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

Yuhang Wu
PhD Student
yuhangw.blog

Xinyu Xing
Associate Professor
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

<table>
<thead>
<tr>
<th></th>
<th>Dirty Pipe</th>
<th>DirtyCred</th>
</tr>
</thead>
<tbody>
<tr>
<td>A generic exploitation method?</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Write a data-only, universal exploit?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Attack with CFI enabled (on Android)?</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Actively escape from container?</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Threat still exists?</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
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
```

- freed
- unprivileged
- freed
- freed
Task Credential

- **Struct cred** in kernel’s implementation

```
struct cred on kernel heap
```

![Diagram showing cred structure and its placement on the kernel heap]
Task Credential

- **Struct cred** in kernel’s implementation

```
struct cred on kernel heap
```
Open File Credentials

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

```
struct file

<table>
<thead>
<tr>
<th>Freed</th>
<th>f_op</th>
<th>f_op</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f_mode</td>
<td>f_mode</td>
</tr>
<tr>
<td></td>
<td>O_RDWR</td>
<td>O_RDWR</td>
</tr>
<tr>
<td></td>
<td>f_cred</td>
<td>f_cred</td>
</tr>
<tr>
<td></td>
<td>~/dummy</td>
<td>~/dummy</td>
</tr>
</tbody>
</table>
```

**struct file** on kernel heap
Open File Credentials

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

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

<table>
<thead>
<tr>
<th></th>
<th>f_op</th>
<th>f_op</th>
<th>f_op</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_mode</td>
<td>f_mode</td>
<td>f_mode</td>
<td></td>
</tr>
<tr>
<td>O_RDWR</td>
<td>O_RDWR</td>
<td>O_RDWR</td>
<td></td>
</tr>
<tr>
<td>f_cred</td>
<td>f_cred</td>
<td>f_cred</td>
<td></td>
</tr>
<tr>
<td>~/dummy</td>
<td>~/dummy</td>
<td>~/dummy</td>
<td></td>
</tr>
</tbody>
</table>

*struct file on kernel heap*
Open File Credentials

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

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

<table>
<thead>
<tr>
<th>f_op</th>
<th>f_op</th>
<th>f_op</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_mode</td>
<td>f_mode</td>
<td>f_mode</td>
</tr>
<tr>
<td>O_RDONLY</td>
<td>O_RDONLY</td>
<td>O_RDONLY</td>
</tr>
<tr>
<td>f_cred</td>
<td>f_cred</td>
<td>f_cred</td>
</tr>
<tr>
<td>~/dummy</td>
<td>~/dummy</td>
<td>~/dummy</td>
</tr>
</tbody>
</table>

**struct file** on kernel heap
Open File Credentials

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

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
```

<table>
<thead>
<tr>
<th>f_op</th>
<th>f_op</th>
<th>f_op</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_mode</td>
<td>f_mode</td>
<td>f_mode</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>O_RDWR</td>
<td>O_RDWR</td>
</tr>
<tr>
<td>f_cred</td>
<td>f_cred</td>
<td>f_cred</td>
</tr>
<tr>
<td>~/dummy</td>
<td>~/dummy</td>
<td>~/dummy</td>
</tr>
</tbody>
</table>

`struct file` on kernel heap
Open File Credentials

- Write content to file on disk if permission is granted

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
```

Write to disk

```
struct file on kernel heap
```

<table>
<thead>
<tr>
<th>f_op</th>
<th>f_op</th>
<th>f_op</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_mode</td>
<td>f_mode</td>
<td>f_mode</td>
</tr>
<tr>
<td>O_RDWR</td>
<td>O_RDWR</td>
<td>O_RDWR</td>
</tr>
<tr>
<td>f_cred</td>
<td>f_cred</td>
<td>f_cred</td>
</tr>
<tr>
<td>~/dummy</td>
<td>~/dummy</td>
<td>~/dummy</td>
</tr>
</tbody>
</table>

check perm
Open File Credentials

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

```c
int fd = open("~/dummy", O_RDONLY);
write(fd, "HACKED", 6);
```

<table>
<thead>
<tr>
<th>f_op</th>
<th>f_op</th>
<th>f_op</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f_mode</th>
<th>f_mode</th>
<th>f_mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_RDONLY</td>
<td>O_RDWR</td>
<td>O_RDWR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f_cred</th>
<th>f_cred</th>
<th>f_cred</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>~/dummy</th>
<th>~/dummy</th>
<th>~/dummy</th>
</tr>
</thead>
</table>

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);
```

**Failed write to disk**

```
check perm
```

**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

- freed
- unprivileged
- unprivileged
- unprivileged
- privileged

struct cred on kernel heap
Attacking Task Credentials

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

![Diagram showing the process of freeing a credential and its impact on security.]
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

```
freed | un-privileged | freed | un-privileged | privileged
```

*struct cred* on kernel heap
Attacking Task Credentials

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

```
struct cred on kernel heap
```
Attacking Task Credentials

Step 3. **Operate** as *privileged* user

```
| freed | un-privileged | privileged | un-privileged | privileged |
```

*struct cred* on kernel heap
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

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
```

![Diagram showing file structure on kernel heap]
Attacking Open File Credentials

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

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
```

![File structure diagram]

**struct file** on kernel heap
Attacking Open File Credentials

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

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
```

```
<table>
<thead>
<tr>
<th></th>
<th>f_op</th>
<th>f_op</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_mode</td>
<td>f_mode</td>
<td></td>
</tr>
<tr>
<td>O_RDWR</td>
<td>O_RDWR</td>
<td></td>
</tr>
<tr>
<td>f_cred</td>
<td>f_cred</td>
<td></td>
</tr>
<tr>
<td>~/dummy</td>
<td>~/dummy</td>
<td></td>
</tr>
</tbody>
</table>
```

struct file on kernel heap
Attacking Open File Credentials

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

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
```

```c
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

```c
int fd = open("~/dummy", O_RDONLY);
write(fd, "HACKED", 6);
```

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

---

**struct file on kernel heap**

- `f_op`
- `f_mode`
- `f_cred`
- `/etc/passwd`
- `~/dummy`

<table>
<thead>
<tr>
<th></th>
<th><code>f_op</code></th>
<th><code>f_op</code></th>
<th><code>f_op</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f_mode</code></td>
<td><code>f_mode</code></td>
<td><code>f_mode</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>O_RDONLY</code></td>
<td><code>O_RDONLY</code></td>
<td><code>O_RDONLY</code></td>
</tr>
<tr>
<td></td>
<td><code>O_RDWR</code></td>
<td><code>O_RDWR</code></td>
<td><code>O_RDWR</code></td>
</tr>
<tr>
<td></td>
<td><code>~/dummy</code></td>
<td><code>~/dummy</code></td>
<td><code>~/dummy</code></td>
</tr>
</tbody>
</table>
Attacking Open File Credentials

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

```plaintext
int fd = open("~/dummy", O_RDONLY);
write(fd, "HACKED", 6);
open("/etc/passwd", O_RDONLY);
```

**Check perm**

- `f_op`:
  - `/etc/passwd`: `f_op`:
  - `f_mode`: `O_RDONLY`
  - `f_cred`: `/dummy`

**Correct**

- `struct file` on kernel heap
- **Write to** `/etc/passwd` on disk

- Counts as an *dirty credential* attack.
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;
    *callout_info;
    callout_len;
    pid;
    op[8];
};
```

```
const struct cred {
    size_t size_t;
    pid_t pid;
    char char;
};
```
Pivoting Invalid-Write

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

For OOB

- Overflow object
  - *cred
  - Victim object

For UAF

- Credential object
  - *cred
  - Victim object
  - Overflow object

credential object
credential object
credential object
credential object
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

![Diagram showing two references to a vulnerable object in kernel memory.](image)
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**

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

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

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

4. **Step 4.** Free the credentials with the left dangling reference
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*

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
close(fd);
```

Write to `/etc/passwd` failed
Challenge 3: Stabilizing File Exploitation

- The swap of file object happens after file write.

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
close(fd);
```
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

```c
int fd = open("~/dummy", O_RDWR);
write(fd, "HACKED", 6);
close(fd);
```

Write to `/etc/passwd`
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

```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://en.wikipedia.org/wiki/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: 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

```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
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](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://zplin.me
zplin@u.northwestern.edu