Framing Signals

a return to portable shellcode

Erik Bosman and Herbert Bos

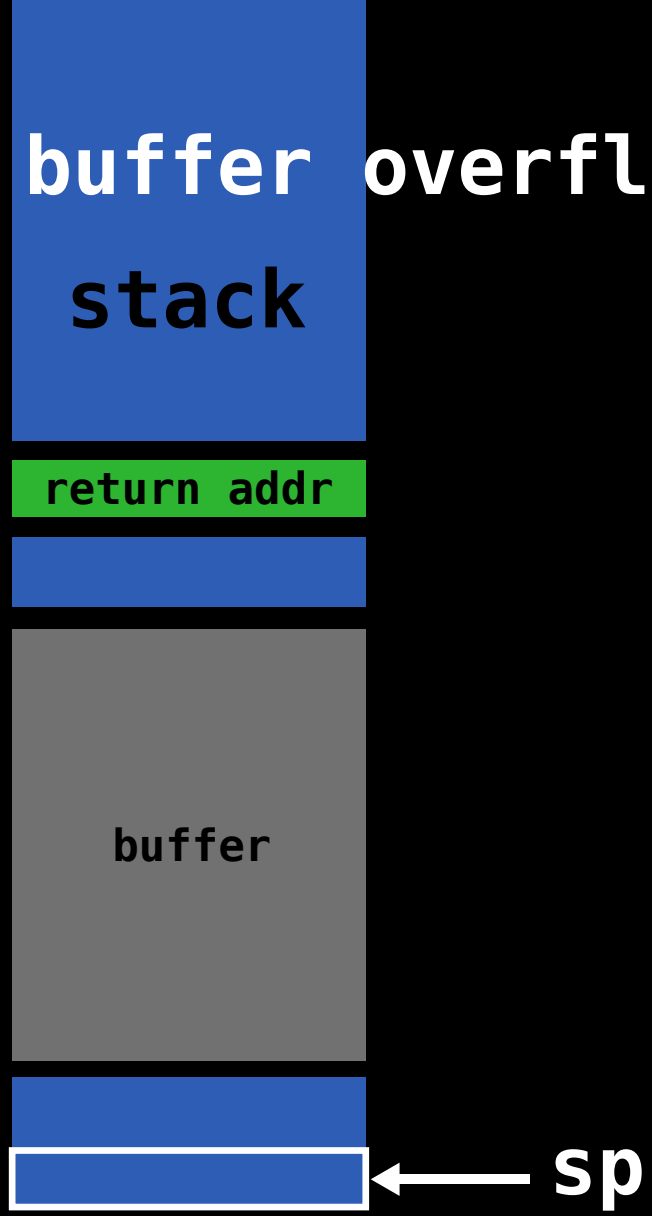
VU VRIJE UNIVERSITEIT AMSTERDAM

memory corruption,

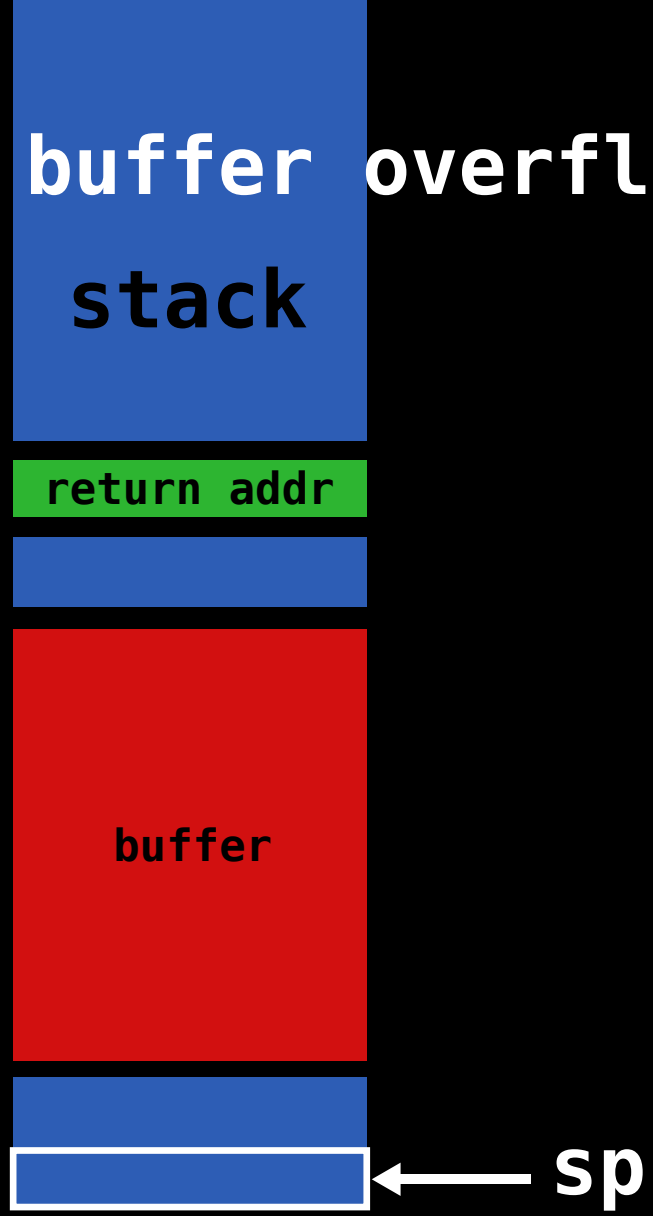
the problem that just won't go away

25+ years after the morris worm and still going strong

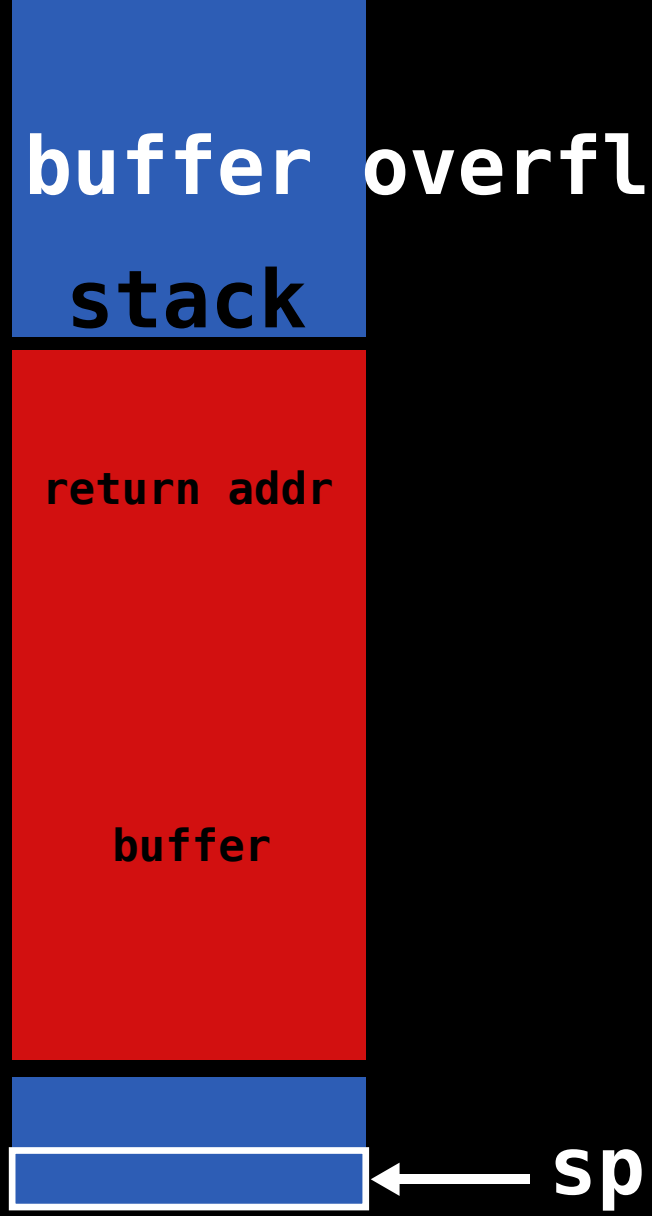
stack buffer overflow



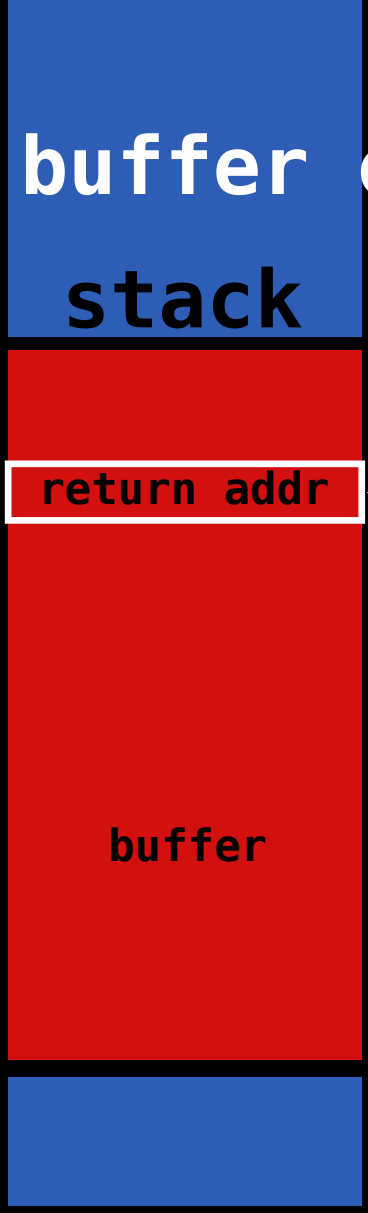
stack buffer overflow



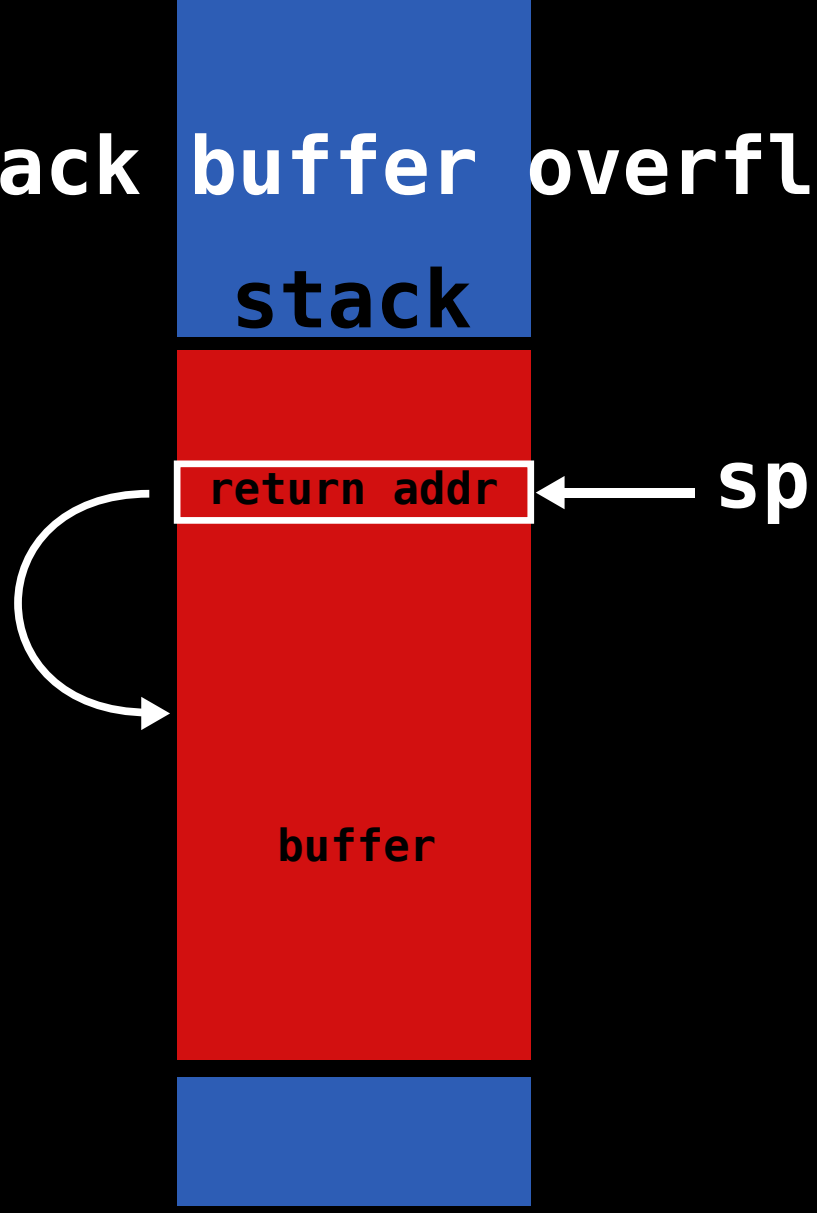
stack buffer overflow



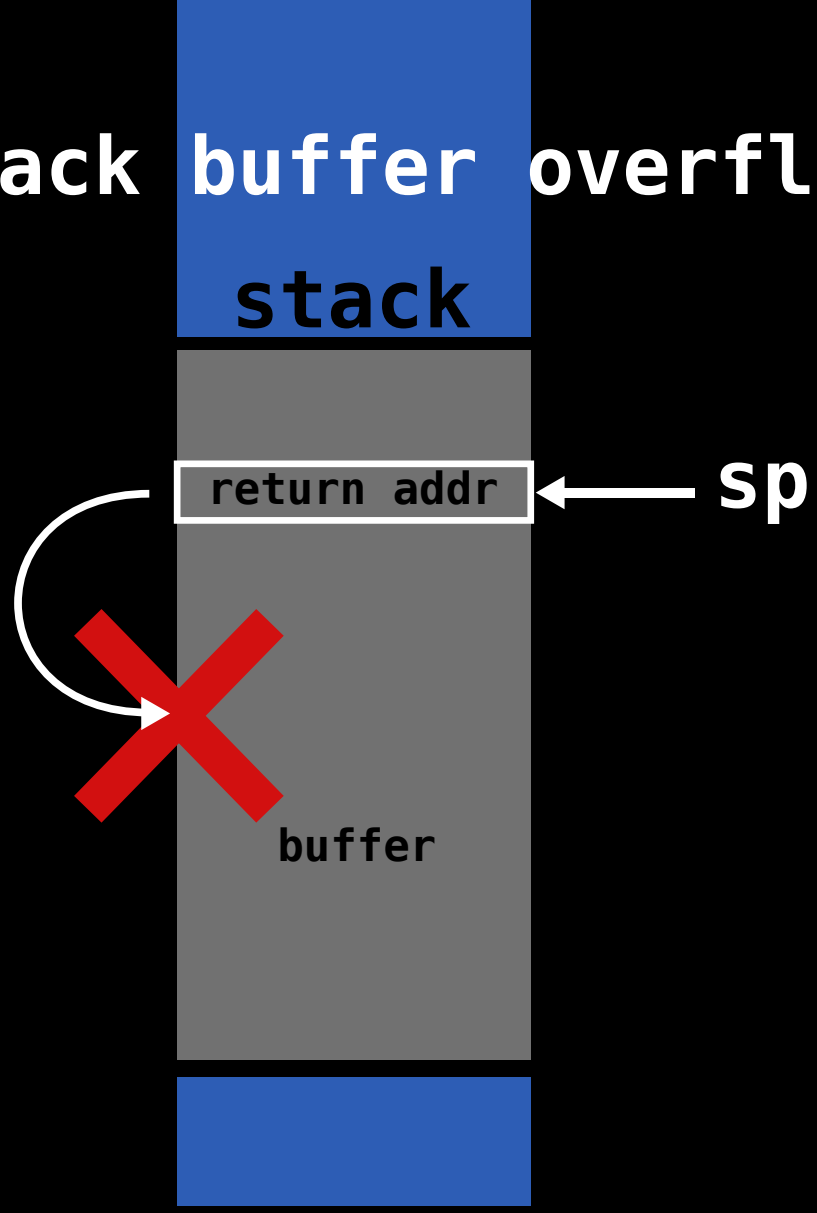
stack buffer overflow

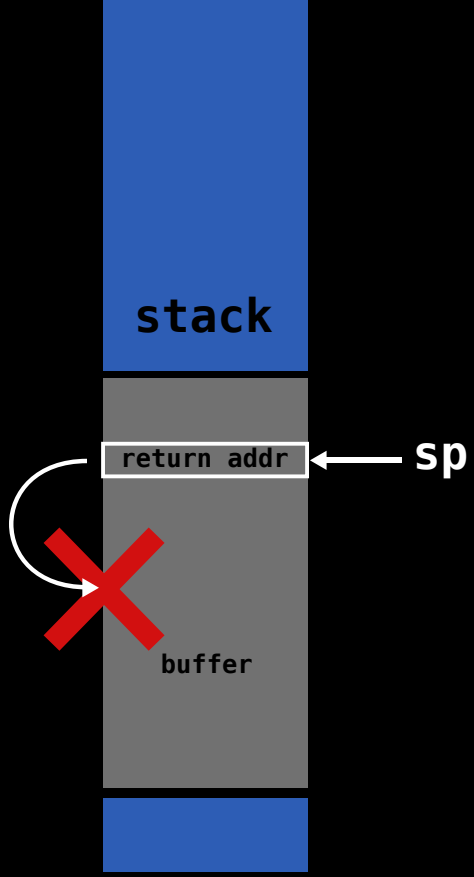


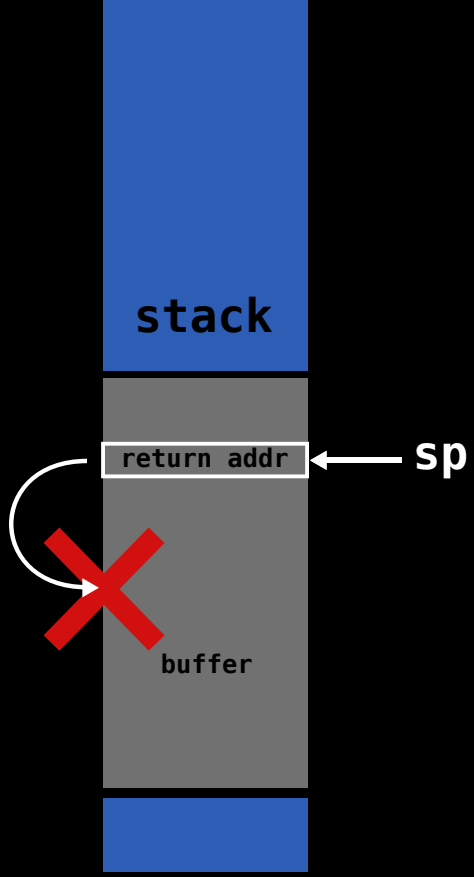
stack buffer overflow



stack buffer overflow

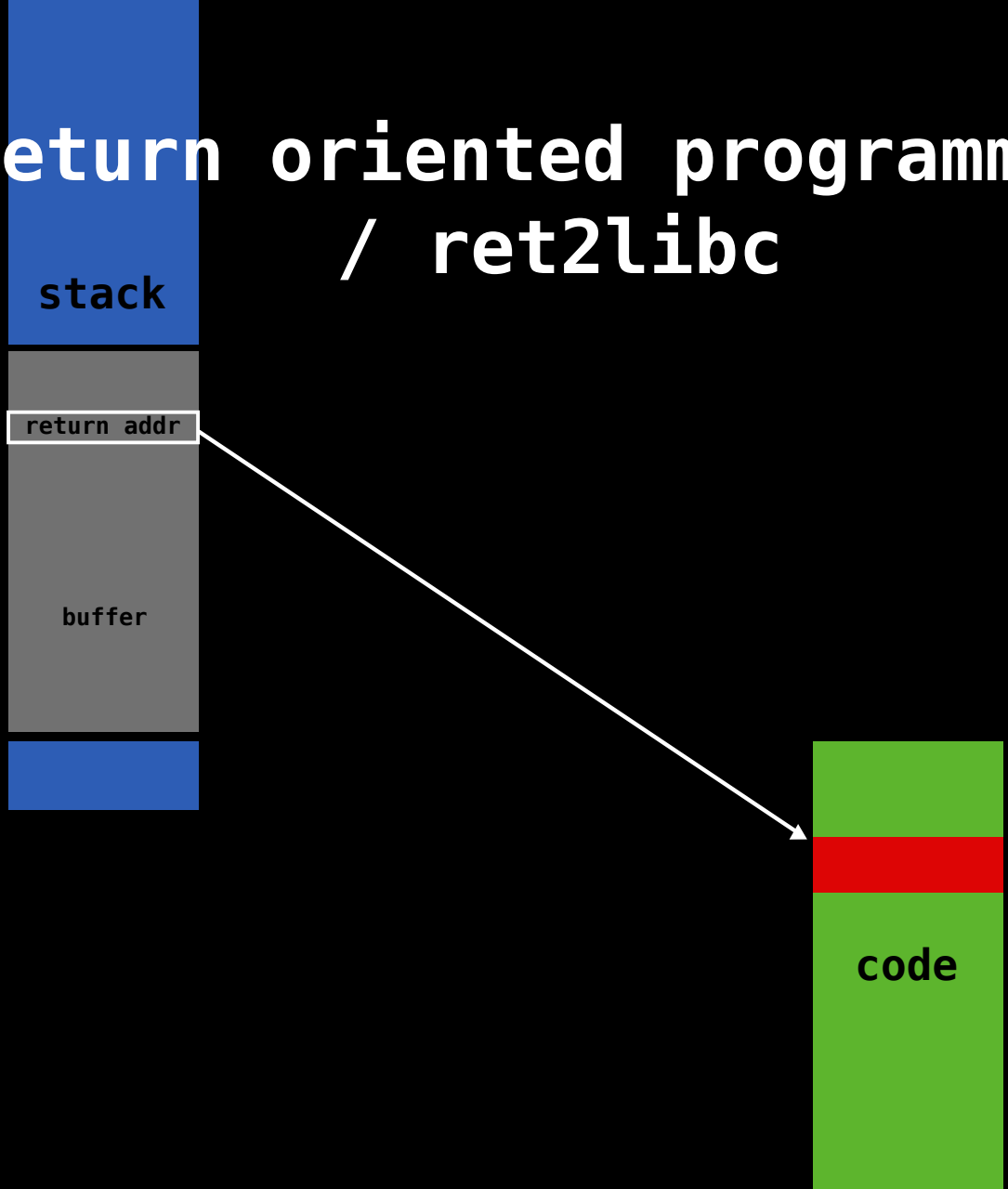



stack
return addr
sp
buffer

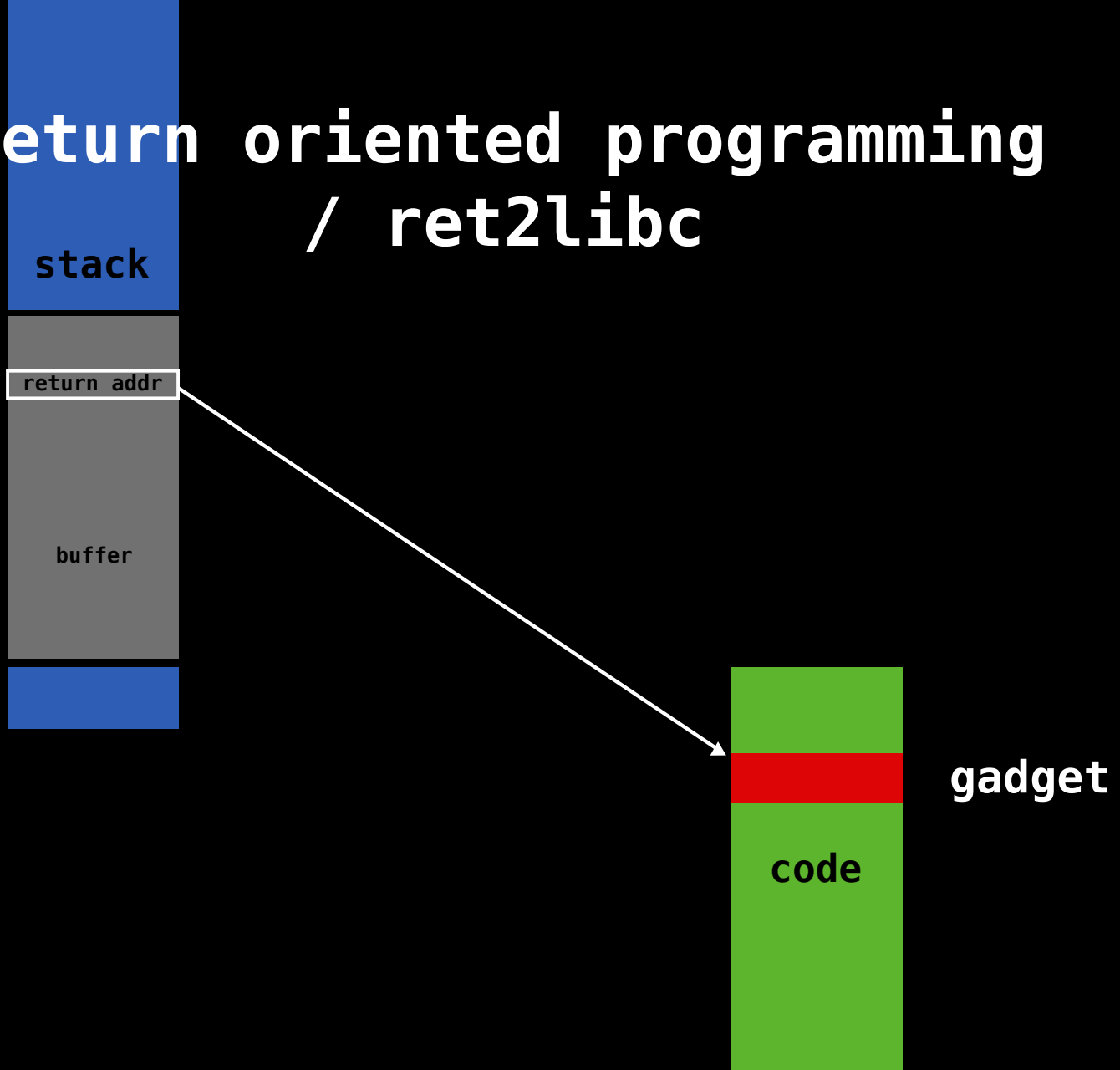


code

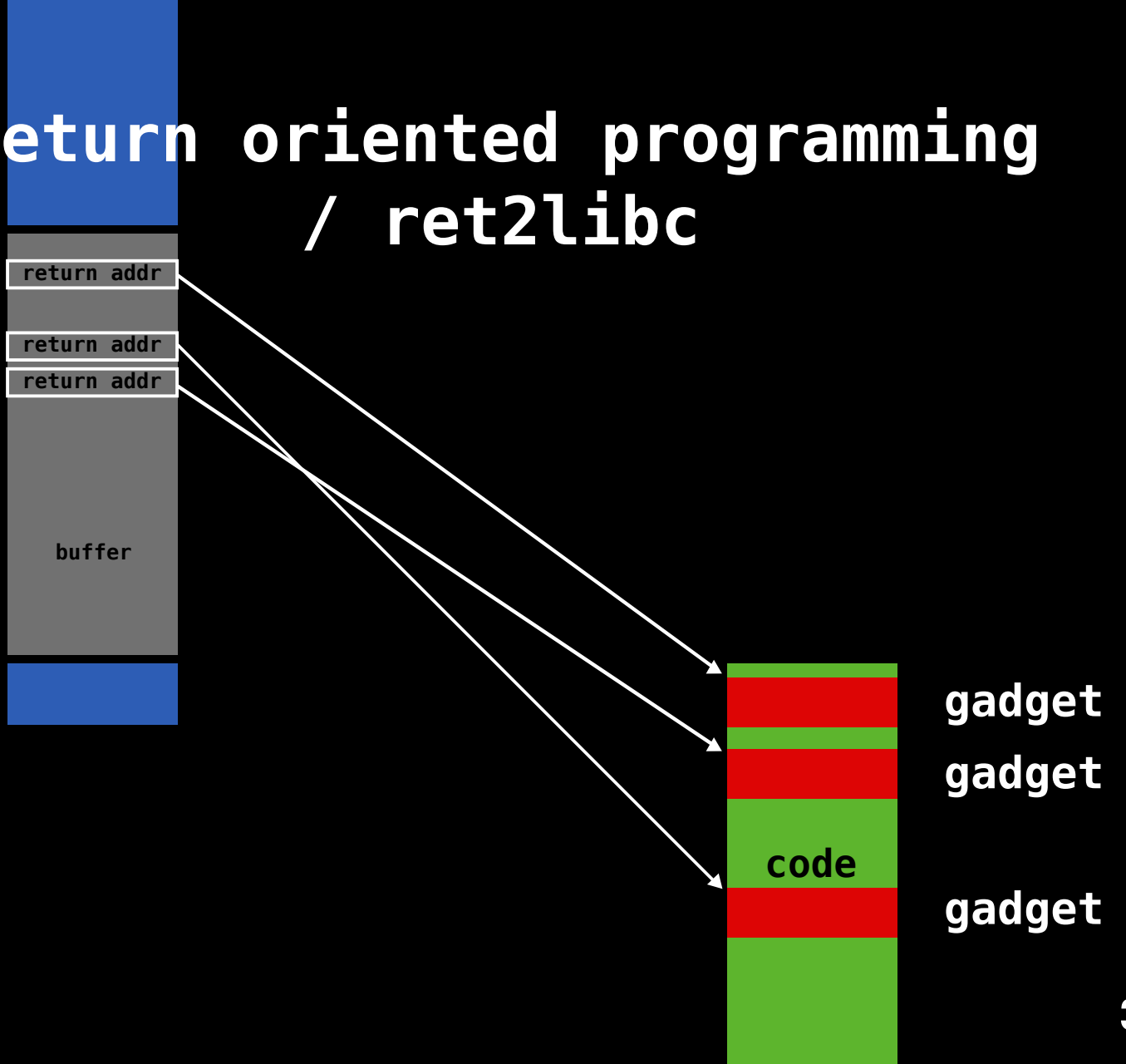
return oriented programming / ret2libc



return oriented programming / ret2libc



return oriented programming / ret2libc



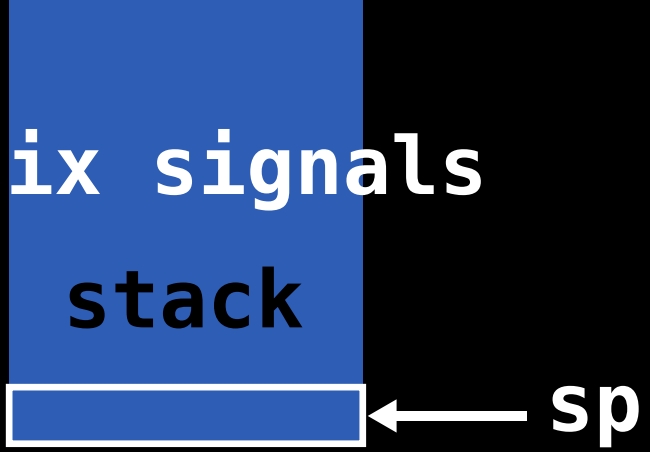
Return Oriented Programming

- 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

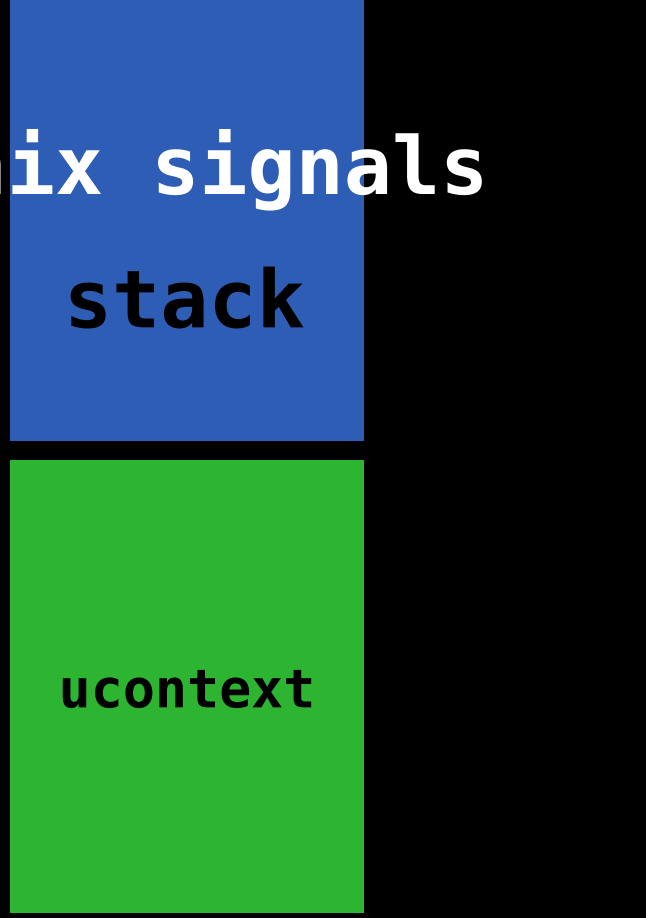
unix signals



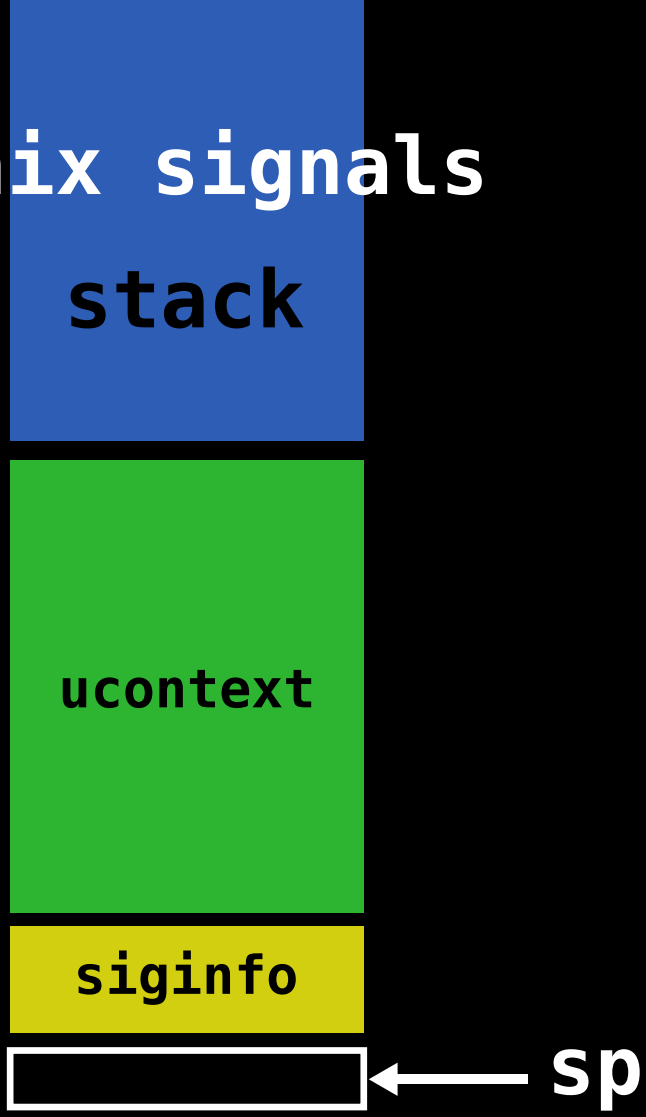
unix signals

stack

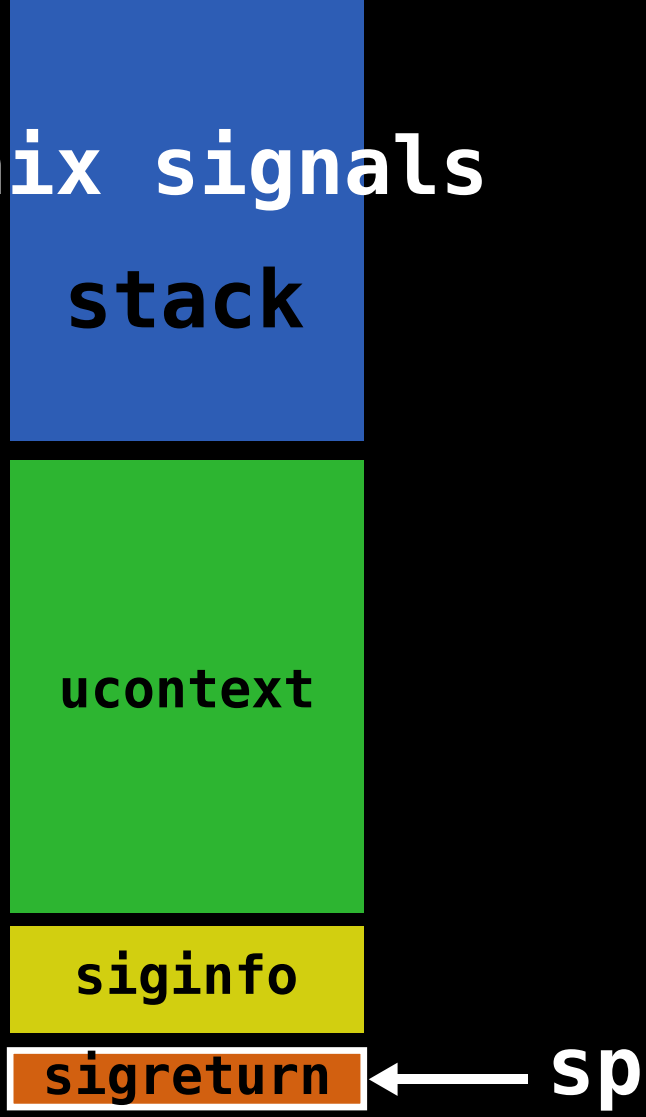
unix signals



unix signals



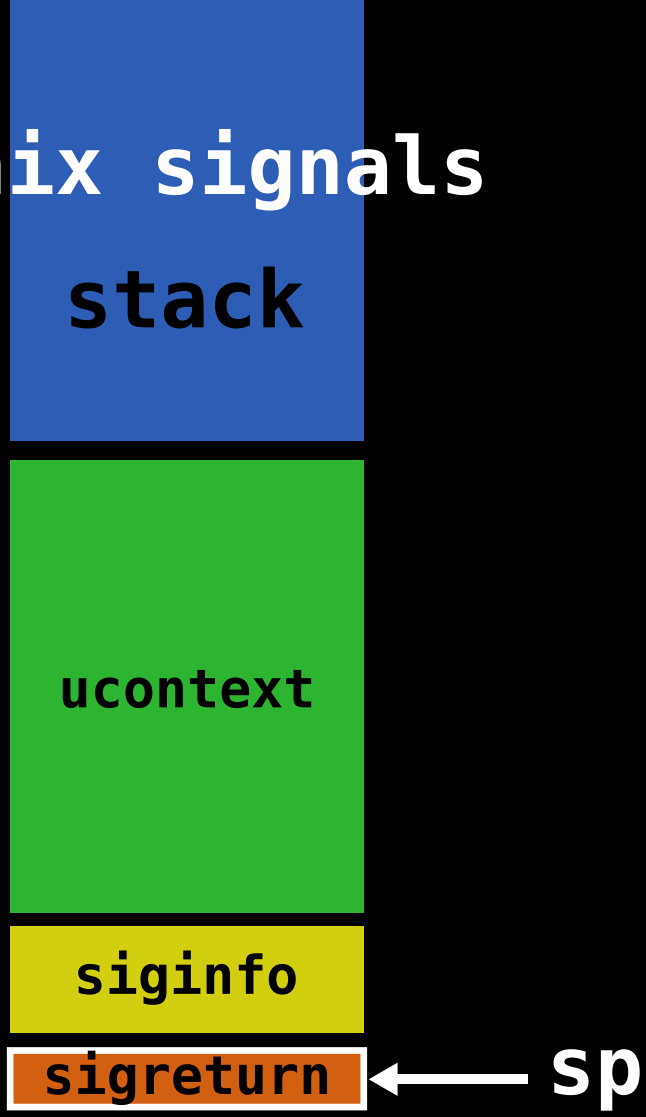
unix signals



unix signals

good:

kernel agnostic about signal handlers

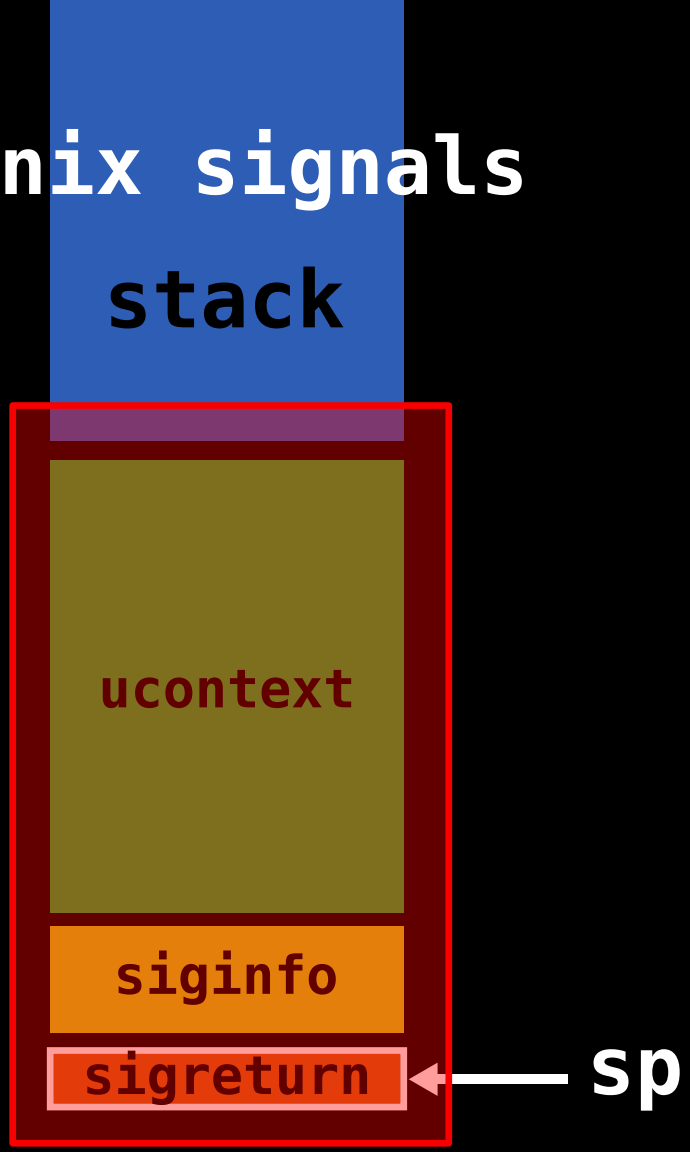


unix signals

bad:

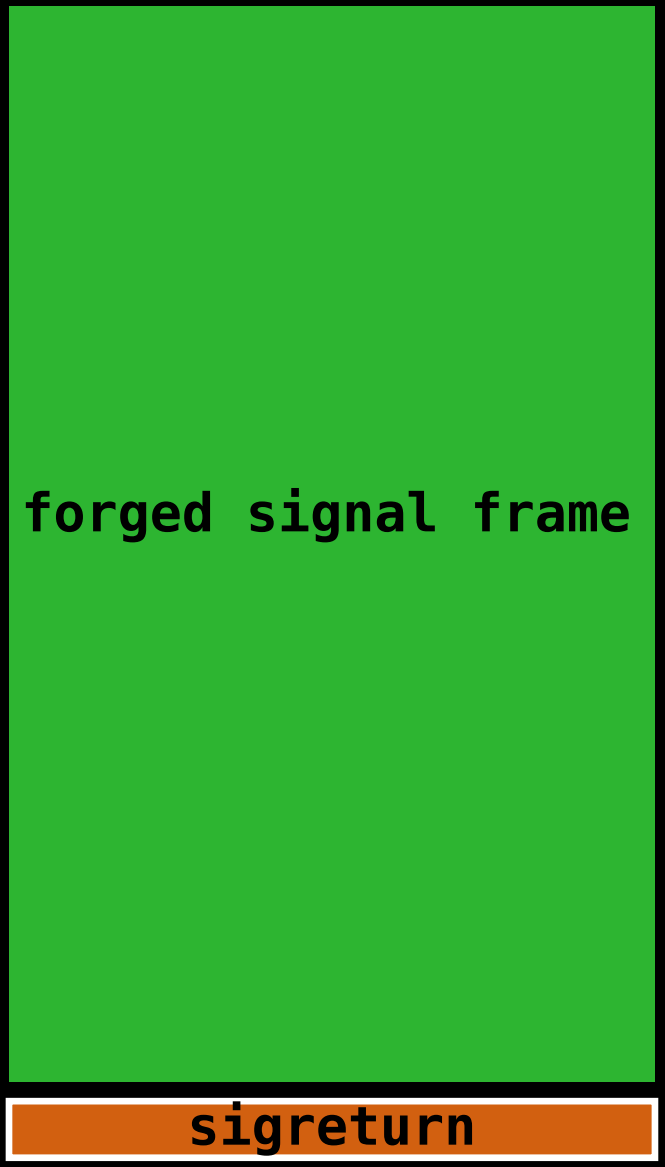
kernel agnostic about signal handlers

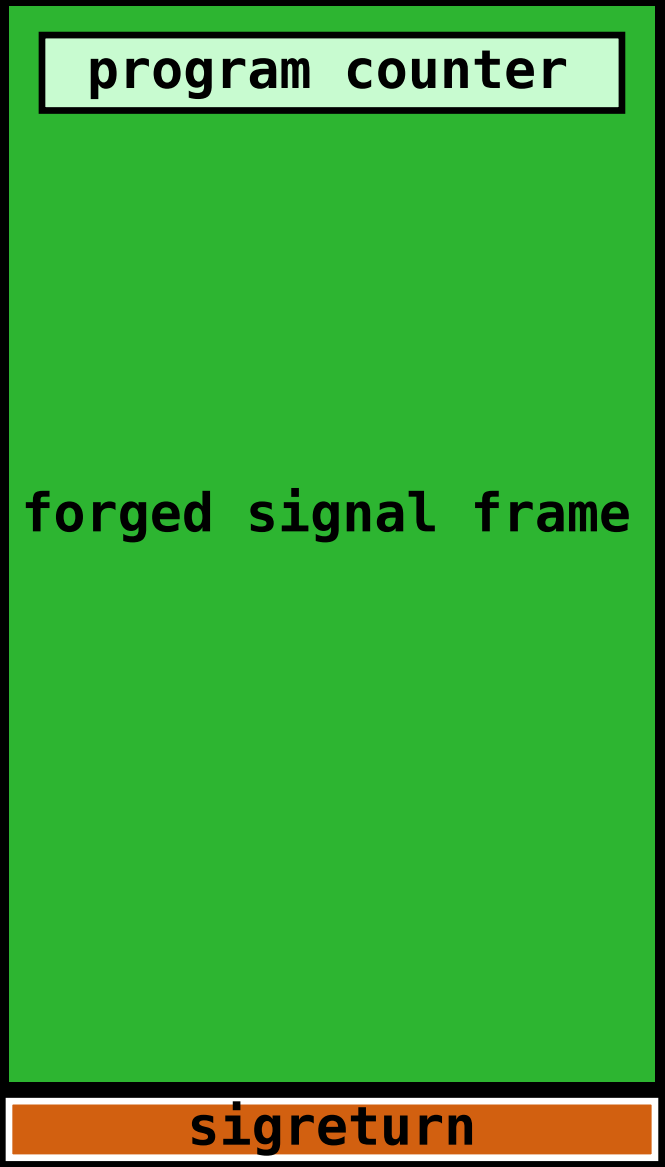
(we can fake 'em)

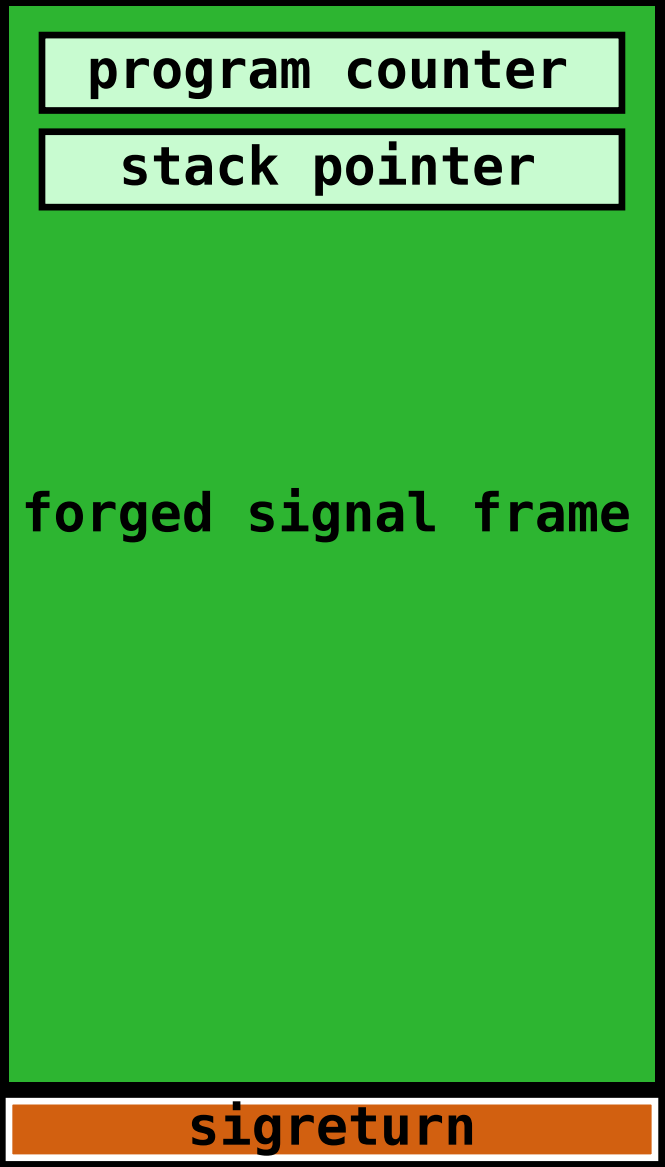


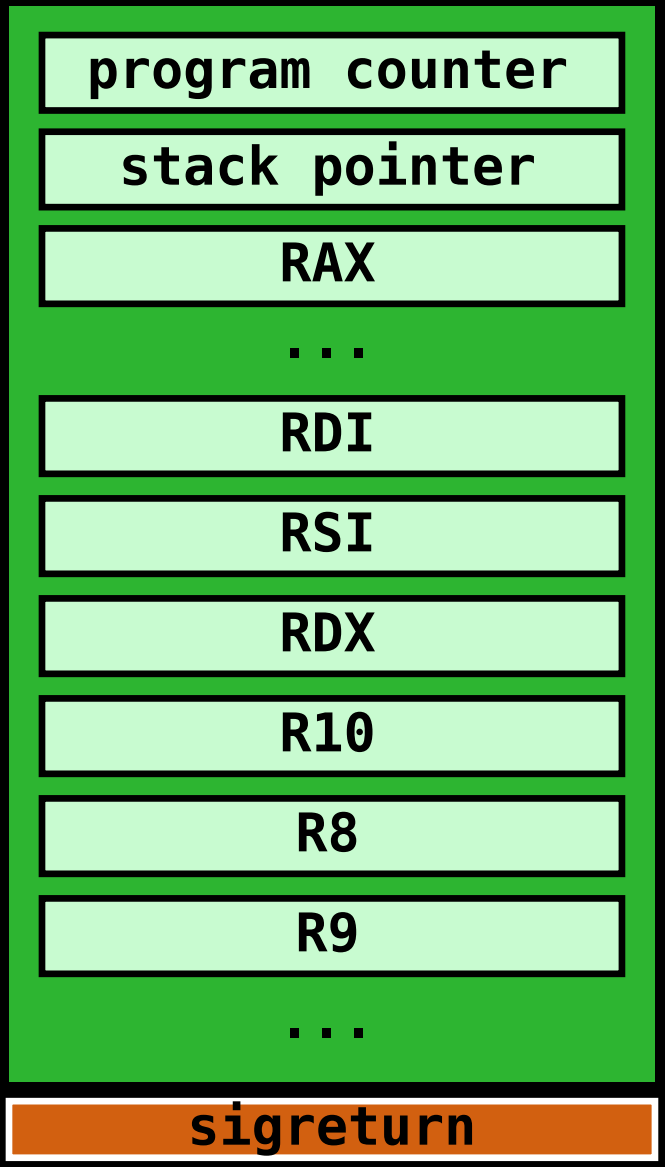
two gadgets

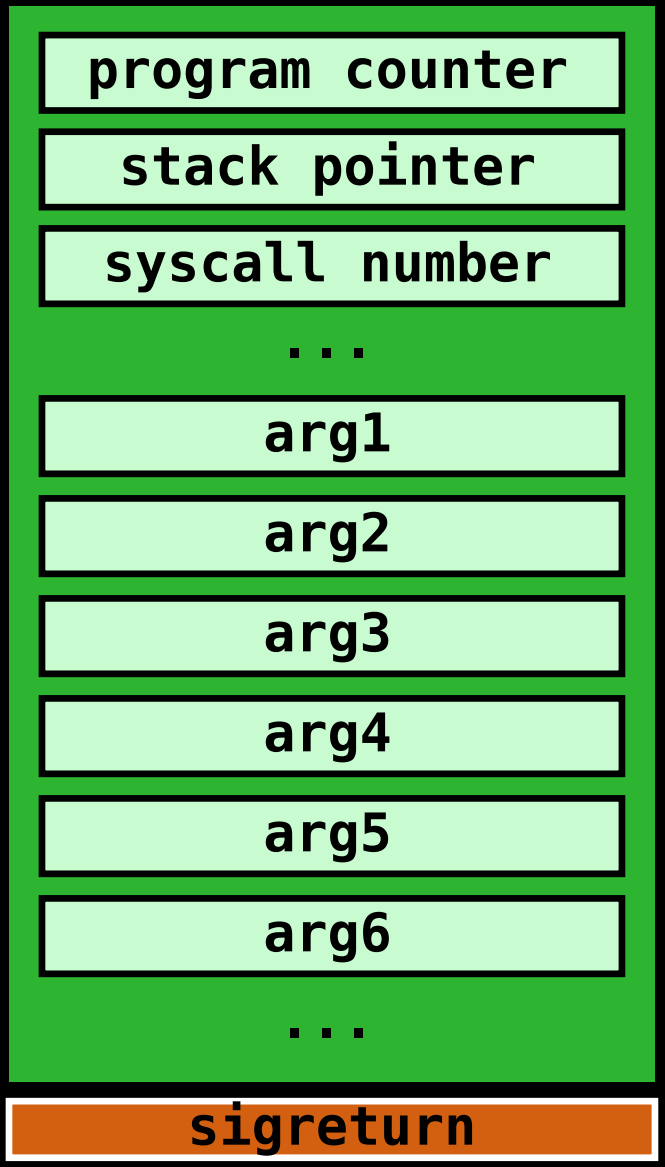
- call to sigreturn
- syscall & return

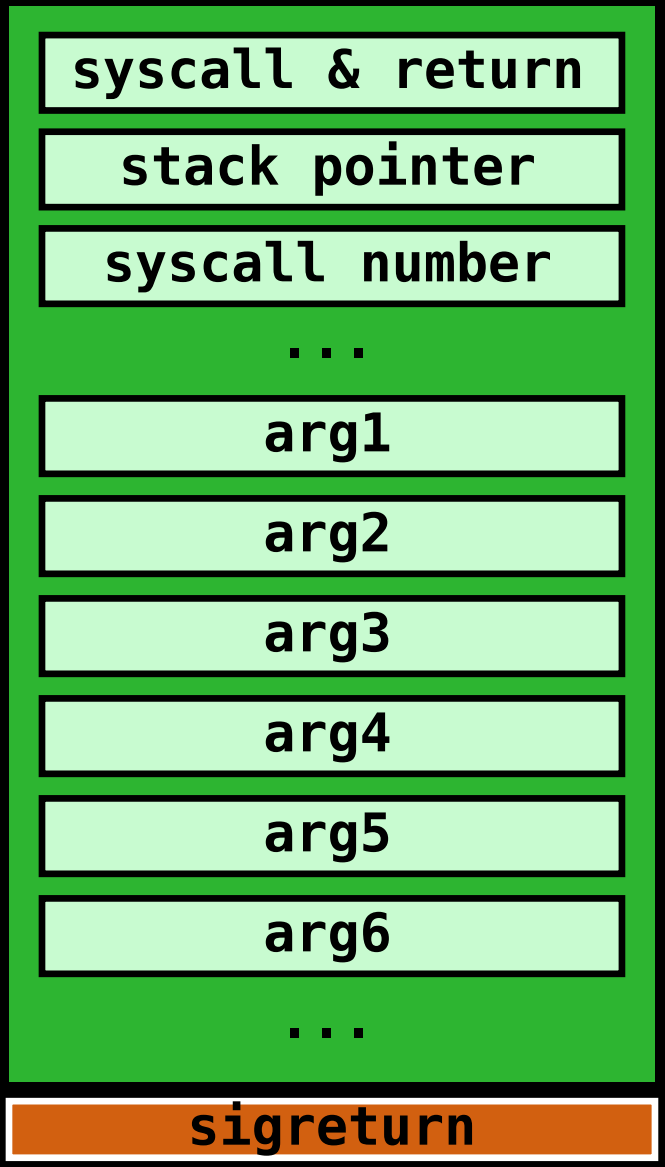

forged signal frame
sigreturn

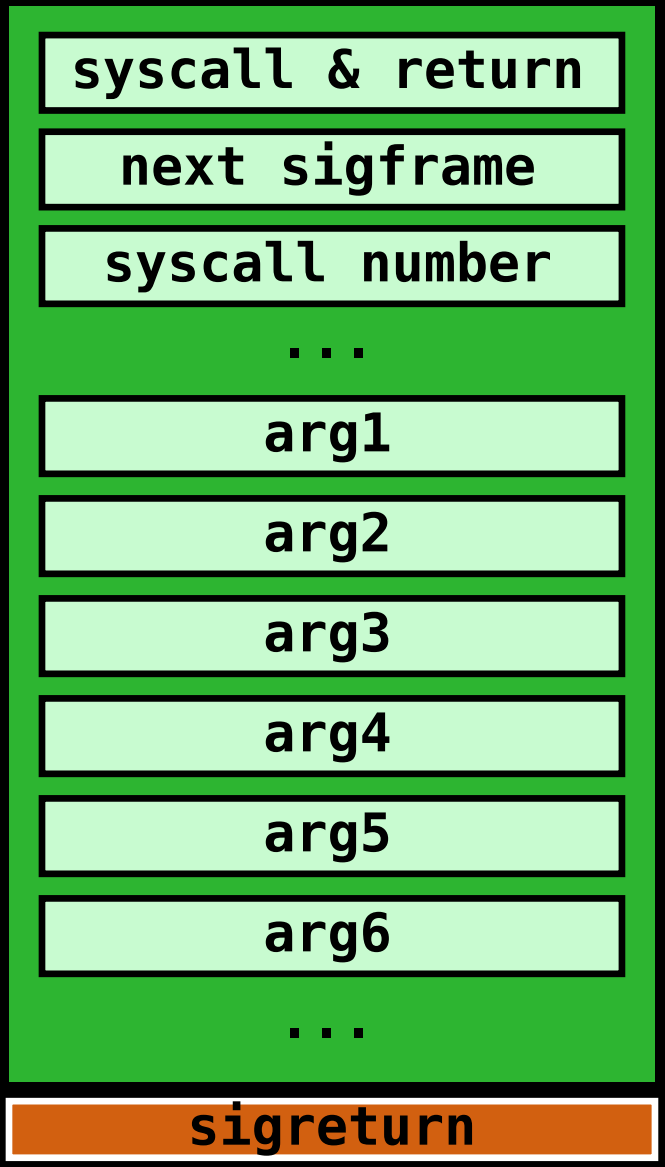

program counter
forged signal frame
sigreturn

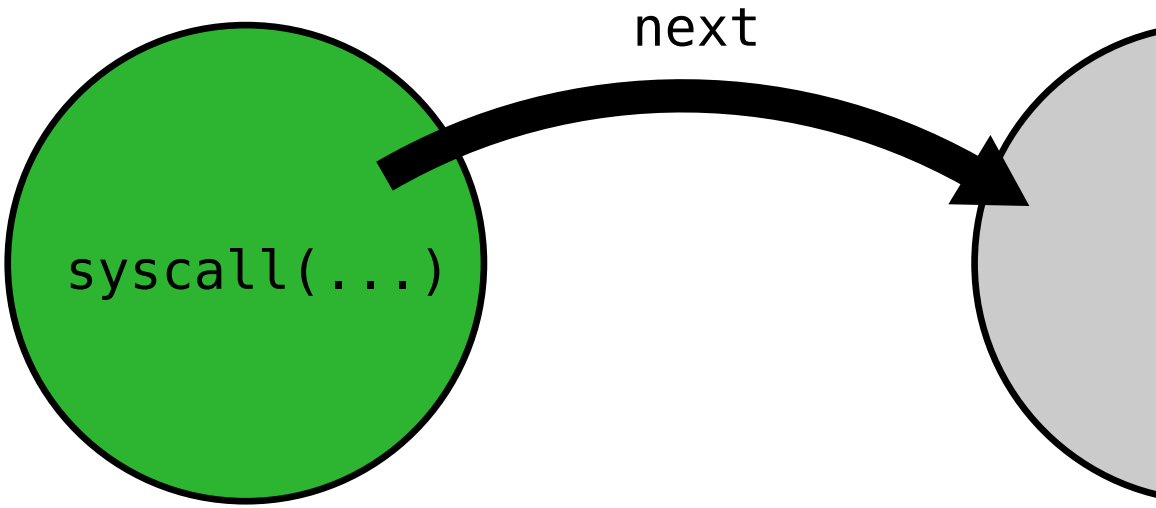

program counter
stack pointer
forged signal frame
sigreturn

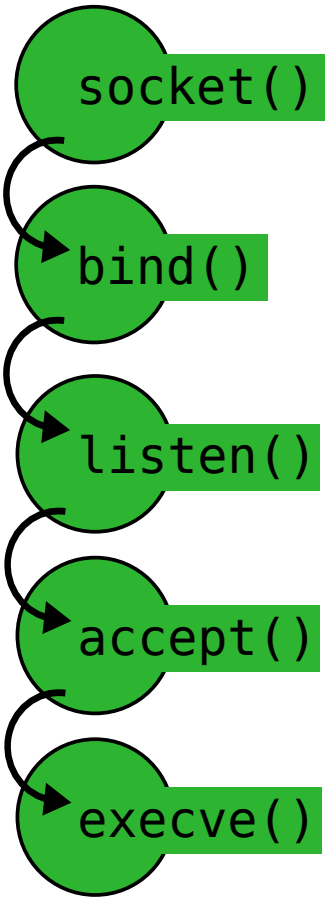

program counter
stack pointer
RAX
...
RDI
RSI
RDX
R10
R8
R9
...
sigreturn

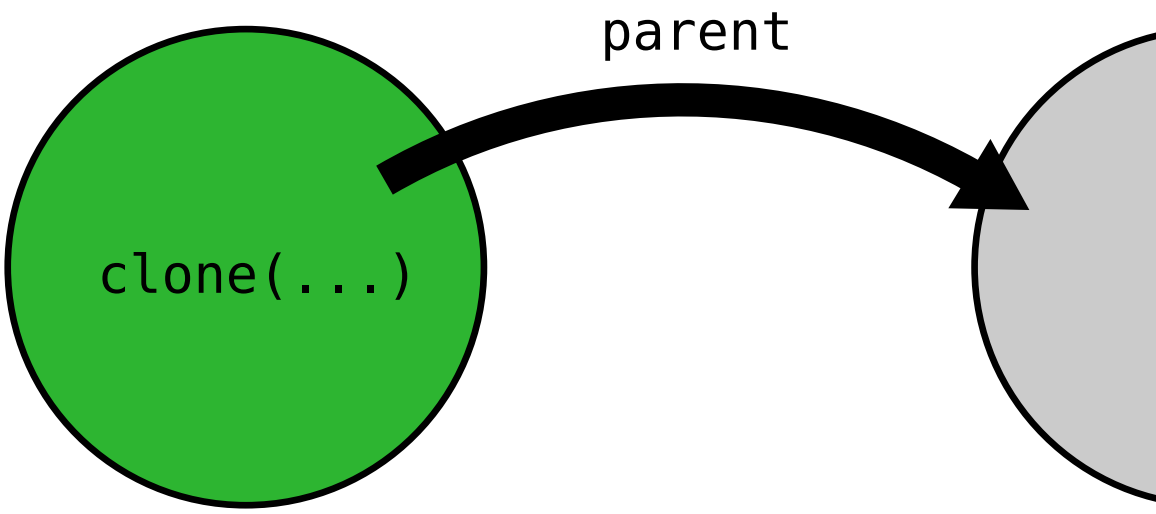

program counter
stack pointer
syscall number
...
arg1
arg2
arg3
arg4
arg5
arg6
...
sigreturn

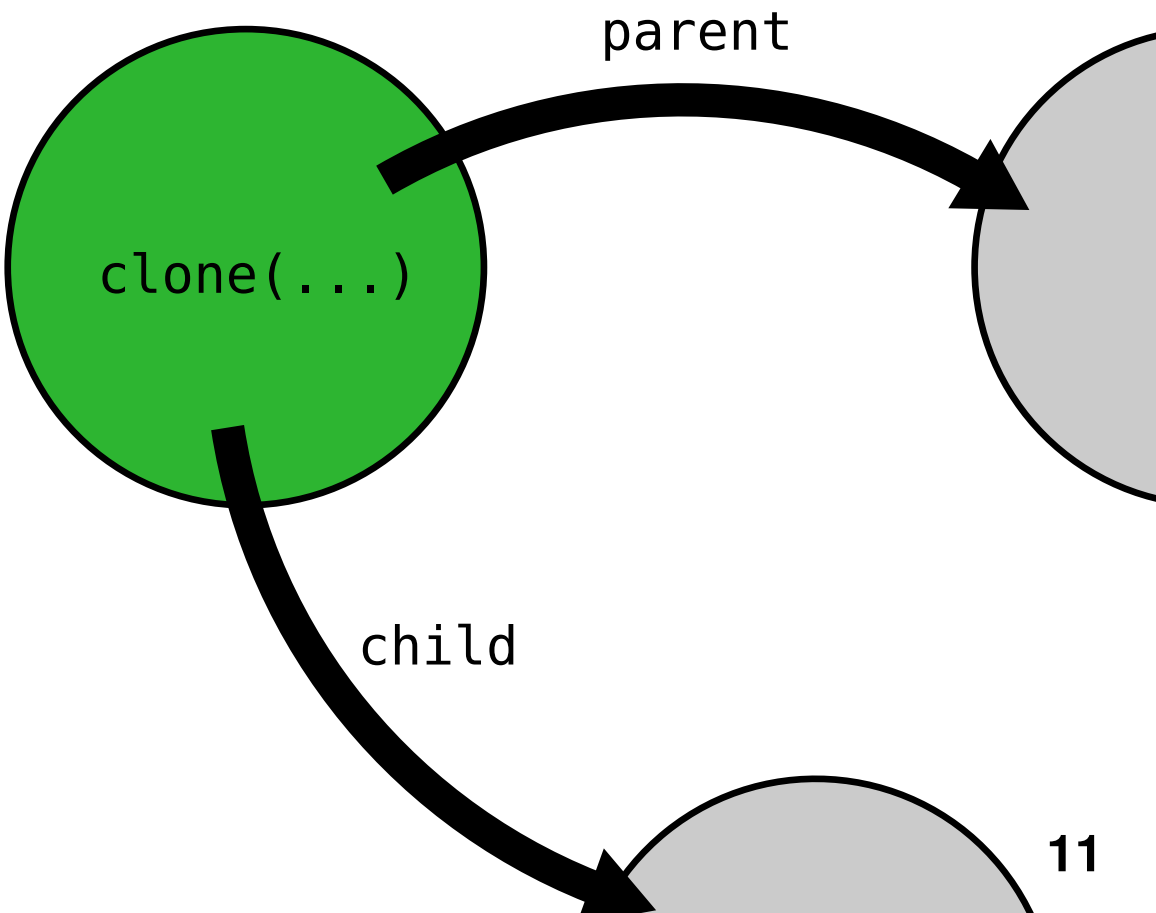

syscall & return
stack pointer
syscall number
...
arg1
arg2
arg3
arg4
arg5
arg6
...
sigreturn

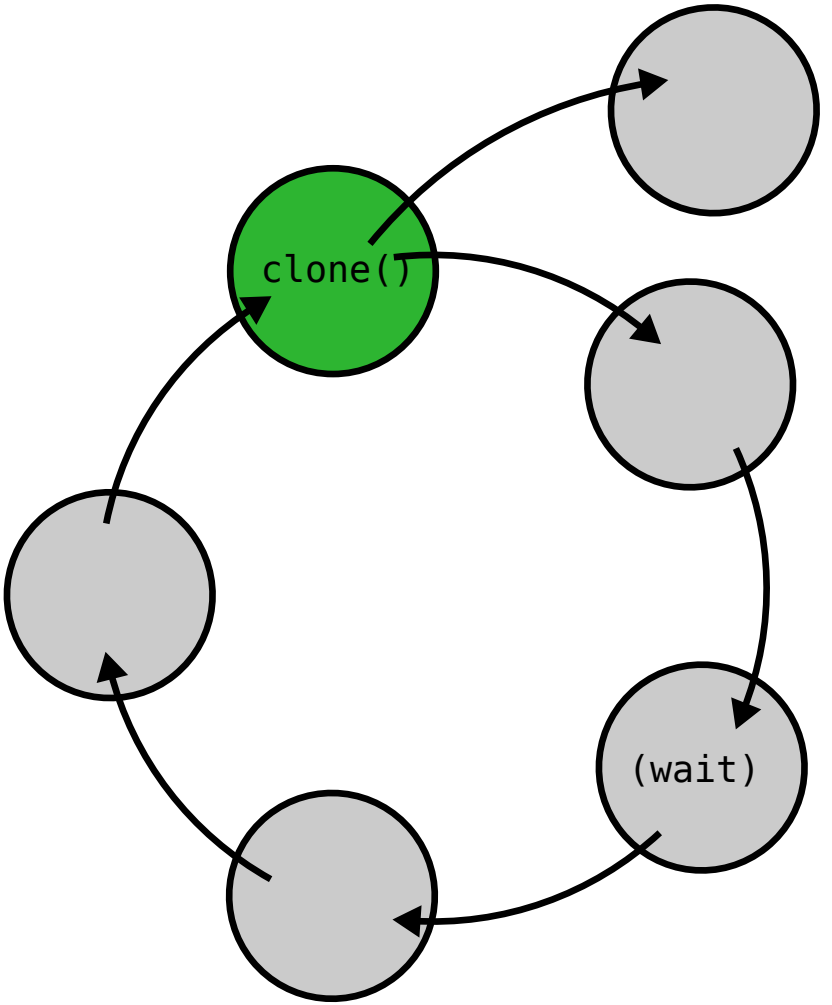

syscall & return
next sigframe
syscall number
...
arg1
arg2
arg3
arg4
arg5
arg6
...
sigreturn


next
syscall(...)


socket()
bind()
listen()
accept()
execve()


parent
clone(...)


parent
clone(...)
child


clone()
(wait)

usage scenarios

- stealthy backdoor
- code signing circumvention
- generic shellcode for exploitation

usage scenarios

- **stealthy backdoor**
- code signing circumvention
- generic shellcode for exploitation

stealthy backdoor

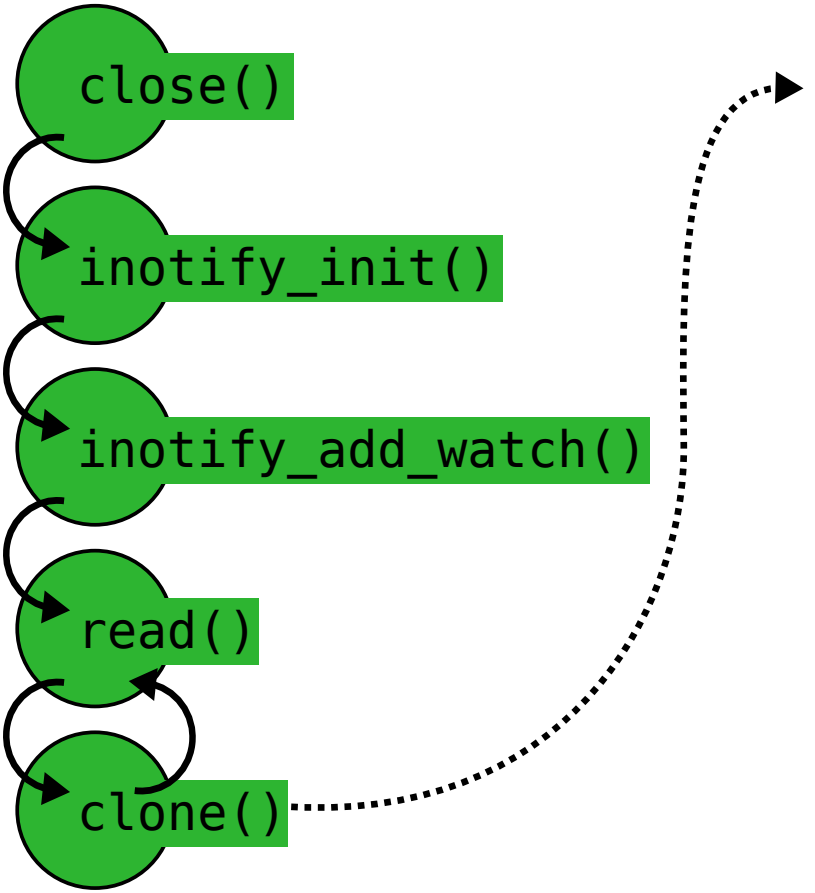
basic idea:

- use the inotify API to wait for a file to be read

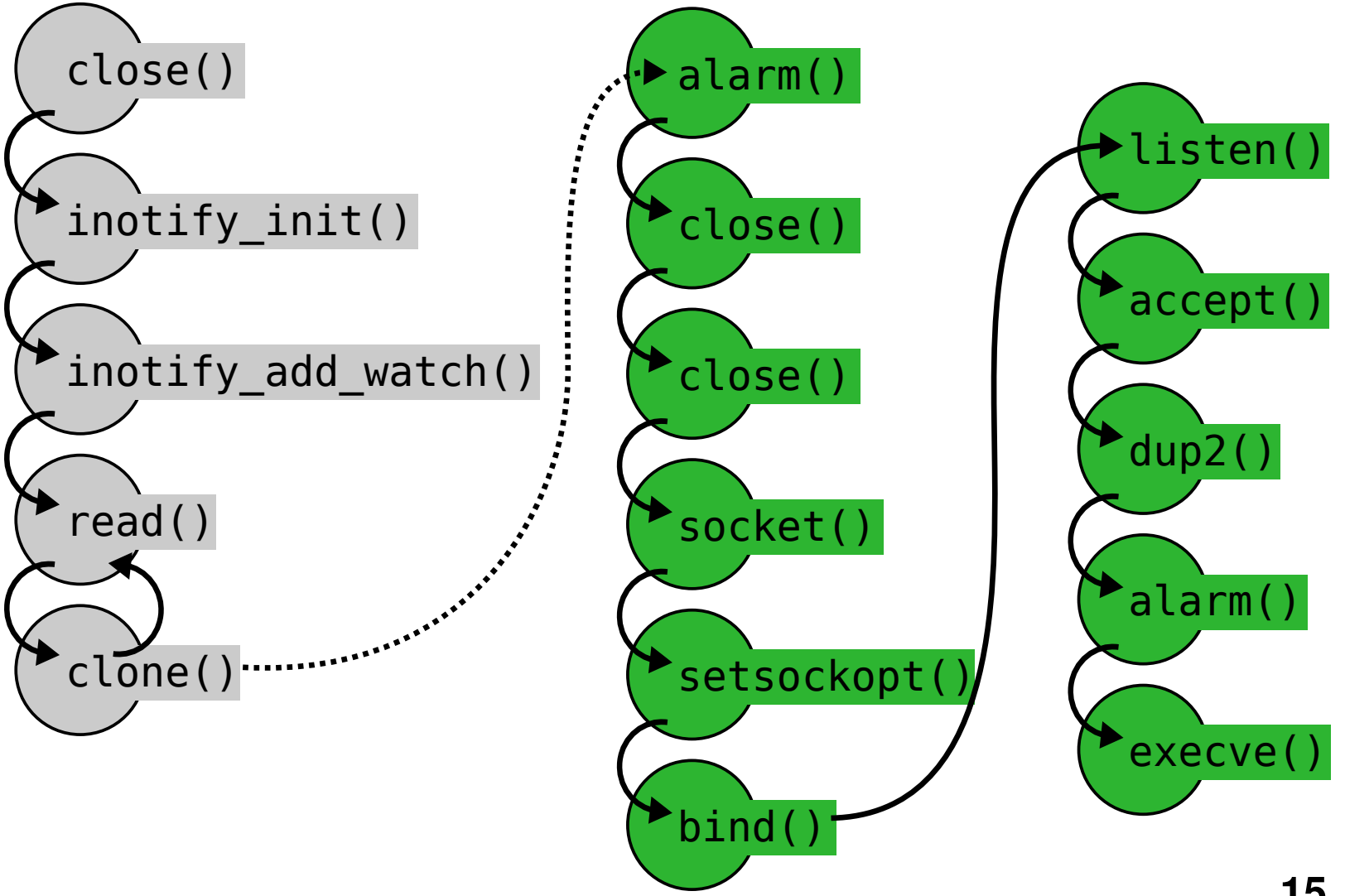
- when this file is read: open a listen socket to spawn a shell

- terminate the listening socket quickly if nobody connects

backdoor



backdoor


close()
inotify_init()
inotify_add_watch()
read()
clone()
alarm()
close()
close()
socket()
setsockopt()
bind()
listen()
accept()
dup2()
alarm()
execve()

usage scenarios

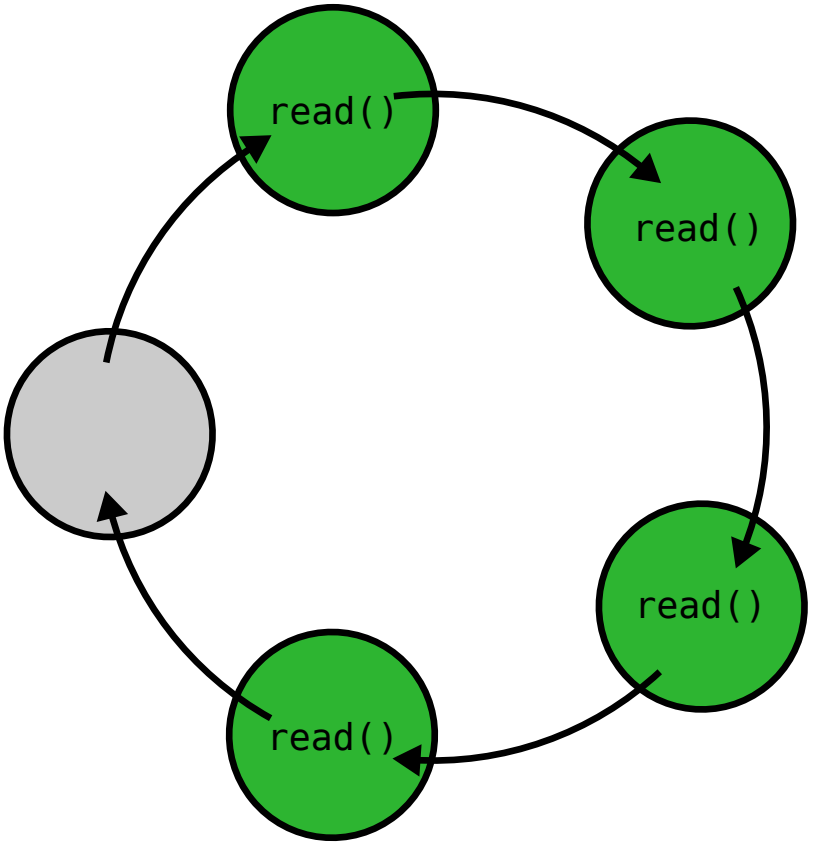
- stealthy backdoor
- **code signing circumvention**
- generic shellcode for exploitation

code signing circumvention

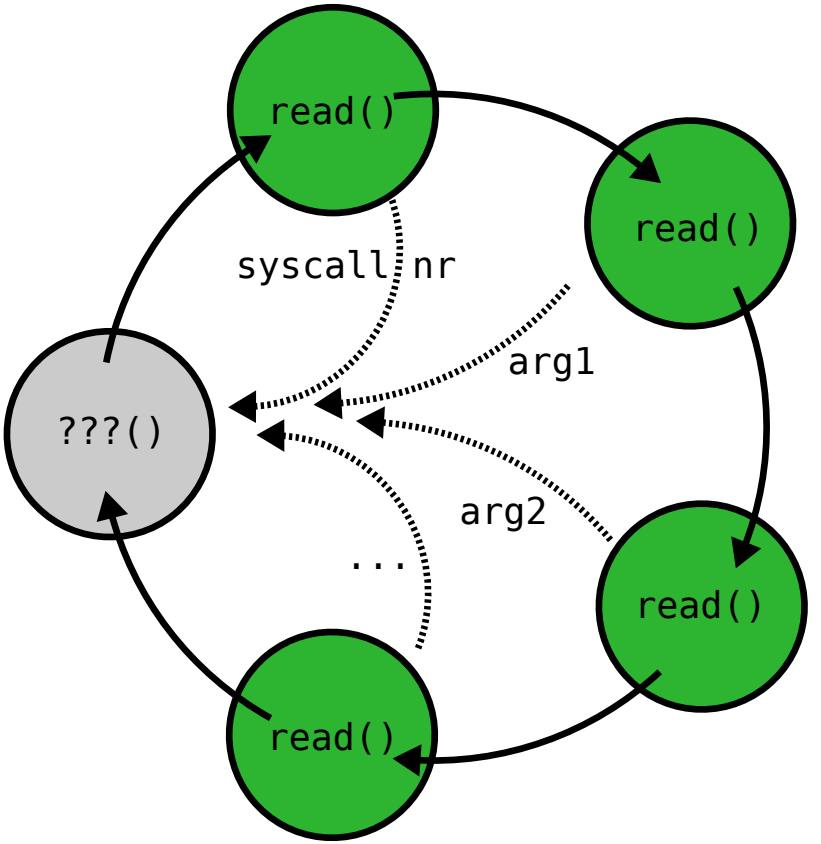
- serialize system calls over a socket
- write into our own signal frames

useful to bypass code-signing restrictions

system call proxy



system call proxy



and... It's turing complete

usage scenarios

- stealthy backdoor
- code signing circumvention
- generic shellcode for exploitation

SROP exploit on x86-64

we have:

- a stack buffer overflow
- not a single gadget from the binary

assumption:

- we can guess/leak the location of a writable address (any address!)

- we have some control over RAX (function's return value)

two gadgets

- call to sigreturn
- syscall & return

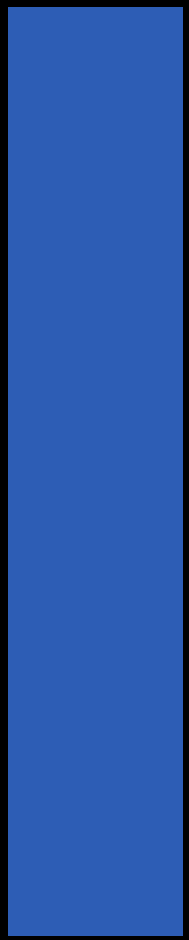
two gadgets

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

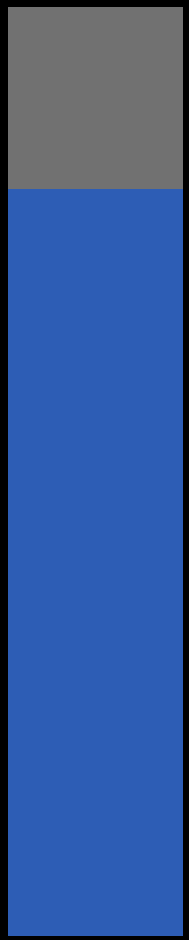
one gadget

- RAX = 15
- syscall & return

linux memory layout

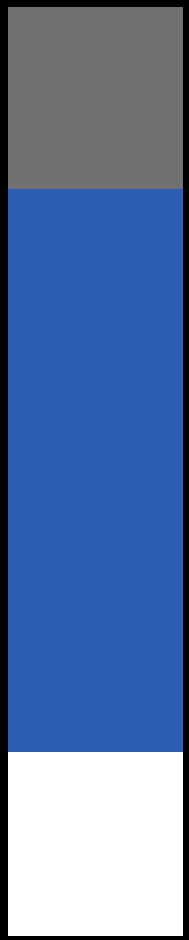


linux memory layout



kernel memory

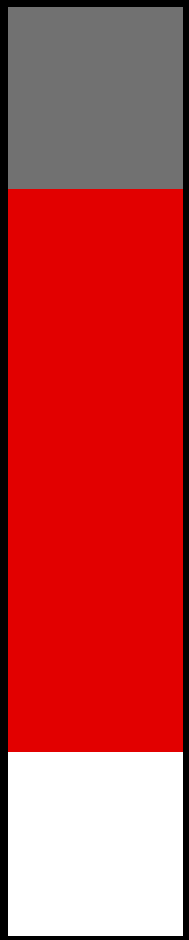
linux memory layout



kernel memory

user memory

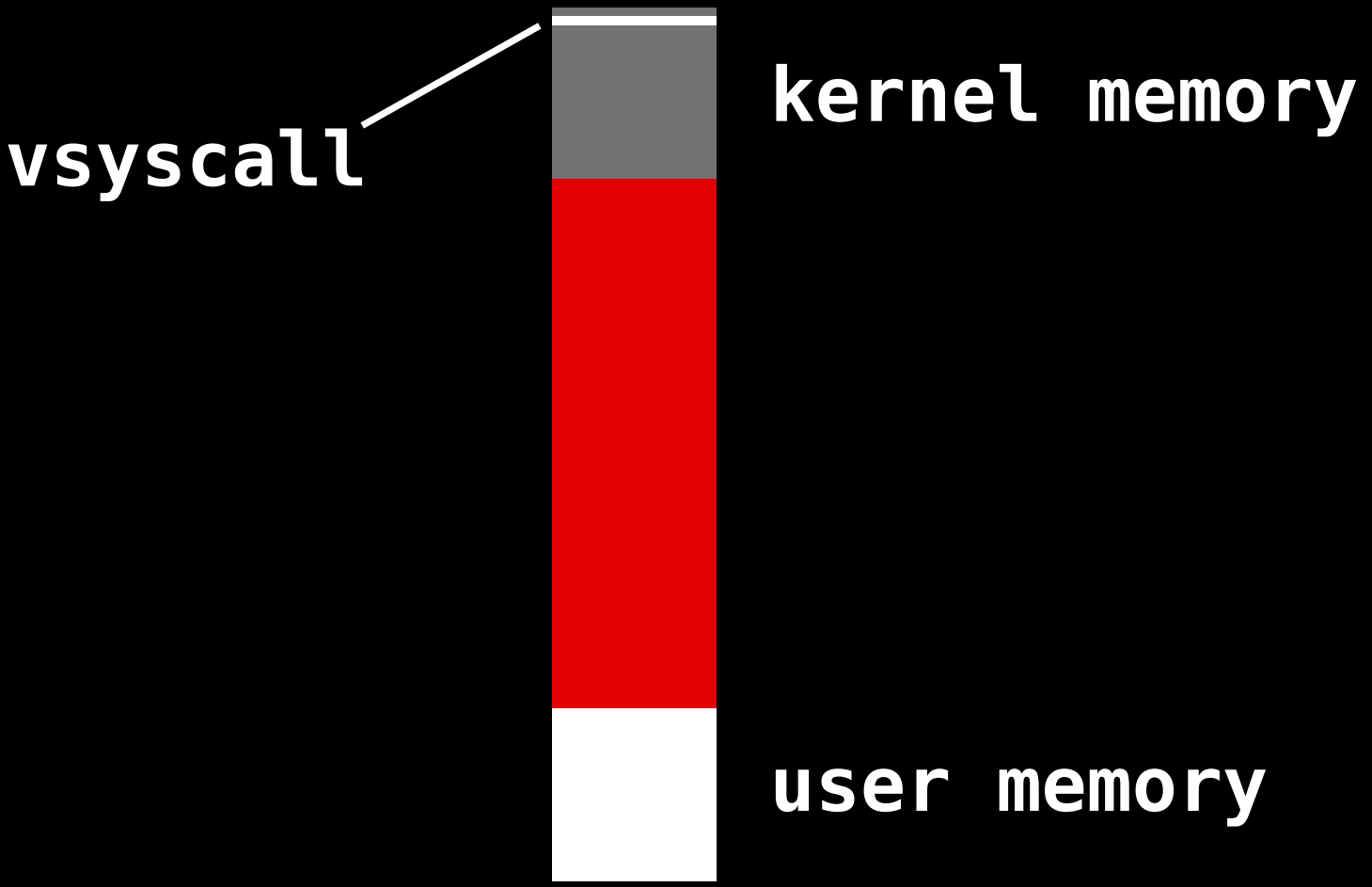
linux memory layout



kernel memory

user memory

linux memory layout



[vsyscall]

```
ffffffffff600000  48 c7 c0 60 00 00 00 0f  05 c3 cc cc cc cc cc cc gettimeofday()
ffffffffff60010  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff600400  48 c7 c0 c9 00 00 00 0f  05 c3 cc cc cc cc cc cc time()
ffffffffff600410  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff600800  48 c7 c0 35 01 00 00 0f  05 c3 cc cc cc cc cc cc getcpu()
ffffffffff600810  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff601000
```

[vsyscall]

```
ffffffffff600000  48 c7 c0 60 00 00 00 0f  05 c3 cc cc cc cc cc cc gettimeofday()
ffffffffff60010  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff600400  48 c7 c0 c9 00 00 00 0f  05 c3 cc cc cc cc cc cc time()
ffffffffff600410  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff600800  48 c7 c0 35 01 00 00 0f  05 c3 cc cc cc cc cc cc getcpu()
ffffffffff600810  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff601000
```

[vsyscall]

```
ffffffffff600000  48 c7 c0 60 00 00 00 0f  05 c3 cc cc cc cc cc cc gettimeofday()
ffffffffff60010  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff600400  48 c7 c0 c9 00 00 00 0f  05 c3 cc cc cc cc cc cc time()
ffffffffff600410  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff600800  48 c7 c0 35 01 00 00 0f  05 c3 cc cc cc cc cc cc getcpu()
ffffffffff600810  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc
*
ffffffffff601000
```

0f05 syscall
c3 return

```
syscall(arg1, arg2, arg3, ...) = result
```

```
execve("/bin/sh",
       ["/bin/sh", "-c", "...", NULL],
       NULL)
```

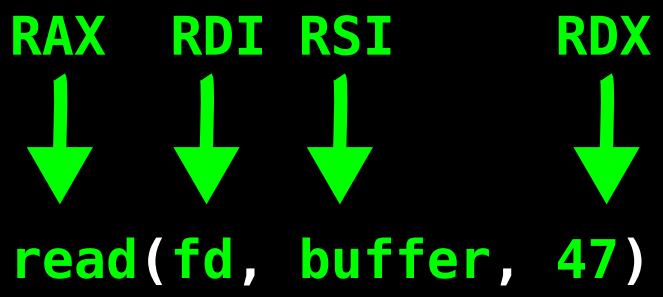
```
execve("/bin/sh",
       ["/bin/sh", "-c", "...", NULL],
       NULL)
```

```
syscall(arg1, arg2, arg3, ...) = result
```

x64 syscall ABI trick

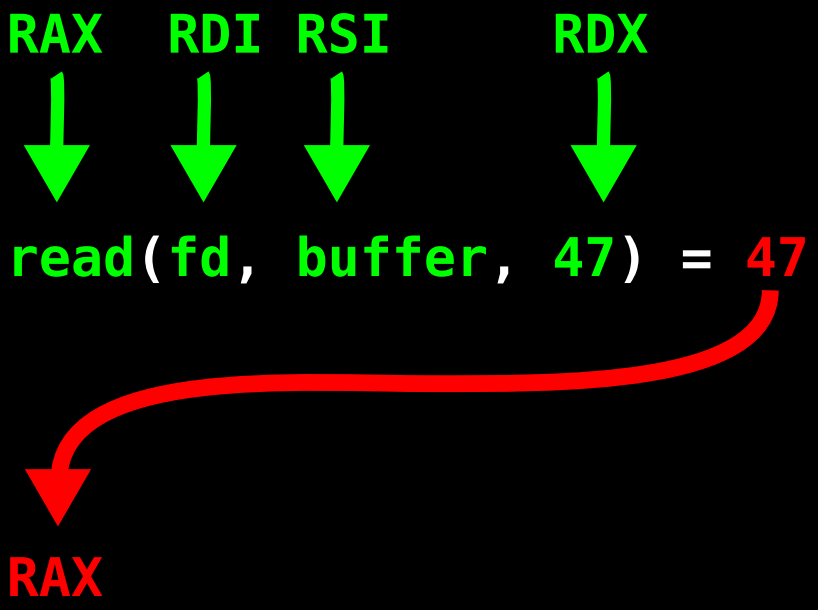
```
read(fd, buffer, 47)
```

x64 syscall ABI trick

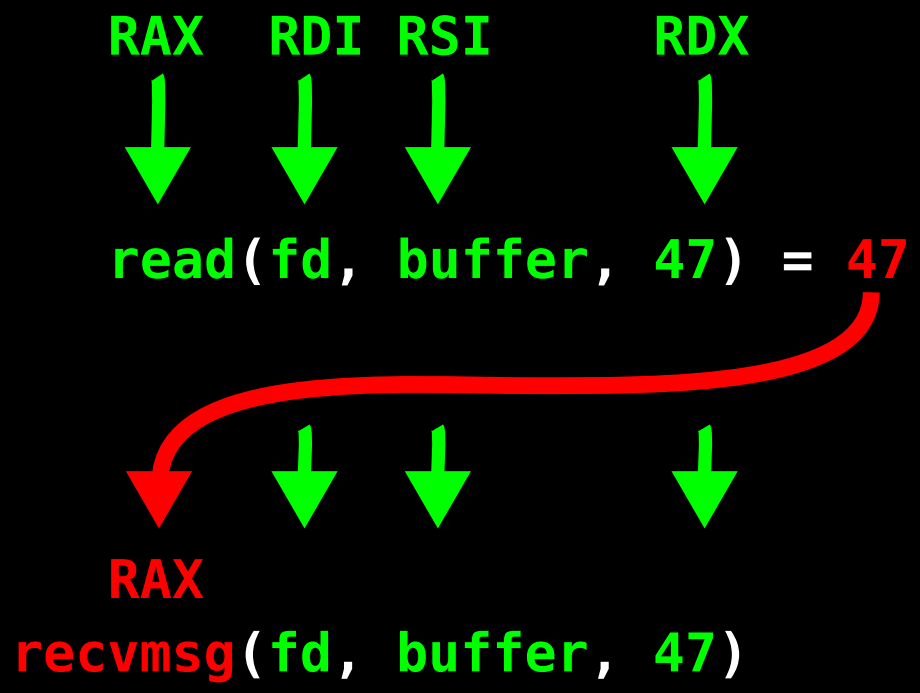


x64 syscall ABI trick

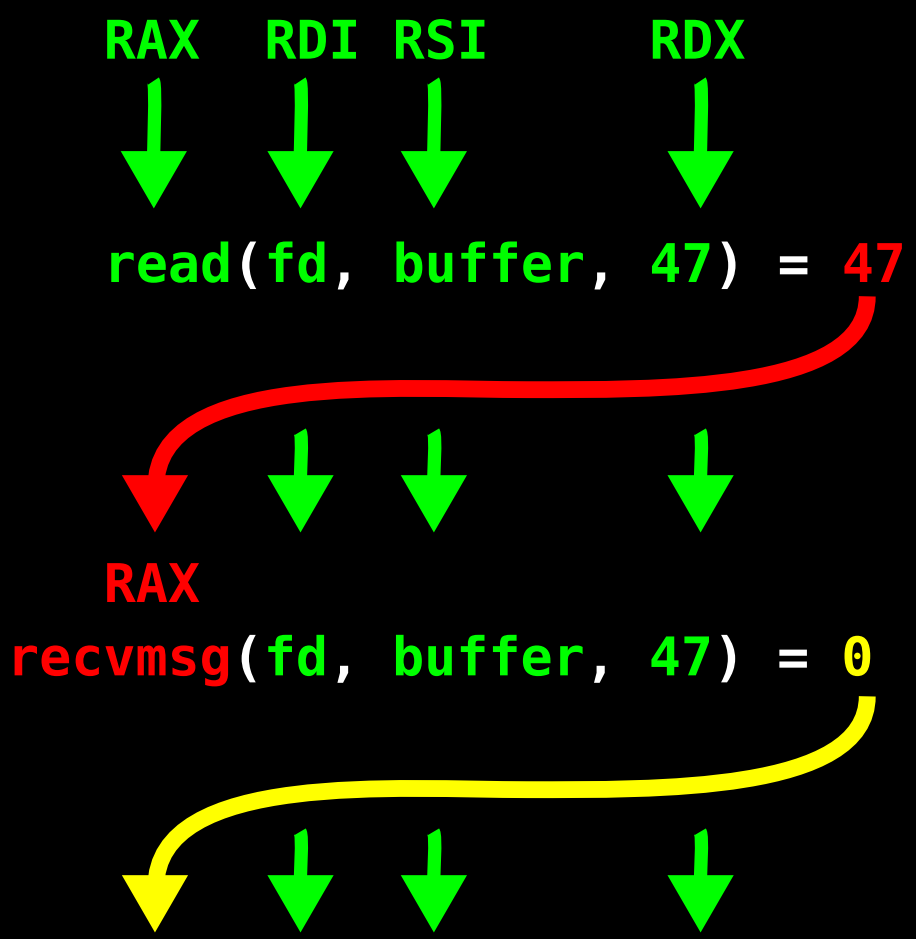
x64 syscall ABI trick



x64 syscall ABI trick



x64 syscall ABI trick



```
syscall(arg1, arg2, arg3, ...) = result
```

```
read(fd, addr, ...) = result
```

```
read(fd, stack_addr, ...) = result
```

```
read(fd, stack_addr, 306) = 306
```

```
read(fd, stack_addr, 306) = 306
     RAX == 306 == __NR_syncfs
```

```
read(fd, stack_addr, 306) = 306
```

RAX == 306 == __NR_syncfs

top of stack points to syscall & return

```
read(fd, stack_addr, 306) = 306
     RAX == 306 == __NR_syncfs
     top of stack points to syscall & return
syncfs(fd) = ...
```

```
read(fd, stack_addr, 306) = 306
     RAX == 306 == __NR_syncfs
     top of stack points to syscall & return
syncfs(fd) = 0
```

```
read(fd, stack_addr, 306) = 306

     RAX == 306 == __NR_syncfs
     top of stack points to syscall & return

syncfs(fd) = 0

     RAX == 0 == __NR_read
     top of stack points to syscall & return
```

```
read(fd, stack_addr, 306) = 306

    RAX == 306 == __NR_syncfs
    top of stack points to syscall & return

syncfs(fd) = 0

    RAX == 0 == __NR_read
    top of stack points to syscall & return

read(fd, stack_addr, 306) = ...
```

```
read(fd, stack_addr, 306) = 306

    RAX == 306 == __NR_syncfs
    top of stack points to syscall & return

syncfs(fd) = 0

    RAX == 0 == __NR_read
    top of stack points to syscall & return

read(fd, stack_addr, 306) = 15
```

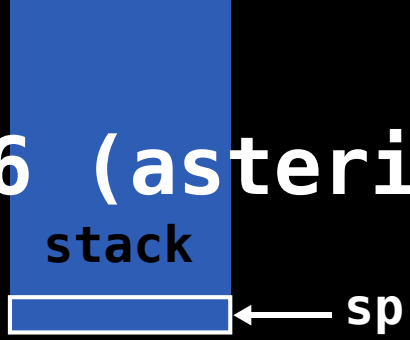
```
read(fd, stack_addr, 306) = 306

    RAX == 306 == __NR_syncfs
    top of stack points to syscall & return

syncfs(fd) = 0

    RAX == 0 == __NR_read
    top of stack points to syscall & return

read(fd, stack_addr, 306) = 15

    RAX == 15 == __NR_rt_sigreturn
    top of stack points to syscall & return
```

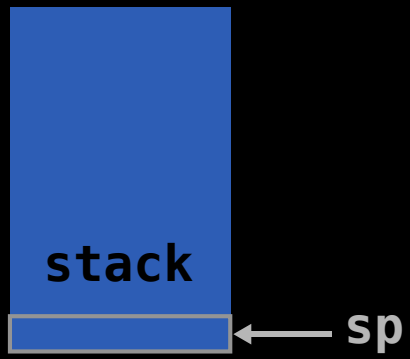
```
read(fd, stack_addr, 306) = 306

    RAX == 306 == __NR_syncfs
    top of stack points to syscall & return

syncfs(fd) = 0

    RAX == 0 == __NR_read
    top of stack points to syscall & return

read(fd, stack_addr, 306) = 15

    RAX == 15 == __NR_rt_sigreturn
    top of stack points to syscall & return

mprotect(stack_addr, 0x1000,
         PROT_READ|PROT_WRITE|PROT_EXEC)
```

```
read(fd, stack_addr, 306) = 306

    RAX == 306 == __NR_syncfs
    top of stack points to syscall & return

syncfs(fd) = 0

    RAX == 0 == __NR_read
    top of stack points to syscall & return

read(fd, stack_addr, 306) = 15

    RAX == 15 == __NR_rt_sigreturn
    top of stack points to syscall & return

mprotect(stack_addr, 0x1000,
         PROT_READ|PROT_WRITE|PROT_EXEC)

    top of stack points to our code
```

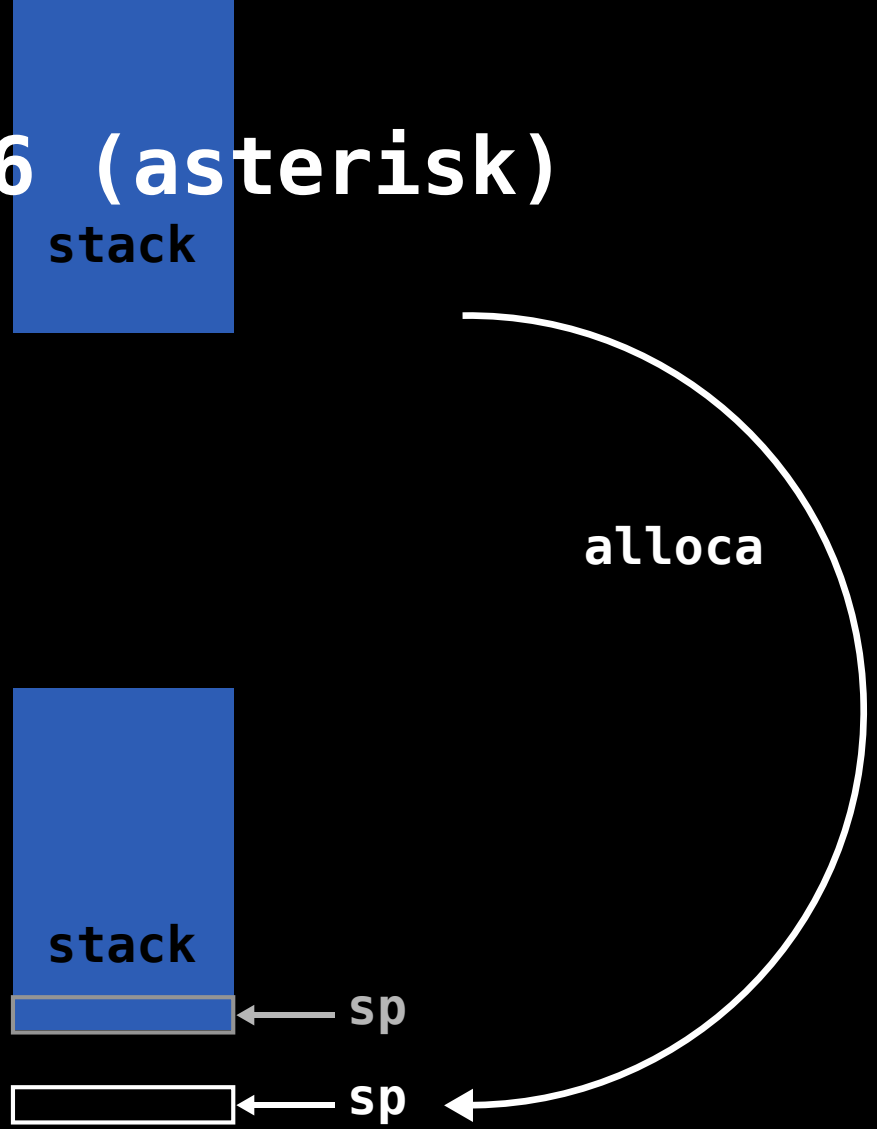
CVE-2012-5976 (asterisk)

CVE-2012-5976 (asterisk)

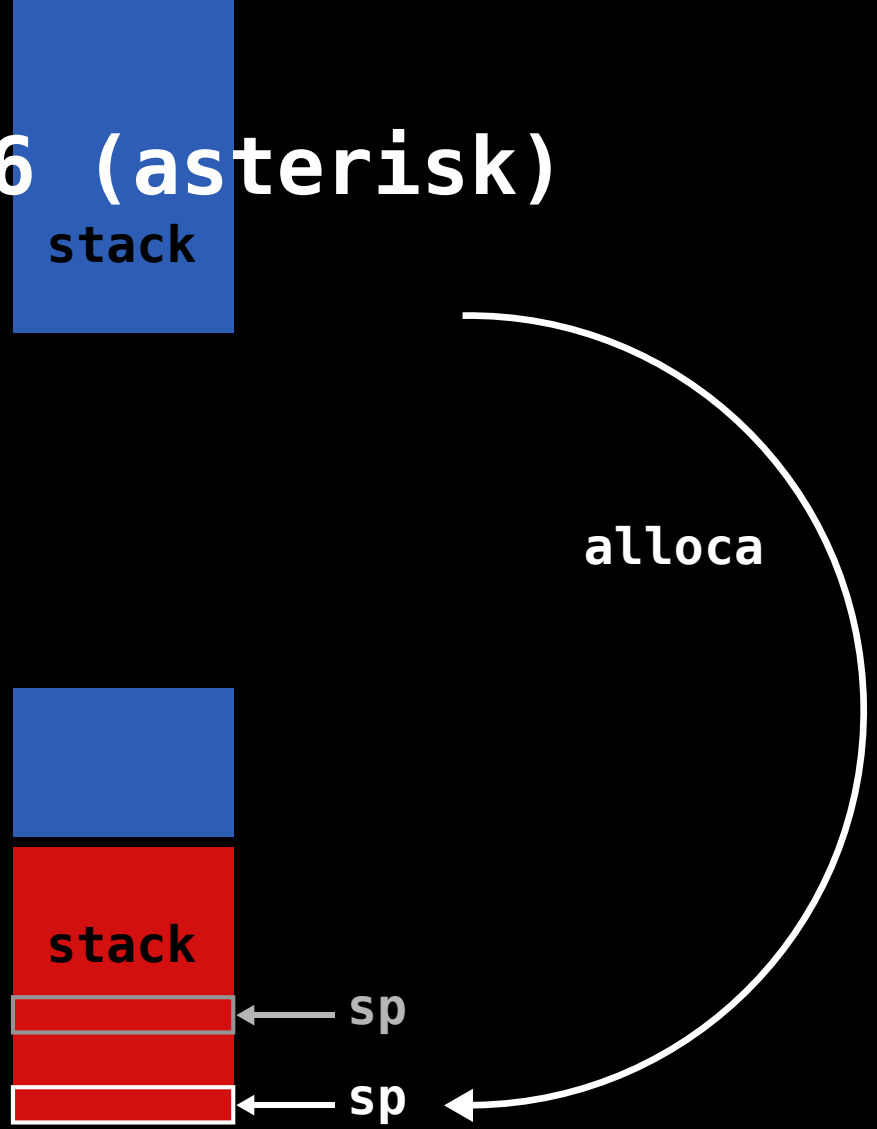




CVE-2012-5976 (asterisk)



CVE-2012-5976 (asterisk)



On some systems SROP 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] 0xffffffffff600000
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 SROP gadgets are randomised, on others, they are not

android

non-ASLR :-(

Operating system	Gadget	Memory map
Linux i386	sigreturn	[vdso]
Linux < 3.11 ARM	sigreturn	[vectors] 0xffff0000
Linux < 3.3 x86-64	syscall & return	[vsyscall] 0xffffffffff600000
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

mitigation:

It may be useful to disable vsyscall

vsyscall=emulate
(default from Linux 3.3 onward)

or

vsyscall=none

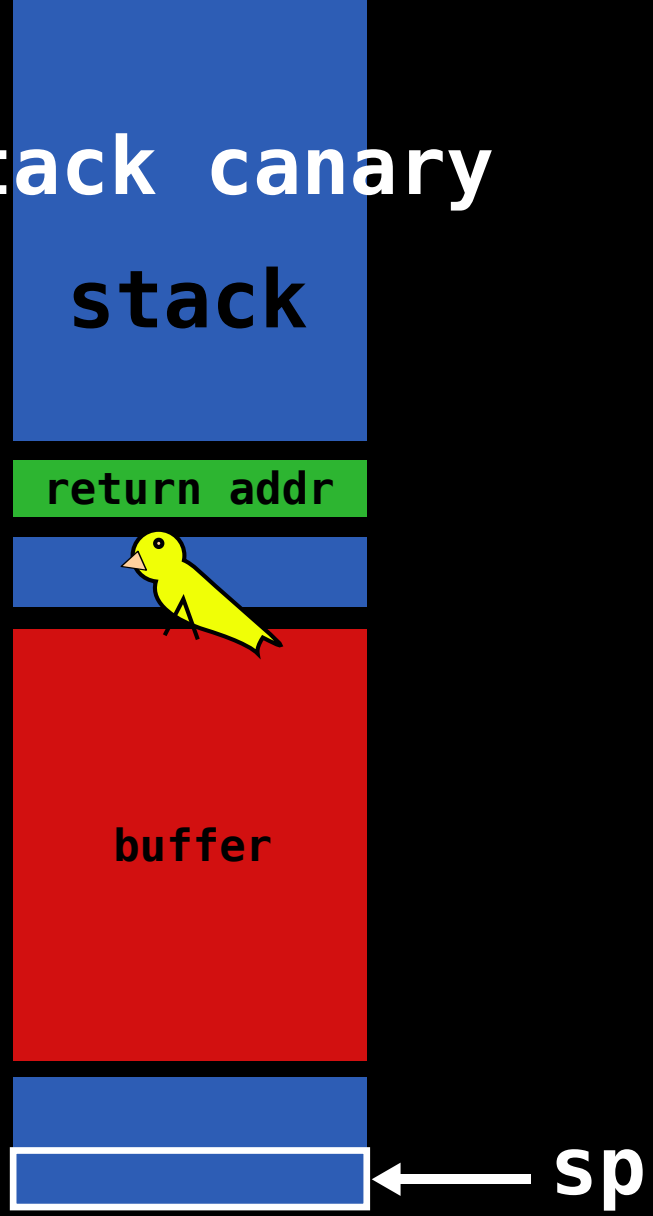
mitigation:

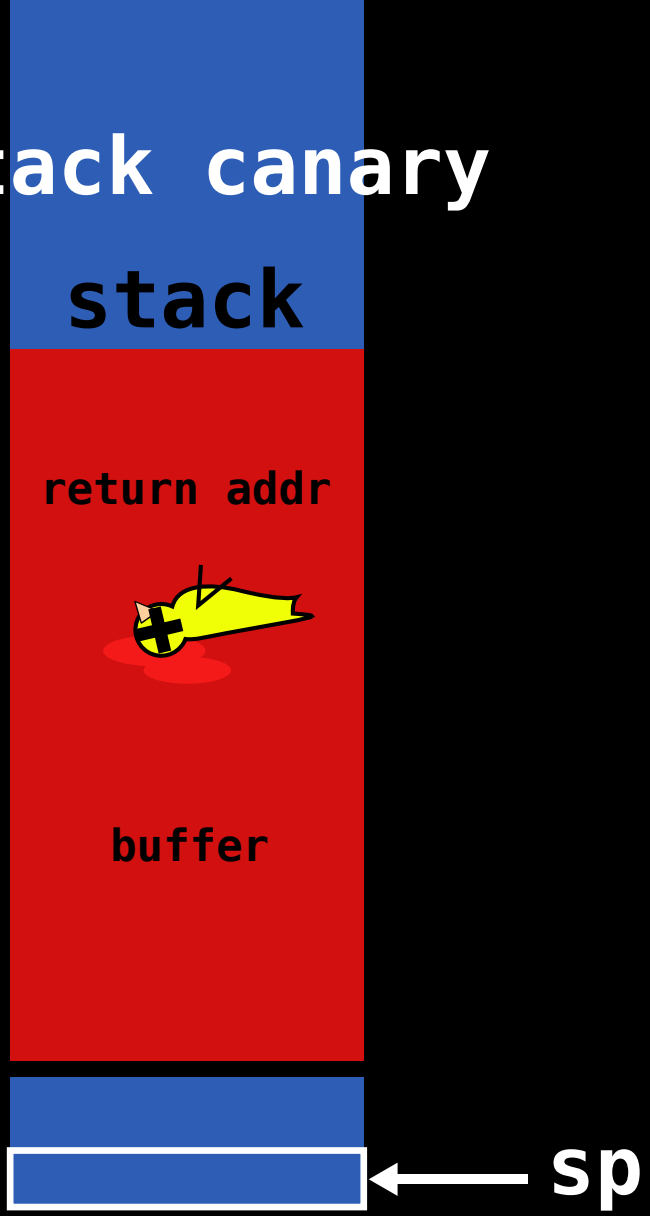
- Counting signals in progress

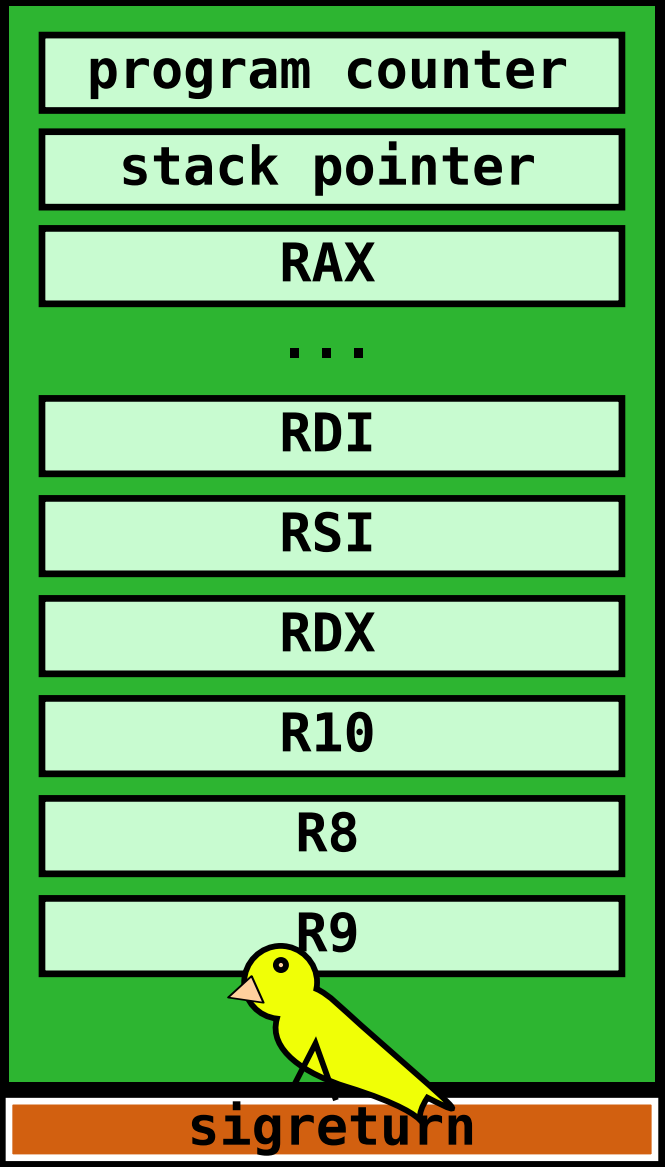
mitigation:

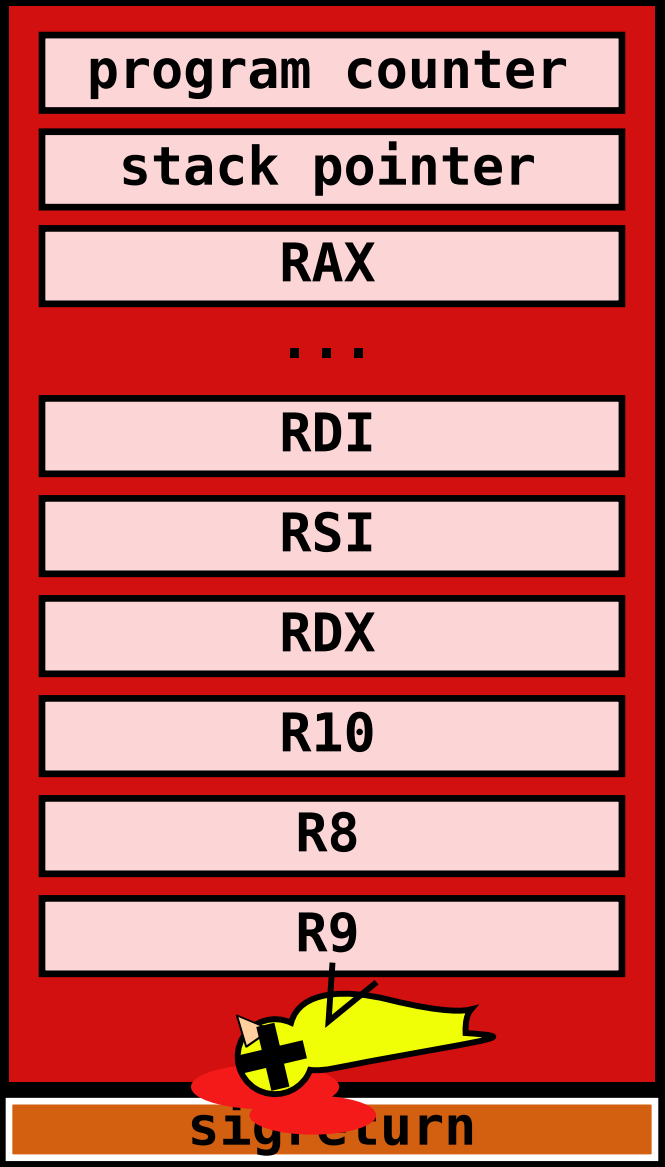
- Counting signals in progress

- Signal frame canaries

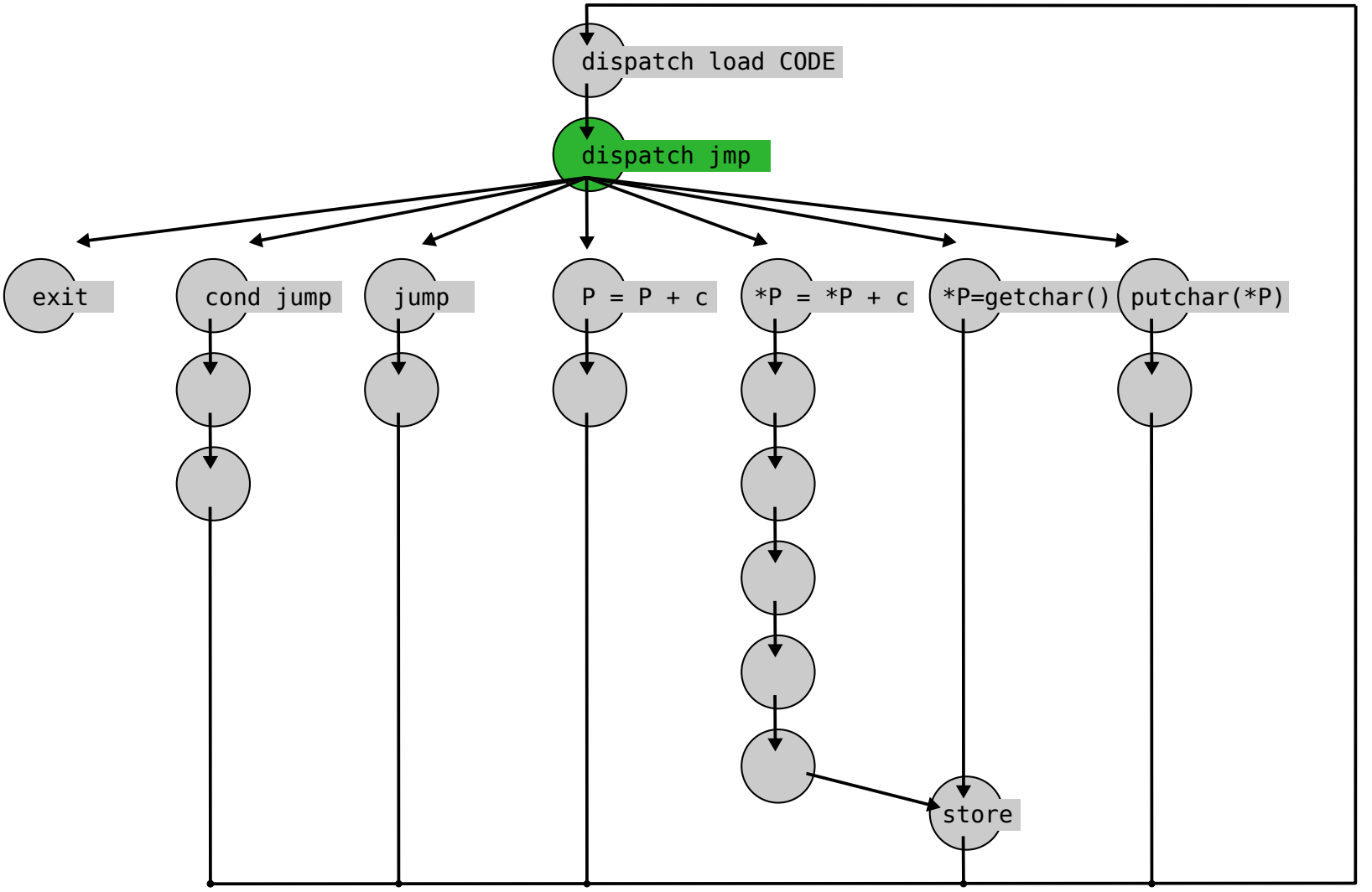

stack canary
stack
return addr
buffer
sp


stack canary
stack
return addr
buffer
sp


program counter
stack pointer
RAX
...
RDI
RSI
RDX
R10
R8
R9
sigreturn


program counter
stack pointer
RAX
...
RDI
RSI
RDX
R10
R8
R9
sigreturn

questions?


dispatch load CODE
dispatch jmp
exit
cond jump
jump
P = P + c
*P = *P + c
*P=getchar()
putchar(*P)
store

```
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);
```