



# Framing Signals

## a return to portable shellcode

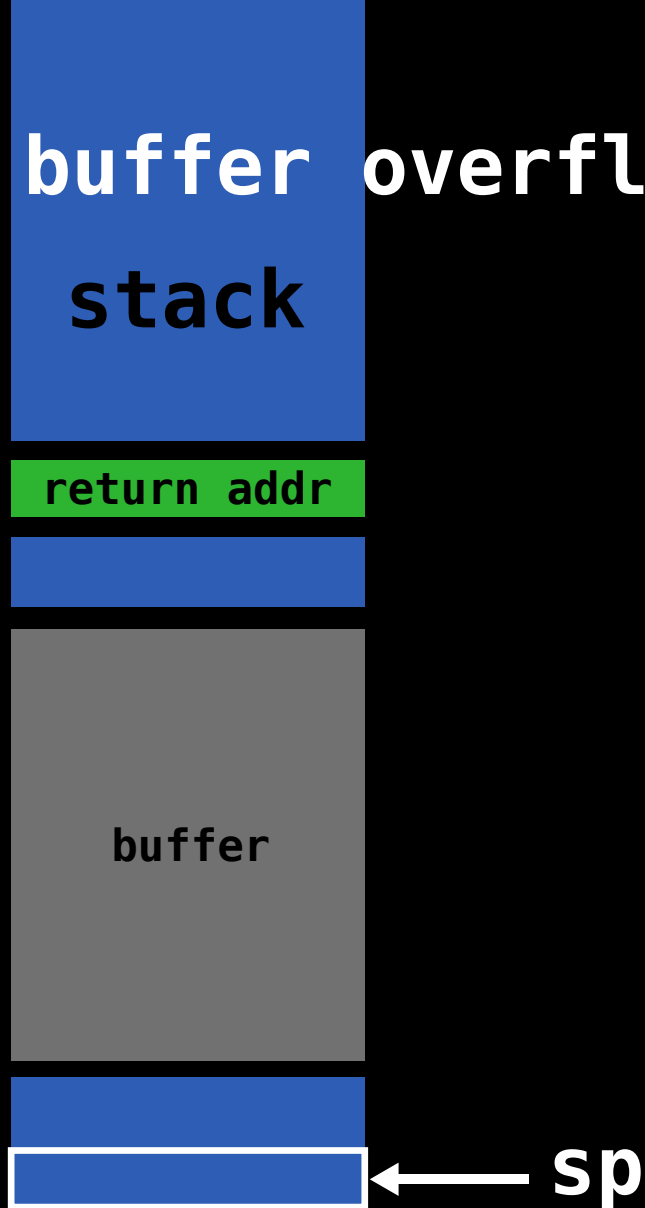
**Erik Bosman** and Herbert Bos

memory corruption,

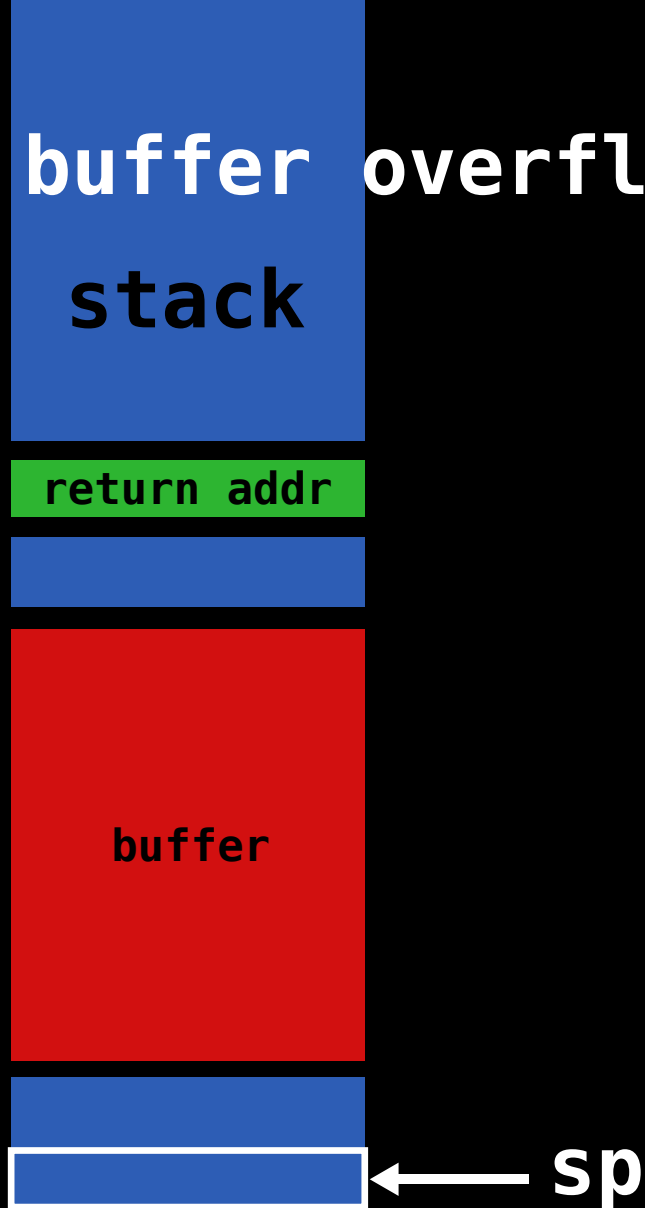
the problem that just  
won't go away

25+ years after the morris worm and  
still going strong

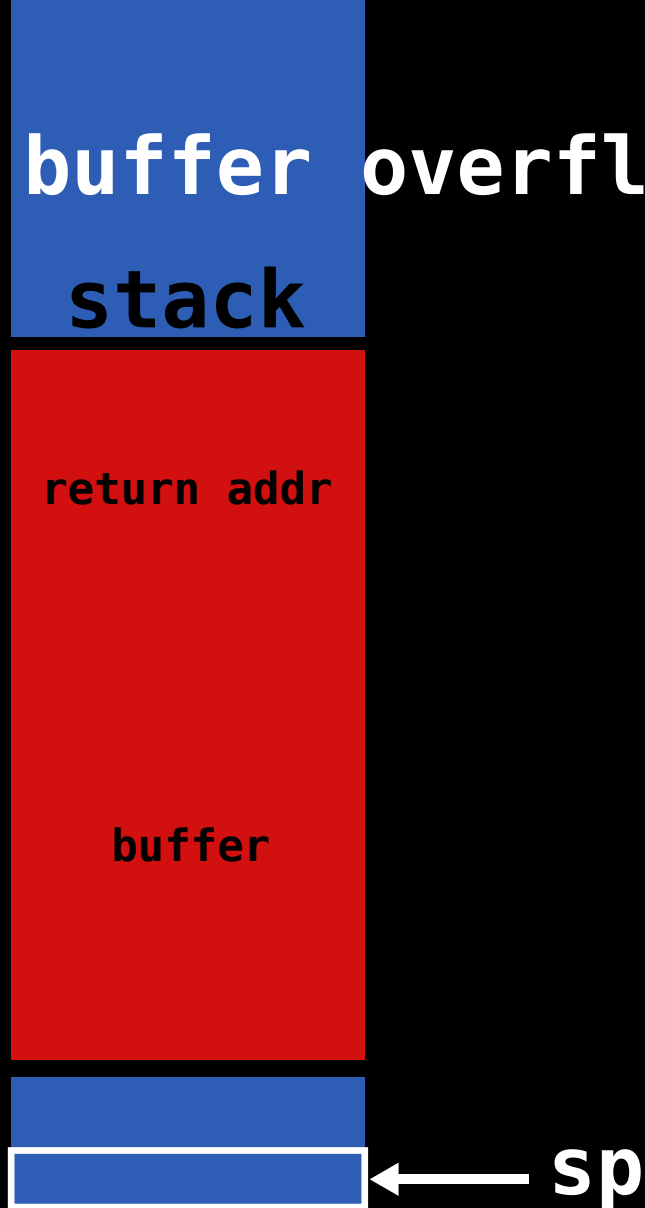
**stack buffer overflow**



**stack buffer overflow**



# stack buffer overflow



# stack buffer overflow

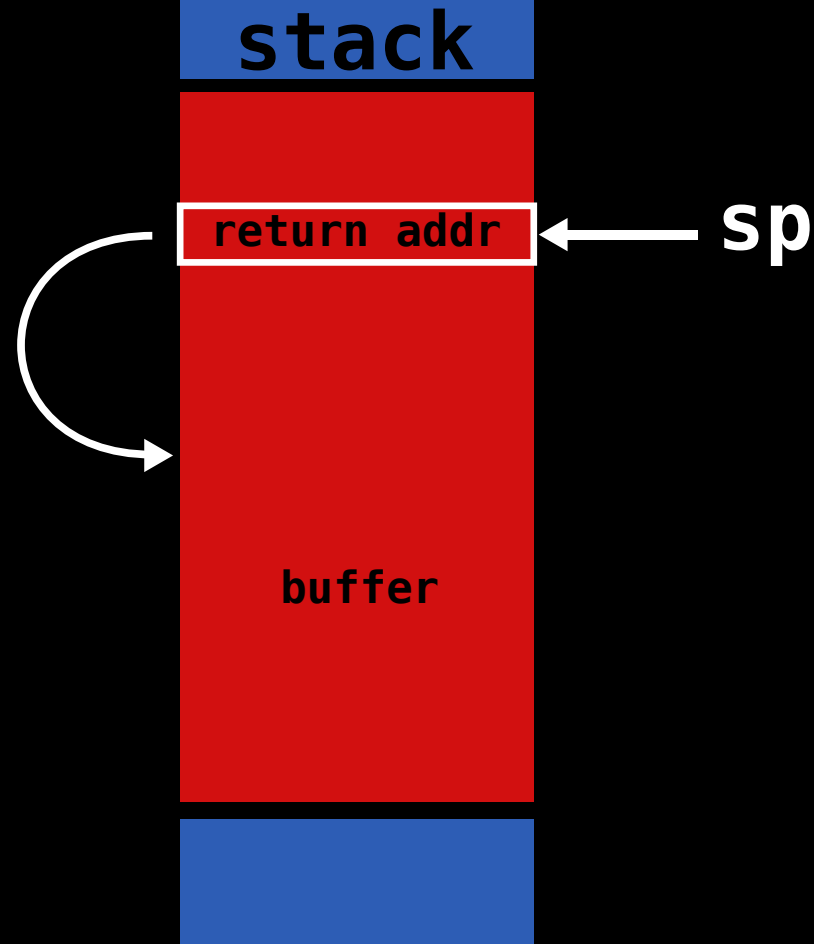
stack

return addr

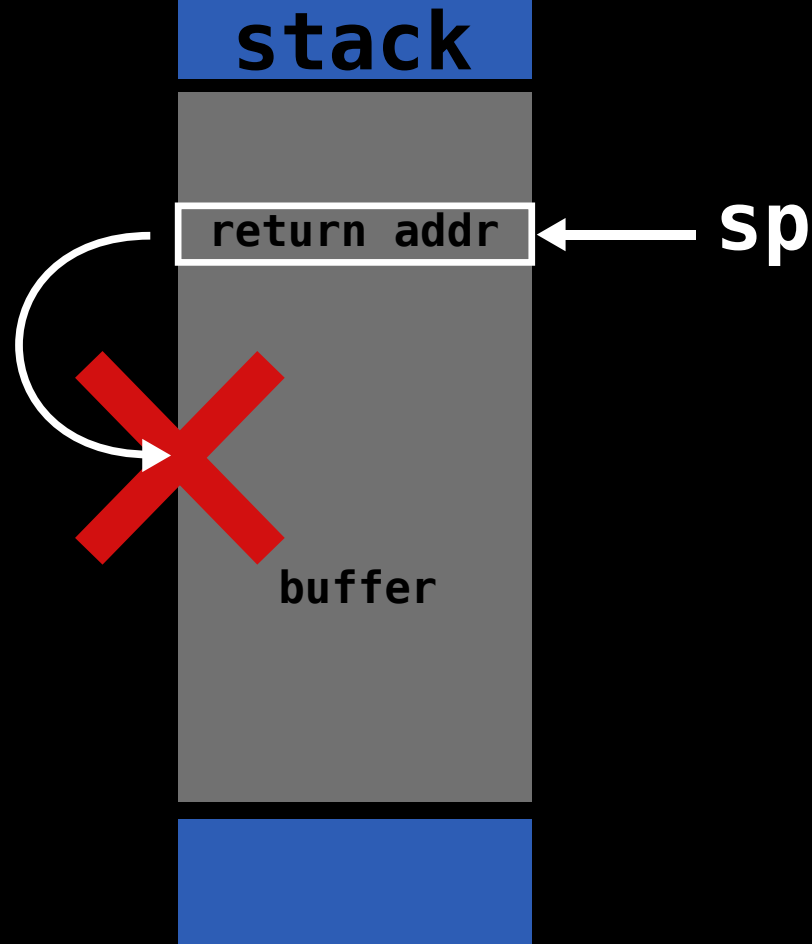
← sp

buffer

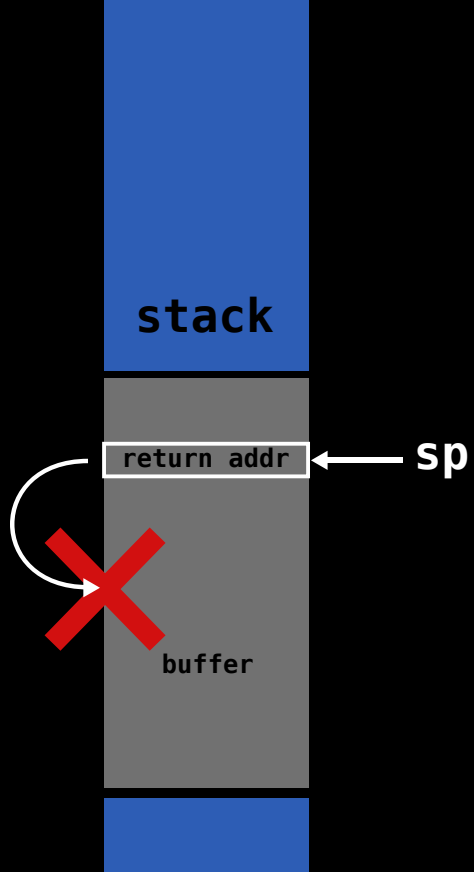
# stack buffer overflow

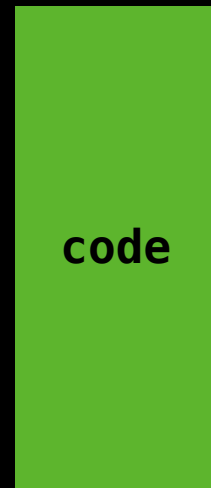
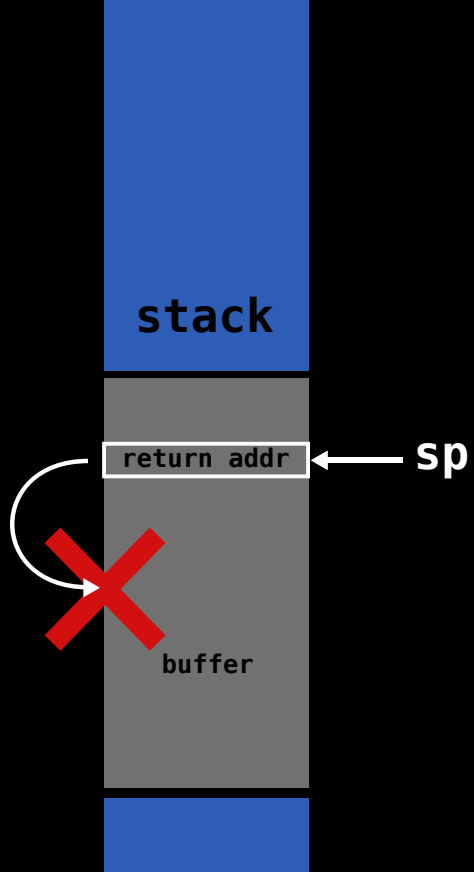


# stack buffer overflow









# return oriented programming

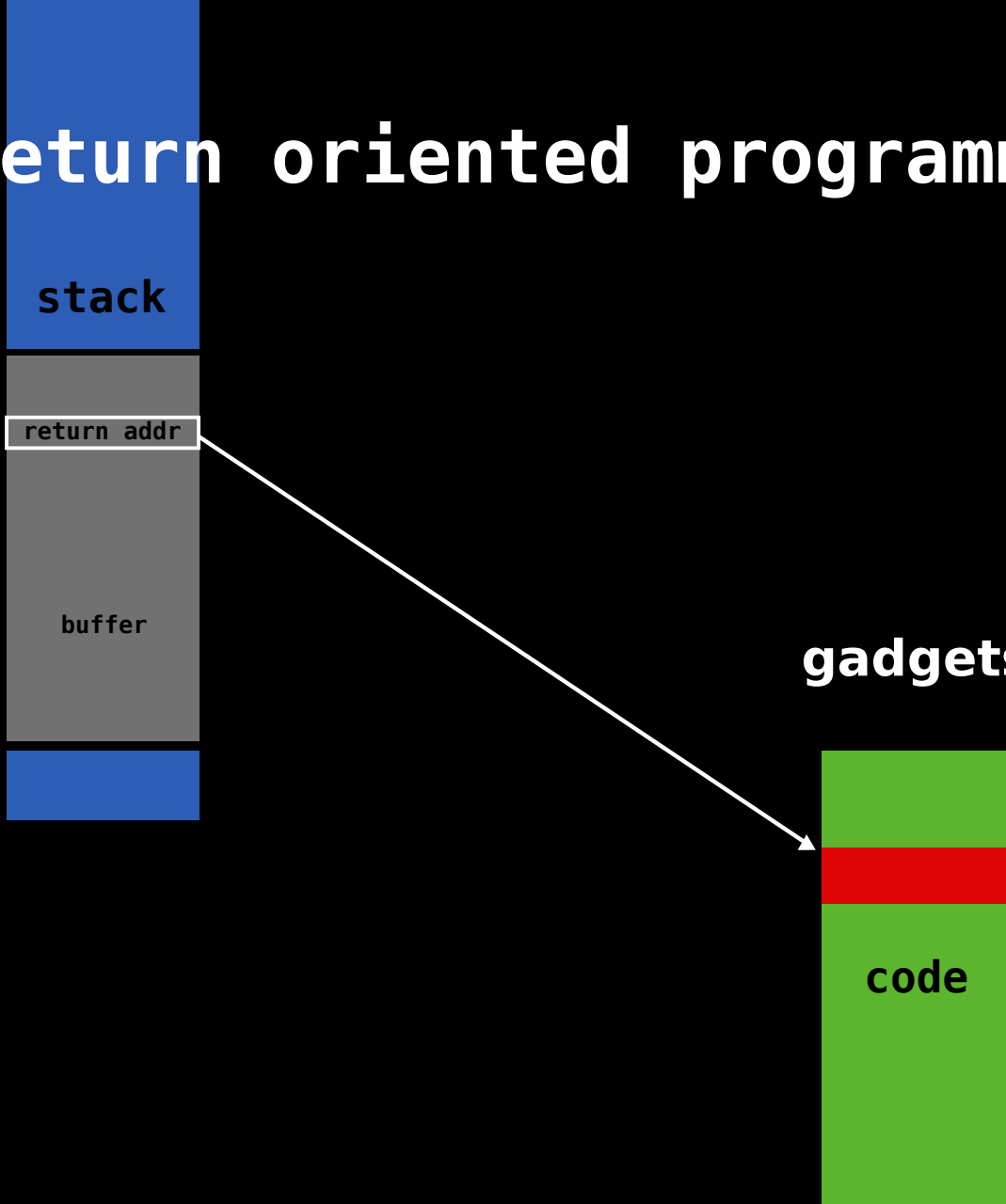
stack

return addr

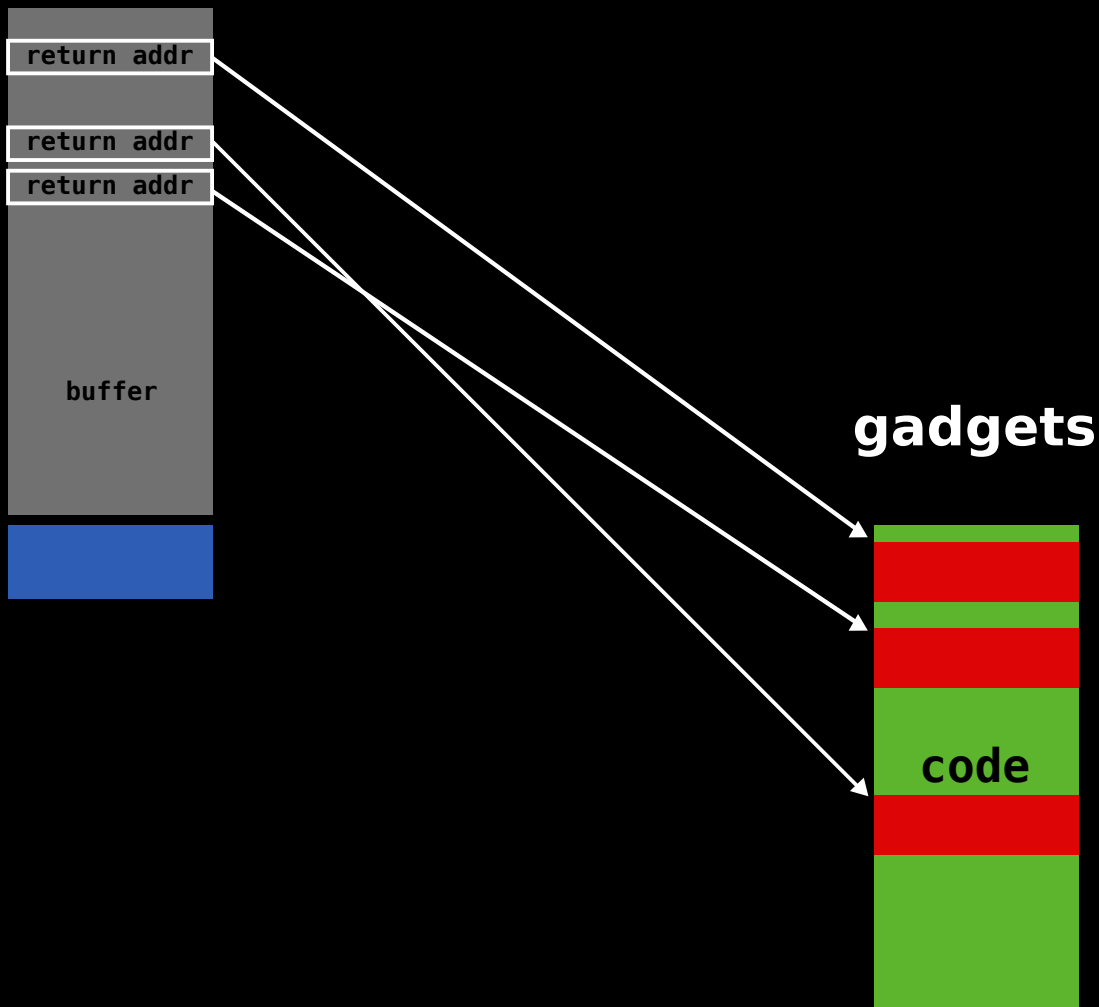
buffer

gadgets

code



# return oriented programming



# Return Oriented Programming

- dependent on available gadgets
- non-trivial to program
- chains may differ greatly between different binaries
- Turing complete

# Sigreturn Oriented Programming

- minimal number of gadgets
- constructing shellcode by chaining system calls
- easy to change functionality of shellcode
- shellcode portable (gadgets are always present)
- Turing complete

**unix signals**  
**stack**



# unix signals stack





# unix signals stack

ucontext

siginfo

← sp

# unix signals stack

ucontext

siginfo

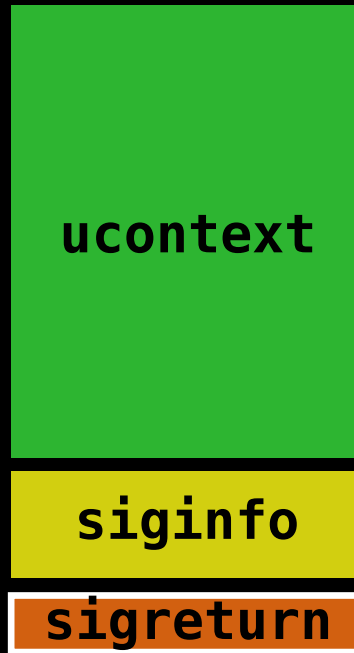
sigreturn

← sp

# unix signals stack

good:

kernel agnostic  
about signal  
handlers

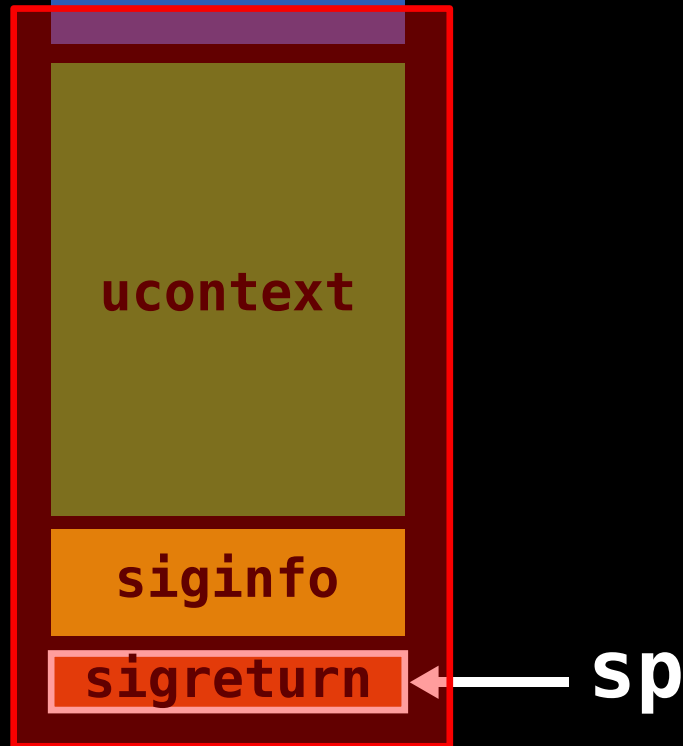


# unix signals stack

bad:

kernel agnostic  
about signal  
handlers

(we can fake 'em)



# two gadgets

- call to sigreturn
- syscall & return

forged signal frame

sigreturn

program counter

forged signal frame

sigreturn

program counter

stack pointer

forged signal frame

sigreturn



program counter

stack pointer

RAX

...

RDI

RSI

RDY

R10

R8

R9

...

sigreturn

program counter

stack pointer

syscall number

...

arg1

arg2

arg3

arg4

arg5

arg6

...

sigreturn

syscall & return

stack pointer

syscall number

...

arg1

arg2

arg3

arg4

arg5

arg6

...

sigreturn

syscall & return

next sigframe

syscall number

...

arg1

arg2

arg3

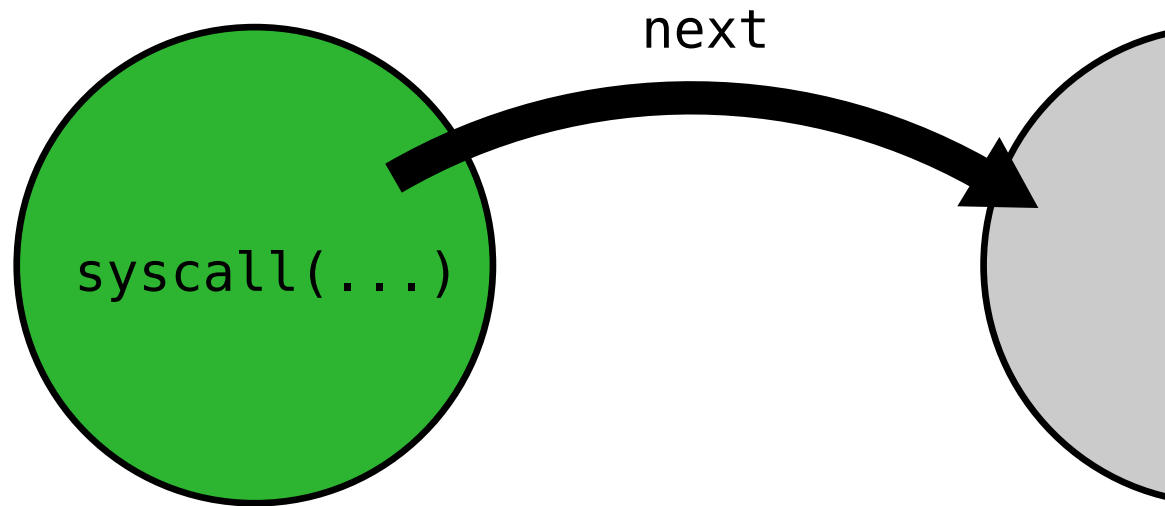
arg4

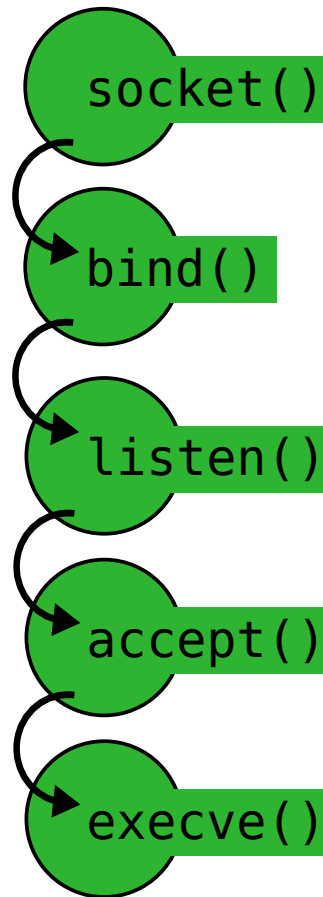
arg5

arg6

...

sigreturn





# SR0P exploit on x86-64

An exploit which does not make use of any gadgets in the target program

- control over the stack
- a known writable memory location

(\*any\* location, and we don't need to write there beforehand)

# two gadgets

- call to sigreturn
- syscall & return



## two gadgets

- call to sigreturn:  $RAX = 15 + syscall$
- syscall & return

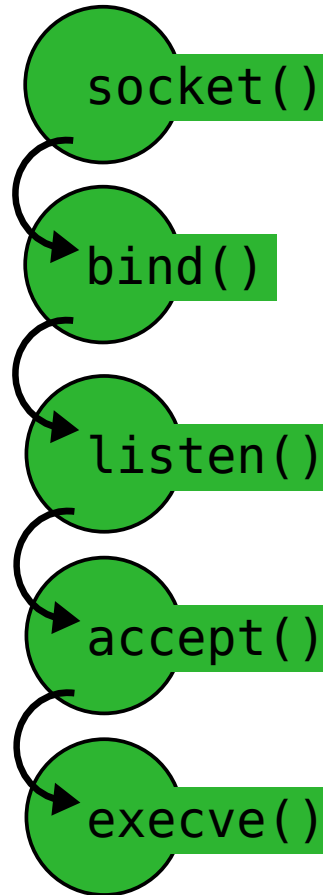
# one gadget

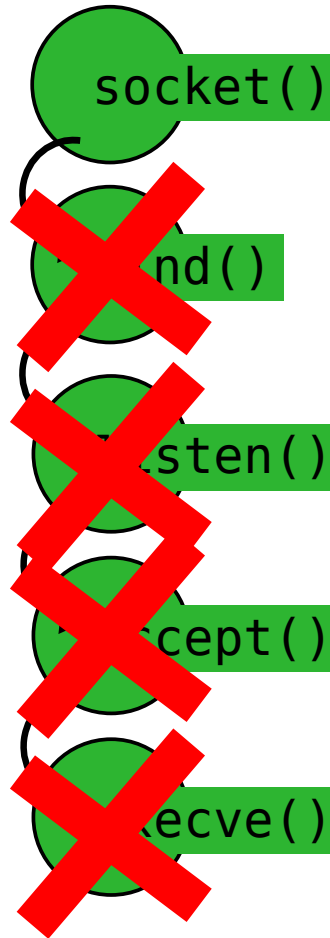
- RAX = 15
- syscall & return

# [vsyscall]

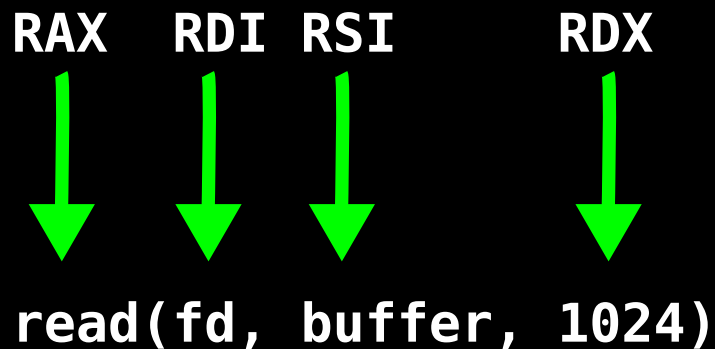
```
ffffffffffff600000 48 c7 c0 60 00 00 00 0f 05 c3 cc cc cc cc cc cc cc gettimeofday()
ffffffffffff60010 cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
*
ffffffffffff600400 48 c7 c0 c9 00 00 00 0f 05 c3 cc cc cc cc cc cc cc time()
ffffffffffff600410 cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
*
ffffffffffff600800 48 c7 c0 35 01 00 00 0f 05 c3 cc cc cc cc cc cc cc getcpu()
ffffffffffff600810 cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc cc
*
ffffffffffff601000
```

0f05 syscall  
c3 return





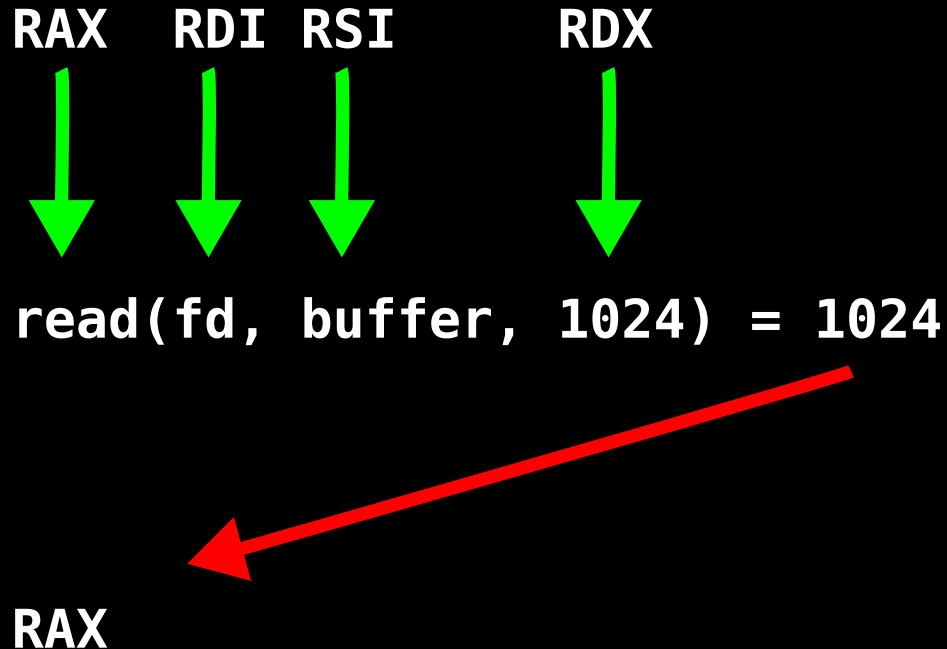
# x64 syscall ABI



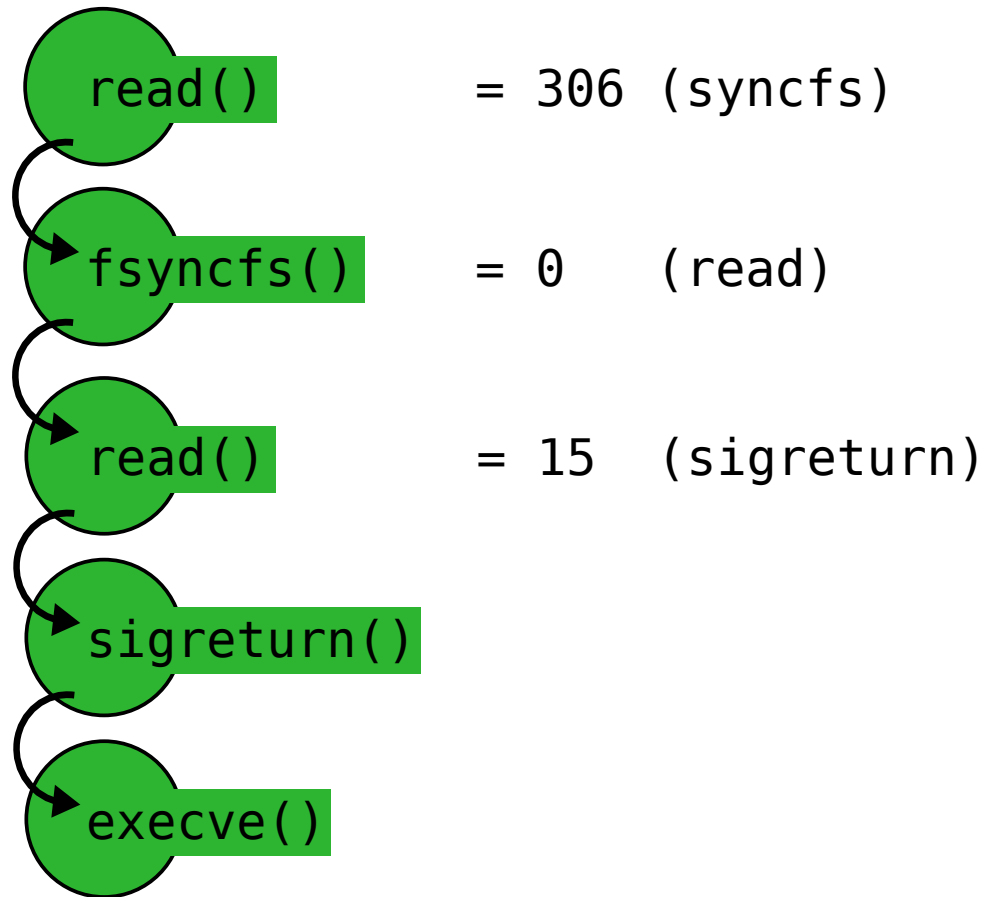
# x64 syscall ABI

RAX	RDI	RSI	RDX
↓	↓	↓	↓
read(fd, buffer, 1024) = 1024			

# x64 syscall ABI







# CVE-2012-5976 (asterisk)

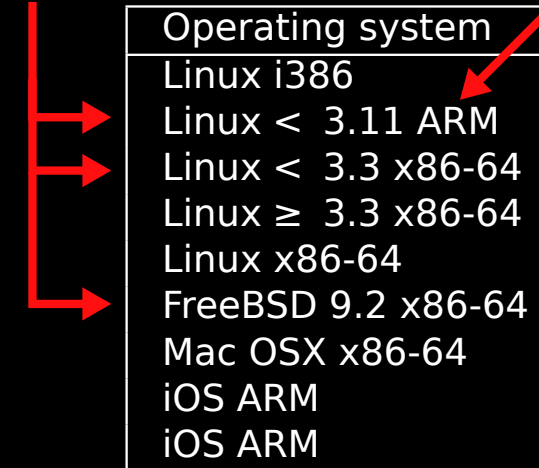


**On some systems SR0P gadgets  
are randomised, on others,  
they are not**

Operating system	Gadget	Memory map
Linux i386	sigreturn	[vdso]
Linux < 3.11 ARM	sigreturn	[vectors] 0xffff0000
Linux < 3.3 x86-64	syscall & return	[vsyscall] 0xffffffff600000
Linux ≥ 3.3 x86-64	syscall & return	Libc
Linux x86-64	sigreturn	Libc
FreeBSD 9.2 x86-64	sigreturn	0x7fffffff000
Mac OSX x86-64	sigreturn	Libc
iOS ARM	sigreturn	Libsystem
iOS ARM	syscall & return	Libsystem

On some systems SR0P gadgets  
are randomised, on others,  
they are not

non-ASLR :-( **android**



Operating system	Gadget	Memory map
Linux i386	sigreturn	[vdso]
Linux < 3.11 ARM	sigreturn	[vectors] 0xffff0000
Linux < 3.3 x86-64	syscall & return	[vsyscall] 0xffffffff600000
Linux ≥ 3.3 x86-64	syscall & return	Libc
Linux x86-64	sigreturn	Libc
FreeBSD 9.2 x86-64	sigreturn	0x7fffffff000
Mac OSX x86-64	sigreturn	Libc
iOS ARM	sigreturn	Libsystem
iOS ARM	syscall & return	Libsystem

questions?



**questions?**

# mitigation:

It may be useful to disable  
vsyscall

vsyscall=emulate  
(default from Linux 3.3 onward)

or

vsyscall=none

# mitigation:

- Signal frame canaries



# stack canary

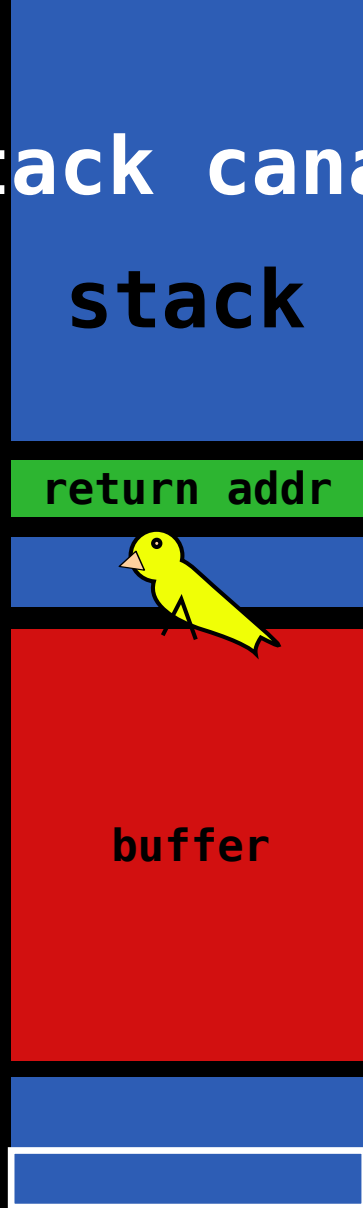
## stack

return addr



buffer

sp



# stack canary

## stack

return addr



buffer



sp

program counter

stack pointer

RAX

...

RDI

RSI

RDY

R10

R8

R9



sigreturn

program counter

stack pointer

RAX

...

RDI

RSI

RDY

R10

R8

R9



signature

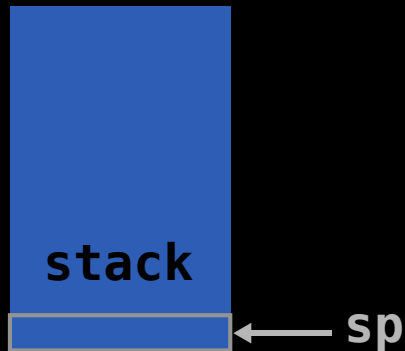
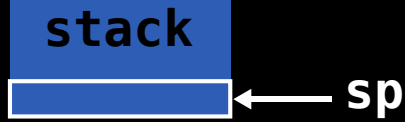
# mitigation:

- Signal frame canaries

# mitigation:

- Signal frame canaries
- Counting signals in progress

# CVE-2012-5976 (asterisk)



# CVE-2012-5976 (asterisk)

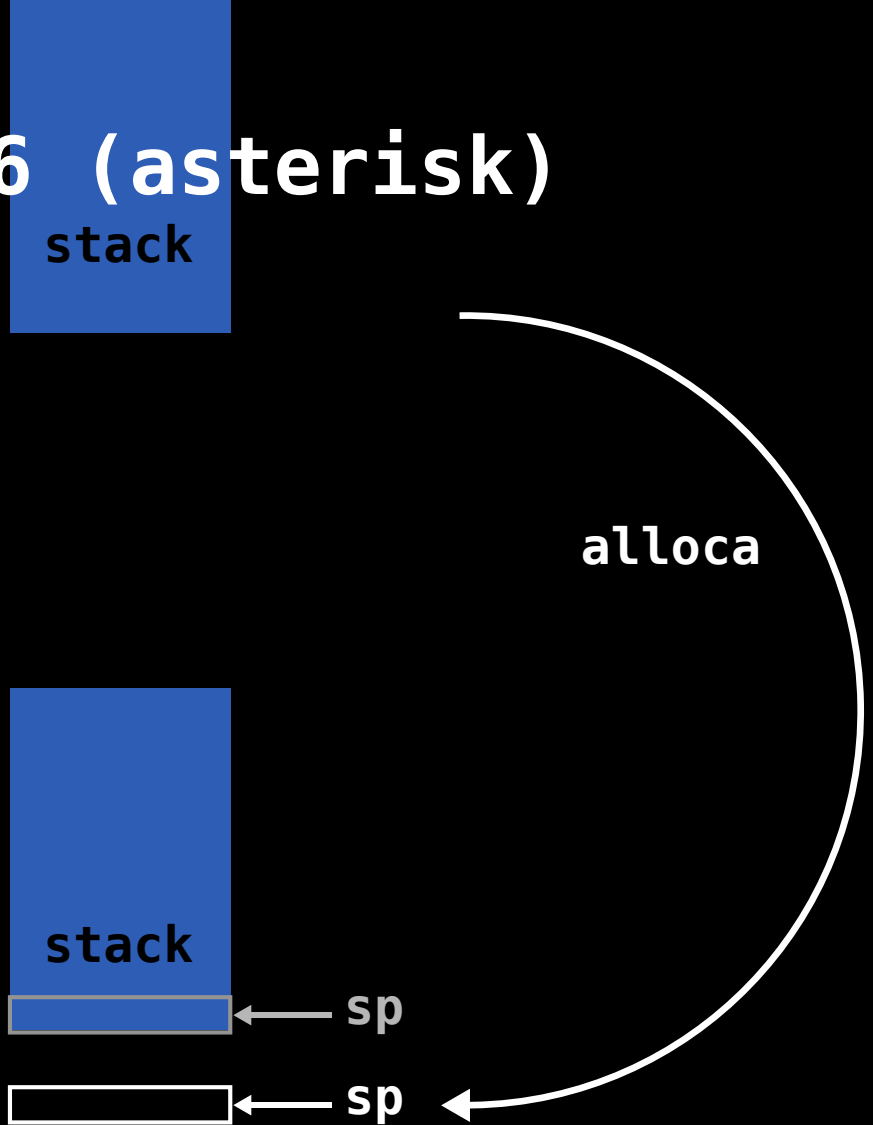
stack

stack

alloca

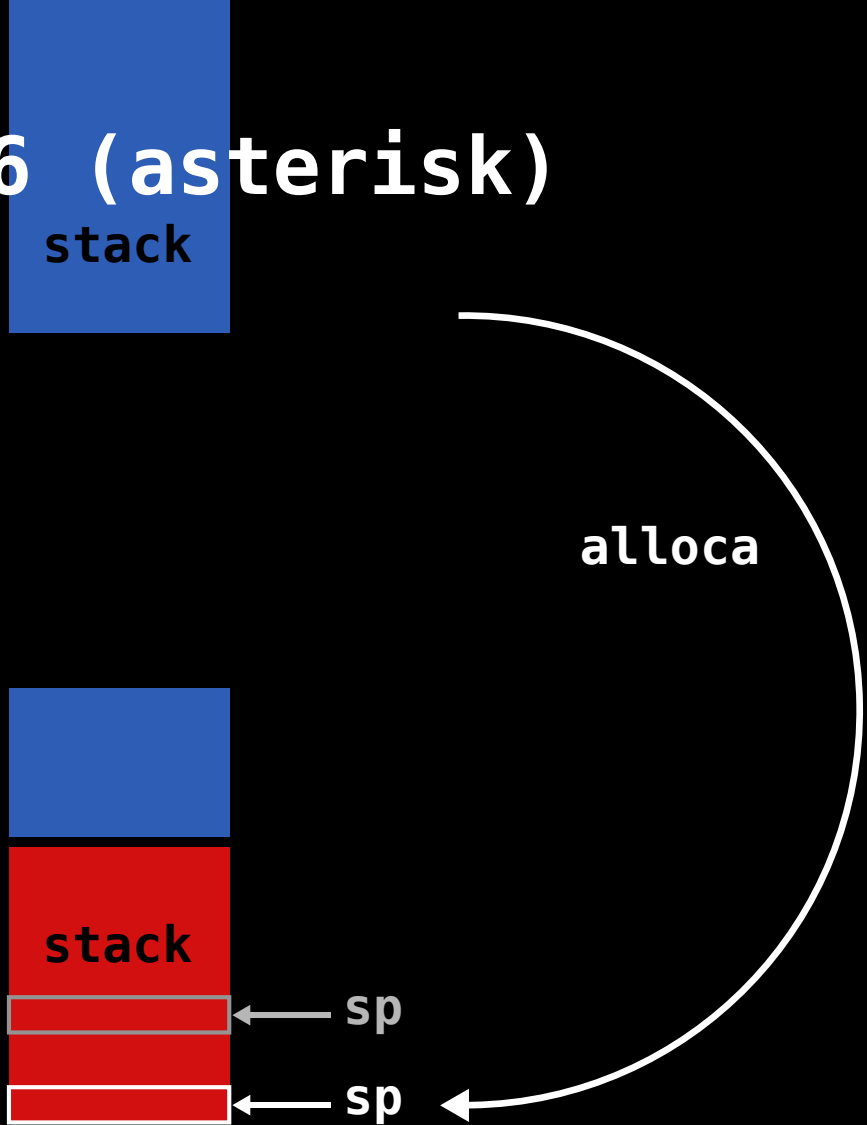
sp

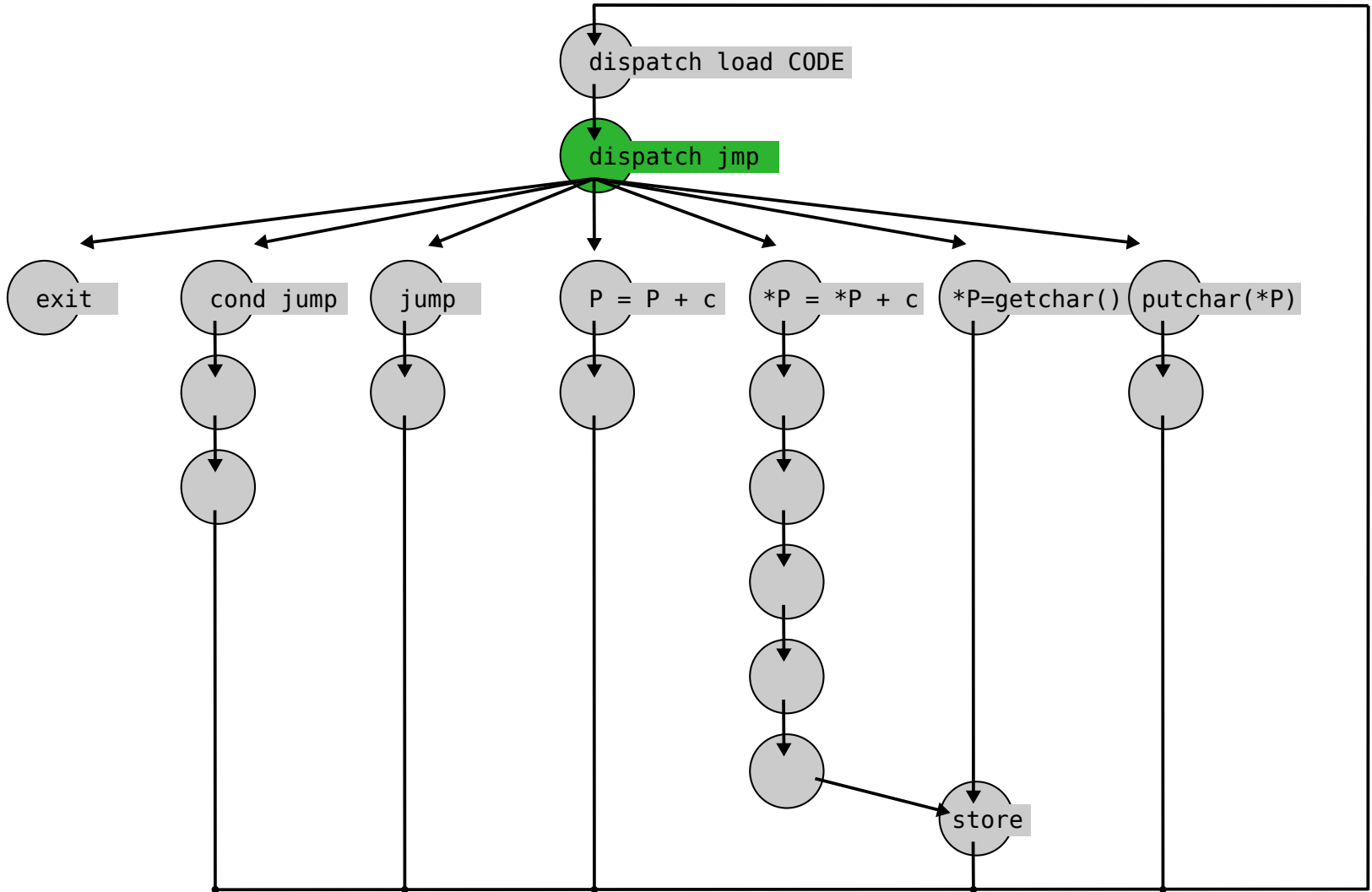
sp





# CVE-2012-5976 (asterisk)





```
code = open("/proc/self/mem", O_RDWR);  
p = open("/proc/self/mem", O_RDWR);  
a = open("/proc/self/mem", O_RDWR);
```

```
code = open("/proc/self/mem", O_RDWR);  
p = open("/proc/self/mem", O_RDWR);  
a = open("/proc/self/mem", O_RDWR);
```

```
instruction dispatch:  
    read(code, &ucontext.sp, sizeof(long));
```

```
code = open("/proc/self/mem", O_RDWR);  
p = open("/proc/self/mem", O_RDWR);  
a = open("/proc/self/mem", O_RDWR);
```

```
instruction dispatch:  
    read(code, &ucontext.sp, sizeof(long));
```

```
pointer ops:  
    p++ -> lseek(p, 1, SEEK_CUR);
```

```
code = open("/proc/self/mem", O_RDWR);  
p = open("/proc/self/mem", O_RDWR);  
a = open("/proc/self/mem", O_RDWR);
```

```
instruction dispatch:  
    read(code, &ucontext.sp, sizeof(long));
```

```
pointer ops:  
    p++ -> lseek(p, 1, SEEK_CUR);
```

```
addition:  
    lseek(a, &identity_table_x2, SEEK_SET);  
    lseek(a, val1, SEEK_SET);  
    lseek(a, val2, SEEK_SET);  
    read(a, dest, 1);
```