DirtyCred: Escalating Privilege in Linux Kernel

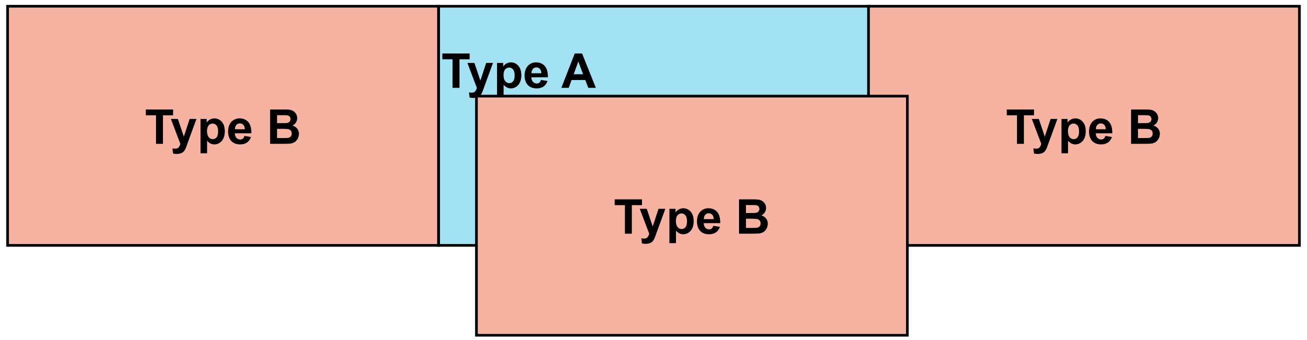
Zhenpeng Lin, Yuhang Wu, Xinyu Xing

Northwestern University

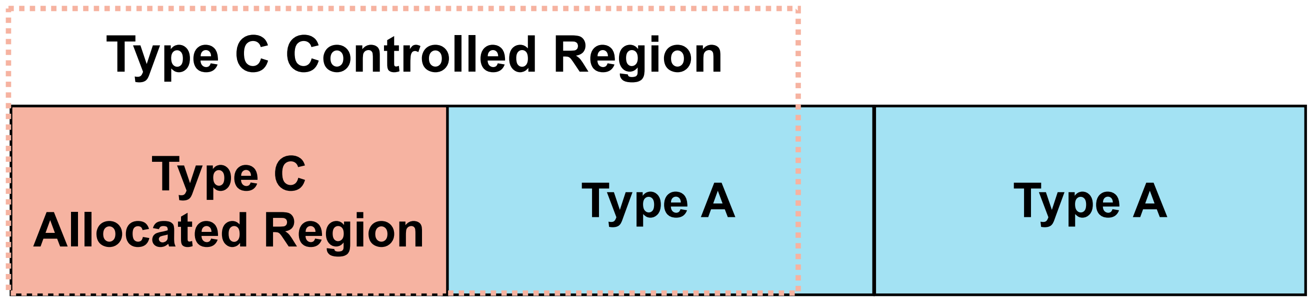
ACM CCS 2022

How Researchers Exploit Kernel Vulns

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



(a) Type confusion between Type A and B

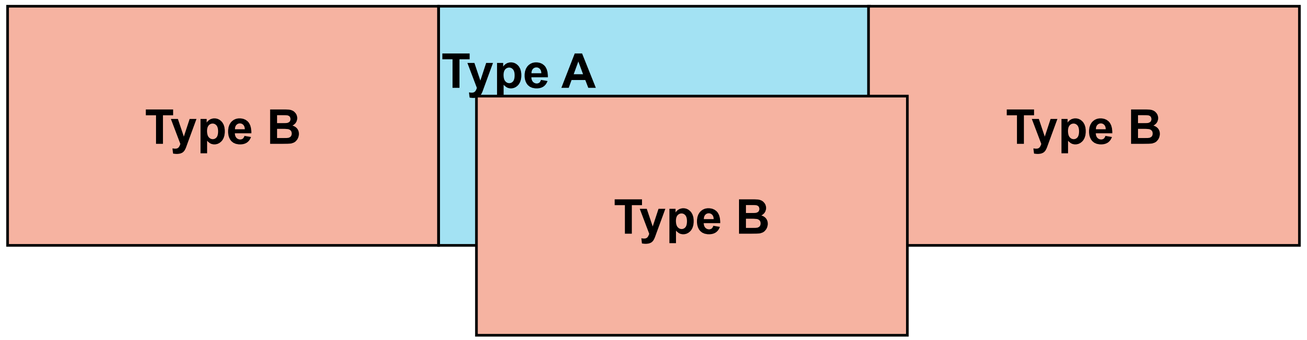


(b) Partial overlap between Type C and A

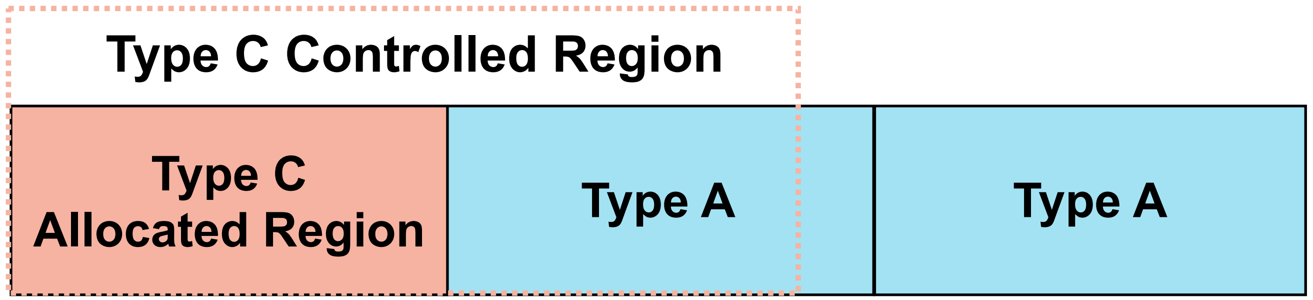
How Researchers Exploit Kernel Vulns

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

Obtain Primitives



(a) Type confusion between Type A and B



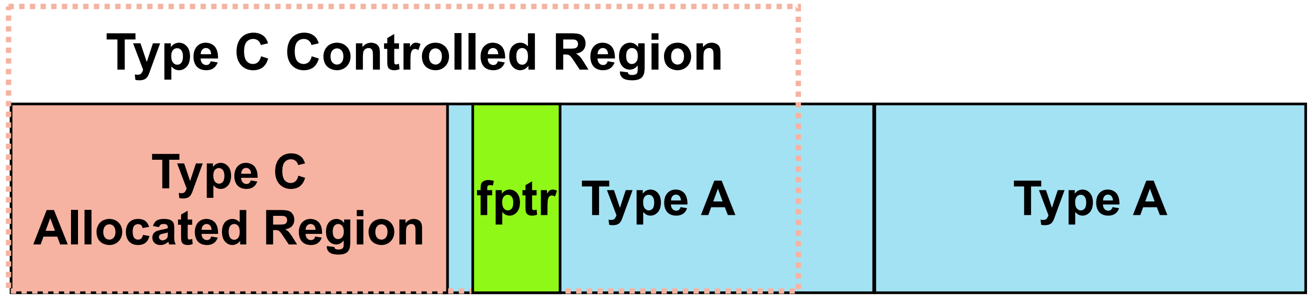
(b) Partial overlap between Type C and A

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***

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

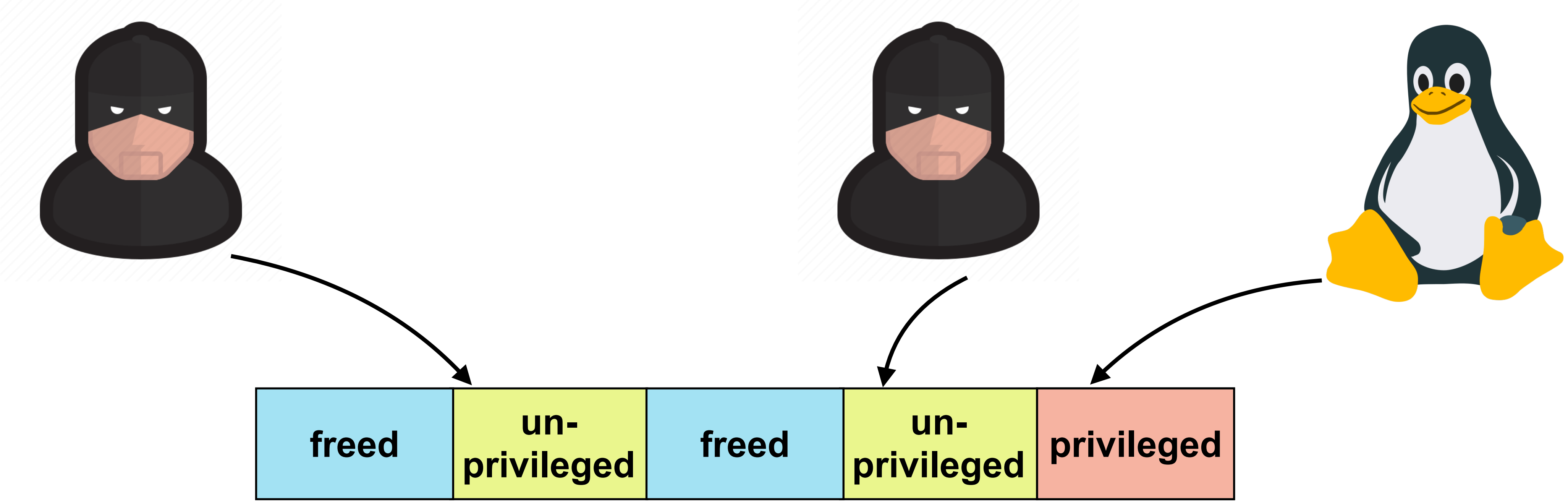
Task Credential

- **`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 */
```

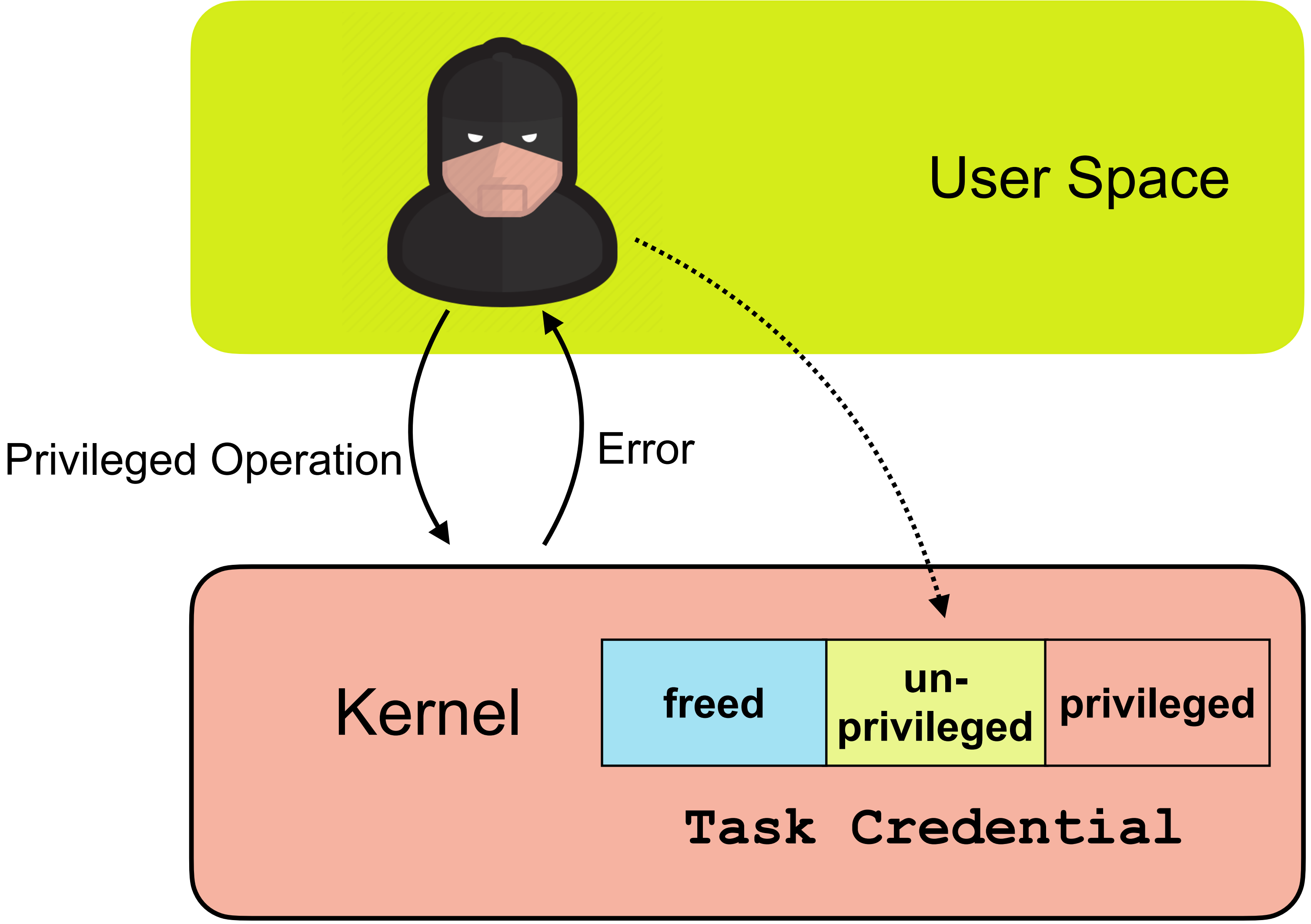
Task Credential

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



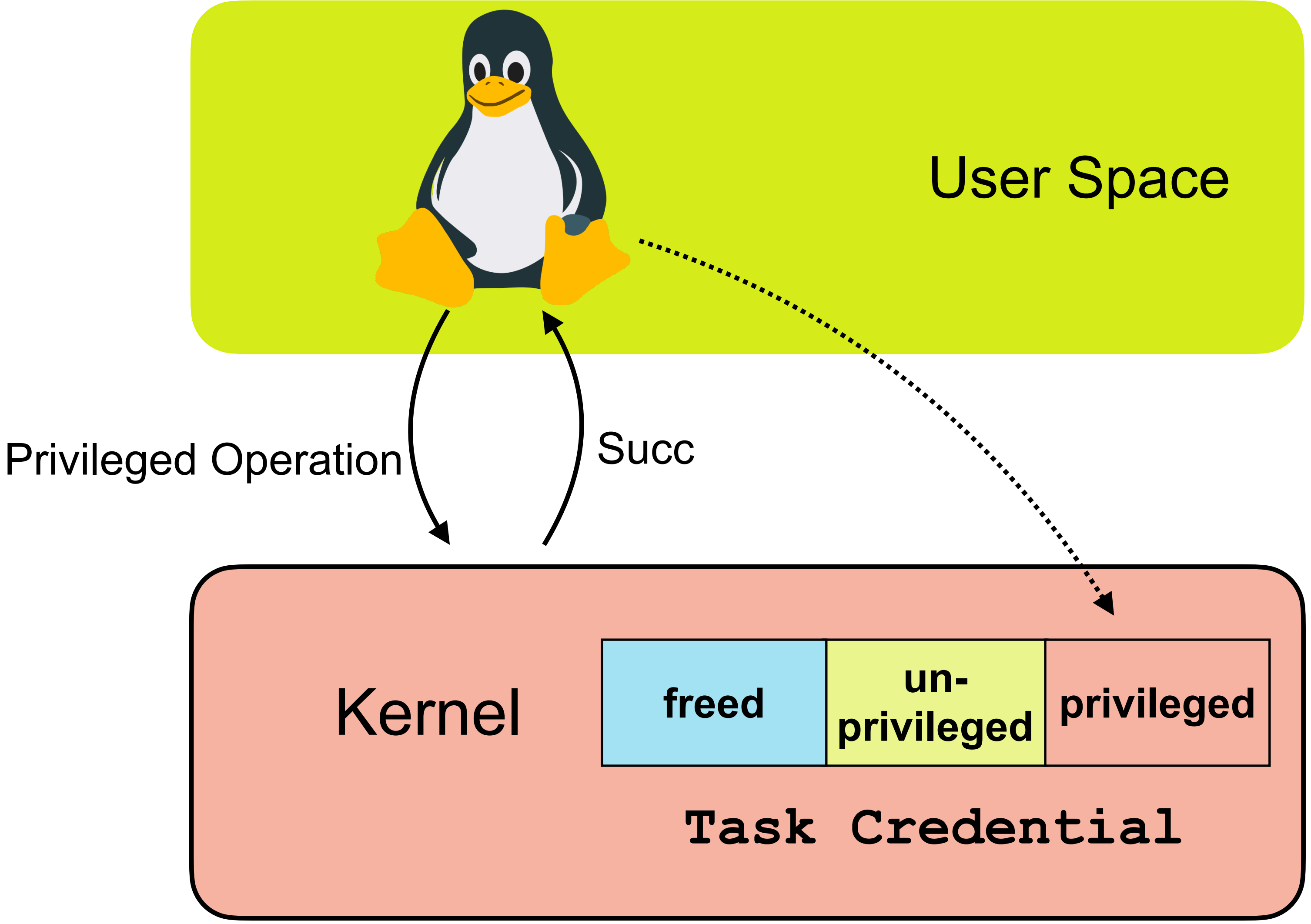
`Task Credential` on kernel heap

How Linux Kernel Uses Task Credential



Northwestern

How Linux Kernel Uses Task Credential



Open File Credential

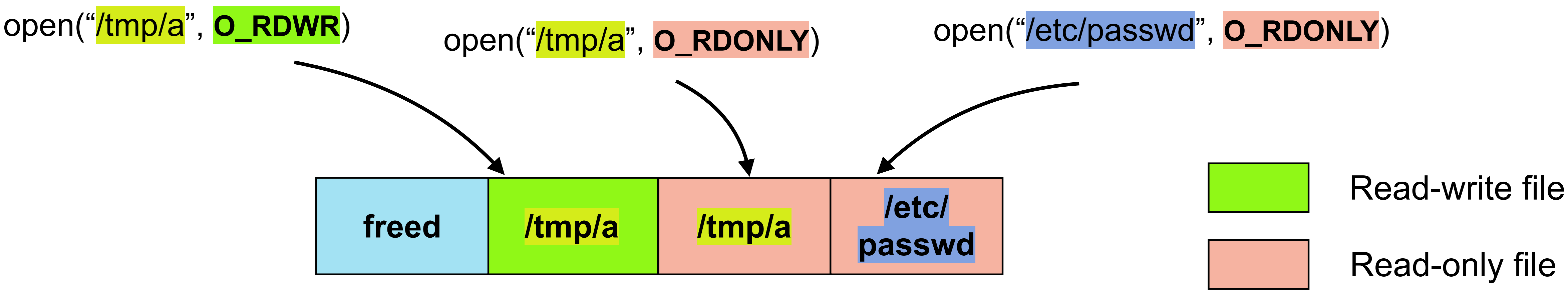
- **`Struct file`** in Linux kernel's implementation

```
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;
```

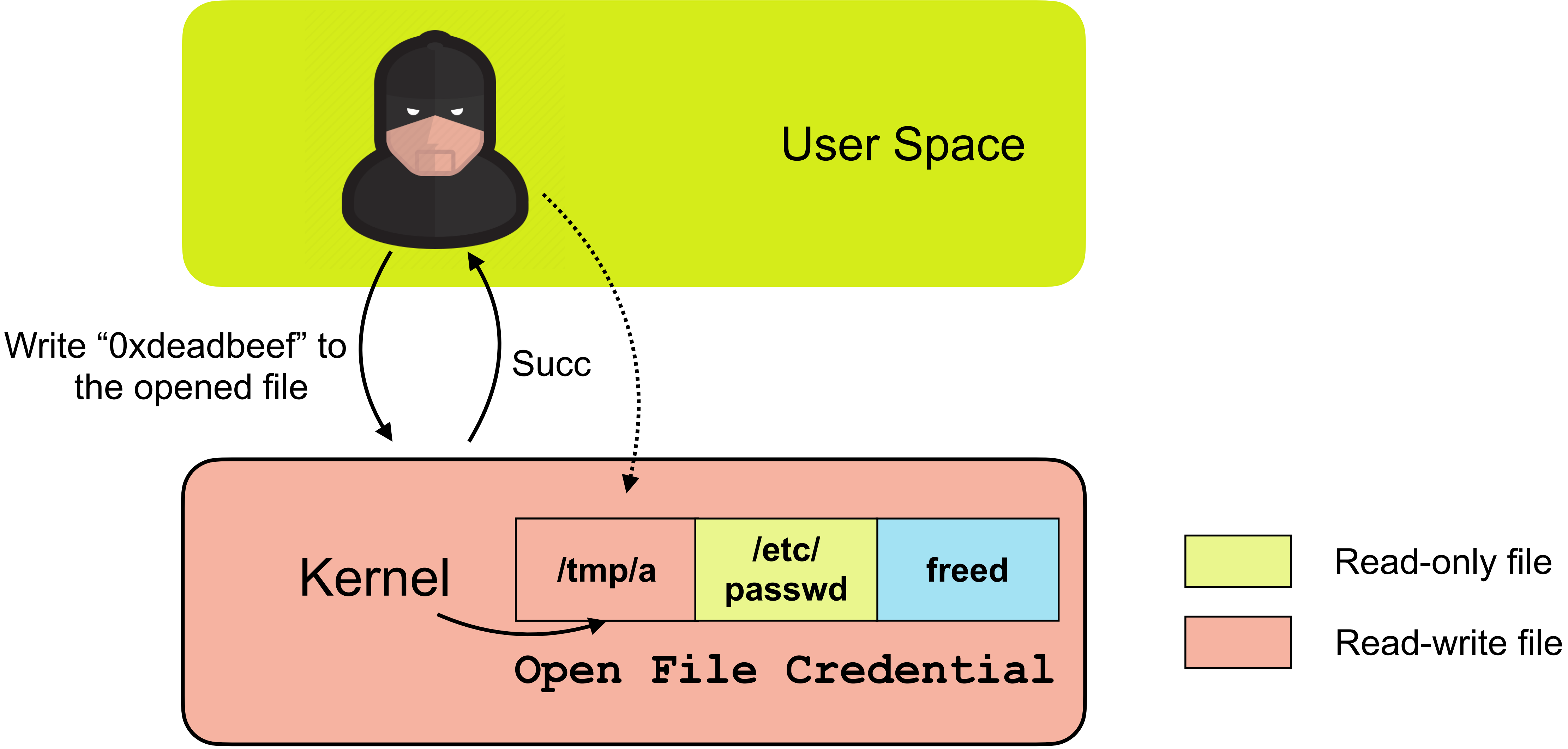
Open File Credential

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

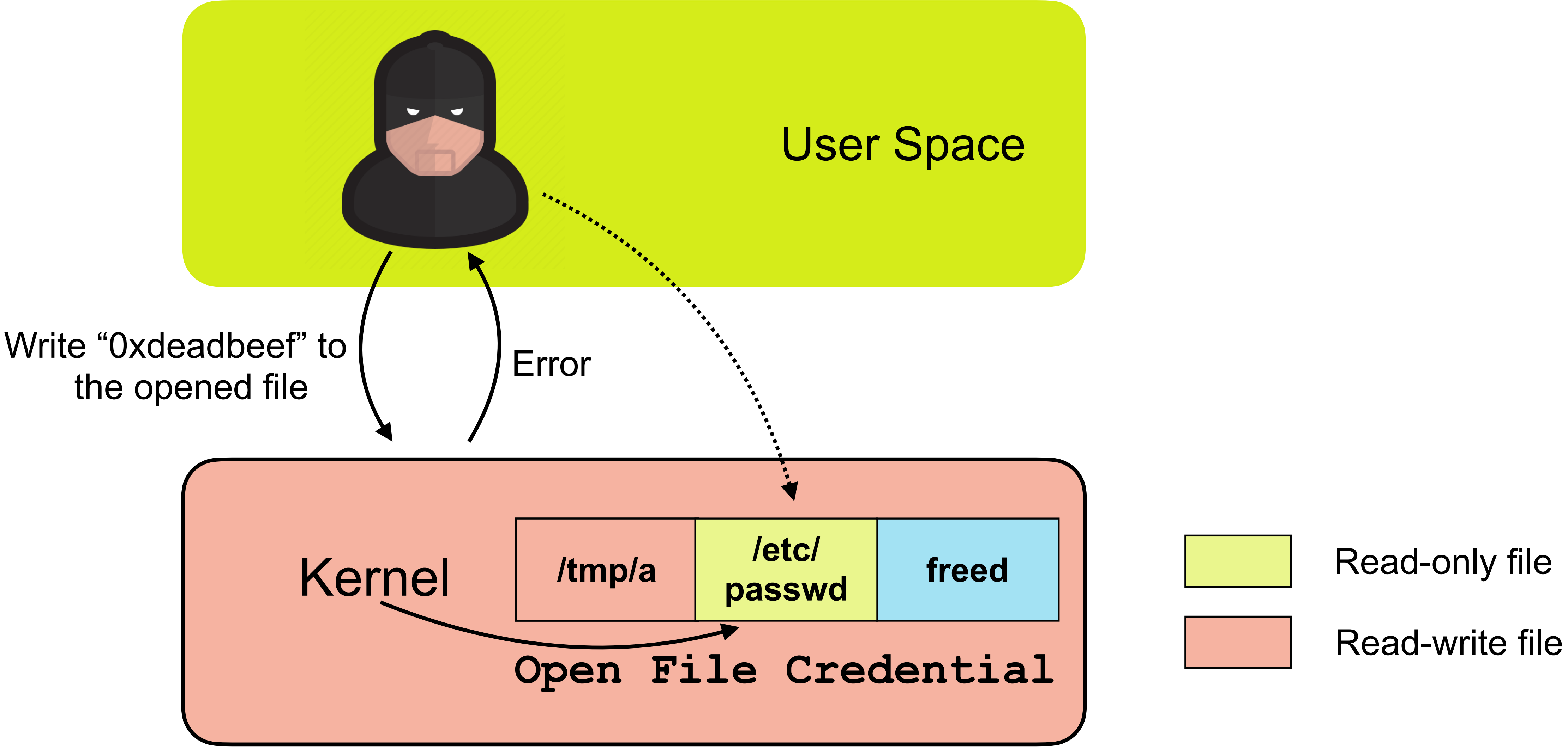


`Open File Credential` on kernel heap

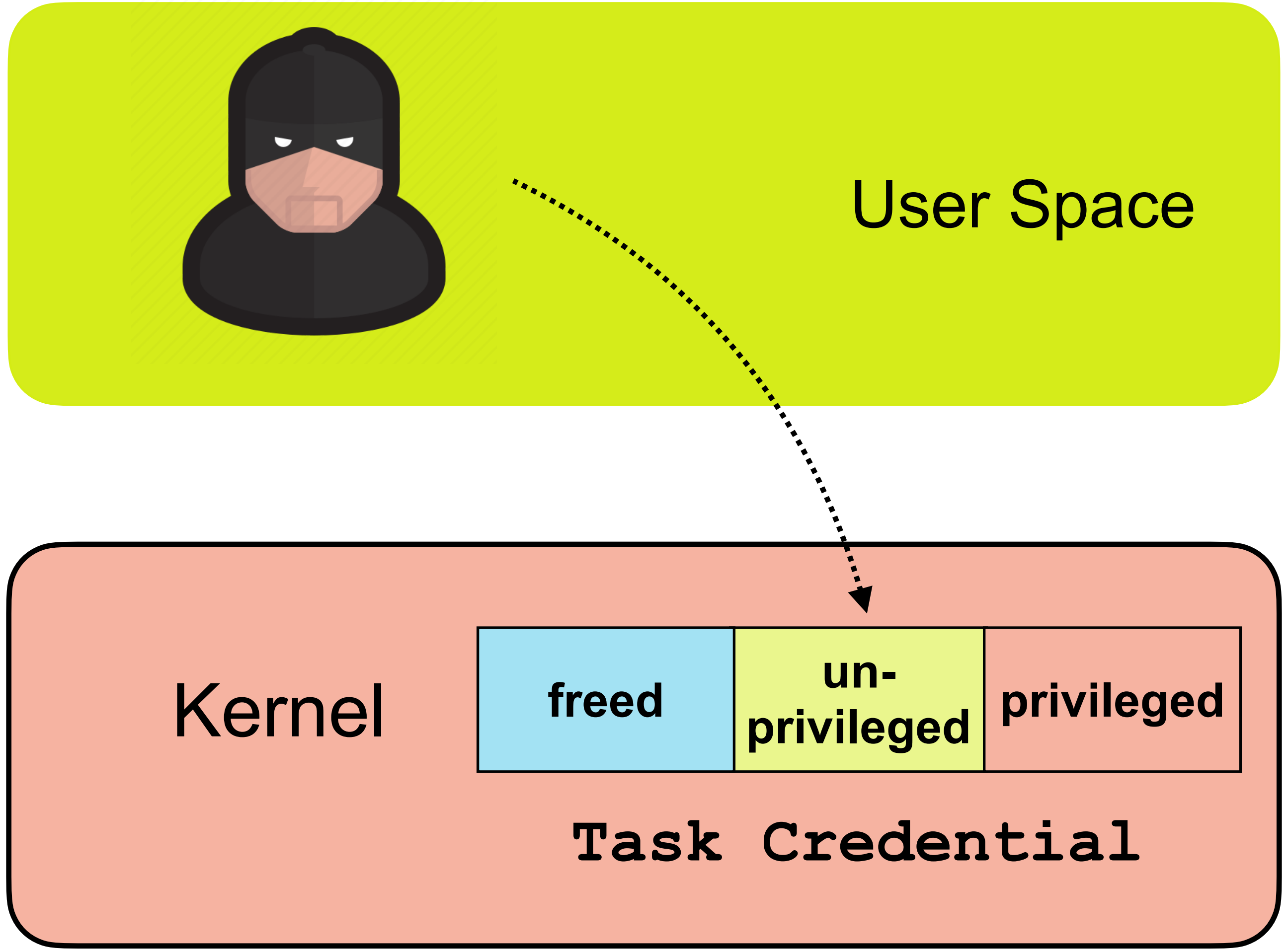
How Linux Kernel Uses Open File Credential



How Linux Kernel Uses Open File Credential

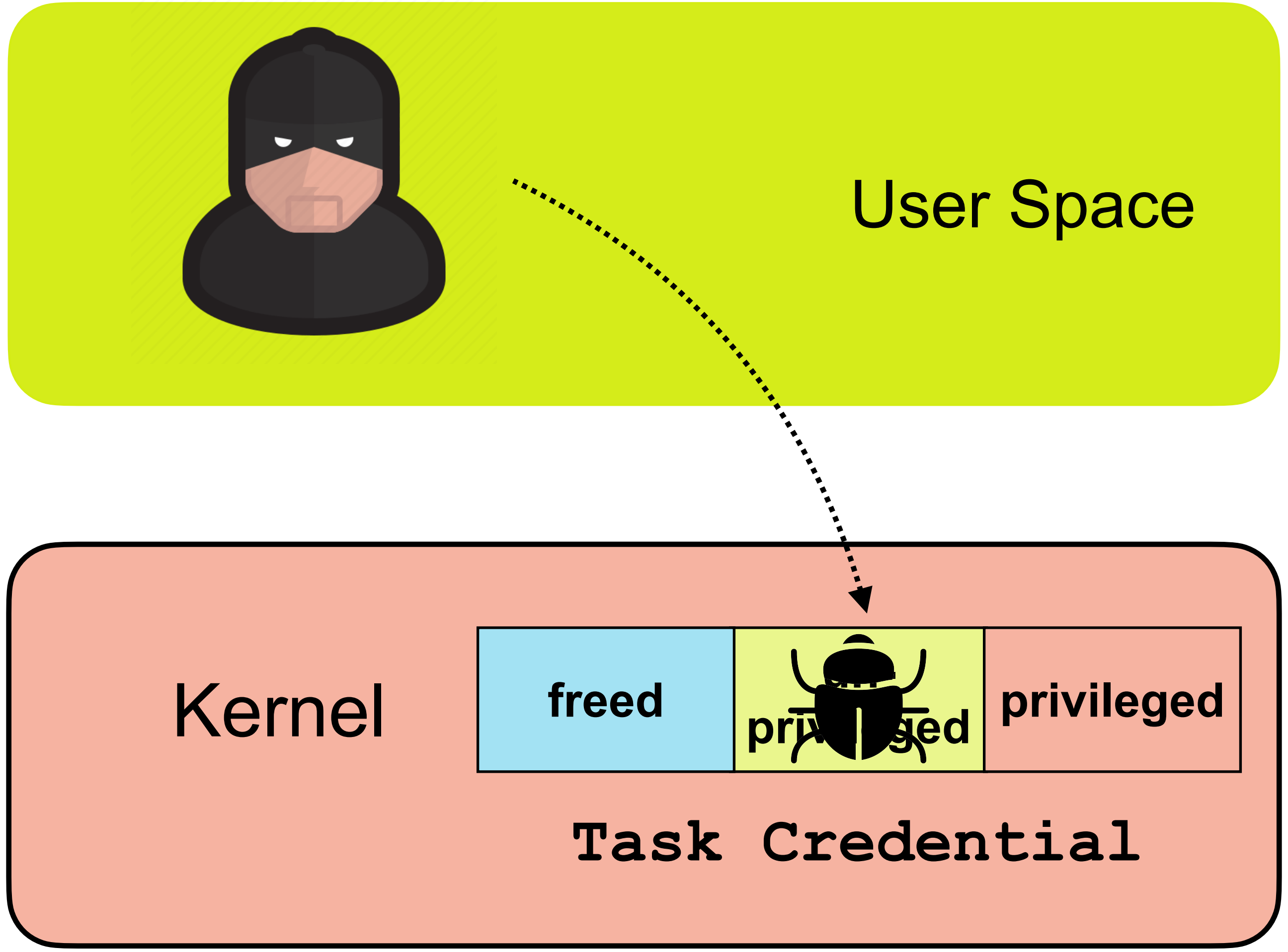


Attacking Task Credential



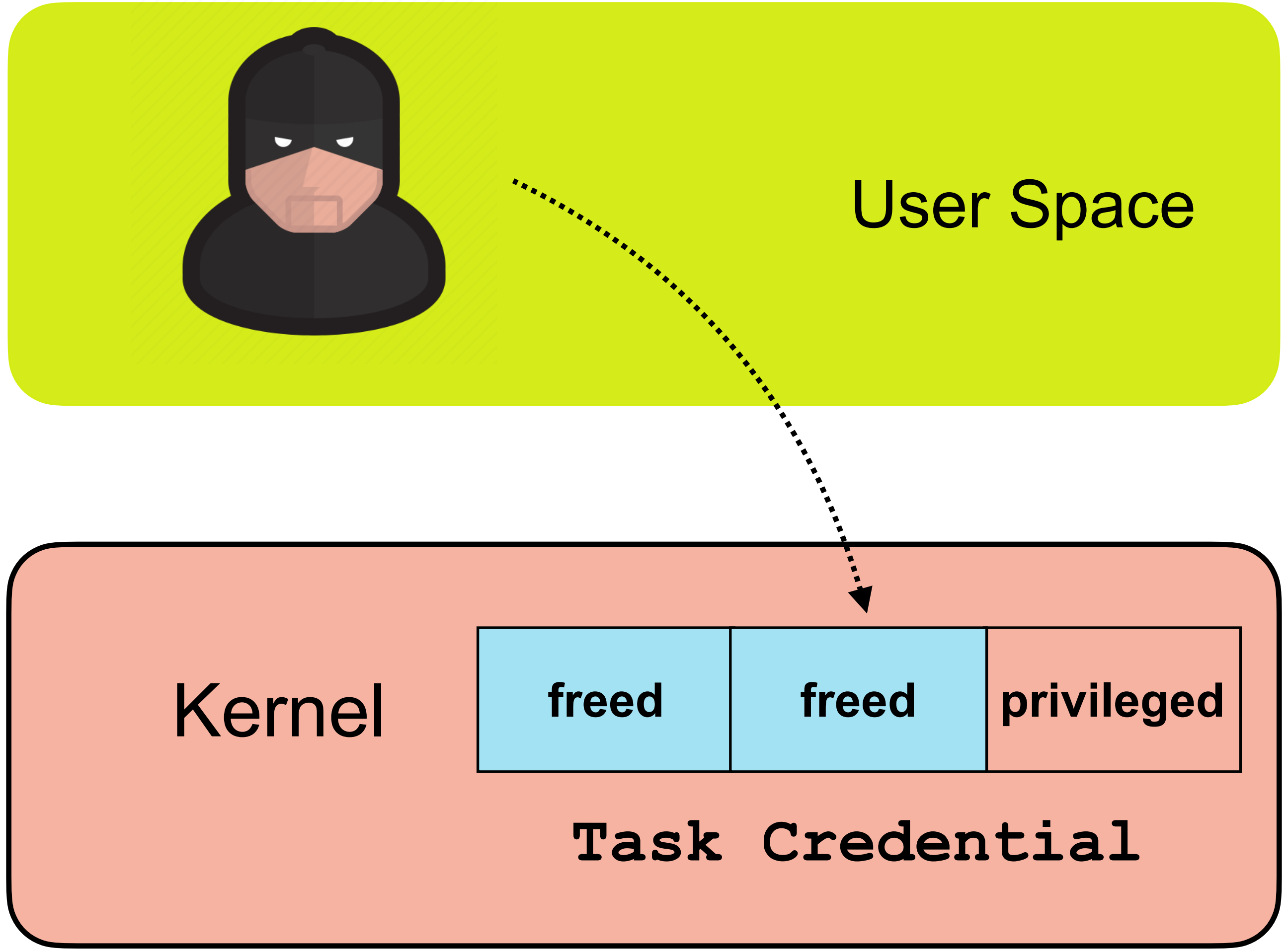
Attacking Task Credential

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



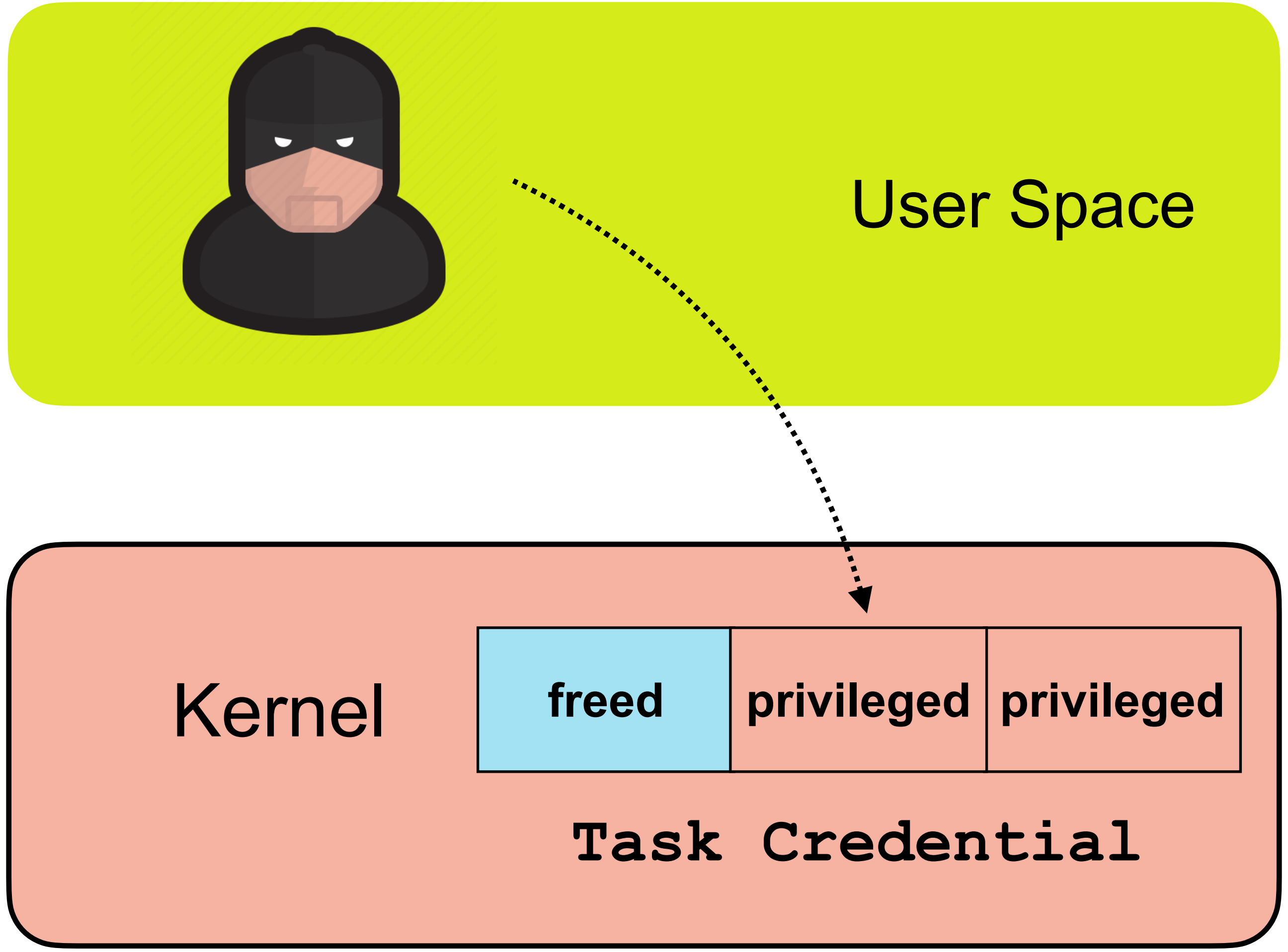
Attacking Task Credential

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



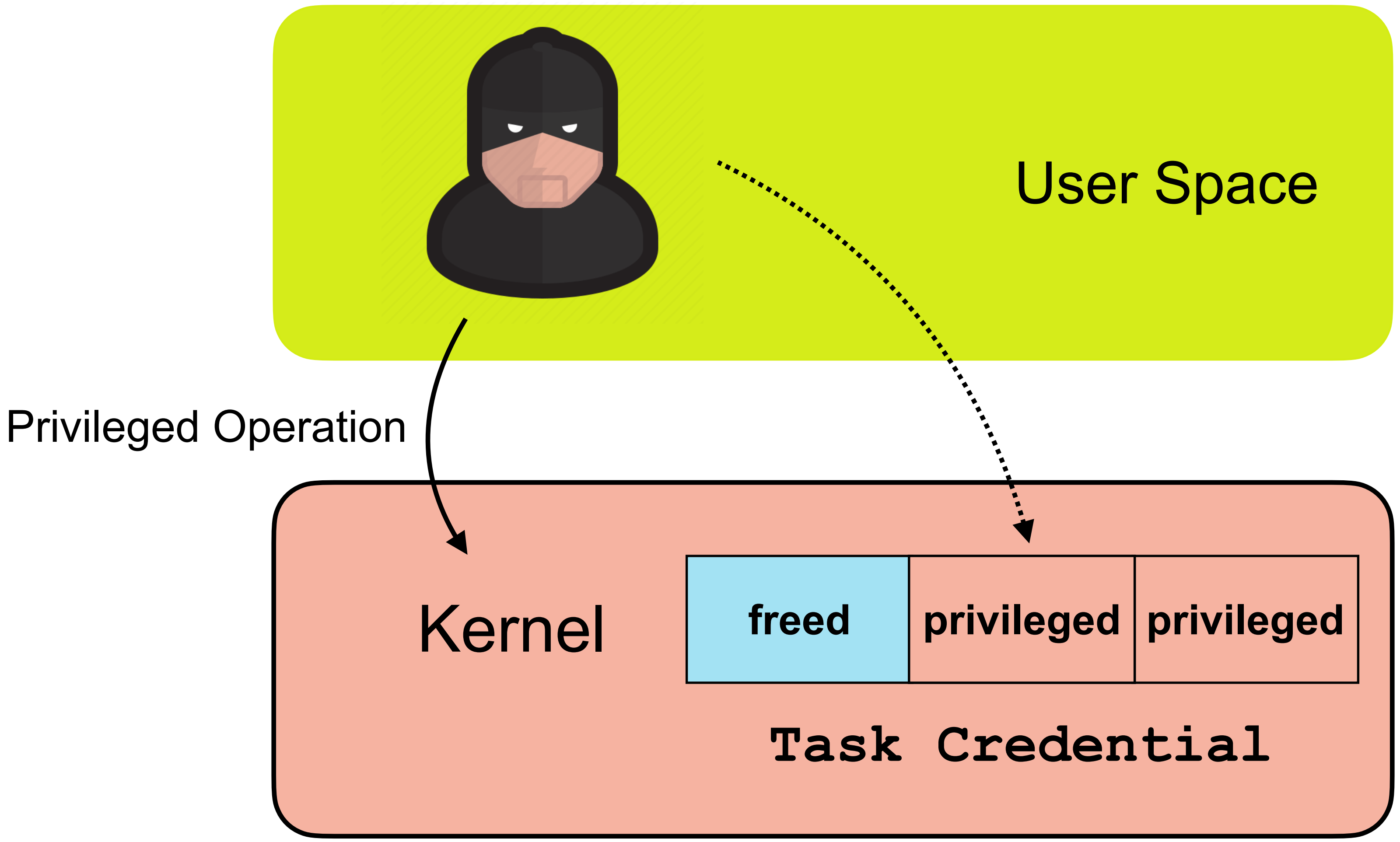
Attacking Task Credential

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



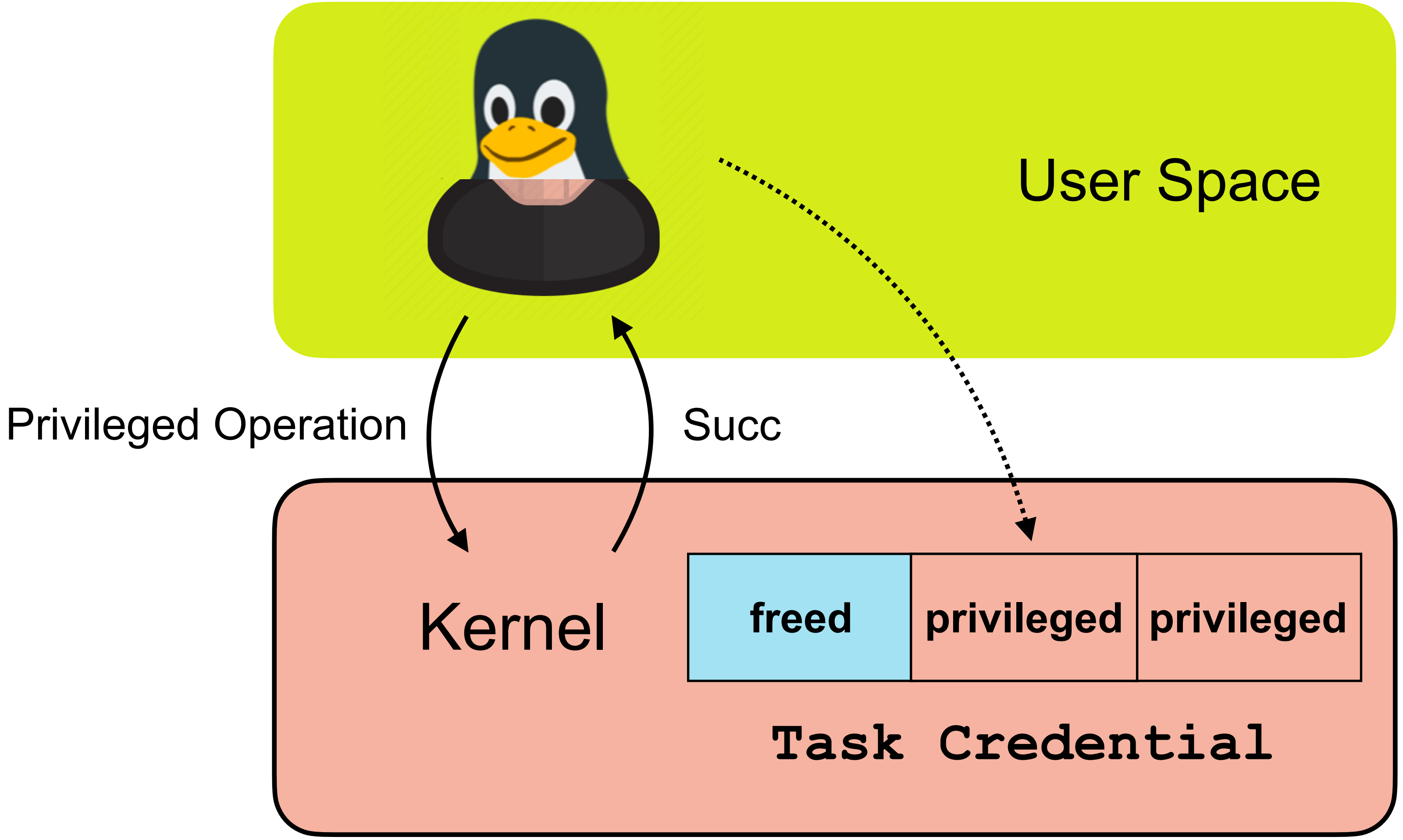
Attacking Task Credential

Result: Becoming a *privileged* user

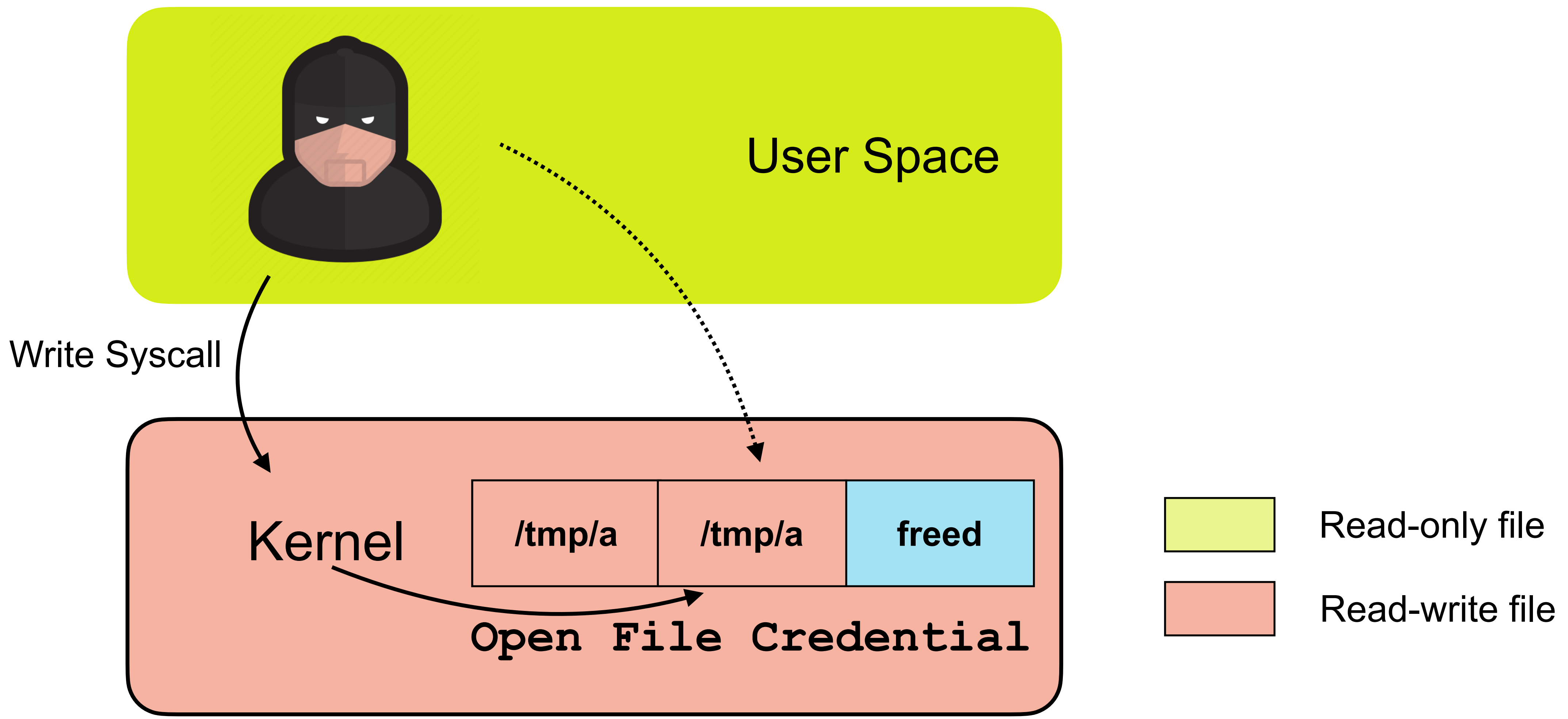


Attacking Task Credential

Result: Becoming a *privileged* user

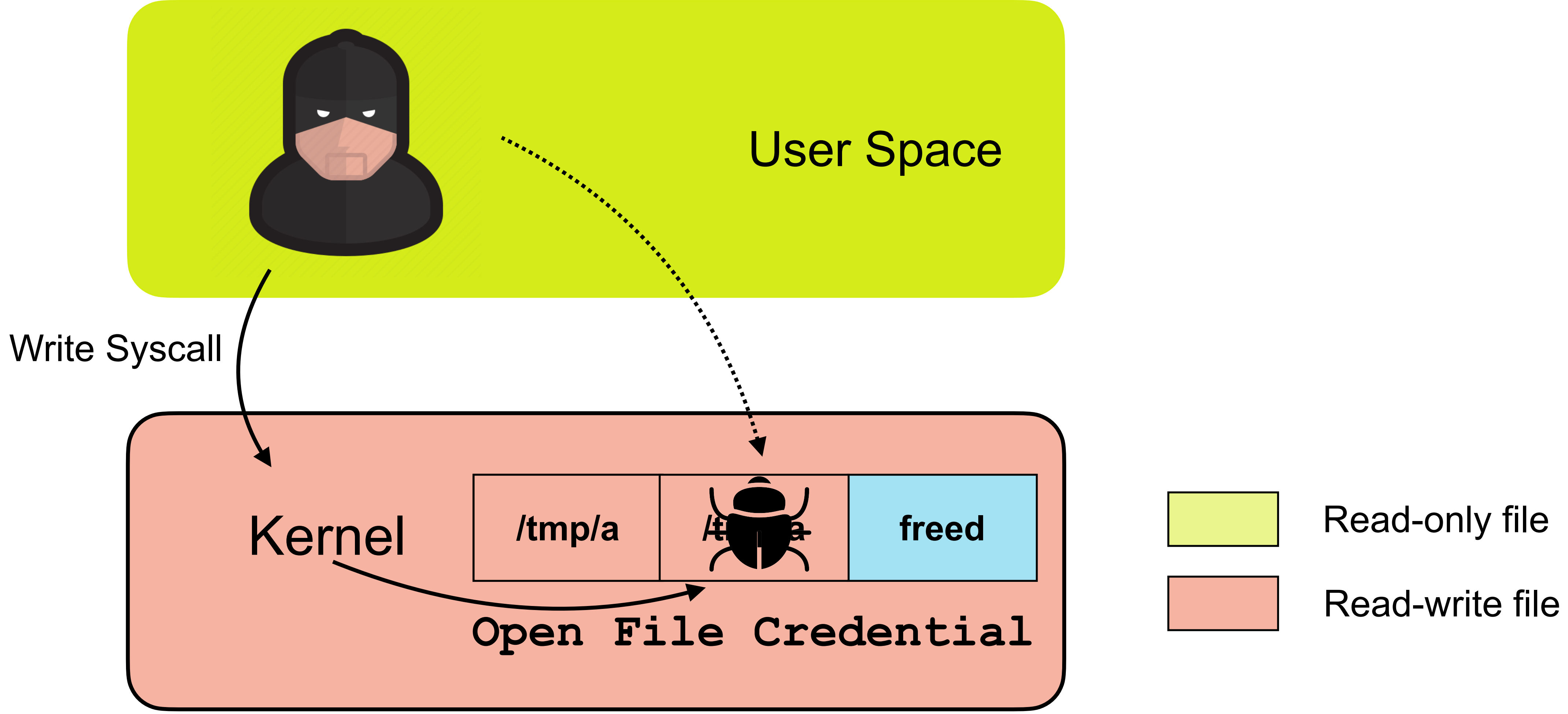


Attacking Open File Credential



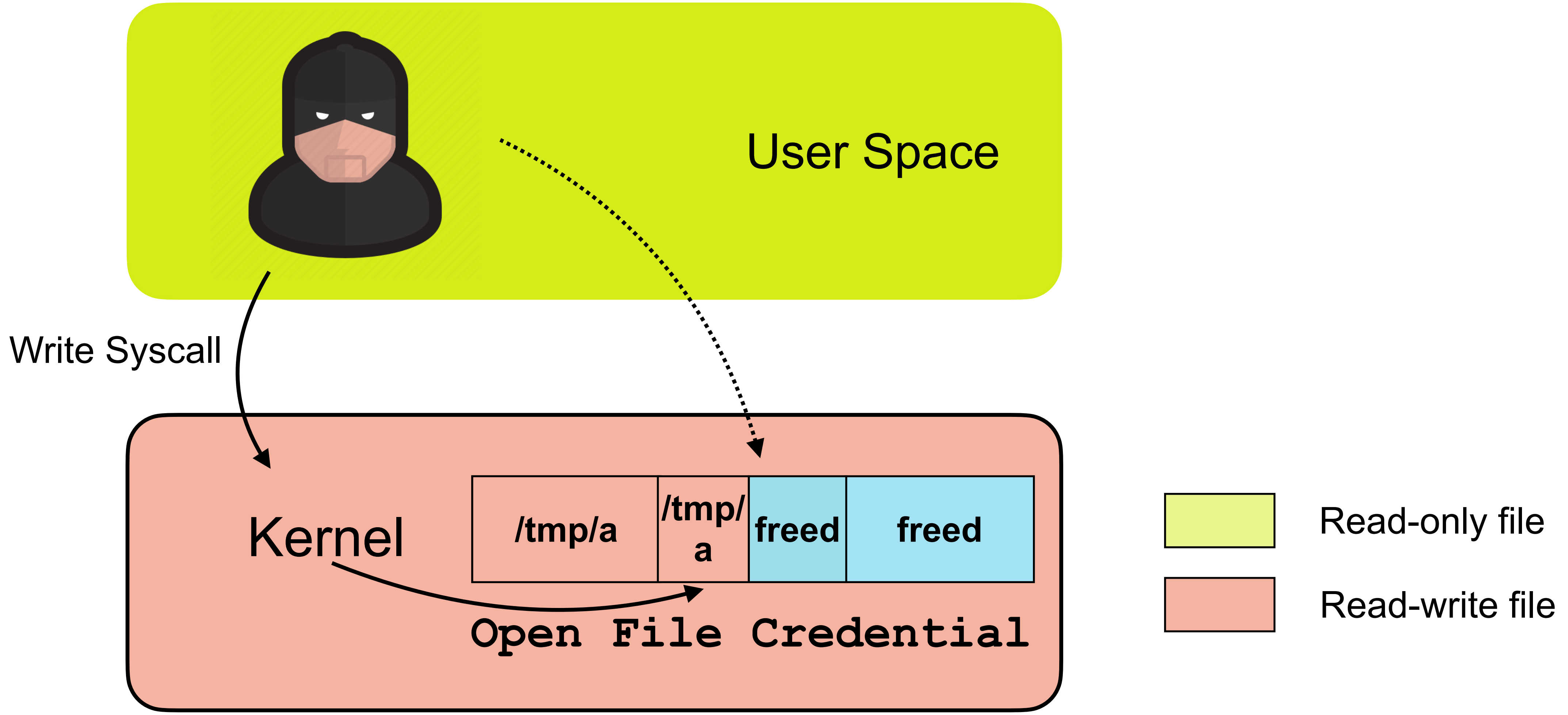
Attacking Open File Credential

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



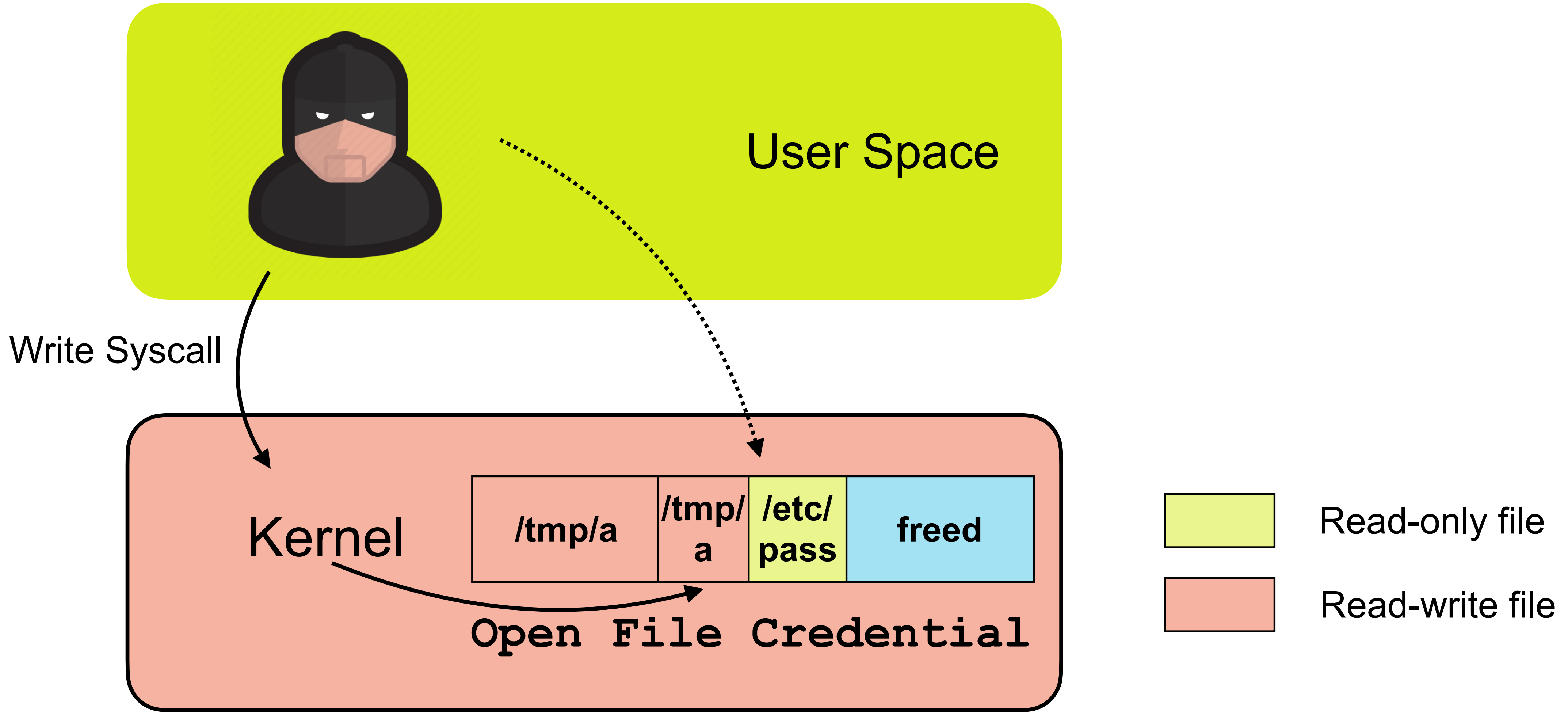
Attacking Open File Credential

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



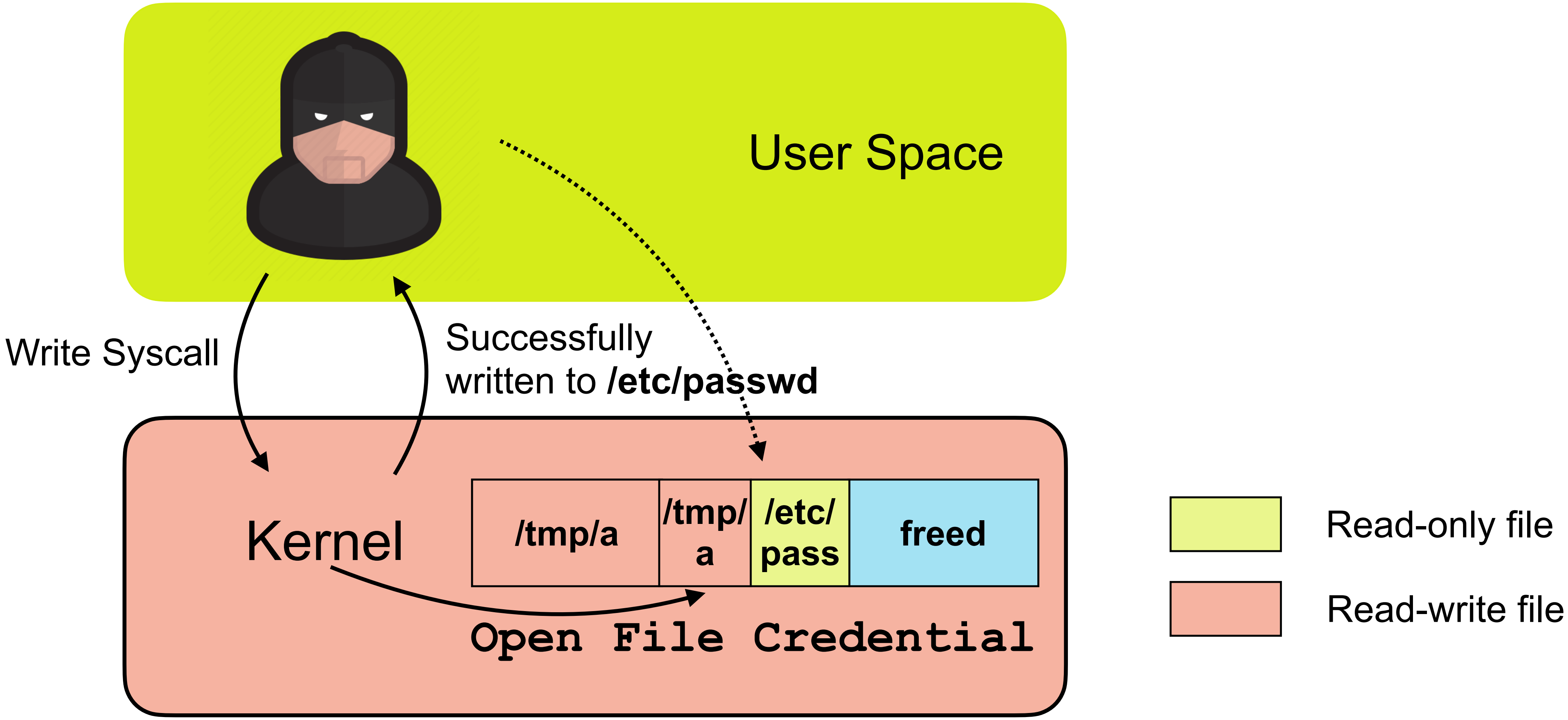
Attacking Open File Credential

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



Attacking Open File Credential

Result: Writing content to read-only files



Challenges

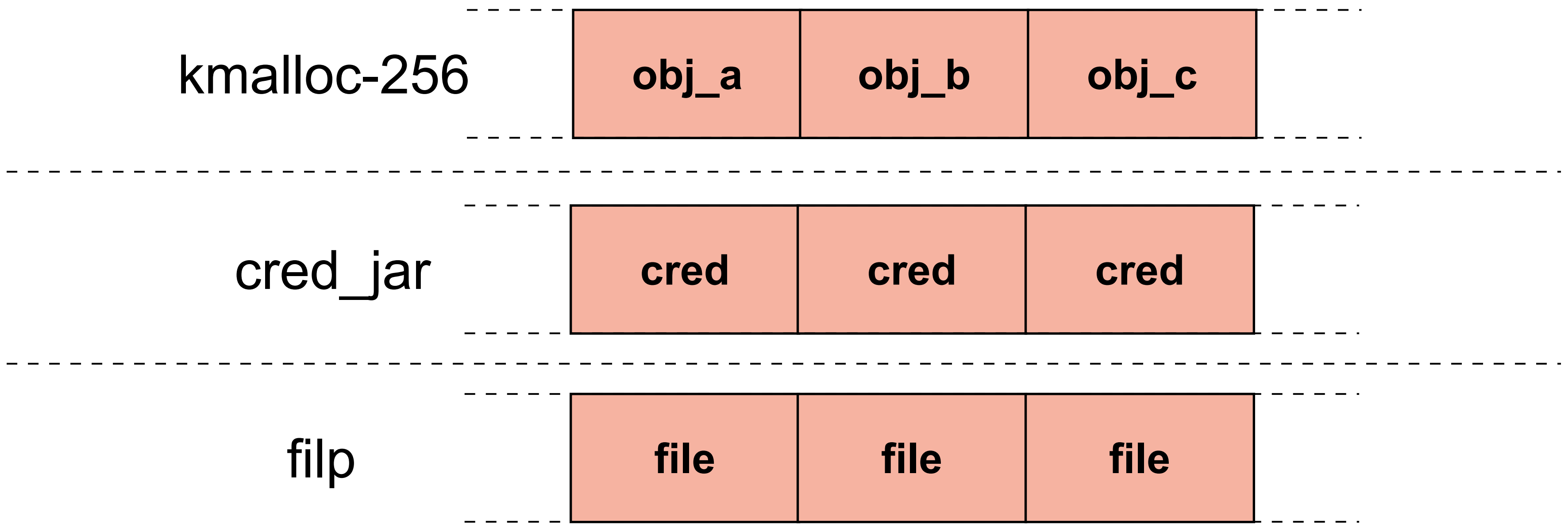
1. How to **free** credentials.
2. How to **allocate** *privileged* credentials as *unprivileged* users. (attacking *task* credentials)
3. How to finish attack in a **small** time window. (attacking *open file* credentials)

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



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

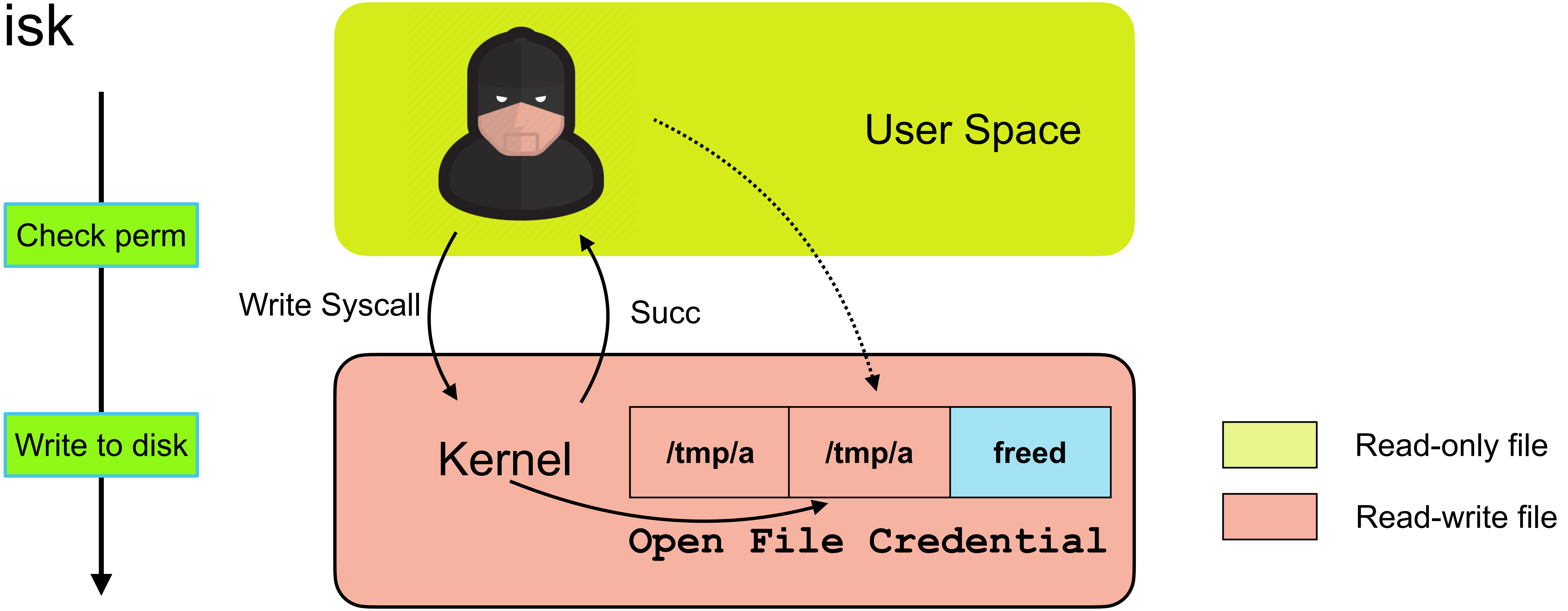
- ***Unprivileged*** users come with ***unprivileged*** task credentials
- Waiting privileged users to allocate task credentials influences the success rate

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)

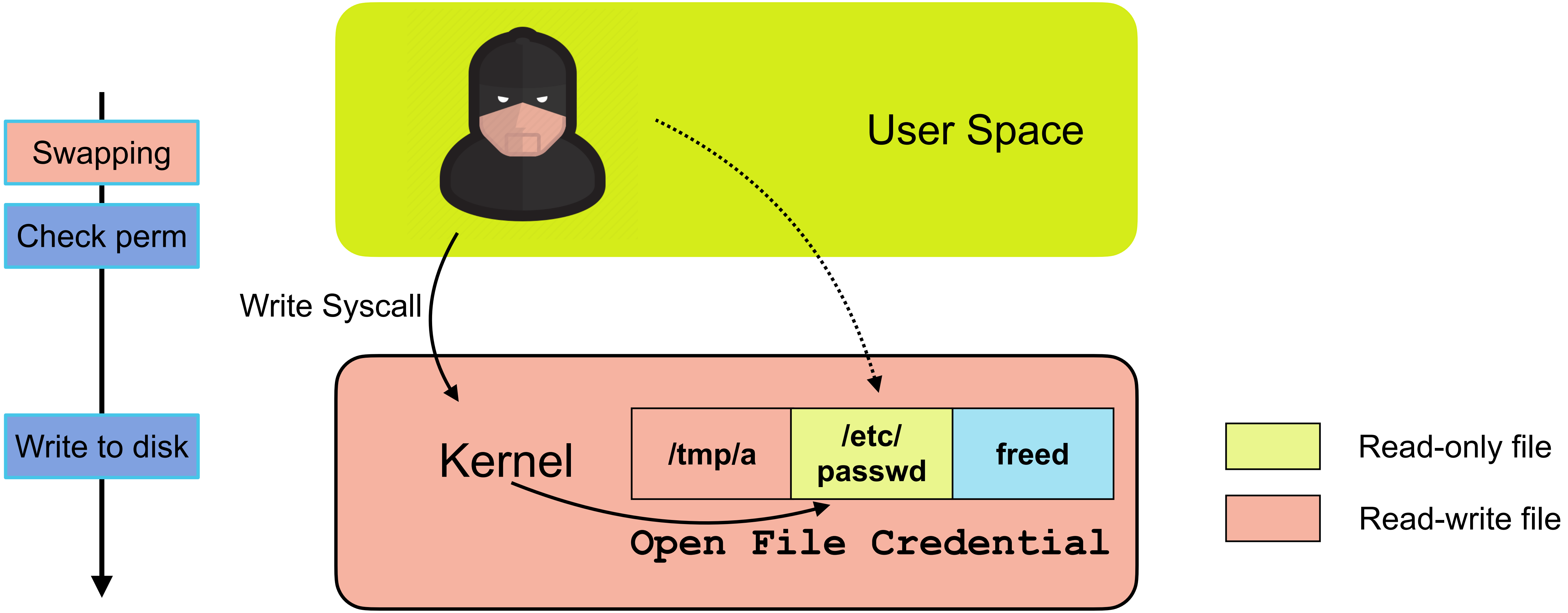
Challenge 3: Wining the race

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



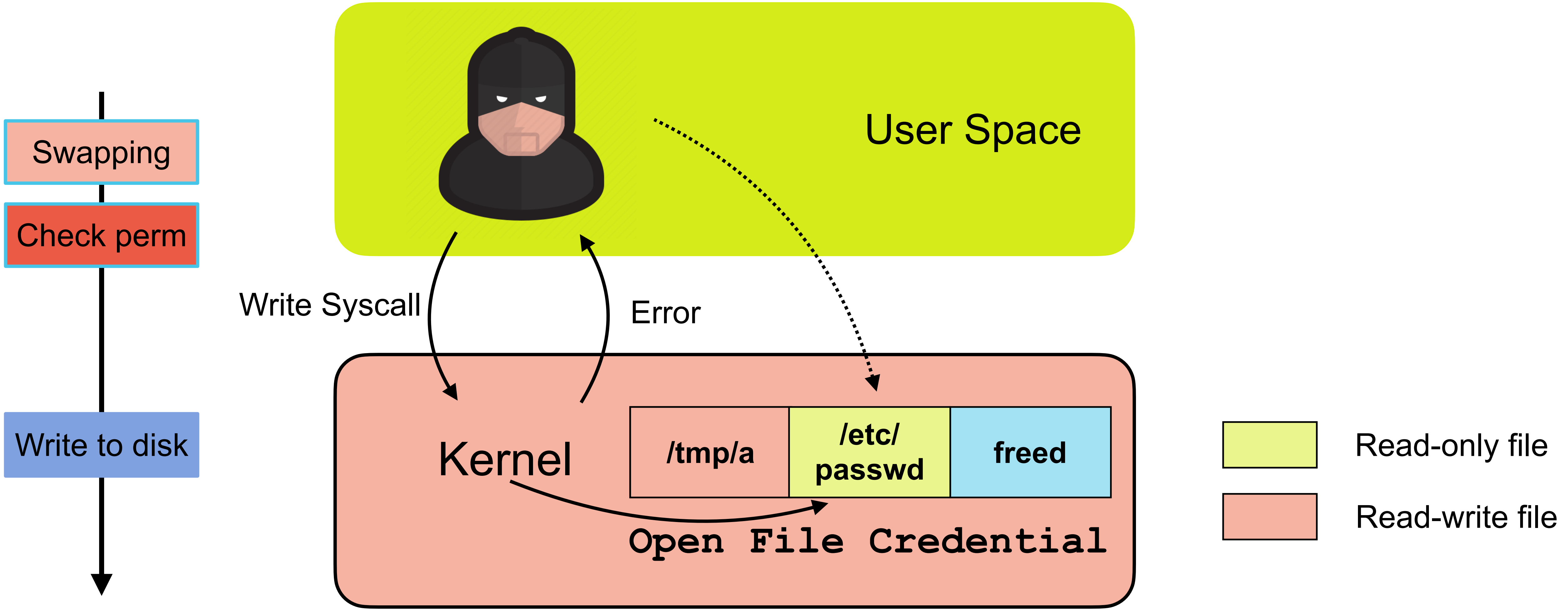
Challenge 3: Wining the race

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



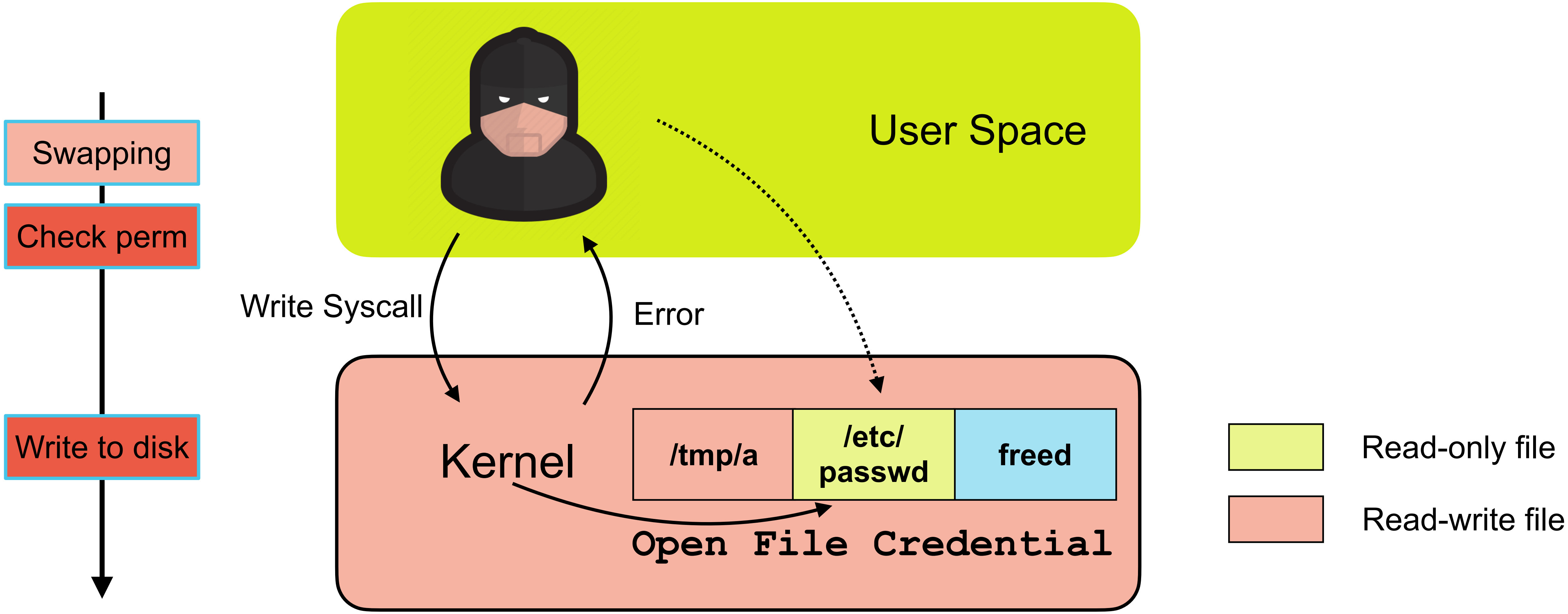
Challenge 3: Wining the race

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



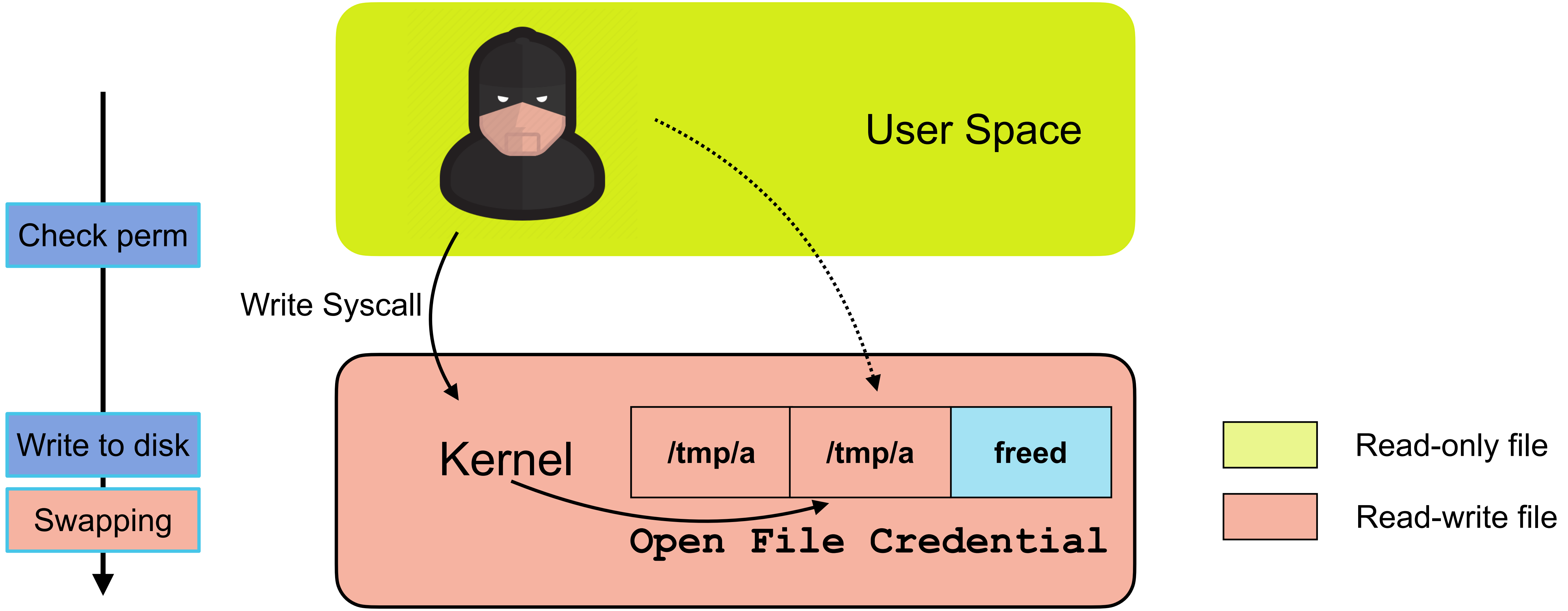
Challenge 3: Wining the race

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



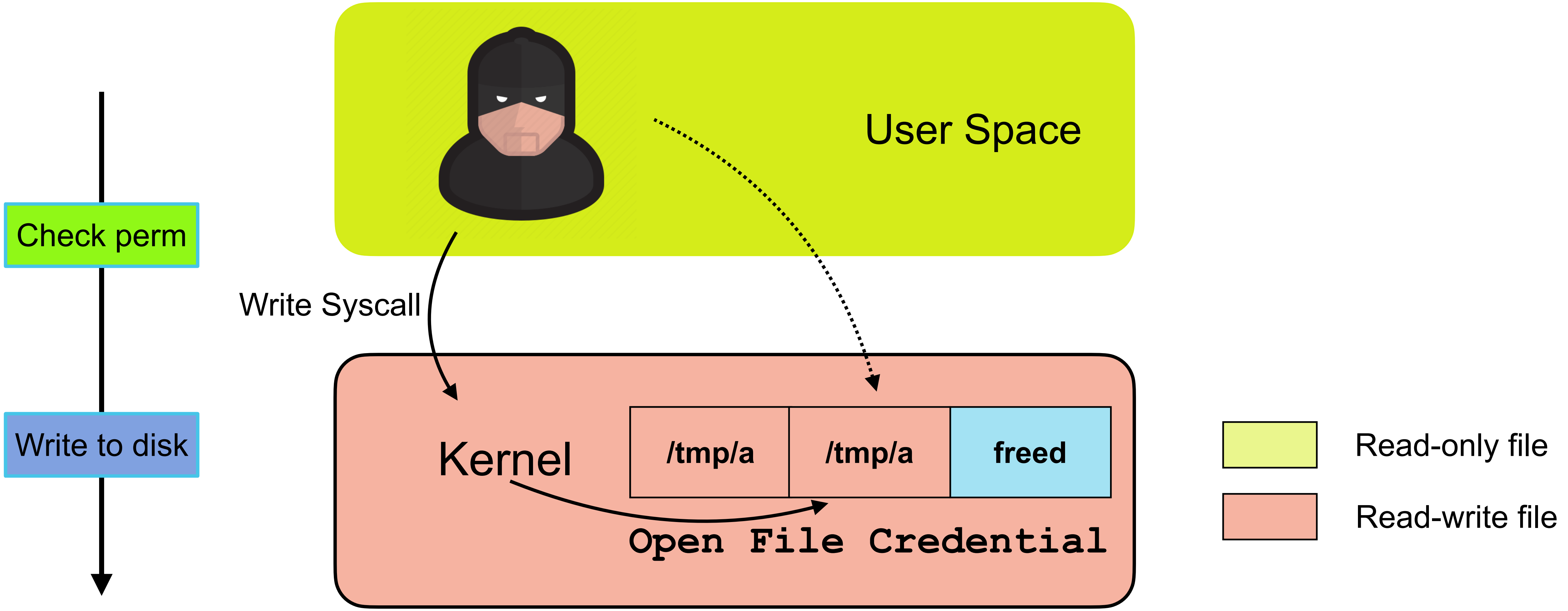
Challenge 3: Wining the race

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



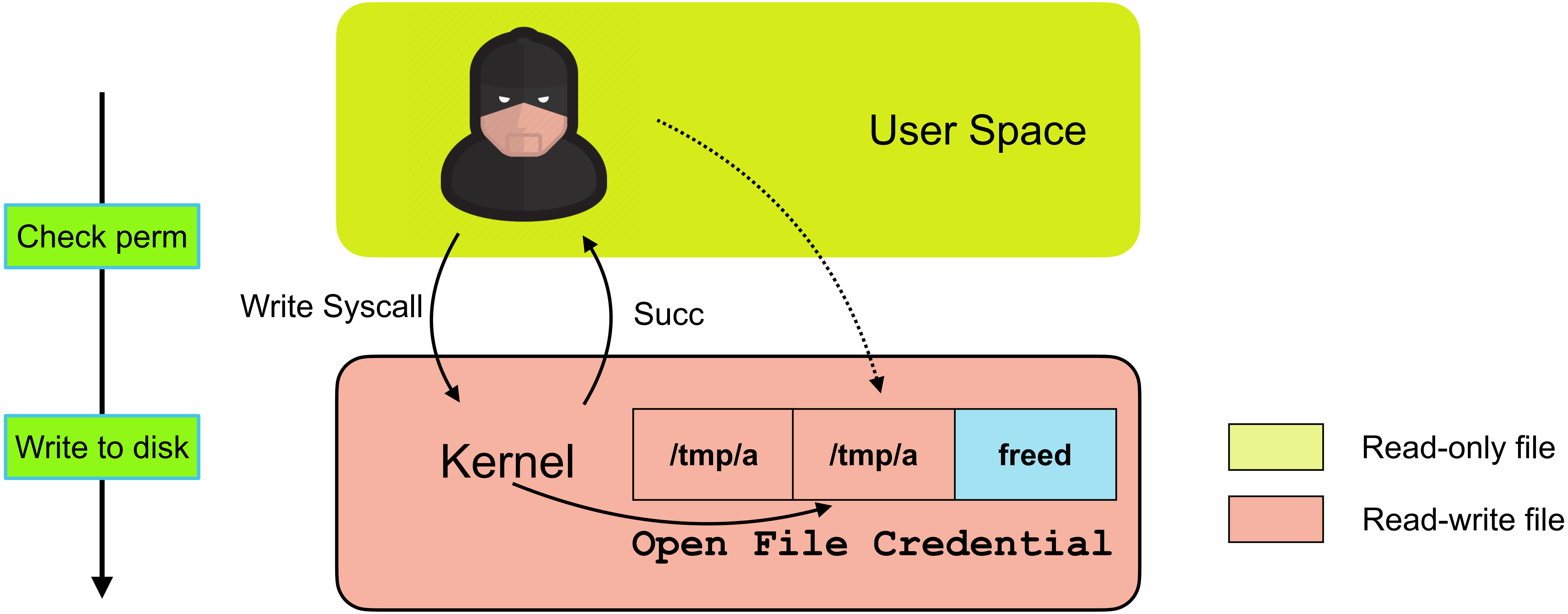
Challenge 3: Wining the race

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



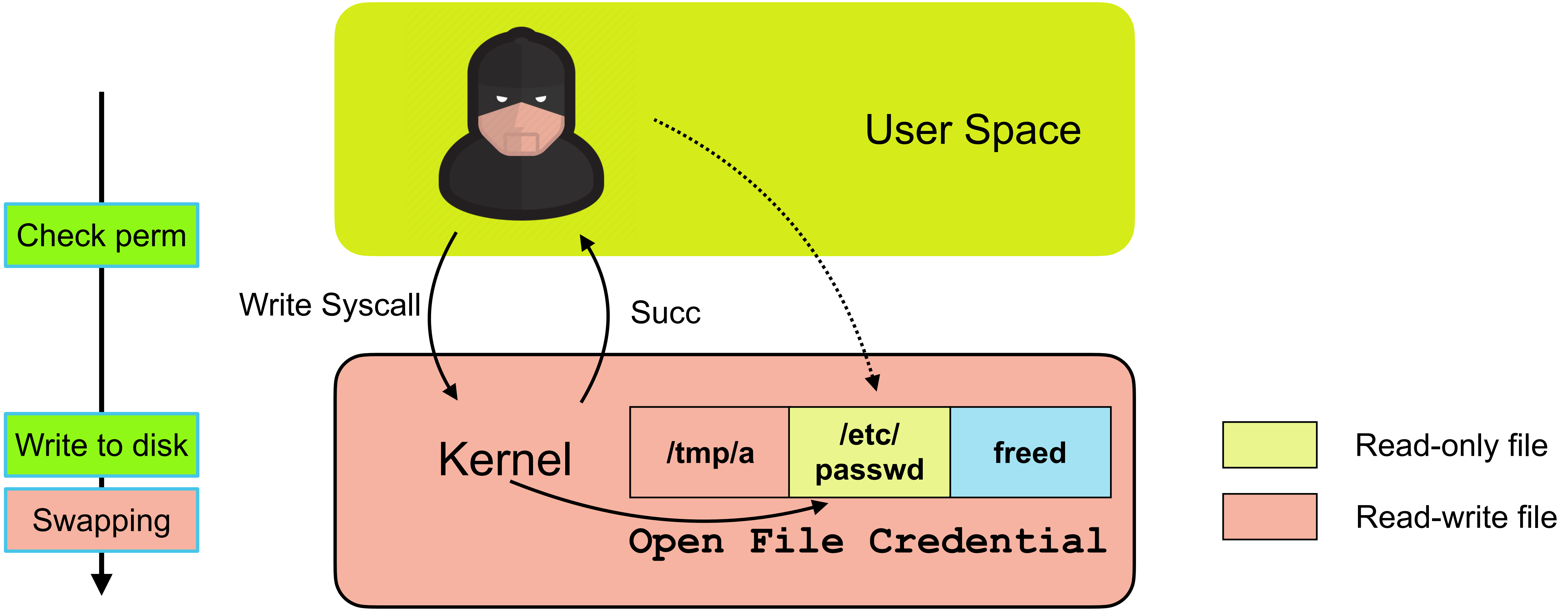
Challenge 3: Wining the race

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



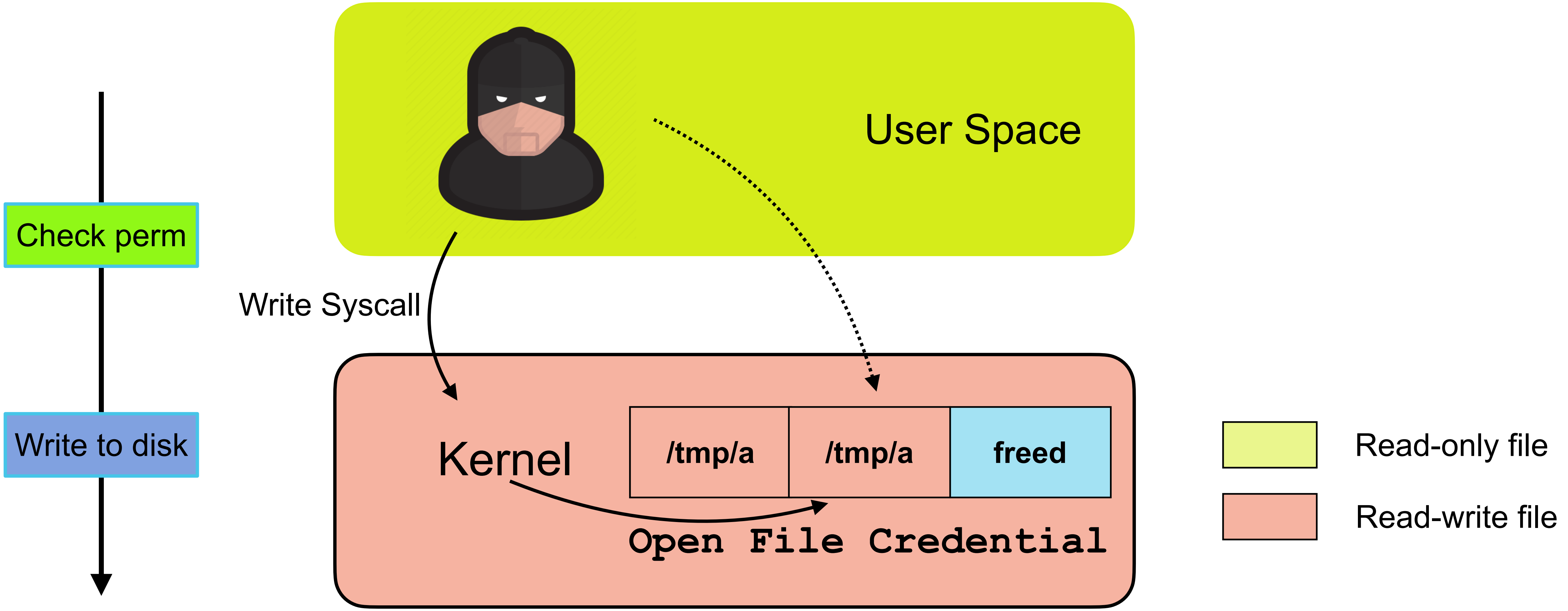
Challenge 3: Wining the race

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



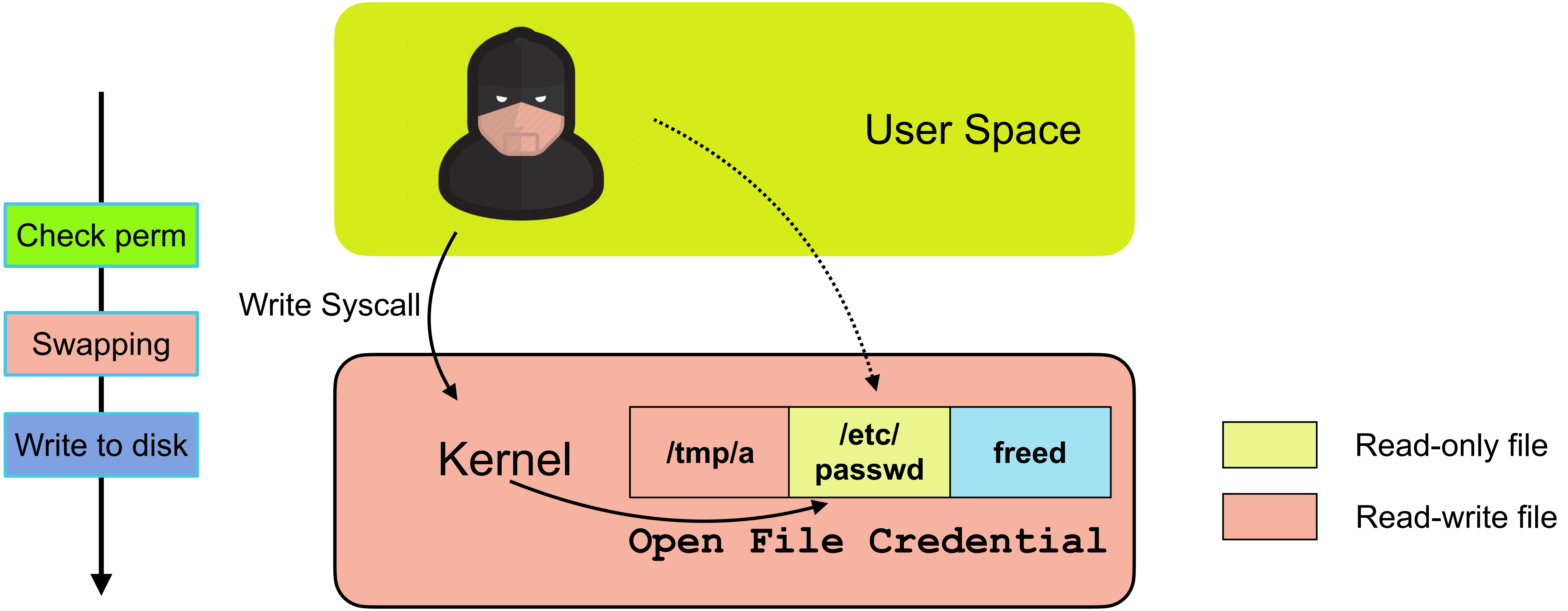
Challenge 3: Wining the race

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



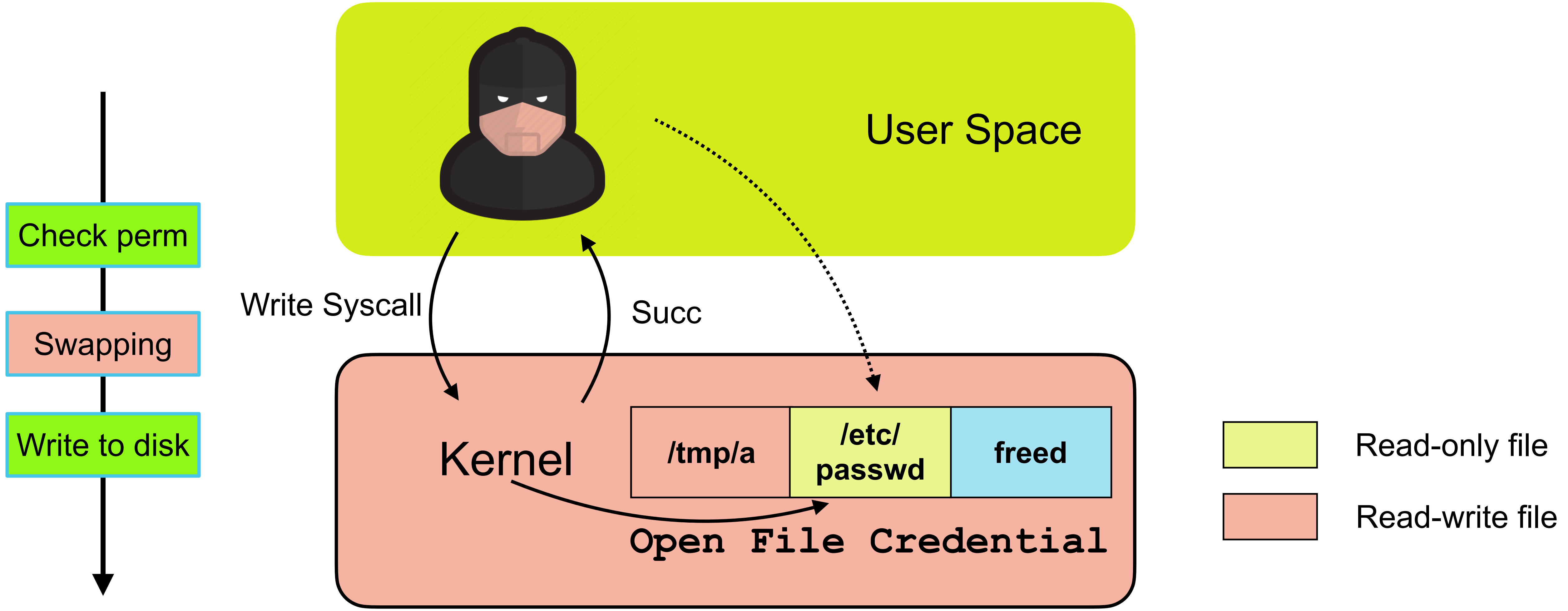
Challenge 3: Wining the race

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



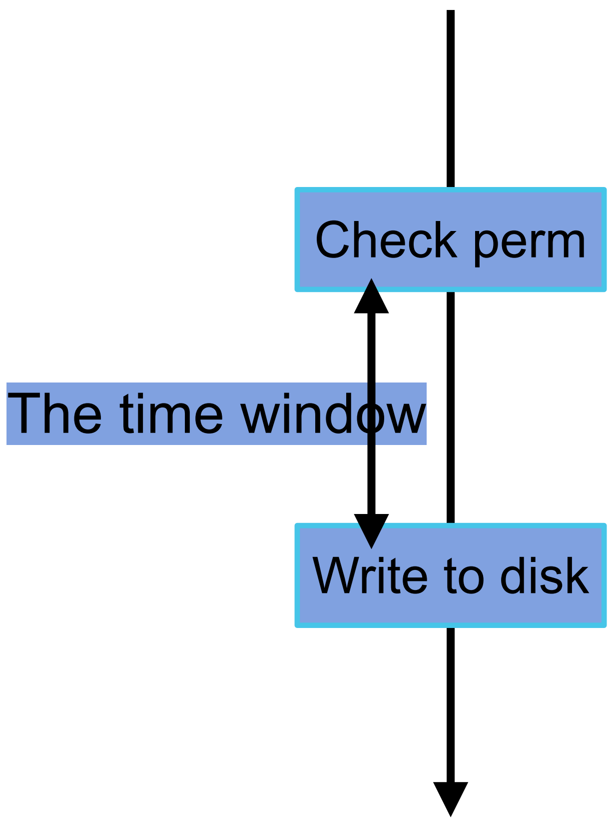
Challenge 3: Wining the race

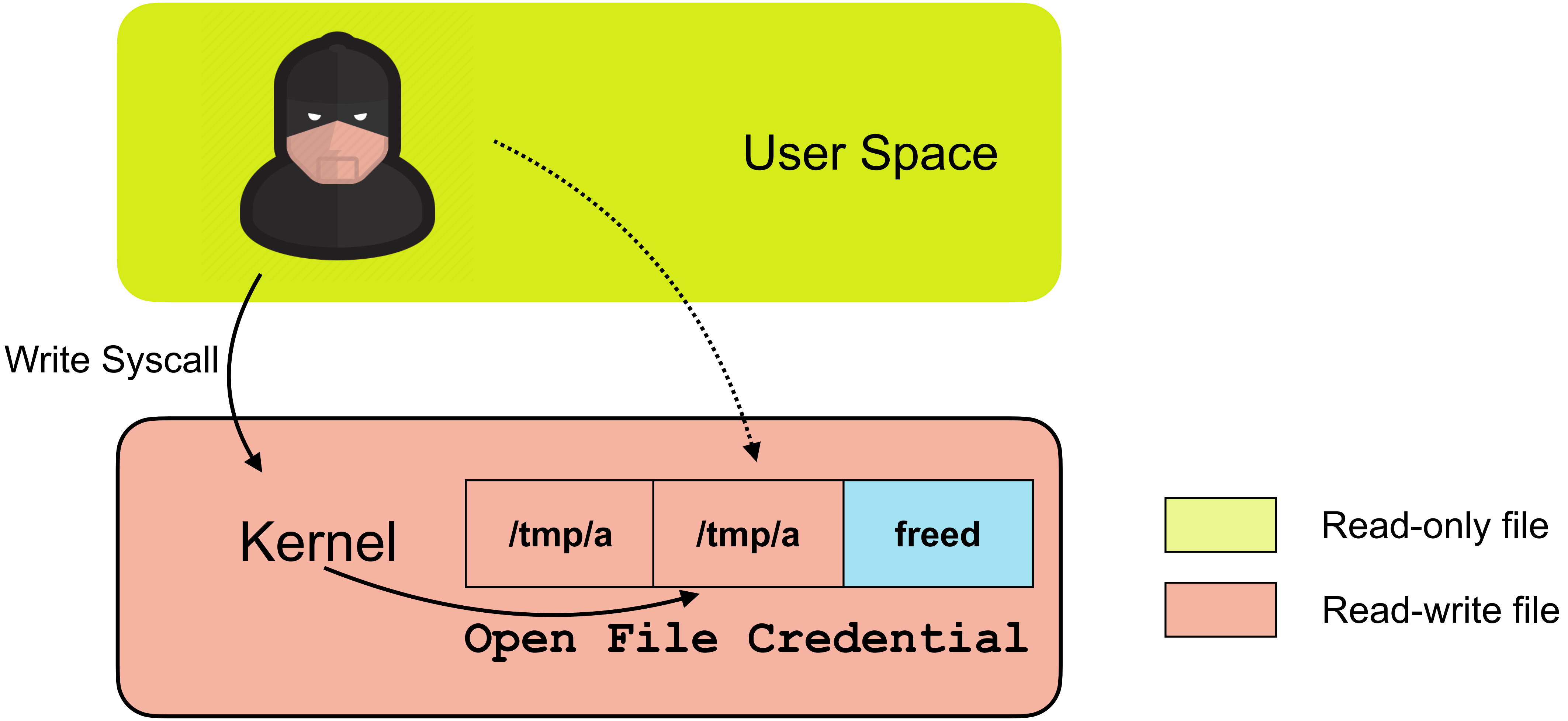
- The swap happens in between *permission check* and *file write*



Challenge 3: Wining the race

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





Northwestern

How We Address The Challenges

DirtyCred: Escalating Privilege in Linux Kernel

Zhenpeng Lin
zplin@u.northwestern.edu
Northwestern University

Yuhang Wu
yuhang.wu@northwestern.edu
Northwestern University

Xinyu Xing
xinyu.xing@northwestern.edu
Northwestern University

ABSTRACT

The kernel vulnerability DirtyPipe was reported to be present in nearly all versions of Linux since 5.8. Using this vulnerability, a bad actor could fulfill privilege escalation without triggering existing kernel protection and exploit mitigation, making this vulnerability particularly disconcerting. However, the success of DirtyPipe exploitation heavily relies on this vulnerability's capability (i.e., injecting data into the arbitrary file through Linux's pipes). Such an ability is rarely seen for other kernel vulnerabilities, making the defense relatively easy. As long as Linux users eliminate the vulnerability, the system could be relatively secure.

This work proposes a new exploitation method – DirtyCred – pushing other Linux kernel vulnerabilities to the level of DirtyPipe. Technically speaking, given a Linux kernel vulnerability, our exploitation method swaps unprivileged and privileged kernel credentials and thus provides the vulnerability with the DirtyPipe-like exploitability. With this exploitability, a bad actor could obtain the ability to escalate privilege and even escape the container. We evaluated this exploitation approach on 24 real-world kernel vulnerabilities in a fully-protected Linux system. We discovered that DirtyCred could demonstrate exploitability on 16 vulnerabilities, implying DirtyCred's security severity. Following the exploitability assessment, this work further proposes a new kernel defense mechanism. Unlike existing Linux kernel defenses, our new defense isolates kernel credential objects on non-overlapping memory regions based on their own privilege. Our experiment result shows that the new defense introduces primarily negligible overhead.

CCS CONCEPTS

- **Security and privacy** → *Operating systems security*; *Software security engineering*;

KEYWORDS

OS Security; Kernel Exploitation; Privilege Escalation

ACM Reference Format:

1 INTRODUCTION

Nowadays, Linux has become a popular target for cybercrooks due to its popularity among mobile devices, cloud infrastructure, and Web servers. To secure Linux, kernel developers and security experts introduce a variety of kernel protection and exploit mitigation techniques (e.g., KASLR [14] and CFI [19]), making kernel exploitation unprecedentedly difficult. To fulfill an exploitation goal successfully, today's bad actor has to identify those powerful kernel vulnerabilities with the capability of disabling corresponding protection and mitigation.

However, a recent vulnerability (cataloged as CVE-2022-0847 [10]) and its exploitation method are getting significant attention from the cybersecurity community. Because of its maliciousness and impact, it was even branded a nickname – DirtyPipe [30]. Unlike non-branded kernel vulnerabilities, DirtyPipe's exploitation fulfills privilege escalation without involving the effort of disabling broadly adopted kernel protection and exploit mitigation. This characteristic results in existing Linux defenses ineffective and thus leads many Linux-kernel-driven systems in danger (e.g., Android devices).

While DirtyPipe is powerful, its exploitability is closely tied to the vulnerability's capability (i.e., abusing the Linux kernel pipe mechanism to inject data to arbitrary files). For other Linux kernel vulnerabilities, such a pipe-abusive ability is rarely provided. As a result, the action taken by the Linux community and device manufacturers (e.g., Google) is to release a patch against the kernel bug rapidly and thus eliminate the attack surface. Without this attack surface, the exploitation against a fully-protected Linux kernel is still challenging. For other kernel vulnerabilities, it is still difficult to bring the same level of security impact as DirtyPipe.

In this work, we present a novel, general exploitation method through which even ordinary kernel vulnerabilities could fulfill the same exploitation objective as DirtyPipe. From a technical perspective, our exploitation method is different from DirtyPipe. It does not rely on the pipeline mechanism of Linux nor the nature of the vulnerability CVE-2022-0847. Instead, it employs a heap memory corruption vulnerability to replace a low privileged kernel creden-

Real-World Impact of DirtyCred

- **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

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)

All codes are available at https://github.com/markakd/DirtyCred

Summary

- **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