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

- Diagrams:
  - (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

(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
- Leak kernel pointers
- Tamper kernel pointers

Obtain Primitives

Partial overlap between Type C and A
How Researchers Exploit Kernel Vulns

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

Obtain Primitives
Bypass Mitigation
How Researchers Exploit Kernel Vulns

- Spatial/Temporal memory error
- Type confusion and memory overlap
- Leak kernel pointers
- Tamper kernel pointers
- Execute ROP in different forms\(^1\)

\(^1\) Joy of exploiting the kernel
How Researchers Exploit Kernel Vulns

<table>
<thead>
<tr>
<th>Obtain Primitives</th>
<th>Bypass Mitigation</th>
<th>Escalate Privilege</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial/Temporal memory error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type confusion and memory overlap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leak kernel pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tamper kernel pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute ROP in different forms[1]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[1] Joy of exploiting the kernel
# How Researchers Exploit Kernel Vulns

<table>
<thead>
<tr>
<th>Spatial/Temporal memory error</th>
<th>Obtain Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type confusion and memory overlap</td>
<td></td>
</tr>
<tr>
<td>Leak kernel pointers</td>
<td>Bypass Mitigation</td>
</tr>
<tr>
<td>Tamper kernel pointers</td>
<td></td>
</tr>
<tr>
<td>Execute ROP in different forms(^1)</td>
<td>Escalate Privilege</td>
</tr>
</tbody>
</table>

Used by 15/17 exploits in [2]

---

\(^1\) Joy of exploiting the kernel

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
- Leak kernel pointers
- Tamper kernel pointers
- Execute ROP
How DirtyCred Exploits Kernel Vulns

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

Obtain Primitives
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

```c
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

![Diagram showing Task Credential on kernel heap with states: freed, unprivileged, freed, unprivileged, privileged]
How Linux Kernel Uses Task Credential

Kernel

- freed
- unprivileged
- privileged

Task Credential

User Space

Privileged Operation

Error
How Linux Kernel Uses Task Credential

Kernel

Task Credential

User Space

Privileged Operation

Succ

freed
un-privileged
privileged
Open File Credential

• **Struct file** in Linux kernel’s implementation

```c
struct file {
    union {
        struct llist_node f_list;
        struct rcu_head f_rcuhead;
        unsigned int f_locb_flags;
    };
    struct path f_path;
    struct inode *f_inode; /* cannot be a symbolic link */
    const struct file_operations *f_op;

    /*
     * Protects f_op, 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("/tmp/a", O_RDWR)
open("/tmp/a", O_RDONLY)
open("/etc/passwd", O_RDONLY)
```

```
freed /tmp/a /tmp/a /etc/passwd
```

Open File Credential on kernel heap
How Linux Kernel Uses Open File Credential

User Space

Write “0xdeadbeef” to the opened file

Kernel

Open File Credential

Succ

/tmp/a
/etc/passwd
freed

Read-only file

Read-write file
How Linux Kernel Uses Open File Credential

Kernel

User Space

Write “0xdeadbeef” to the opened file

Error

Read-only file

Read-write file
Attacking Task Credential

Kernel

User Space
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

Kernel

User Space

Write “Oxdeadbeef” to the opened file

Read-only file

Read-write file
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

[Diagram showing a process involving kernel and user space interactions, with labeling of file paths and states.]

- Write “0xdeadbeef” to the opened file
- Color-coding indicates read-only and read-write files:
  - Yellow: Read-only file
  - Pink: Read-write file
Attacking Open File Credential

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

Write “0xdeadbeef” to the opened file
Attacking Open File Credential

Result: Writing content to read-only files

Write “0xdeadbeef” to the opened file

Successfully written to /etc/passwd

Open File Credential

Kernel

User Space

Read-only file
Read-write file
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* objects are in **dedicated** caches.
- Most vulnerabilities happen in **generic** caches.

<table>
<thead>
<tr>
<th>kmalloc-256</th>
<th>obj_a</th>
<th>obj_b</th>
<th>obj_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>cred_jar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cred</td>
<td>cred</td>
<td>cred</td>
</tr>
<tr>
<td>filp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>file</td>
<td>file</td>
<td>file</td>
</tr>
</tbody>
</table>
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

```c
struct request_key_auth {
    struct rcu_head
    struct key
    struct key
    const struct cred
    void
    size_t
    pid_t
    char
} __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: Winning 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 *permission check*

![Diagram showing process flow of file object swap and permission check](image-url)
Challenge 3: Wining the race

- The swap of file object happens before permission check.
Challenge 3: Winning 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.

![Diagram showing the process of file operations and security context.]
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*.

![Diagram showing the flow of operations from User Space to Kernel and back, including file access and swap operations.](image)
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*

The time window

Check perm

Write perm

Write to disk

Open File Credential

User Space

Kernel

Read-only file

Read-write file
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 v4.13

  - `import_iovec` copies userspace memory

```c
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](https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4557)
  - *Dead after v4.13*
Solution I: Userfaultfd & FUSE

• `vfs_writev after v4.13`

```c
ssize_t vfs_writev(...) {
    ...
    // import iovec to kernel, where kernel would be paused
    // using userfaultfd
    res = import_iovec(type, uvect, 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

```c
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

```c
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
- check perm
- Lock
- Do the write
- Unlock

Thread B
- check perm
- Lock
- Do the write
- Unlock
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
Android Kernel with CFI enabled*

* 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](https://github.com/markakd/DirtyCred)
# Overhead of The Defense

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