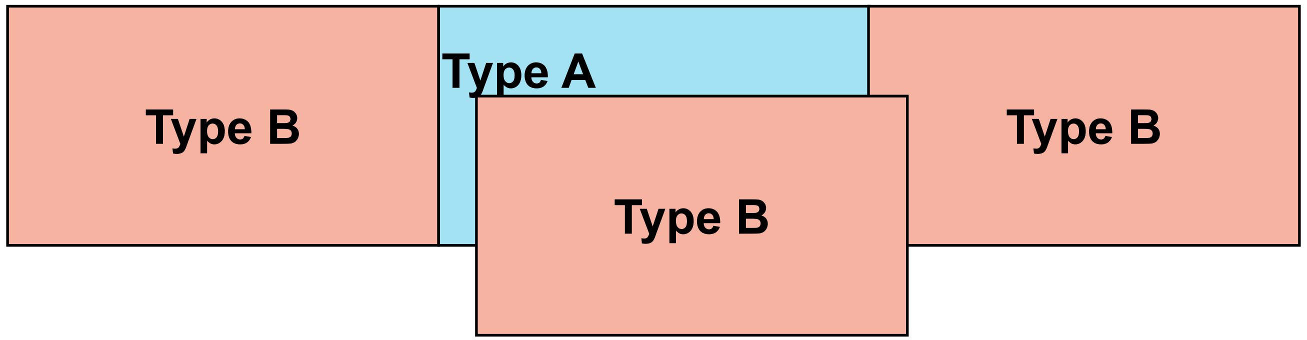
DirtyCred: Escalating Privilege in Linux Kernel

Zhenpeng Lin

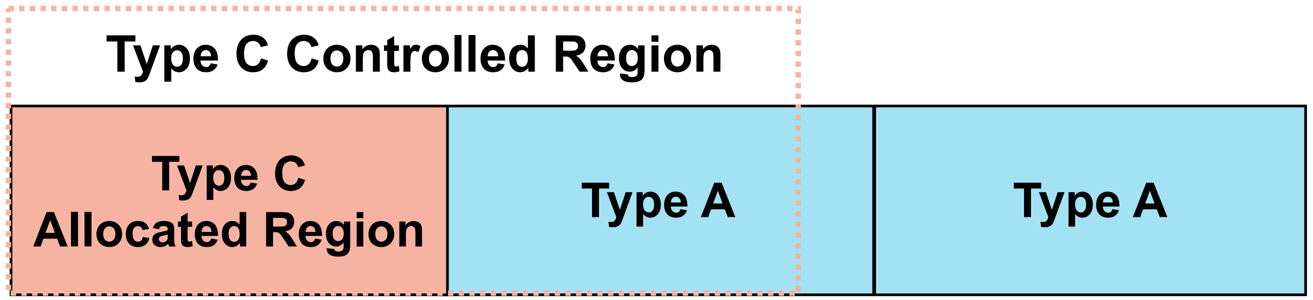
11/07/2022

How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap



(a) Type confusion between Type A and B

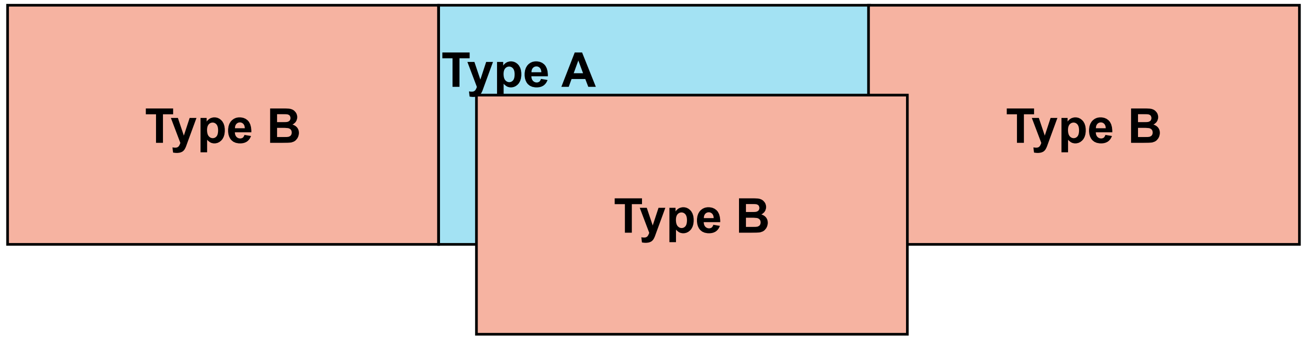


(b) Partial overlap between Type C and A

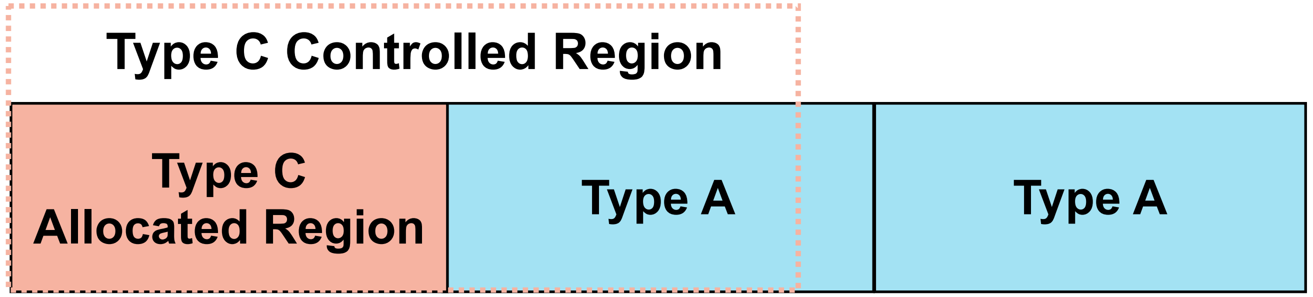
How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives



(a) Type confusion between Type A and B



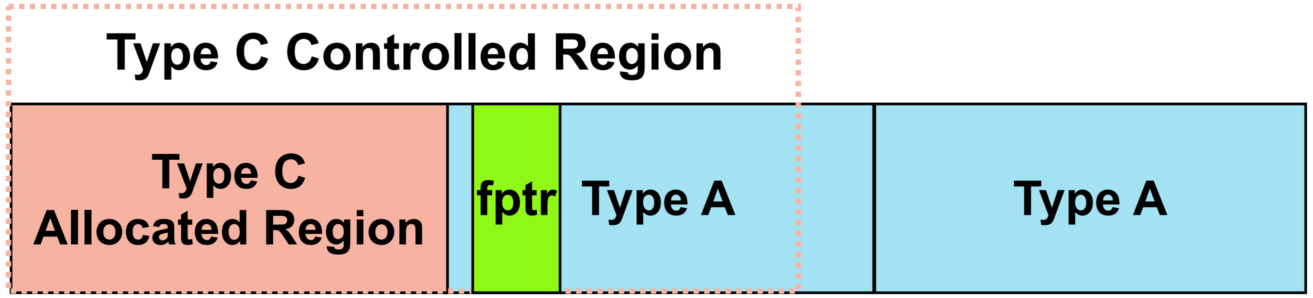
(b) Partial overlap between Type C and A

How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- Leak kernel pointers
- Tamper kernel pointers



Partial overlap between Type C and A

How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- Leak kernel pointers
- Tamper kernel pointers

Bypass Mitigation

How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- Leak kernel pointers
- Tamper kernel pointers

Bypass Mitigation

- Execute ROP in different forms[1]

[1] *Joy of exploiting the kernel*

How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- Leak kernel pointers
- Tamper kernel pointers

Bypass Mitigation

- Execute ROP in different forms[1]

Escalate Privilege

[1] *Joy of exploiting the kernel*

How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- Leak kernel pointers
- Tamper kernel pointers

Bypass Mitigation

- Execute ROP in different forms[1]

Escalate Privilege

Used by 15/17 exploits in [2]

[1] *Joy of exploiting the kernel*

[2] *Kernel Exploit Recipes Notebook*

How DirtyCred Exploits Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and overlap
- Leak kernel pointers
- Tamper kernel pointers
- Execute ROP

How DirtyCred Exploits Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- Leak kernel pointers
- Tamper kernel pointers
- Execute ROP

How DirtyCred Exploits Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- ~~Leak kernel pointers~~
- ~~Tamper kernel pointers~~
- ~~Execute ROP~~

How DirtyCred Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap

Obtain Primitives

- Swap kernel credentials

Escalate Privilege

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

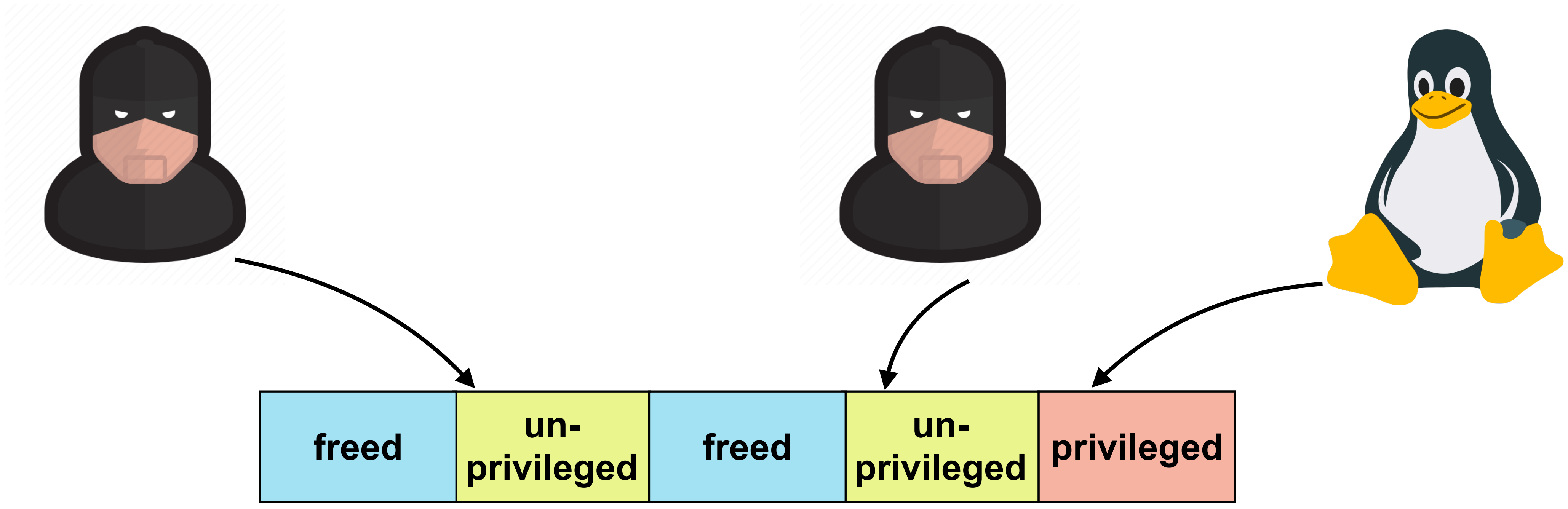
Task Credential

- `Struct cred` in Linux kernel's implementation

```
struct cred {
        atomic_t        usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
        atomic_t        subscribers;    /* number of processes subscribed */
        void            *put_addr;
        unsigned        magic;
#define CRED_MAGIC      0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
        kuid_t          uid;            /* real UID of the task */
        kgid_t          gid;            /* real GID of the task */
        kuid_t          suid;           /* saved UID of the task */
        kgid_t          sgid;           /* saved GID of the task */
        kuid_t          euid;           /* effective UID of the task */
        kgid_t          egid;           /* effective GID of the task */
        kuid_t          fsuid;          /* UID for VFS ops */
        kgid_t          fsgid;          /* GID for VFS ops */
```

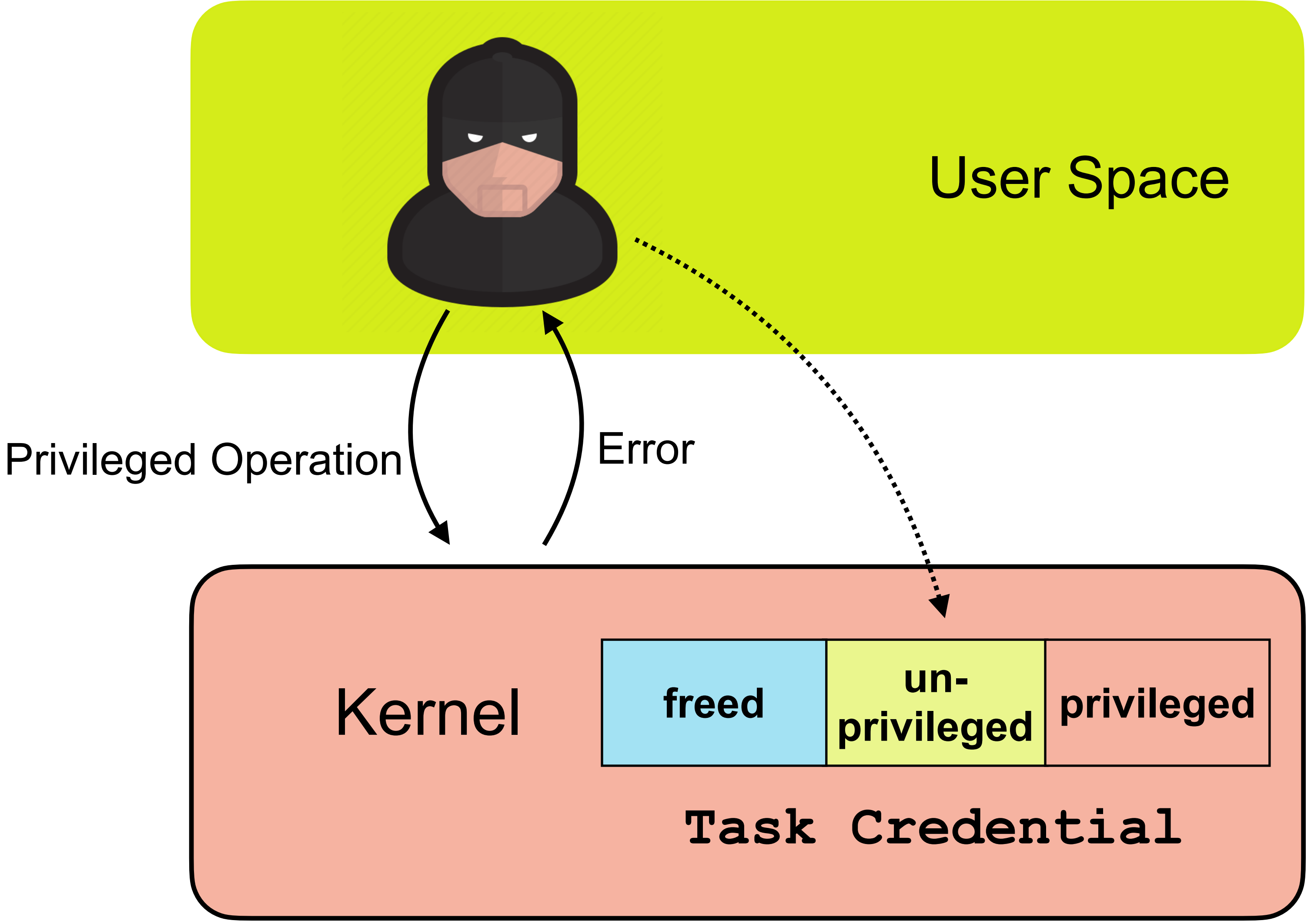
Task Credential

- **`Struct cred`** in Linux kernel's implementation
- Represents the ***privilege*** of kernel tasks

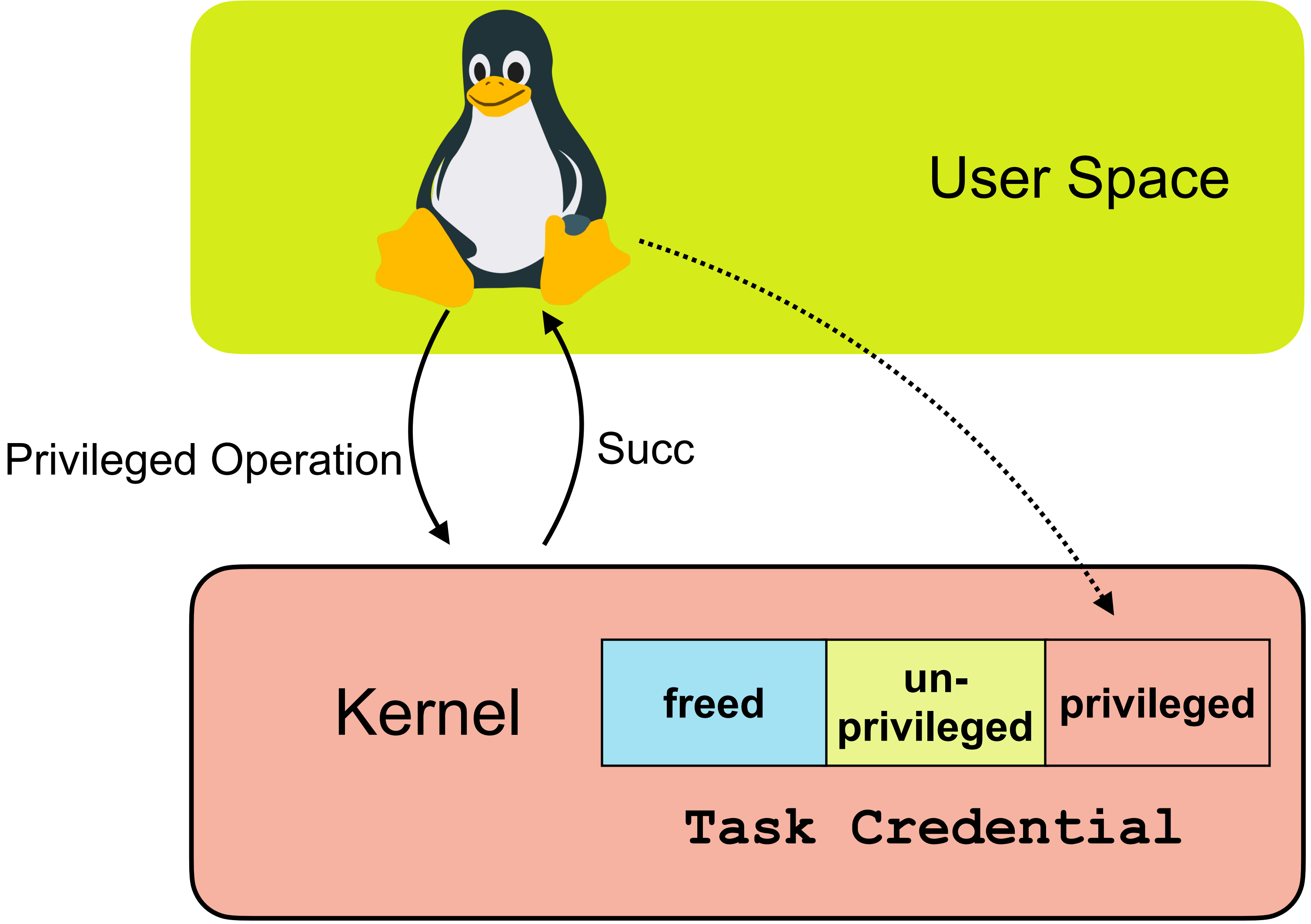


`Task Credential` on kernel heap

How Linux Kernel Uses Task Credential



How Linux Kernel Uses Task Credential



Open File Credential

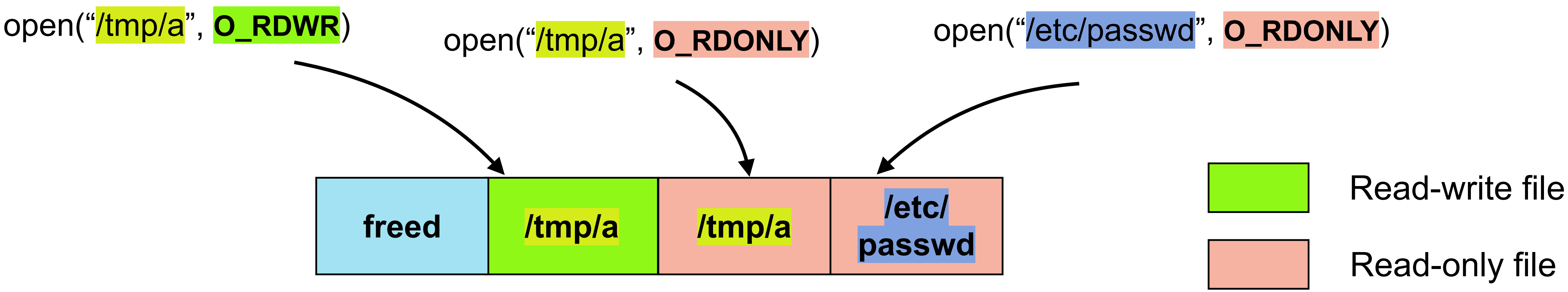
- **`Struct file`** in Linux kernel's implementation

```
struct file {
        union {
                struct llist_node       f_llist;
                struct rcu_head         f_rcuhead;
                unsigned int            f_iocb_flags;
        };
        struct path             f_path;
        struct inode            *f_inode;       /* cac
        const struct file_operations    *f_op;

        /*
         * Protects f_ep, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;
```

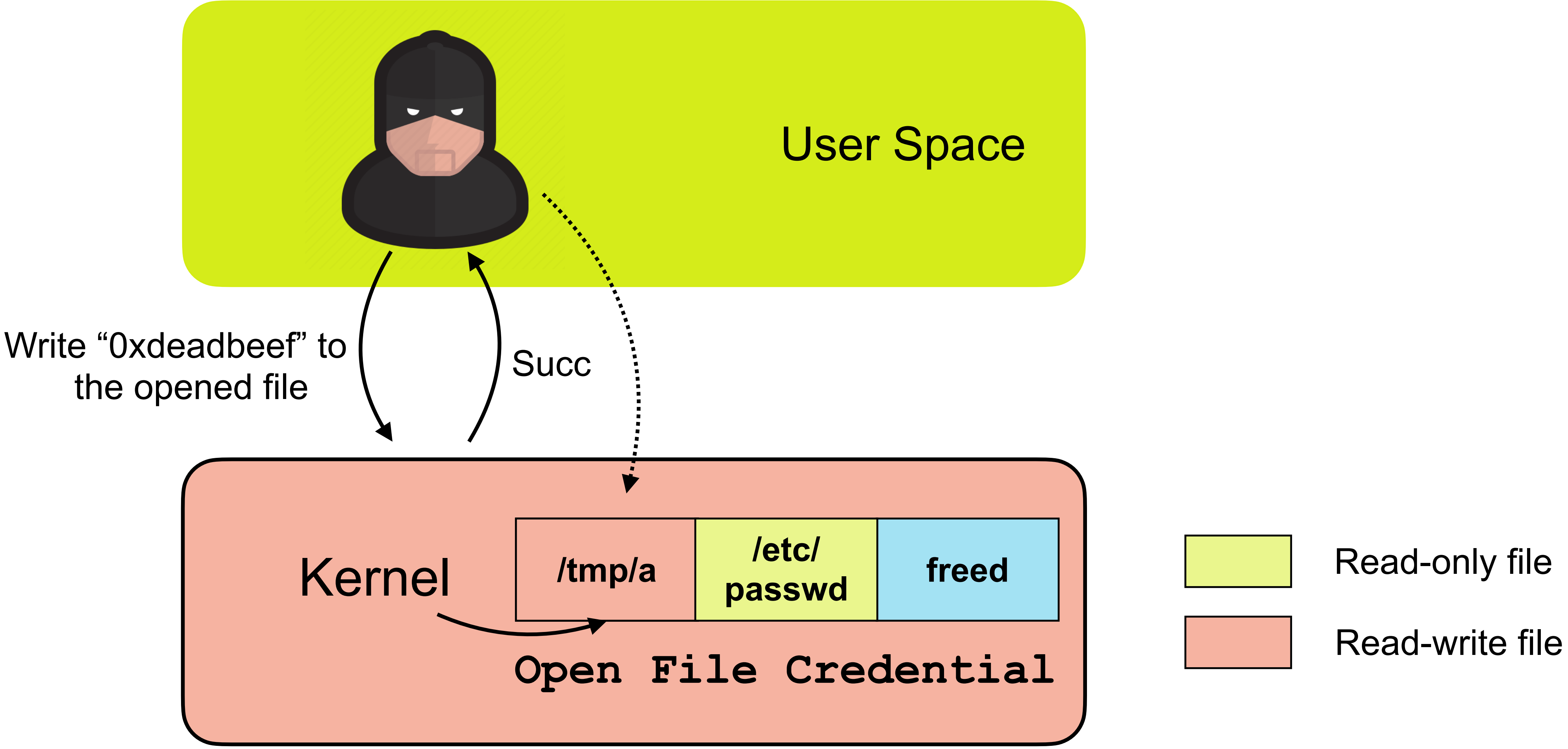
Open File Credential

- Carries the information of opened files (*e.g.* mode, path, *etc.*)

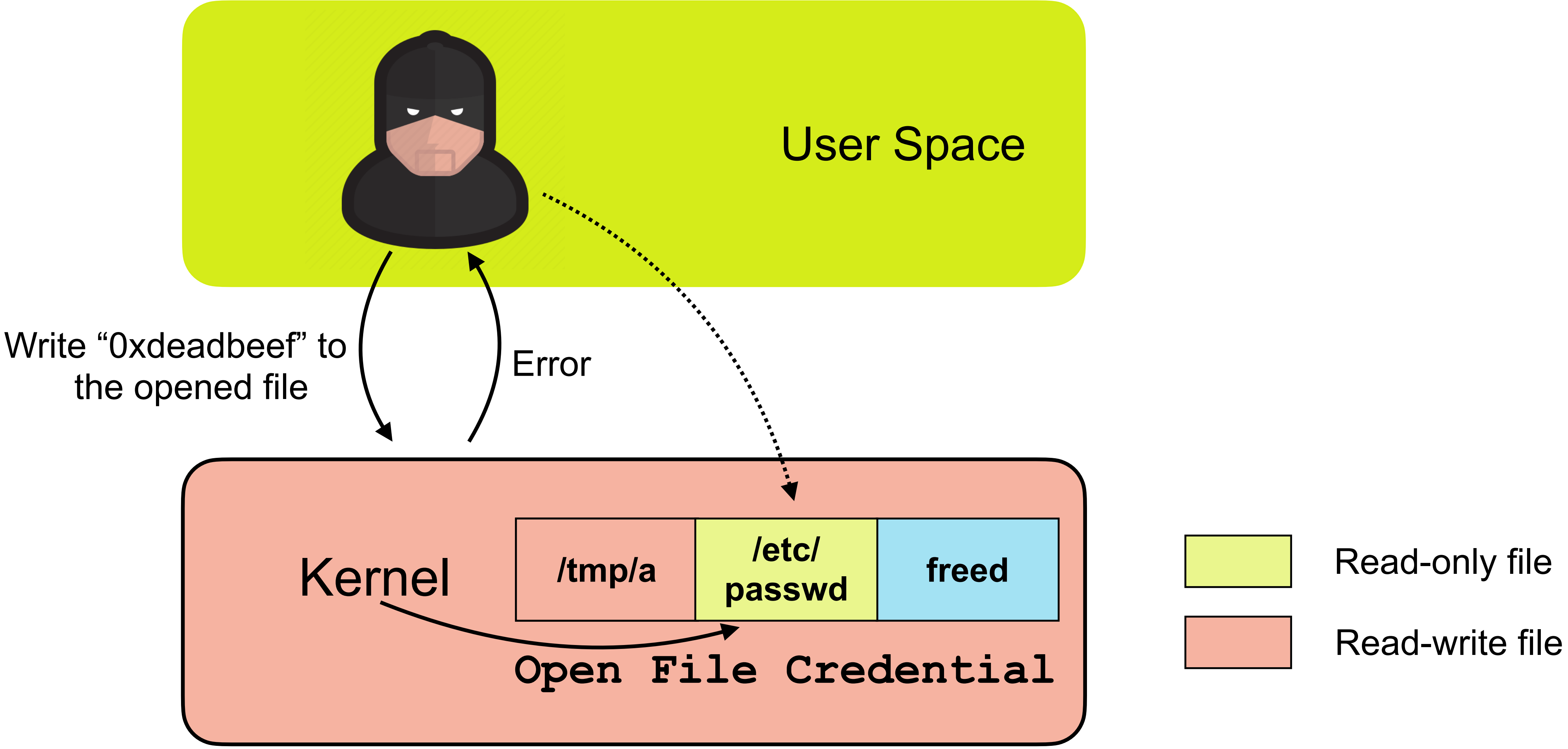


`Open File Credential` on kernel heap

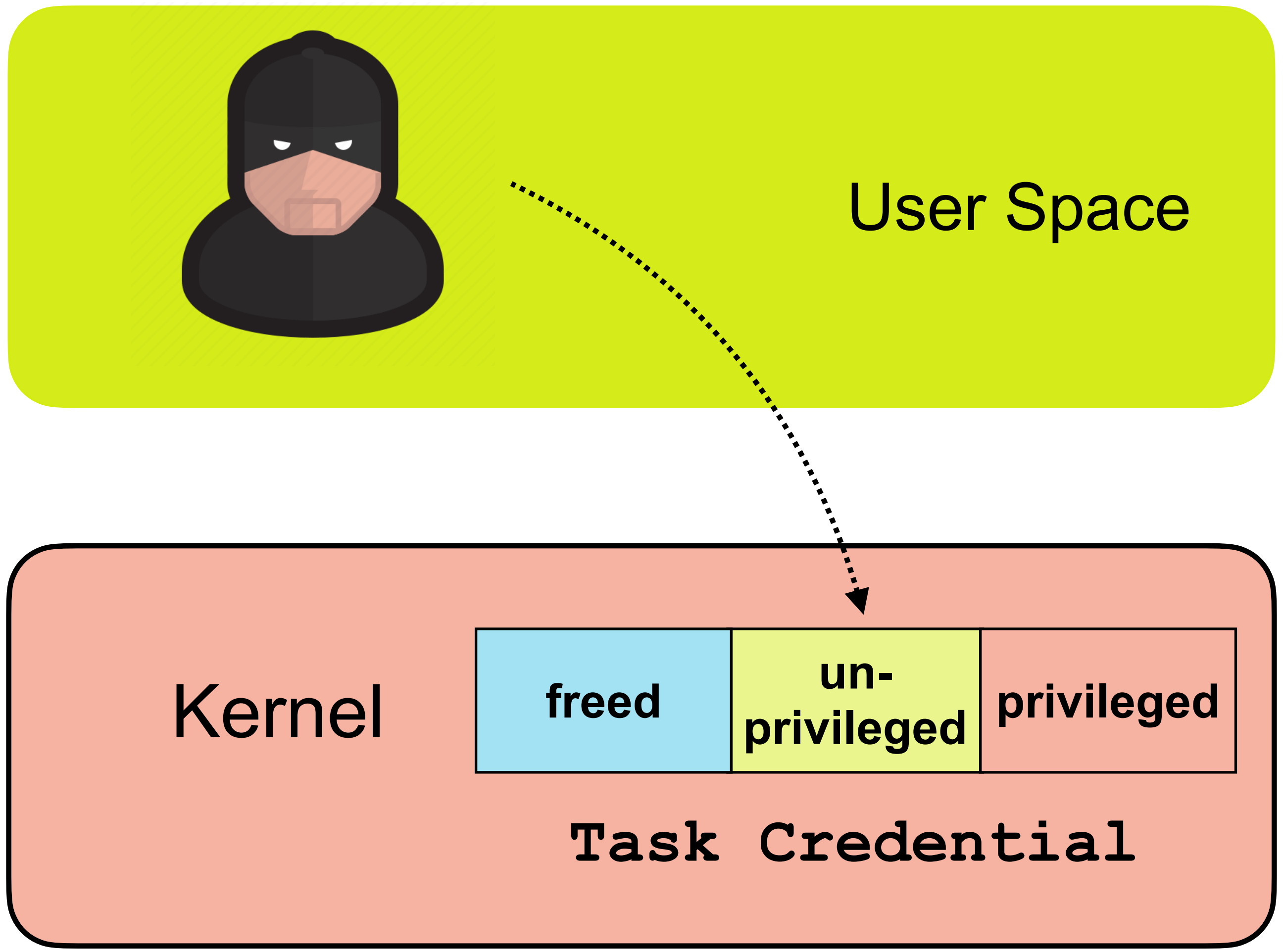
How Linux Kernel Uses Open File Credential



How Linux Kernel Uses Open File Credential

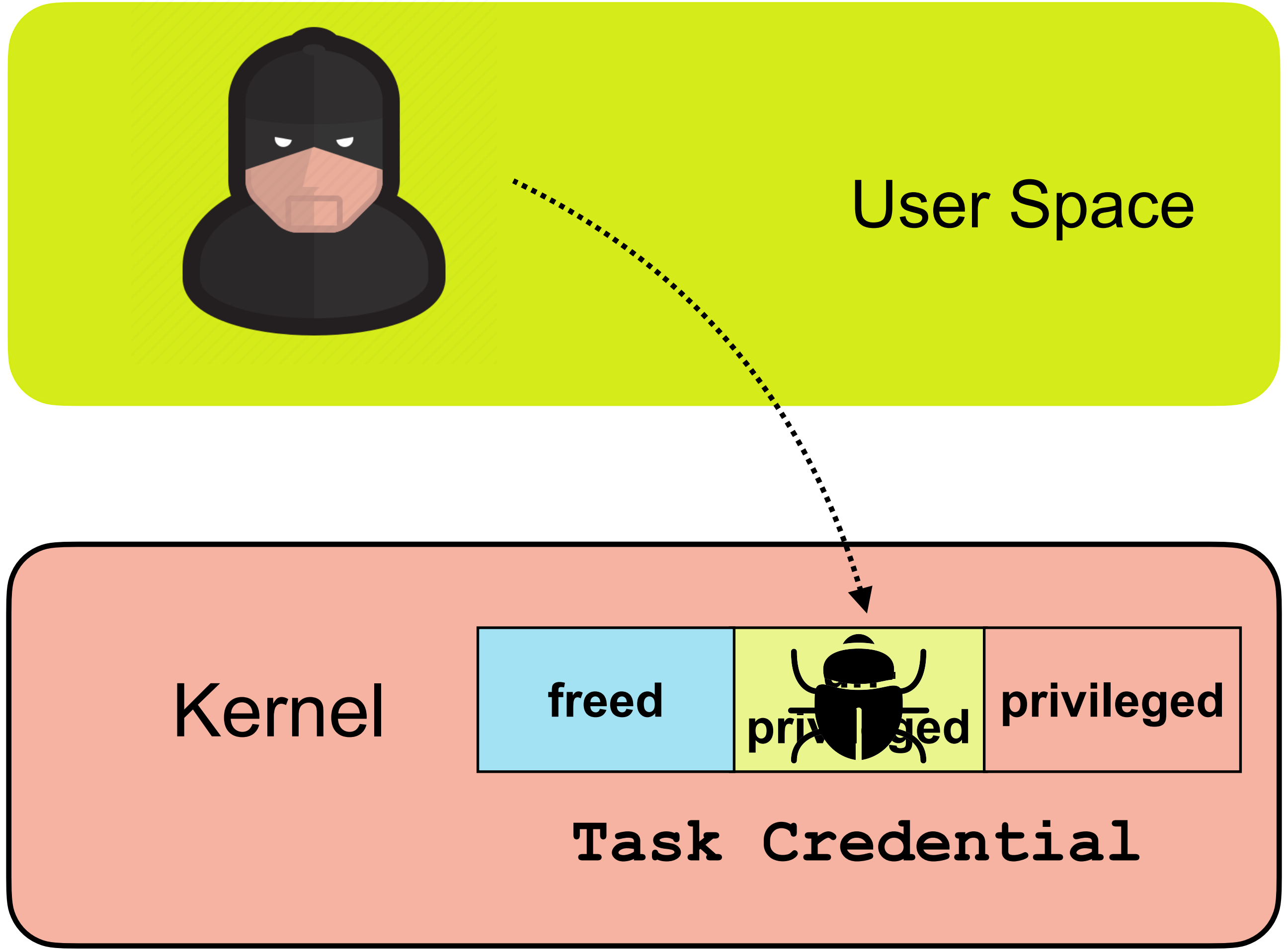


Attacking Task Credential



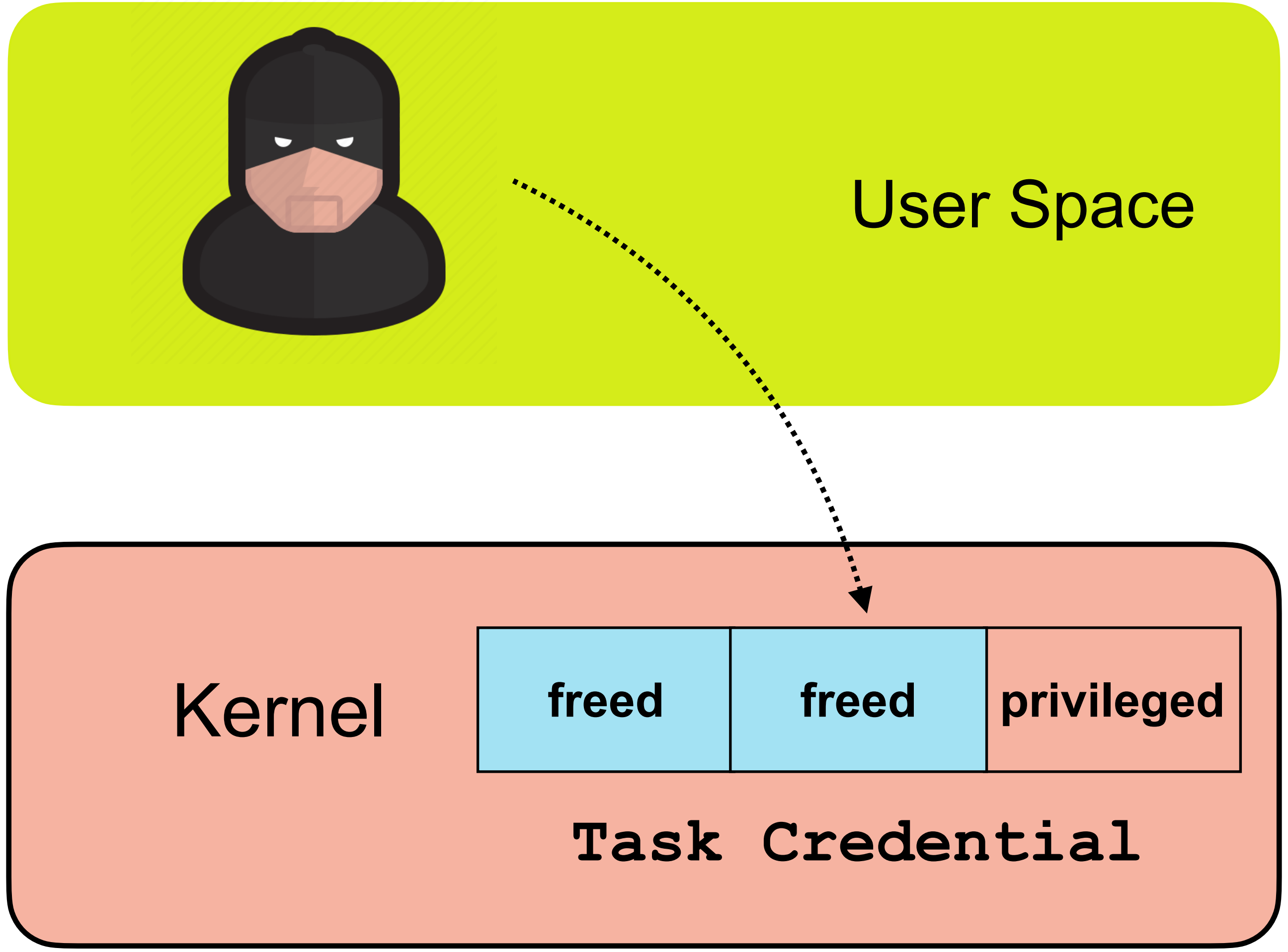
Attacking Task Credential

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



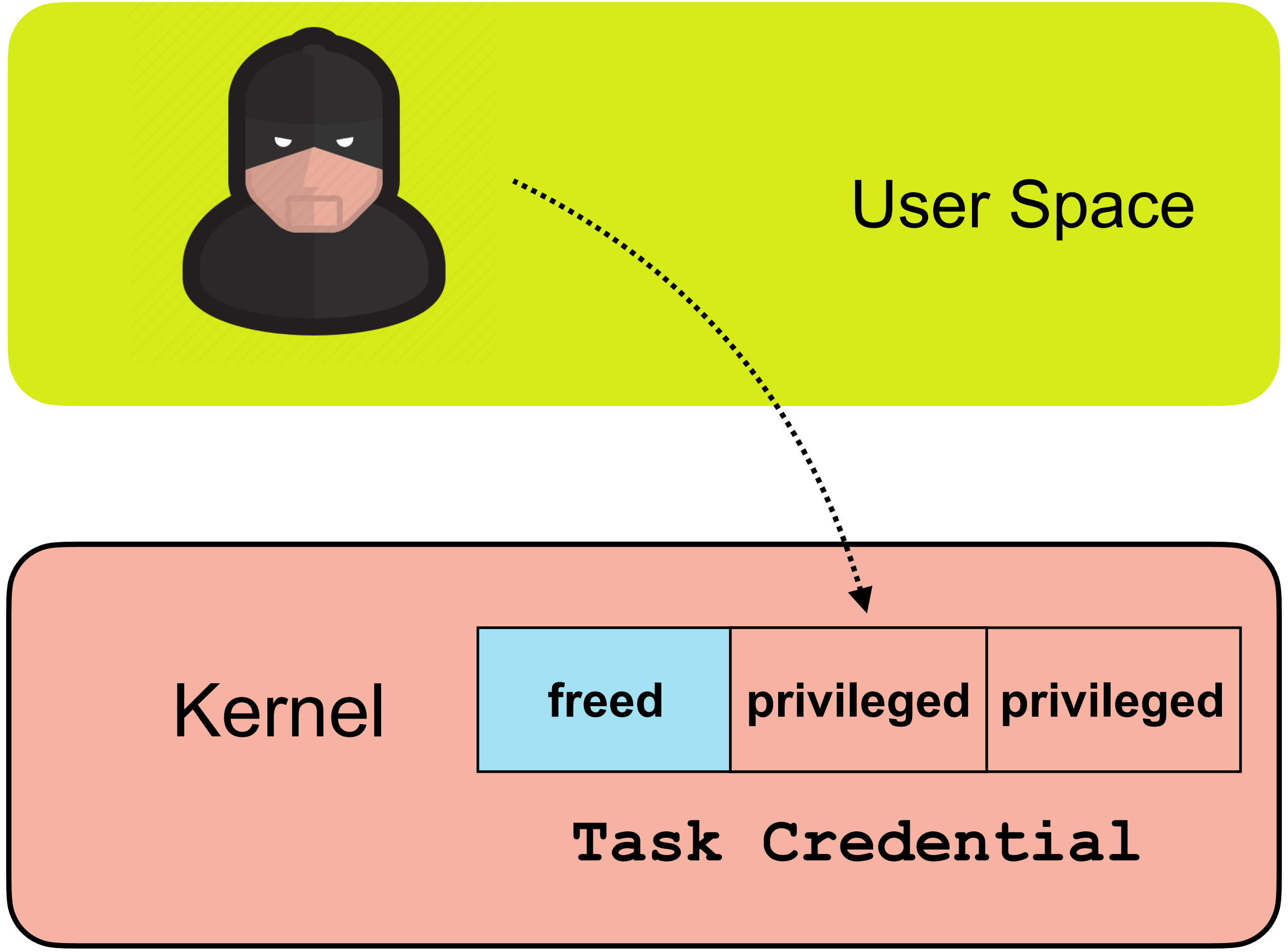
Attacking Task Credential

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



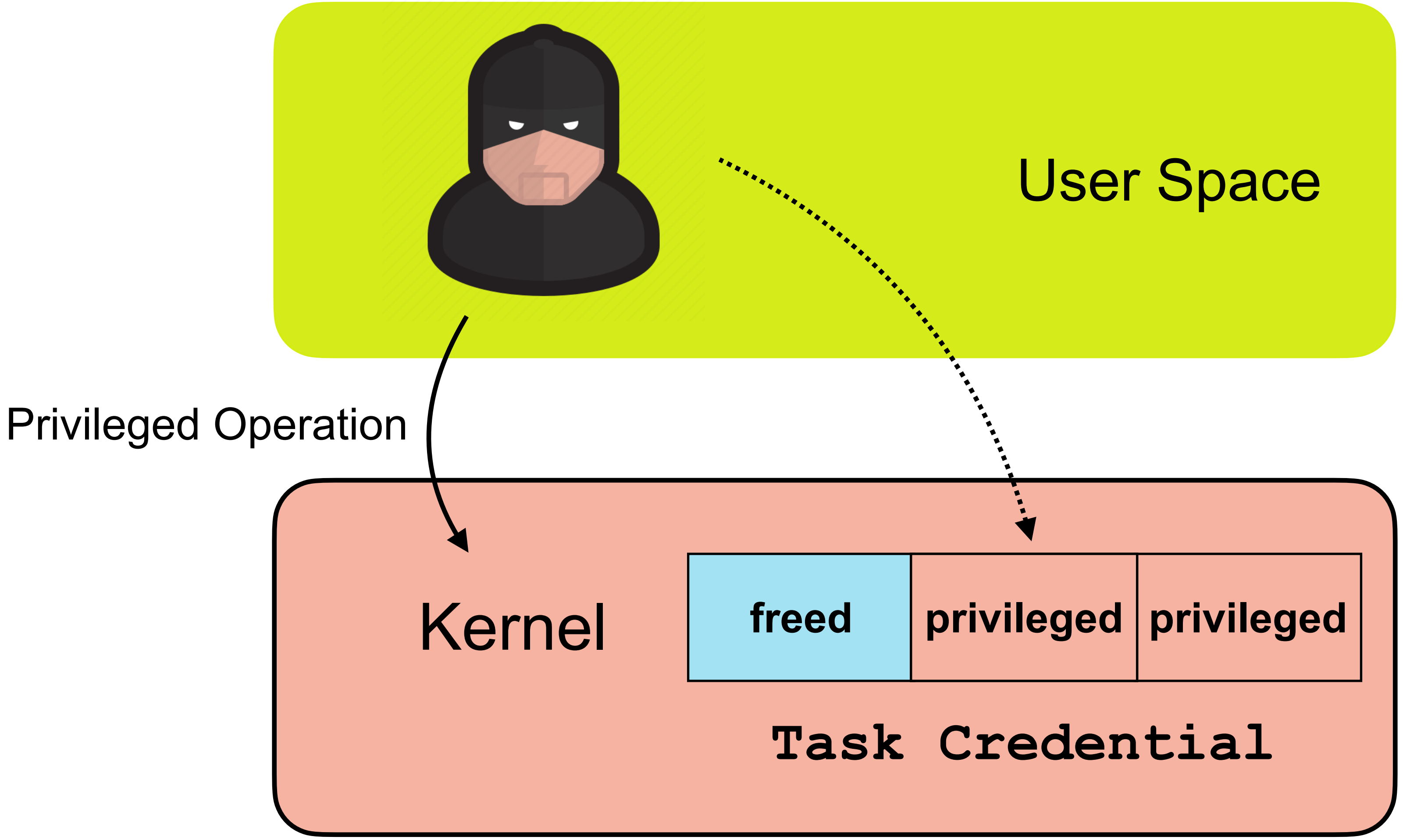
Attacking Task Credential

Step 2. **Allocate** a privileged credential in the **freed** memory slot



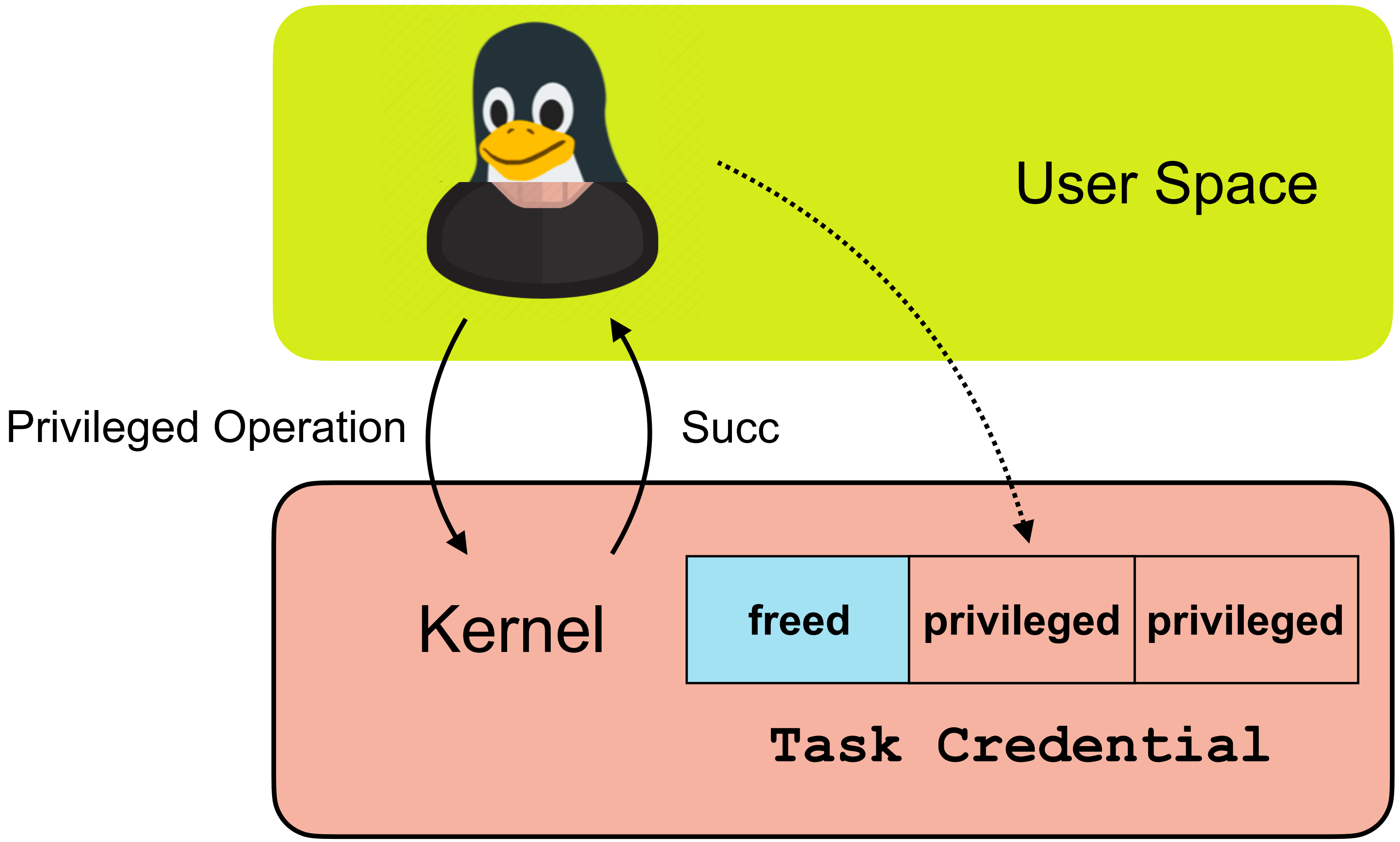
Attacking Task Credential

Result: Becoming a *privileged* user

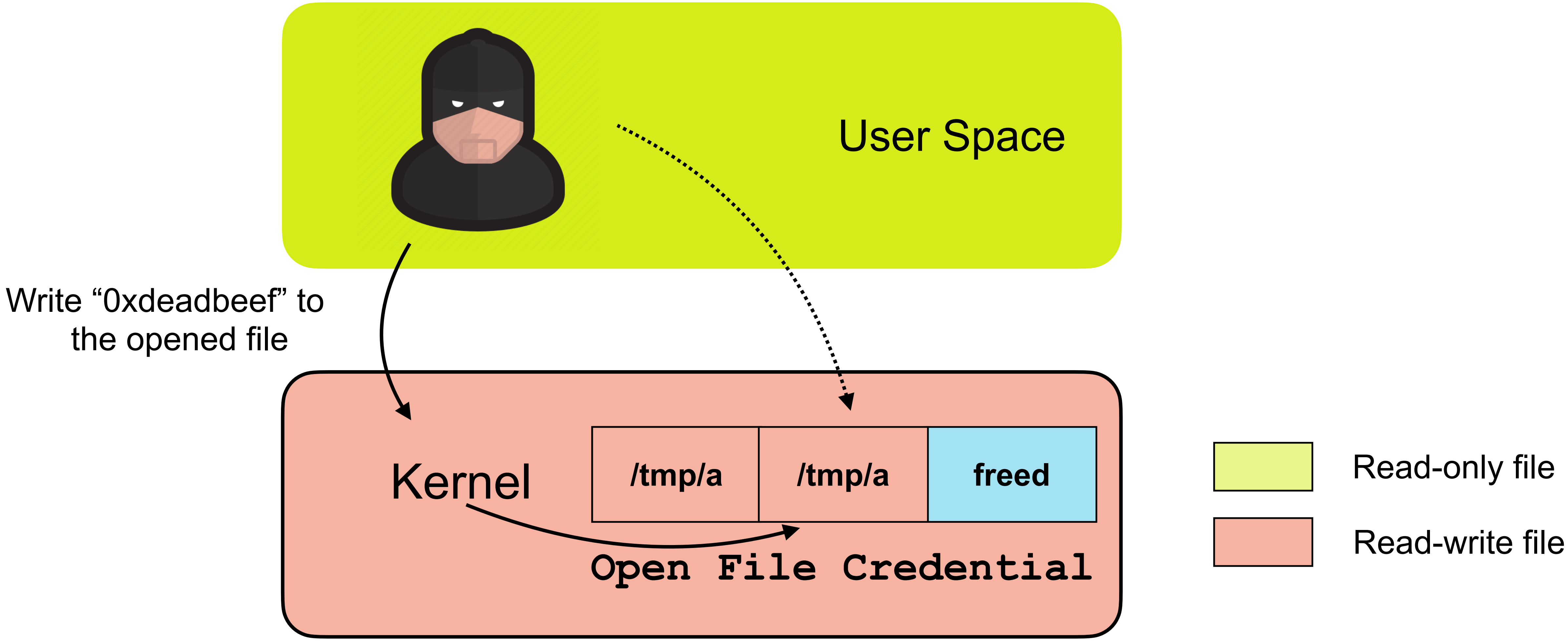


Attacking Task Credential

Result: Becoming a *privileged* user

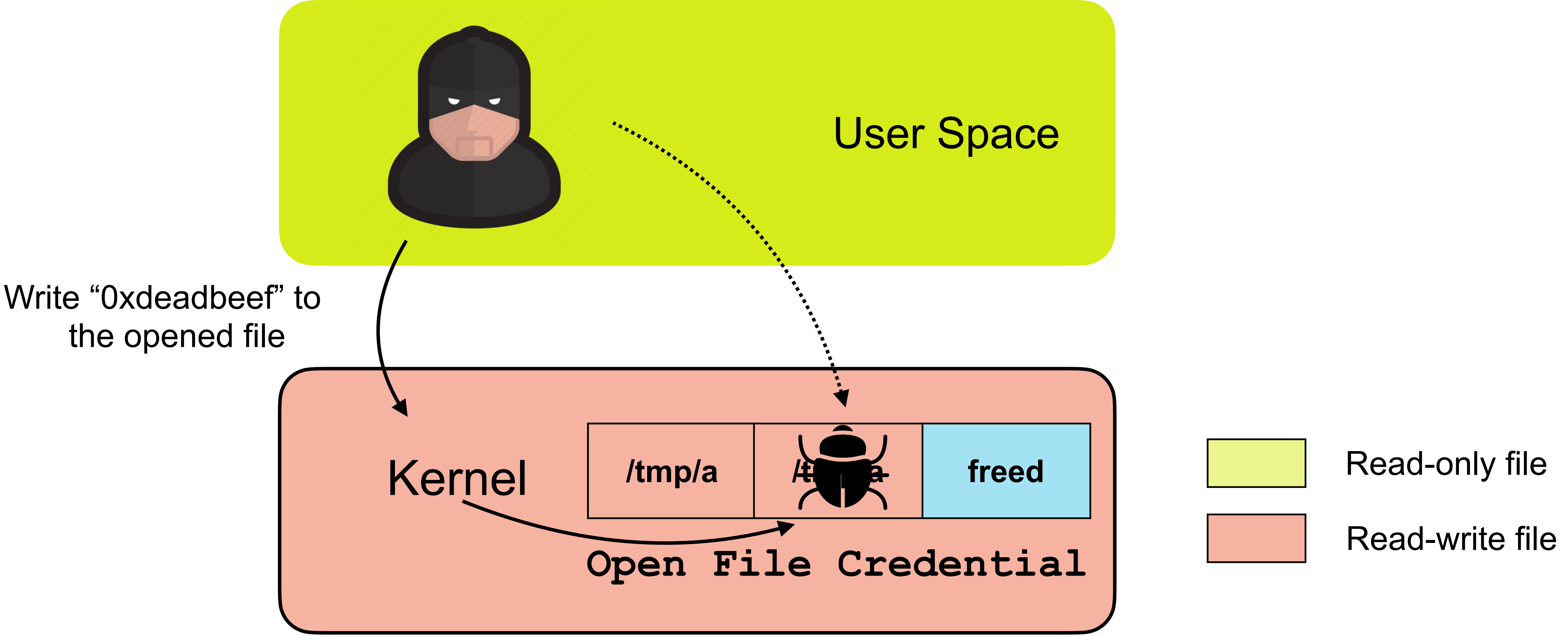


Attacking Open File Credential



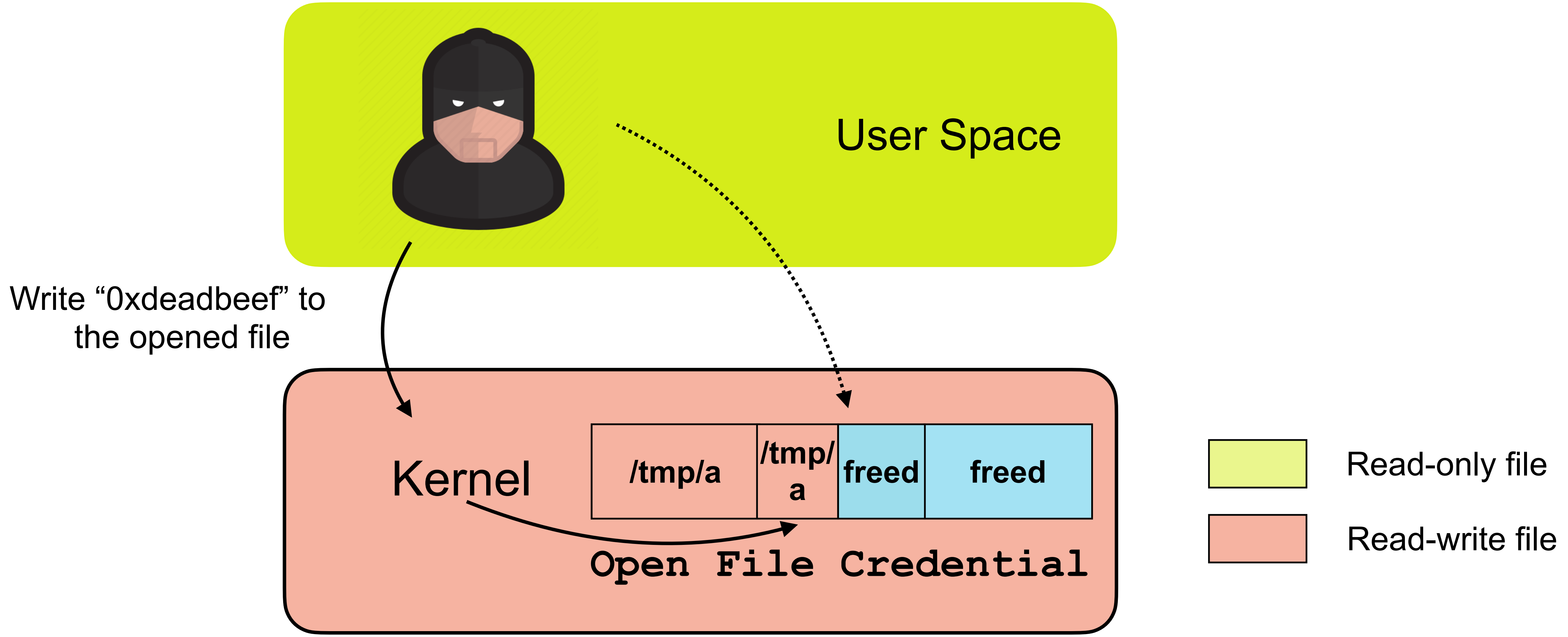
Attacking Open File Credential

Step 1. **Free** a *read-write* file ***after*** checks, but ***before*** writing to disk



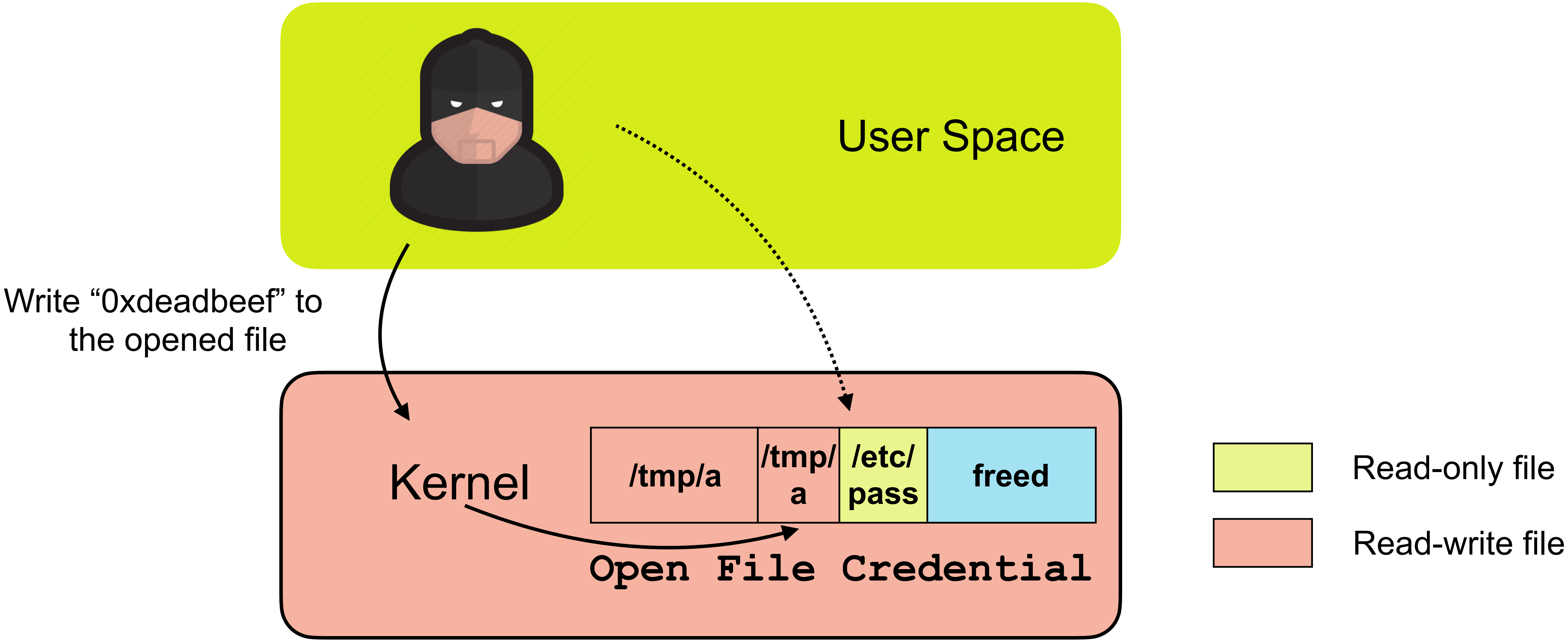
Attacking Open File Credential

Step 1. **Free** a *read-write* file ***after*** checks, but ***before*** writing to disk



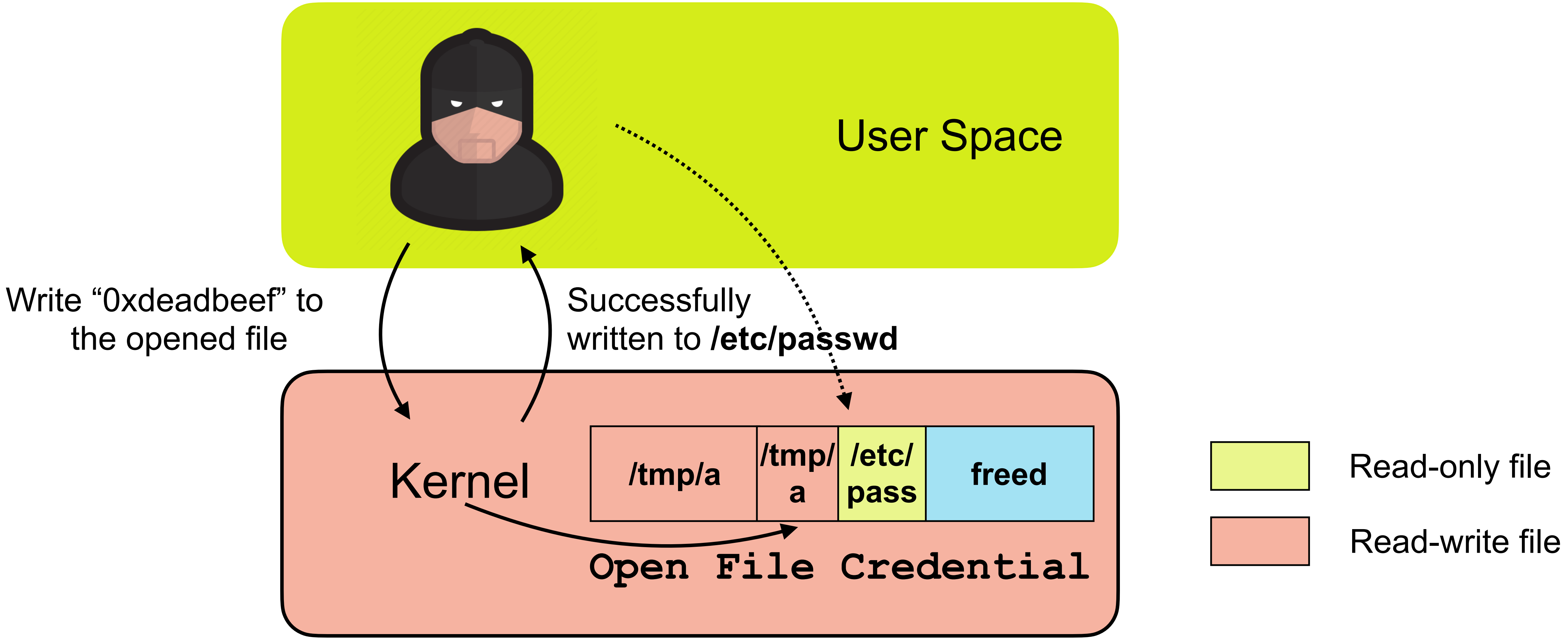
Attacking Open File Credential

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



Attacking Open File Credential

Result: Writing content to read-only files



Challenges

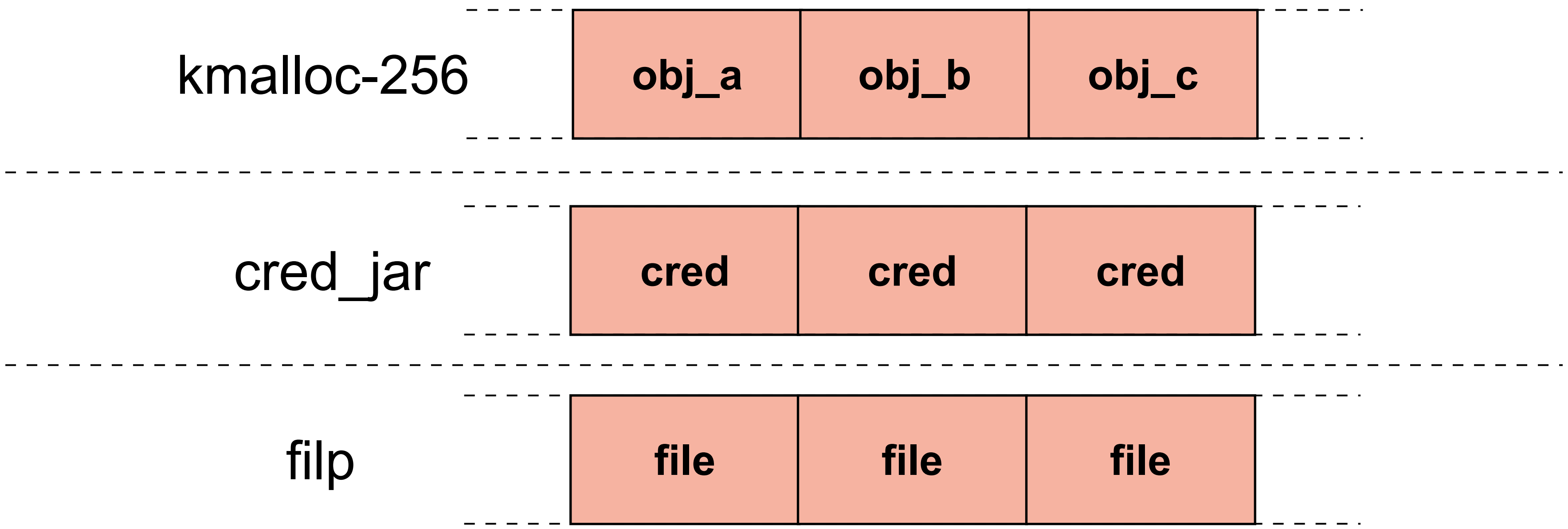
1. How to **free** credentials.
2. How to **allocate** *privileged* credentials as *unprivileged* users. (attacking *task* credentials)
3. How to finish attack in a **small** time window. (attacking *open file* credentials)

Challenges

1. How to **free** credentials.
2. How to **allocate** *privileged* credentials as *unprivileged* users. (attacking *task* credentials)
3. How to finish attack in a **small** time window. (attacking *open file* credentials)

Challenge 1: Free Credentials Invalidly

- Both ***cred*** and ***file*** object are in **dedicated** caches
- Most vulnerabilities happens in **generic** caches



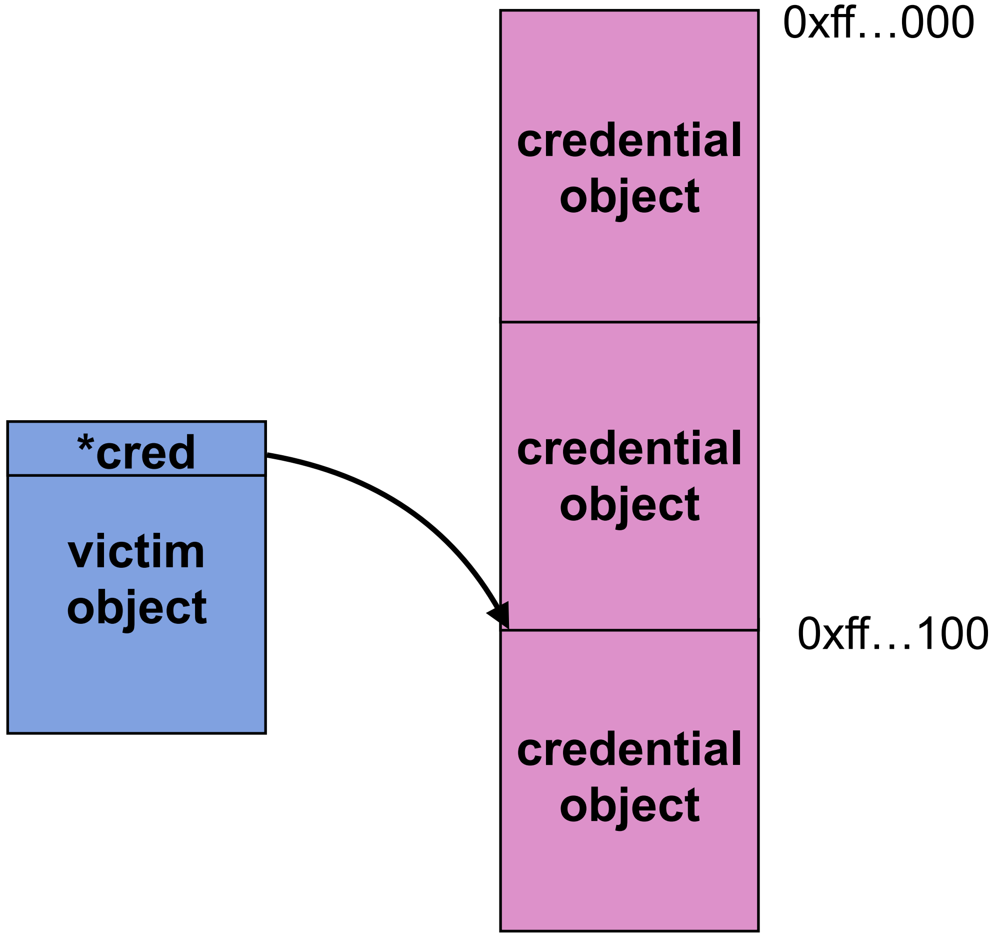
Challenge 1: Free Credentials Invalidly

- **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

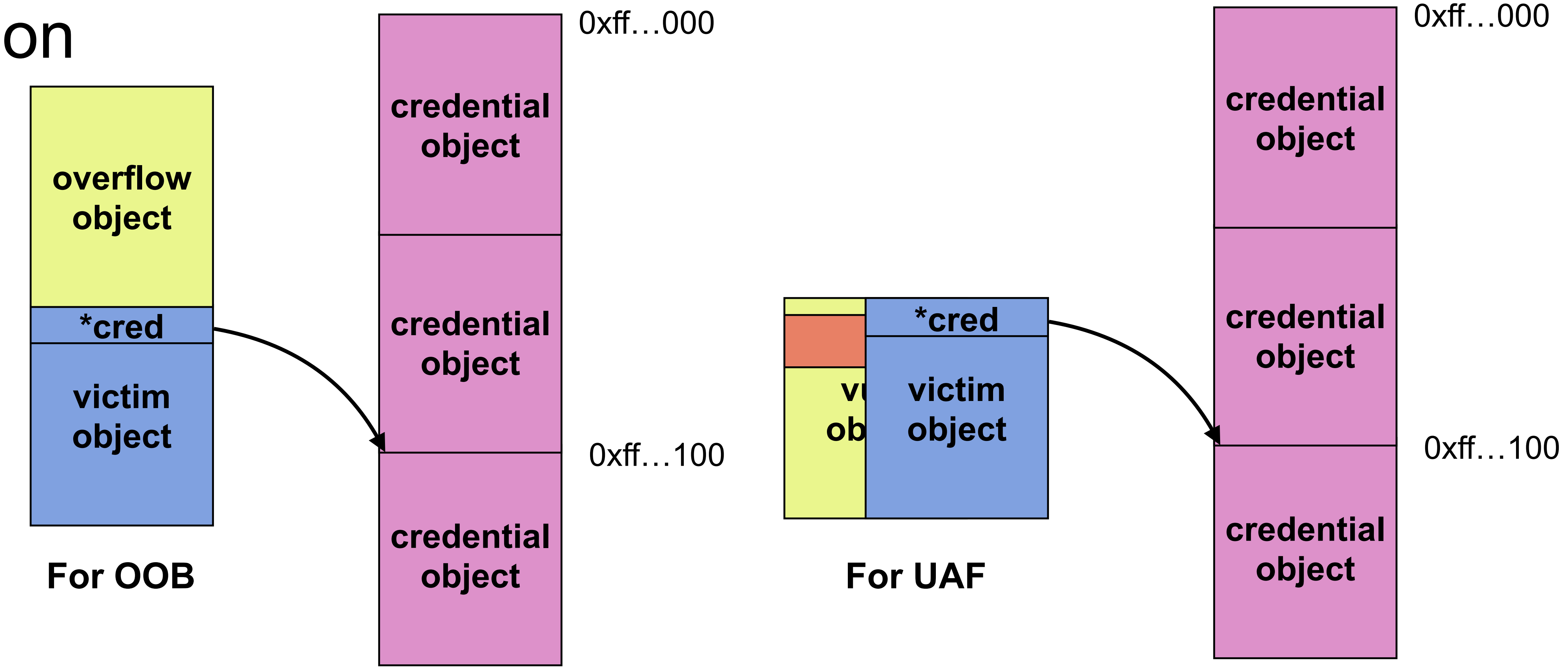
- Leverage victim objects with a reference to credentials




```
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;
```

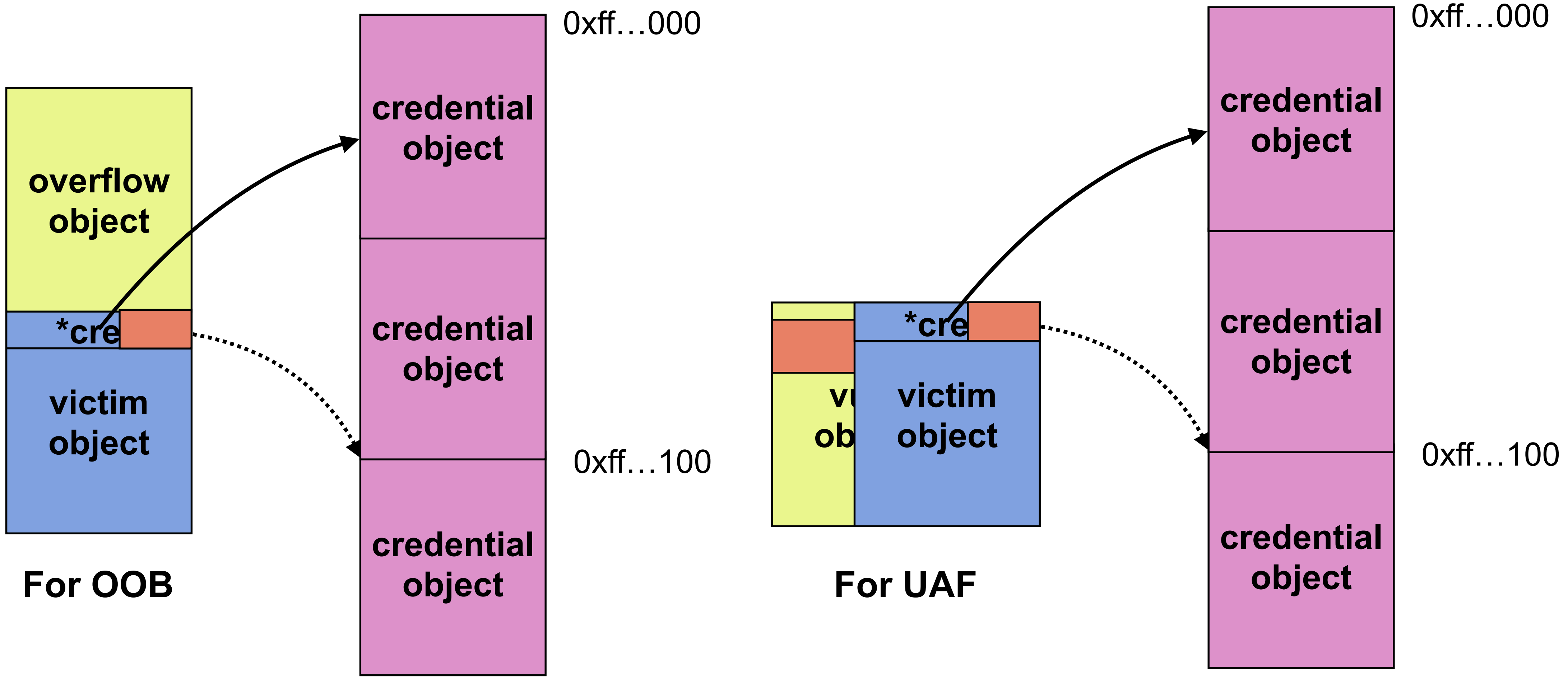
Pivoting Invalid-Write

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



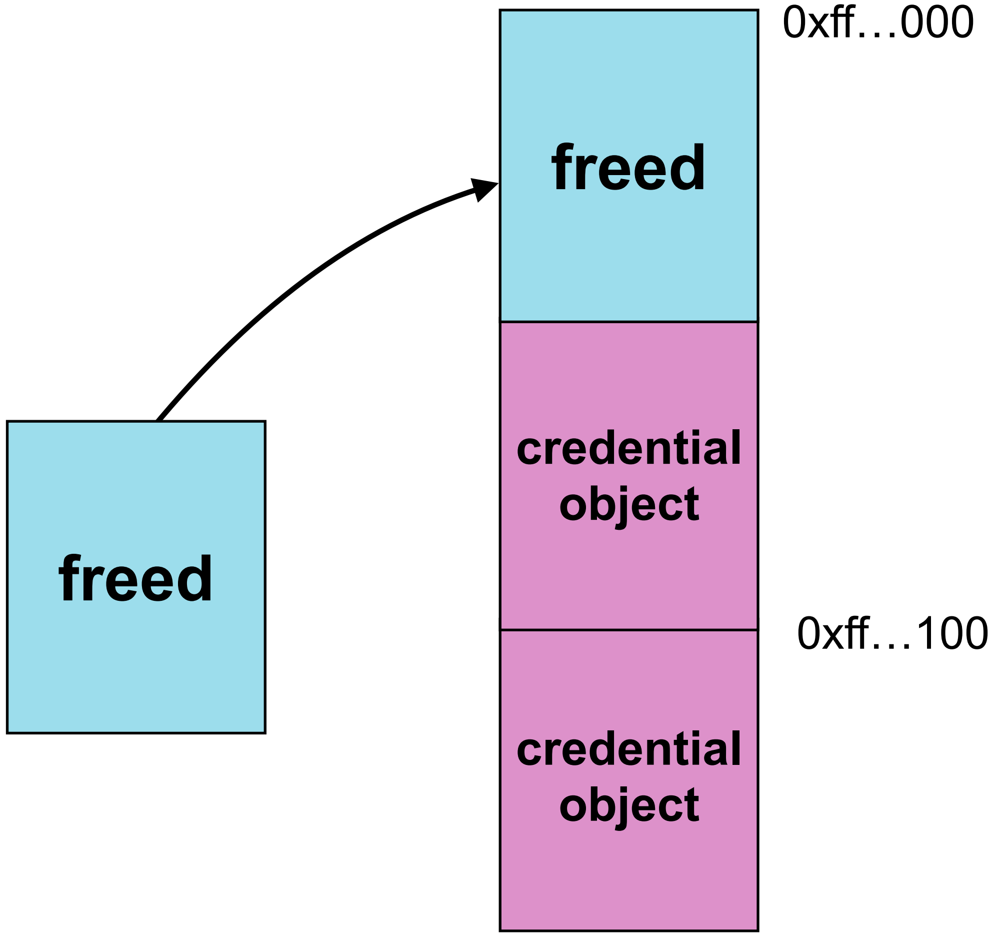
Pivoting Invalid-Write

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



Pivoting Invalid-Write

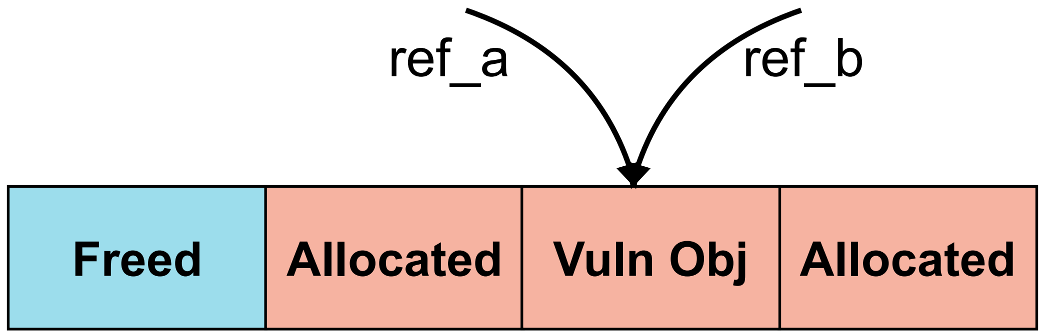
- Free the credential object when freeing the victim object



Pivoting Invalid-Free

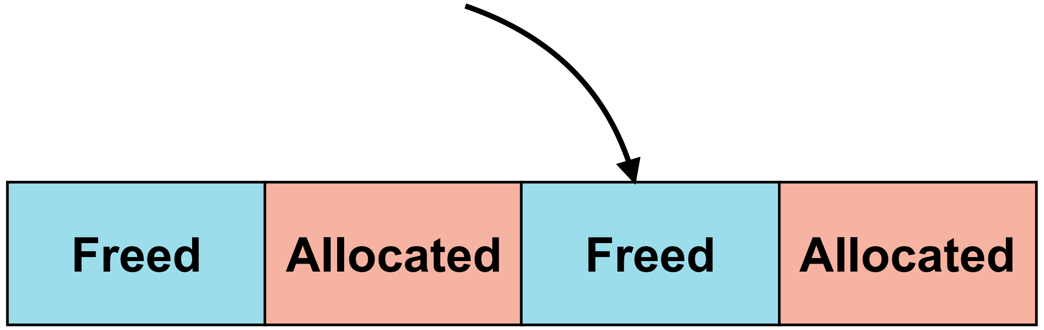
Pivoting Invalid-Free

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



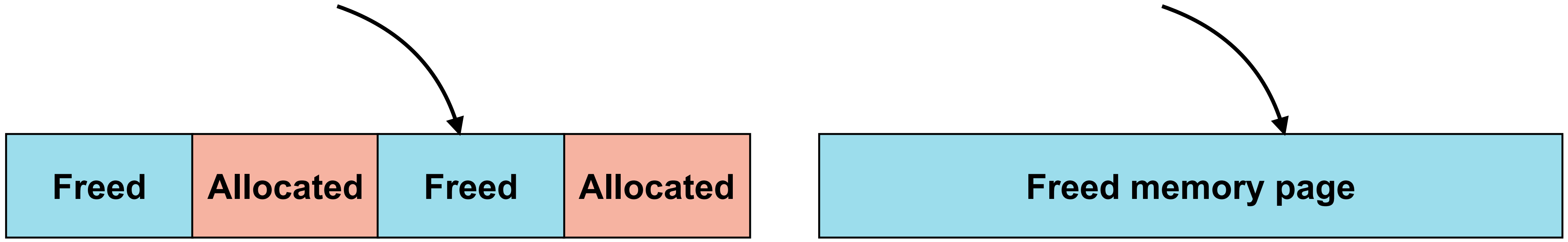
Vulnerable object in kernel memory

Pivoting Invalid-Free



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

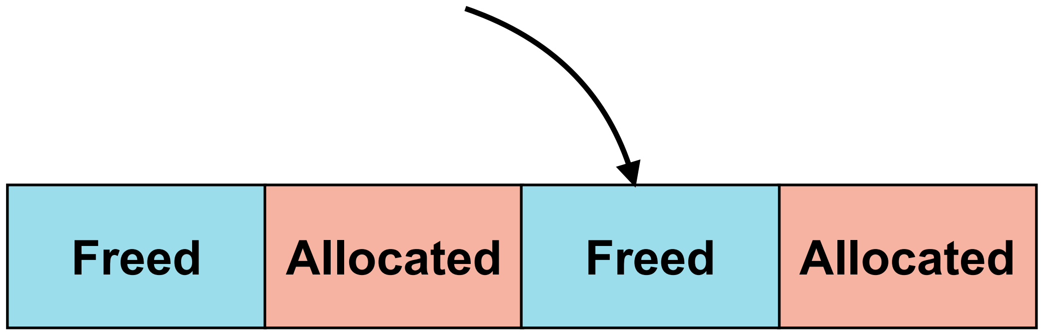
Pivoting Invalid-Free



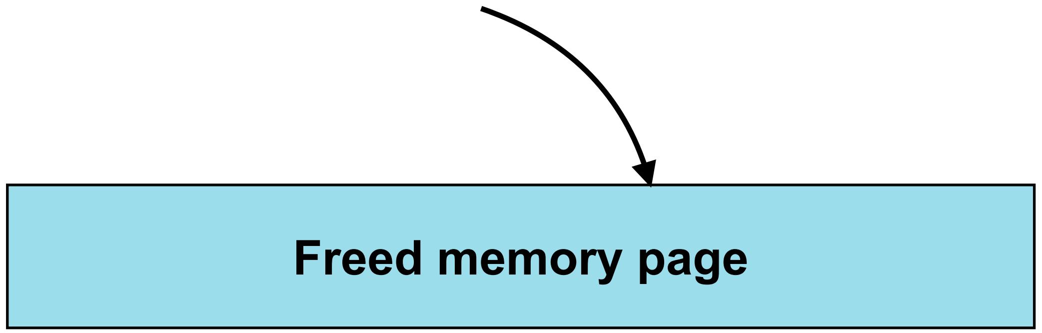
Step 1. Trigger the vuln, free the vuln object with one reference

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

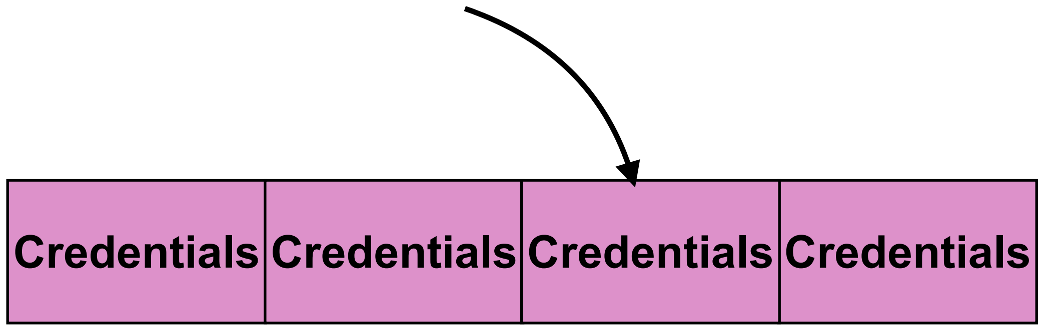
Pivoting Invalid-Free



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

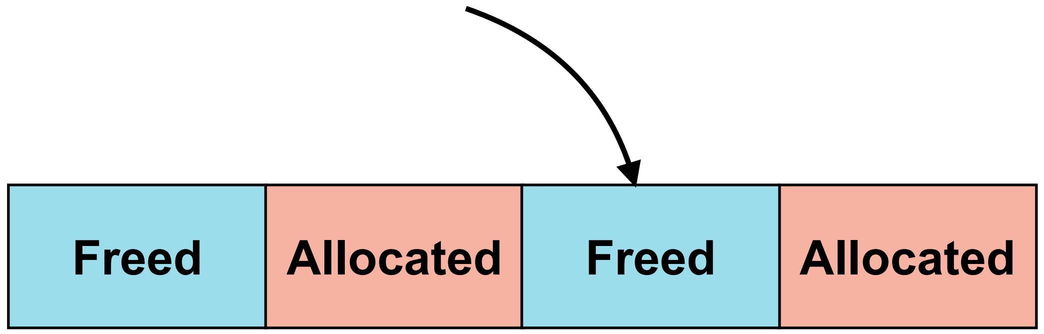


Step 2. Free the object in the memory cache to free the memory page

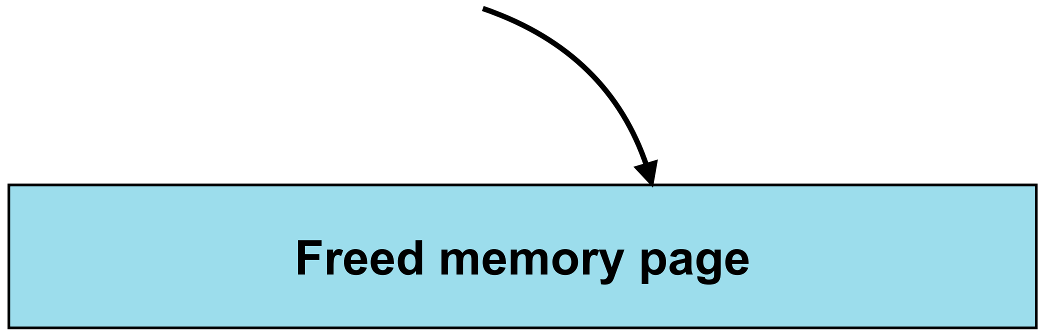


Step 3. *Allocate* credentials to reclaim the *freed* memory page (*Cross Cache Attack*)

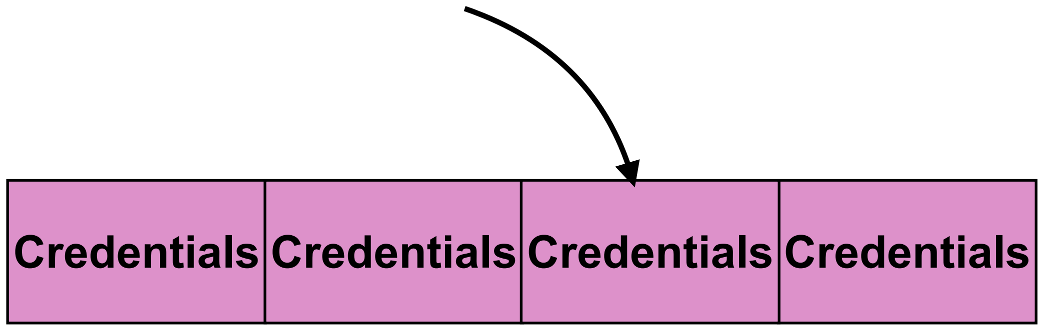
Pivoting Invalid-Free



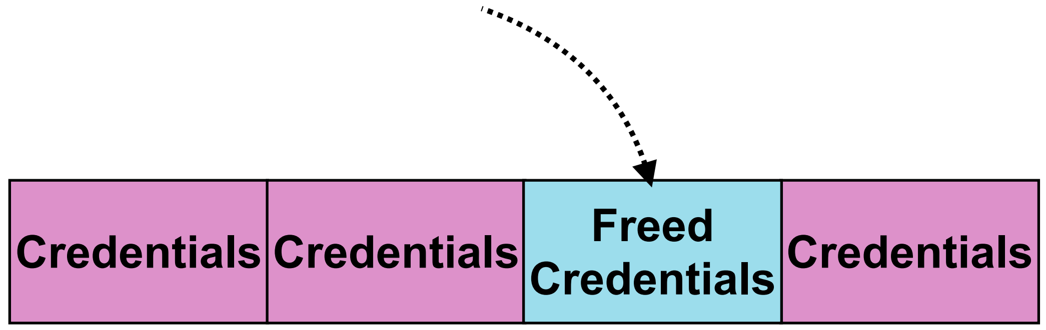
Step 1. Trigger the vuln, free the vuln object with one reference



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



Step 3. *Allocate* credentials to reclaim the *freed* memory page (*Cross Cache Attack*)



Step 4. Free the credentials with the left dangling reference

Challenges

1. How to **free** credentials.
2. How to **allocate** *privileged* credentials as *unprivileged* users. (attacking *task* credentials)
3. How to finish attack in a **small** time window. (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: Triggering Privileged Userspace Process**
  - Executables with root SUID (e.g. su, mount)
  - Daemons running as root (e.g. sshd)

Challenge 2: Allocating Privileged Task Credentials

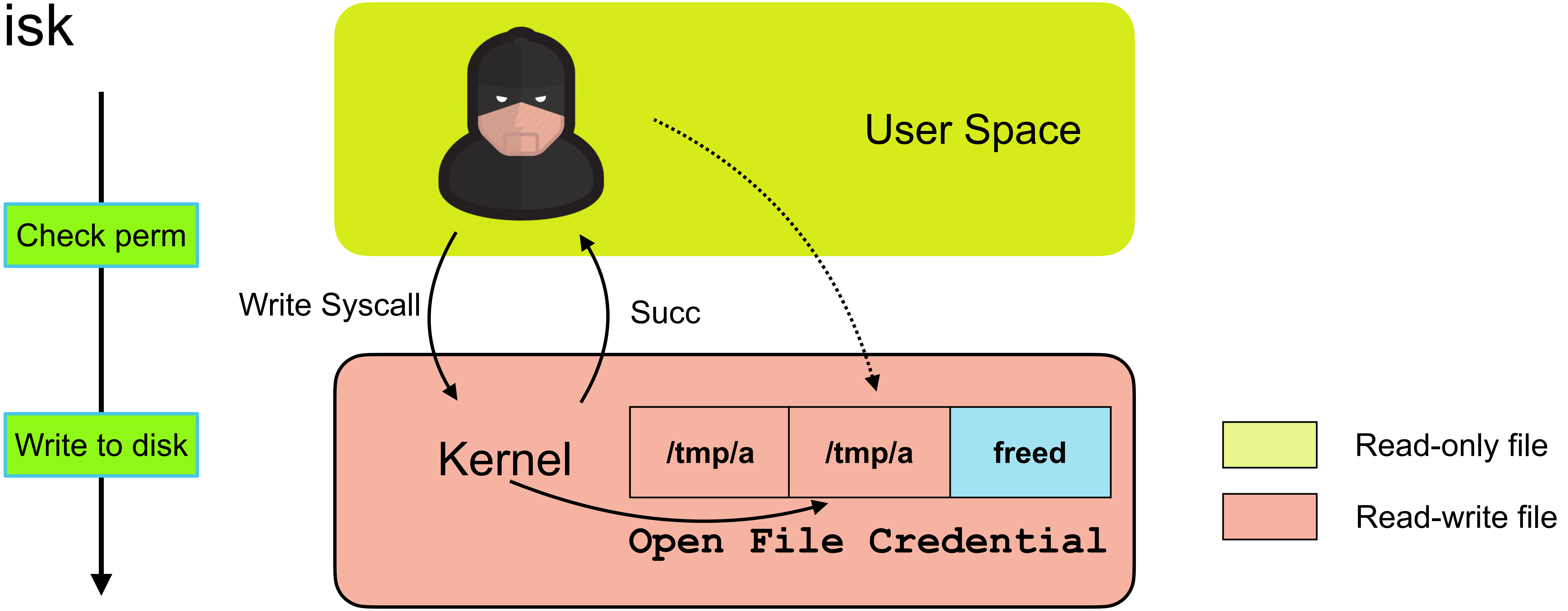
- **Solution I: Triggering Privileged Userspace Process**
  - Executables with root SUID (e.g. su, mount)
  - Daemons running as root (e.g. sshd)
- **Solution II: Triggering Privileged Kernel Thread**
  - Kernel Workqueue — spawn new workers
  - Usermode helper — load kernel modules from userspace

Challenges

1. How to **free** credentials.
2. How to **allocate** *privileged* credentials as *unprivileged* users. (attacking *task* credentials)
3. How to finish attack in a **small** time window. (attacking *open file* credentials)

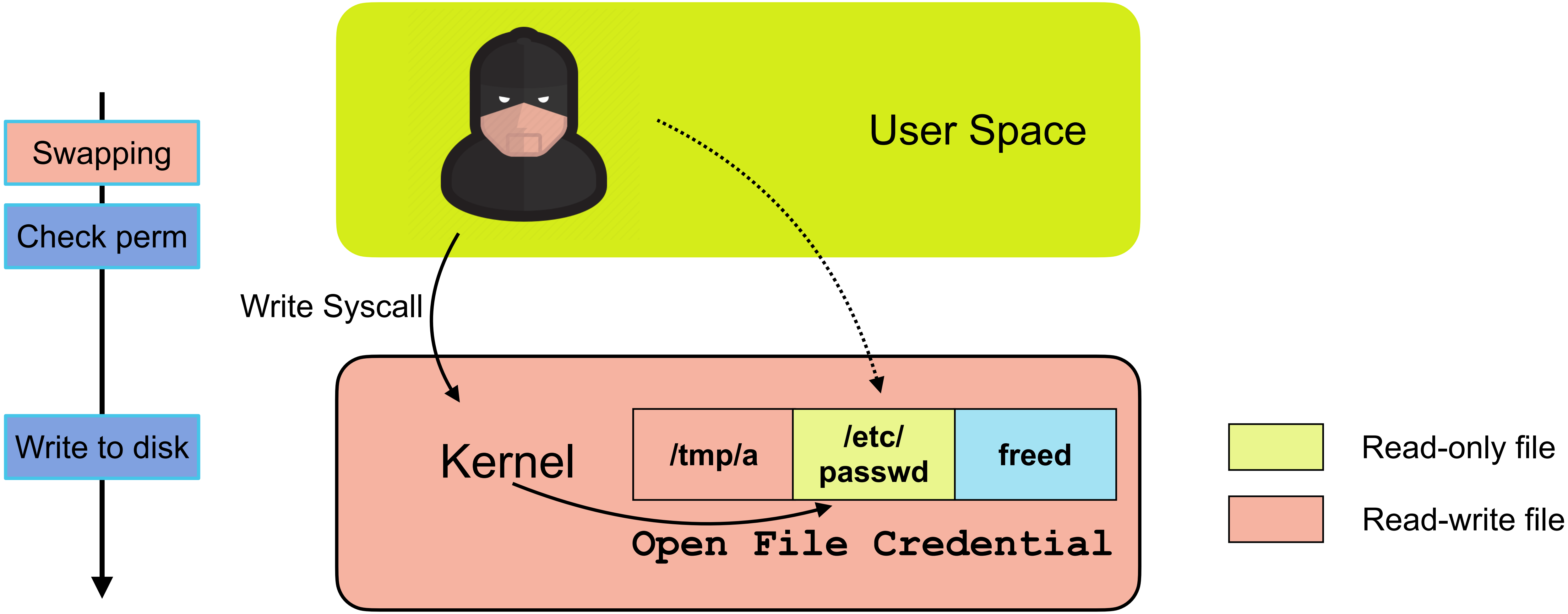
Challenge 3: Wining the race

- Kernel will examine the access permission before writing to the disk



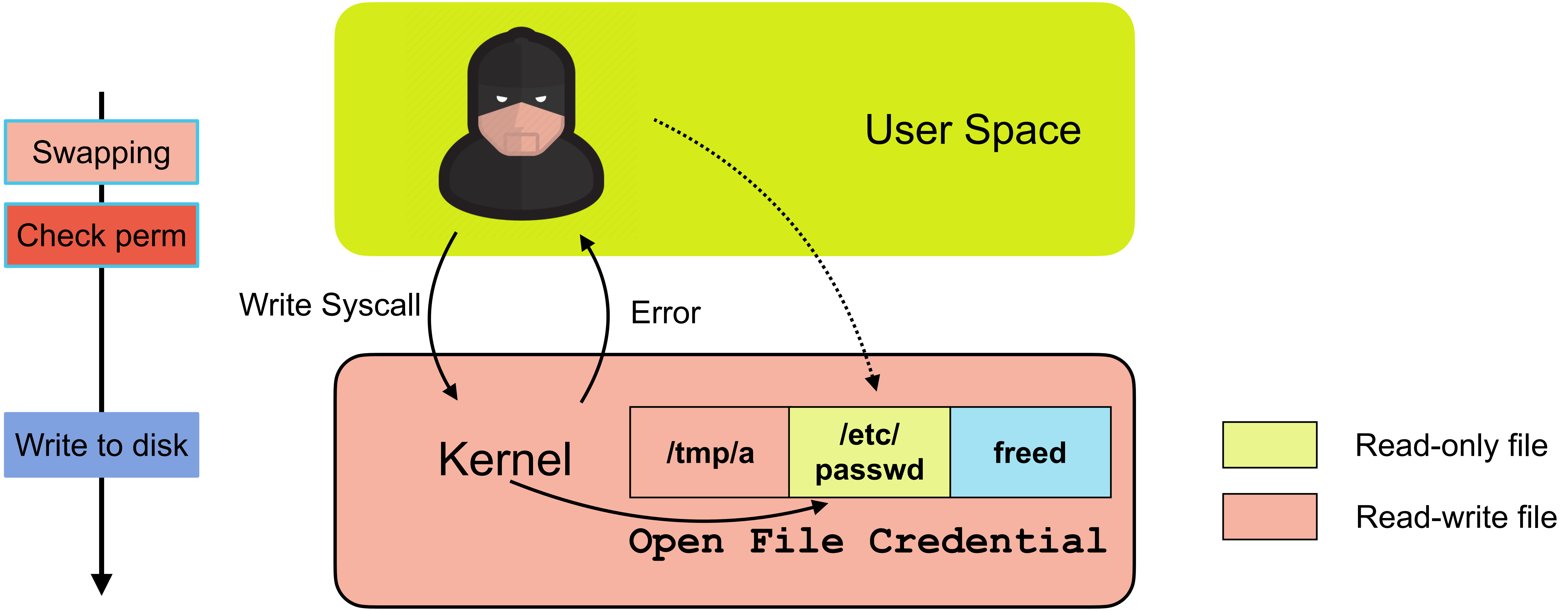
Challenge 3: Wining the race

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



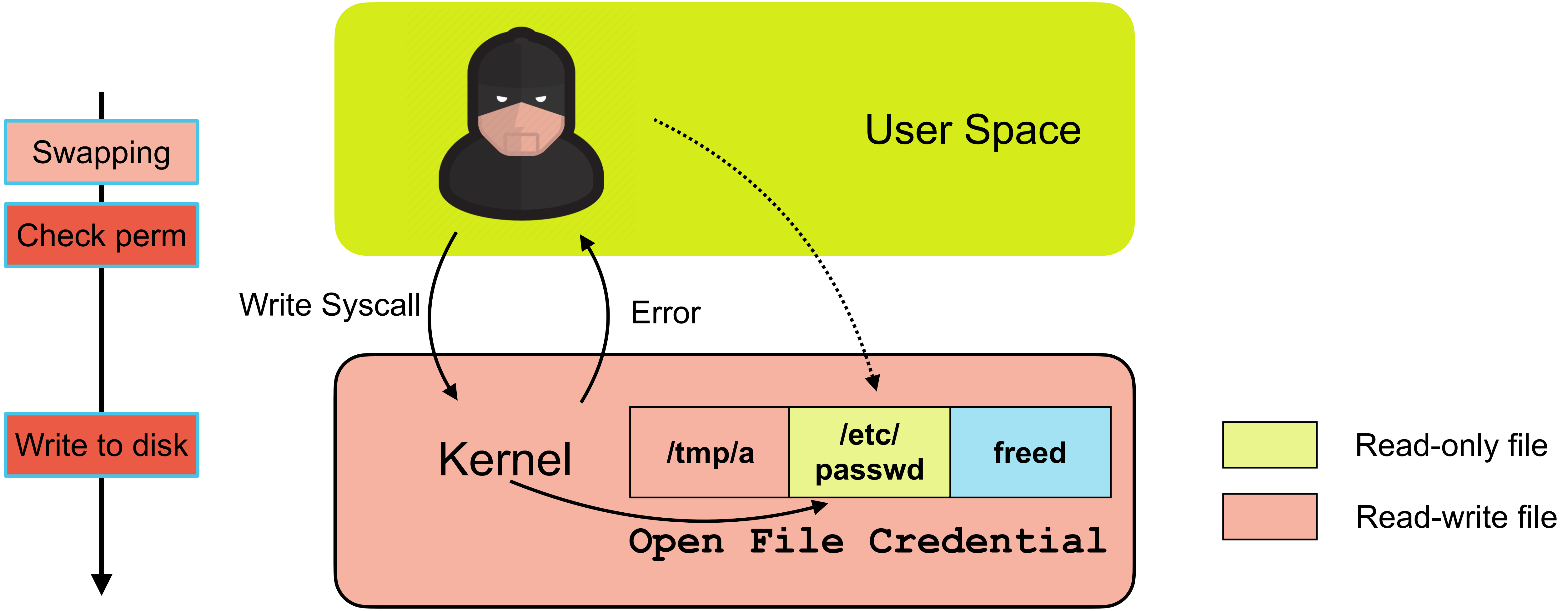
Challenge 3: Wining the race

- The swap of *file* object happens before <u>*permission check*</u>



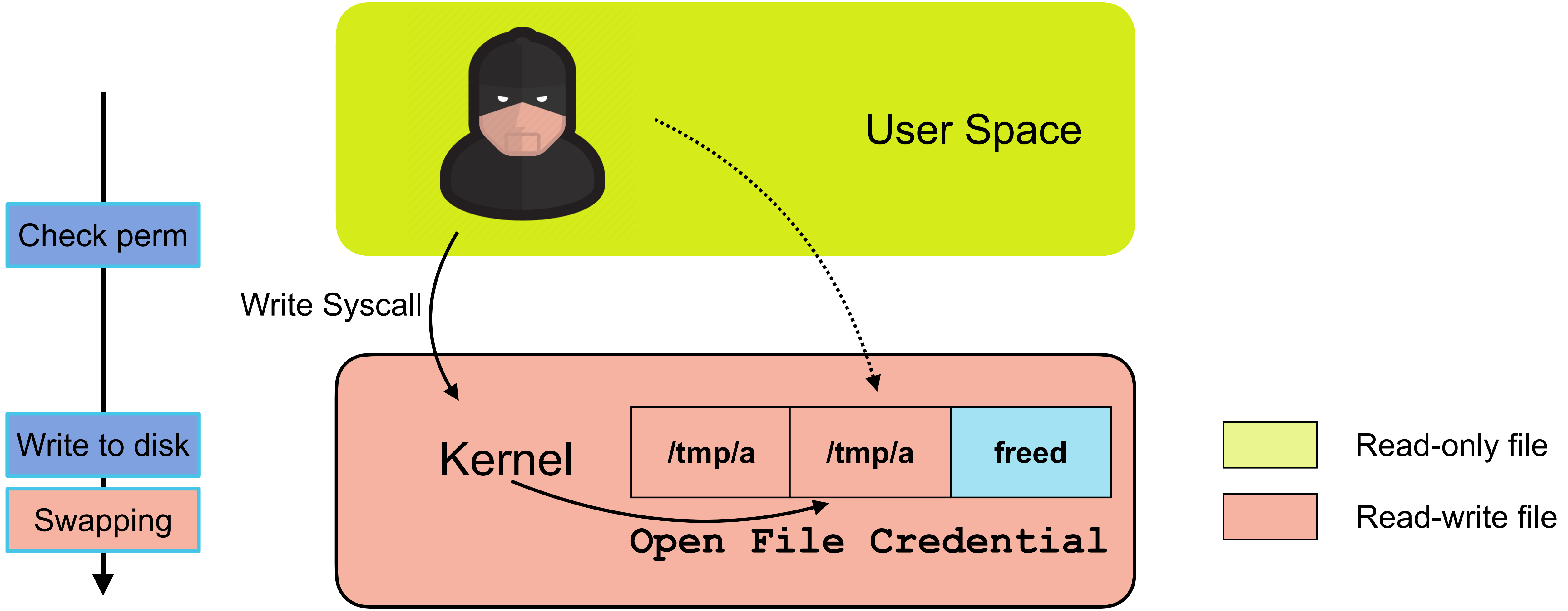
Challenge 3: Wining the race

- The swap of *file* object happens before <u>*permission check*</u>



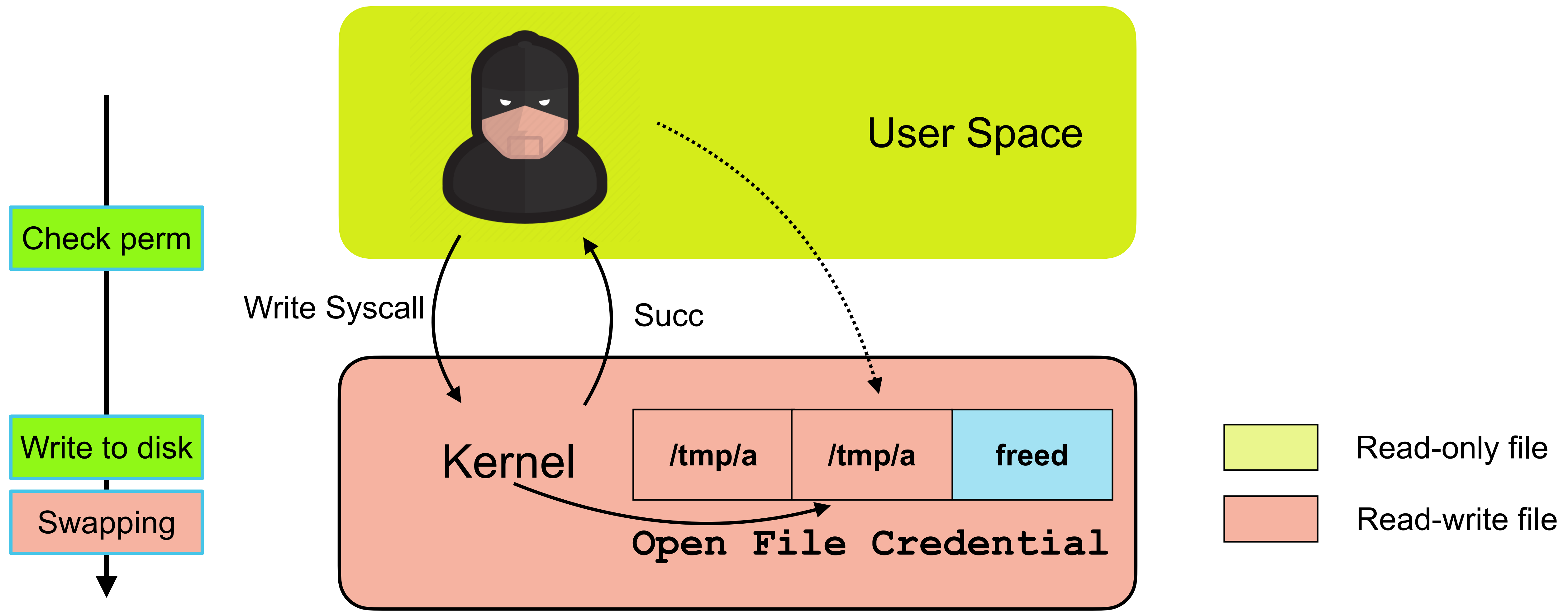
Challenge 3: Wining the race

- The swap of *file* object happens after <u>*file write*</u>.



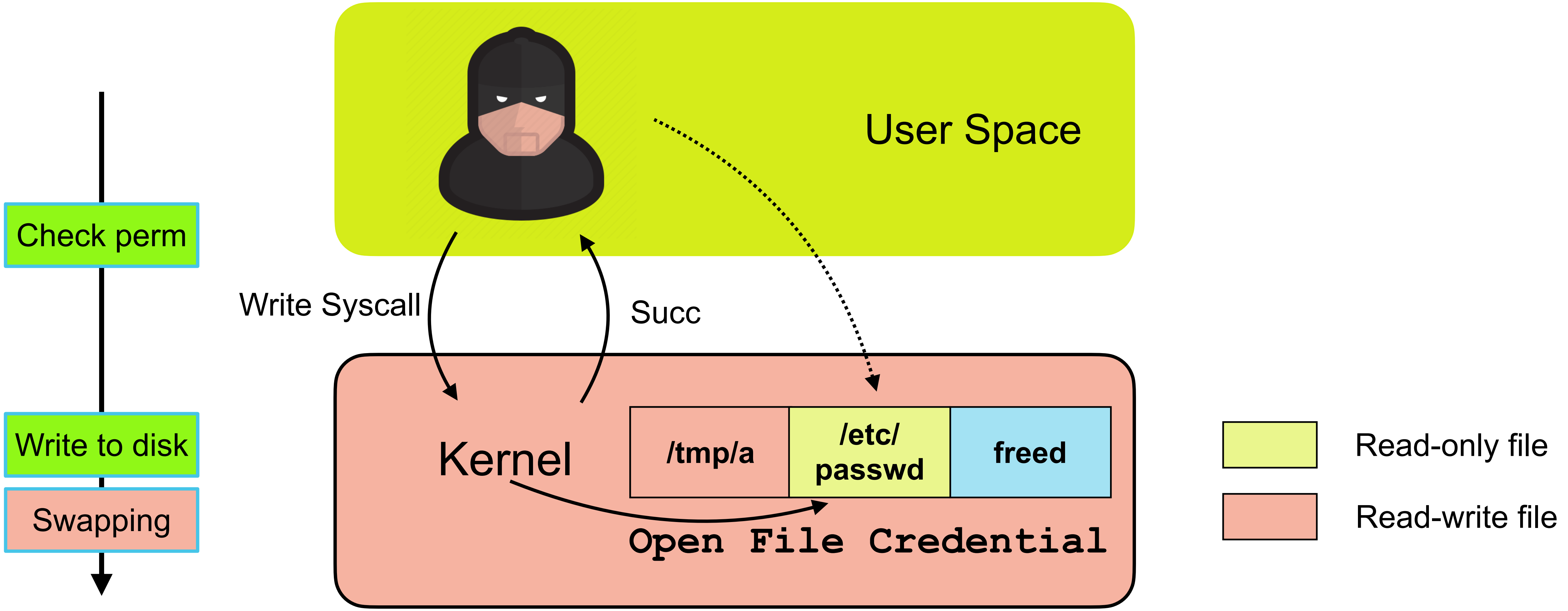
Challenge 3: Wining the race

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



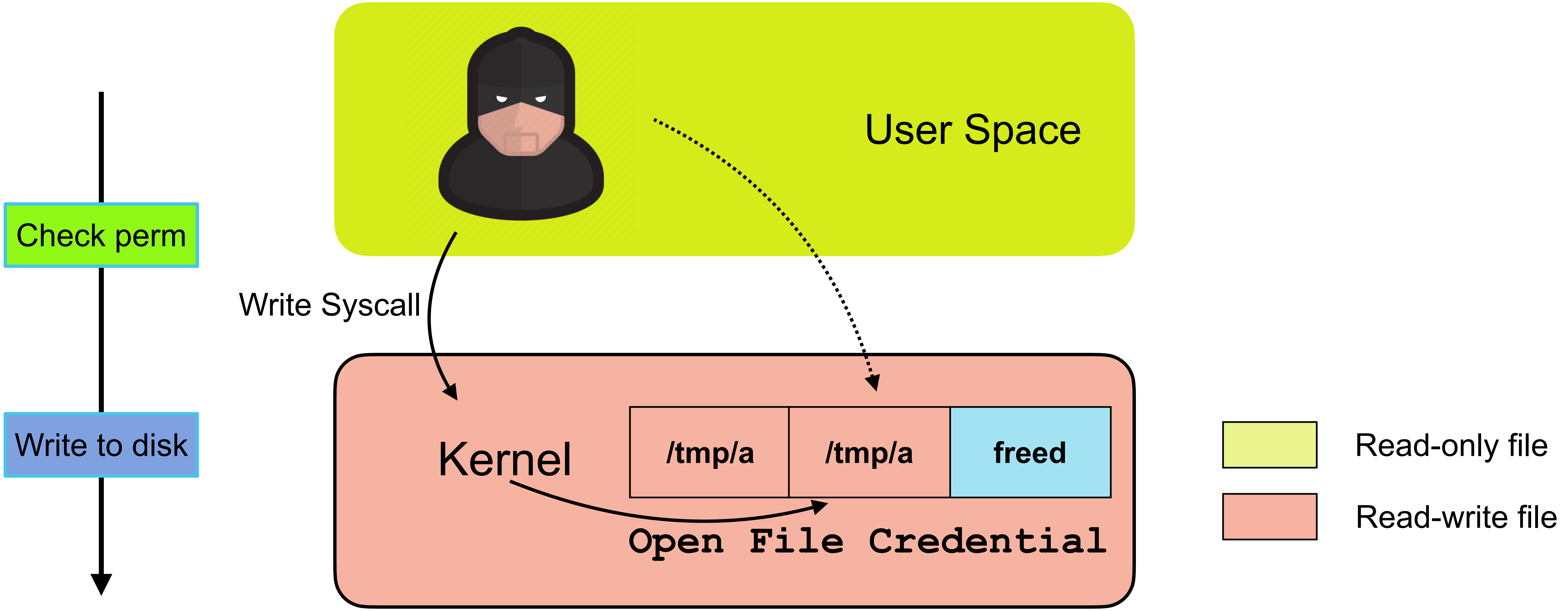
Challenge 3: Wining the race

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



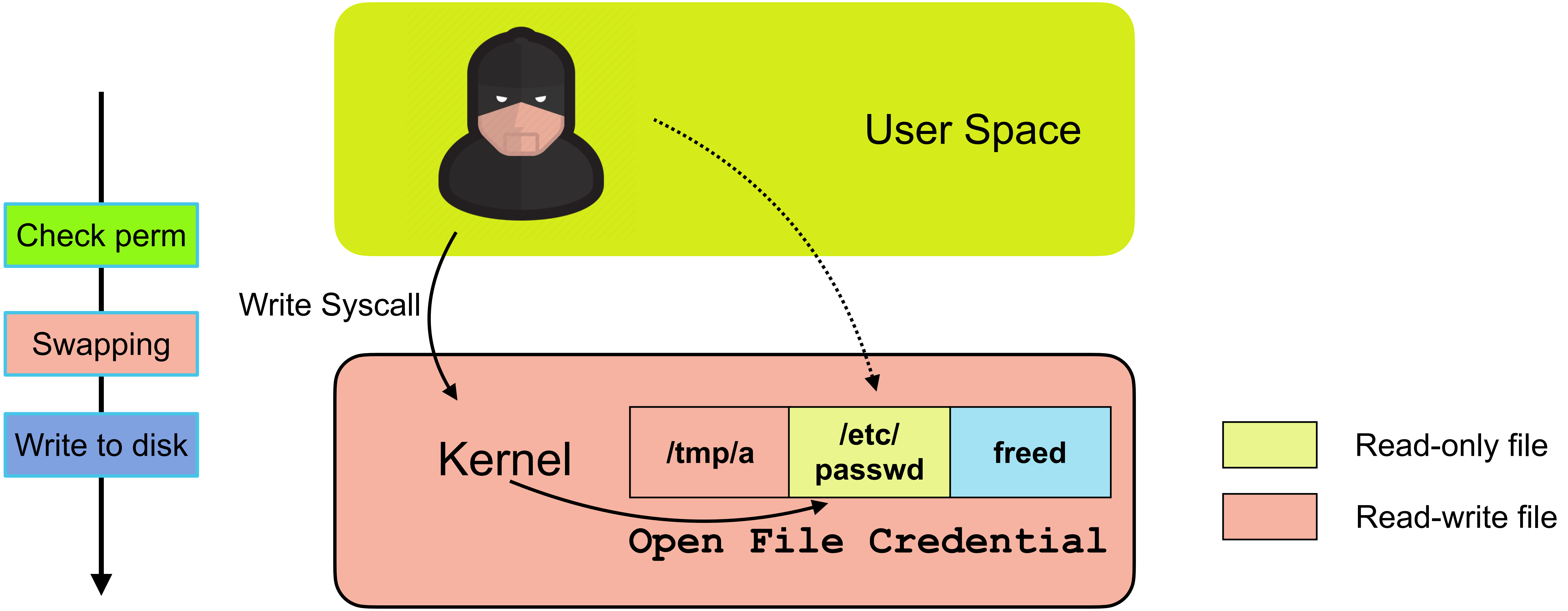
Challenge 3: Wining the race

- The swap happens in between *permission check* and *file write*



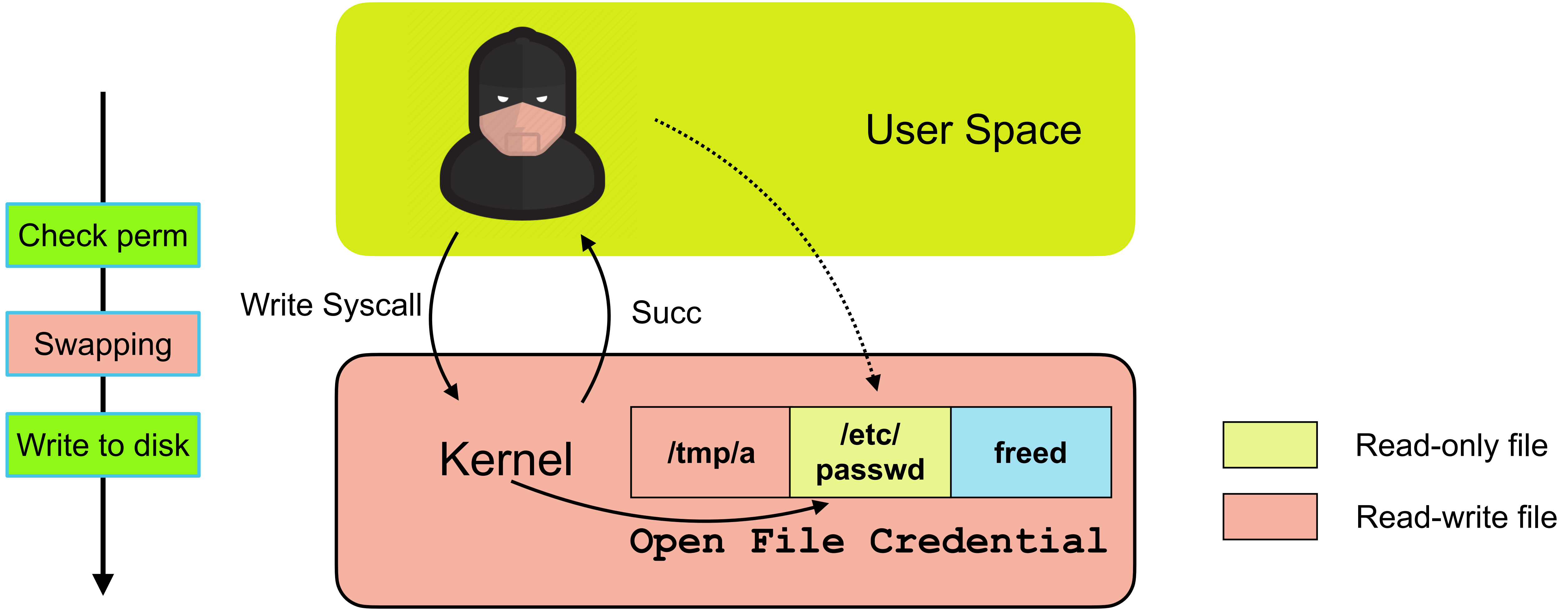
Challenge 3: Wining the race

- The swap happens in between *permission check* and *file write*



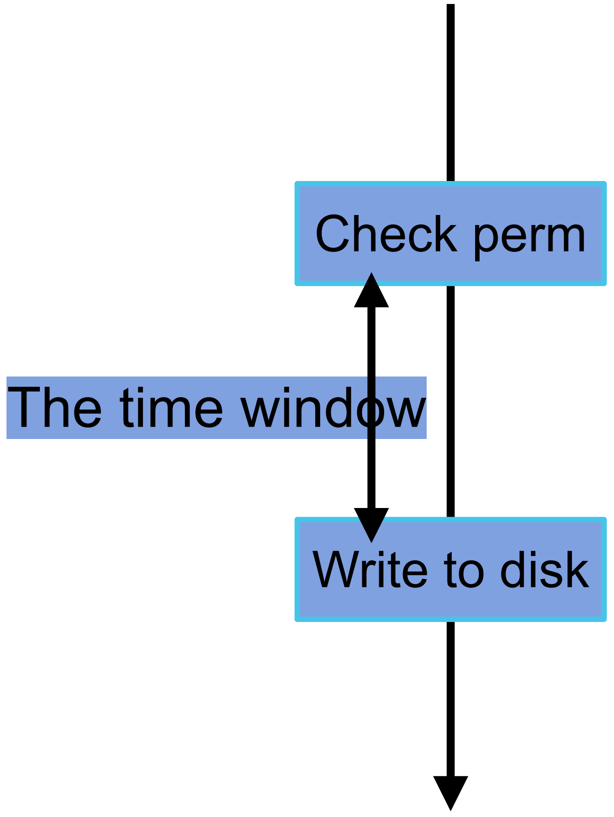
Challenge 3: Wining the race

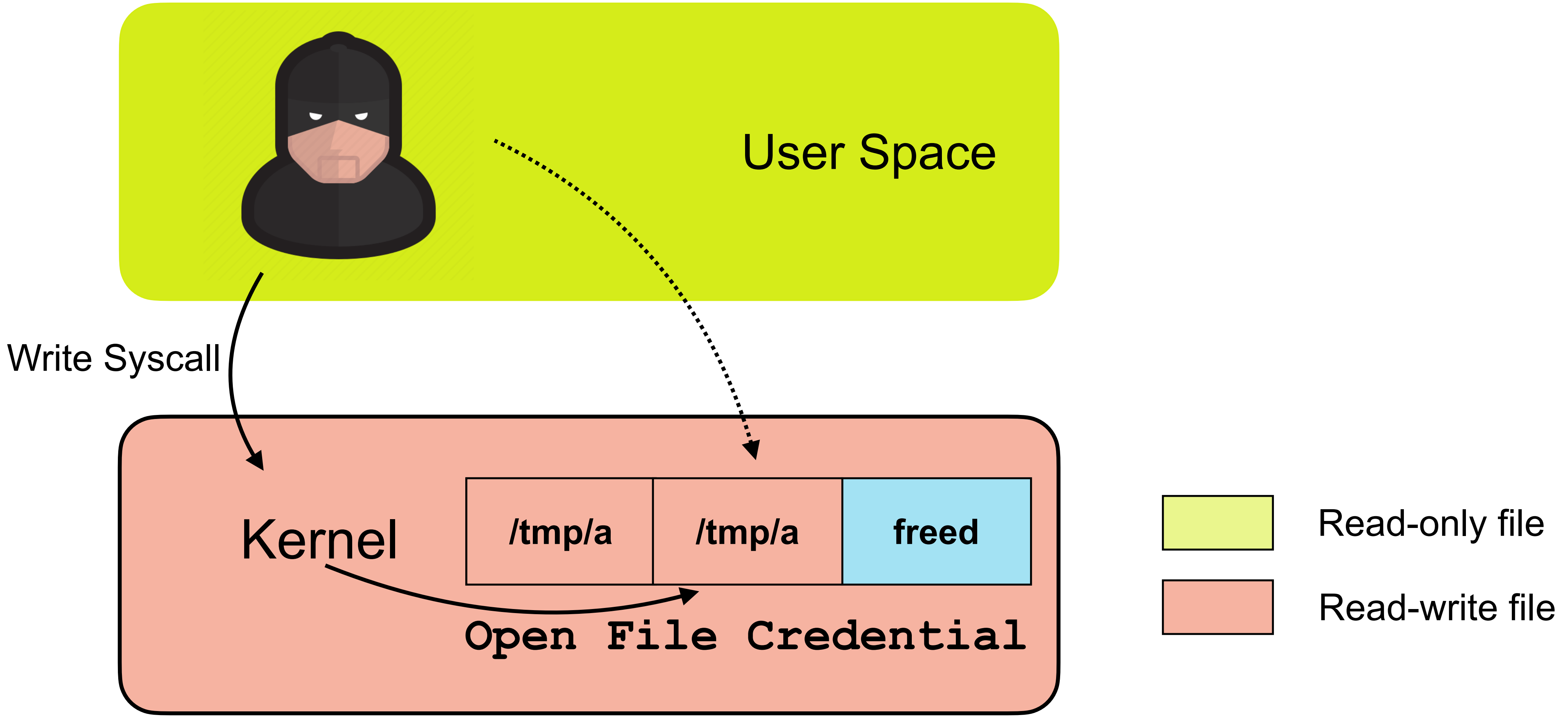
- The swap happens in between *permission check* and *file write*



Challenge 3: Wining the race

- The swap must happen after *permission check* and before *file write*





Challenge 3: Wining the race

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

Solution I: Userfaultfd & FUSE

- **Pause at *`import_iovec`* before v*4.13***
  - *`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 v*4.13***
  - *`import_iovec`* copies userspace memory
  - Used in Jann Horn’s exploitation for [CVE-2016-4557]
  - *Dead* after v*4.13*

Solution I: Userfaultfd & FUSE

- `vfs_writev` after v*4.13*

```
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: Wining the race

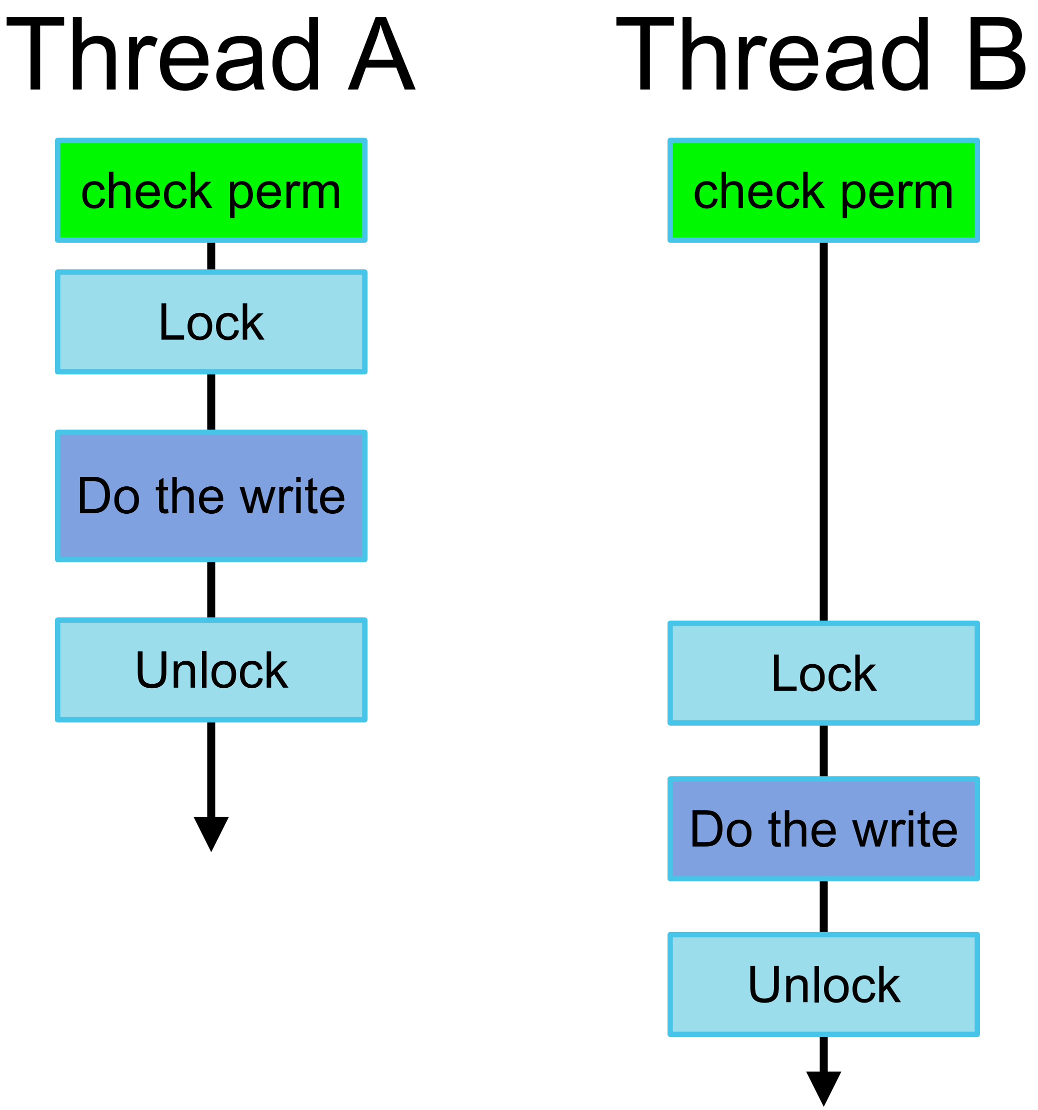
- **Solution I: Extending with Userfaultfd & FUSE**
  - *Pause* kernel execution when accessing userspace memory
  - Userfaultfd & FUSE might not be available
- **Solution II: Extending with file lock**
  - Pause kernel execution with lock

Solution II: File Lock

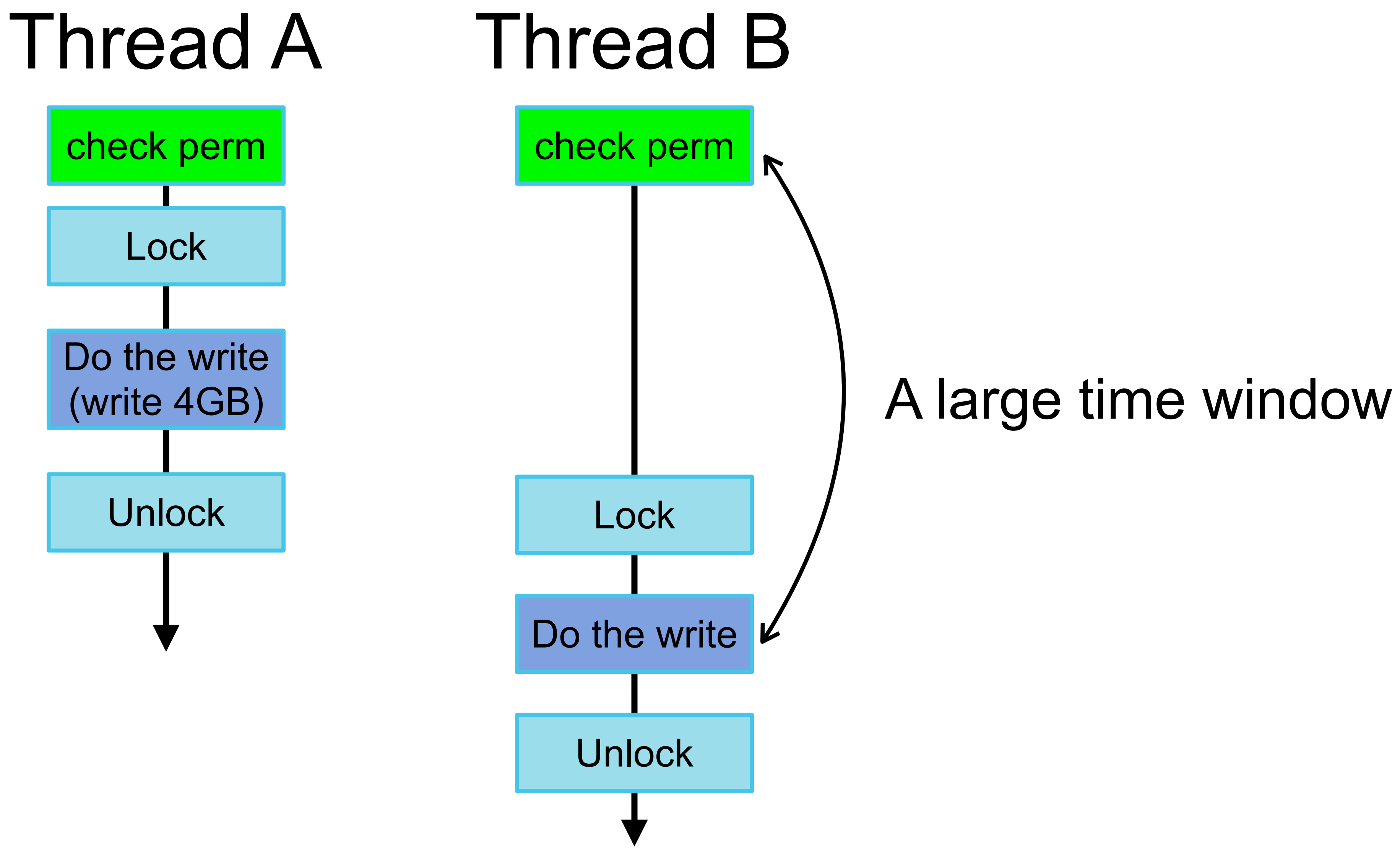
- A lock of the *inode* of the file
- Lock the file when it is being writing to

```
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



Solution II: File Lock



Demo Time!

CVE-2021-4154

Centos 8 and Ubuntu 20

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

Android Kernel with CFI enabled*

```
oriole:/data/local/tmp $
```

* access check removed for demonstration

Real-World Impact

- **CVE-2021-4154**
  - Received rewards from Google's KCTF
  - The exploit works across kernel v4.18 ~ v5.10
- **CVE-2022-2588**
  - Pwn2own exploitation
  - The exploit works across kernel v3.17 ~ v5.19
- **CVE-2022-20409**
  - Received rewards from Google's KCTF and Android
  - The exploit works on both Android and generic Linux kernel

Advantages of DirtyCred

- **Simple but effective**
  - Shorter exploit chain with fewer steps
- **No effective mitigation**
  - A new exploitation path, can bypass AUTOSLAB
  - No need to deal with KASLR, KCFI, KPTI, SMAP/SMEP
- **Exploitation friendly**
  - Make your exploit **universal**!

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 (privileged credentials will be *vmalloc*-ed)

Code is available at https://github.com/markakd/DirtyCred

Overhead of The Defense

Benchmark	Vanilla	Hardened	Overhead
Phoronix			
Apache (Reqs/s)	28603.29	29216.48	-2.14%
Sys-RAM (MB/s)	10320.08	10181.91	1.34%
Sys-CPU (Events/s)	4778.41	4776.69	0.04%
FFmpeg(s)	7.456	7.499	0.58%
OpenSSL (Byte/s)	1149941360	1150926390	-0.09%
OpenSSL (Sign/s)	997.2	993.2	0.40%
PHPBench (Score)	571583	571037	0.09%
PyBench (ms)	1303	1311	0.61%
GIMP (s)	12.357	12.347	-0.08%
PostMark (TPS)	5034	5034	0%
LMBench			
Context Switch (ms)	2.60	2.57	-1.15%
UDP (ms)	9.2	9.26	0.65%
TCP (ms)	12.75	12.73	-0.16%
10k File Create (ms)	13.8	14.79	7.17%
10k File Delete (ms)	6.35	6.62	4.25%
Mmap (ms)	80.23	81.91	2.09%
Pipe (MB/s)	4125.3	4028.9	2.34%
AF Unix (MB/s)	8423.5	8396.7	0.32%
TCP (MB/s)	6767.4	6693.3	1.09%
File Reread (MB/s)	8380.43	8380.65	0%
Mmap Reread (MB/s)	15.7K	15.69K	0.06%
Mem Read (MB/s)	10.9K	10.9K	0%
Mem Write (MB/s)	10.76K	10.77K	-0.09%

Takeaways

- A new exploitation concept — DirtyCred
- Principled approaches to different challenges
- A way to produce *Universal* kernel exploits
- Effective defense with negligible overhead

Zhenpeng Lin (@Markak_)

https://zplin.me

zplin@u.northwestern.edu

Logo comes from @sirdarckcat