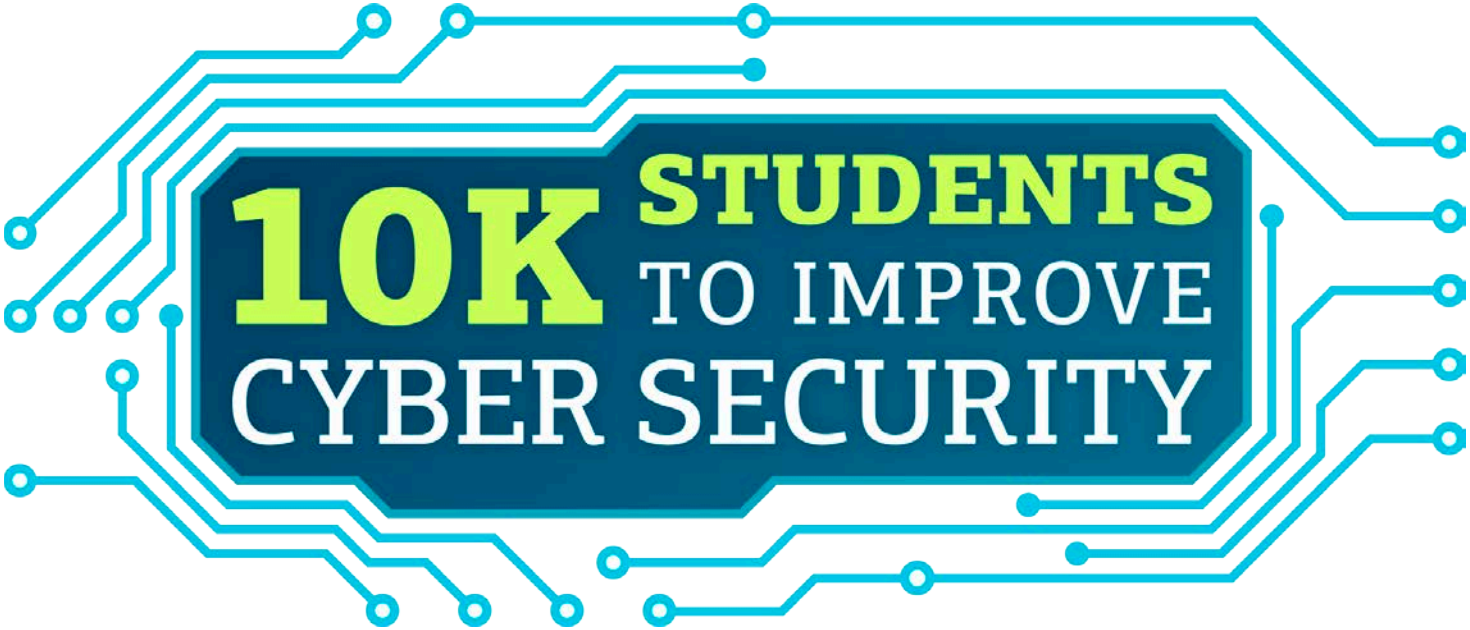


syssec

Part III
Counter measures

HOW DO WE STOP THE ATTACKS?

- The best defense is proper bounds checking
- but there are many C/C++ programmers and some are bound to forget

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

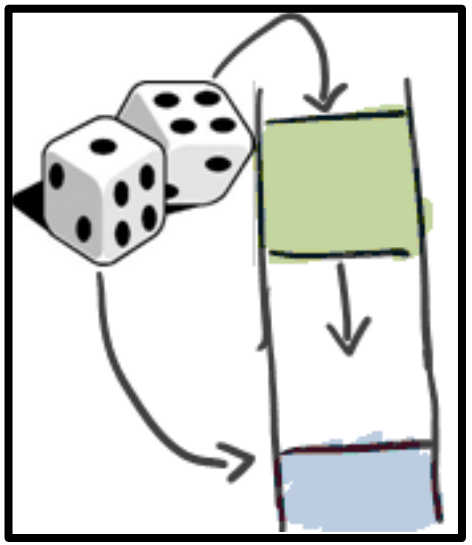
HOW DO WE STOP THE ATTACKS?

- A variety of tricks in combination

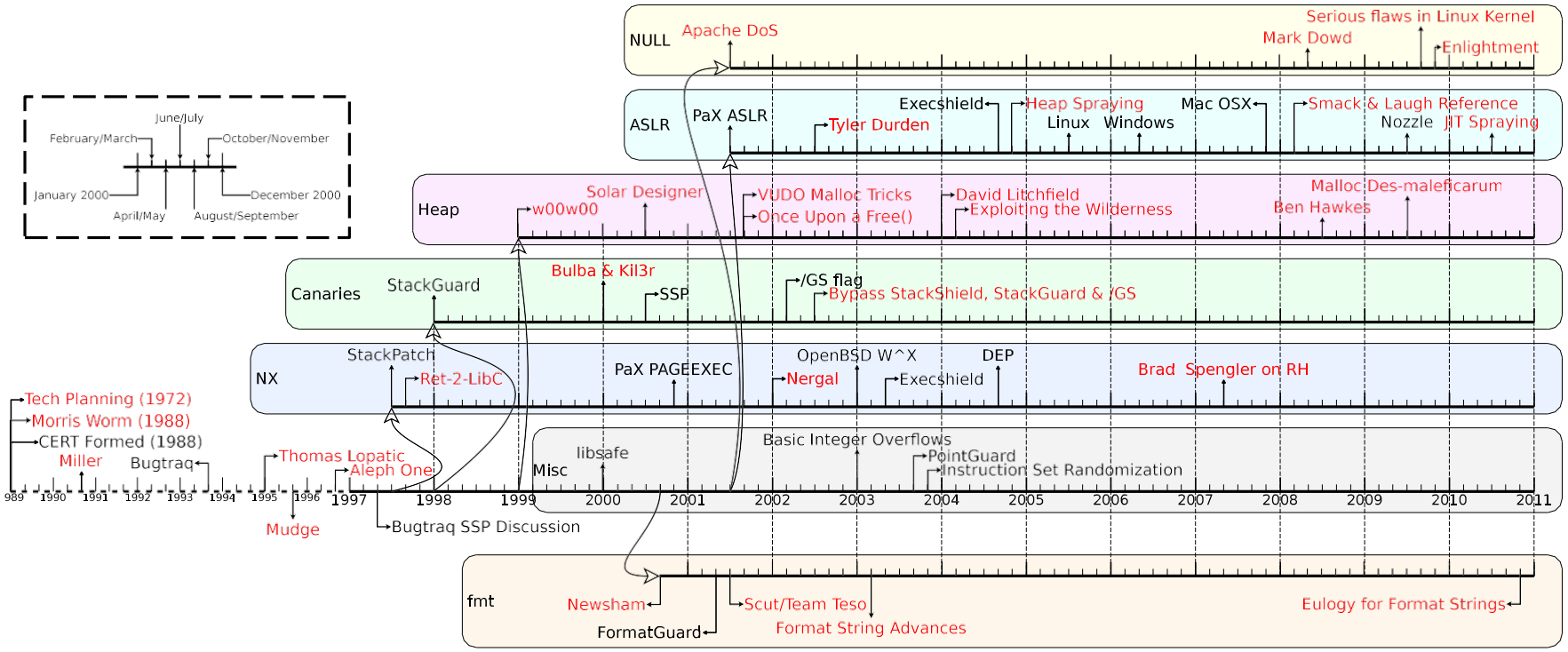
NX bit

Canaries

ASLR

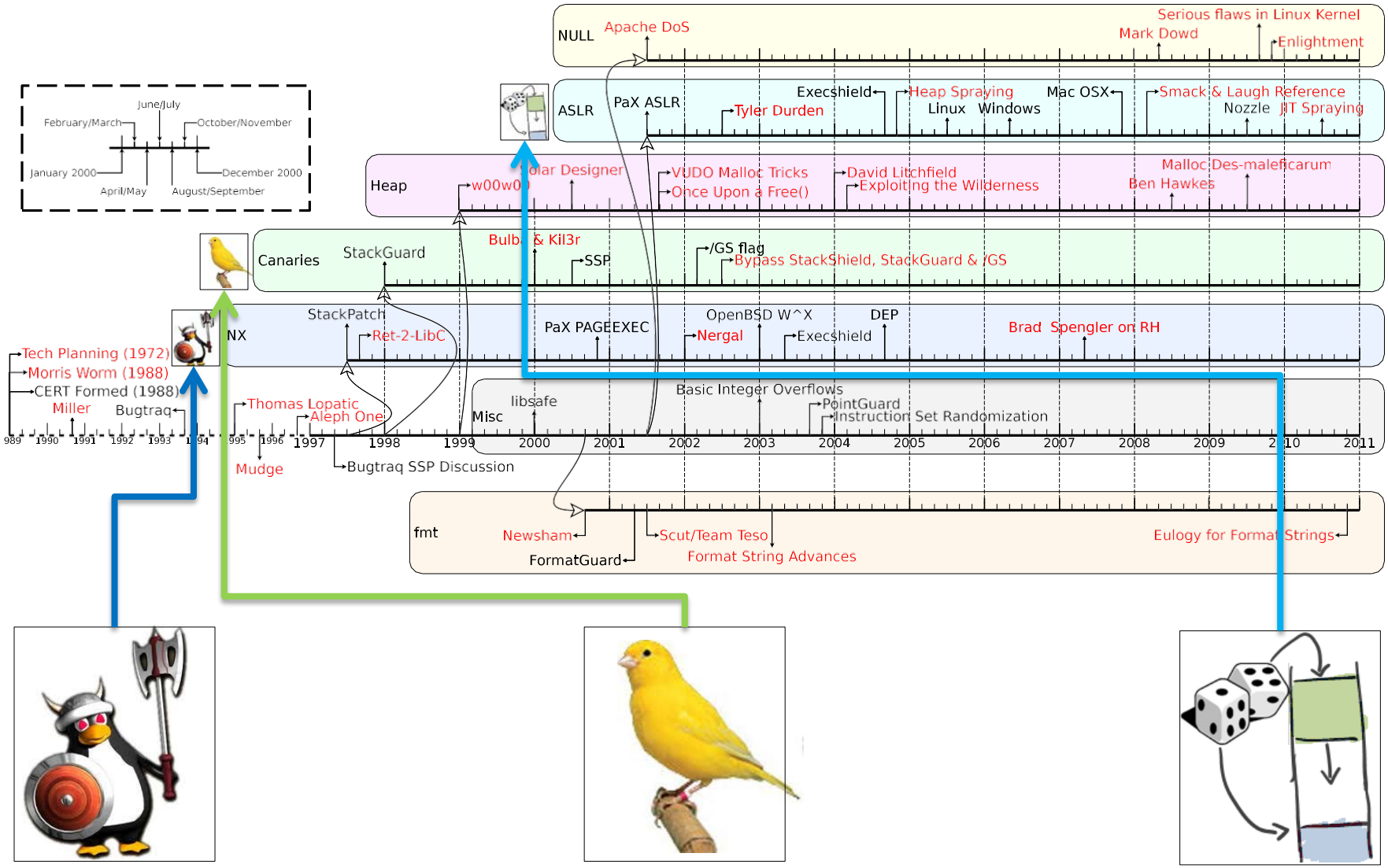


History of memory errors



https://www.isg.rhul.ac.uk/sullivan/pubs/tr/technicalreport-ir-cs-73.pdf

History of memory errors



https://www.isg.rhul.ac.uk/sullivan/pubs/tr/technicalreport-ir-cs-73.pdf

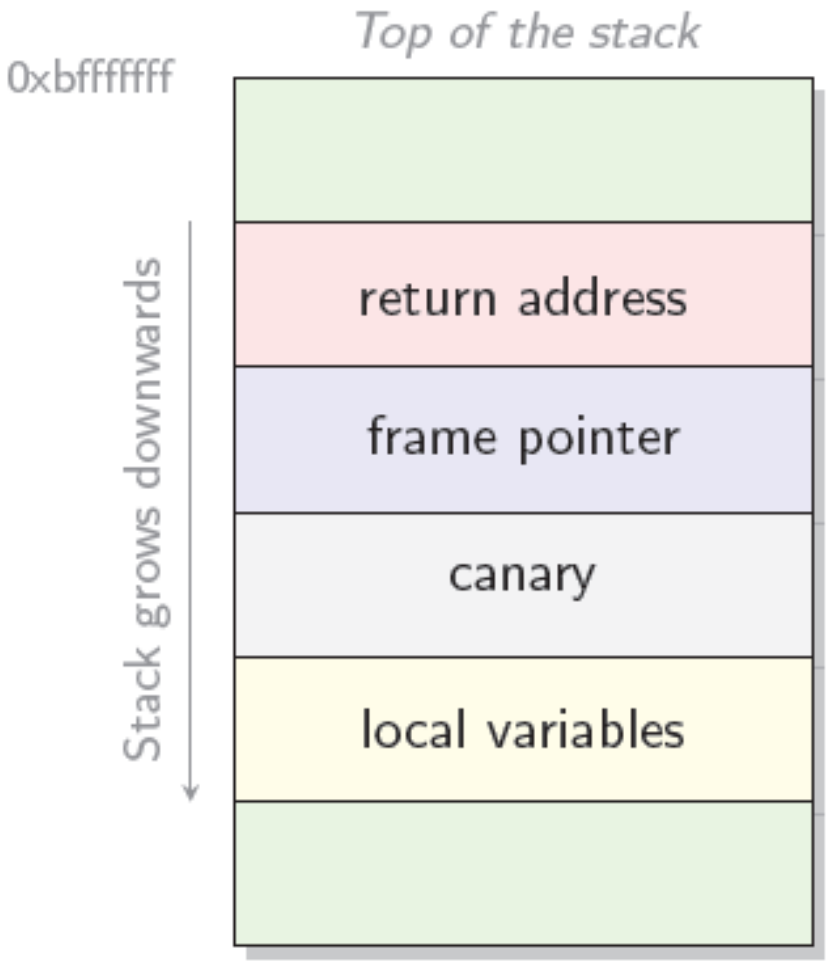
III.A
Canaries

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

How good are they?

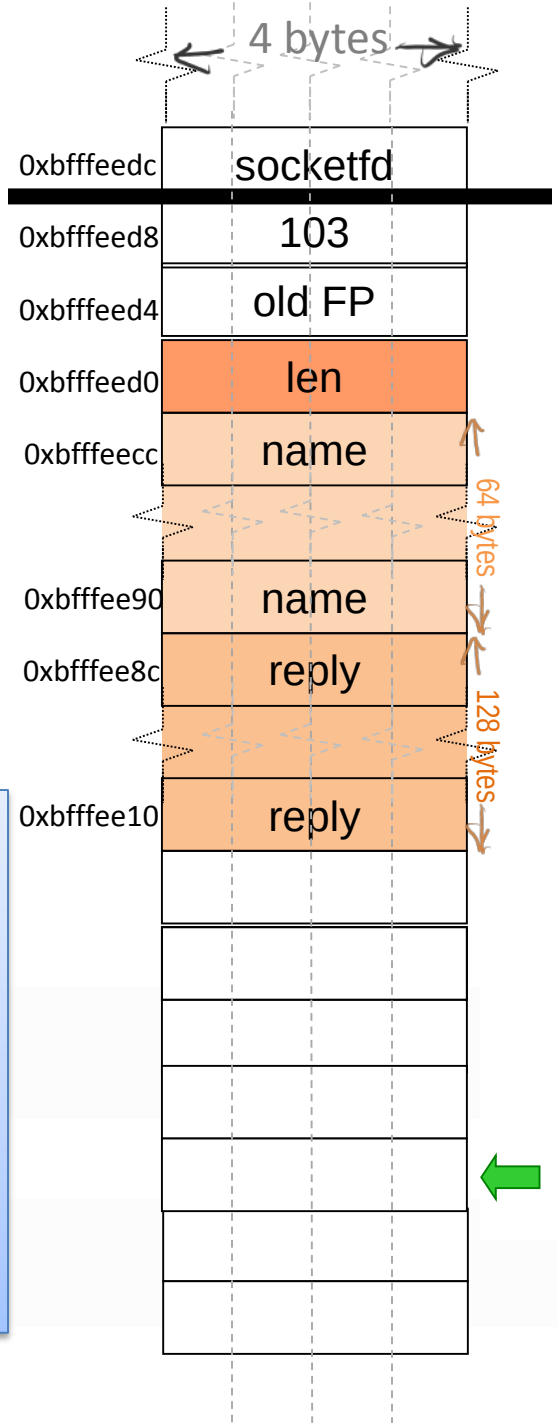
- Assume random canaries protect the stack

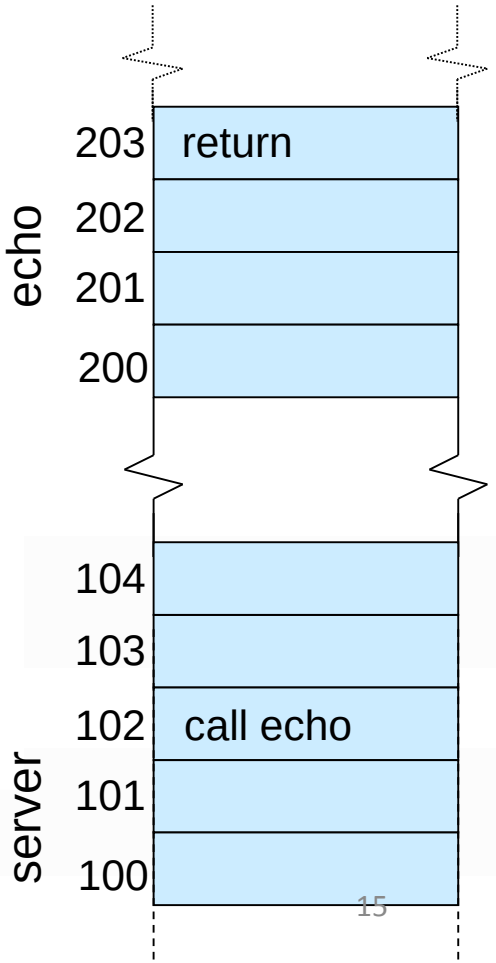
Can you still exploit this?

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

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

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

  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)

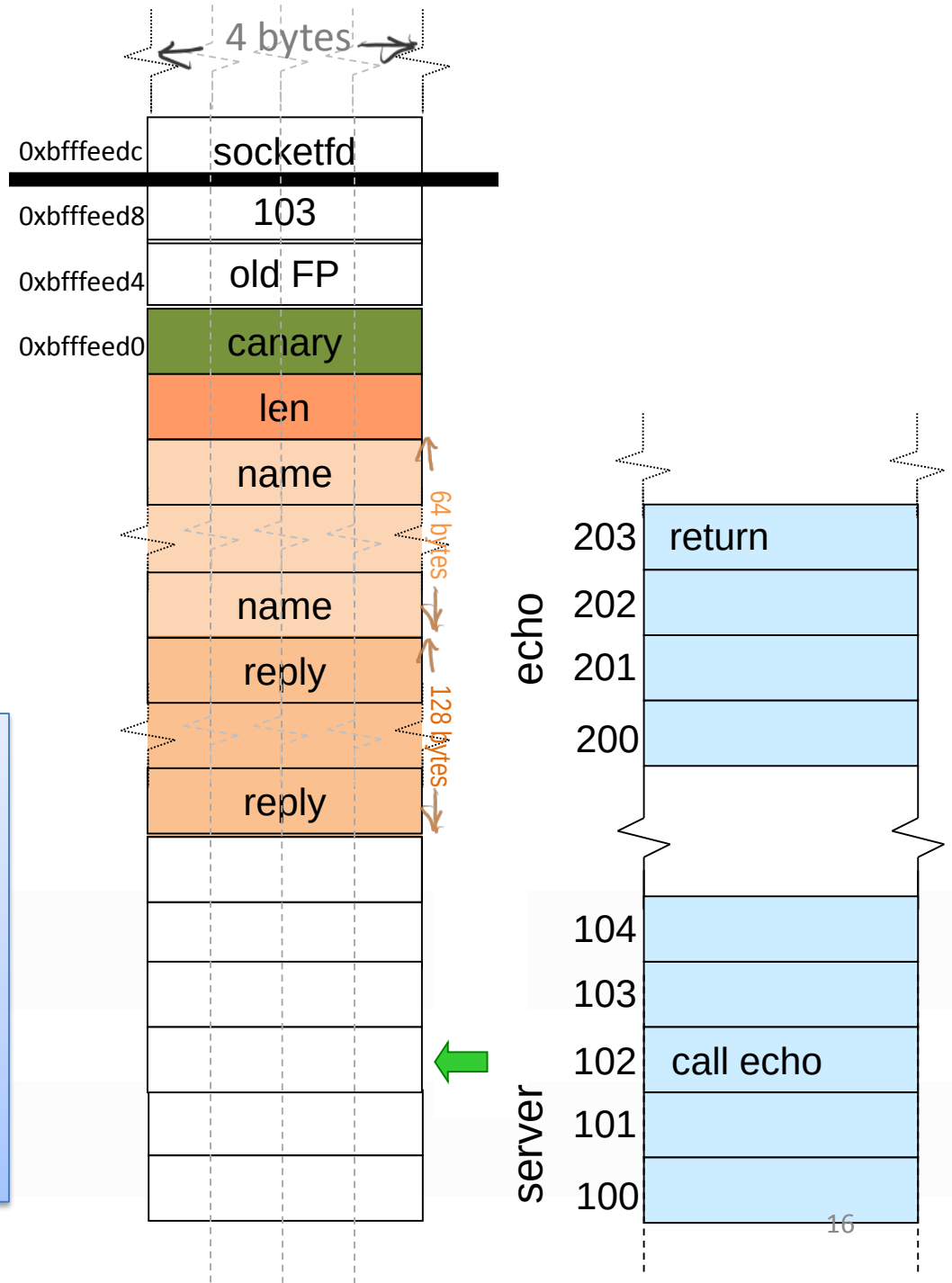
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```





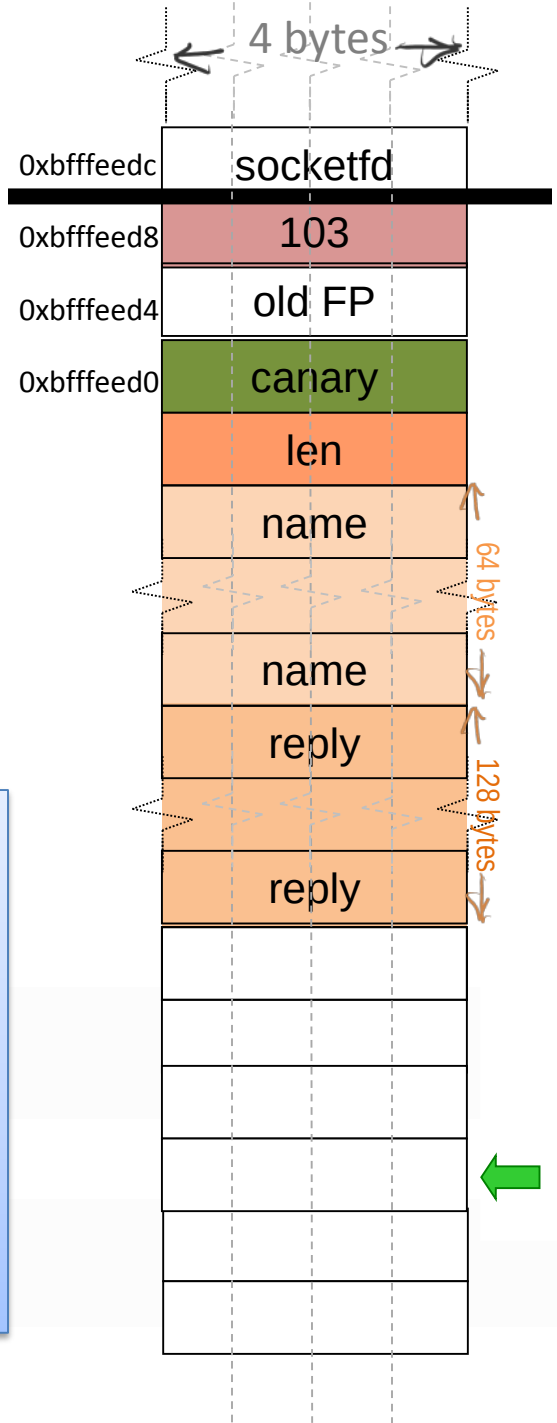
Code (echo)

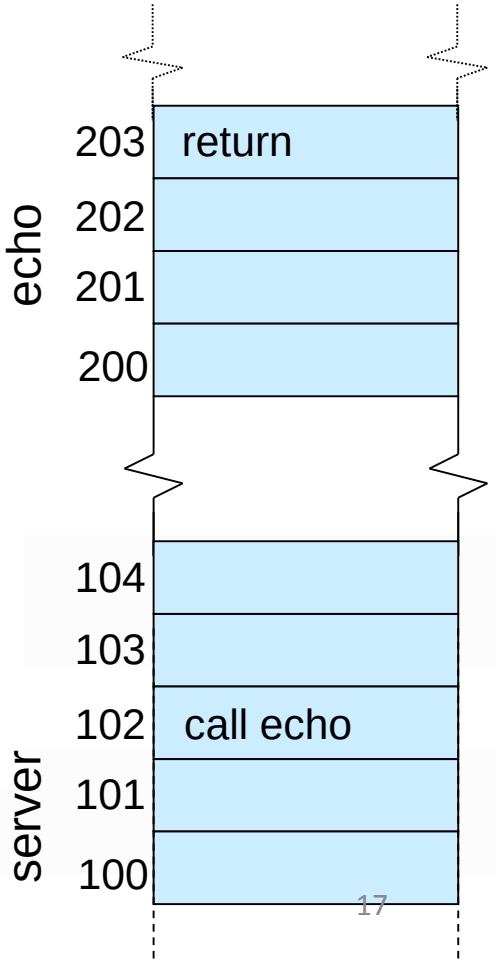
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```



Code (echo)

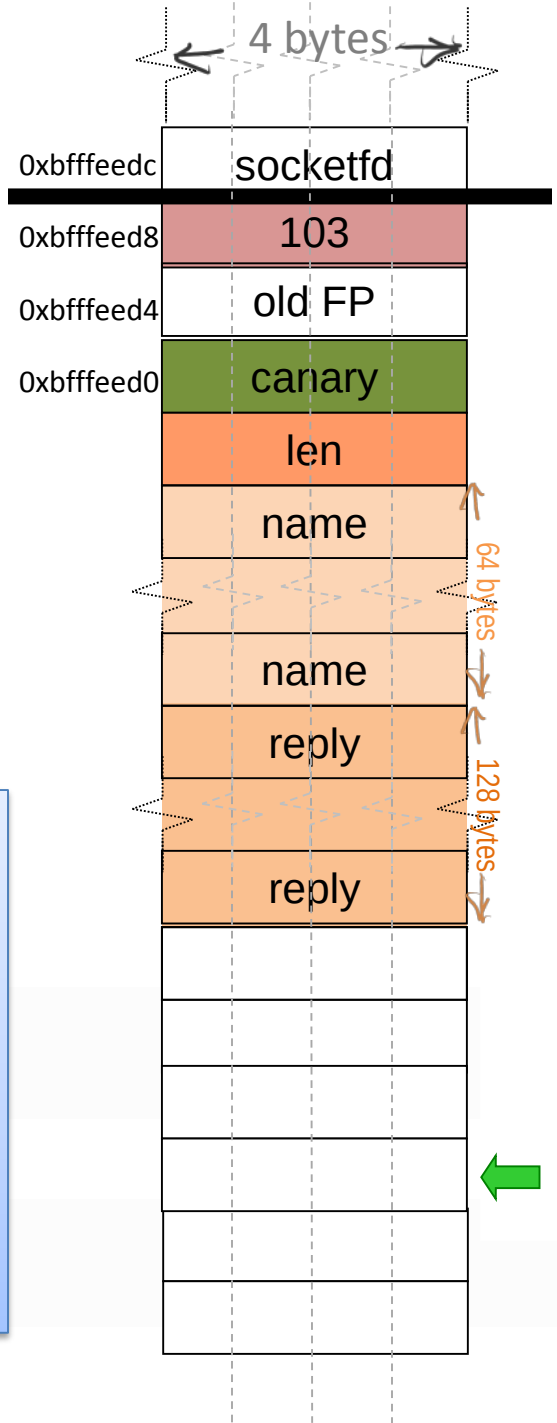
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```

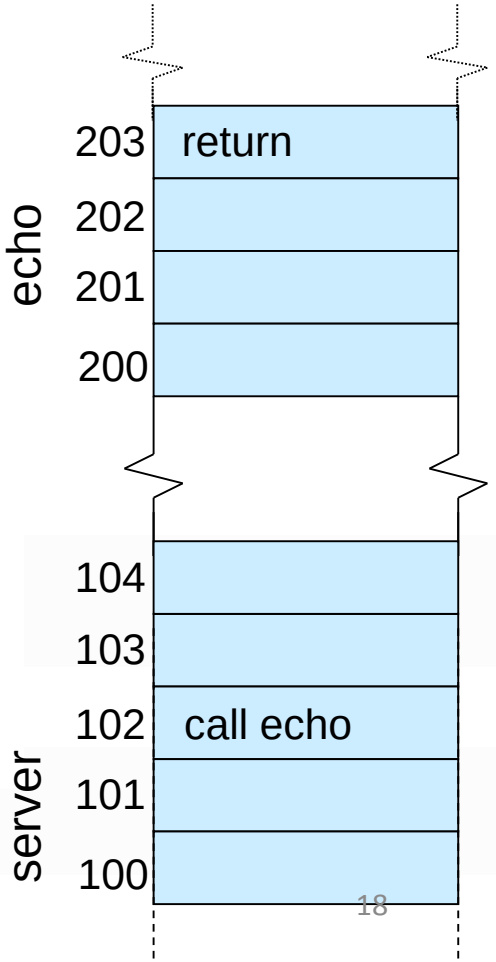




Code (echo)

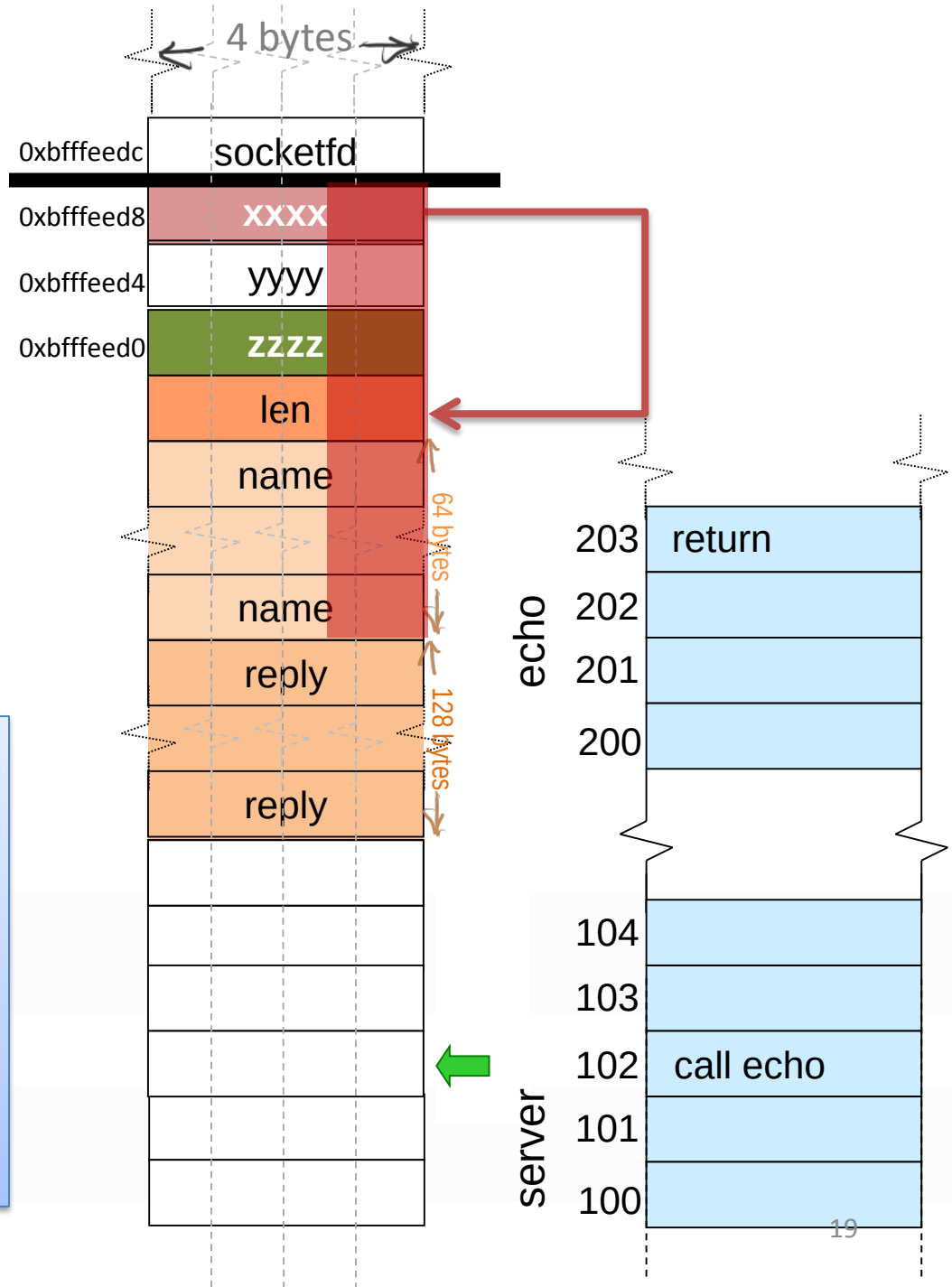
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```



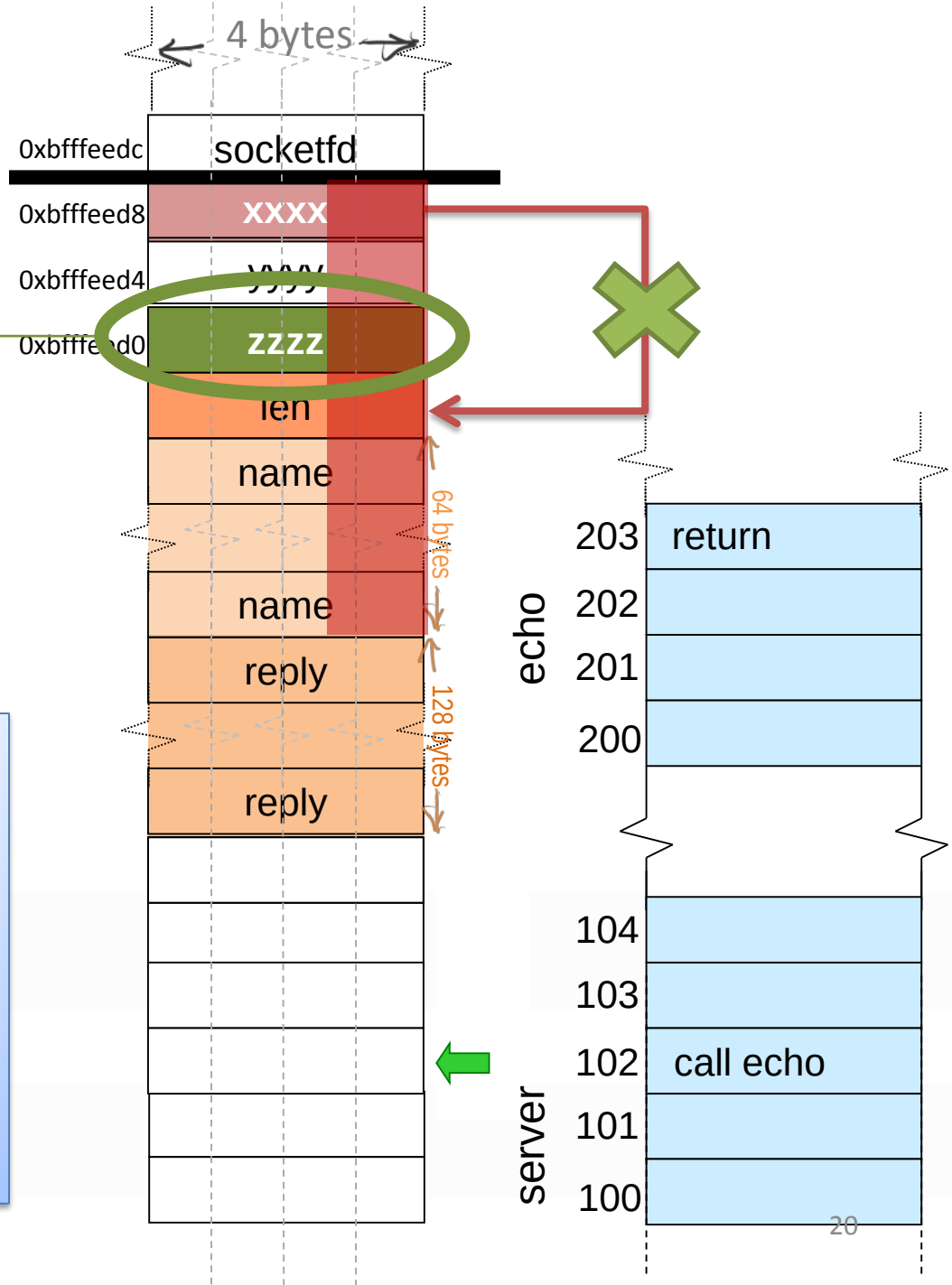


Code (echo)

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

read (fd, name, 128)

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

return;
```



Will not be the same value, so program will crash. OK for causing problems but we can no longer run our own code.



Code (echo)

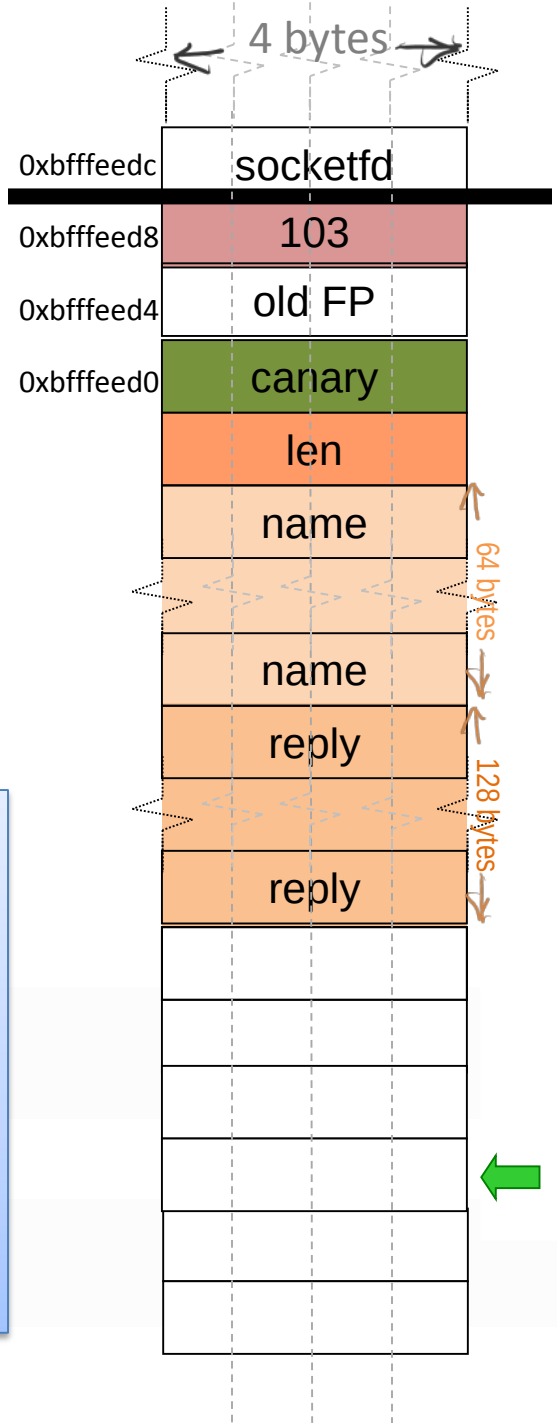
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```

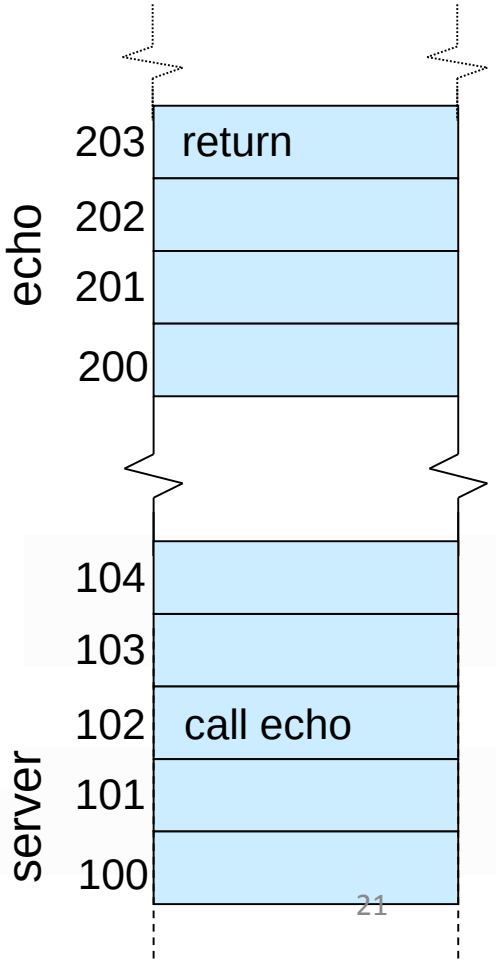
Are we safe?

Any ideas?

Code (echo)

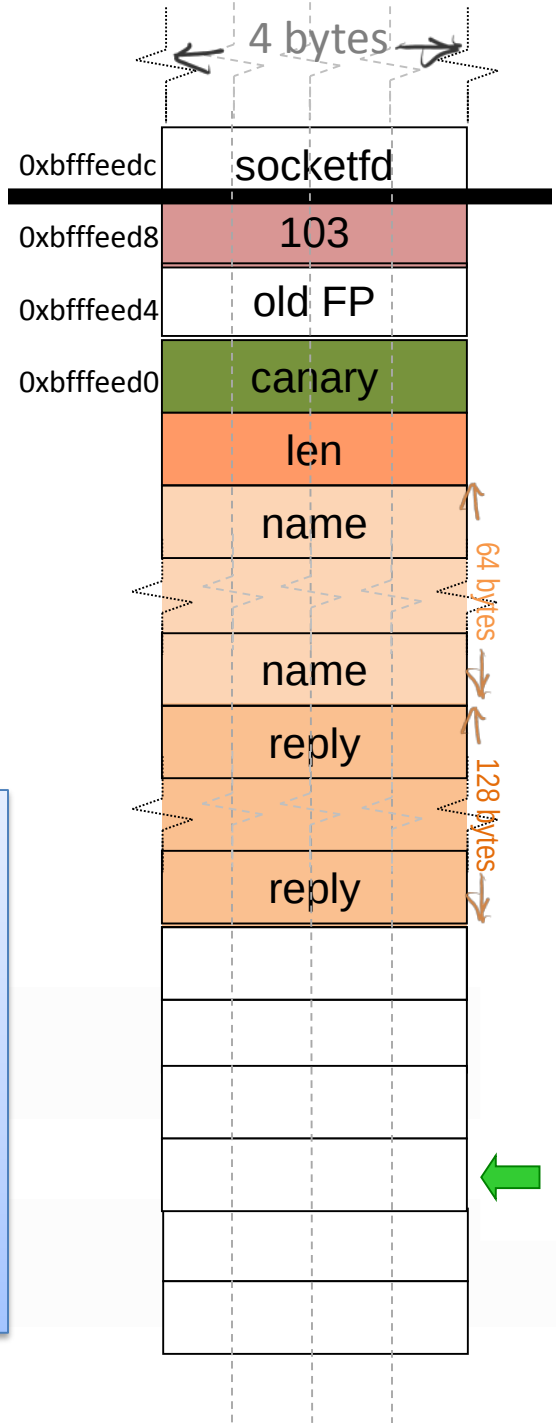
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```

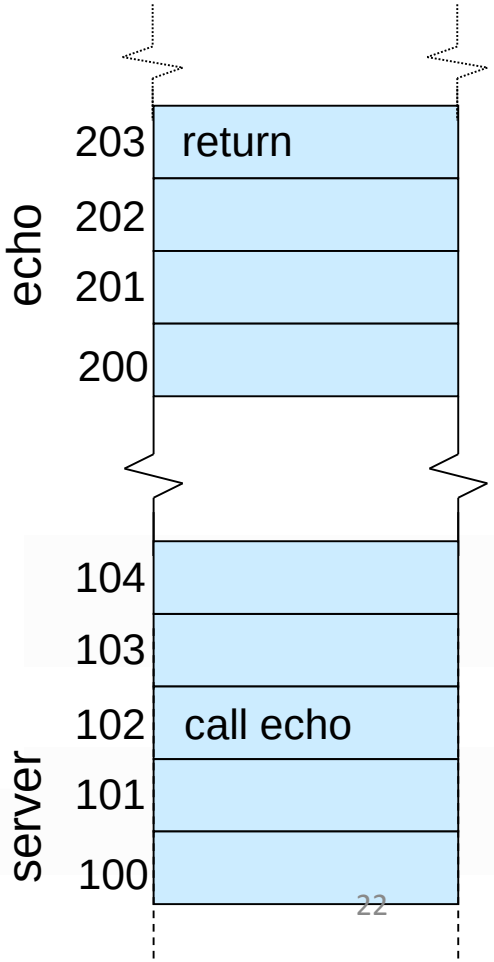




Code (echo)

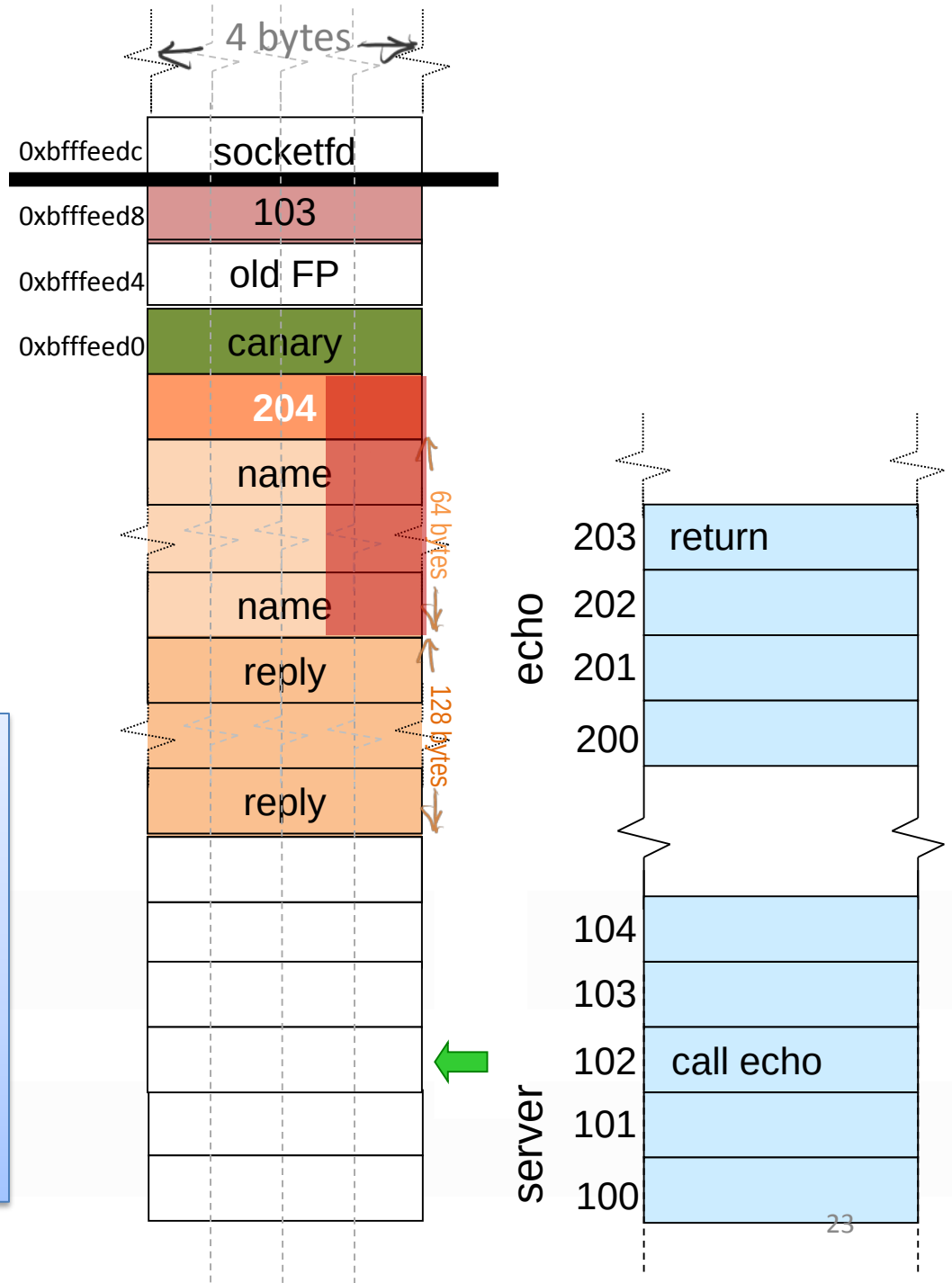
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```





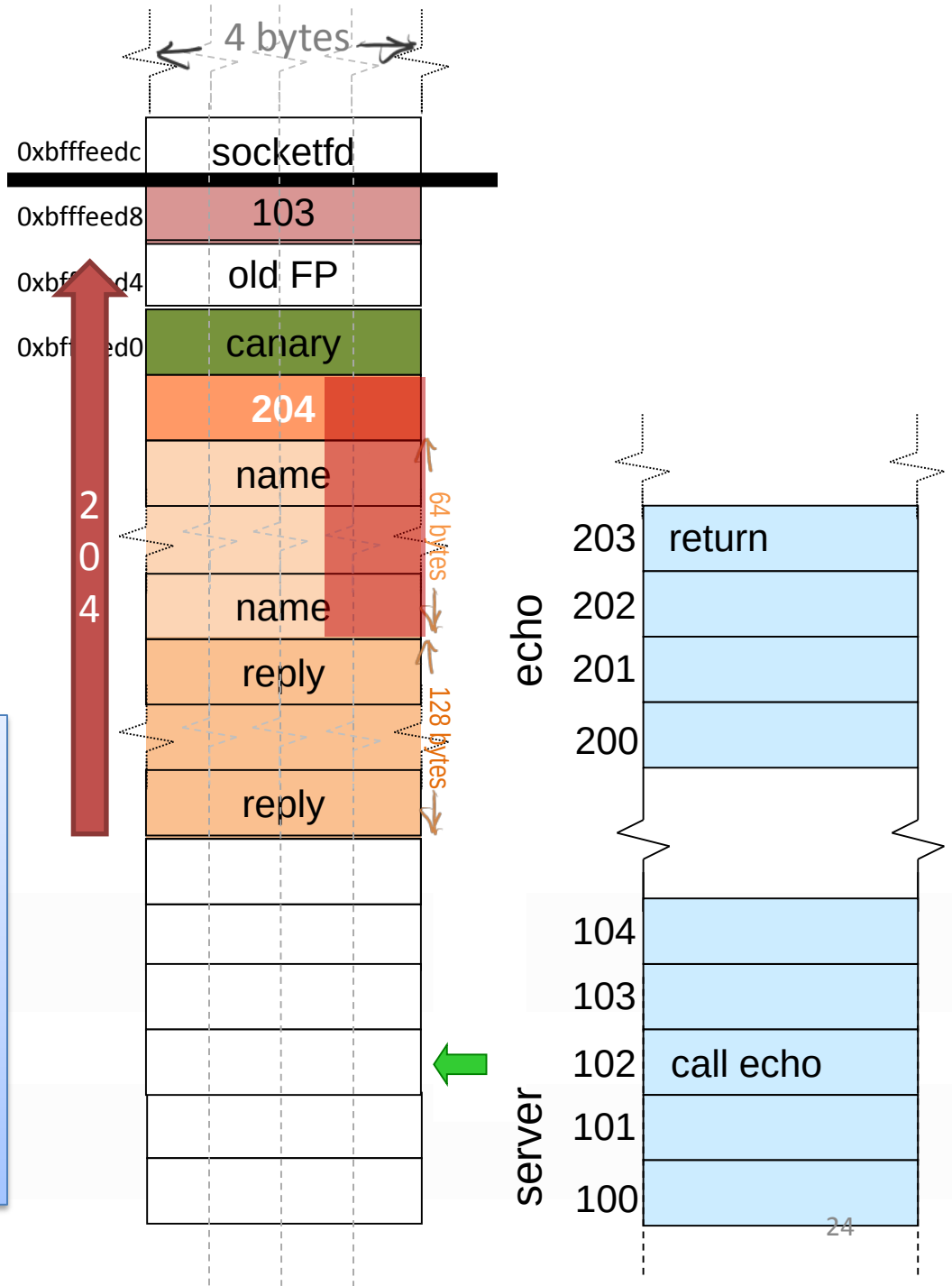
Code (echo)

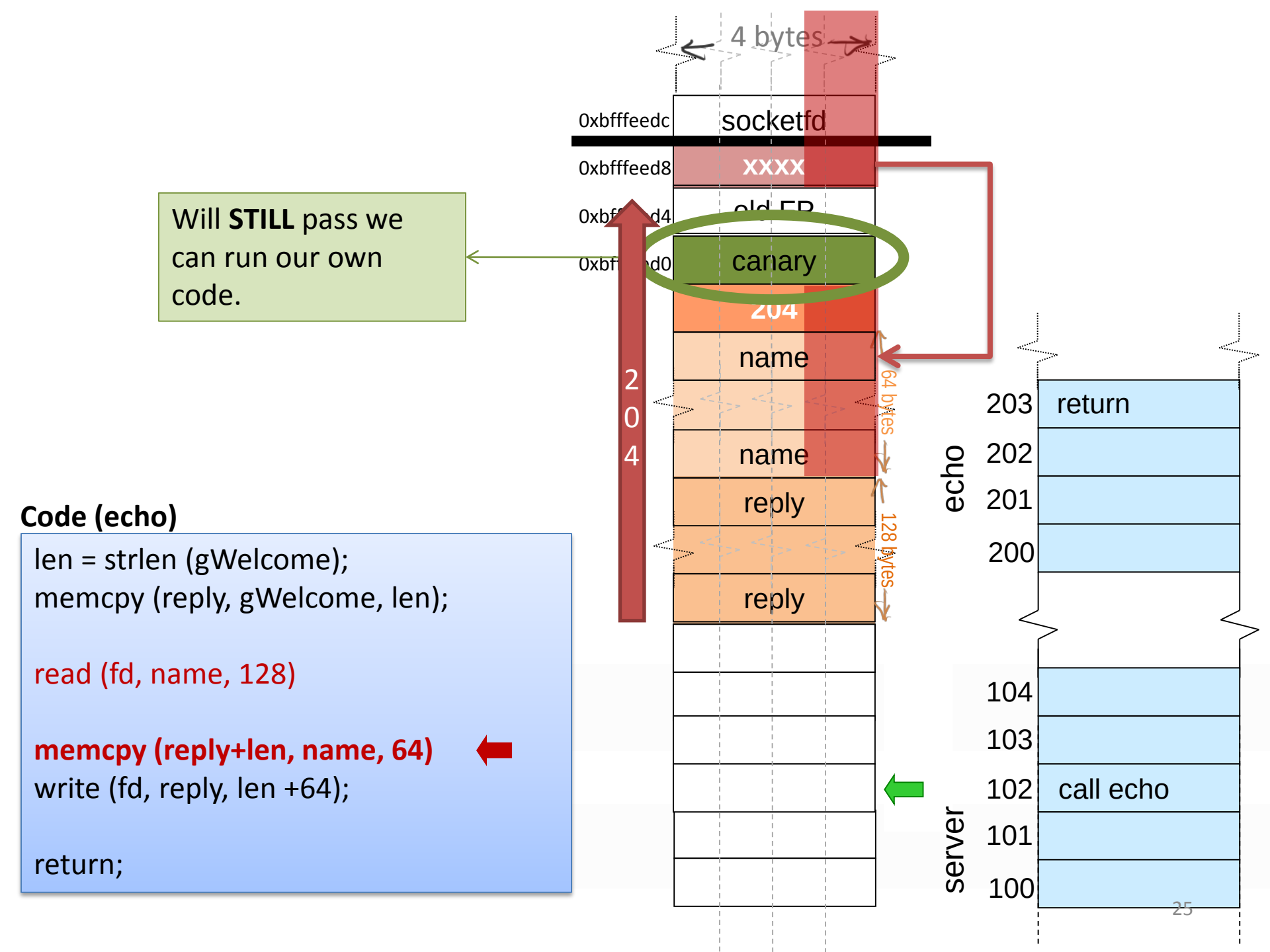
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

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;
```




4 bytes
0xbfffeedc
socketfd
0xbfffeed8
XXXX
canary
204
name
name
reply
reply
204
64 bytes
128 bytes
Will **STILL** pass we can run our own code.
Code (echo)
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);
read (fd, name, 128)
memcpy (reply+len, name, 64)
write (fd, reply, len +64);
return;
echo
203
return
202
201
200
server
104
103
102
call echo
101
100

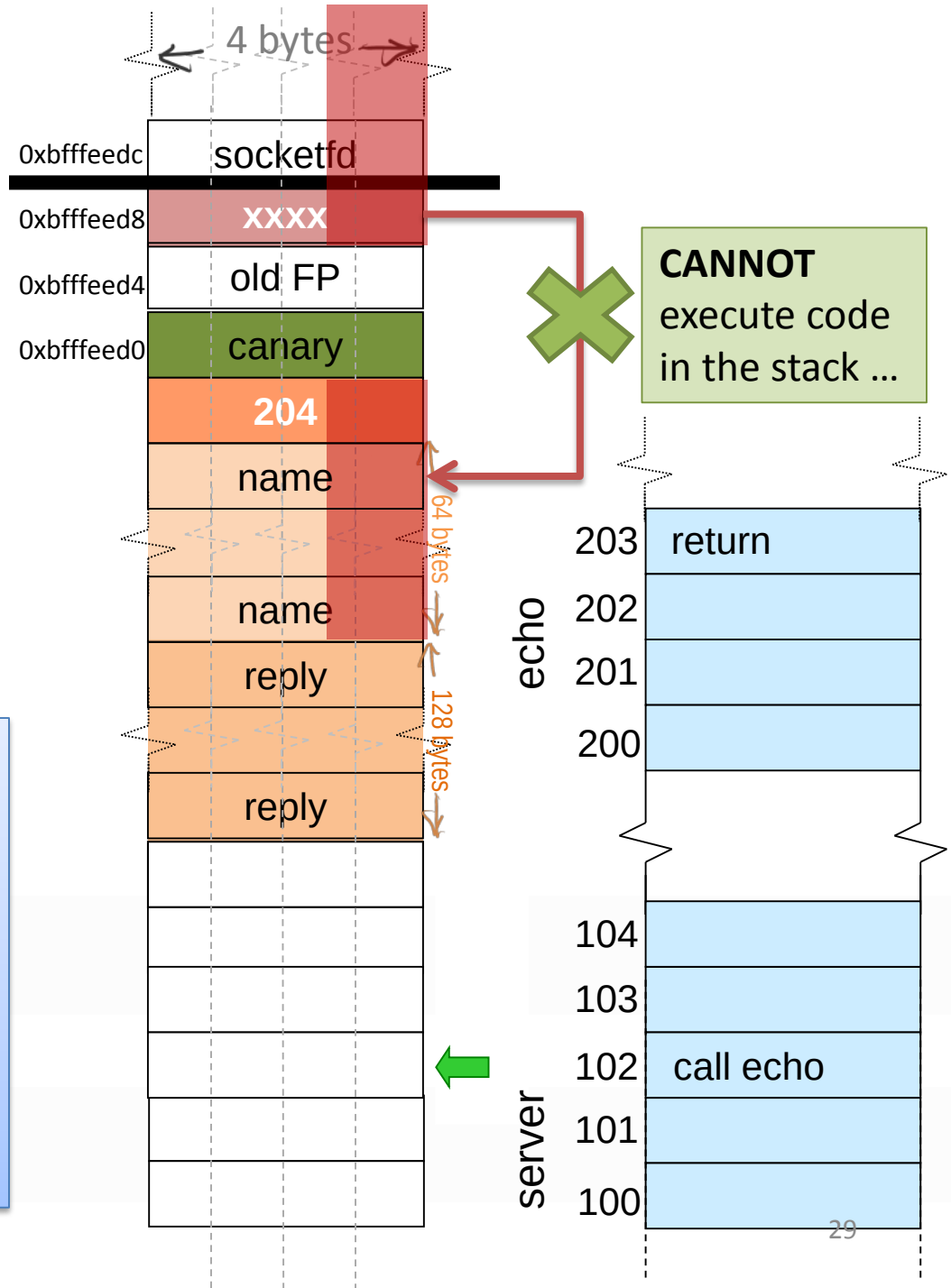
III.B
“DEP”

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)"

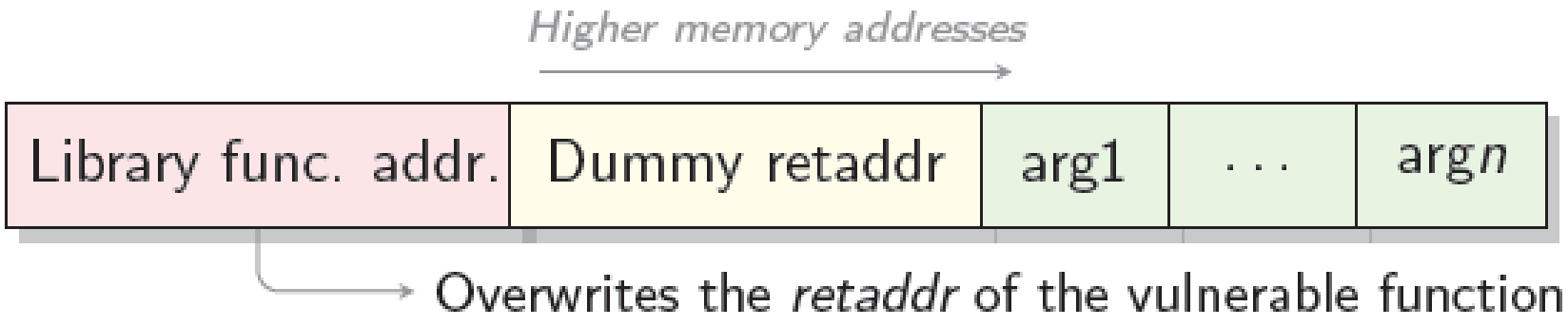
Code (echo)

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

read (fd, name, 128)

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

return;
```

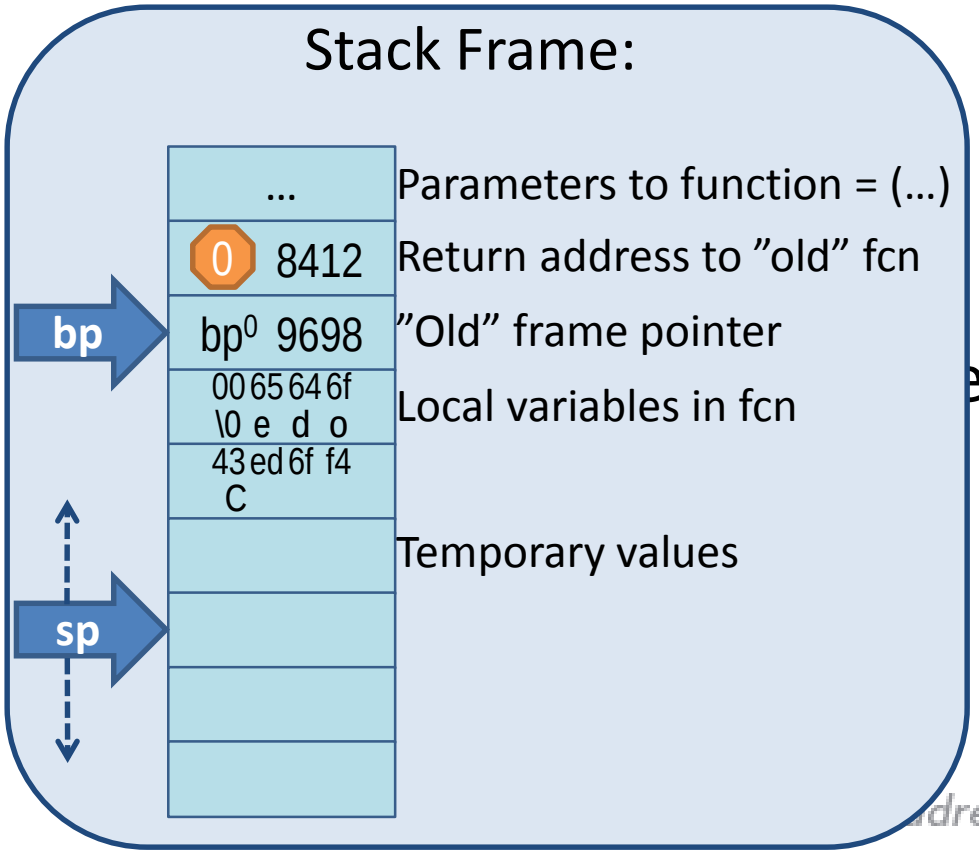


Bypassing W⊕X

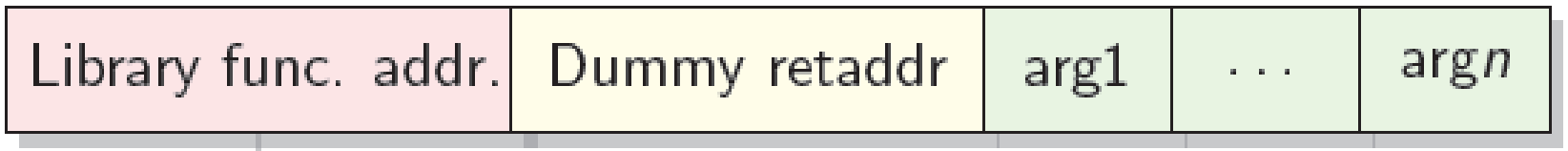
- Return into libc
- Three assumptions:
  - We can manipulate a code pointer
  - The stack is writable
  - We know the address of a “suitable" library function (e.g., system())



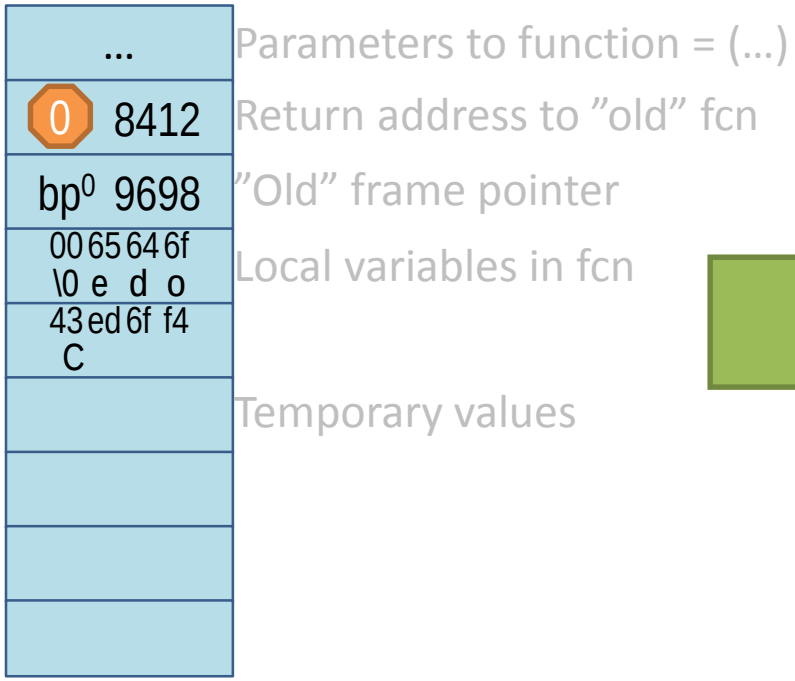
Bypassing W⊕X

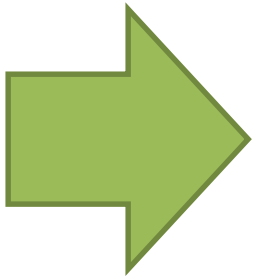


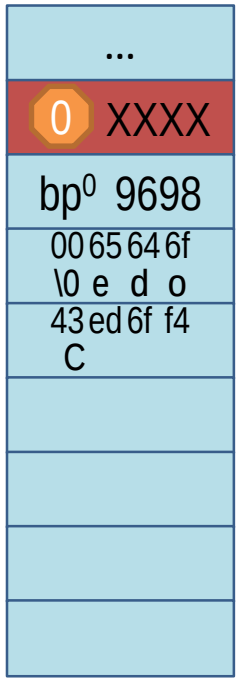
When ***calling*** a function:
(0) Setup fcn parameters.
(1) Push ip of next instruction
(2) Jump to new fcn
(3) Update bp
(4) Update sp
(5) Setup local vars.



Overwrites the *retaddr* of the vulnerable function

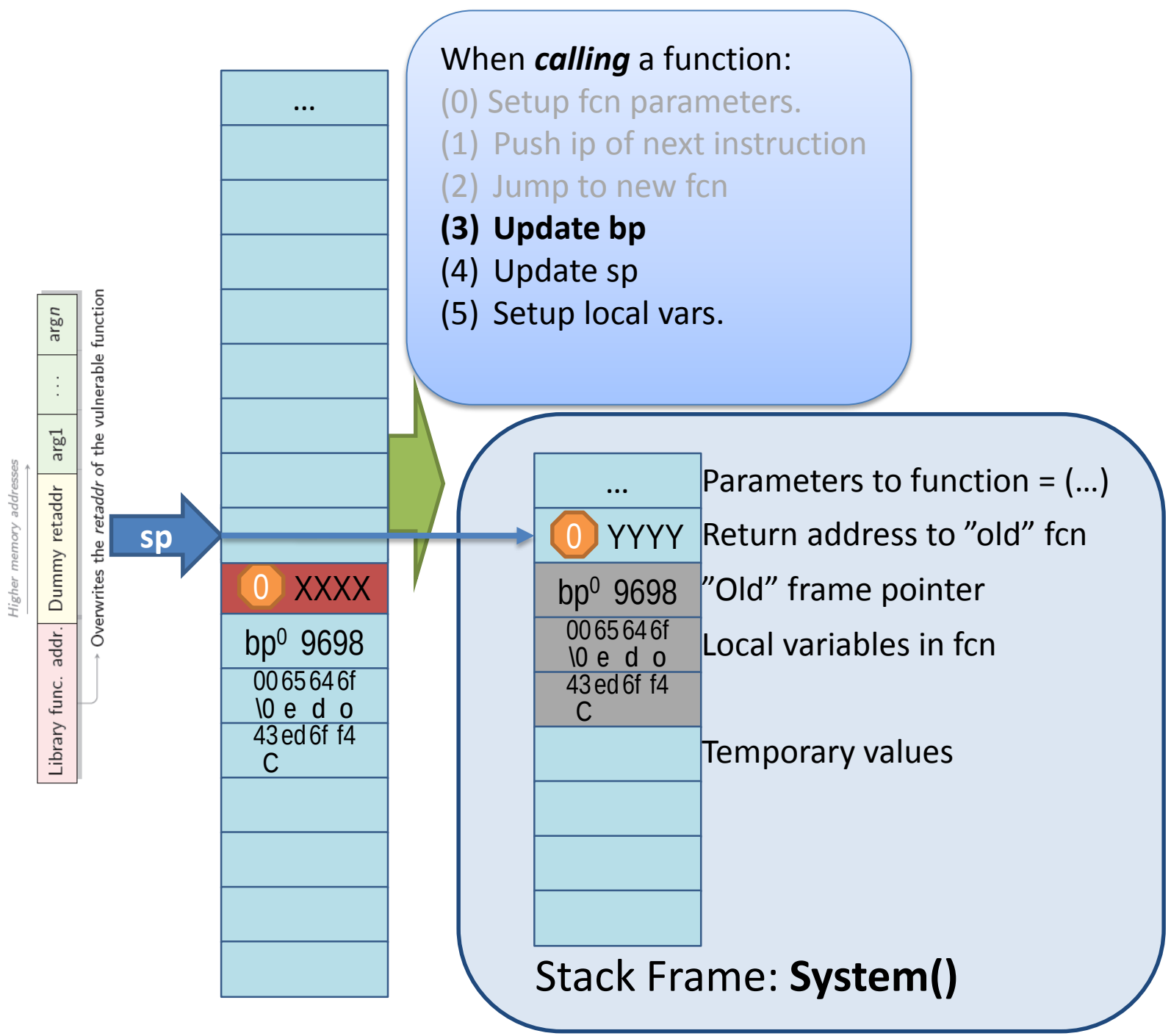


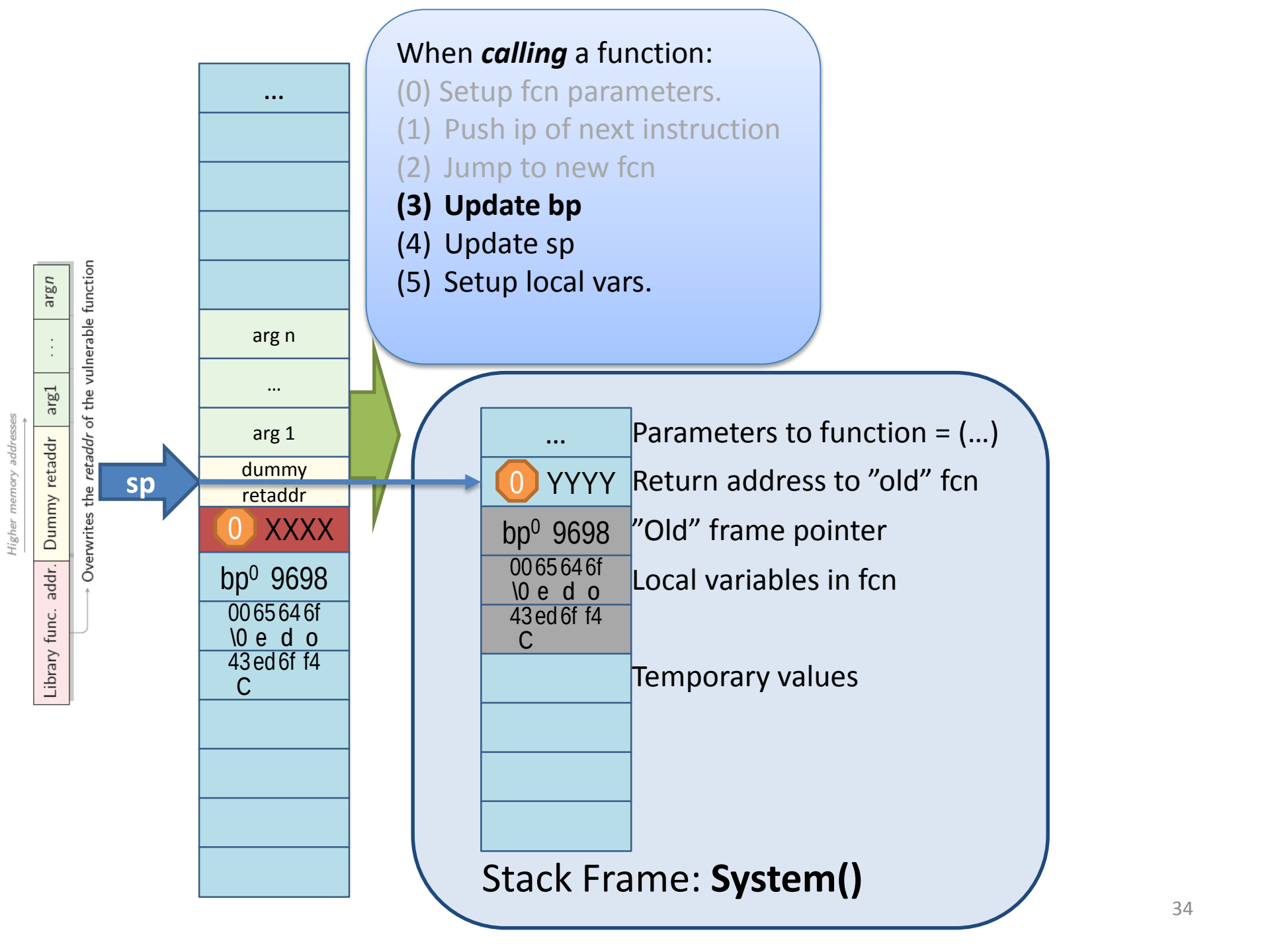


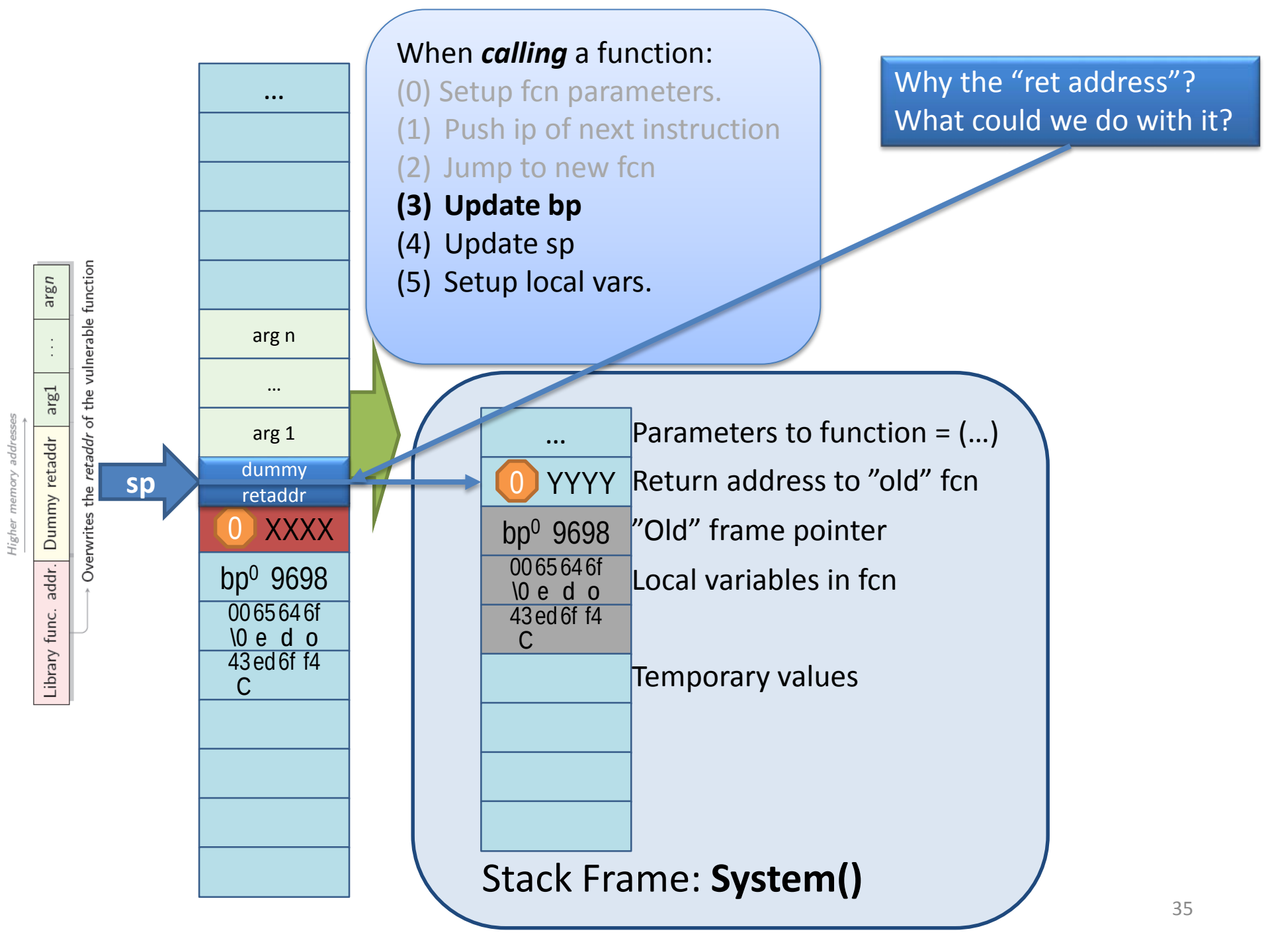


Jump to the “fcn” we want to execute, for example:

system()

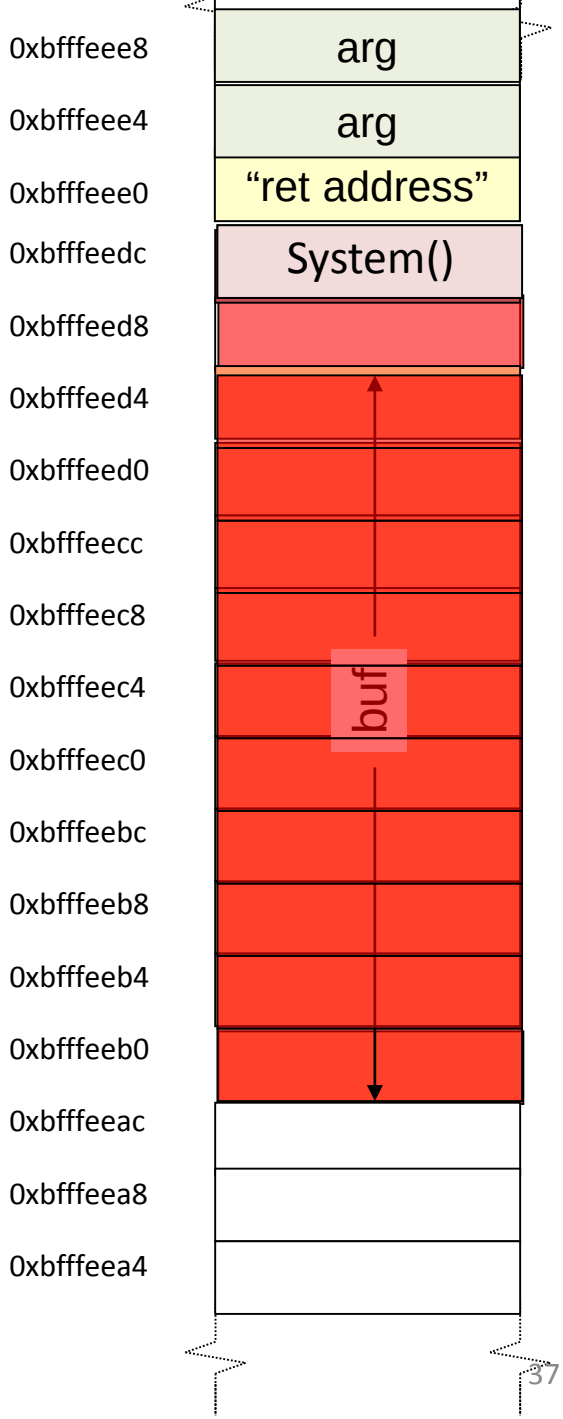

When *calling* a function:
(0) Setup fcn parameters.
(1) Push ip of next instruction
(2) Jump to new fcn
(3) Update bp
(4) Update sp
(5) Setup local vars.
...
sp
0 XXXX
bp0 9698
00 65 64 6f
\0 e d o
43 ed 6f f4
C
Higher memory addresses
argn
...
arg1
Dummy retaddr
Library func. addr.
Overwrites the retaddr of the vulnerable function
...
0 YYYY
bp0 9698
00 65 64 6f
\0 e d o
43 ed 6f f4
C
Parameters to function = (...)
Return address to "old" fcn
"Old" frame pointer
Local variables in fcn
Temporary values
Stack Frame: **System()**


When *calling* a function:
(0) Setup fcn parameters.
(1) Push ip of next instruction
(2) Jump to new fcn
(3) Update bp
(4) Update sp
(5) Setup local vars.
...
arg n
...
arg 1
dummy
retaddr
0 XXXX
bp0 9698
00 65 64 6f
\0 e d o
43 ed 6f f4
C
sp
Higher memory addresses
argn
...
arg1
Dummy retaddr
Library func. addr.
Overwrites the retaddr of the vulnerable function
...
0 YYYY
bp0 9698
00 65 64 6f
\0 e d o
43 ed 6f f4
C
Parameters to function = (...)
Return address to "old" fcn
"Old" frame pointer
Local variables in fcn
Temporary values
Stack Frame: **System()**


When *calling* a function:
(0) Setup fcn parameters.
(1) Push ip of next instruction
(2) Jump to new fcn
(3) Update bp
(4) Update sp
(5) Setup local vars.
Why the "ret address"?
What could we do with it?
...
arg n
...
arg 1
dummy
retaddr
0 XXXX
bp0 9698
00 65 64 6f
\0 e d o
43 ed 6f f4
C
sp
Higher memory addresses
argn
...
arg1
Dummy retaddr
Library func. addr.
Overwrites the retaddr of the vulnerable function
...
0 YYYY
bp0 9698
00 65 64 6f
\0 e d o
43 ed 6f f4
C
Parameters to function = (...)
Return address to "old" fcn
"Old" frame pointer
Local variables in fcn
Temporary values
Stack Frame: **System()**

Stack

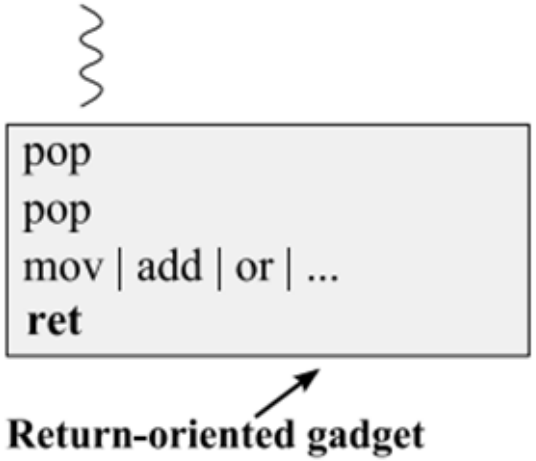
- Why the “ret address”?
- What could we do with it?



VU UNIVERSITY AMSTERDAM

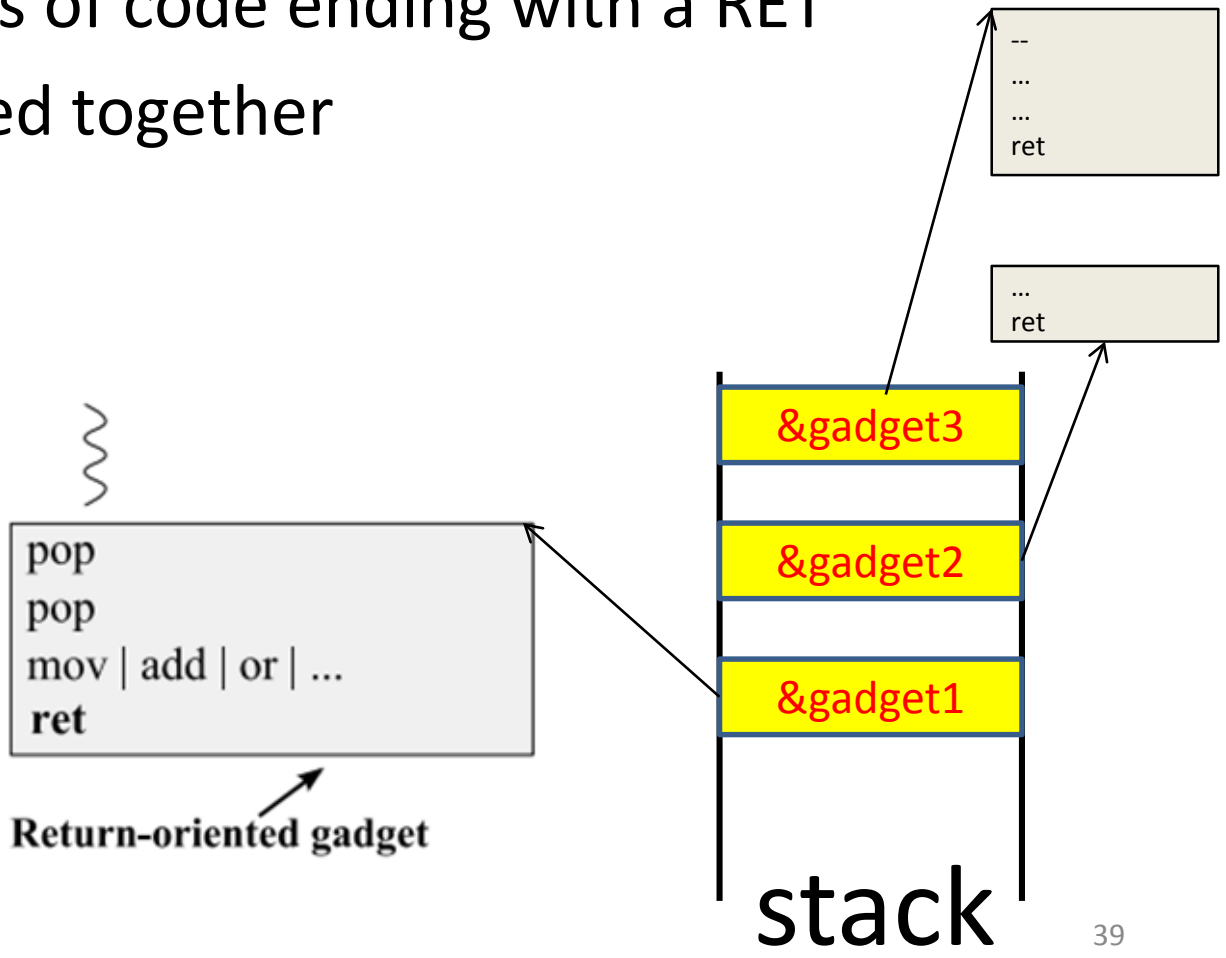
Return Oriented Programming

- 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



How good are they?

- Assume random canaries protect the stack
- Assume DEP prevents execution of the stack

Can you still 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);

  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);
}
```

III.C
ASLR

Let us make it a little harder still…

Address Space Layout Randomisation

- Idea:
  - 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, . . .

ASLR: Problems

- 32-bit implementations use few randomisation bits
- An attacker can still exploit non-randomised areas, or rely on other information leaks (e.g., format bug)

- So… (I bet you saw this one coming)….

How good are they?

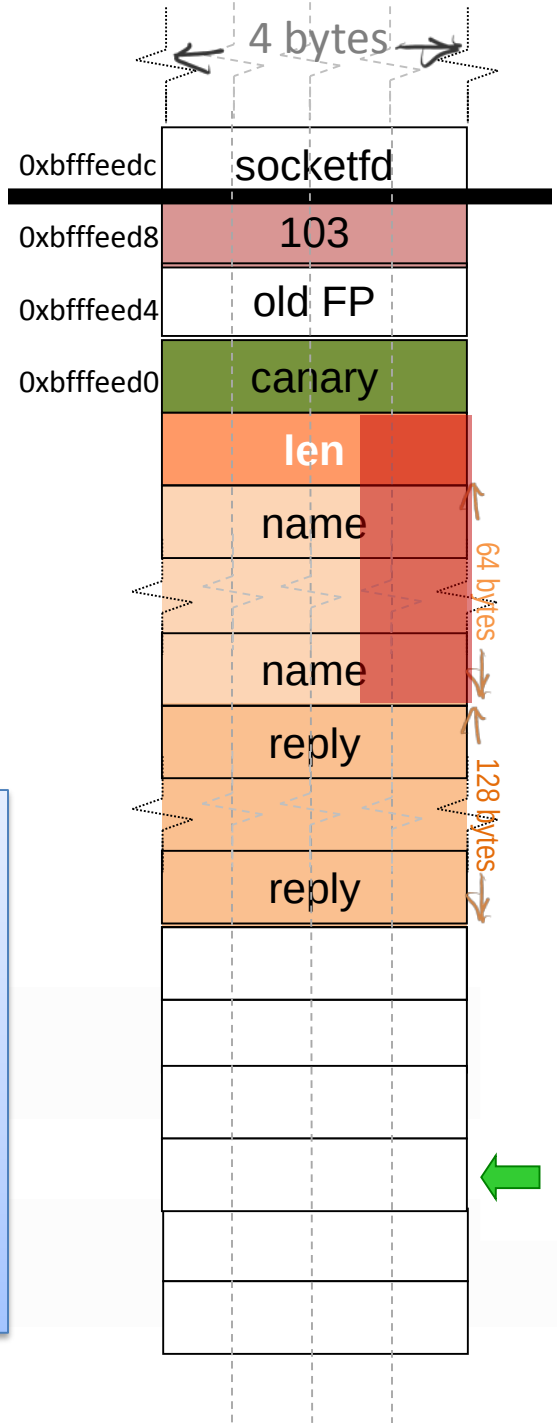
- Assume random canaries protect the stack
- Assume DEP prevents execution of the stack
- Assume ASLR randomized the stack *and* the start address of the code
  - but let us assume that all functions are still at the same relative offset from start address of code
  - (in other words: need only a single code pointer)

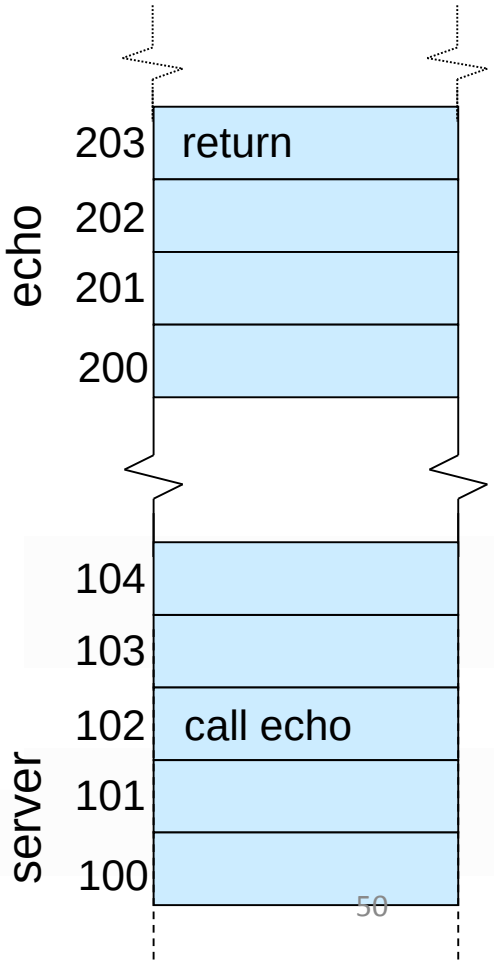
Can you still exploit this?

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

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

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

  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;
```

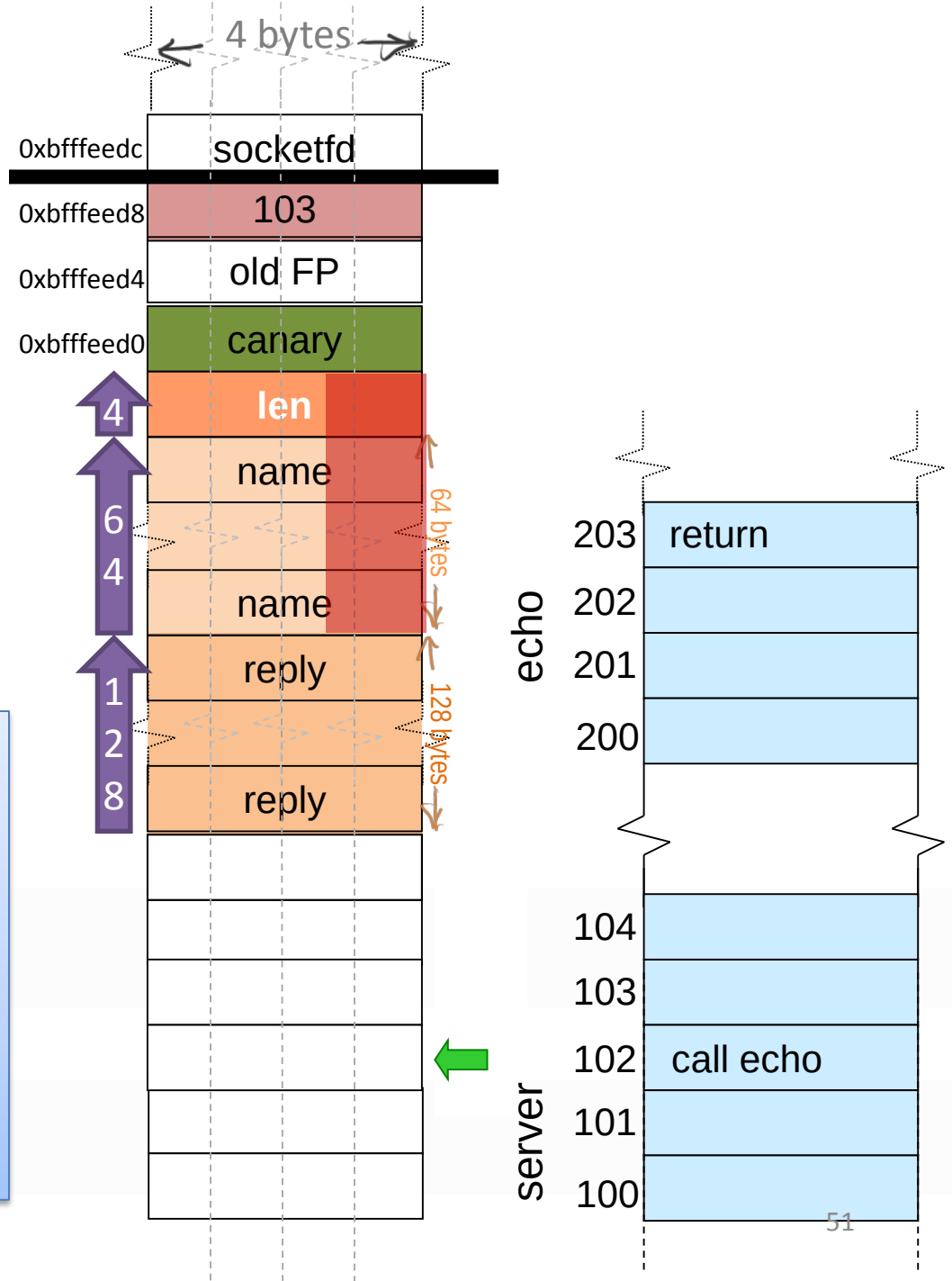


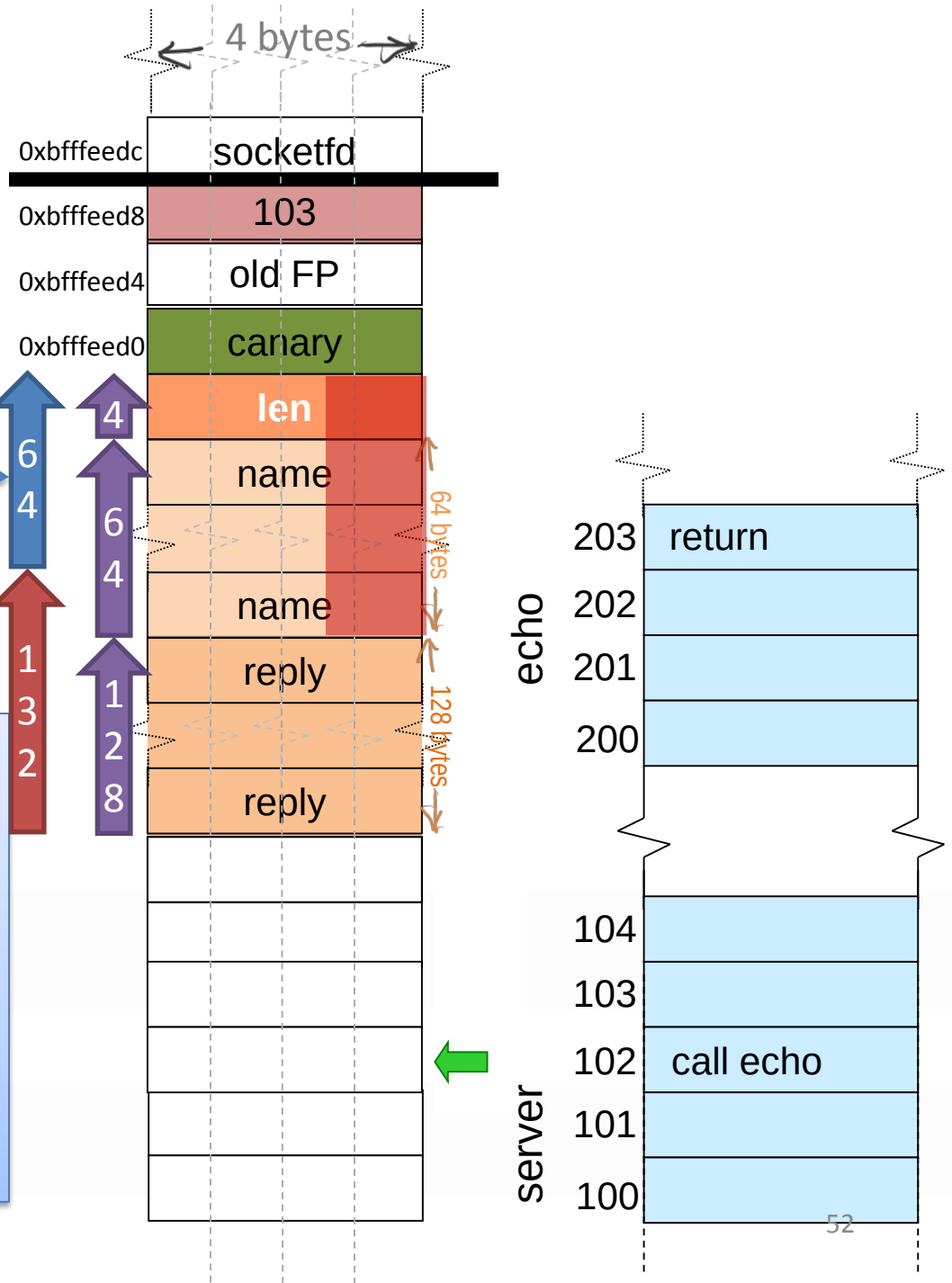


VU UNIVERSITY AMSTERDAM

Code (echo)

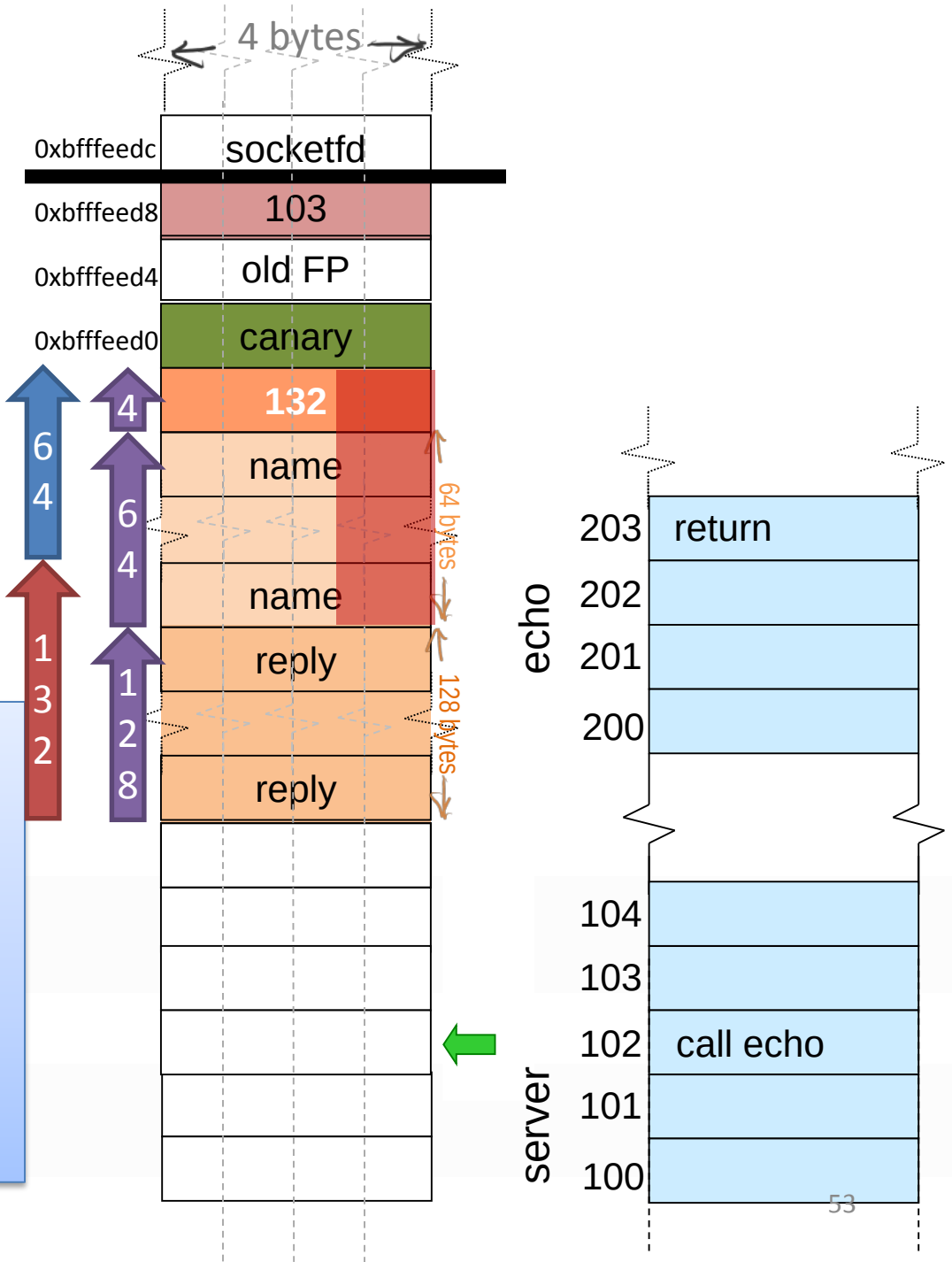
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

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;
```

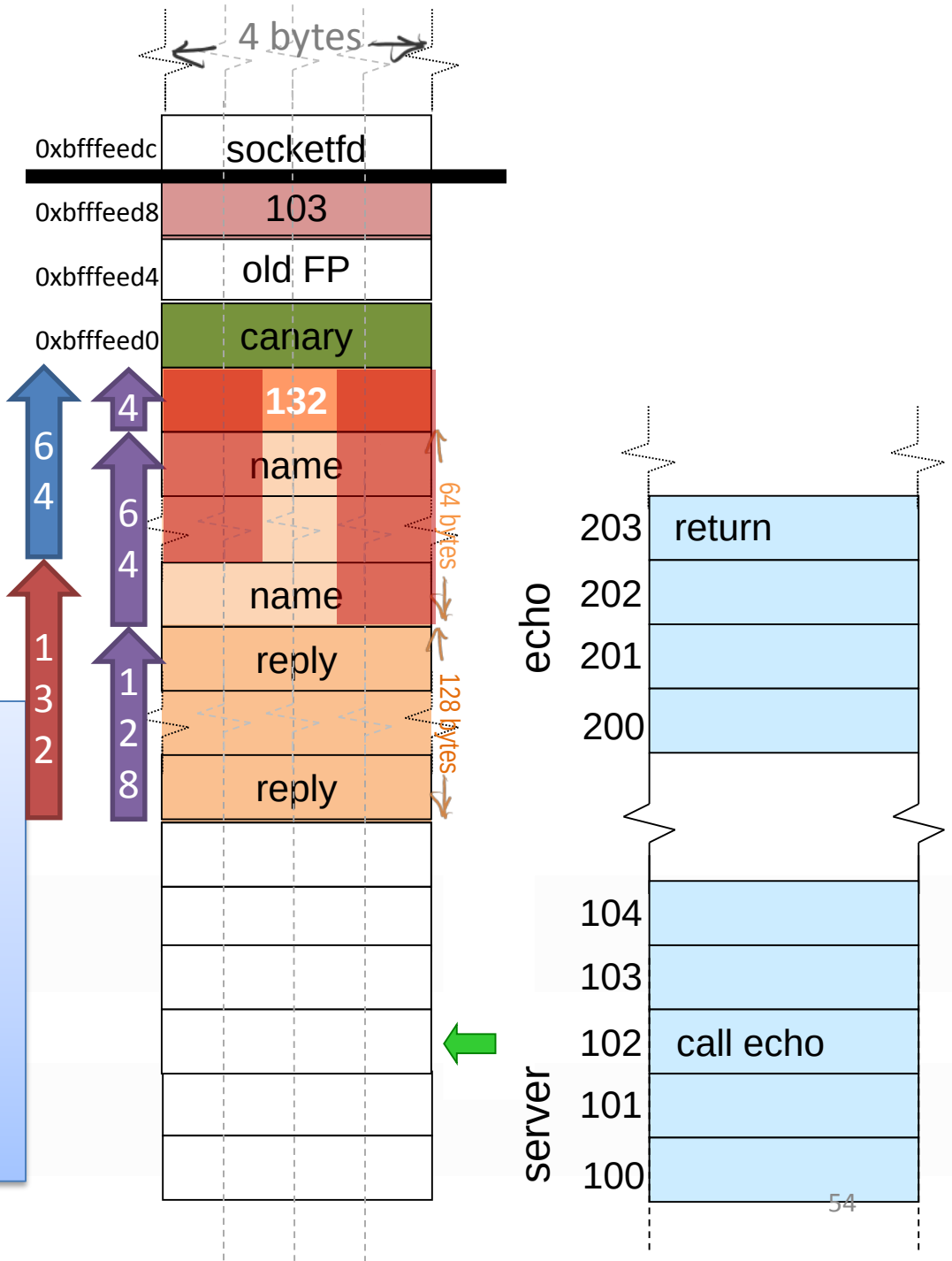


Code (echo)

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

read (fd, name, 128)

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

return;
```

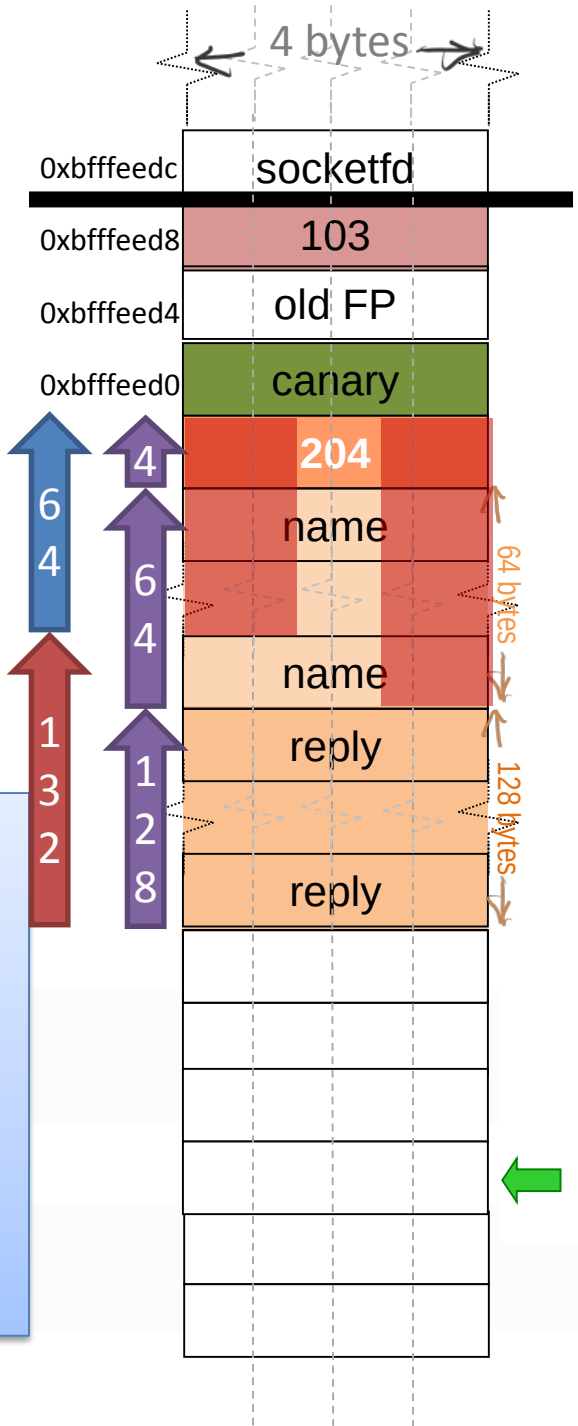
Code (echo)

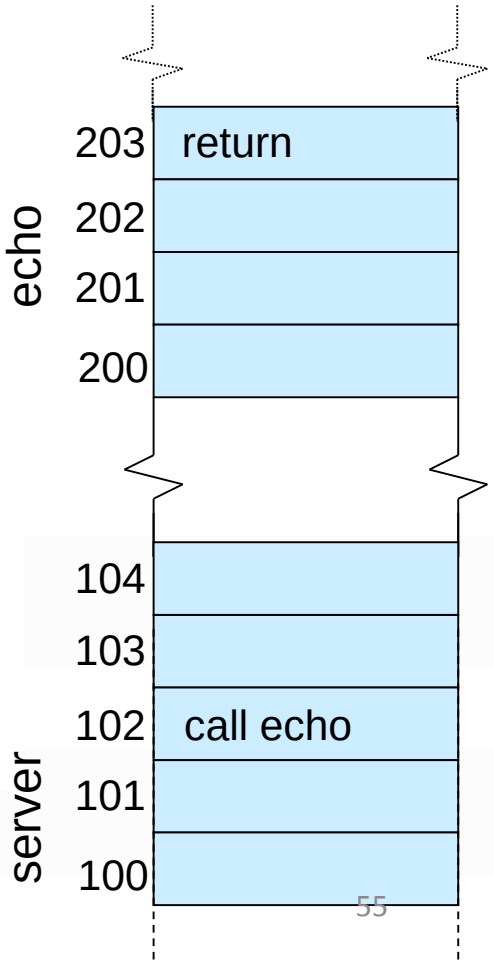
```
len = strlen (gWelcome);
memcpy (reply, gWelcome, len);

read (fd, name, 128)

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

return;
```

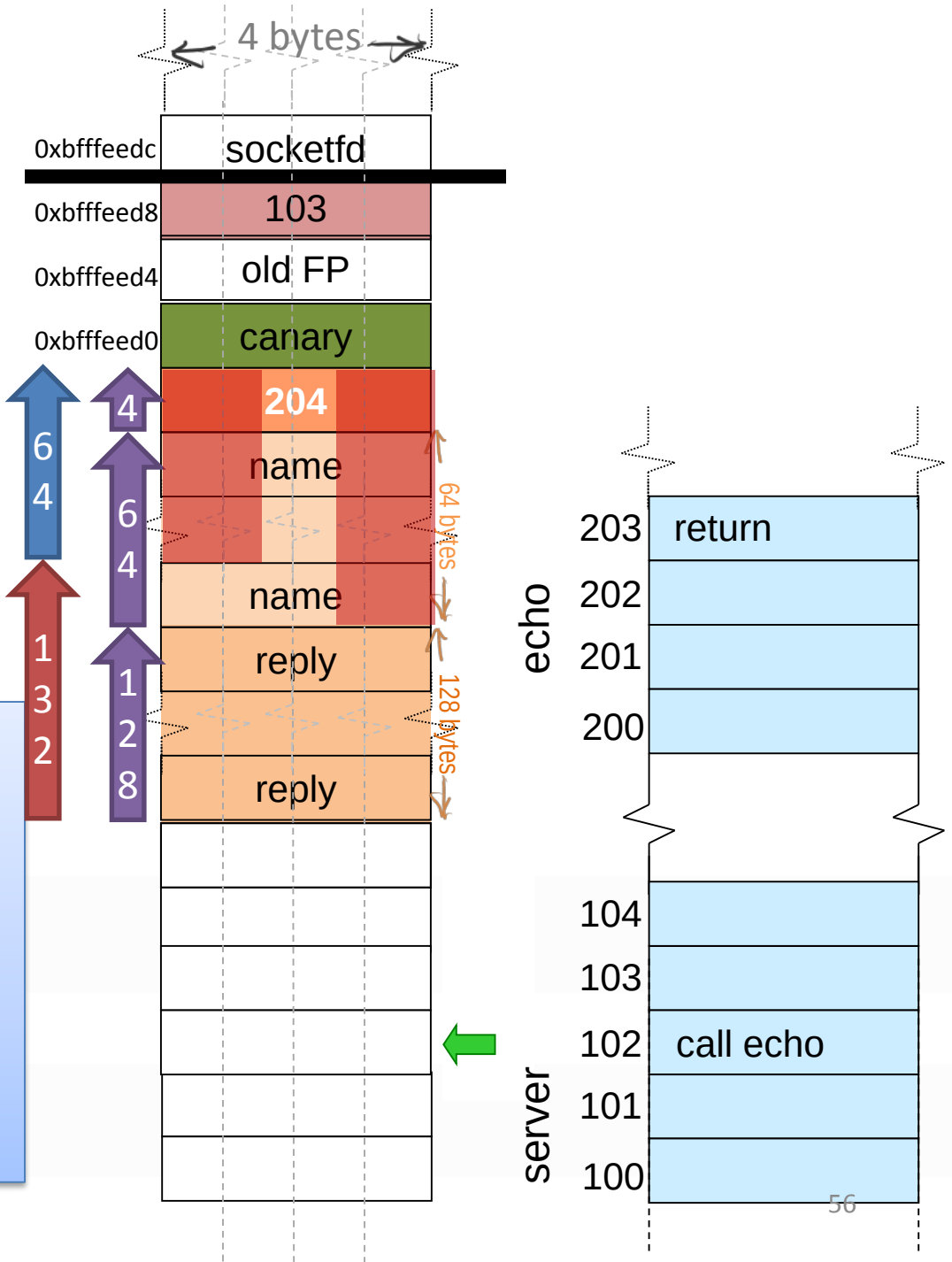


Code (echo)

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

read (fd, name, 128)

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

return;
```

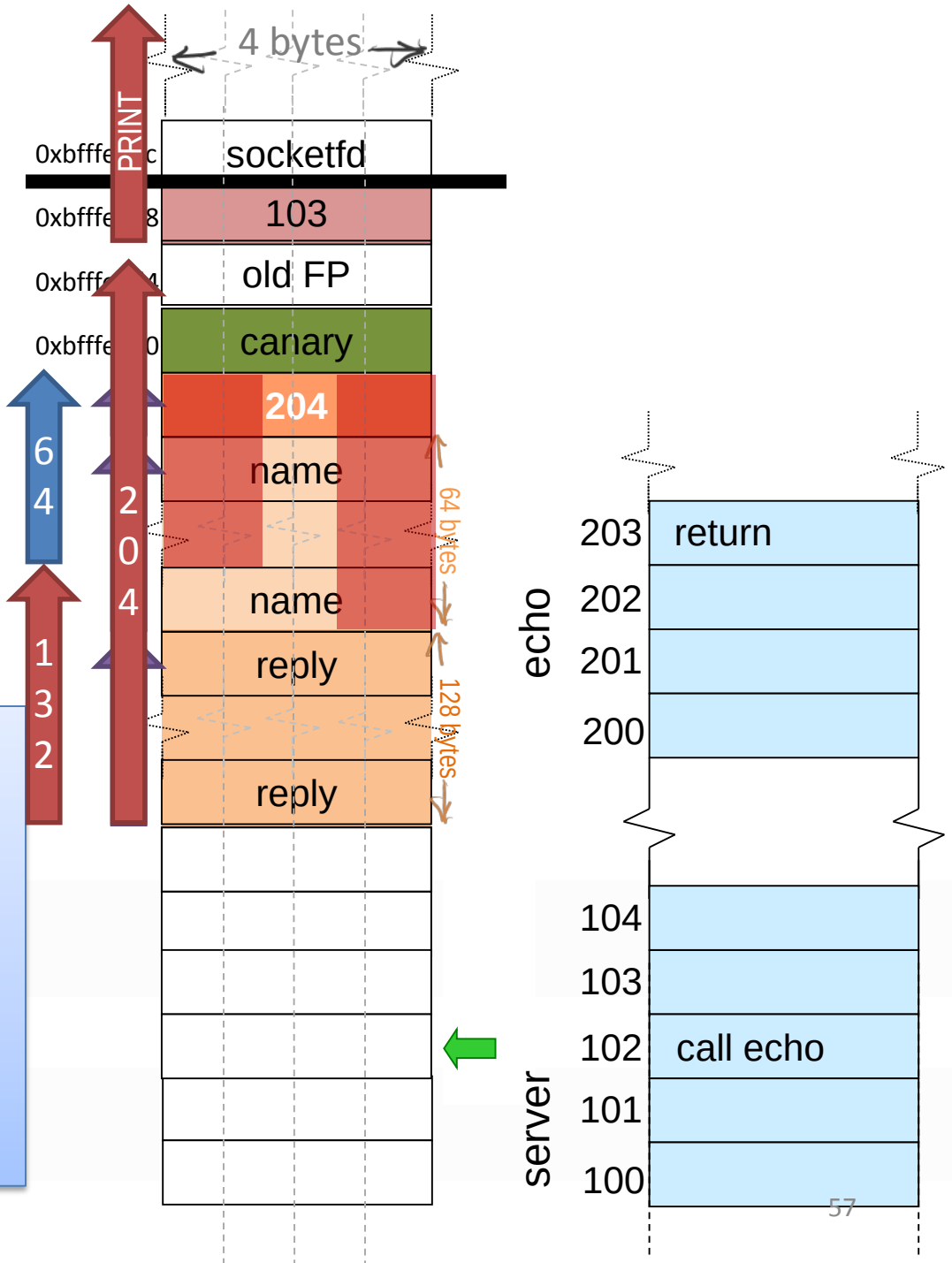






Code (echo)

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

read (fd, name, 128)

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

return;
```

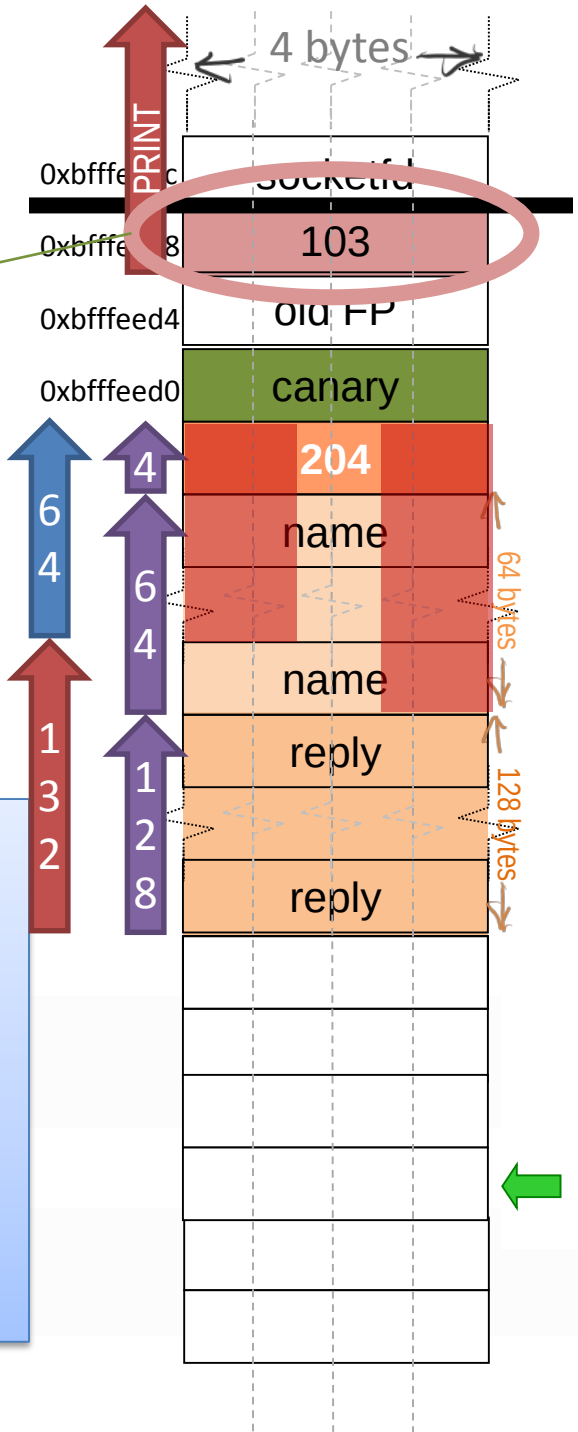


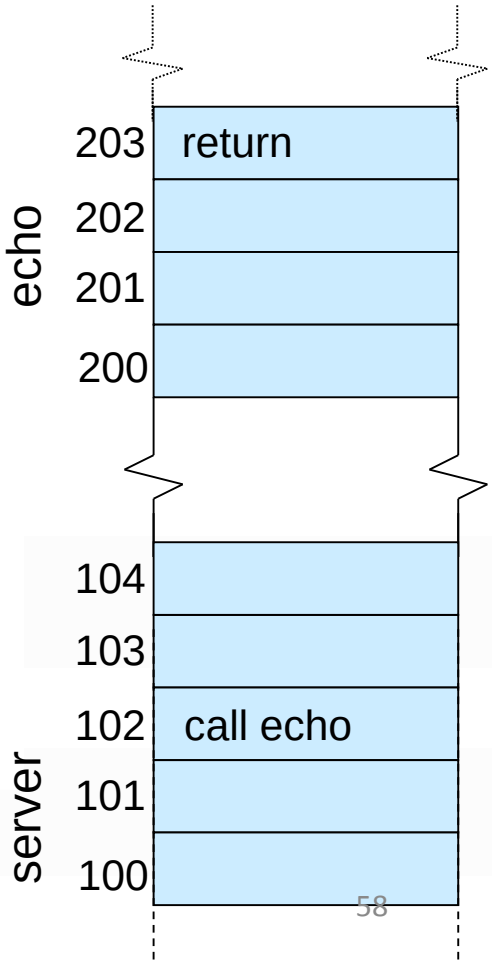
Code (echo)

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

read (fd, name, 128)

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

return;
```

Print a code address.

Then rerun program because then we can defeat ASLR.





Code (echo)

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

read (fd, name, 128)

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

return;
```

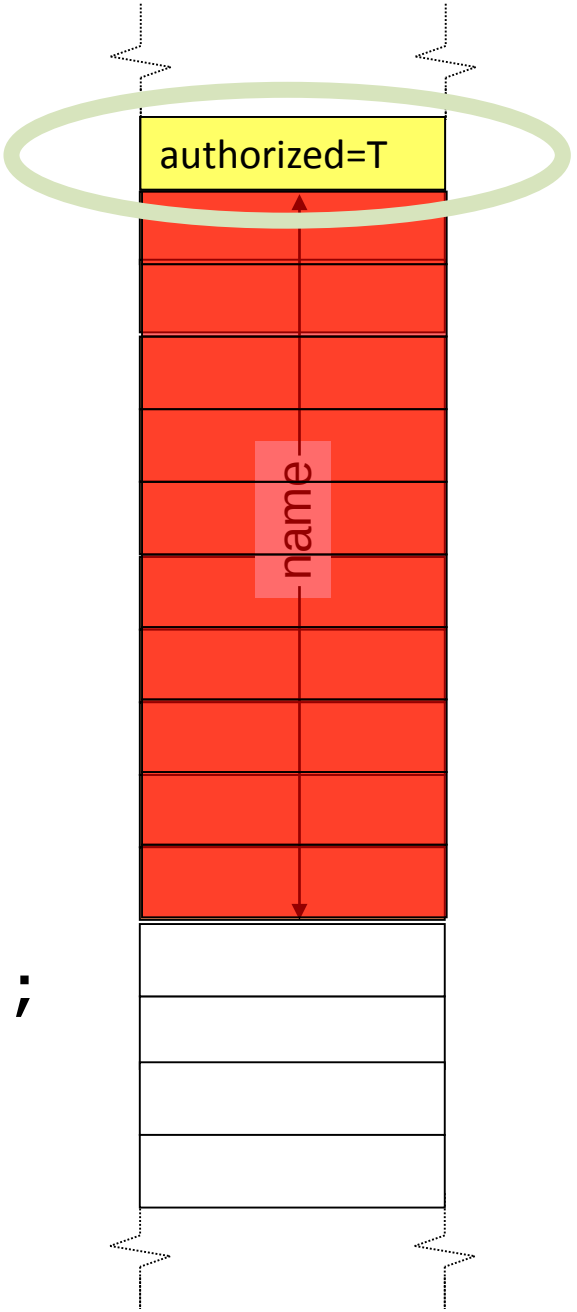
Finally

You may also overwrite other things

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

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”);
}
```



Memory Corruption

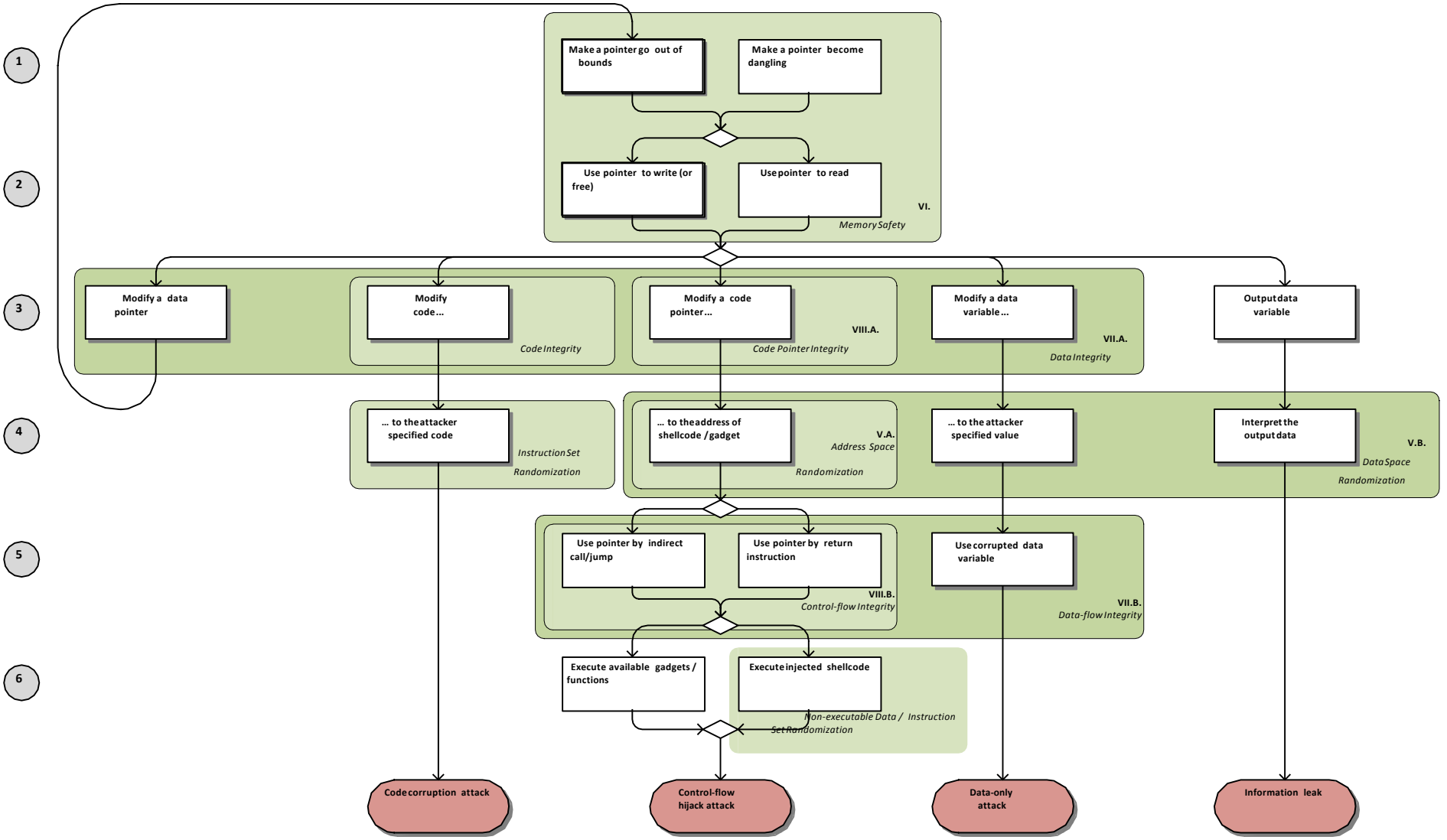
Summary

- We have sketched only the most common memory corruption attack
  - many variations, e.g.:
    - heap ← → stack
    - more complex overflows
    - off-by-one
- But there are others also
  - integer overflows
  - format string attacks
  - double free
  - etc.
- Not now, perhaps later…

*different kinds

We constructed “weird machines”

- New spin on fundamental questions:
  - ➔ “What is computable?”
- Shellcode, ROP, Ret2Libc
  - ➔ Turing Complete



https://www.cs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf

That is all folks!

- We have covered quite a lot:
  - Simple buffer overflows
  - Counter measures
  - Counter counter measures
- Research suggests that buffer overflows will be with us for quite some time
- Best avoid them in your code!