# How to Fuzz Your Way to Android Universal Root:

## Attacking Android Binder

Eugene Rodionov, Zi Fan Tan, Gulshan Singh

10-May-2024


Google

# Agenda

- Introduction in Android Binder
- CVE-2023-20938 & CVE-2023-21255 UAF Details
- Exploitation of CVE-2023-20938
- Fuzzing Binder with LKL
- Conclusion

# Introduction

# Who we are

Increase **Android** security by attacking key components and features, identifying critical vulnerabilities before adversaries

- Offensive Security Reviews to verify (break) security assumptions
- Scale through tool development (e.g. continuous fuzzing)
- Develop proof of concepts to demonstrate real-world impact
- Assess the efficacy of security mitigations

# Binder Overview

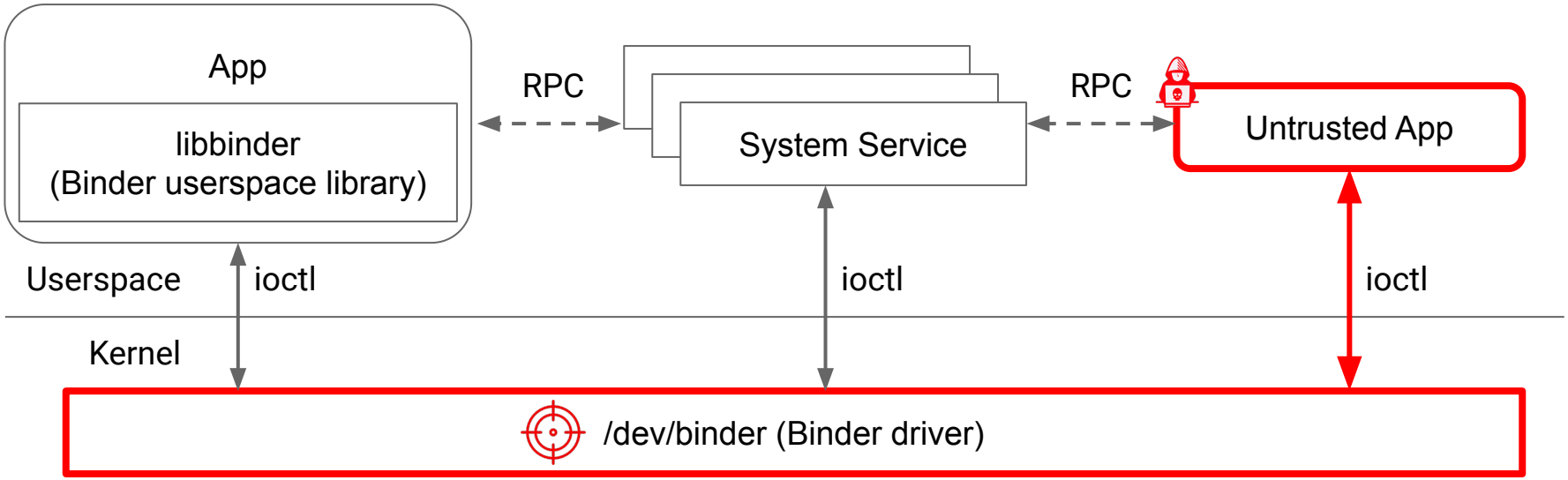
# What is Binder?

- Primary inter-process communication (IPC) channel on Android
- Support passing file descriptors, objects containing pointers, etc.
- Composed of a userspace library (libbinder) and a kernel driver (/dev/binder)
- Provide Remote Procedure Call (RPC) framework for Java and C/C++

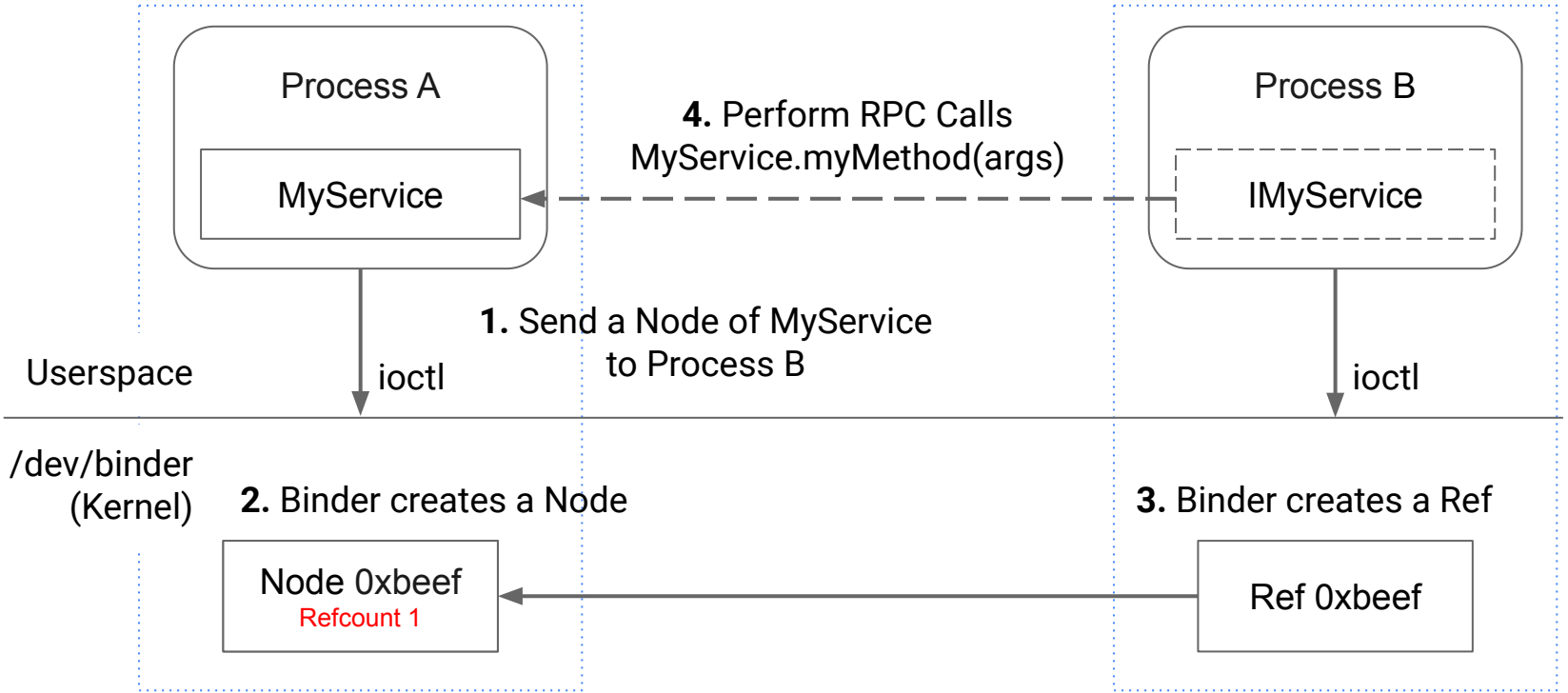
# Why Binder?

- High vulnerabilities per thousand lines of code
  - ~7k lines of C code (3 vulns / 1K lines)
- Wide attack surface
  - Accessible by all untrusted apps
  - Historically, exploited from Chrome for sandbox escapes
- Recently exploited for root privilege escalation
  - Waterdrop (2019), Bad Binder (2019), CVE-2020-0041 (2020), Typhoon Mangkhut (2020), Bad Spin (2022)
- Complex object lifetime, memory management model, and highly complex multithreading model
  - 5 different locks, 6 reference counters, as well as atomic variables
  - Our initial assumption was data races would be the primary cause of vulnerabilities, but refcount bugs were more prevalent

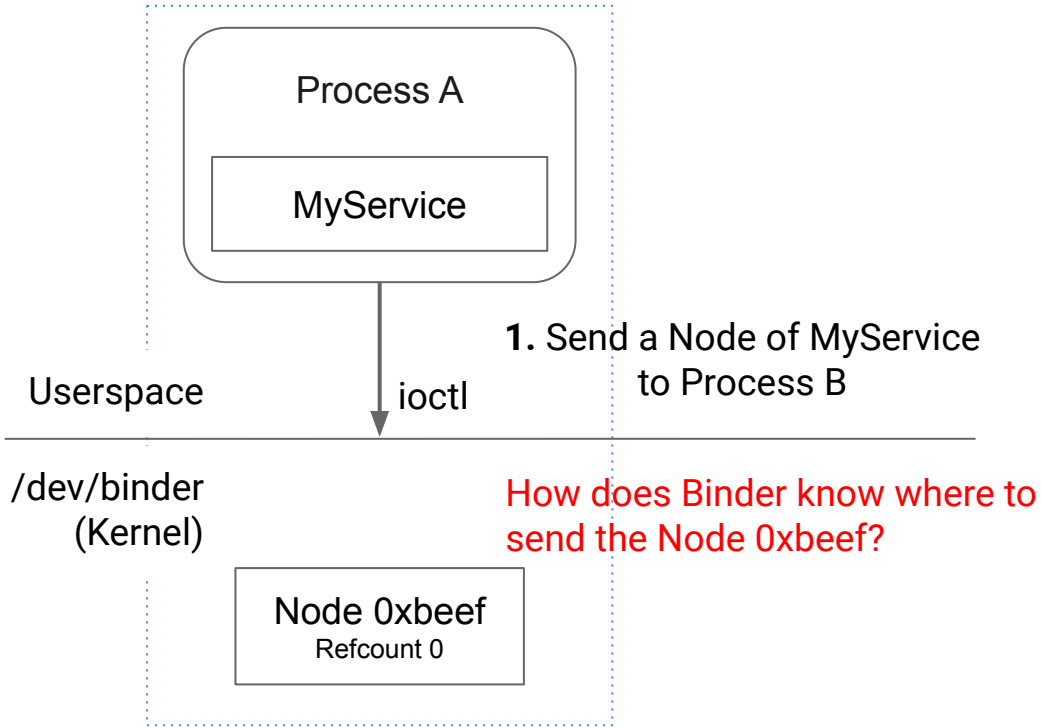
# Binder Threat Model



# Binding IPC Endpoints: Workflow



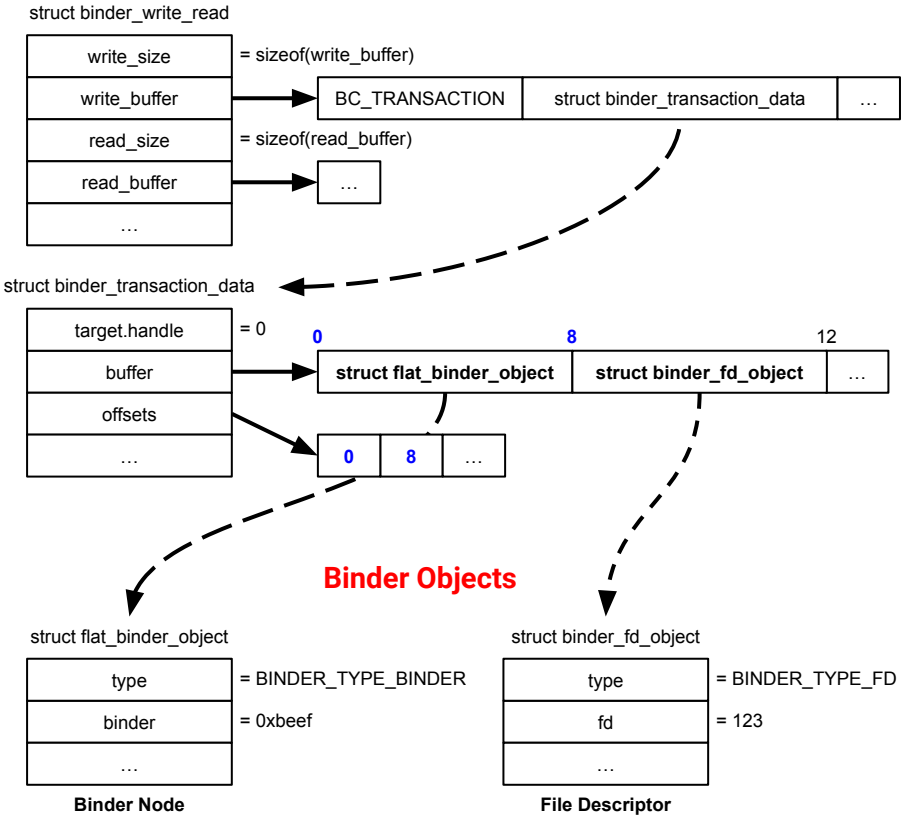
# Binder Context Manager




- **Context Manager** is a special Binder IPC endpoint always accessible at handle 0
- In Android, **ServiceManager** process serves as Binder **Context Manager**
- Android components register their Binder Nodes with the **ServiceManager** to be discoverable by other endpoints

# Binder Transactions

- Accessible via BINDER_WRITE_READ ioctl
- Transfer **Binder objects** between the IPC endpoints:
  - Binder Node
  - Binder Ref
  - Linux file descriptor
  - Binder buffer pointers
- **Binder objects** are “translated” from the sender’s context into the recipient context



# Vulnerabilities

## CVE-2023-20938 & CVE-2023-21255

# Vulnerability Description (1)

- When sending a transaction, Binder translates objects to another form
  - Translate a Binder Node to a Binder Ref or vice versa
  - Install new file descriptors in another process for file sharing

```
binder_size_t buffer_offset = 0;



for (buffer_offset = off_start_offset; buffer_offset < off_end_offset;
    buffer_offset += sizeof(binder_size_t)) {
  // ...
}
```

# Vulnerability Description (2)

- If an error occurs in the loop, we need to clean all objects translated so far
- Cleanup function is passed the offset in the buffer it's reached and `is_failure` set to true

```
binder_size_t buffer_offset = 0;



for (buffer_offset = off_start_offset; buffer_offset < off_end_offset;
    buffer_offset += sizeof(binder_size_t)) {
  // if error: goto err_bad_offset
}
err_bad_offset:
  binder_transaction_buffer_release(target_proc, NULL, t->buffer,
                                    buffer_offset, /*is_failure*/true);
```

# Vulnerability Description (3)

- What happens if an error happens before any objects are processed?

```
binder_size_t buffer_offset = 0;
if (!IS_ALIGNED(tr->offsets_size, sizeof(binder_size_t))) {
  goto err_bad_offset;
}
for (buffer_offset = off_start_offset; buffer_offset < off_end_offset;
    buffer_offset += sizeof(binder_size_t)) {
  // if error: goto err_bad_offset
}
err_bad_offset:
  binder_transaction_buffer_release(target_proc, NULL, t->buffer,
                                    buffer_offset, /*is_failure*/true);
```

# Vulnerability Description (4)

```
void binder_transaction_buffer_release(binder_proc *target_proc, binder_buffer *buffer,
                                       size_t failed_at /*buffer_offset*/, bool is_failure) {
  off_start_offset = ALIGN(buffer->data_size, sizeof(void *));
  off_end_offset = is_failure && failed_at /*0*/ ? failed_at
                                                 : off_start_offset + buffer->offsets_size;
  for (buffer_offset = off_start_offset; buffer_offset < off_end_offset;
       buffer_offset += sizeof(size_t)) {
    case BINDER_TYPE_BINDER: {
      flat_binder_object *fp = to_flat_binder_object(hdr);
      binder_node *node = binder_get_node(target_proc, fp->binder);
      binder_dec_node(node, hdr->type == BINDER_TYPE_BINDER, 0);
      binder_put_node(node);
    }
  }
}
```

# Vulnerability Description (4)

- There are cases where a buffer offset of zero is passed to this function with `is_failure` set to true, to indicate that the entire buffer needs to be cleaned

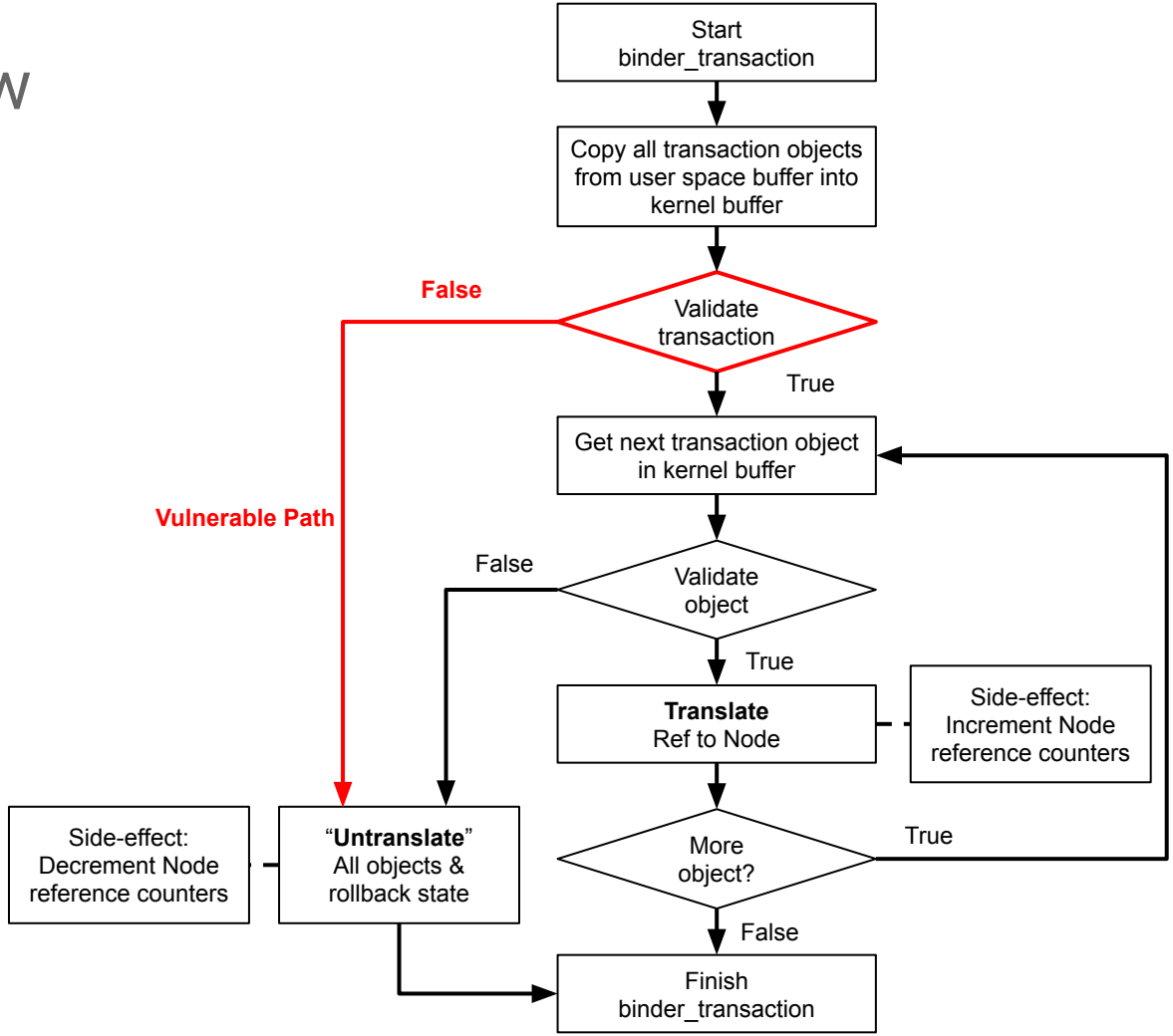
```
void binder_transaction_buffer_release(binder_proc *target_proc, binder_buffer *buffer,
                                       size_t failed_at /*buffer_offset*/, bool is_failure) {
  off_start_offset = ALIGN(buffer->data_size, sizeof(void *));
  off_end_offset = is_failure && failed_at /*0*/ ? failed_at
                                                 : off_start_offset + buffer->offsets_size;
  for (buffer_offset = off_start_offset; buffer_offset < off_end_offset;
       buffer_offset += sizeof(size_t)) {
    case BINDER_TYPE_BINDER: {
      flat_binder_object *fp = to_flat_binder_object(hdr);
      binder_node *node = binder_get_node(target_proc, fp->binder);
      binder_dec_node(node, hdr->type == BINDER_TYPE_BINDER, 0);
      binder_put_node(node);
    }
  }
}
```

# Vulnerability Description (4)

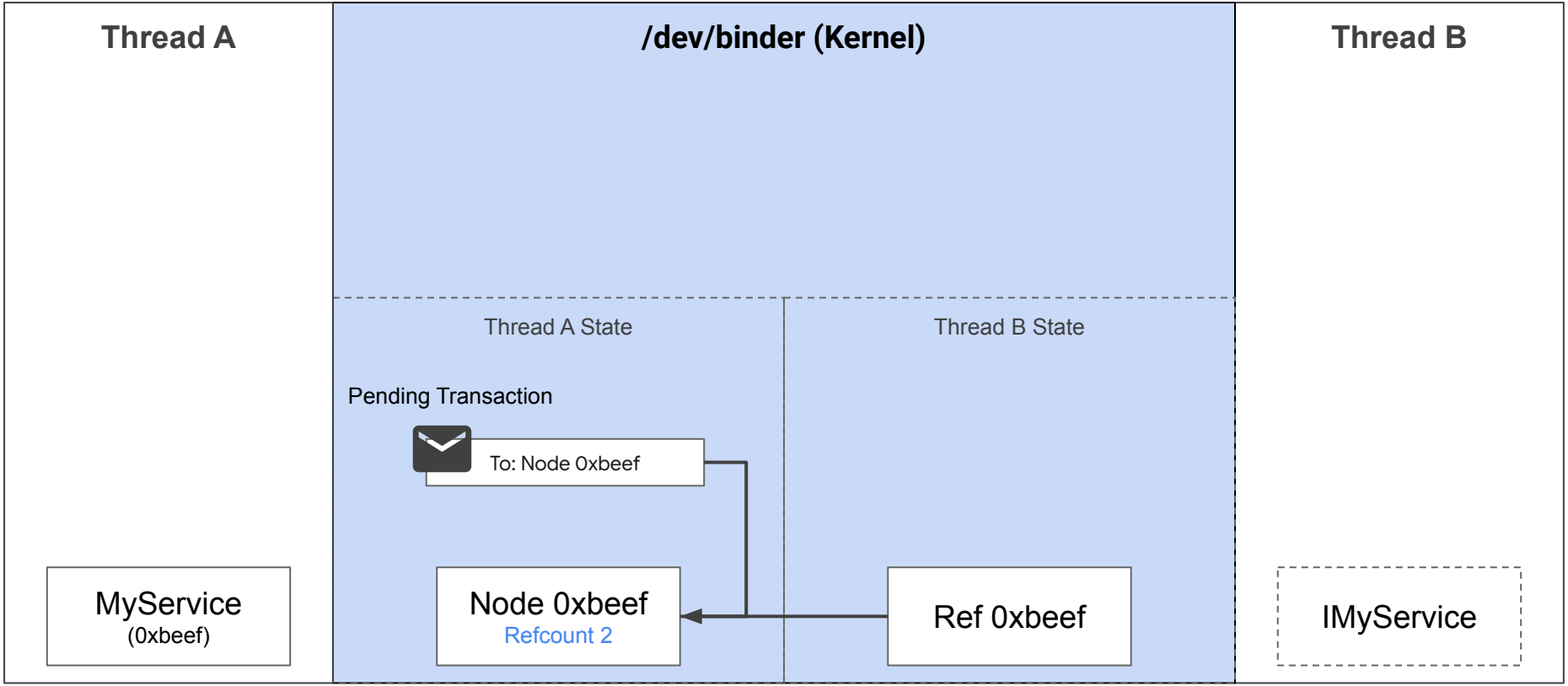
- There are cases where a buffer offset of zero is passed to this function with `is_failure` set to true, to indicate that the entire buffer needs to be cleaned

```
void binder_transaction_buffer_release(binder_proc *target_proc, binder_buffer *buffer,
                                       size_t failed_at /*buffer_offset*/, bool is_failure) {
  off_start_offset = ALIGN(buffer->data_size, sizeof(void *));
  off_end_offset = is_failure && failed_at /*0*/ ? failed_at
                                                 : off_start_offset + buffer->offsets_size;
  for (buffer_offset = off_start_offset; buffer_offset < off_end_offset;
       buffer_offset += sizeof(size_t)) {
    case BINDER_TYPE_BINDER: {
      flat_binder_object *fp = to_flat_binder_object(hdr);
      binder_node *node = binder_get_node(target_proc, fp->binder);
      binder_dec_node(node, hdr->type == BINDER_TYPE_BINDER, 0);
      binder_put_node(node);
    }
  }
}
```

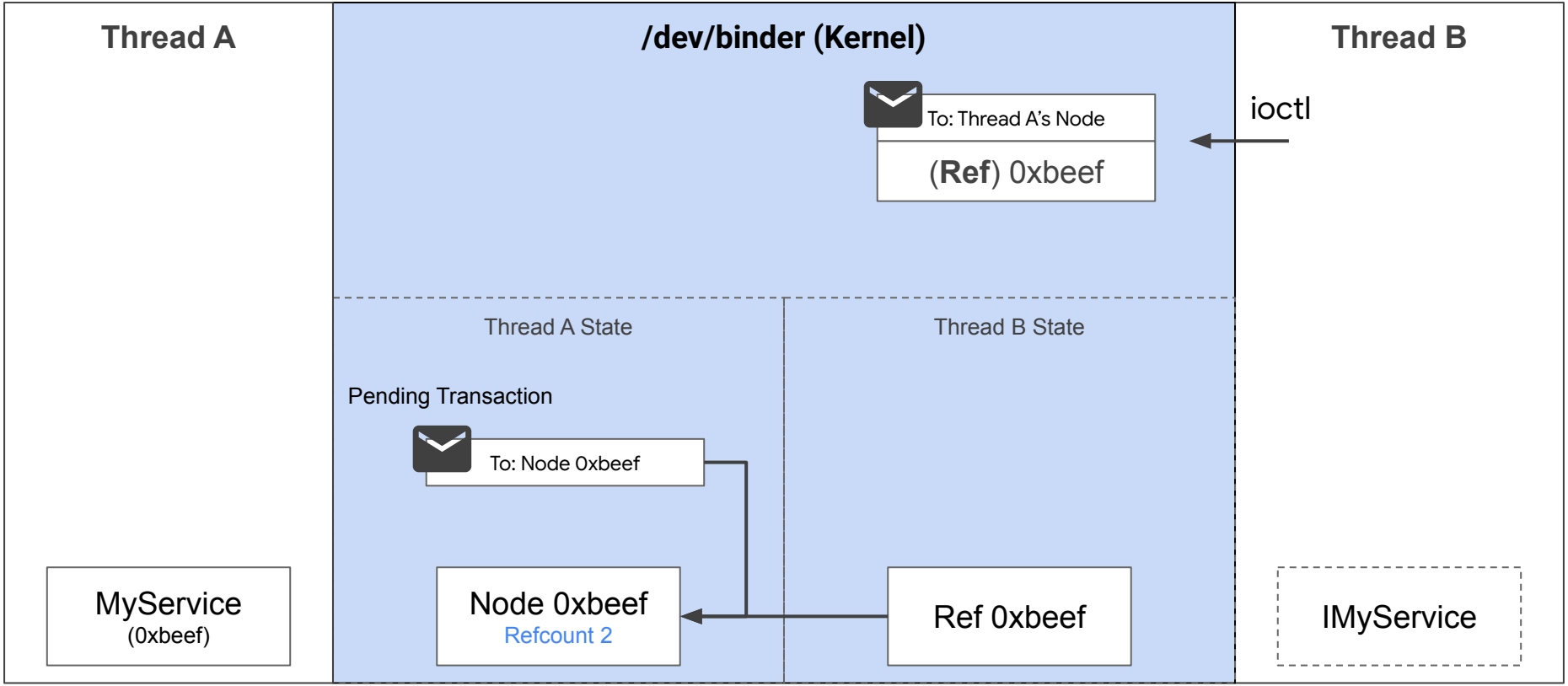
# Vulnerability Overview



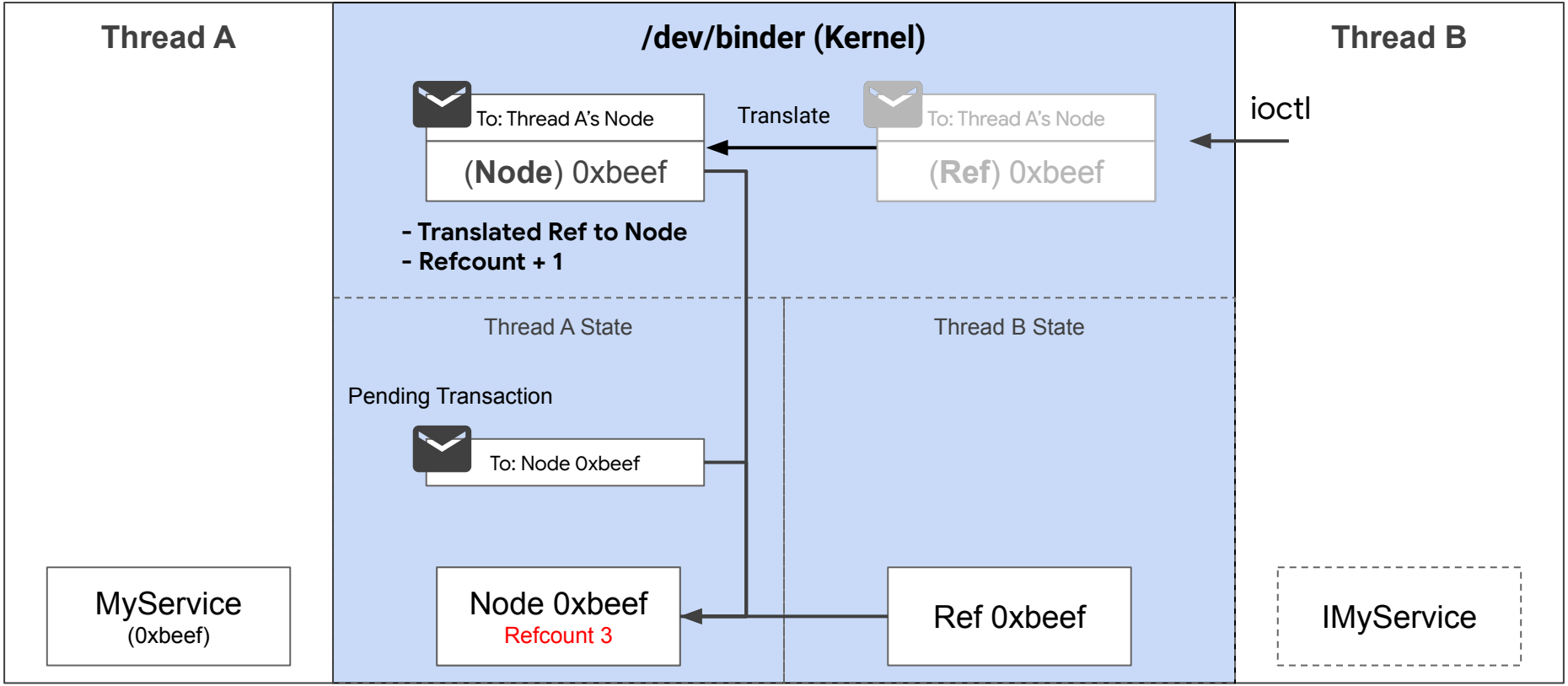
# Vulnerability Explained (Initial State)



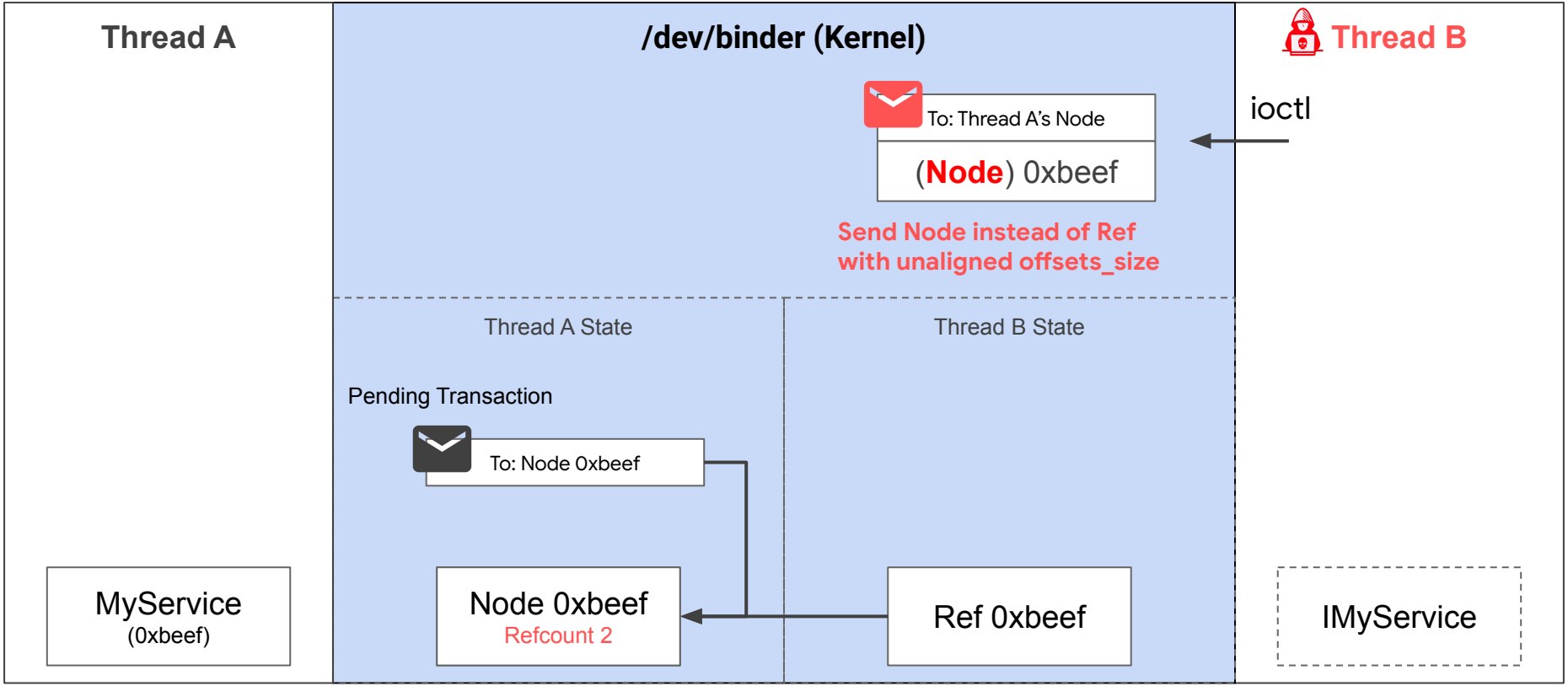
# Vulnerability Explained (Normal Workflow)



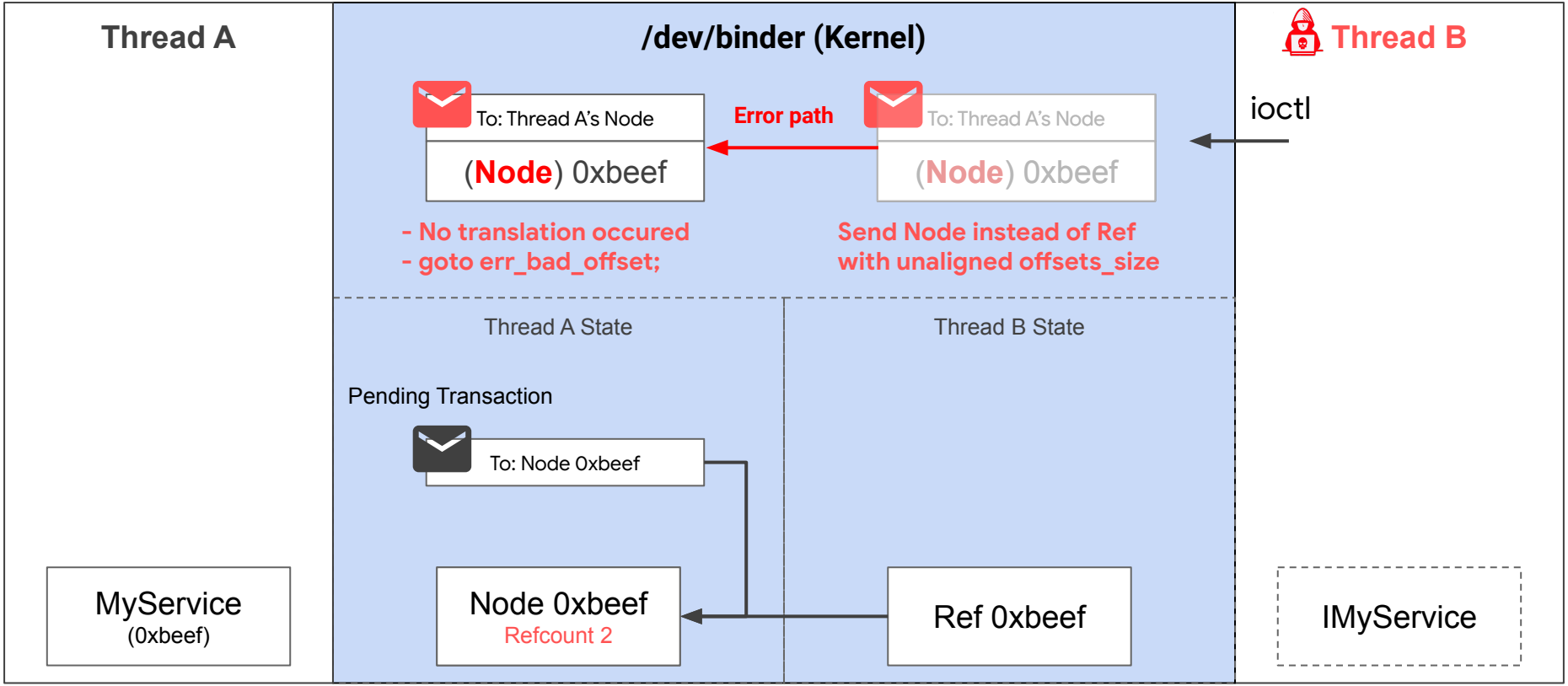
# Vulnerability Explained (Normal Workflow)



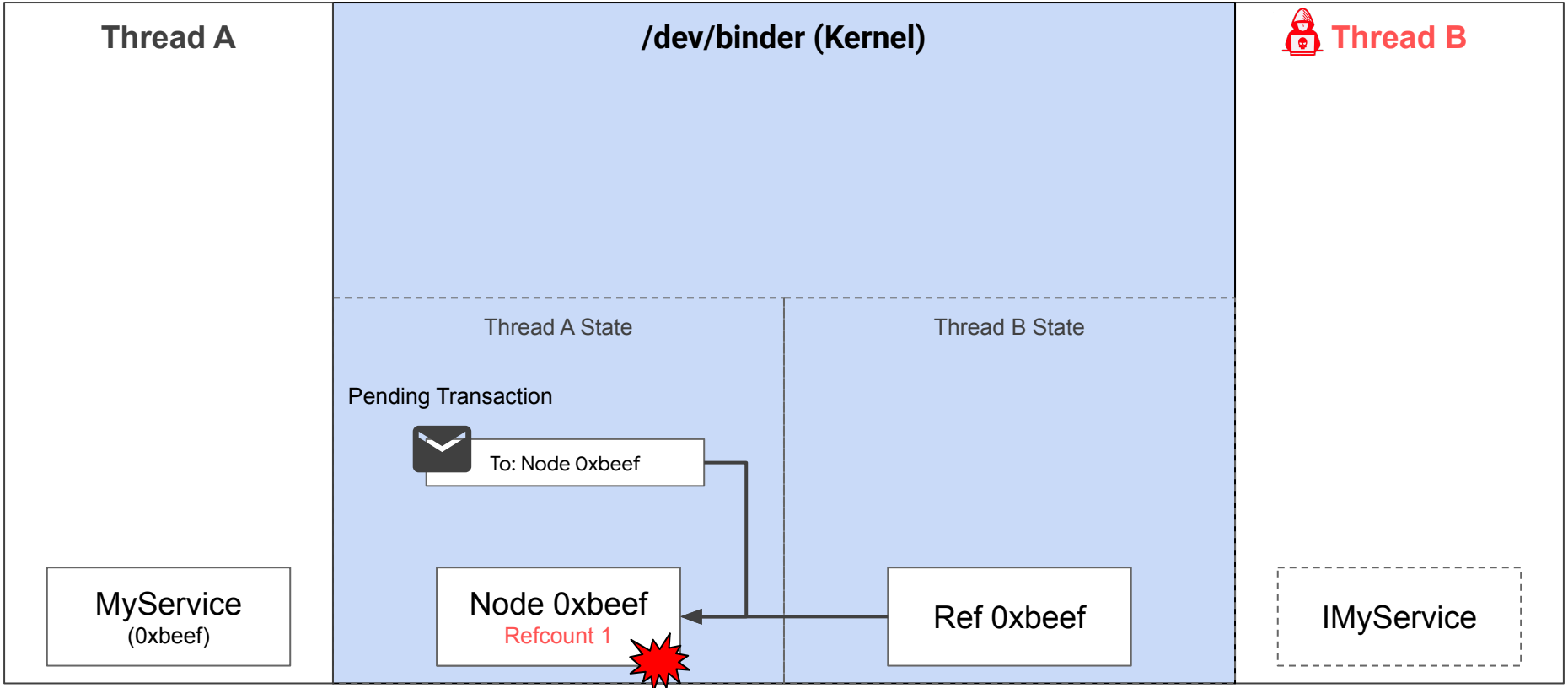
# Vulnerability Explained (Attacker Workflow)



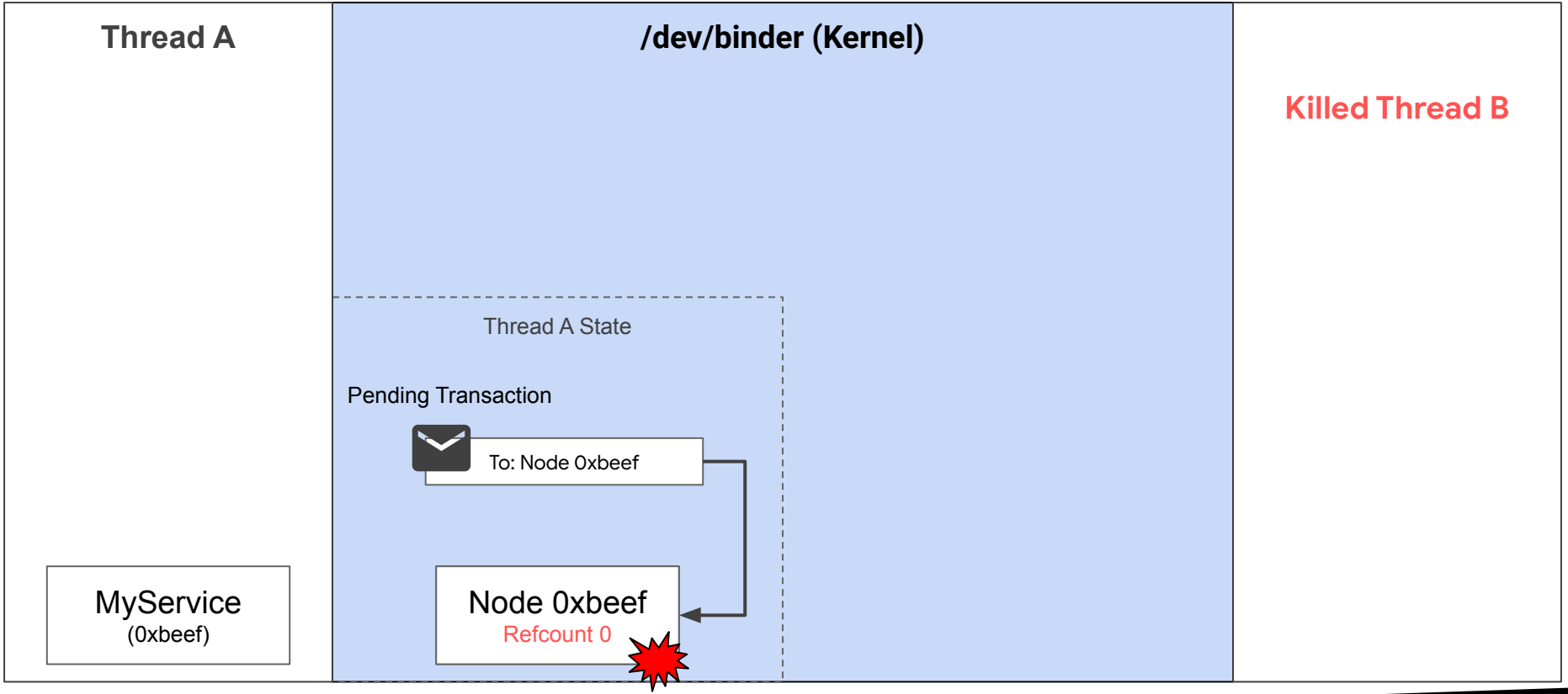
# Vulnerability Explained (Attacker Workflow)



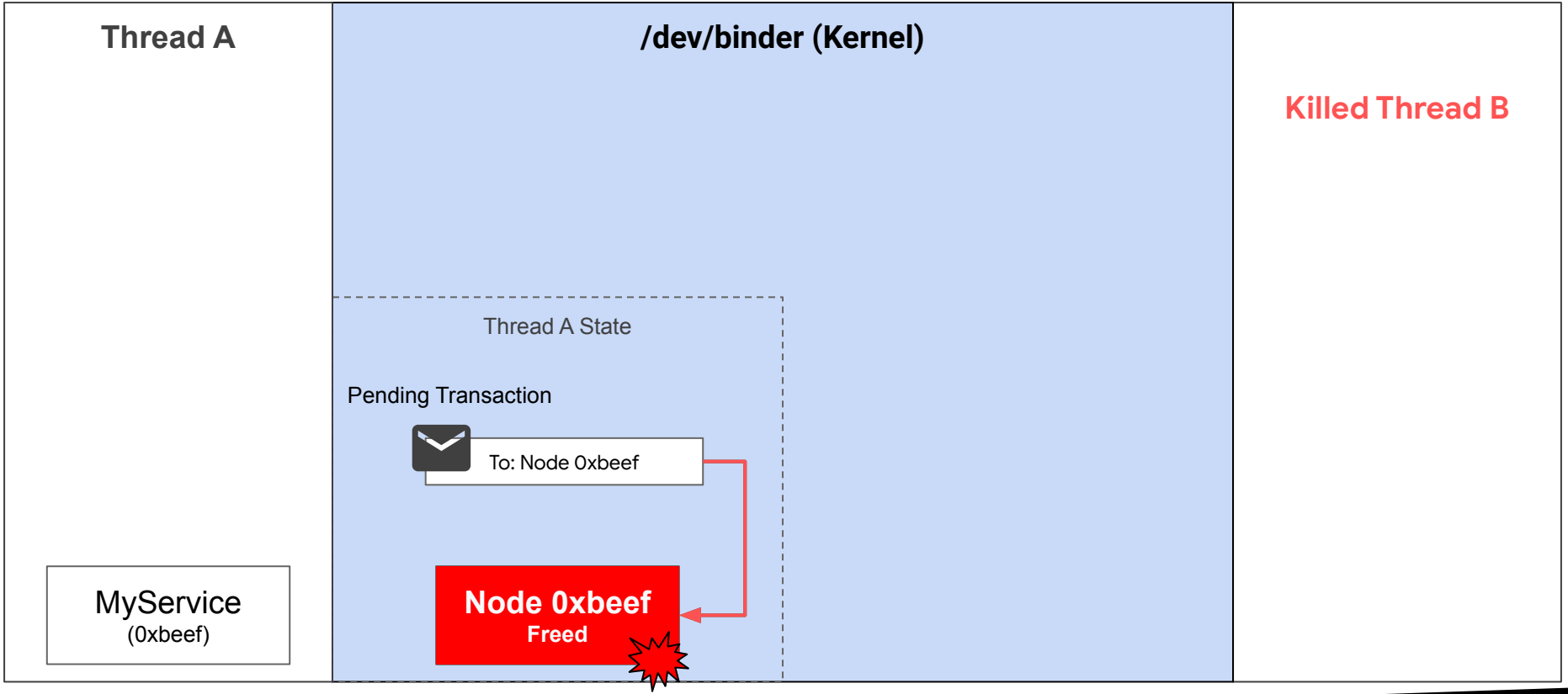
# Vulnerability Explained (Attacker Workflow)



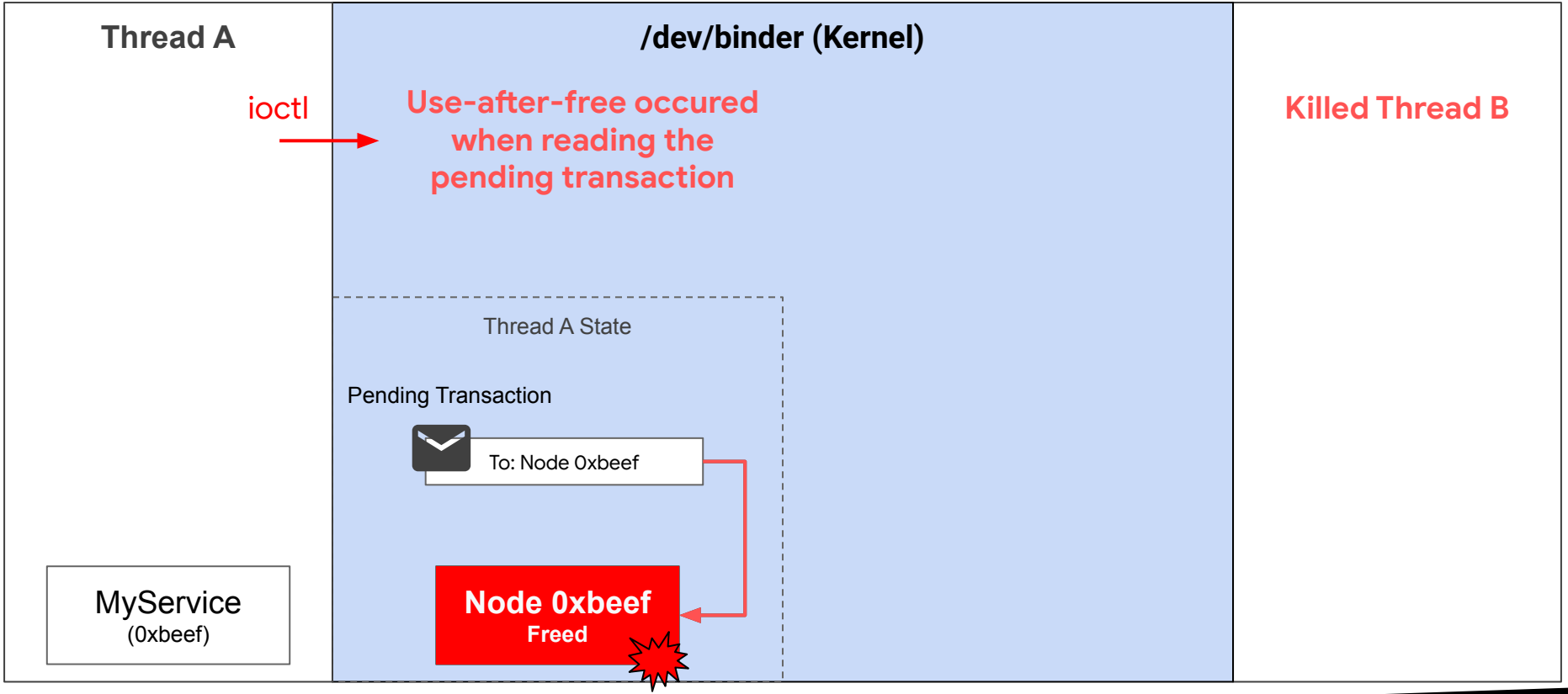
# Vulnerability Explained (Attacker Workflow)



# Vulnerability Explained (Attacker Workflow)



# Vulnerability Explained (Attacker Workflow)



# CVE-2023-20938 Remediation

- CVE-2023-20938 was identified via fuzzing on `android13-5.10` kernel
- The test case wasn't reproducible on `android13-5.15` ACK due to this patch[1]

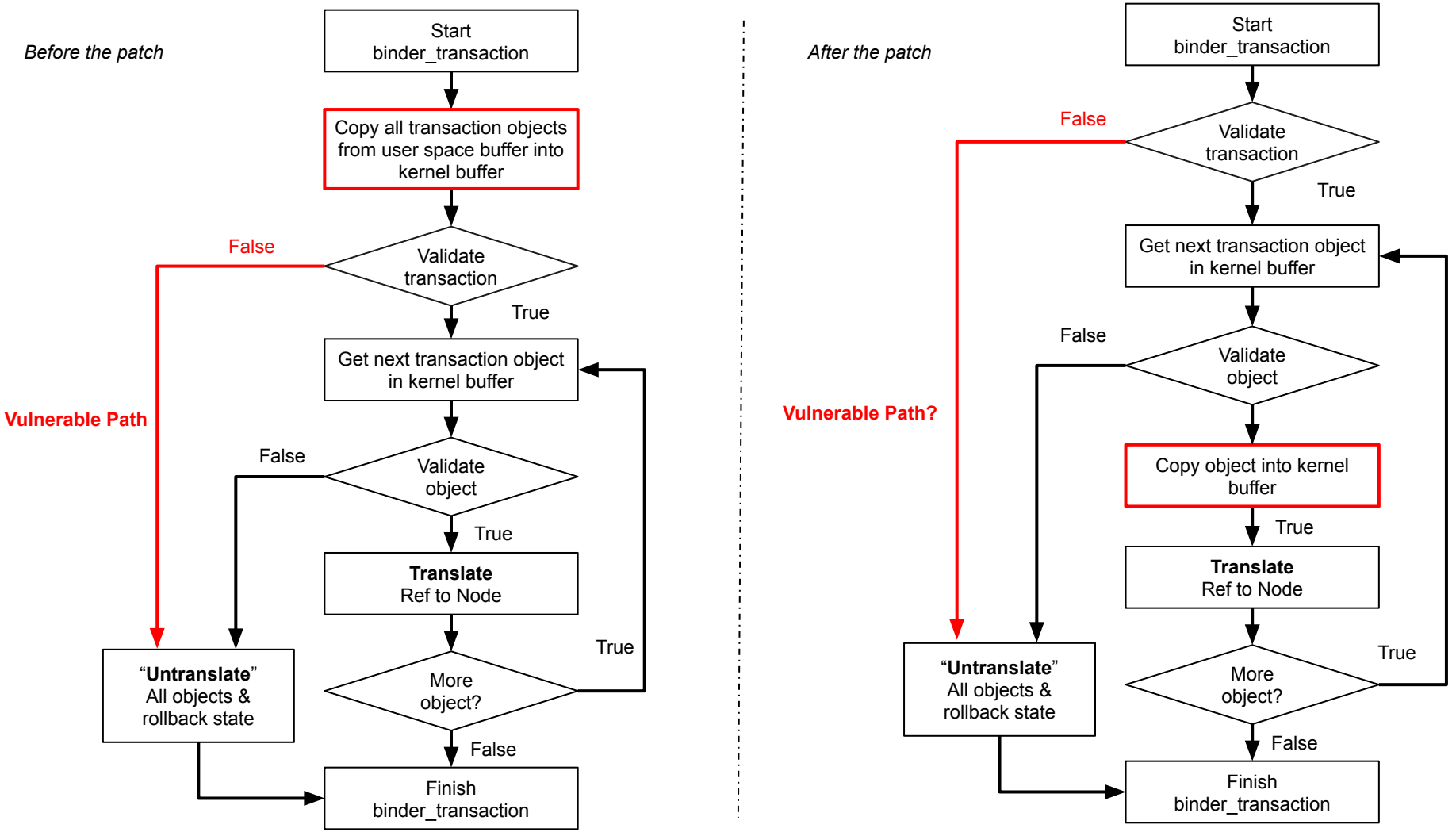
**Before patch**: all the binder transaction objects are copied from user space into the binder kernel buffer **before** translating the objects

**After patch**: binder transaction objects are copied from user space into the binder kernel buffer **during** translation of the objects as they are being processed

- CVE-2023-20938 was fixed by back-porting the patch to the vulnerable kernels in 2023-02-01 ASB[2]:
  - android13-5.10
  - android12-5.4

```
-     if (binder_alloc_copy_user_to_buffer(
-                         &target_proc->alloc,
-                         t->buffer, 0,
-                         (const void __user *)
-                               (uintptr_t)tr->data.ptr.buffer,
-                         tr->data_size)) {

// ...
for (buffer_offset = off_start_offset;
     buffer_offset < off_end_offset;
     buffer_offset += sizeof(binder_size_t)) {

+  if (copy_size && (user_offset > object_offset ||
+                         binder_alloc_copy_user_to_buffer(
+                               &target_proc->alloc,
+                               t->buffer, user_offset,
+                               user_buffer + user_offset,
+                               copy_size))) {

  // translate binder object
}
```

[1] https://github.com/torvalds/linux/commit/6d98eb95b450a75adb4516a1d33652dc78d2b20c
[2] https://source.android.com/docs/security/bulletin/2023-02-01#kernel


Before the patch
Start
binder_transaction
Copy all transaction objects
from user space buffer into
kernel buffer
False
Validate
transaction
True
Get next transaction object
in kernel buffer
Vulnerable Path
False
Validate
object
True
Translate
Ref to Node
True
"Untranslate"
All objects &
rollback state
More
object?
False
Finish
binder_transaction
After the patch
Start
binder_transaction
False
Validate
transaction
True
Get next transaction object
in kernel buffer
False
Validate
object
Vulnerable Path?
Copy object into kernel
buffer
True
Translate
Ref to Node
True
"Untranslate"
All objects &
rollback state
More
object?
False
Finish
binder_transaction

# Discovery of CVE-2023-21255

- **The binder kernel buffer isn't zeroed between ioctls**
- Any data from the previously processed transactions would remain in the buffer
  - … and would be processed in `binder_transaction_buffer_release` on the error path!

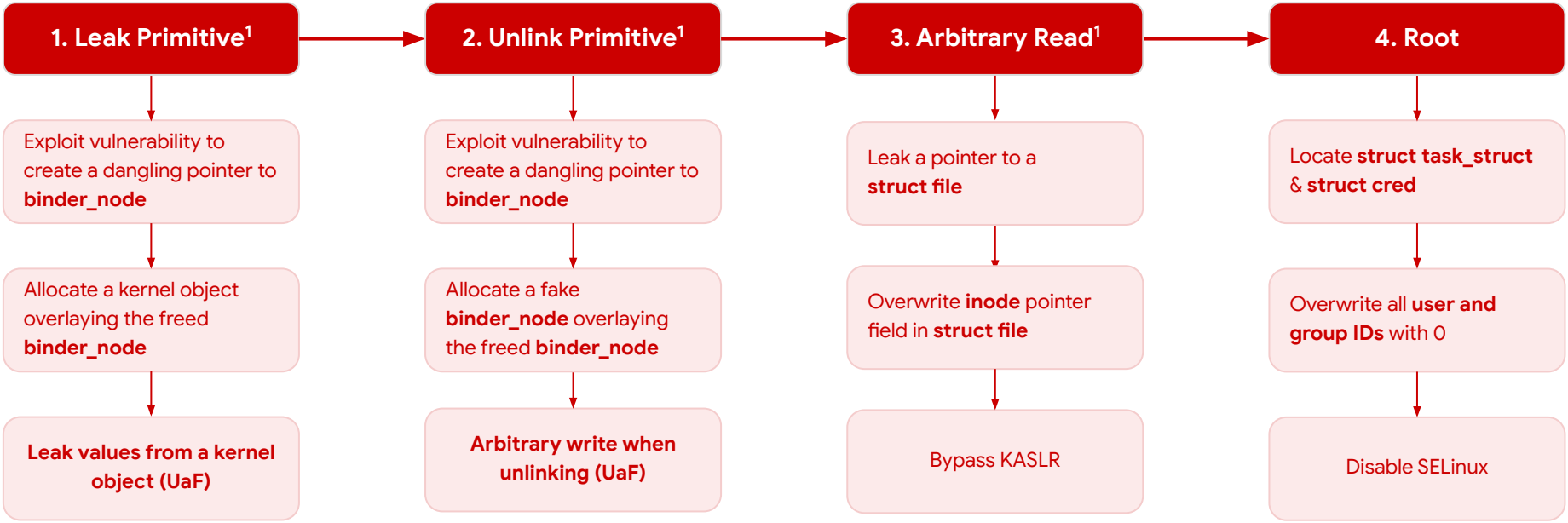
- The fuzzer identified another test case triggering the same UAF issue
  - Send a valid transaction with a binder node to pollute the binder kernel buffer
  - Send the malformed transaction with misaligned offsets to trigger the error path
    - `binder_transaction_buffer_release` would roll back the binder node translation at the previous step => UAF

- CVE-2023-21255 was fixed in 2023-07-01 ASB[1]

*[1]* https://source.android.com/docs/security/bulletin/2023-07-01#kernel

# Exploitation

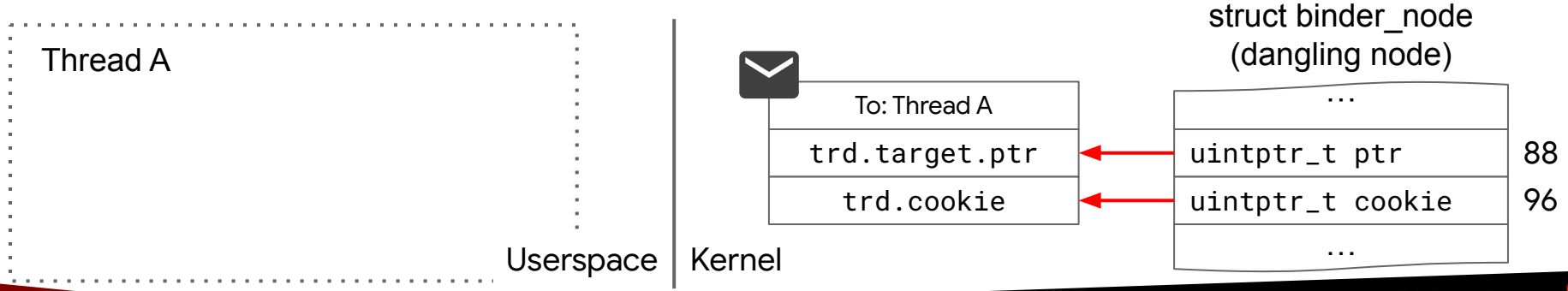
# Exploitation Steps




*[1] https://labs.bluefrostsecurity.de/blog/2020/04/08/cve-2020-0041-part-2-escalating-to-root/*

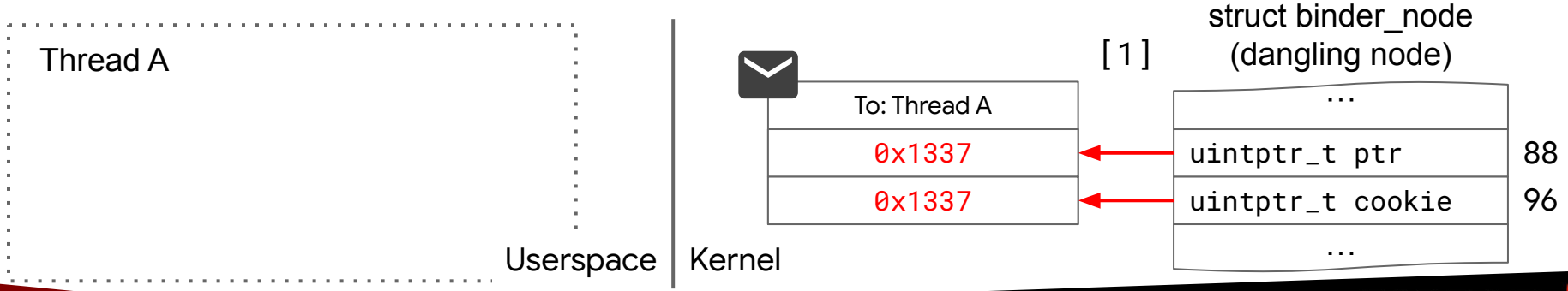
# Leak Primitive

```
static int binder_thread_read(...)
  struct binder_transaction_data_secctx tr;
  struct binder_transaction_data *trd = &tr.transaction_data;
...
          trd->target.ptr = target_node->ptr; [1]
          trd->cookie =  target_node->cookie; [1]
...
  if (copy_to_user(ptr, &tr, trsize)) { [2]
```



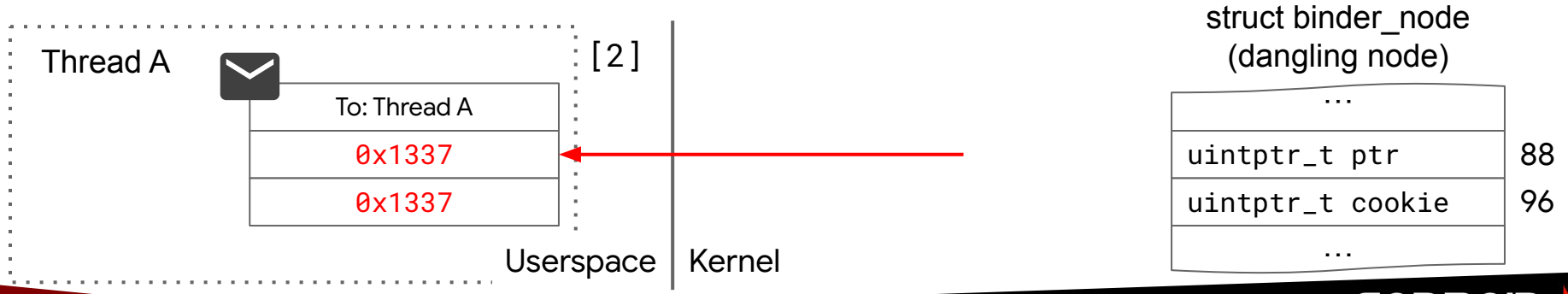
# Leak Primitive

```
static int binder_thread_read(...)
  struct binder_transaction_data_secctx tr;
  struct binder_transaction_data *trd = &tr.transaction_data;
...
          trd->target.ptr = target_node->ptr; [1]
          trd->cookie =  target_node->cookie; [1]
...
  if (copy_to_user(ptr, &tr, trsize)) { [2]
```



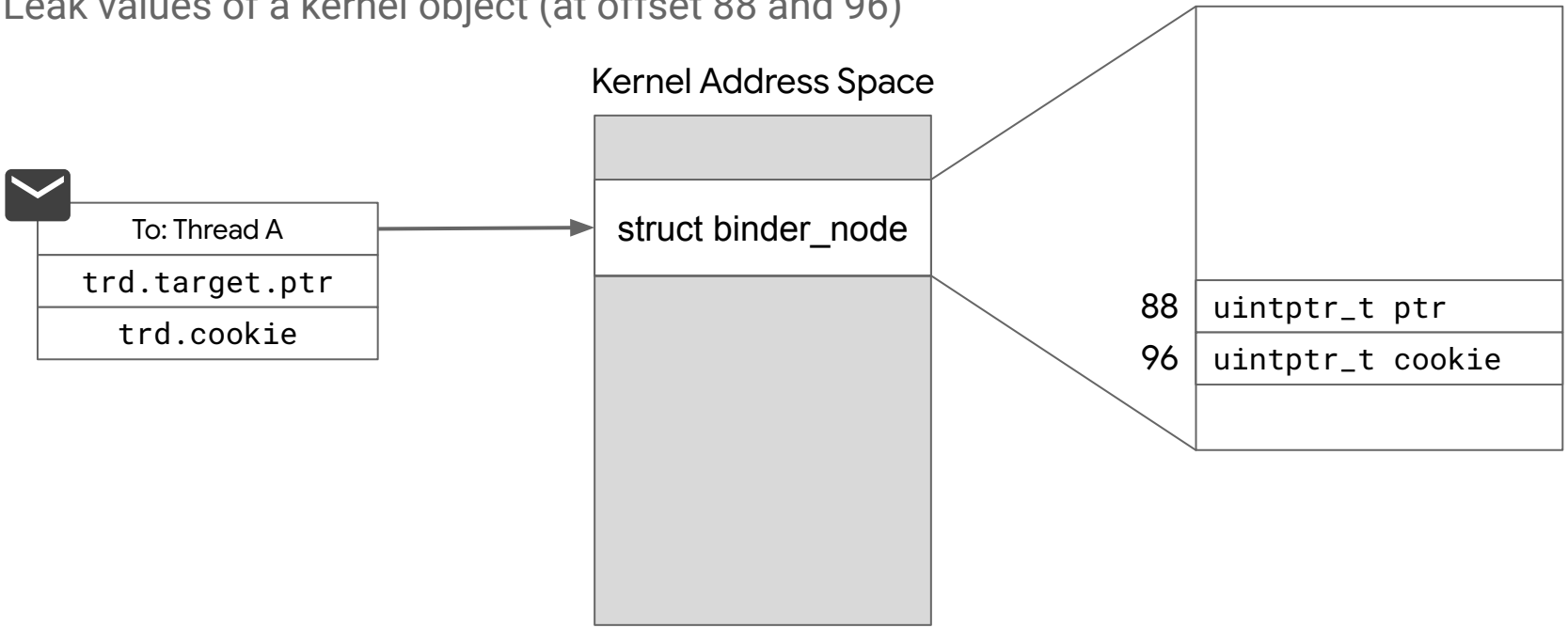
# Leak Primitive

```
static int binder_thread_read(...)
  struct binder_transaction_data_secctx tr;
  struct binder_transaction_data *trd = &tr.transaction_data;
...
          trd->target.ptr = target_node->ptr; [1]
          trd->cookie =  target_node->cookie; [1]
...
  if (copy_to_user(ptr, &tr, trsize)) { [2]
```



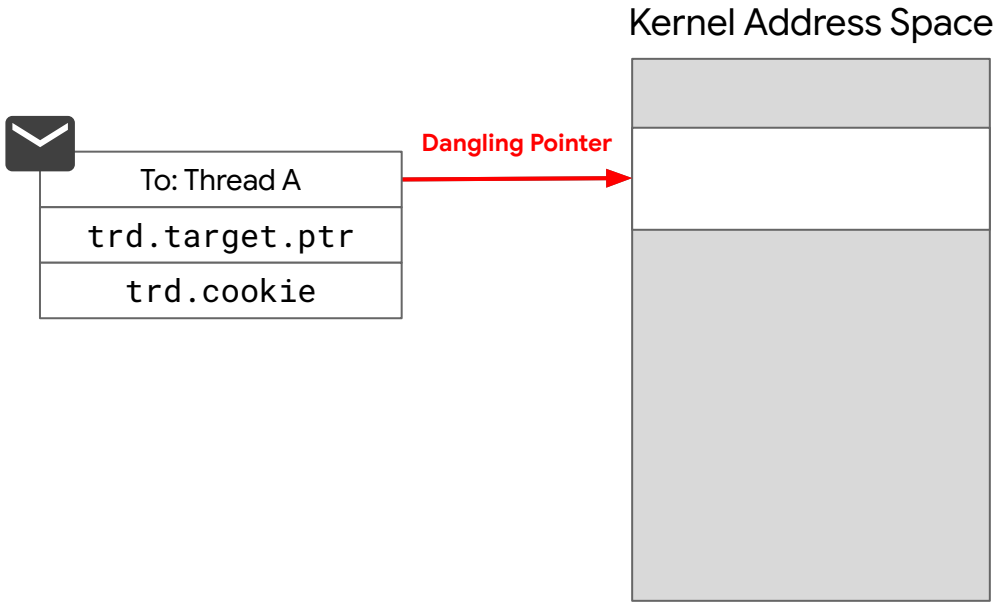
# Leak Primitive

1. Exploit the vulnerability to free a **binder_node**
2. Allocate a kernel object at the same memory location
3. Leak values of a kernel object (at offset 88 and 96)



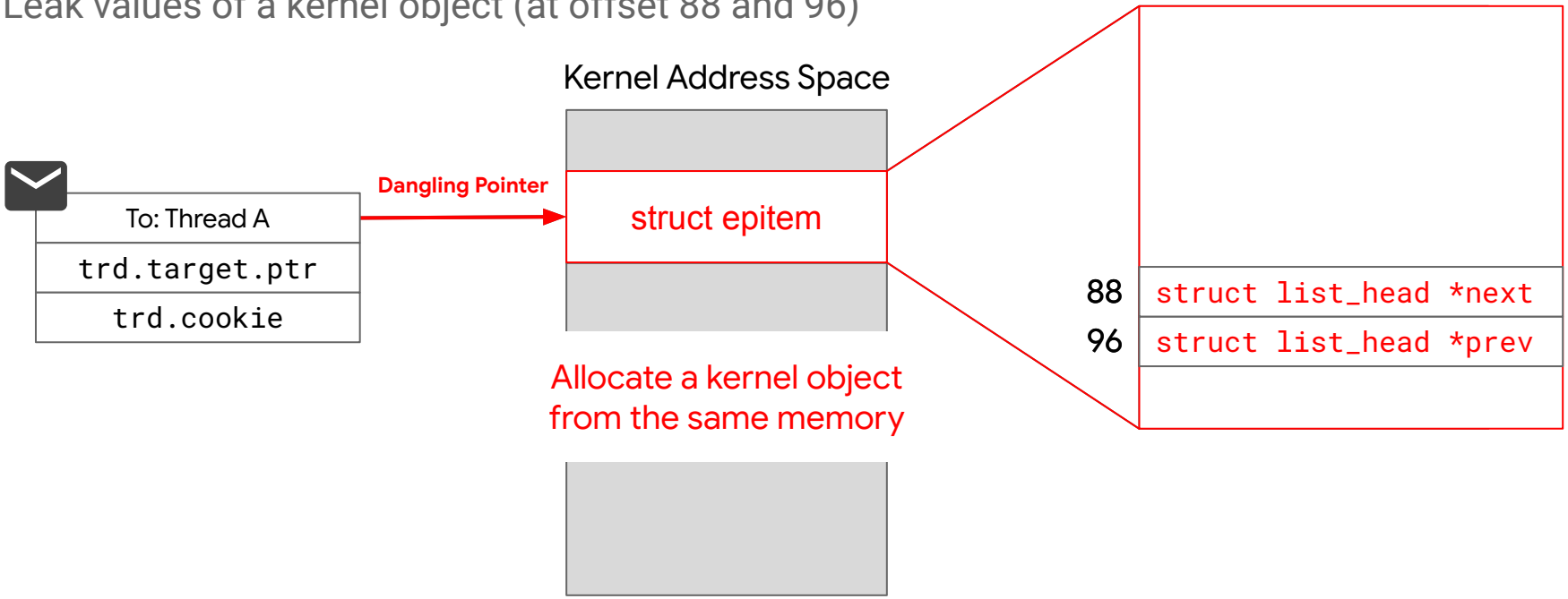
# Leak Primitive

1. Exploit the vulnerability to free a **binder_node**
2. Allocate a kernel object at the same memory location
3. Leak values of a kernel object (at offset 88 and 96)



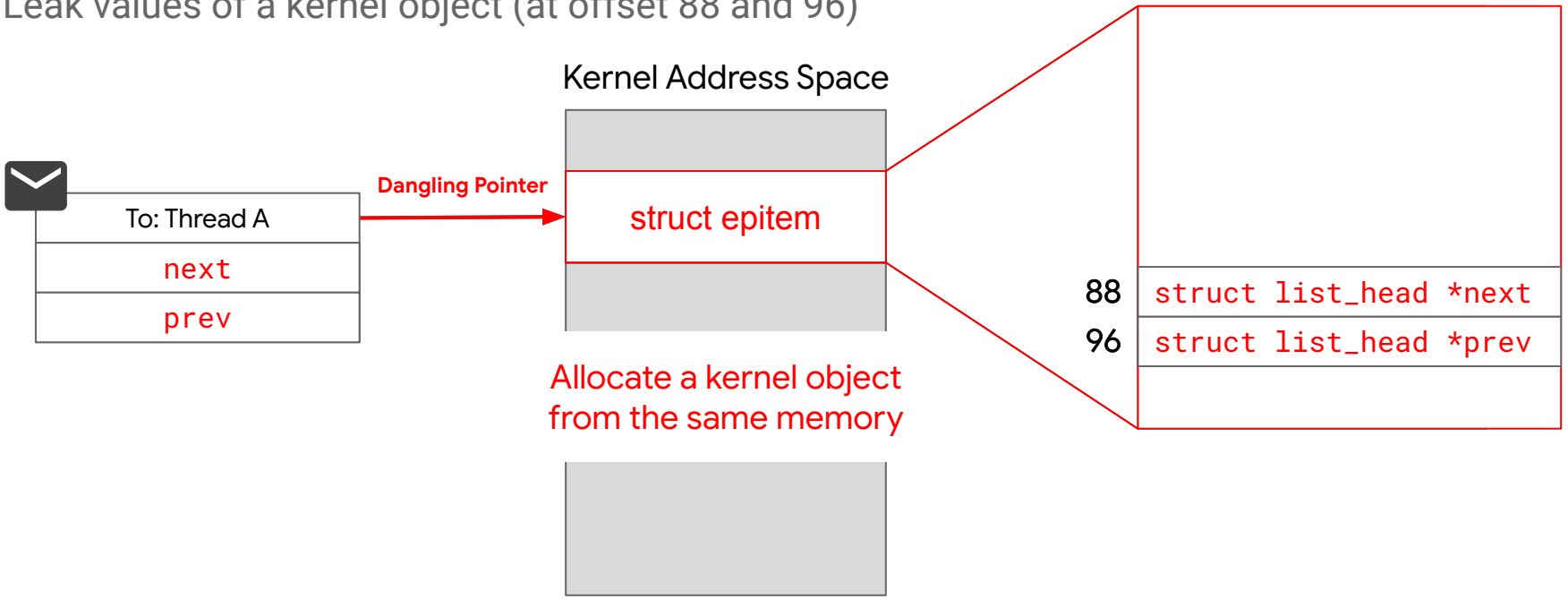
# Leak Primitive

1. Exploit the vulnerability to free a **binder_node**
2. Allocate a kernel object at the same memory location
3. Leak values of a kernel object (at offset 88 and 96)



# Leak Primitive

1. Exploit the vulnerability to free a **binder_node**
2. Allocate a kernel object at the same memory location
3. Leak values of a kernel object (at offset 88 and 96)



# Leak Primitive

How can we overlay another kernel object (**epitem**) onto the freed **binder_node**?

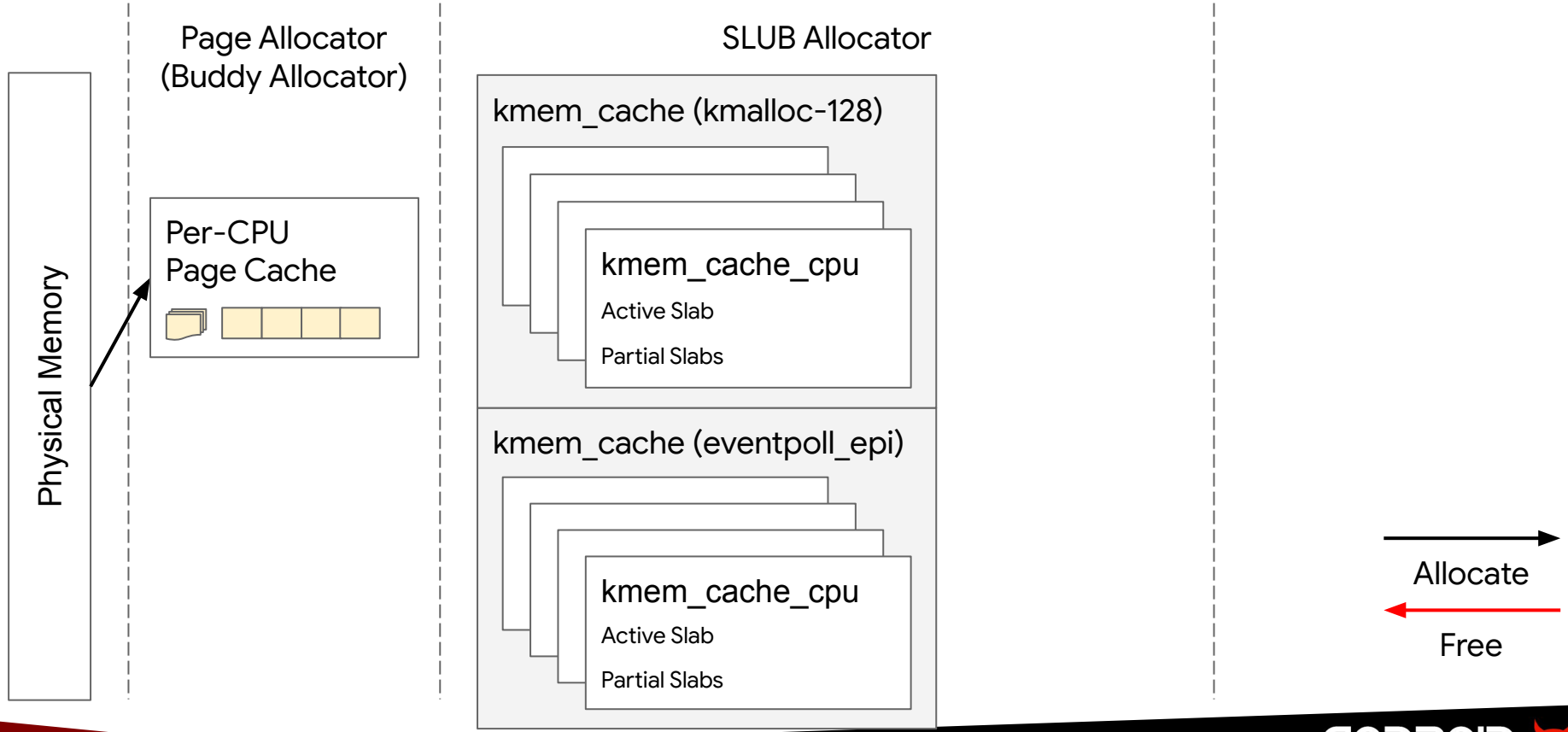
- Much easier in userspace (e.g. glibc)
  - malloc uses the same freelist regardless of object types
- Linux Kernel is more complicated: SLUB allocator
  - Different object types can be allocated from different caches
- Many mitigations enabled in recent years
  - CONFIG_SLAB_FREELIST_HARDENED
  - CONFIG_SLAB_FREELIST_RANDOM
  - GFP_KERNEL_ACCOUNT*
  - **No more cache aliasing for `epitem` (SLAB_ACCOUNT)**
  - No unprivileged userfaultd (for heap spraying)

* Removed in 5.9 and added back in 5.14 (we're targeting 5.10)

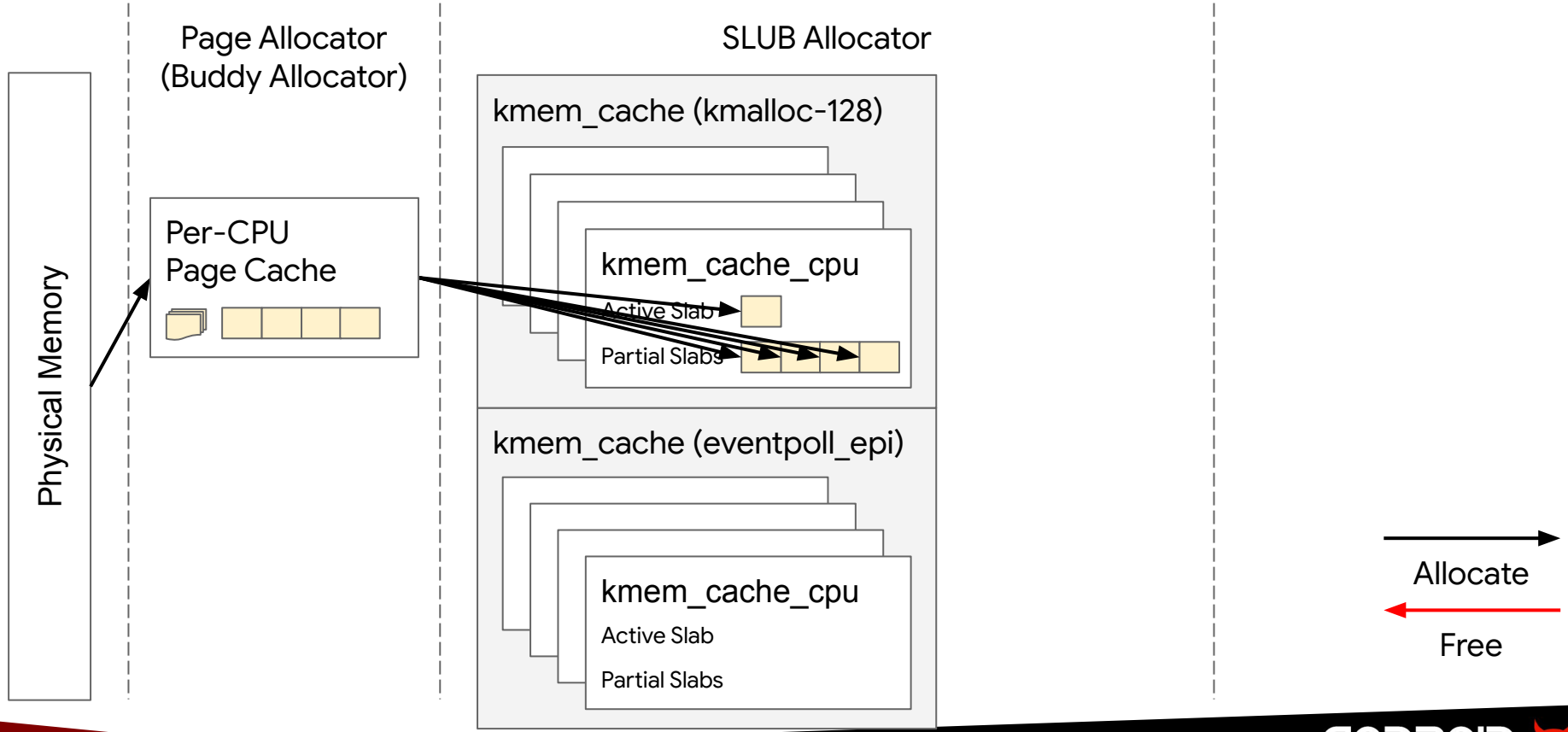
# Cross-cache Attack (SLUB Allocator)

# Cross-cache Attack (SLUB Allocator)

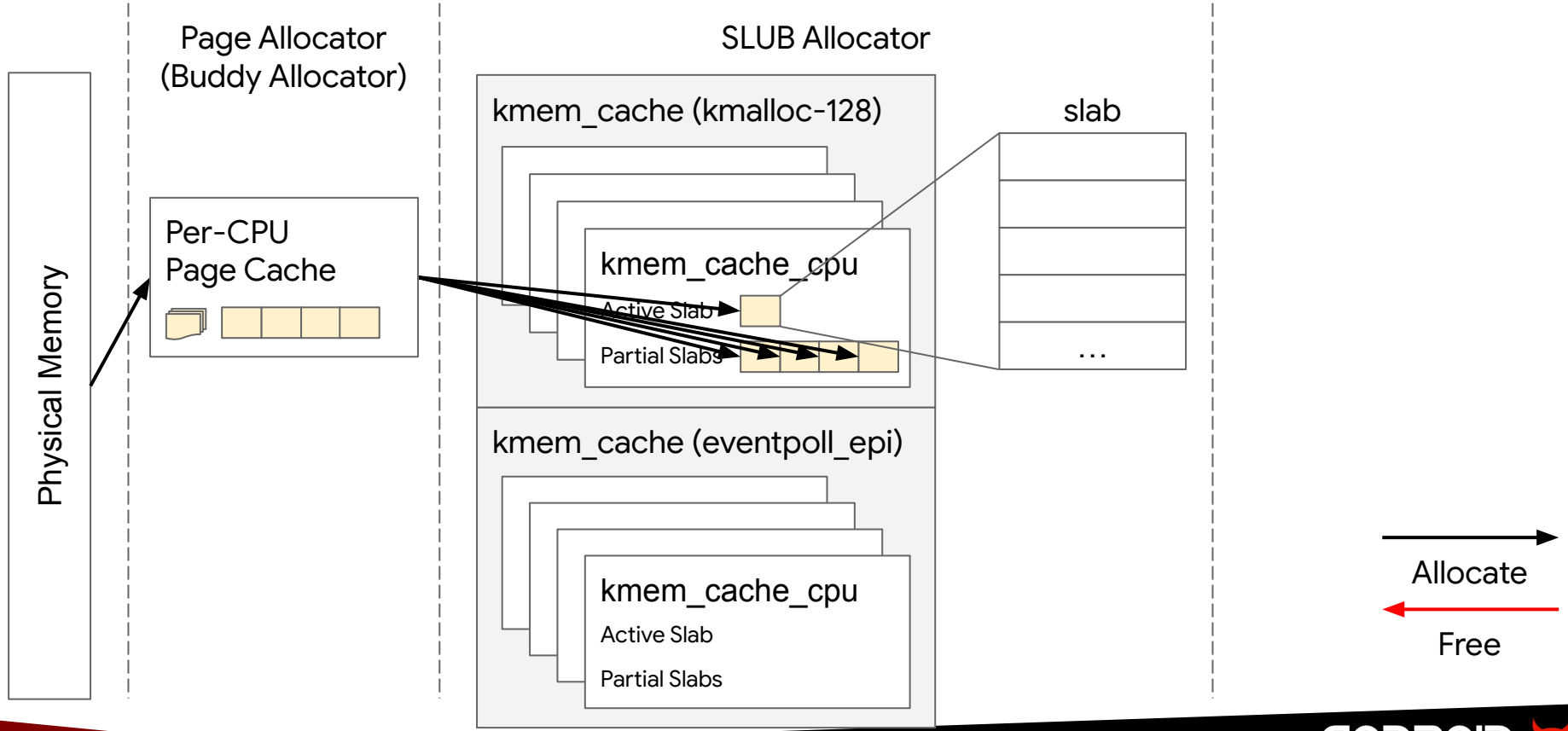
# Cross-cache Attack (SLUB Allocator)



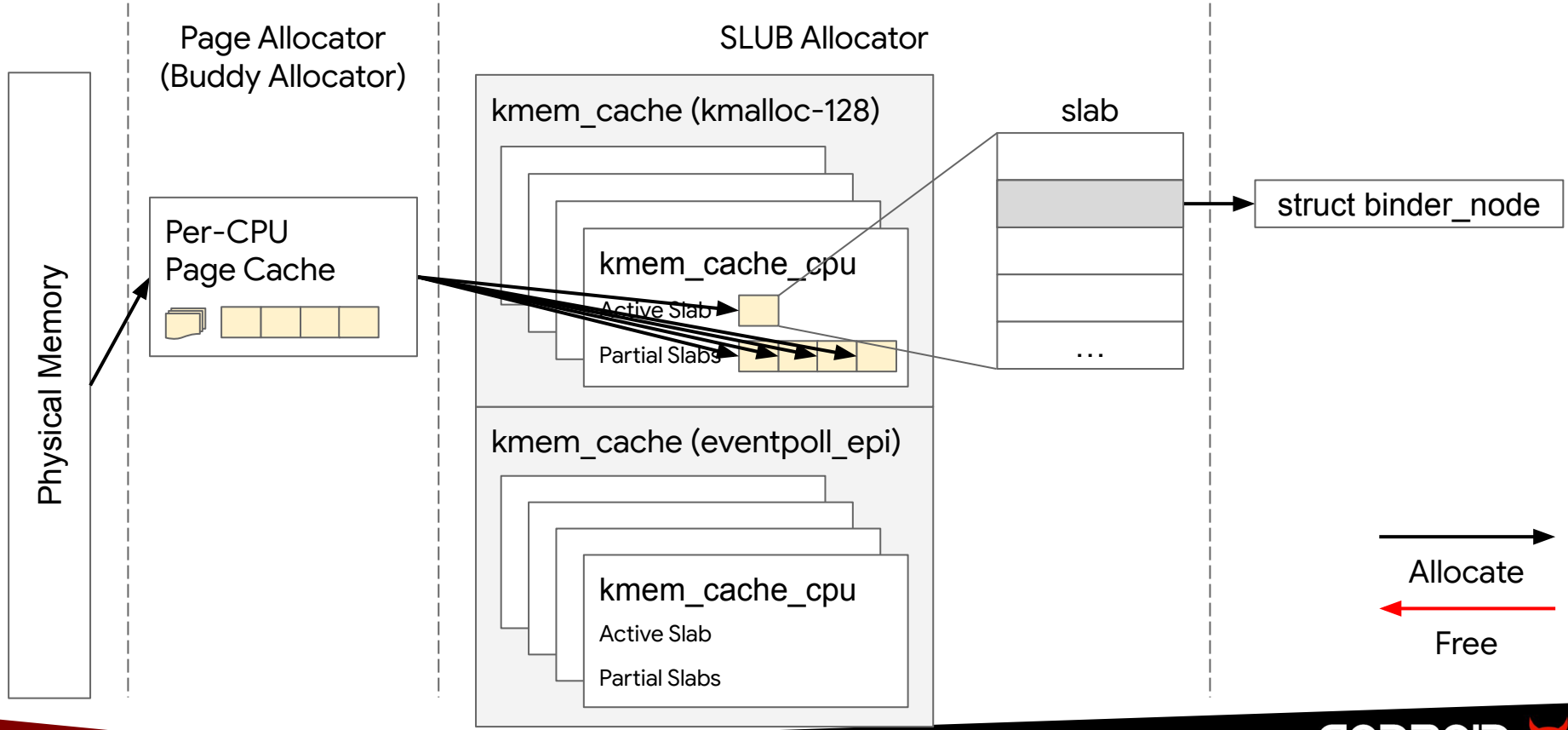
# Cross-cache Attack (SLUB Allocator)



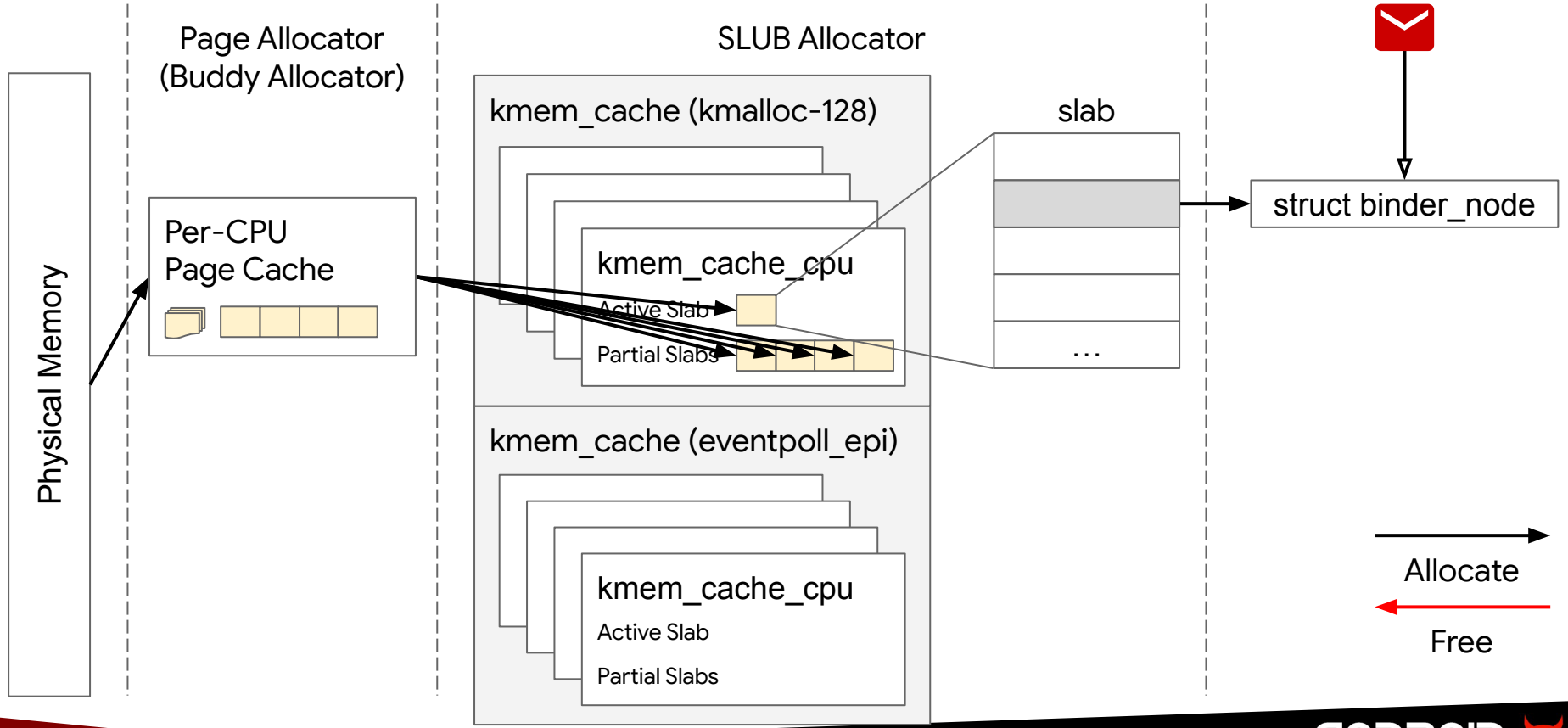
# Cross-cache Attack (SLUB Allocator)



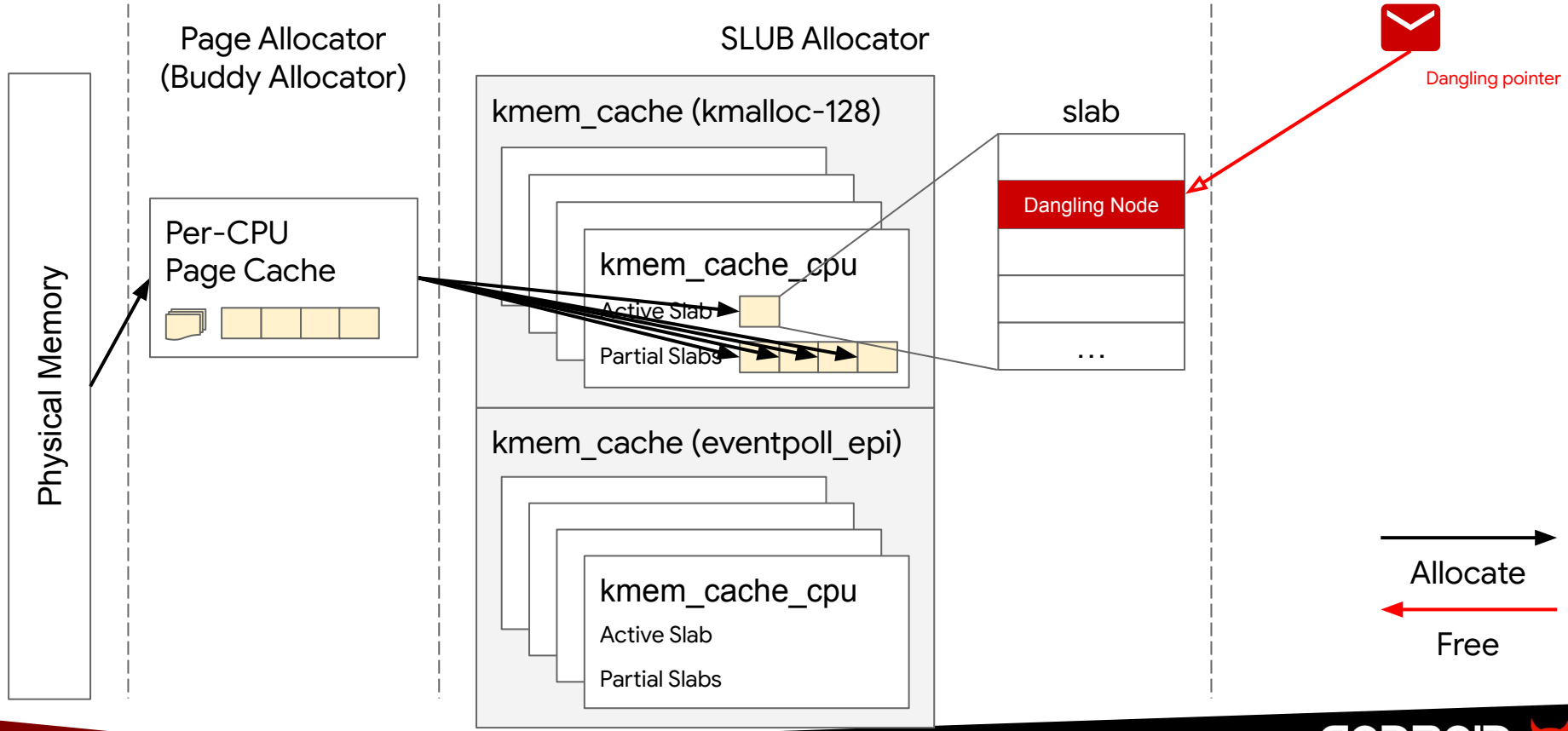
# Cross-cache Attack (SLUB Allocator)



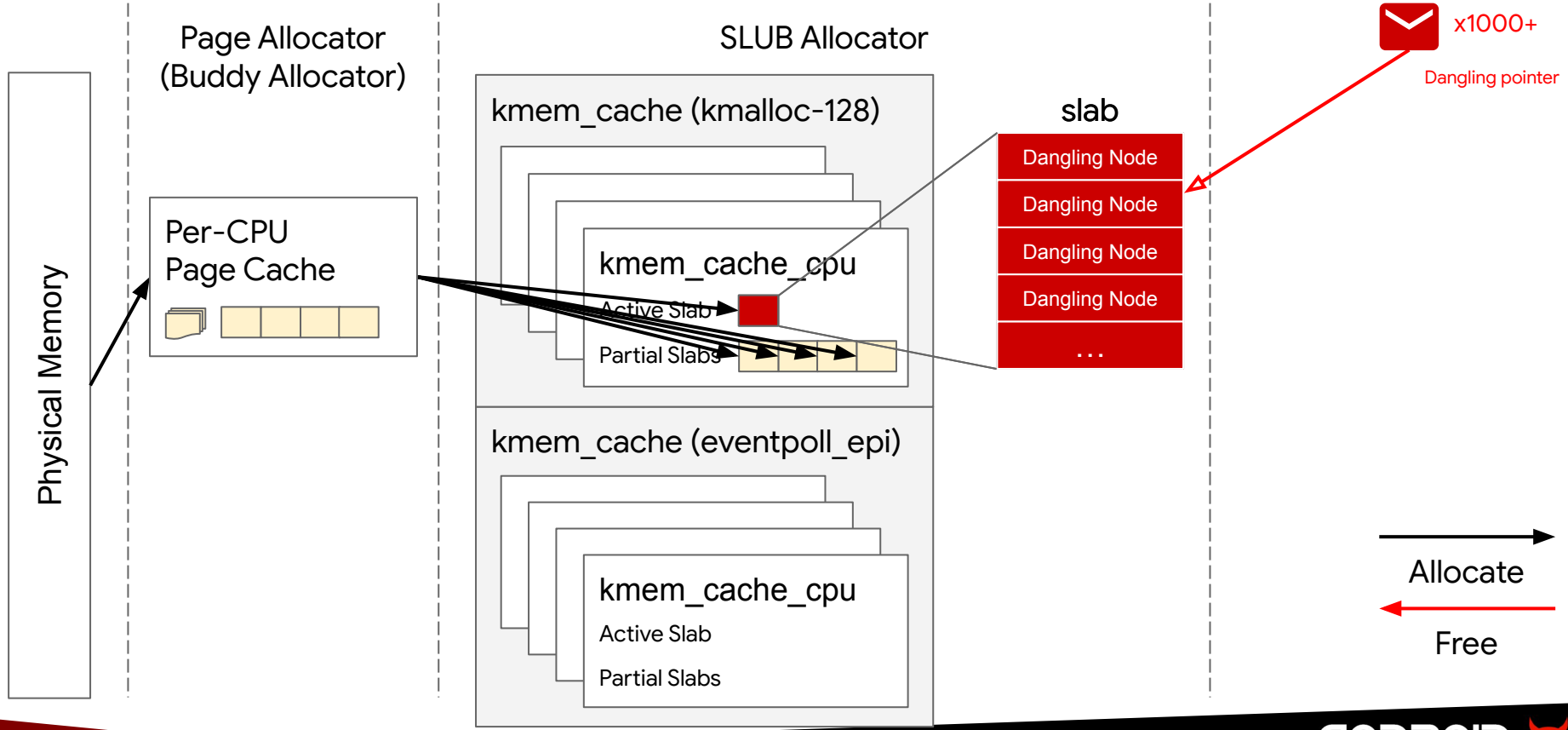
# Cross-cache Attack (SLUB Allocator)



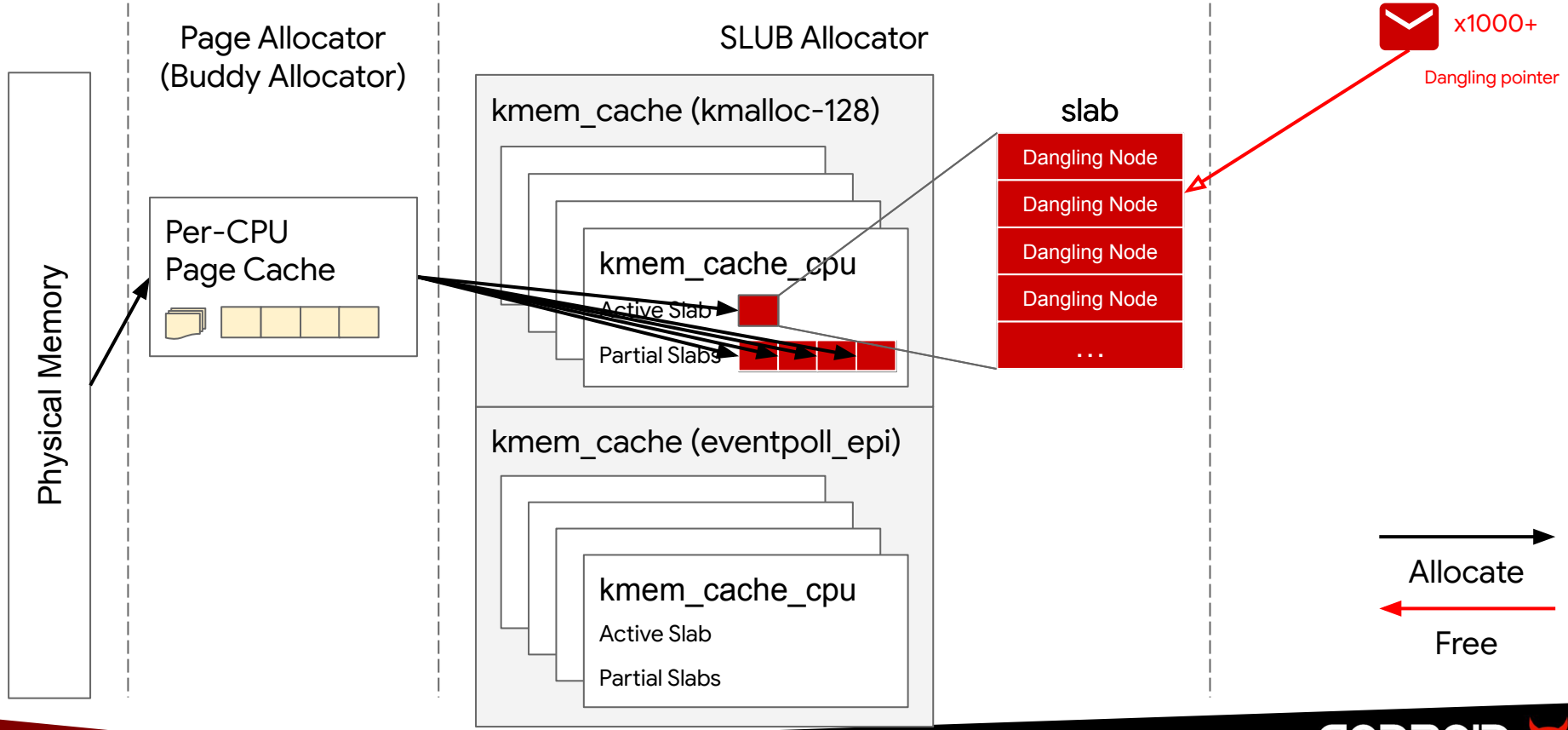
# Cross-cache Attack (SLUB Allocator)



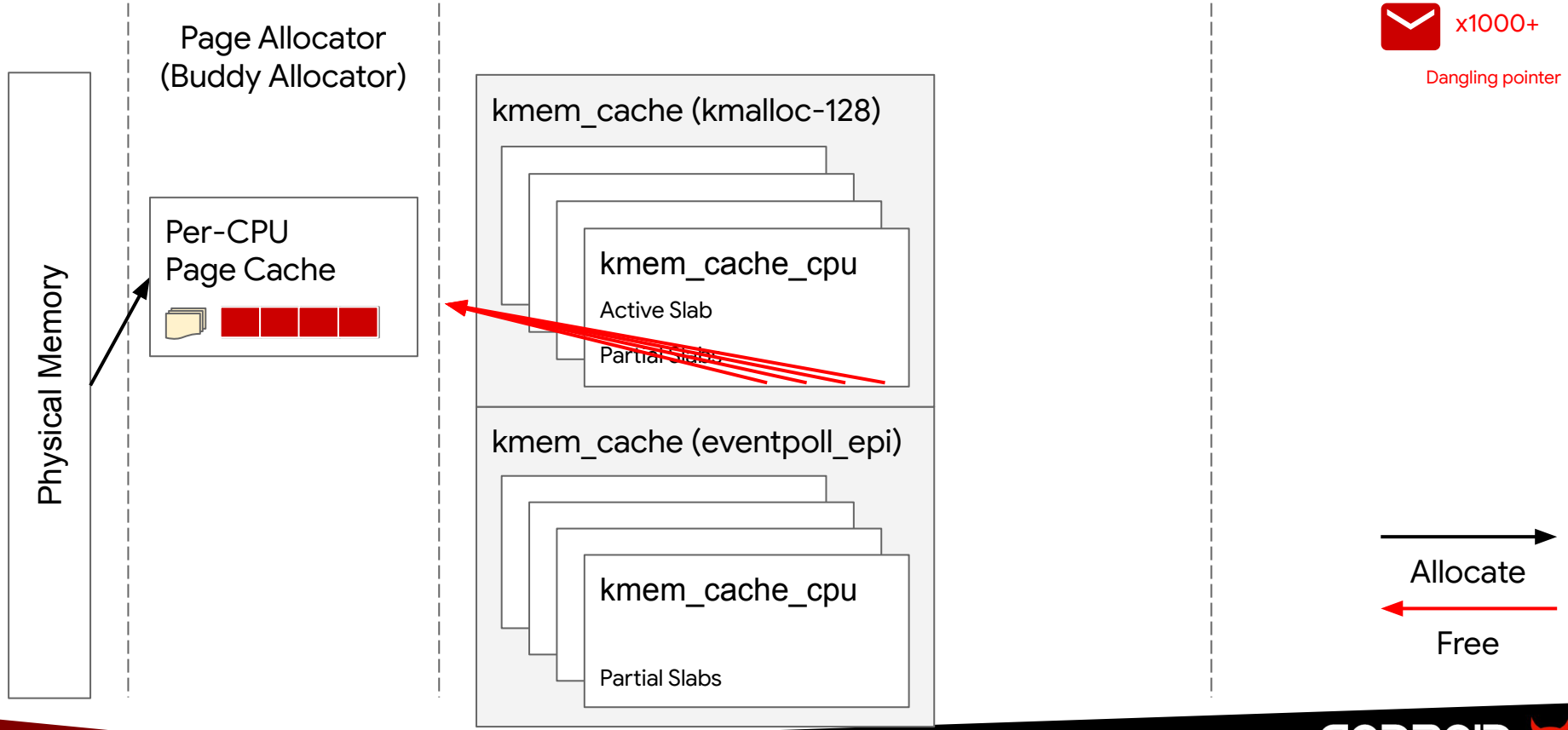
# Cross-cache Attack (SLUB Allocator)



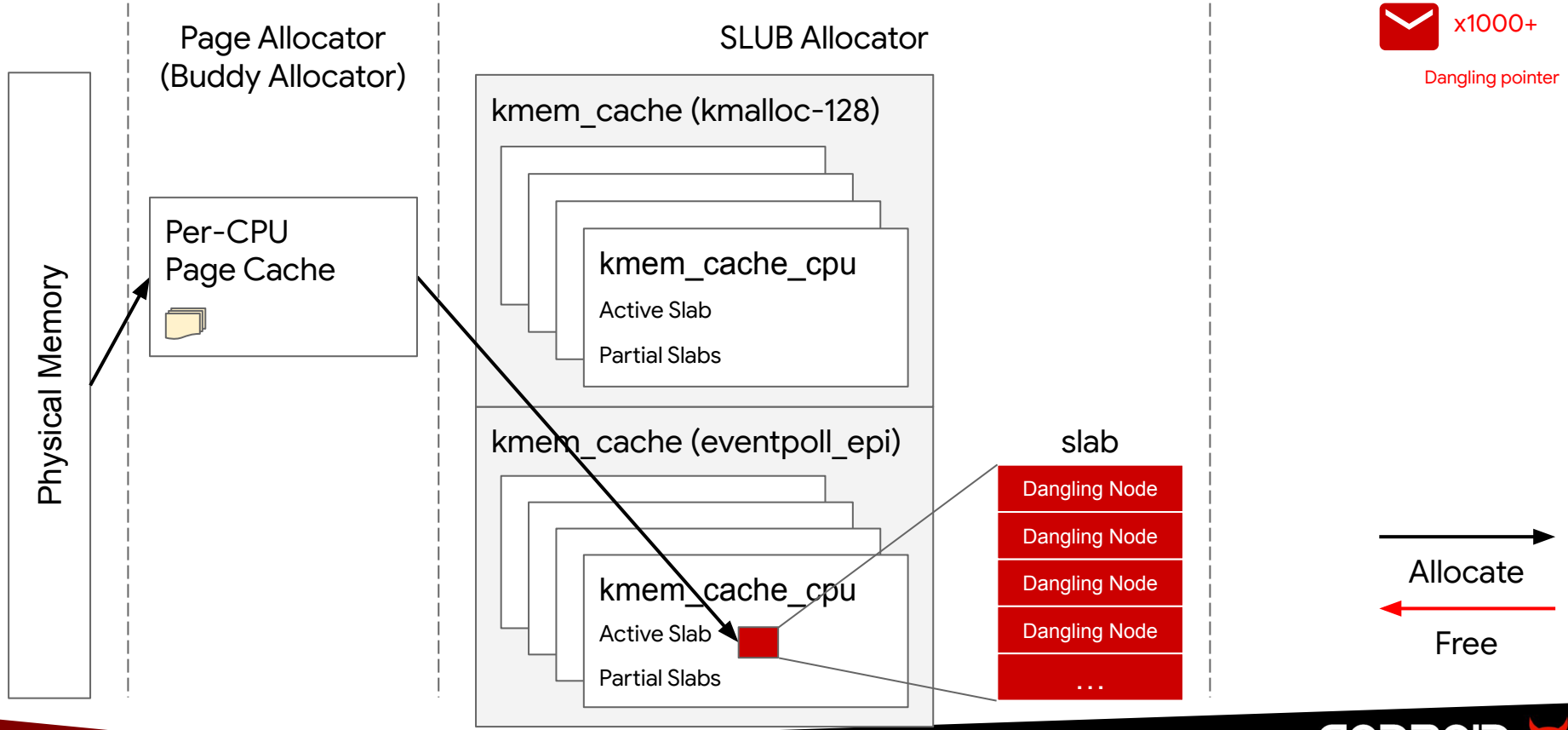
# Cross-cache Attack (SLUB Allocator)



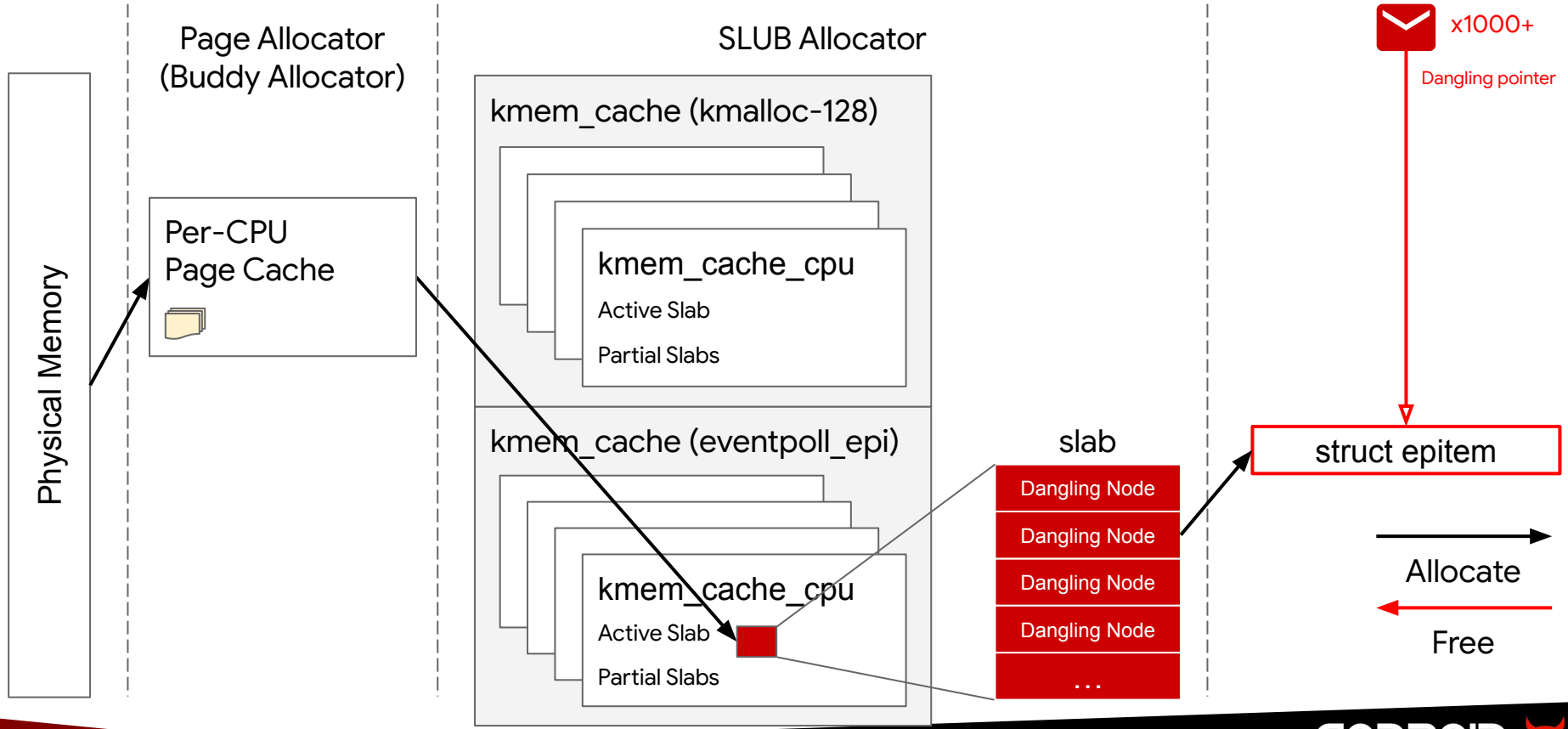
# Cross-cache Attack (SLUB Allocator)



# Cross-cache Attack (SLUB Allocator)



# Cross-cache Attack (SLUB Allocator)

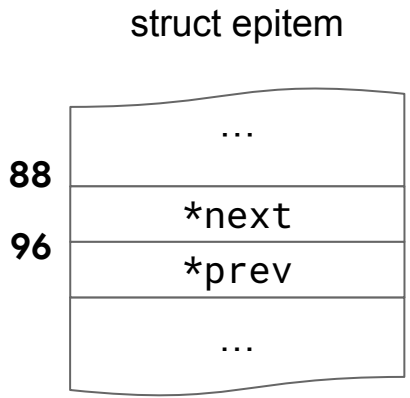


VIDEO DEMONSTRATION OF Cross-cache attack

https://youtu.be/VJWEUsTDtuc

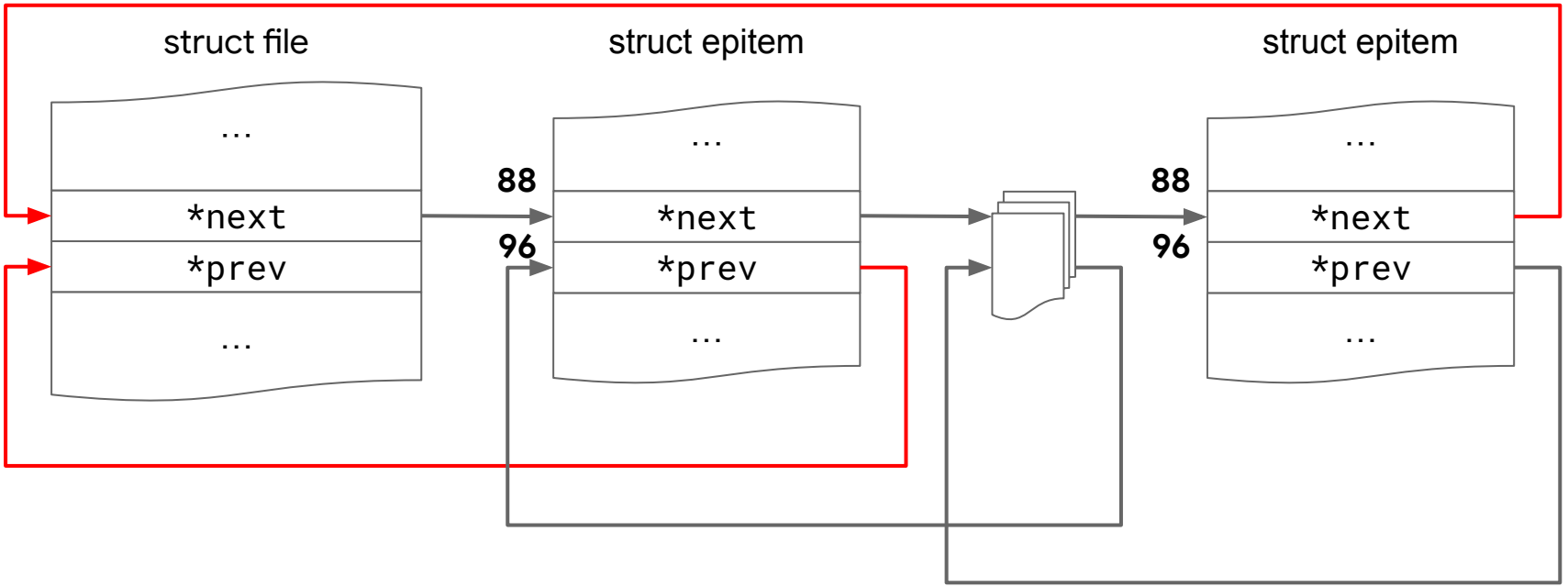
# Arbitrary Read

Use **Leak Primitive** to leak a pointer to a `struct file`



# Arbitrary Read

Use **Leak Primitive** to leak a pointer to a `struct file`



# Arbitrary Read

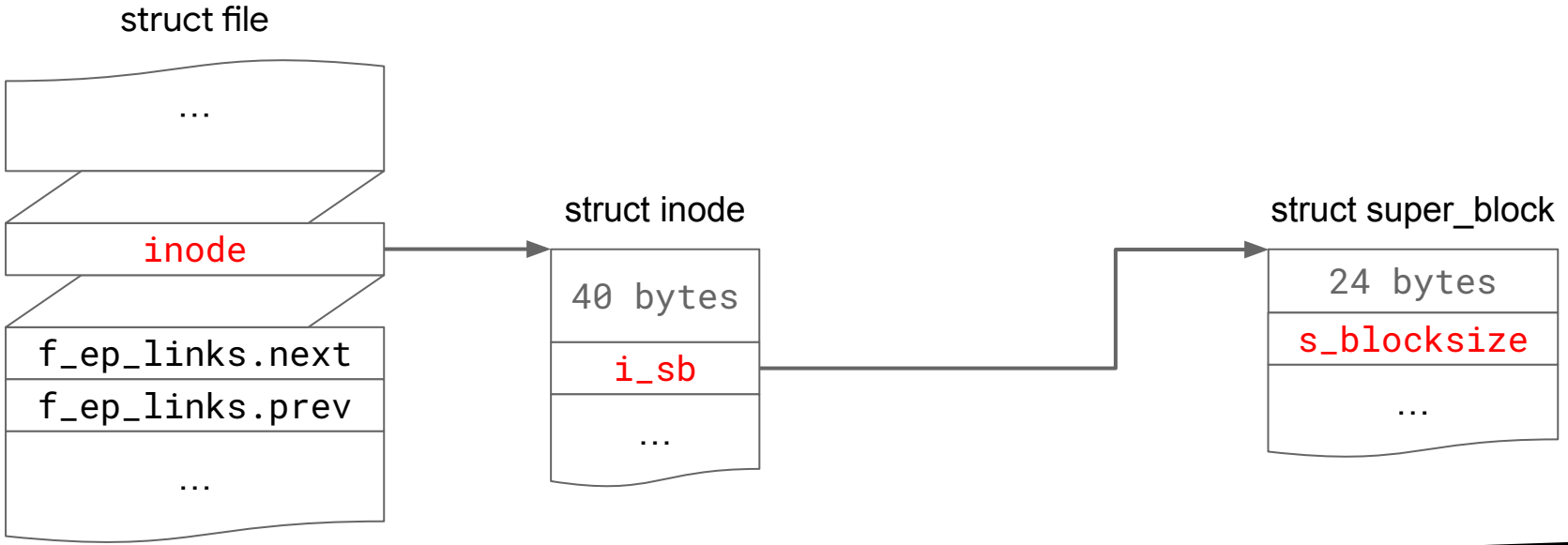
```
ioctl(fd, FIGETBSZ, &value); // &value == argp
```

```
static int do_vfs_ioctl(struct file *filp, ...) {
...
  struct inode *inode = file_inode(filp)
...
    case FIGETBSZ:
...
      return put_user(inode->i_sb->s_blocksize, (int __user *)argp);
```

# Arbitrary Read

```
ioctl(fd, FIGETBSZ, &value); // &value == argp
```

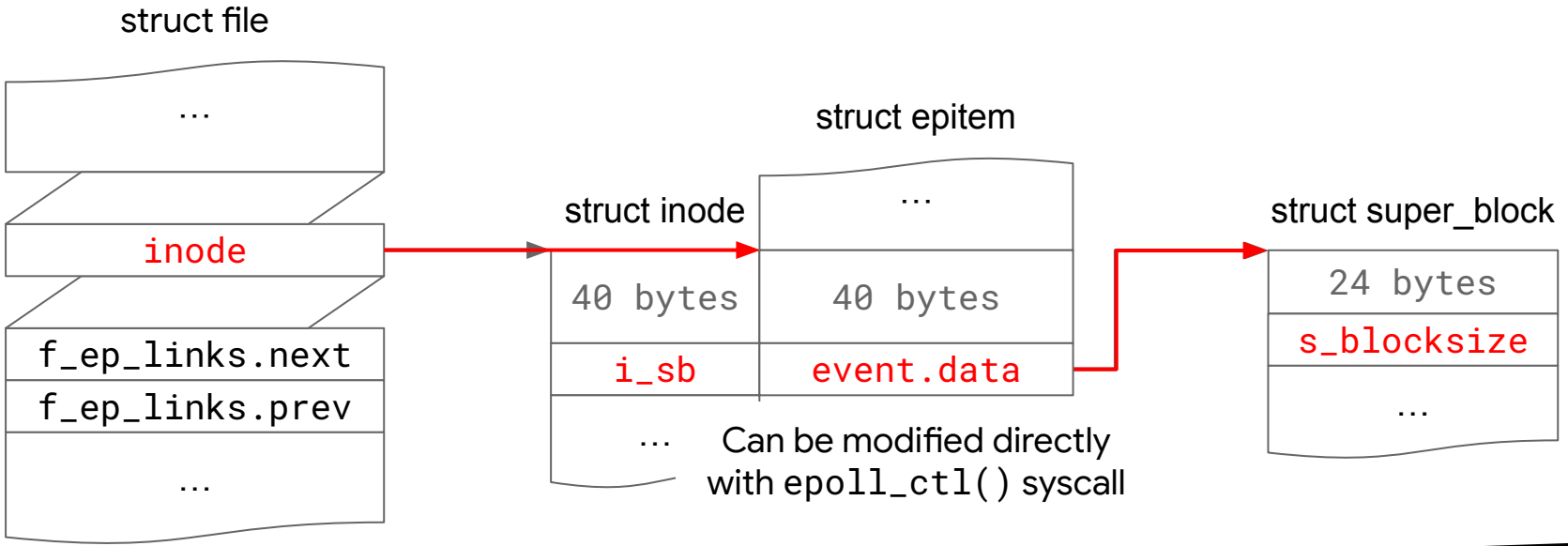
```
return put_user(inode->i_sb->s_blocksize, (int __user *)argp);
```



# Arbitrary Read

```
ioctl(fd, FIGETBSZ, &value); // &value == argp
```

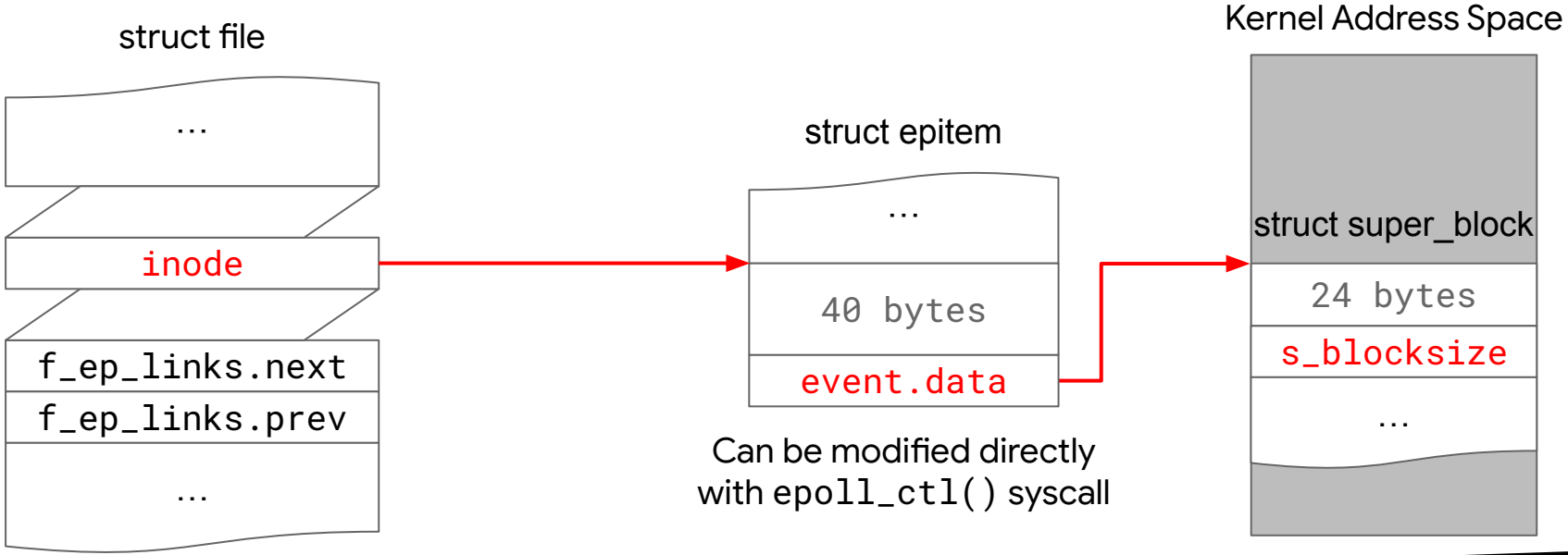
```
return put_user(inode->i_sb->s_blocksize, (int __user *)argp);
```



# Arbitrary Read

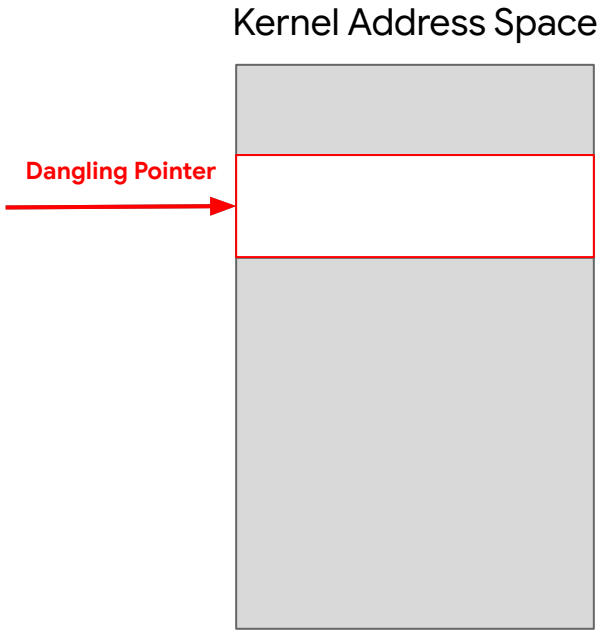
```
ioctl(fd, FIGETBSZ, &value); // &value == argp
```

```
return put_user(inode->i_sb->s_blocksize, (int __user *)argp);
```



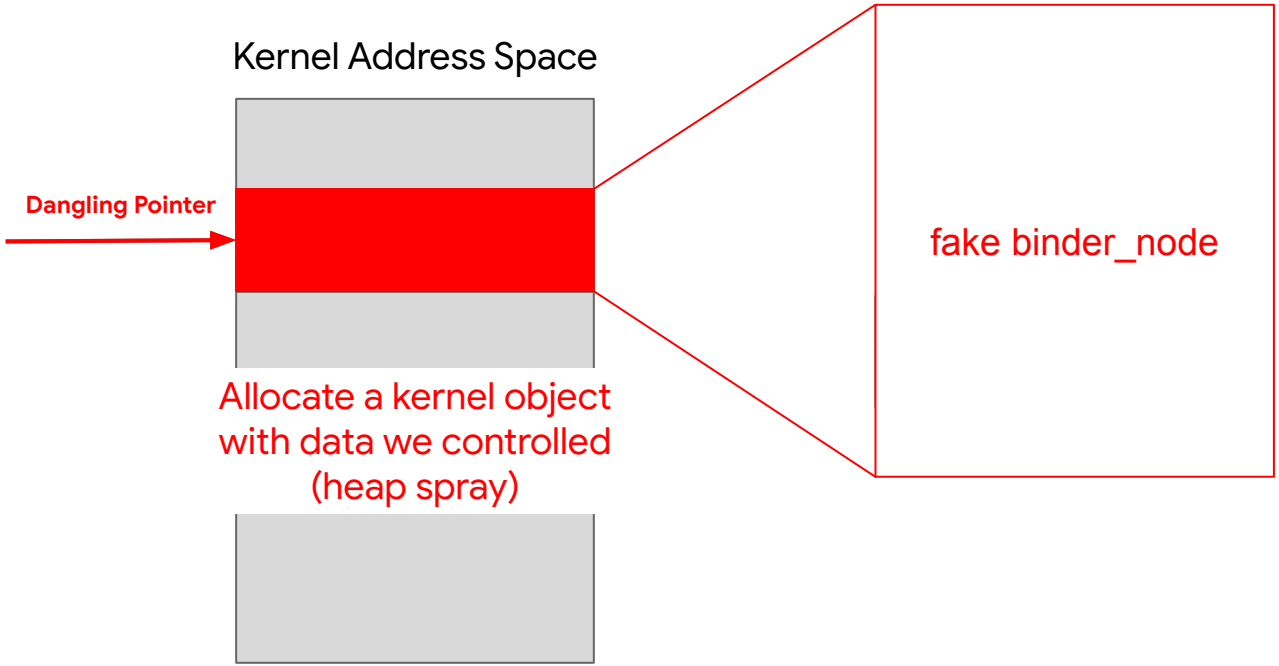
# Unlink primitive

1. Exploit the vulnerability to create a dangling Node.
2. Allocate a fake binder_node with **`sendmsg()`** syscall (heap spray).



# Unlink primitive

1. Exploit the vulnerability to create a dangling Node.
2. Allocate a fake binder_node with **`sendmsg()`** syscall (heap spray).



# Unlink primitive

- Use-after-free when decrementing the refcount of a binder_node

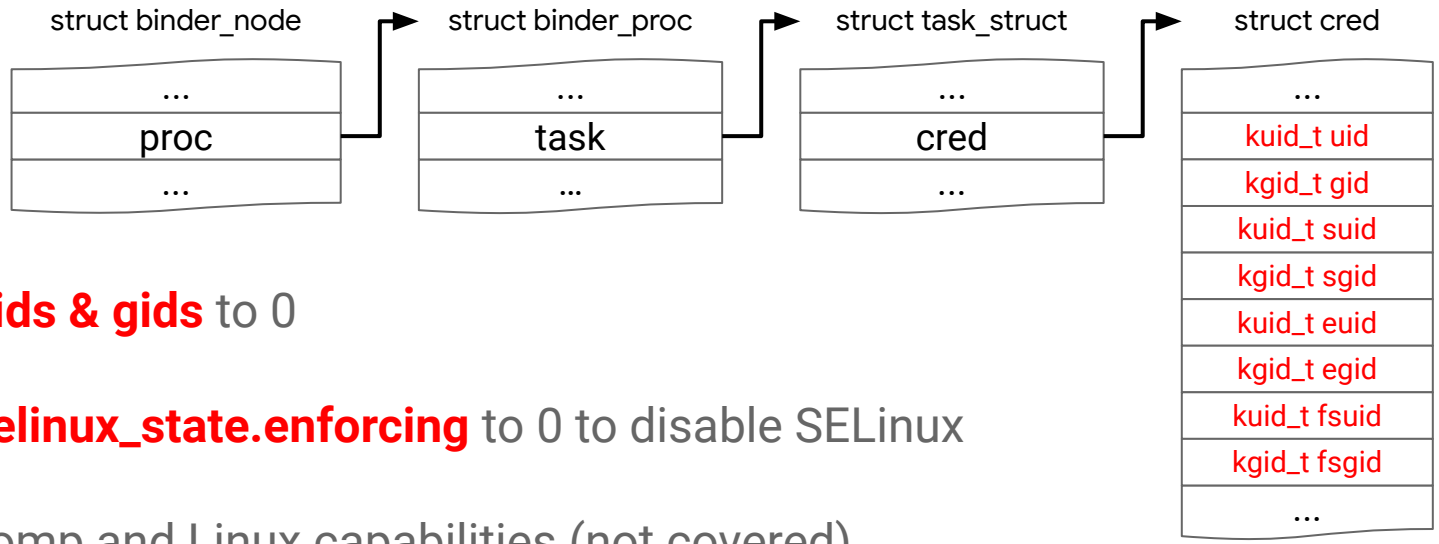
```
static bool binder_dec_node_nilocked(struct binder_node *node, ...) {
...
// If binder_node`s refcount reaches 0
// and satisfy all checks
...
        hlist_del(&node->dead_node);
...
}
```

```
*pprev = next
*(next + 8) = pprev
```

# Root Privilege Escalation

1. Obtain an address to **struct cred**

   a.




2. Set **uids & gids** to 0

3. Set **selinux_state.enforcing** to 0 to disable SELinux

4. Seccomp and Linux capabilities (not covered)

# Demo

Root privilege escalation on Pixel 6 & Pixel 7

VIDEO DEMONSTRATION OF FULL EXPLOIT ON
PIXEL 6 PRO & PIXEL 7 PRO

https://youtu.be/7qFb6RUHnnU

# Fuzzing Binder

## with Linux Kernel Library (LKL)

Google

# Binder Fuzzing

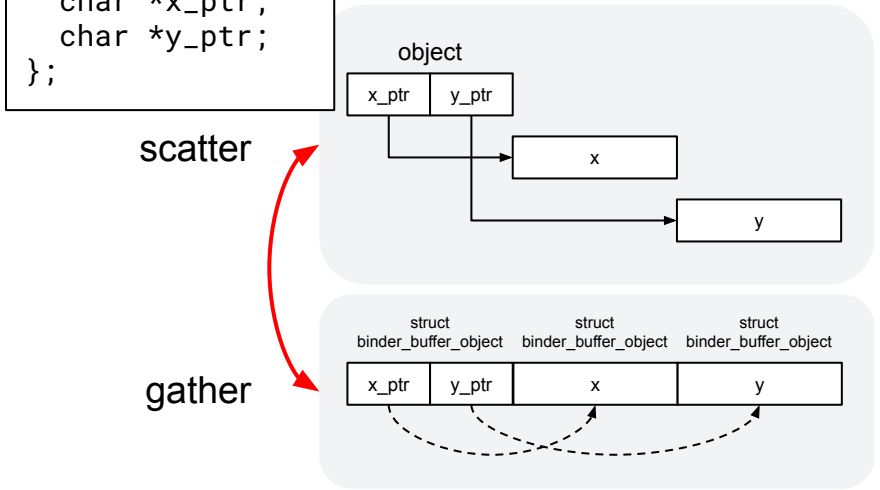
- **Syzkaller** has been fuzzing Binder for years
  - Generate programs based on **syscall descriptions** to be executed on the target machine
  - Discovered CVE-2019-2215 (Bad Binder)
  - ~25% line coverage
- **Challenges**
  - Data dependencies
  - State dependencies
  - Multi-process coordination
  - Reproduce and catch issues involving race conditions

# Binder Fuzzing Challenges

## Data dependencies

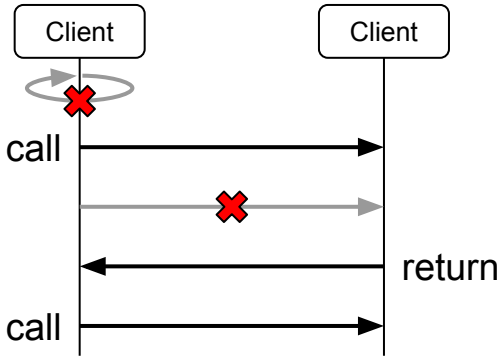
- Binder commands
- Scatter-gather data structures (BINDER_TYPE_PTR)

```
struct object {
  char *x_ptr;
  char *y_ptr;
};
```




## State dependencies

- Synchronous IPC
  - Cannot send transaction to oneself
  - Multiple outstanding transactions not allowed



# Binder Fuzzing Challenges

## State dependencies

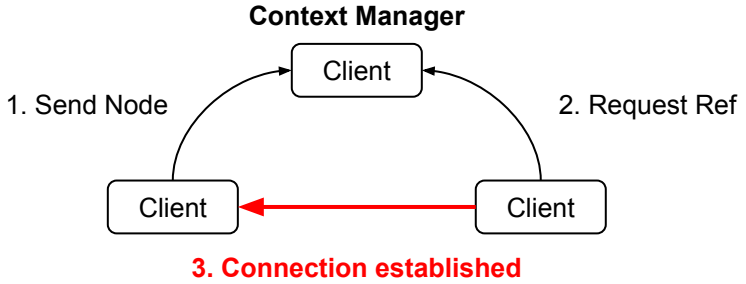
- Some inputs depend on previous IOCTL calls
  - e.g. Transaction buffers (BC_FREE_BUFFER)

```
ioctl(binder_fd, BINDER_WRITE_READ, x) // 1.

// y = x->read_buffer->...->buffer
ioctl(binder_fd, BC_FREE_BUFFER, y)    // 2.
```

Binder memory map

## Multi-process coordination

- All communication requires a **Context Manager**
- Node & Ref setup is required to establish a connection



# LKL Overview

## Linux Kernel Library (LKL)[1] builds Linux kernel as a user-space library

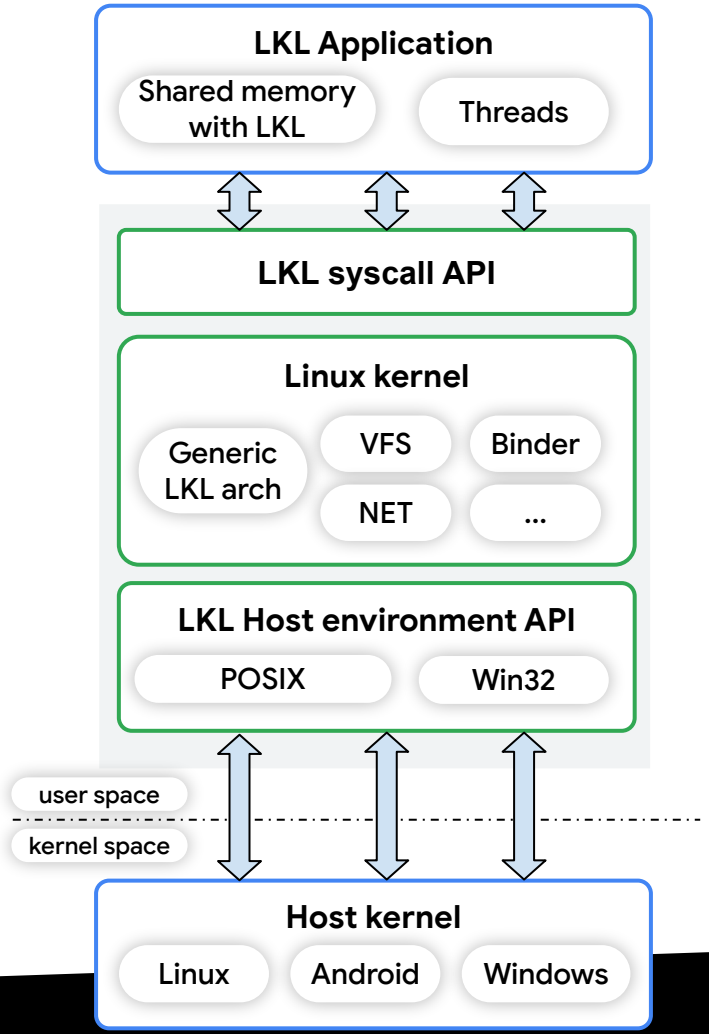
- Implemented as Linux arch-port
- LKL vs UML

## LKL building blocks

- Host environment API -- portability layer
- Linux kernel code
- LKL syscall API exposed to the user-space application

## Run kernel code without launching a VM

- Kernel unit testing
- Fuzzing![2,3]




*[1] https://github.com/lkl/linux*
*[2] Xu et al., Fuzzing File Systems via Two-Dimensional Input Space Exploration*
*[3] https://github.com/atrosinenko/kbdysch*

# Anatomy of LKL fuzzer

## LKL enables fuzzing Linux kernel code in user-space

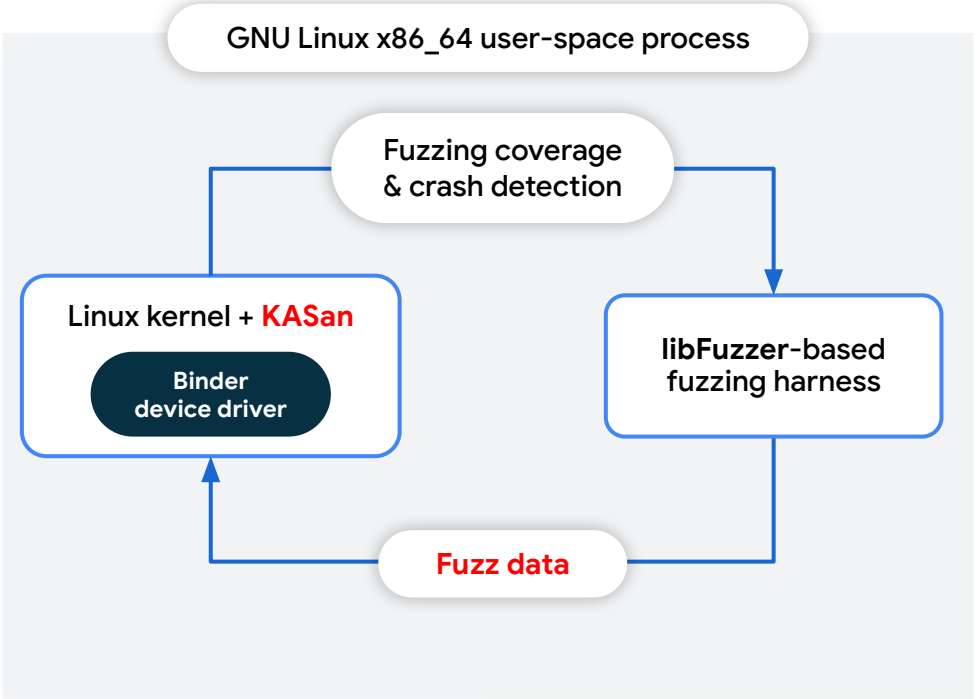
- Use in-process fuzzing engine, such as **libFuzzer**

## Advantages

- High fuzzing performance on x86_64
- Ease of custom modifications
  - e.g. mocking hardware, custom scheduler(?)

## Limitations

- No SMP in LKL
- x86_64 vs aarch64 -- potential false positives, false negatives



# Using LKL from your C program

```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

# Using LKL from your C program

```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

# Using LKL from your C program

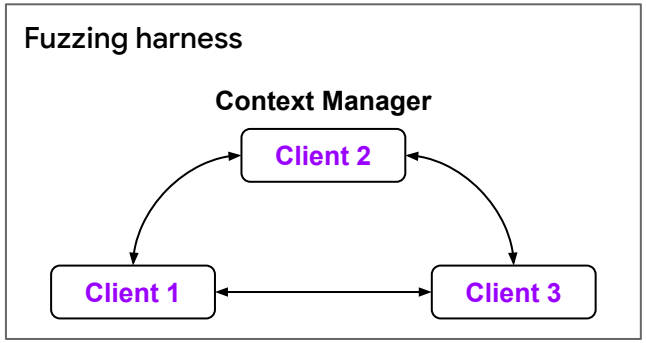
```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

# Using LKL from your C program

```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

# Fuzzing harness

- Simulate IPC interactions between multiple clients
- 3 **clients** (1 Context Manager)
  - **IOCTL calls** and **data**




Fuzz data

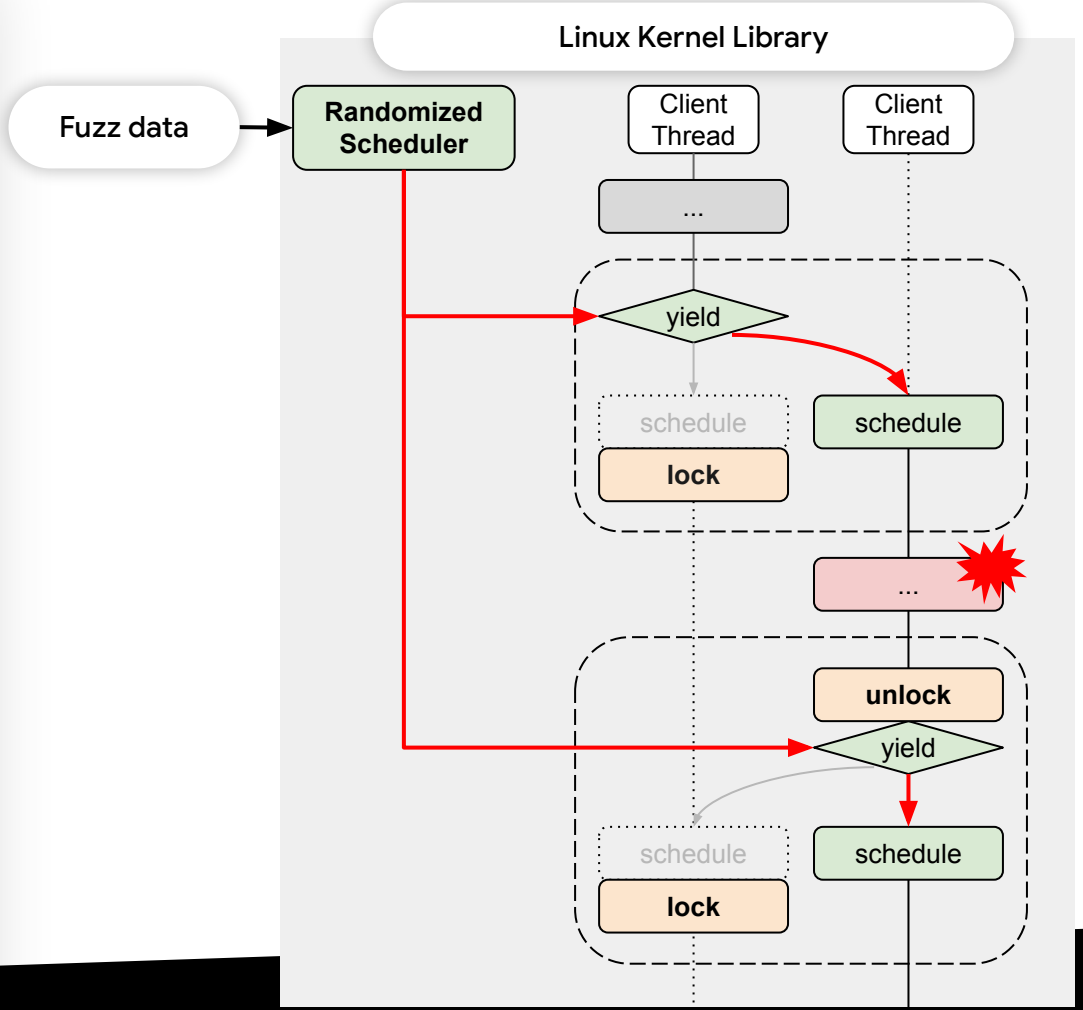
```
client_1 {
  binder_write {
    binder_commands {
      transaction {
        binder_objects { binder { ptr: 0xbeef } }
      }
    }
  }
}
client_2 {
  binder_read { ... }
  binder_write {
    binder_commands { free_buffer { ... } }
  }
}
client_3 { ... }
client_2 { ... }
client_3 { ... }
client_1 { ... }
```

# Randomized Scheduler

**Deterministically** simulate thread interleaving based on fuzz data[1]

Insert yield points before/after synchronization primitives

- spin_lock, spin_unlock
- mutex_lock, mutex_unlock




*[1] Williamson, N., Catch Me If You Can: Deterministic Discovery of Race Conditions with Fuzzing. Black Hat USA, (2022).*

# Results

- Achieved 68% line coverage
- Discovered CVE-2023-20938 & CVE-2023-21255

## Coverage Report

**Created: 2024-05-06 09:49**

Click here for information about interpreting this report.

| Filename | Function Coverage | Line Coverage | Region Coverage | Branch Coverage |
|---|---|---|---|---|
| binder.c | 79.14% (110/139) | 68.55% (3086/4502) | 50.90% (4486/8813) | 50.50% (1317/2608) |
| binder_alloc.c | 78.38% (29/37) | 70.41% (564/801) | 49.87% (752/1508) | 49.20% (185/376) |
| binder_alloc.h | 50.00% (1/2) | 11.11% (1/9) | 50.00% (1/2) | - (0/0) |
| binder_internal.h | 60.00% (3/5) | 68.75% (11/16) | 73.68% (14/19) | - (0/0) |
| **Totals** | **78.14% (143/183)** | **68.73% (3662/5328)** | **50.79% (5253/10342)** | **50.34% (1502/2984)** |

**Generated by llvm-cov -- llvm version 12.0.6git**

# Future work

- Upstream Binder fuzzer to github.com/lkl/linux
- Explore ways to add thread interleaving information into fuzzing engine
- Improve Syzkaller Binder code coverage by tackling those challenges

# Tools

# Tools

**linux-exploit-dev-env**

- gsingh93/linux-exploit-dev-env
- Environment for exploit development on Linux and Android Common Kernel
- QEMU - supports x86_64 and arm64

**pwndbg: slab, pcp, binder plugin**

- slab - inspect SLAB allocator states
- pcp - inspect per-cpu cache (pwndbg/pull/1487)
- binder - inspect Binder states: nodes, refs, transactions and etc. (pwndbg/pull/1488)

**bpftrace scripts**

- Trace SLAB and page allocations for heap grooming and cross-cache attacks
- Keep an eye on https://github.com/androidoffsec for a future release

# Thank You! Questions?

*androidoffsec-external@google.com*

Google