

  
**black hat**<sup>®</sup>  
USA 2022

AUGUST 10-11, 2022

---

BRIEFINGS

# Cautious! A New Exploitation Method! No Pipe but as Nasty as Dirty Pipe

Zhenpeng Lin, Yuhang Wu, Xinyu Xing

Northwestern University



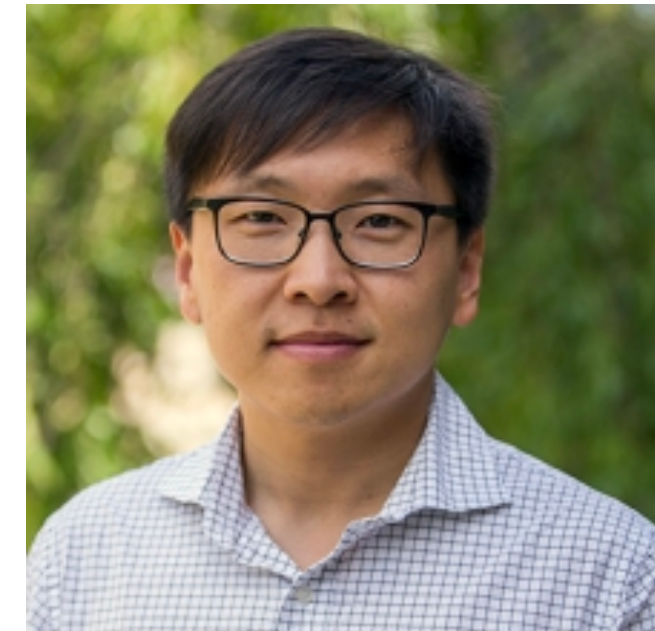
# Who Are We



**Zhenpeng Lin**  
PhD Student  
[zplin.me](http://zplin.me)



**Yuhang Wu**  
PhD Student  
[yuhangw.blog](http://yuhangw.blog)



**Xinyu Xing**  
Associate Professor  
[xinyuxing.org](http://xinyuxing.org)

# Recap About Dirty Pipe

- CVE-2022-0847
- An uninitialized bug in Linux kernel's pipe subsystem
- Affected kernel v5.8 and higher
- Data-only, no effective exploitation mitigation
- Overwrite any files with read permission
- Demonstrated LPE on Android



# What We Learned

- **Data-only is powerful**
  - Universal exploit
  - Bypass CFI (enabled in Android kernel)
  - New mitigation required



# What We Learned

- **Data-only is powerful**
  - Universal exploit
  - Bypass CFI (enabled in Android kernel)
  - New mitigation required
- **Dirty Pipe is not perfect**
  - Cannot actively escape from container
  - Not a generic exploitation method

# Introducing DirtyCred

- **High-level idea**
  - **Swapping Linux kernel *Credentials***
- **Advantages**
  - A generic exploitation method, simple and effective
  - Write a data-only, universal (i.e., Dirty-Pipe-liked) exploit
  - Actively escape from container



# Comparison with Dirty Pipe

	Dirty Pipe	DirtyCred
• A generic exploitation method?	✗	✓
• Write a data-only, universal exploit?	✓	✓
• Attack with CFI enabled (on Android)?	✓	✓
• Actively escape from container?	✗	✓
• Threat still exists?	✗	✓

# Kernel Credential

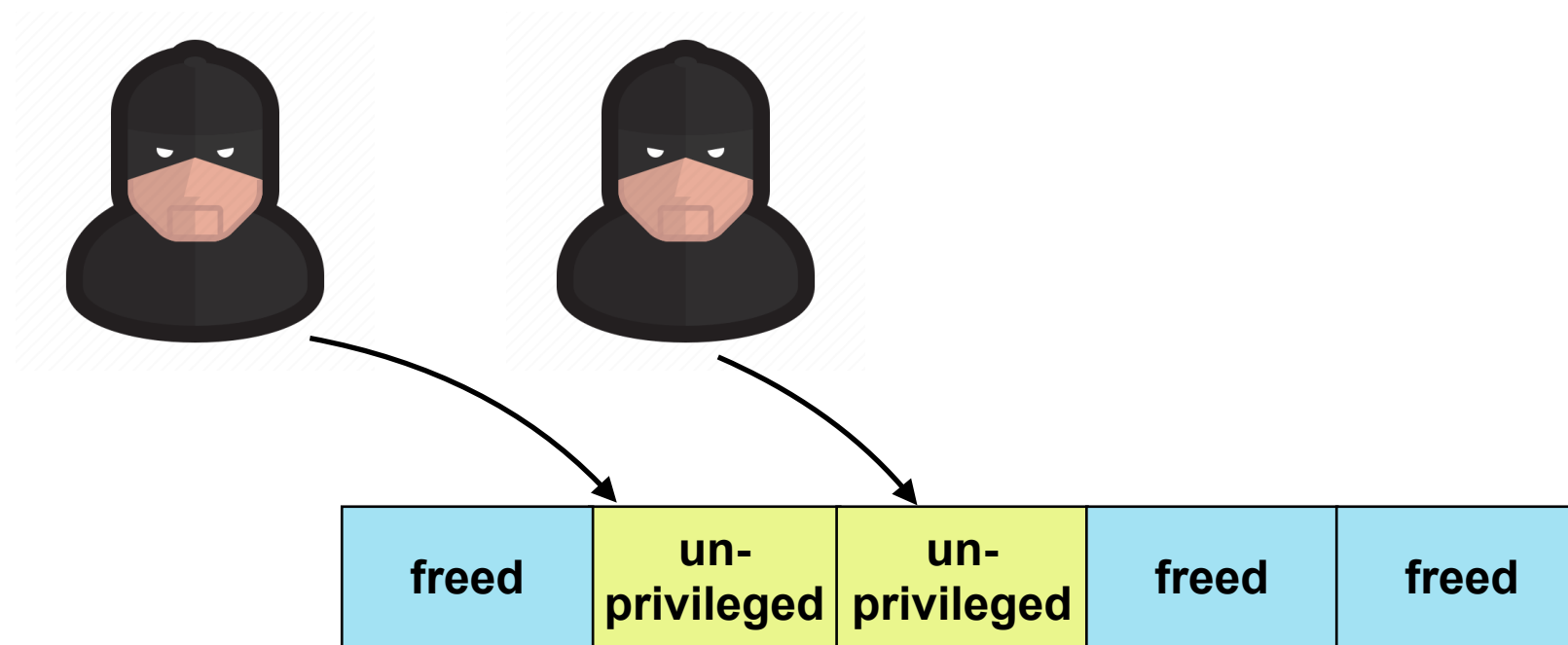
- **Properties that carry privilege information in kernel**
  - Defined in kernel documentation
  - Representation of **privilege** and **capability**
  - Two main types: *task credentials* and *open file credentials*
  - Security checks act on credential objects

Source: <https://www.kernel.org/doc/Documentation/security/credentials.txt>



# Task Credential

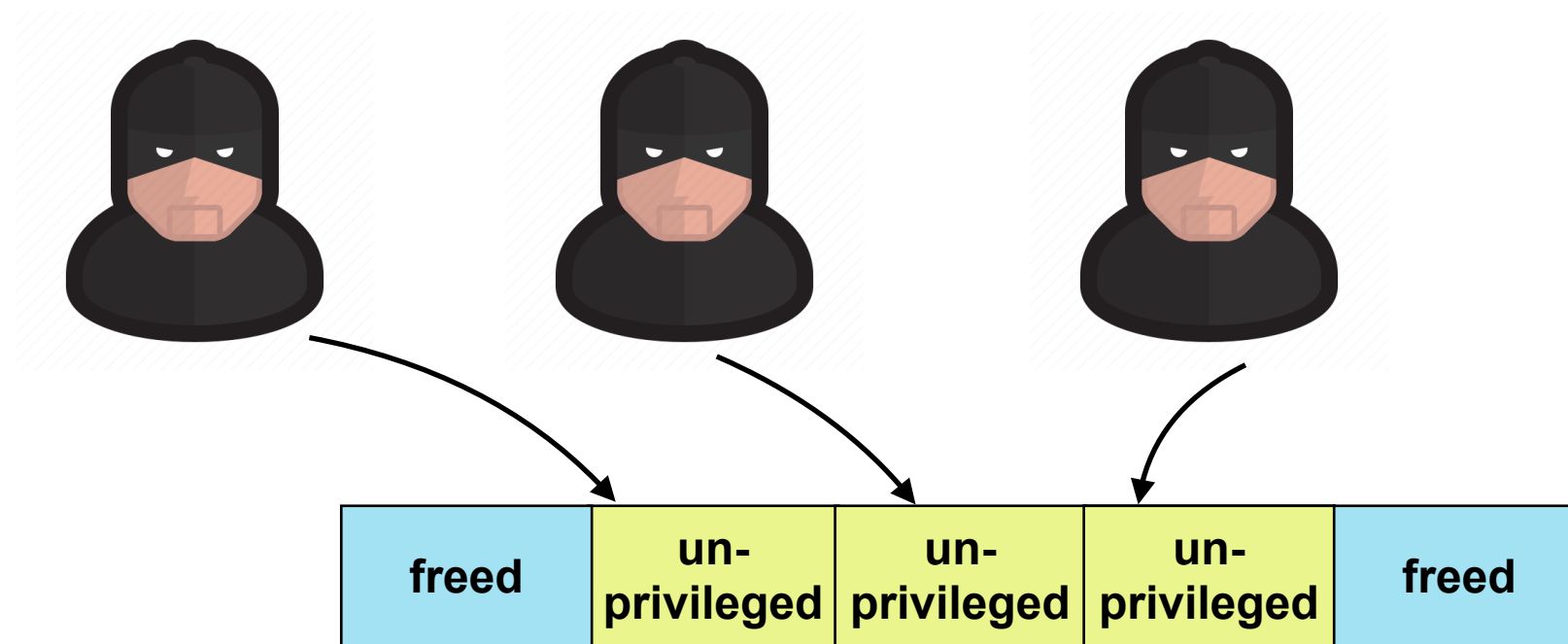
- `struct cred` in kernel's implementation



`struct cred` on kernel heap

# Task Credential

- `struct cred` in kernel's implementation

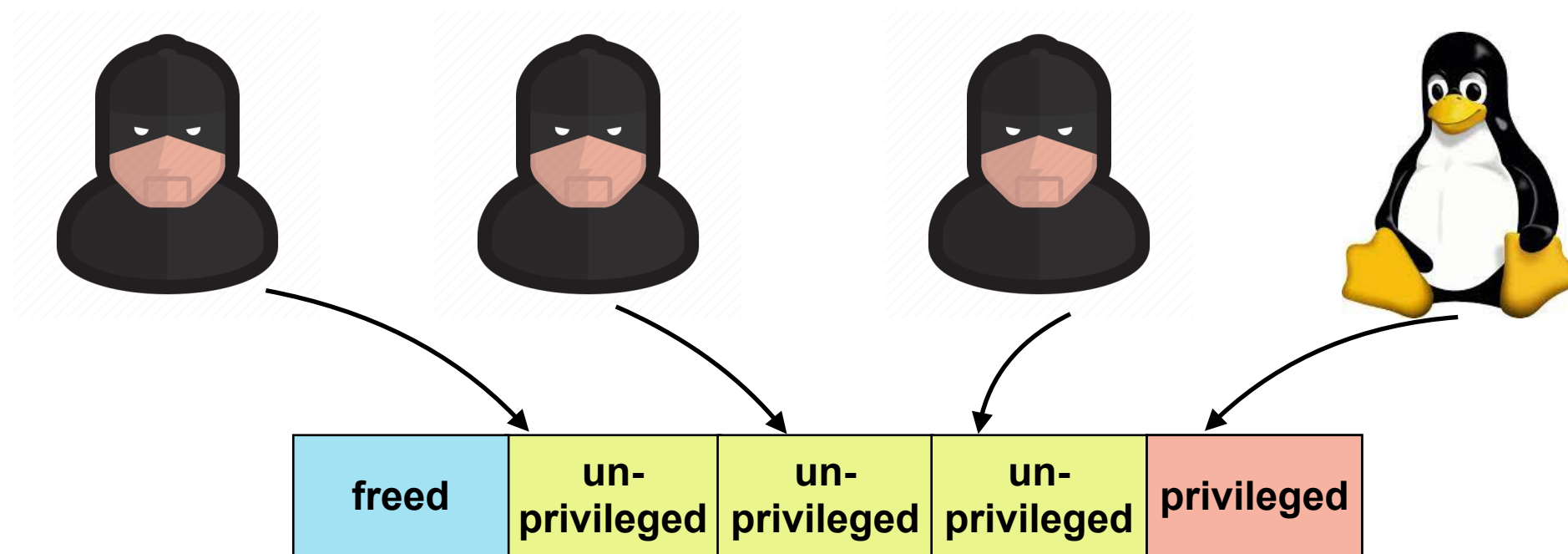


`struct cred` on kernel heap



# Task Credential

- Struct cred in kernel's implementation



struct cred on kernel heap

# Open File Credentials

- `struct file` in kernel's implementation

Freed		
	f_op	f_op
	f_mode O_RDWR	f_mode O_RDWR
	f_cred	f_cred
	~/dummy	~/dummy

`struct file` on kernel heap

# Open File Credentials

- **Struct file** in kernel's implementation

```
int fd = open("~/dummy", O_RDWR);
```



f_op	f_op	f_op
f_mode O_RDWR	f_mode O_RDWR	f_mode O_RDWR
f_cred	f_cred	f_cred
~/dummy	~/dummy	~/dummy

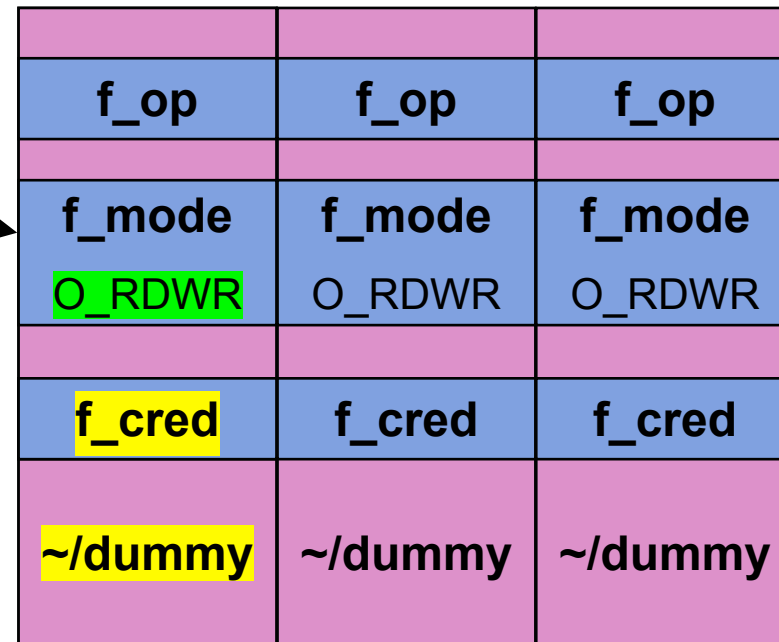
**struct file** on kernel heap



# Open File Credentials

- Struct `file` in kernel's implementation

```
int fd = open("~/dummy", O_RDWR);
```



The diagram illustrates the layout of the `struct file` in kernel memory. It is a 3x3 grid of cells. The top row contains three cells, each labeled `f_op`. The middle row contains three cells, each labeled `f_mode`. The bottom row contains three cells, each labeled `f_cred`. The first cell in the first row is highlighted in blue. The first cell in the second row is highlighted in green, and an arrow points from the `O_RDWR` in the code above to this cell. The first cell in the third row is highlighted in yellow. The first cell in the fourth row is highlighted in yellow and contains the text `~/dummy`.

<code>f_op</code>	<code>f_op</code>	<code>f_op</code>
<code>f_mode</code> <code>O_RDWR</code>	<code>f_mode</code> <code>O_RDWR</code>	<code>f_mode</code> <code>O_RDWR</code>
<code>f_cred</code>	<code>f_cred</code>	<code>f_cred</code>
<code>~/dummy</code>	<code>~/dummy</code>	<code>~/dummy</code>

struct `file` on kernel heap

# Open File Credentials

- Kernel checks permission on the `file` object when accessing

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm

f_op	f_op	f_op
f_mode	f_mode	f_mode
O_RDWR	O_RDWR	O_RDWR
f_cred	f_cred	f_cred
~/dummy	~/dummy	~/dummy

struct file on kernel heap

# Open File Credentials

- Write content to file on disk if permission is granted

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm



Write to disk

f_op	f_op	f_op
f_mode	f_mode	f_mode
O_RDWR	O_RDWR	O_RDWR
f_cred	f_cred	f_cred
~/dummy	~/dummy	~/dummy

struct file on kernel heap



# Open File Credentials

- Write *denied* if the file is opened *read-only*

```
int fd = open("~/dummy", O_RDONLY);
```

```
write(fd, "HACKED", 6);
```

check perm

f_op	f_op	f_op
f_mode	f_mode	f_mode
O_RDONLY	O_RDWR	O_RDWR
f_cred	f_cred	f_cred
~/dummy	~/dummy	~/dummy

struct file on kernel heap

# Open File Credentials

- Write *denied* if the file is opened *read-only*

```
int fd = open("~/dummy", O_RDONLY);
```

```
write(fd, "HACKED", 6);
```

check perm



Failed write to  
disk

f_op	f_op	f_op
f_mode	f_mode	f_mode
O_RDONLY	O_RDWR	O_RDWR
f_cred	f_cred	f_cred
~/dummy	~/dummy	~/dummy

struct file on kernel heap

# DirtyCred: Swapping Linux Kernel Credentials

## High-level idea

- Swapping *unprivileged* credentials with *privileged* ones

## Two-Path attacks

- Attacking *task credentials* (`struct cred`)
- Attacking *open file credentials* (`struct file`)

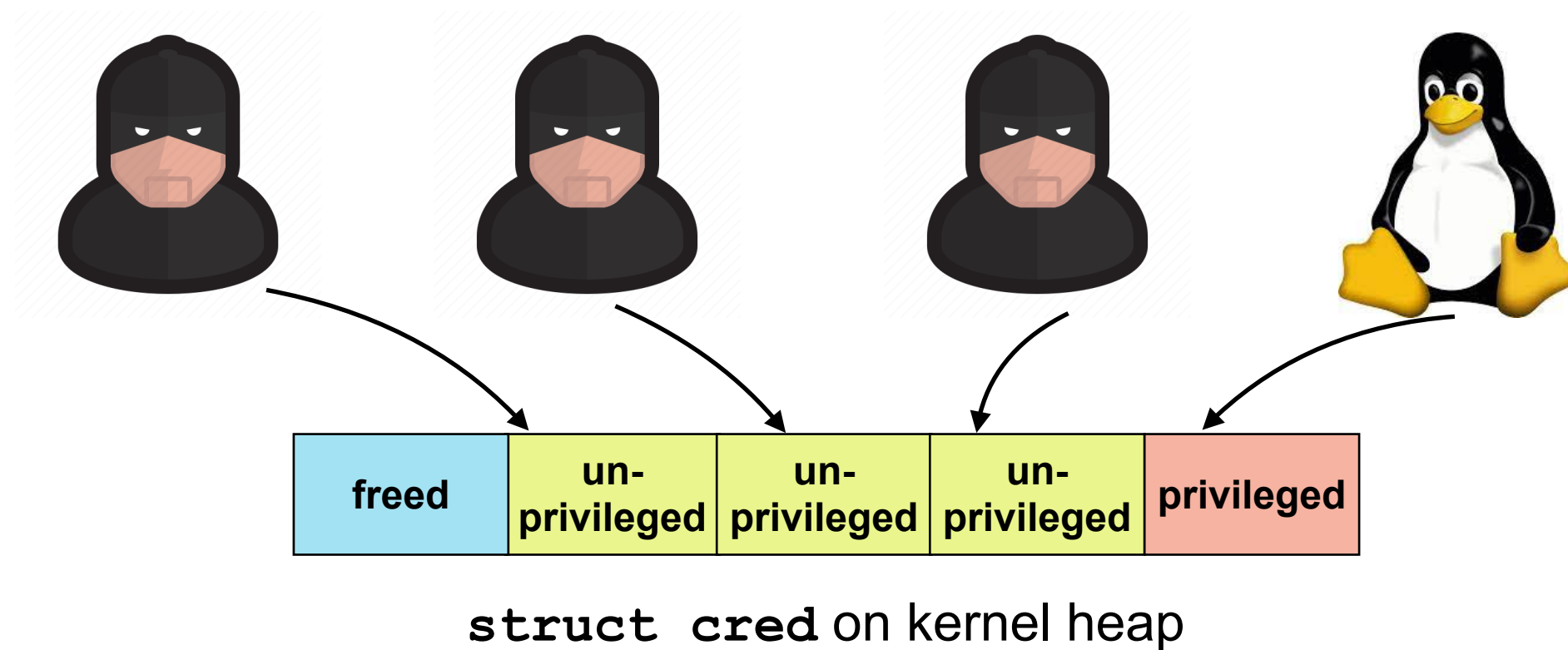


# DirtyCred: Swapping Linux Kernel Credentials

## Two-Path attacks

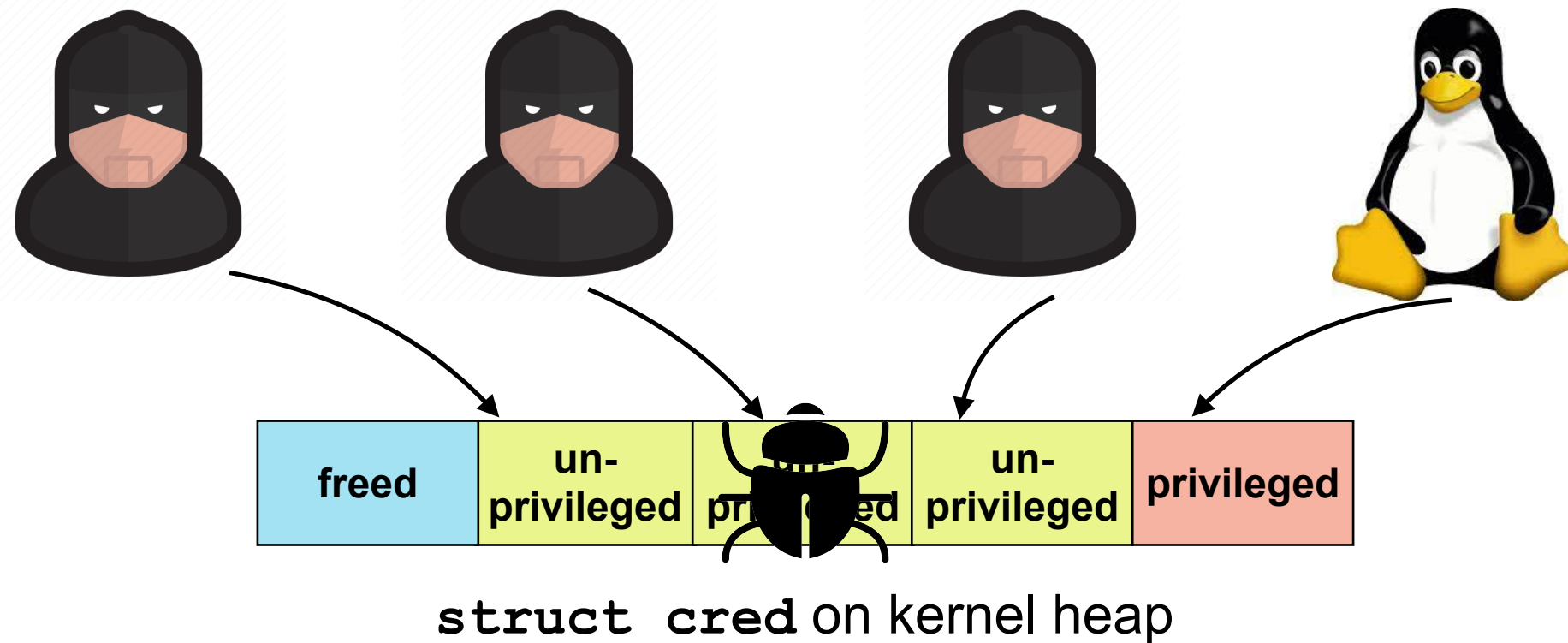
- Attacking *task credentials* (`struct cred`)
- Attacking *open file credentials* (`struct file`)

# Attacking Task Credentials



# Attacking Task Credentials

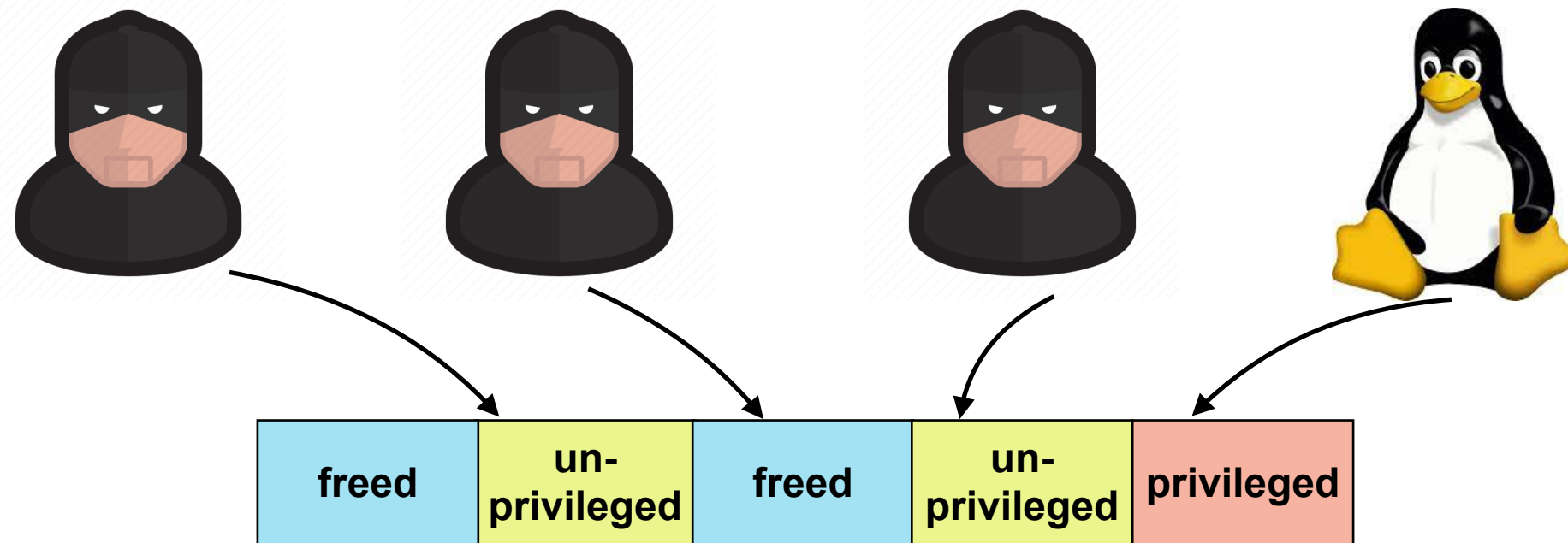
Step 1. **Free** a *unprivileged* credential with the vulnerability





# Attacking Task Credentials

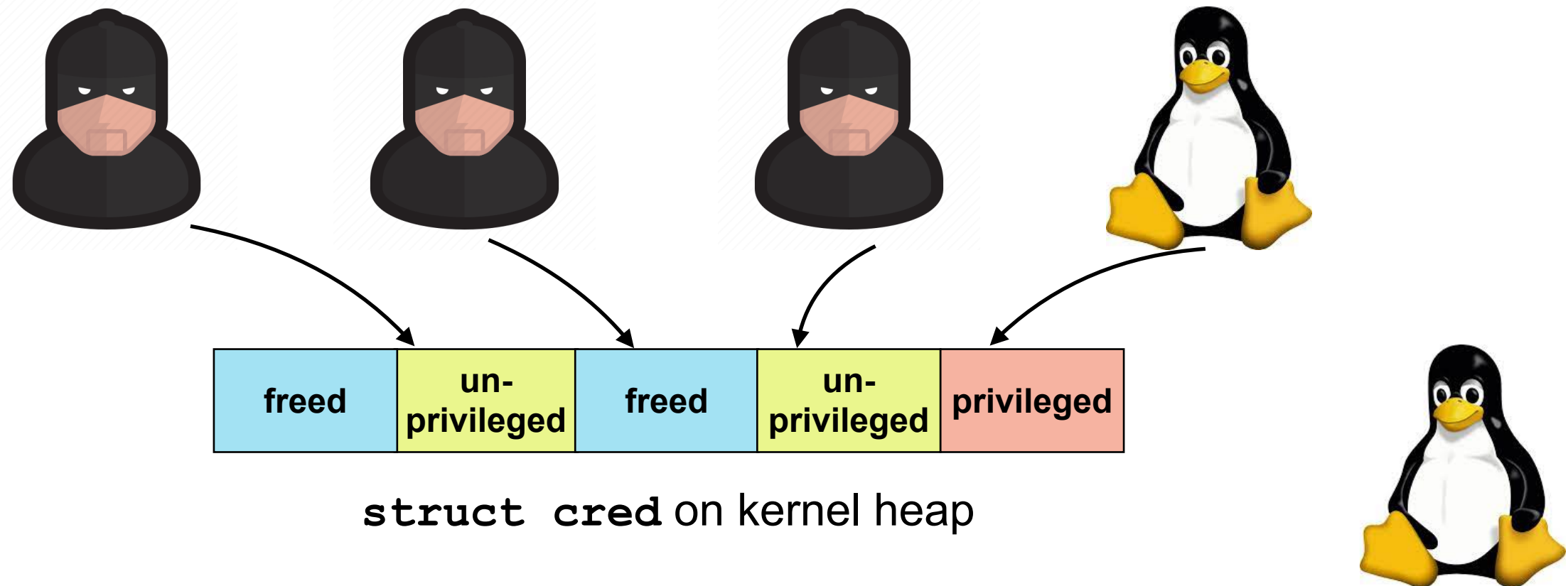
Step 1. **Free** a *unprivileged* credential with the vulnerability



`struct cred` on kernel heap

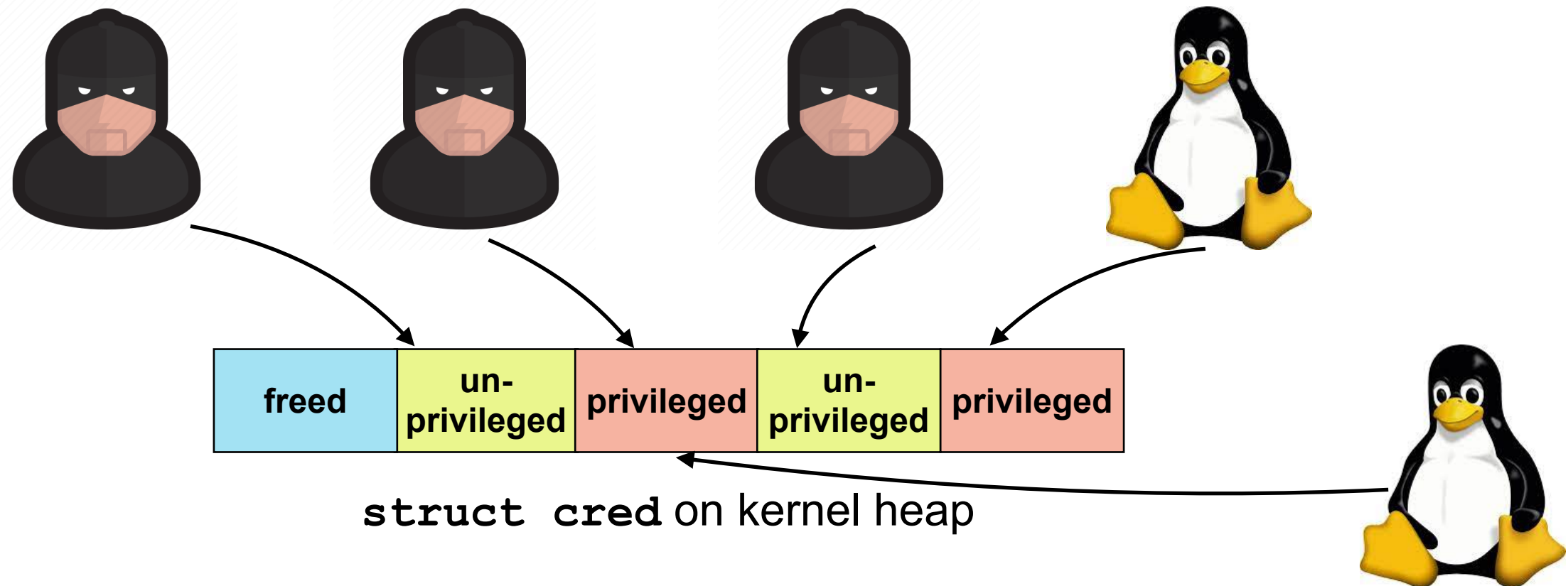
# Attacking Task Credentials

Step 2. Allocate *privileged* credentials in the *freed* memory slot



# Attacking Task Credentials

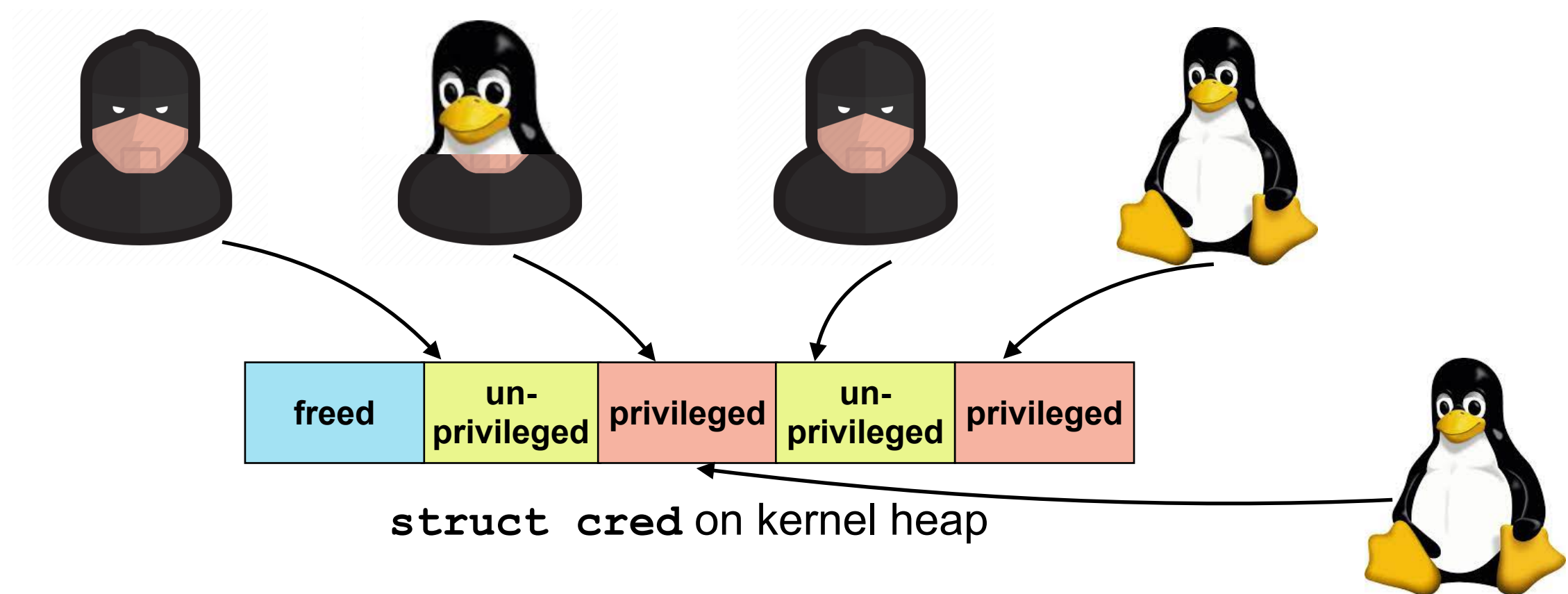
Step 2. Allocate *privileged* credentials in the *freed* memory slot





# Attacking Task Credentials

Step 3. Operate as *privileged* user



# DirtyCred: Swapping Linux Kernel Credentials

## Two-Path attacks

- Attacking *task credentials* (`struct cred`)
- Attacking *open file credentials* (`struct file`)

# Attacking Open File Credentials

- Write content to file on disk if permission is granted

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm

f_op	f_op	f_op
f_mode	f_mode	f_mode
O_RDWR	O_RDWR	O_RDWR
f_cred	f_cred	f_cred
~/dummy	~/dummy	~/dummy

struct file on kernel heap



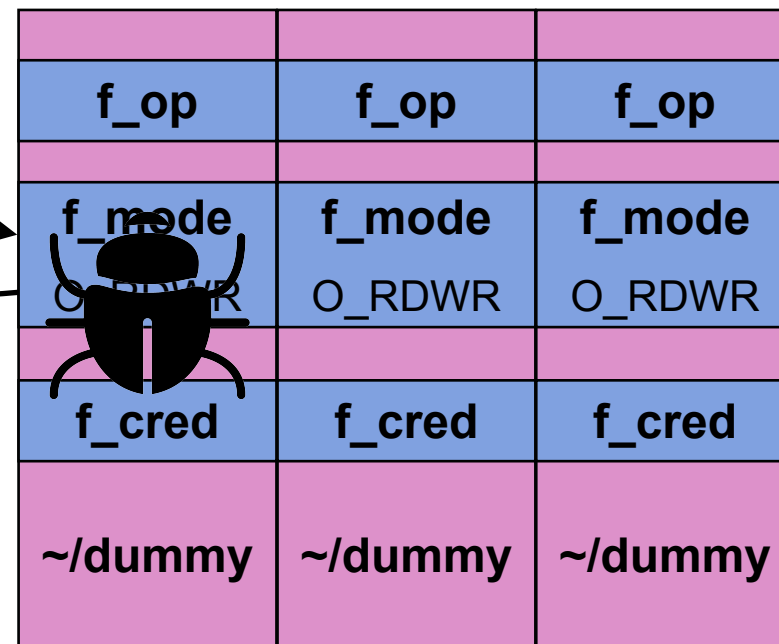
# Attacking Open File Credentials

Step 1. **Free** file obj *after* checks, but *before* writing to disk

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm



struct file on kernel heap

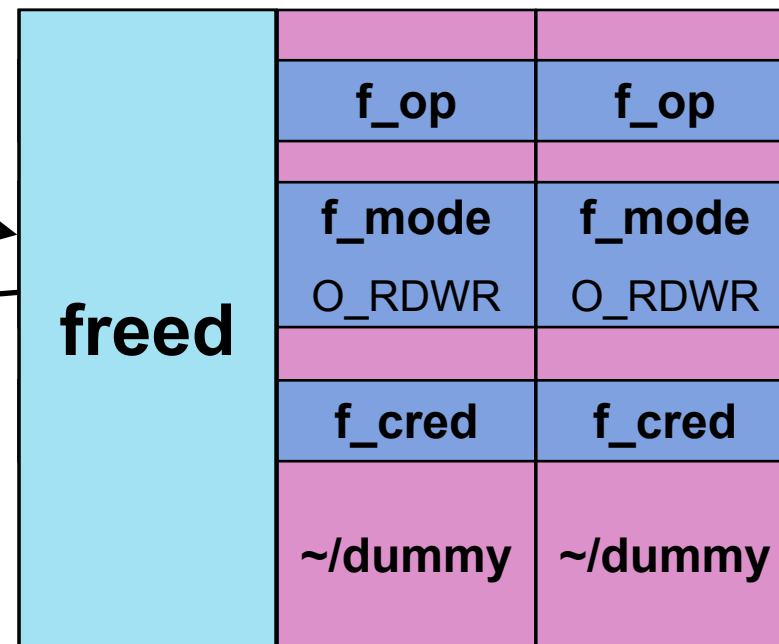
# Attacking Open File Credentials

Step 1. **Free** file obj *after* checks, but *before* writing to disk

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm



struct file on kernel heap

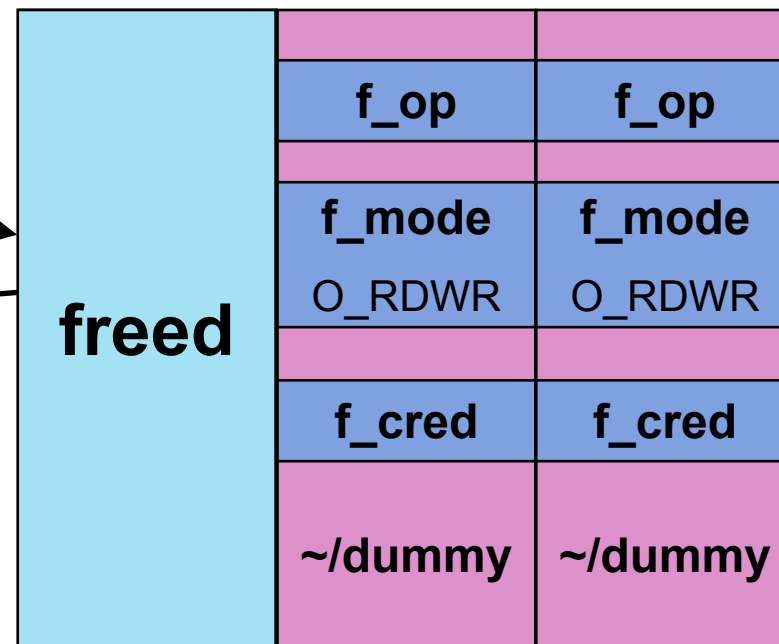
# Attacking Open File Credentials

Step 2. Allocate a *read-only* file obj in the **freed** memory slot

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm



```
open("/etc/passwd", O_RDONLY);
```

struct file on kernel heap



# Attacking Open File Credentials

Step 2. Allocate a *read-only* file obj in the freed memory slot

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm

f_op	f_op	f_op
f_mode	f_mode	f_mode
O_RDONLY	O_RDWR	O_RDWR
f_cred	f_cred	f_cred
<u>/etc/passwd</u>	~/dummy	~/dummy

```
open("/etc/passwd", O_RDONLY);
```

struct file on kernel heap

# Attacking Open File Credentials

Step 3. Operate as *privileged* user — Writing content to the file

```
int fd = open("~/dummy", O_RDWR);
```

```
write(fd, "HACKED", 6);
```

check perm



Write to /etc/passwd on disk

```
open("/etc/passwd", O_RDONLY);
```

f_op	f_op	f_op
f_mode	f_mode	f_mode
O_RDONLY	O_RDWR	O_RDWR
f_cred	f_cred	f_cred
<u>/etc/passwd</u>	~/dummy	~/dummy

struct file on kernel heap

# DirtyCred: Swapping Linux Kernel Credentials

## Three Steps:

1. **Free** an inuse *unprivileged* credential with the vulnerability
2. **Allocate** *privileged* credentials in the **freed** memory slot
3. **Operate** as *privileged* user



# Three Challenges

1. How to **free** credentials.
2. How to **allocate** *privileged* credentials as *unprivileged* users.  
(attacking *task* credentials)
3. How to **stabilize** file exploitation. (attacking *open file* credentials)

# Challenge 1: Free Credentials

- Both *cred* and *file* object are in **dedicated** caches
- Most vulnerabilities happens in **generic** caches
- Most vulnerabilities may not have free capability

# Challenge 1: Free In-use Credentials Invalidly

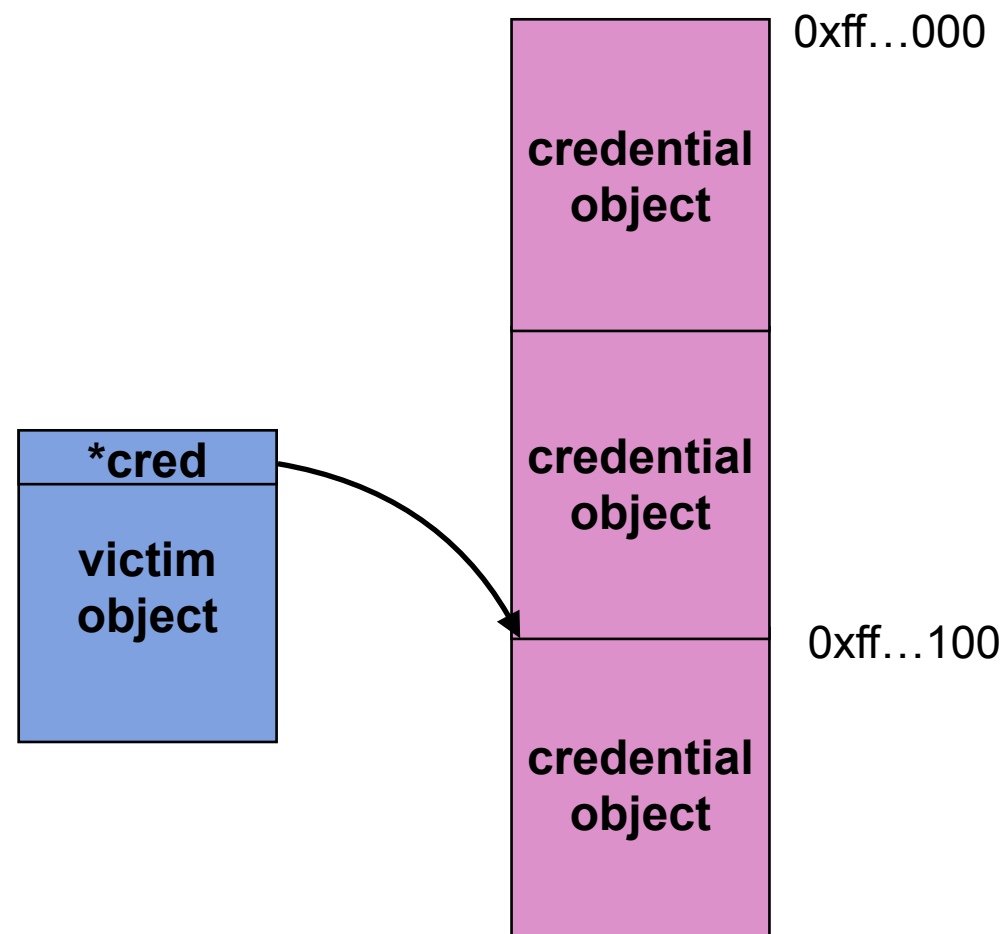
- **Solution: Pivoting Vulnerability Capability**
  - Pivoting Invalid-Write (e.g., OOB & UAF write)
  - Pivoting Invalid-Free (e.g., Double-Free)



# Pivoting Invalid-Write

# Pivoting Invalid-Write

- Leverage victim objects with a reference to credentials



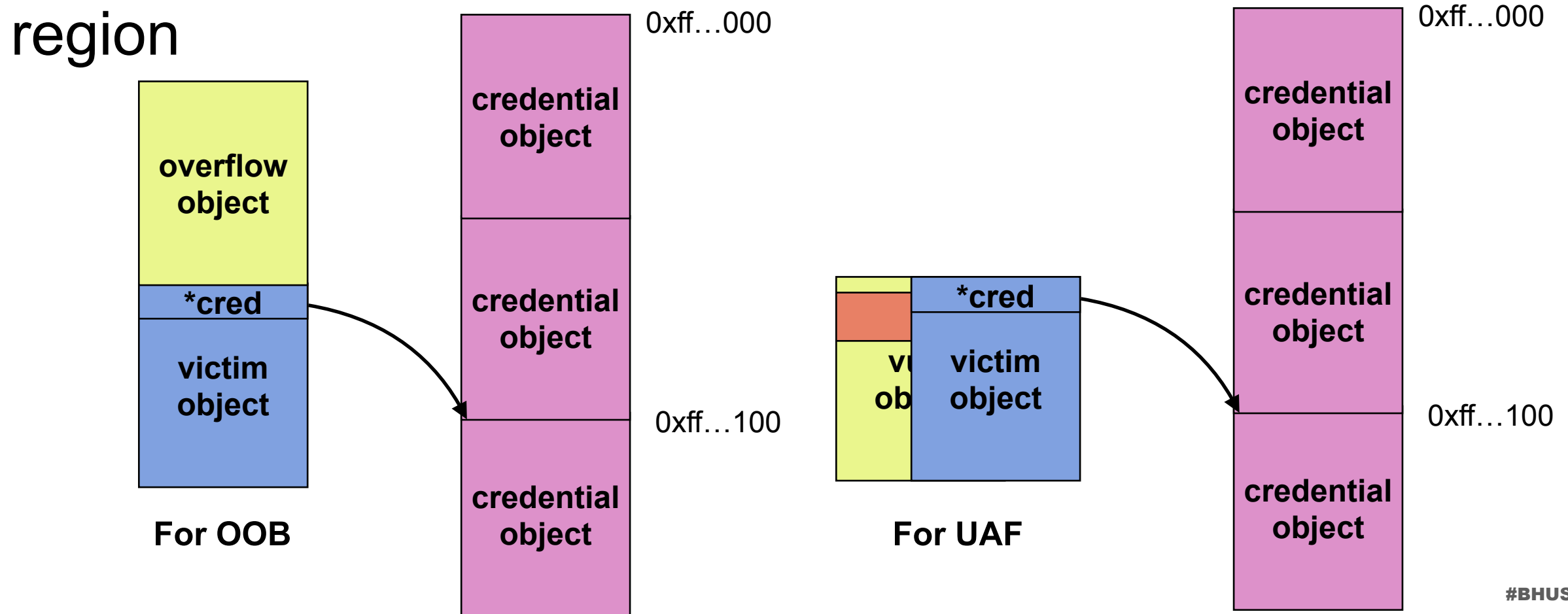
```

struct request_key_auth {
    struct rcu_head      rcu;
    struct key          *target_key;
    struct key          *dest_keyring;
    const struct cred   *cred;
    void               *callout_info;
    size_t             callout_len;
    pid_t              pid;
    char               op[8];
} __randomize_layout;

```

# Pivoting Invalid-Write

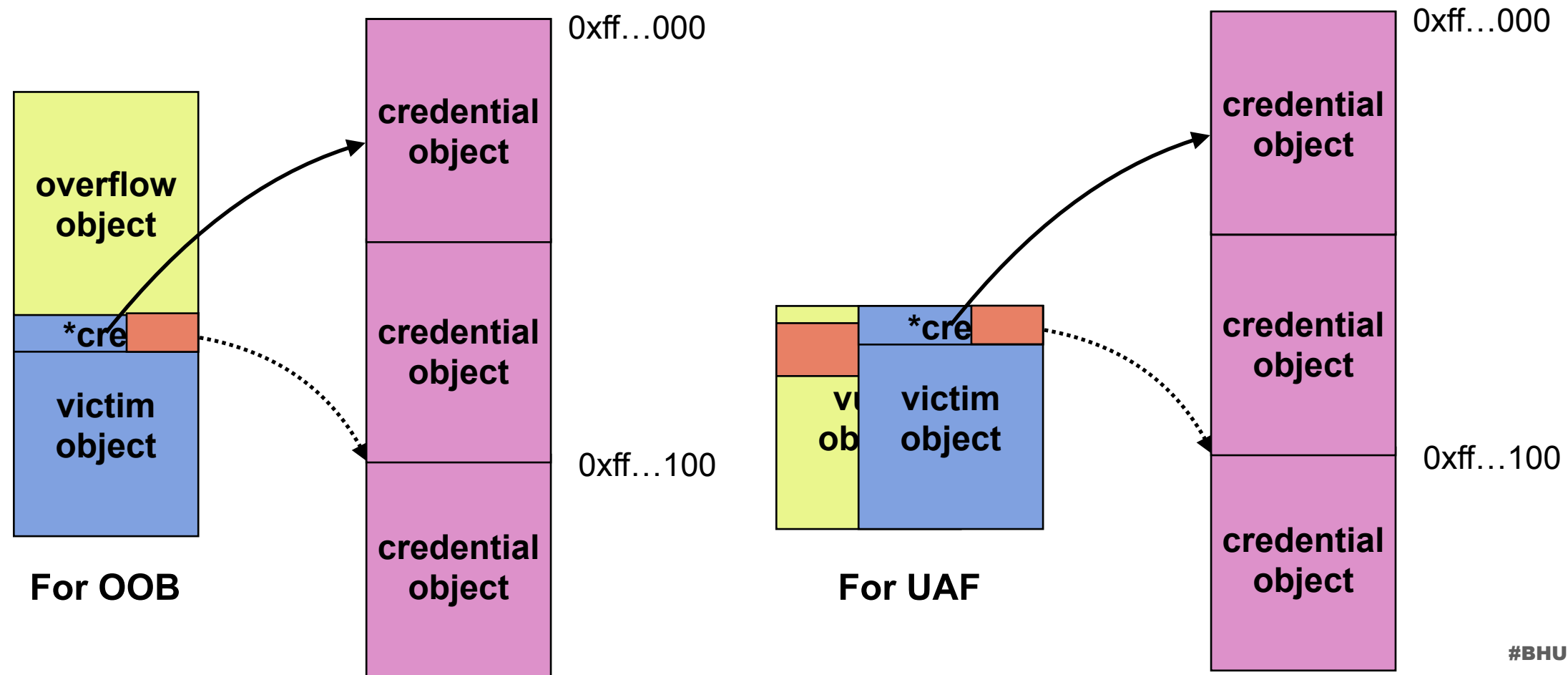
- Manipulate the memory layout to put the *cred* in the overwrite region





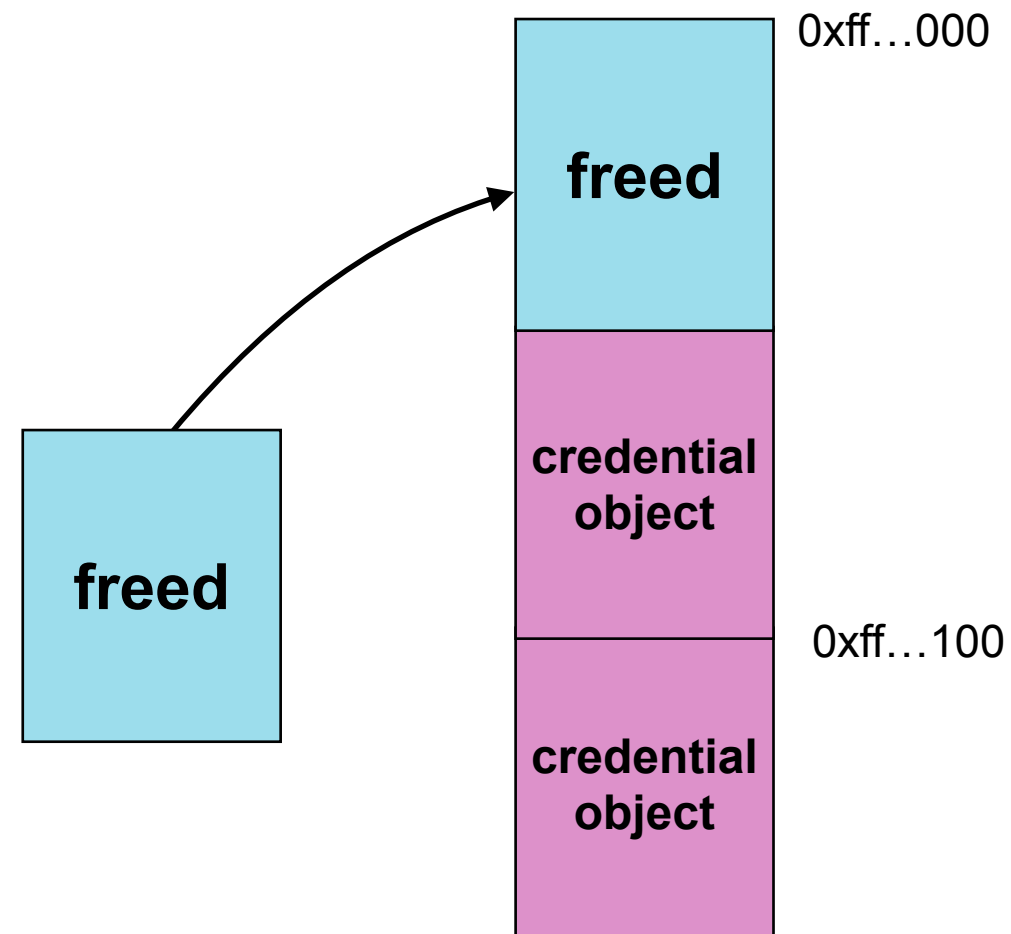
# Pivoting Invalid-Write

- *Partially* overwrite the pointer to cause a reference unbalance



# Pivoting Invalid-Write

- Free the credential object when freeing the victim object

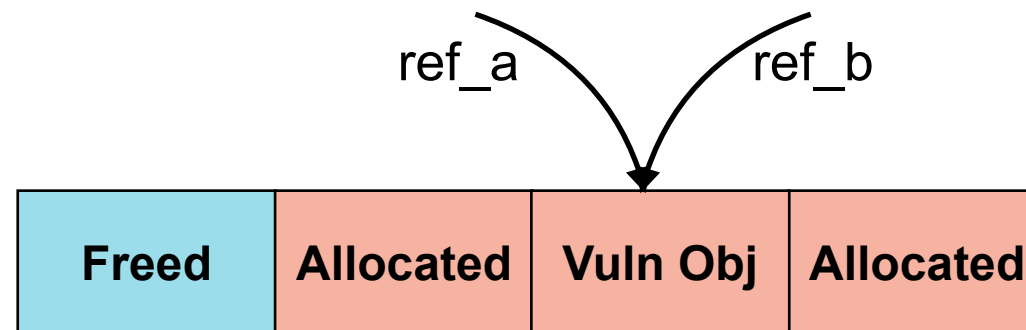


# Pivoting Invalid-Free



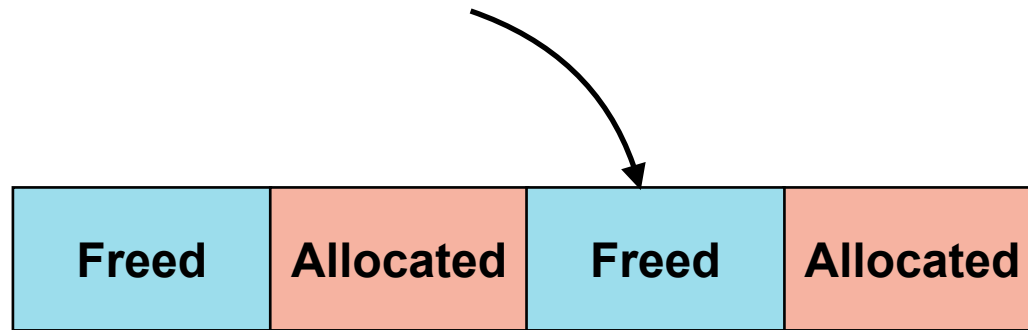
# Pivoting Invalid-Free

- **Two** references to free the same object



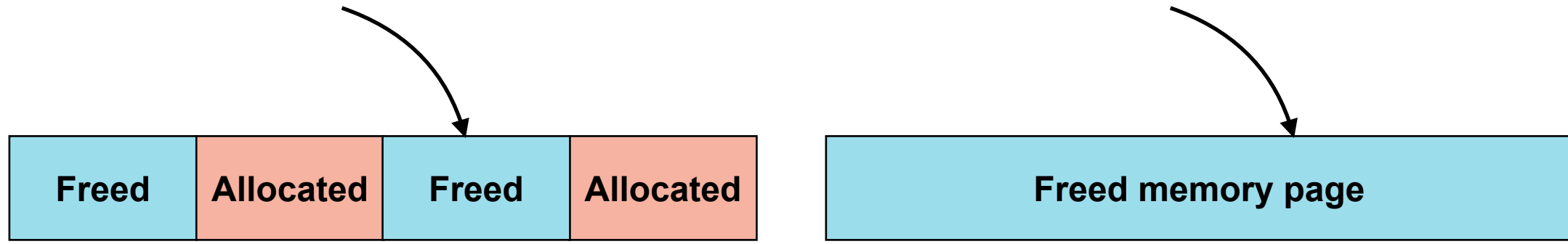
Vulnerable object in kernel memory

# Pivoting Invalid-Free



**Step 1. Trigger the vuln, free the vuln object  
with one reference**

# Pivoting Invalid-Free

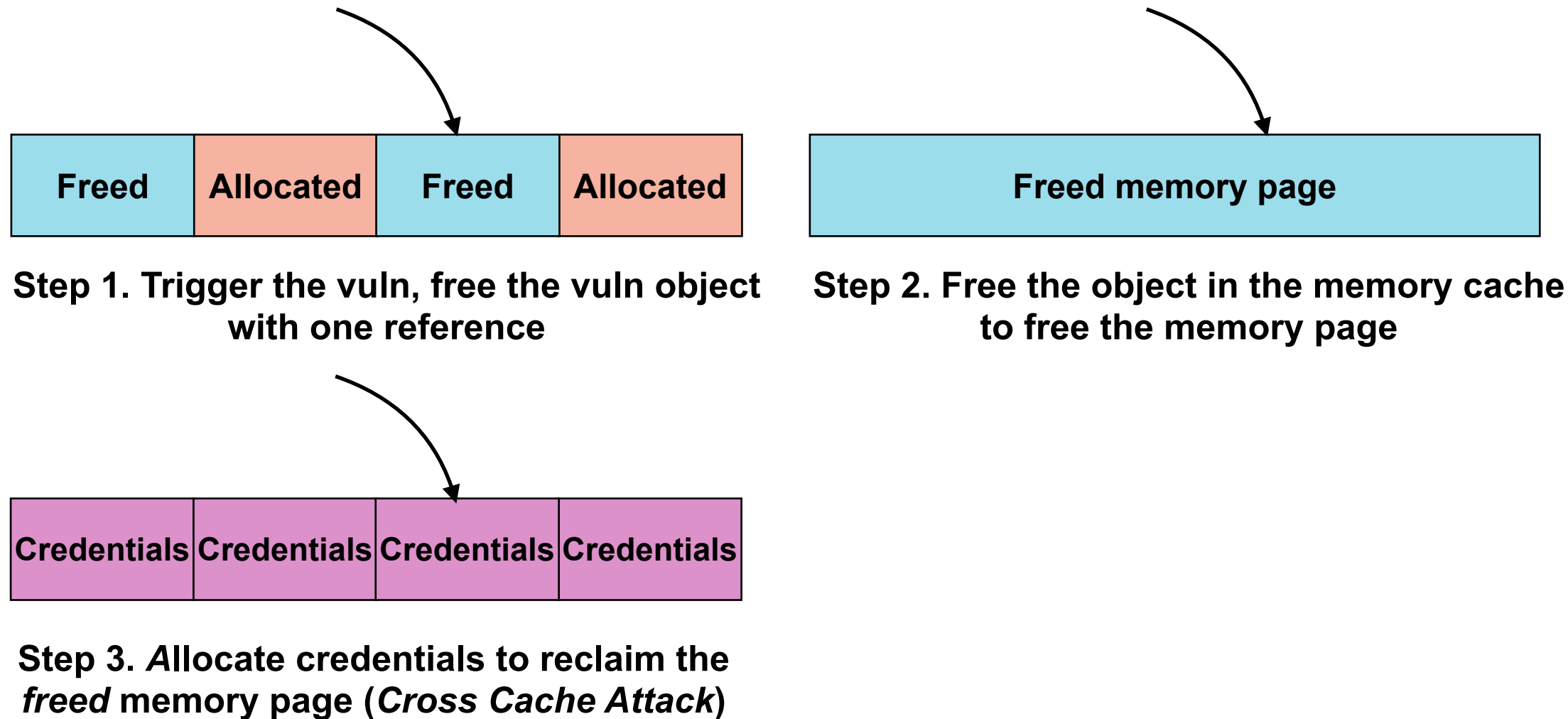


**Step 1. Trigger the vuln, free the vuln object with one reference**

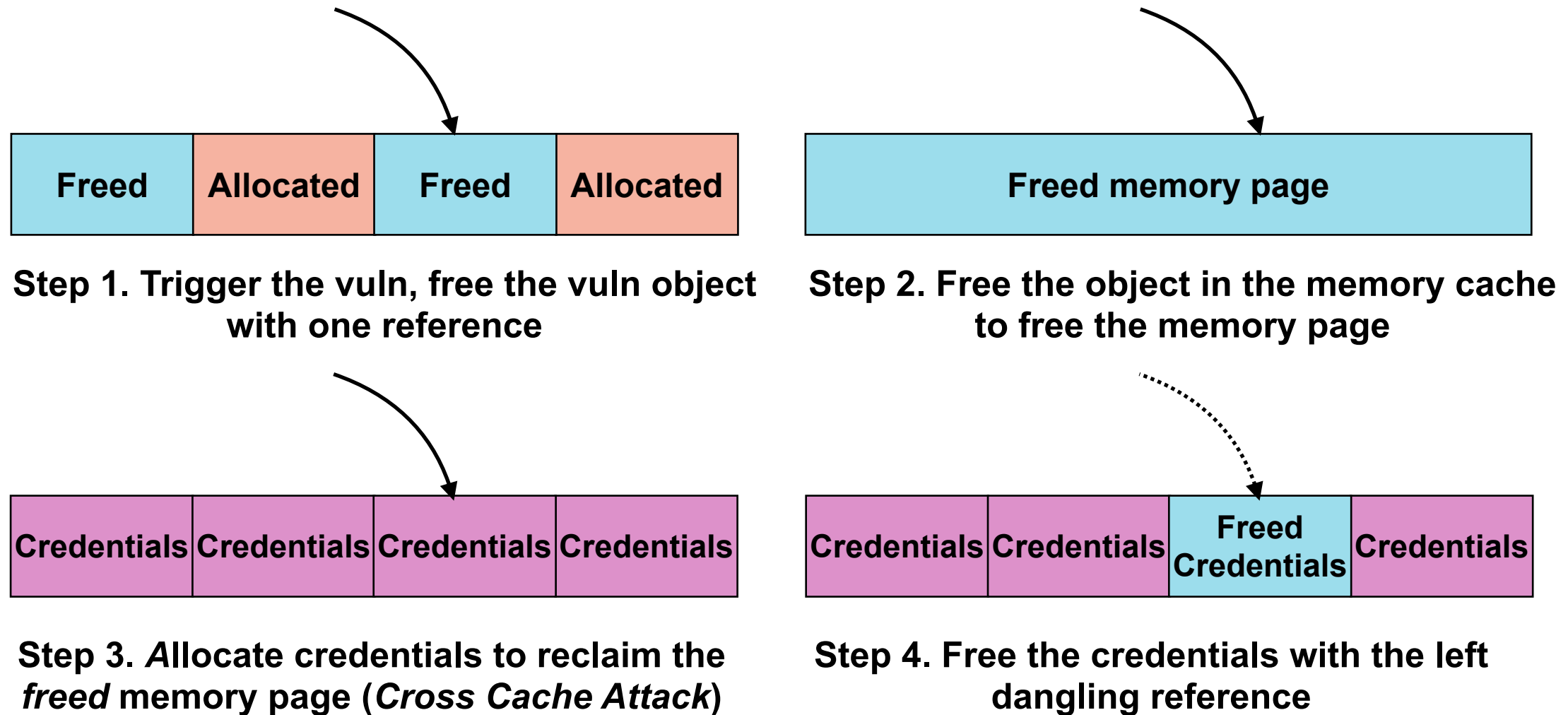
**Step 2. Free the object in the memory cache to free the memory page**



# Pivoting Invalid-Free



# Pivoting Invalid-Free



# Three Challenges

1. How to **free** credentials.
2. How to **allocate** privileged credentials as unprivileged users.  
(attacking *task* credentials)
3. How to **stabilize** file exploitation. (attacking *open file* credentials)



## Challenge 2: Allocating Privileged Task Credentials

- *Unprivileged* users come with *unprivileged* task credentials
- Waiting privileged users to allocate task credentials influences the success rate

# Challenge 2: Allocating Privileged Task Credentials

- **Solution I: Trigger Privileged Userspace Process**
  - Executables with root SUID (e.g. su, mount)
  - Daemons running as root (e.g. sshd)

# Challenge 2: Allocating Privileged Task Credentials

- **Solution I: Trigger Privileged Userspace Process**
  - Executables with root SUID (e.g. su, mount)
  - Daemons running as root (e.g. sshd)
- **Solution II: Trigger Privileged Kernel Thread**
  - Kernel Workqueue — spawn new workers
  - Usermode helper — load kernel modules from userspace

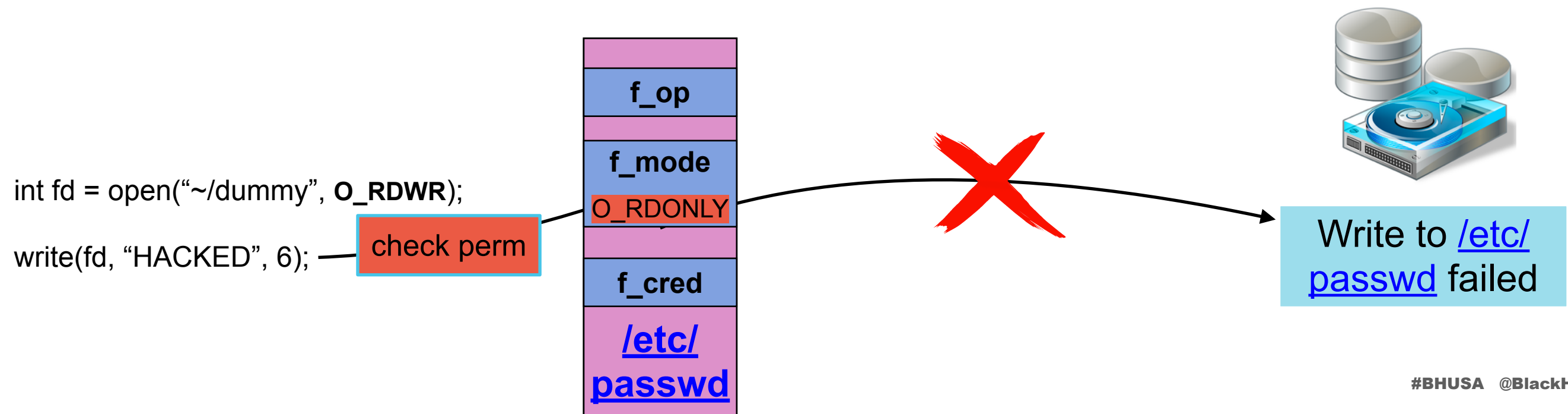


# Three Challenges

1. How to **free** credentials.
2. How to **allocate** *privileged* credentials as *unprivileged* users.  
(attacking *task* credentials)
3. How to **stabilize** file exploitation. (attacking *open file* credentials)

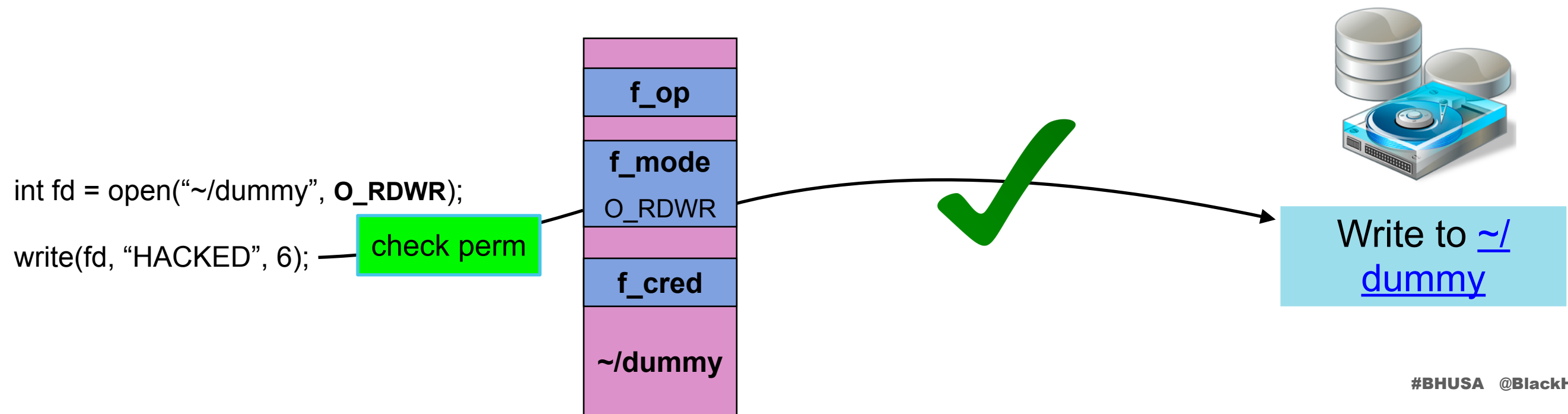
# Challenge 3: Stabilizing File Exploitation

- The swap of *file* object happens before permission check



# Challenge 3: Stabilizing File Exploitation

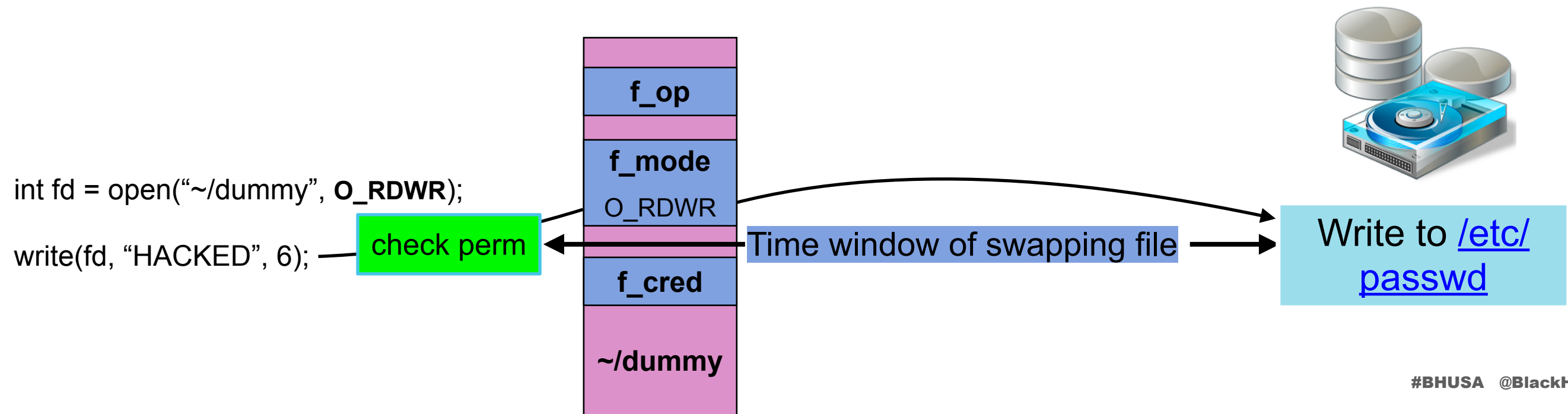
- The swap of *file* object happens after file write.





# Challenge 3: Stabilizing File Exploitation

- The swap of *file* object should happen between permission check and actual file write
- The desired time window is small



# Challenge 3: Stabilizing File Exploitation

- **Solution I: Extend with Userfaultfd or FUSE**
  - *Pause* kernel execution when accessing userspace memory

# Solution I: Userfaultfd & FUSE

- Pause at *import\_iovec* before v4.13
- *import\_iovec* copies userspace memory

```
ssize_t vfs_writev(...)  
{  
    // permission checks  
    if (!(file->f_mode & FMODE_WRITE))  
        return -EBADF;  
    if (!(file->f_mode & FMODE_CAN_WRITE))  
        return -EINVAL;  
  
    ...  
    // import iovec to kernel, where kernel would be paused  
    // using userfaultfd & FUSE  
    res = import_iovec(type, uvector, nr_segs,  
                      ARRAY_SIZE(iovstack), &iov, &iter);  
    ...  
    // do file writev  
}
```



# Solution I: Userfaultfd & FUSE

- **Pause at *import\_iovec* before v4.13**
  - *import\_iovec* copies userspace memory
  - Used in Jann Horn's exploitation for [CVE-2016-4557](#)
  - *Dead* after v4.13

# Solution I: Userfaultfd & FUSE

- `vfs_writev` after v4.13

```
ssize_t vfs_writev(...)
{
    ...
    // import iovec to kernel, where kernel would be paused
    // using userfaultfd
    res = import_iovec(type, uvector, nr_segs,
                      ARRAY_SIZE(iovstack), &iov, &iter);
    ...
    // permission checks
    if (!(file->f_mode & FMODE_WRITE))
        return -EBADF;
    if (!(file->f_mode & FMODE_CAN_WRITE))
        return -EINVAL;
    ...
    // do file writev
}
```

# Solution I: Userfaultfd & FUSE

- Pause at `generic_perform_write`
  - *prefaults* user pages
  - **Pauses** kernel execution at the page fault

```
ssize_t generic_perform_write(struct file *file,
                             struct iov_iter *i, loff_t pos)
{
    /*
     * Bring in the user page that we will copy from _first_.
     * Otherwise there's a nasty deadlock on copying from the
     * same page as we're writing to, without it being marked
     * up-to-date.
     */
    if (unlikely(iov_iter_fault_in_readable(i, bytes))) {
        status = -EFAULT;
        break;
    }
    ...
    // call the write operation of the file system
    status = a_ops->write_begin(file, mapping, pos, bytes, flags,
                               &page, &fsdata);
    ...
}
```



# Challenge 3: Stabilizing File Exploitation

- **Solution I: Extend with Userfaultfd & FUSE**
  - *Pause* kernel execution when accessing userspace memory
  - Userfaultfd & FUSE might not be available
- **Solution II: Extend with file lock**
  - Pause kernel execution with lock

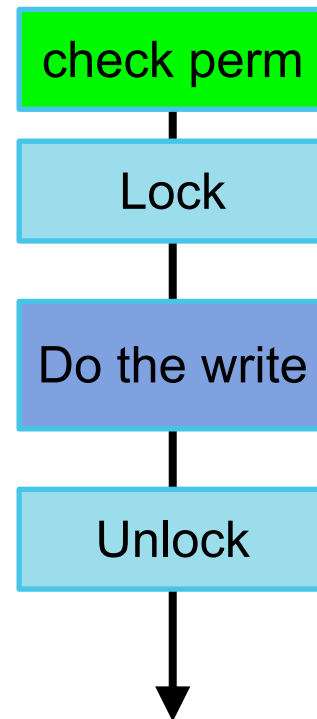
# Solution II: File Lock

- A lock of the *inode* of the file
- Lock the file when it is being writing to

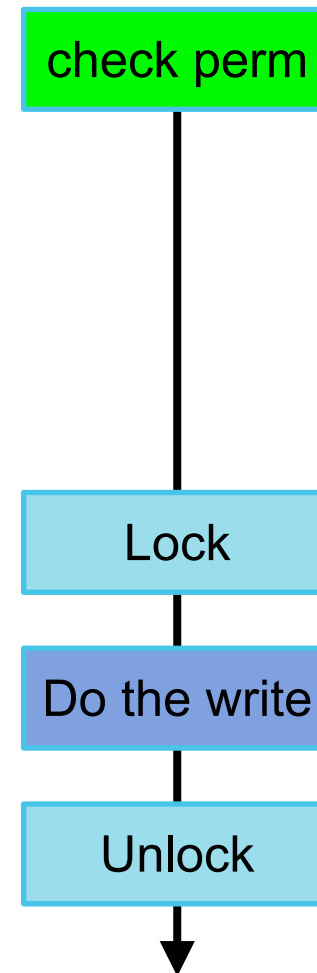
```
static ssize_t ext4_buffered_write_iter(struct kiocb *iocb,  
                                       struct iov_iter *from)  
{  
    ssize_t ret;  
    struct inode *inode = file_inode(iocb->ki_filp);  
    inode_lock(inode);  
    ...  
    ret = generic_perform_write(iocb->ki_filp, from,  
                               ↔ iocb->ki_pos);  
    ...  
    inode_unlock(inode);  
    return ret;  
}
```

# Solution II: File Lock

Thread A



Thread B





# Solution II: File Lock

Thread A

check perm

Lock

Do the write  
(write 4GB)

Unlock



Thread B

check perm

Lock

Do the write

Unlock



A large time window

# Demo Time!

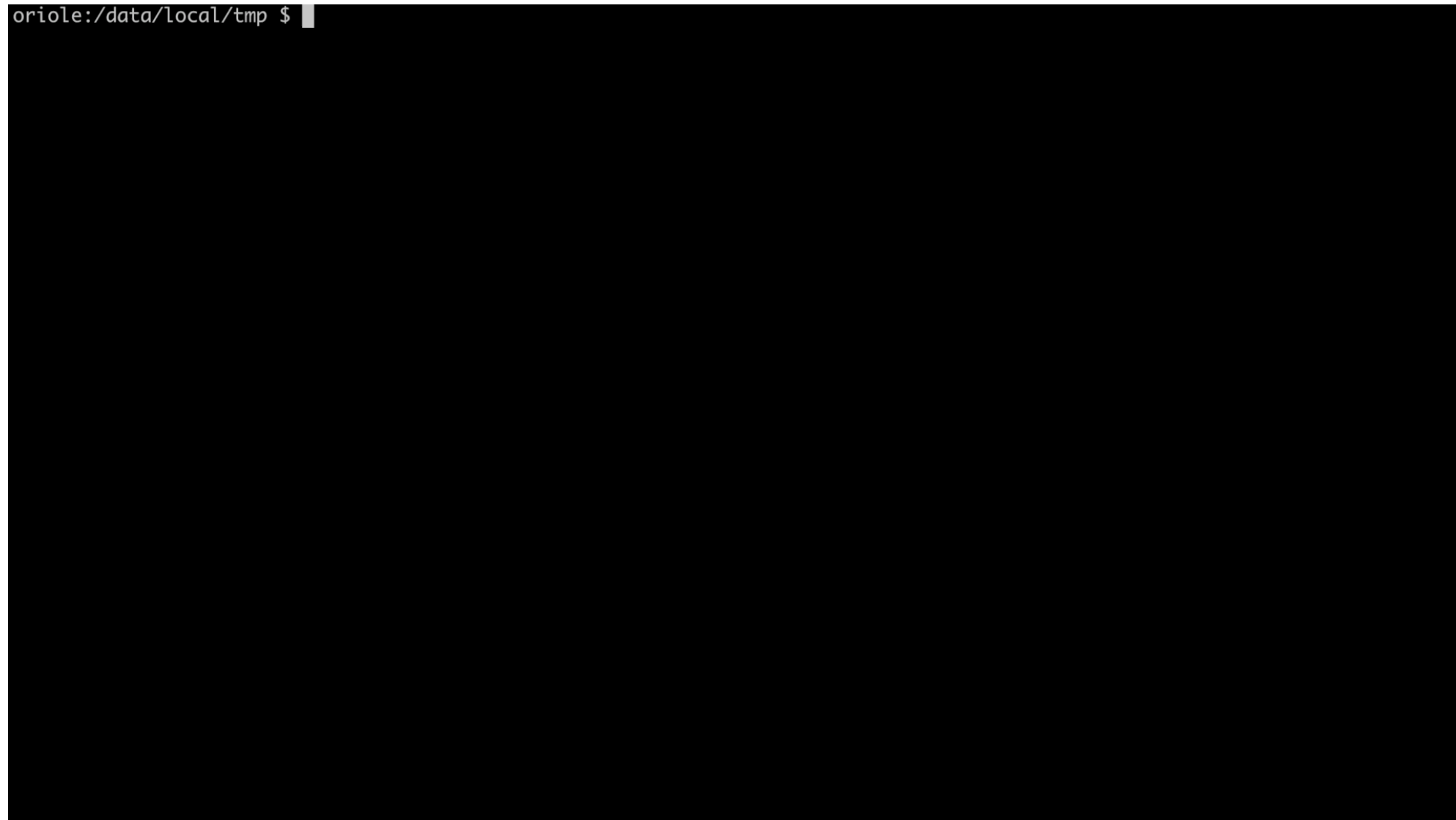
# CVE-2021-4154



# Centos 8 and Ubuntu 20

```
[low@centos8 ~]$  
low@ubuntu:~$ id
```

# Android Kernel with CFI enabled\*



\* access check removed for demonstration

# Advantages of DirtyCred

- **A generic method**
  - The method applies to container and Android.
- **Simple but powerful**
  - No need to deal with KASLR, CFI.
  - Data-only method.
- **Exploitation friendly**
  - Make your exploit **universal!**
  - empowers different bugs to be Dirty-Pipe-liked (sometimes even better).



# Defense Against DirtyCred

- **Fundamental problem**
  - Object isolation is based on *type* not *privilege*
- **Solution**
  - *Isolate* privileged credentials from unprivileged ones
- **Where to isolate?**
  - Virtual memory (using *vmalloc*): No ***cross cache attack*** anymore!
- Code is available at <https://github.com/markakd/DirtyCred>

# Takeaways

- **New exploitation concept — DirtyCred: swapping credentials**
- **Principled approach to different challenges**
- ***Universal* exploits to different kernels**
- **Effective defense**

Zhenpeng Lin ([@Markak\\_](https://twitter.com/Markak_))

<https://zplin.me>

zplin@u.northwestern.edu

