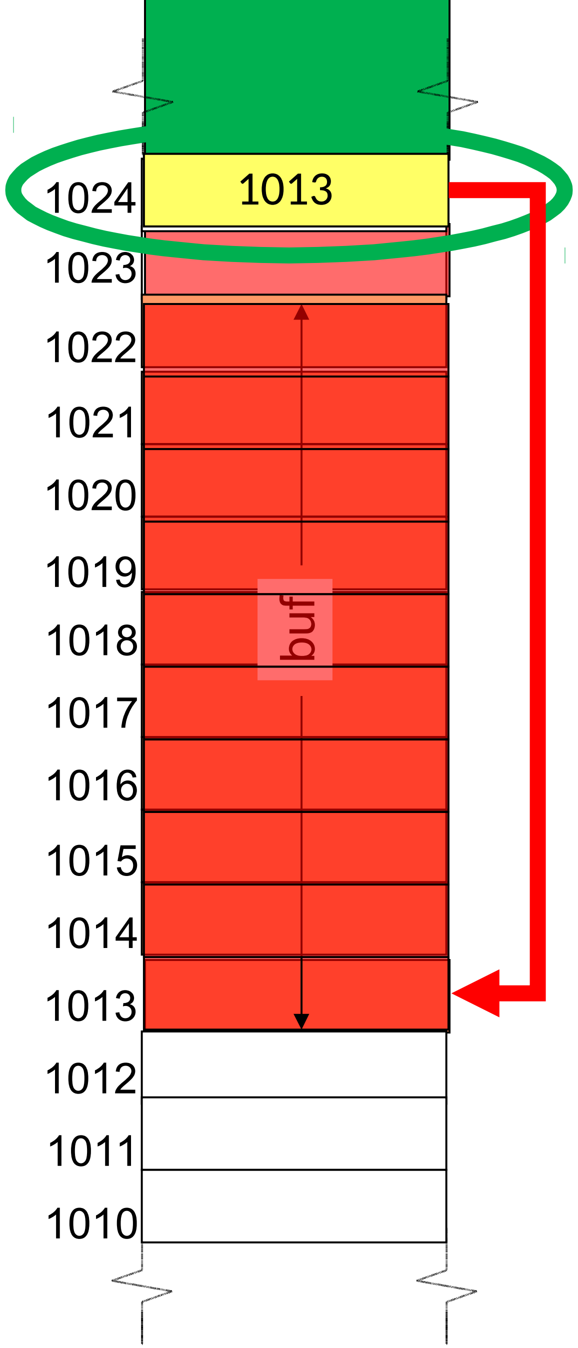
```
getURL ()
{
   char buf[10];
   read(keyboard,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



Where is the vulnerability?

Exploit

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



Caveats

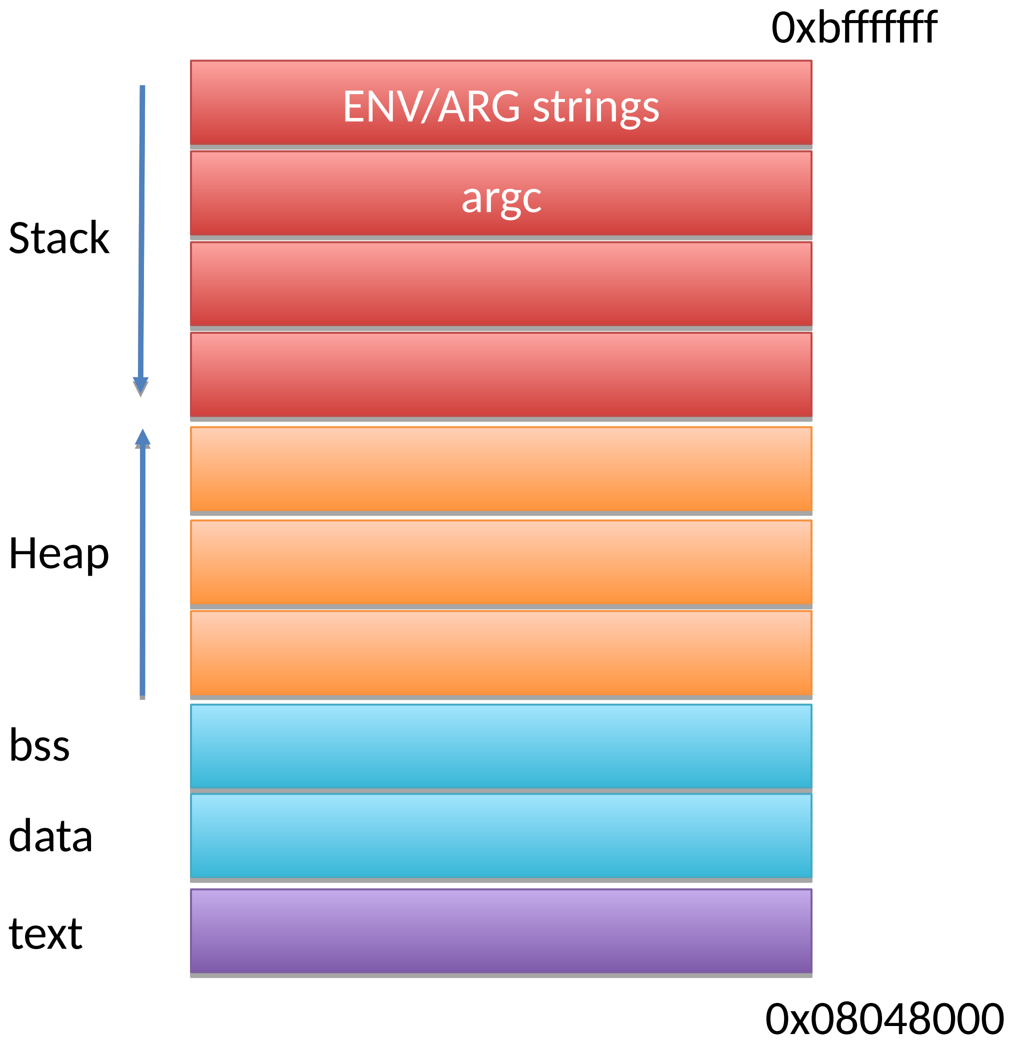
If you make the program crash, exploitation fails

By exploiting the stack, you may overwrite other things

- Other variables that are also on the stack
- Other addresses
- Etc.

Let’s make it real

Memory Layout of a Process



Real Code

Addresses are written in hexadecimal

Consider the assembly code for IE()

Beginning of function

Push Frame Pointer on the stack

Store Stack Pointer in Frame Pointer

```
0x08048428 <+0>:   push   %ebp
0x08048429 <+1>:   mov    %esp,%ebp
0x0804842b <+3>:   call   0x8048404 <getURL>
0x08048430 <+8>:   pop    %ebp
0x08048431 <+9>:   ret
```

Call getURL()

Return

Restore Frame Pointer

$$2^0 \cdot 1 + 2^4 \cdot 3 + 2^8 \cdot 4 + 2^{12} \cdot 8 + 2^{16} \cdot 4 + 2^{20} \cdot 0 + 2^{24} \cdot 8 + 2^{28} \cdot 0$$

Two characters denote the value of a byte

Similarly

Same function preamble

Make space for buffer and the 3 parameters to read()

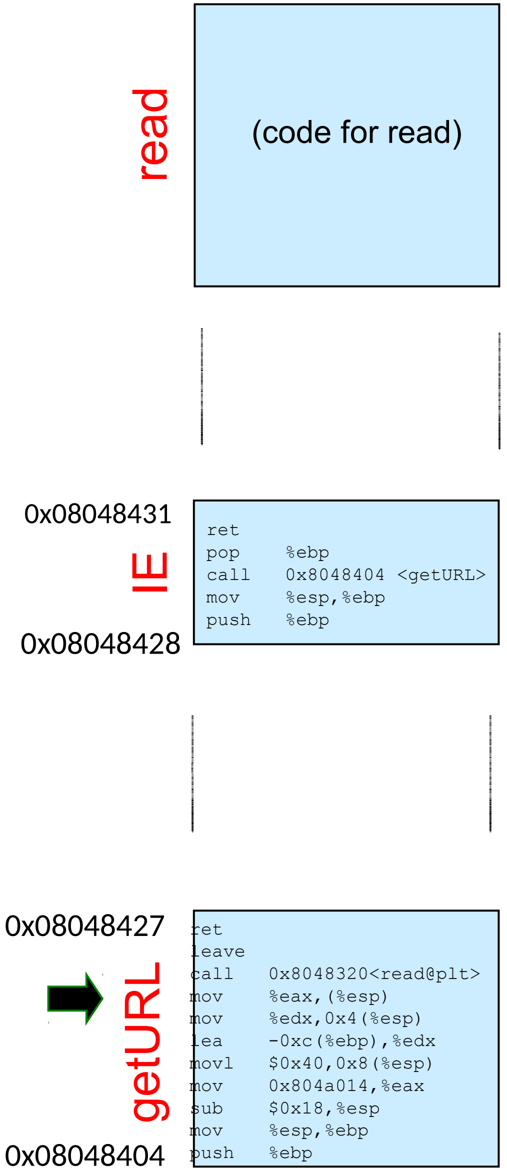
The assembly code for getURL():

```
0x08048404 <+0>:  push   %ebp
0x08048405 <+1>:  mov    %esp,%ebp
0x08048407 <+3>:  sub    $0x18,%esp
0x0804840a <+6>:  mov    0x804a014,%eax
0x0804840f <+11>:    movl   $0x40,0x8(%esp)
0x08048417 <+19>:    lea    -0xc(%ebp),%edx
0x0804841a <+22>:    mov    %edx,0x4(%esp)
0x0804841e <+26>:    mov    %eax,(%esp)
0x08048421 <+29>:    call   0x8048320 <read@plt>
0x08048426 <+34>:    leave
0x08048427 <+35>:    ret
```

Call to read()

Putting the pieces together

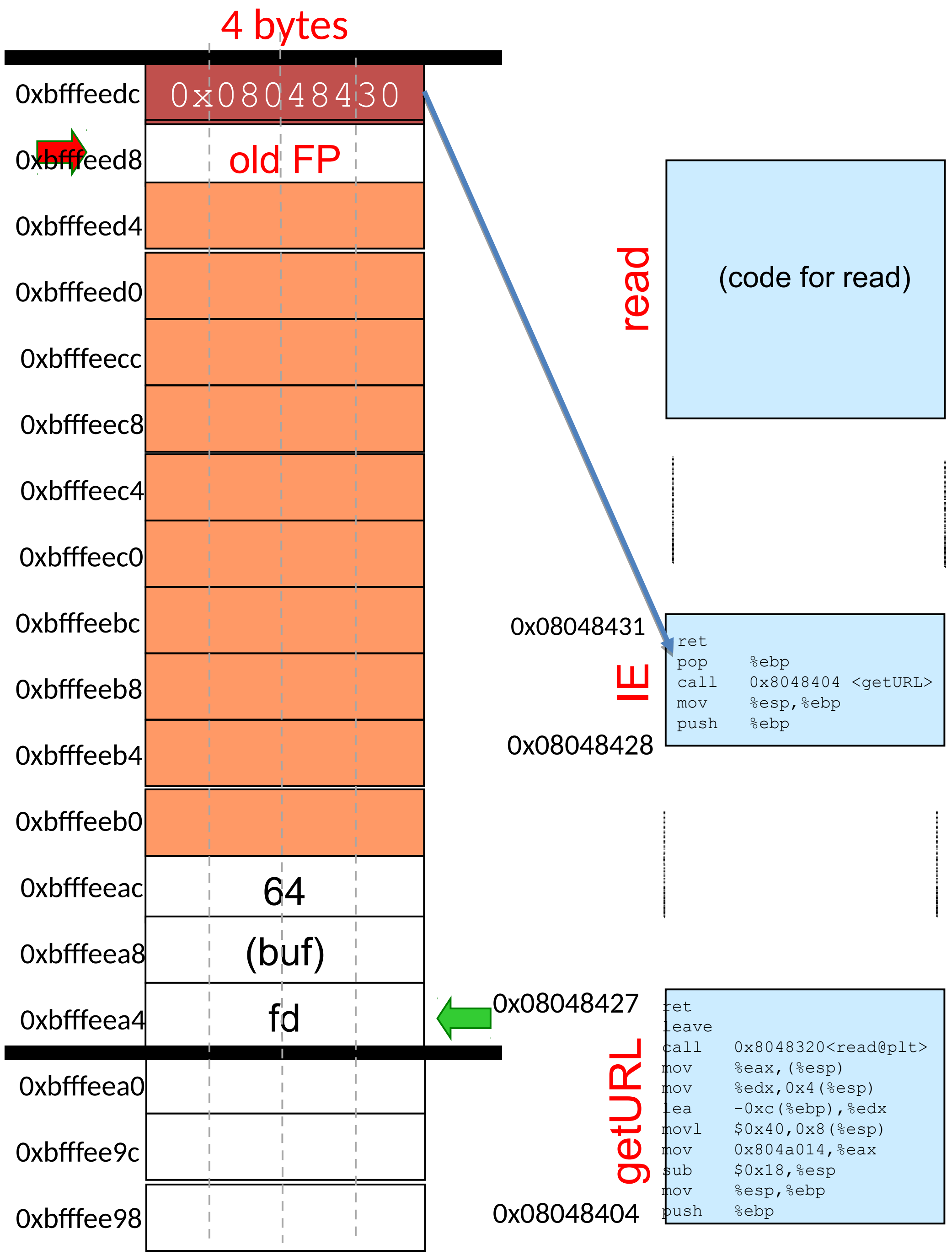
```
getURL ()
{
   char buf[40];
   read(stdin,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



What about the stack?

getURL() is about to call read()

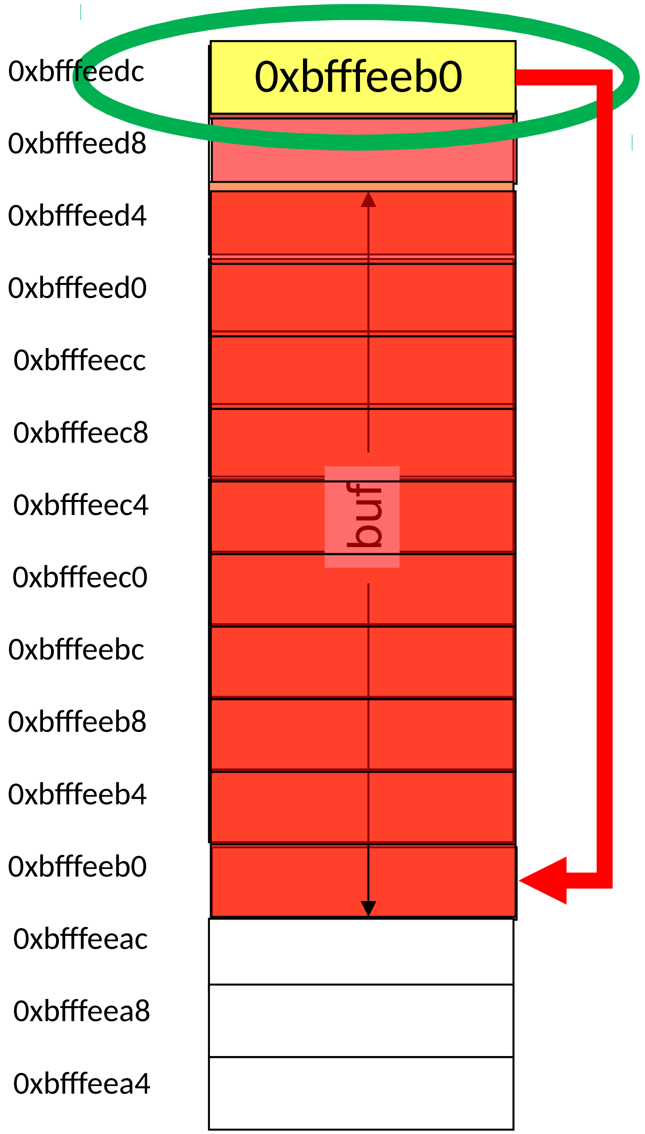
```
getURL ()
{
   char buf[40];
   read(stdin,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



And now the exploit

Exploit

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



That’s it, really

- 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 code!

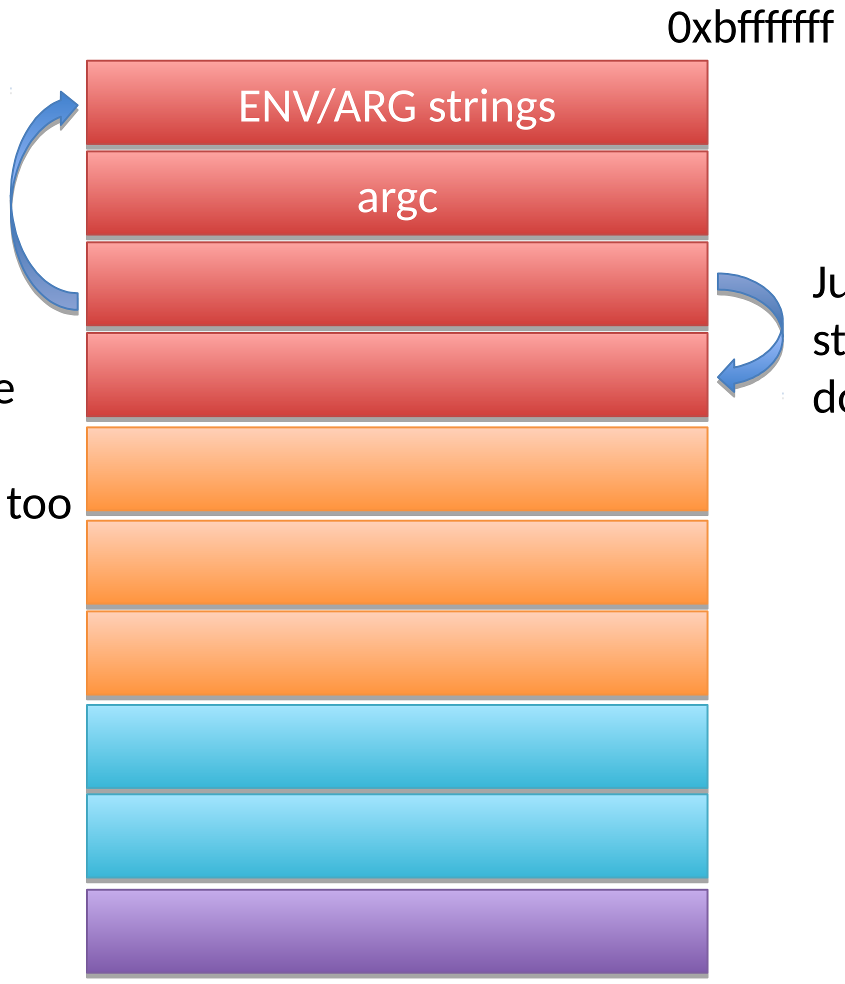
Exploit against non control data

```
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”);
}
```

The **authorized** variable is on the stack, therefore **we can overwrite it to true**!

Possible return targets on the stack



Jump to environment variable – if we called the program ourselves, we control the environment too

Jump to address in the stack (what we have done in the example)

What the attacker needs to do is putting code of his/her choice in one of these two locations and have the program jump to it

Typical injection vector

Address of vector – 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 (NOPs). It is used to ease the exploitation: the attacker can jump anywhere in this region, and will eventually reach the shellcode (optional)

Typical shellcodes

By injecting code of their choice, the attacker can make the program do whatever they want

Typical shellcodes include

- Opening a shell ("/bin/sh") – good for local programs
- Binding a shell to a network port – good for remote programs
- Opening a **reverse shell** that connects to the attacker computer – useful in the presence of firewalls or NATs
- Read a password file and email it to the attacker through the **mail command**

How do you create the vector?

NOP sled	shellcode	address of vector

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

   python -c 'print "\x90"*100'

3. Add the address

   `0xbfffeeb0 (keep in mind that x86 is a little endian architecture)`

```
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
        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

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

        int $0x80

string_addr:
        call mystart
        .asciz "/bin/shNAAAABBBB"
```

In reality, things are more complicated

If **strcpy()** is used to overflow the buffer, it will stop when it encounters a null byte (‘`\0`’). 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

There are tools (**metasploit**, **shellnoob**) that can create custom shellcode and avoid using forbidden characters

Countermeasures

How do we defend from buffer overflows?

Best defense: **bound checking**

Example: using **strncpy()** instead of **strcpy()**

Many modern programming languages are memory safe (e.g., Java) – no buffer overflows possible

System level countermeasures are also provided

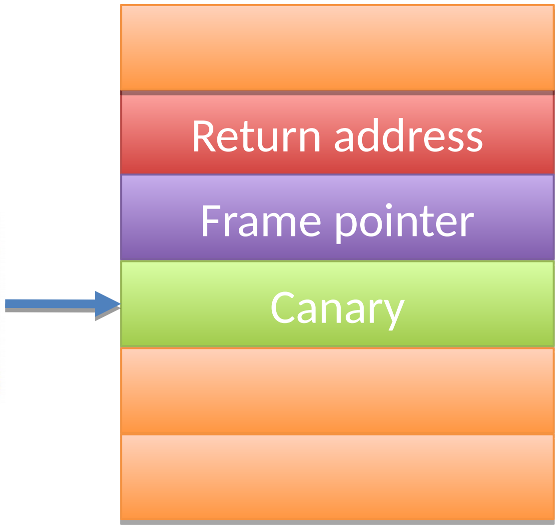
- Stack canaries
- Non-executable stack
- Address Space Layout Randomization (ASLR)

Stack Canaries

Compiler-based countermeasures: **requires recompilation of programs**

The function's prologue inserts a canary on the stack
The function's epilogue checks that the canary value is still there before returning

Does not prevent the overflow, just **makes it more difficult to overwrite the return address**



Types of canaries:

- **Terminator (deterministic)**
  Evasion: an attacker can insert the canary back as part of the attack
- **Random value**
  Evasion: an attacker can exploit a pointer used as a destination of a strcpy()-like function and overwrite the return address without touching the canary
- **XOR: random value ^ return address**
  Evasion: an attacker can still overwrite pointers in the function's stack frame

If the canary value doesn't match, an exception is raised – attackers could **overwrite the pointer to the exception handler (SEH)** and make it point to their own function!

Non-executable stack

The buffer overflow example that we used earlier stores executable code on the stack and jumps to it – we could force the stack to only contain data and not code

Data Execution Prevention (DEP): separate executable memory locations from writable ones – a memory page cannot be both writable and executable at the same time.

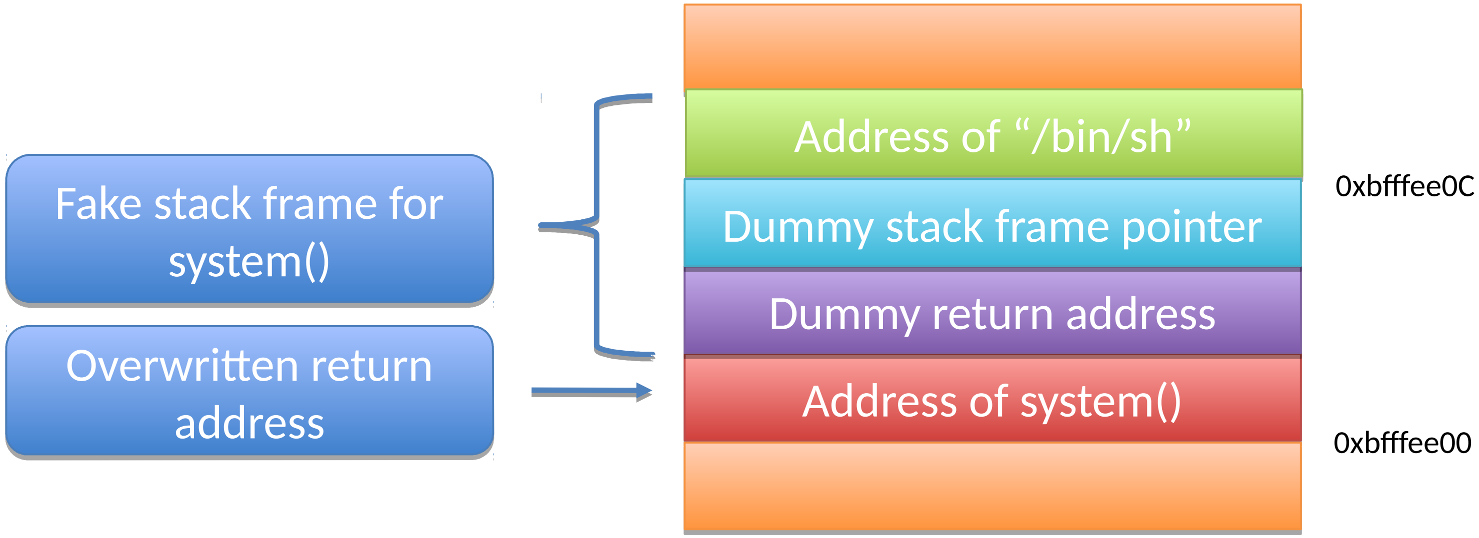
Modern computer architectures (x86_64) provide a **NX bit** in hardware to specify which memory pages contain non-executable data

DEP prevents attackers from storing the shellcode on the stack, does not prevent them from overwriting the return address → **return into libc** attacks are possible

Return into libc

UNIX programs are linked to binary functions to make them portable – in particular, if libc is included, a number of handy functions are available to an attacker (e.g., **system()**)

The attacker can then exploit a buffer overflow to set a fake function call frame for the **system()** function and have the program jump to **system()** after the function terminates
No executable code on the stack was used!



The attacker could **chain function calls**, by setting up multiple function frames in the stack and achieving more complex effects

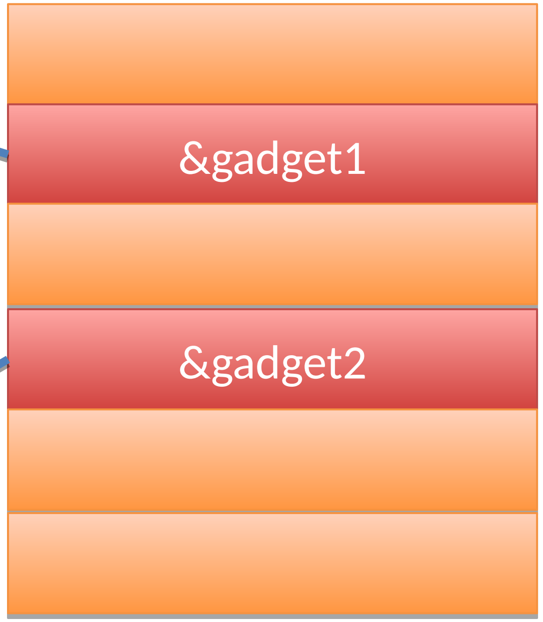
Return oriented programming (ROP)

Return-to-libc attacks are fairly versatile, but sometimes the functionalities needed by an attacker cannot be found in library functions

```
…
mov eax,ebx
ret
…
```

Return oriented programming allows the attacker to build their own functions, by repeatedly jumping to small snippets of code (**gadgets**) located near return instructions

```
…
inc eax
ret
…
```



It is surprisingly easy to find gadgets that allow an attacker to do whatever they want by chaining them

If you think about it, you don't need that many gadgets to achieve Turing completeness

Other locations that attackers can overwrite

Global Offset Table (GOT) – used by the linker to keep the addresses of the functions called in a program. The GOT is writable, therefore an attacker can overwrite the address to a function (for example `printf`) and have a function of his choice executed instead next time `printf` is called

The .dtors Section – contains code to be executed after the actual program code (stands for destructors). This section is writable too, and the attacker can use it to point to code that will be run when the program is terminating

Off by one overflows – programs start counting from 0, humans from 1 – this can cause code to allow writing **one byte too much** in memory!

Attackers can use this to overwrite the least significant byte in the previous contiguous memory address → overwrite the least significant byte of an address

Address space layout randomization (ASLR)

Idea: re-arrange the position of key data areas randomly (stack, code, shared libraries)

Effect on exploits:

- Buffer overflow: the attacker does not know the address of the shellcode
- Return-into-libc: the attacker cannot predict the address of the library function
- ROP: the attacker cannot predict the address of the code gadgets

Natively implemented on Linux > 2.6.11, Windows Vista or greater, …

Problems:

- 32-bit implementations use few randomization bits (bruteforce possible)
- If an attacker can read an address at runtime (through memory leakage) she can adjust her exploits accordingly

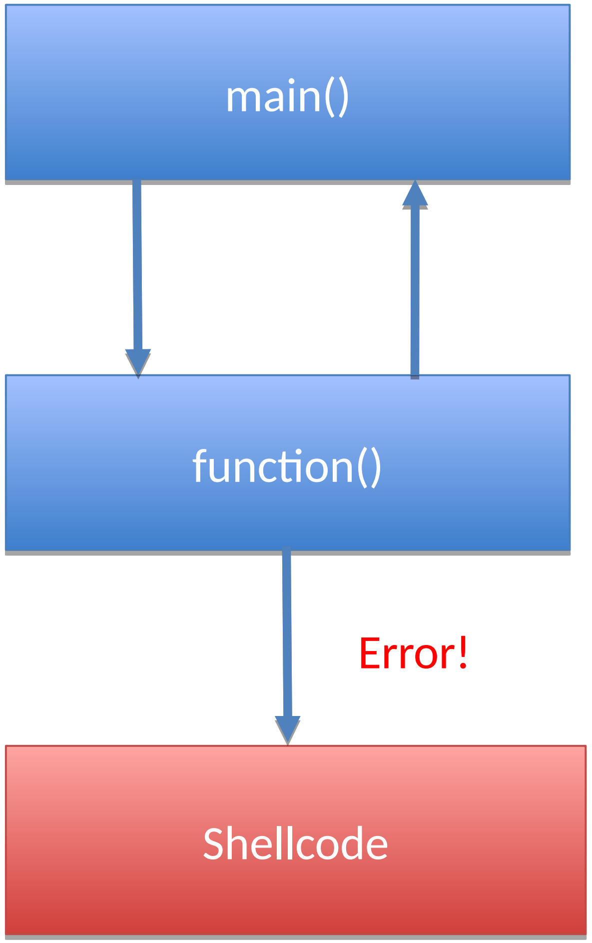
Control Flow Integrity

Can we make sure that a program follows the expected flow?

The Control Flow Integrity security policy dictates that **software execution must follow a path that is determined ahead of time**

Typical implementation:

- Static analysis to determine the expected control flow graph (CFG)
- Code rewriting to enforce the program flow + runtime checks



Control flow integrity: issues

CFI presents multiple challenges

- The control flow of a program is not deterministic – needs to take into account dynamic checks
- Static analysis is not perfect, especially in the case of pointer analysis and if no source code is present – needs to allow a somewhat "flexible" control flow
- Enforcing CFI has performance implications (5-10% usually)

Researchers keep breaking CFI schemes and coming up with improved ones

Some reading

"Smashing the Stack for Fun and Profit" by Aleph One

"The Geometry of Innocent Flesh on the Bone: Return-into-libc without functions calls (on the x86)" by H. Shacham

"Control Flow Integrity" by Abadi et al.