A novel page-UAF exploit strategy to privilege escalation in Linux systems.

Authors: Jiayi Hu, Jinmeng Zhou, Qi Tang, Wenbo Shen, Zhiyun Qian

0 - Introduction

Many critical heap objects are allocated in dedicated slab caches in the Linux kernel. Corrupting such objects requires unreliable cross-cache corruption methods [1][2][3][4], due to the unstable page-level fengshui. To overcome this, we propose a novel page-UAF-based exploit strategy to overwrite critical heap objects located in dedicated slab caches. This method can achieve local privilege escalation without requiring any pre-existing infoleak primitive (i.e., no need to bypass KASLR), but it can help derive the infoleak primitive and arbitrary write primitive.

1 - Exploitation background

1.0 - Object-level UAF

The standard exploitation of object-level UAF leverages the invalid use of a freed heap object (vulnerable object) via a dangling pointer. The use can corrupt another object (target object) that takes the place of the freed slot. For instance, to hijack the control

flow, we can corrupt a function pointer within the target objects using the target value obtained from a pre-existing information leakage primitive. The object-level UAF exploitation requires the target object to reuse the freed slot (previously the vulnerable object). Thus, the vulnerable object and the target object must be allocated in the same slab cache (typically in standard caches, e.g., kmalloc-192) and require object-level heap fengshui to manipulate the memory layout. Dirtycred utilizes the critical-object-UAF to achieve privilege escalation, using cred and file objects [4].

1.1 - Previous page-level heap fengshui in cross-cache attacks

Nowadays many critical target objects are allocated in the dedicated caches (e.g., cred), rendering simple object-level heap fengshui ineffective. To corrupt these objects, we have to launch cross-cache attacks that usually rely on page-level heap fengshui [1][2][6]. The page-level heap fengshui of OOB and UAF are different.

For OOB bugs, typically, an attacker would trigger overflows onto a subsequent page [1] [2]. For instance, after saturating a page (referred to as "page 1") with many vulnerable objects, the attacker can allocate target objects to make the slab cache request the following page (referred to as "page 2"). Consequently, the last vulnerable object on page 1 becomes adjacent to the first target object on page 2, facilitating overflow from the former to the latter. And we can't make sure that the vulnerability object is the last object on page 1 due to some protections such as CONFIG_SLAB_FREELIST_RANDOM. Otherwise, This page-level heap fengshui requires the manipulation of page allocation in the buddy system, making the exploits more unstable.

For UAF bugs, a common strategy is to convert an object-level UAF into a page-level UAF. Specifically, the idea is to release many vulnerable objects (one or more of which are freed illegally with dangling pointers) to force the entire page to be freed. After that, it allocates many target objects so that the same page will be repurposed for the objects' dedicated slab cache. Finally, the dangling pointer can be used to overwrite the target object in another cache by page-level UAF.

2 - General idea

Our idea is to induce page-level UAF by causing the free() of specialized objects (we term them "bridge objects") that correspond to memory pages. A bridge object is of a type that contains a pointer to the struct page and is located in standard slab caches. Among others, struct pipe_buffer is such an example with a field named page (and located in kmalloc-192), and we list the code snippet below:

```
struct pipe_buffer {
    struct page *page;
    unsigned int offset, len;
    const struct pipe_buf_operations *ops;
    unsigned int flags;
    unsigned long private;
};
```

Freeing such bridge objects will automatically lead the corresponding 4KB page to be freed, effectively leading to a page-level UAF primitive. This is because an object of type struct page (which by itself is 64 bytes in recent Linux kernels) is used to manage a 4KB physical page in the kernel. Through such a bridge object, an attacker can read/write the entire 4KB memory, which can be reclaimed for other objects (including those stored in dedicated slab caches, e.g., struct cred). This is because the freed pages can be returned to the buddy allocator for future slab allocations. Finally, attackers can overwrite the critical objects in such slabs to achieve privilege escalation (e.g., setting the uid in cred to 0).

3 - Exploitation steps

The page-UAF exploit strategy consists of two main steps: page-level UAF construction and critical object corruption, as discussed in the following.

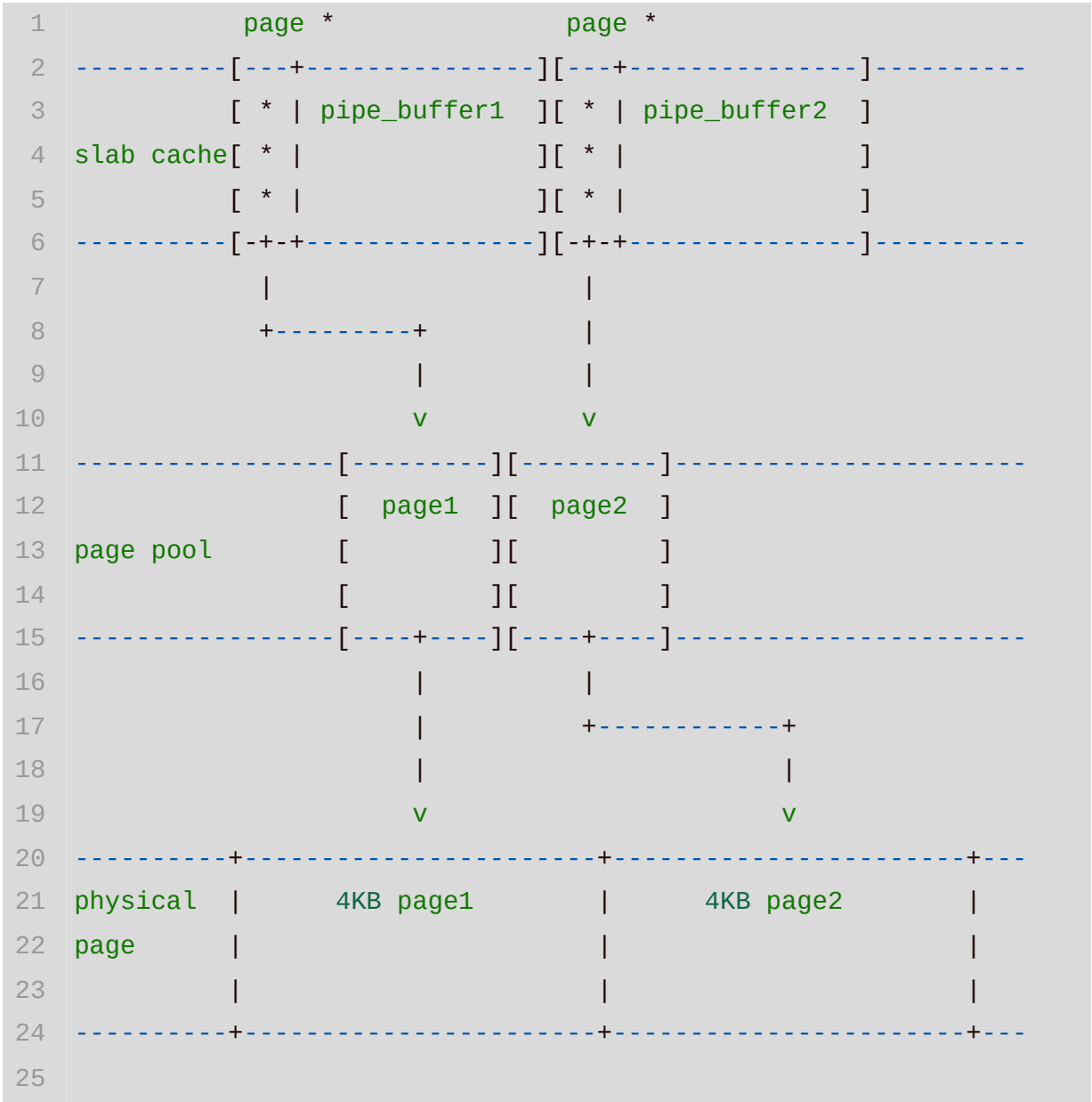
3.0 - Page-level UAF construction

Starting from a memory corruption bug that provides an invalid write of the memory, e.g., OOB, UAF, or double free, we can first spray multiple bridge objects (at least two) that co-locate with the vulnerable object.

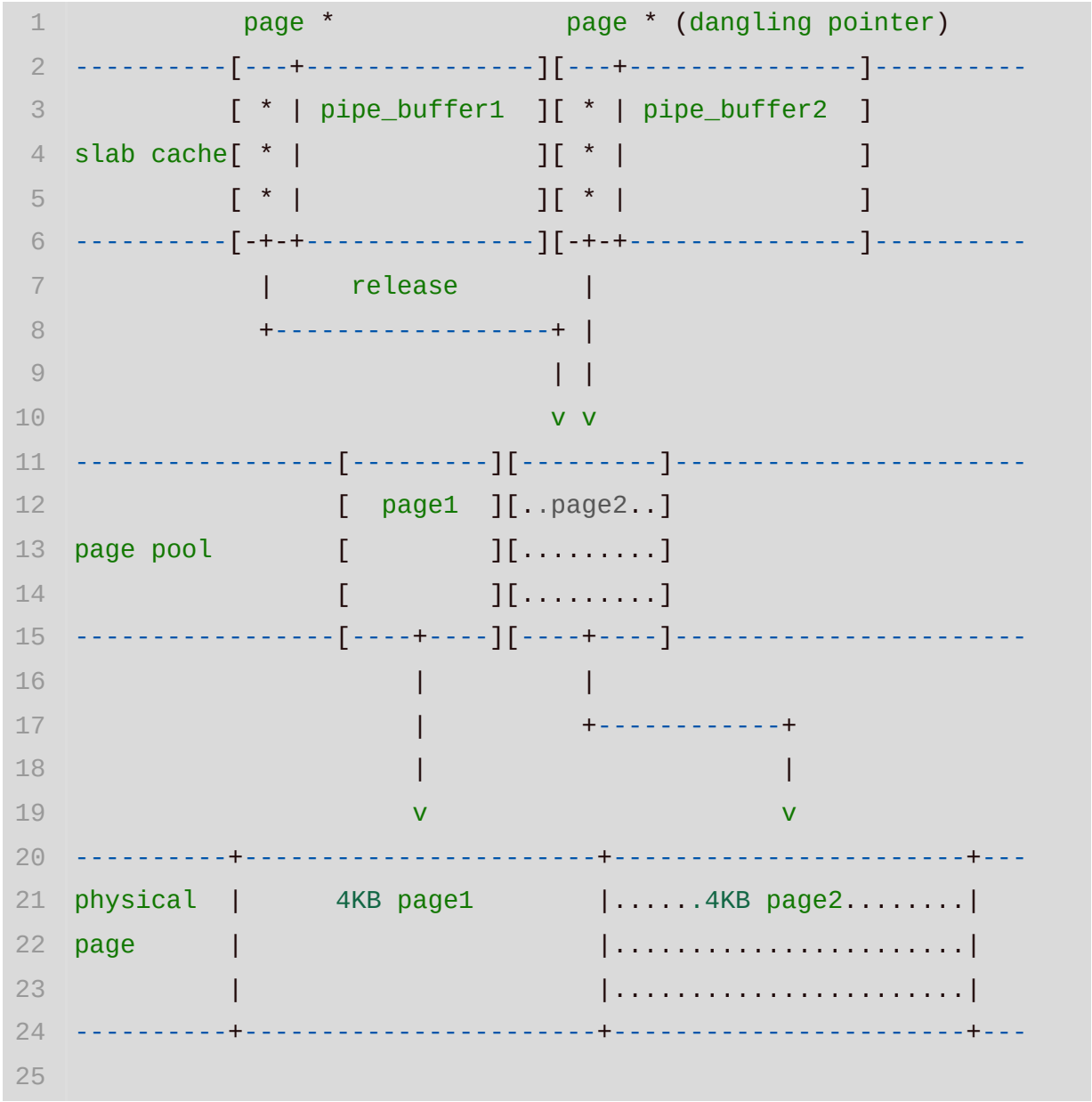
Specifically, for bugs with standard OOB or UAF write primitives, we can use the write primitive to corrupt the page pointer field in a bridge object (e.g., the first field of pipe_buffer) such that it points to another 64-byte page object nearby. This effectively causes two pointers to point to the same object. A user-space program can trigger free_pages() on one of the objects (e.g., by calling close()), which will create a dangling pointer to the freed page object and the corresponding physical page. In other words, we can read/write the corresponding physical page that is now considered freed by the OS kernel. For example, one can write to a pipe, which will lead to a write of the physical page via the pipe_buffer object.

For bugs that have double-free primitives, which can often be achieved from UAF by triggering the free() operation twice, we execute the following. For the first free, we spray a harmless object (e.g., msg_msg) to take the freed slot. The object should take attacker-controlled value from the user space. Then, we trigger the kernel code to write the harmless object until reaching a certain offset, using the FUSE technique [1] to stop the writing right before a planned offset – corresponding to the page pointer field of a planned bridge object. Now, we trigger the free for a second time to spray the planned bridge object to take the slot. The writing process is restarted to continue overwriting the lower bits of the page pointer field, which leads to a page UAF. Previously, to trigger page-level frees from double frees, one had to release an entire slab and then do a cross-cache technique [1][2], whereas no such requirement is needed in our exploit method.

For a standard OOB and UAF bug, we use the following figures to better demonstrate the memory layout in various exploitation steps. At first, we have at least two bridge objects (pipe_buffer1 and pipe_buffer2) that contain a field of type struct page*. The pointers point to two adjacent page objects that correspond to two continuous 4KB physical pages. Below is the memory layout before triggering the OOB/UAF corruption:



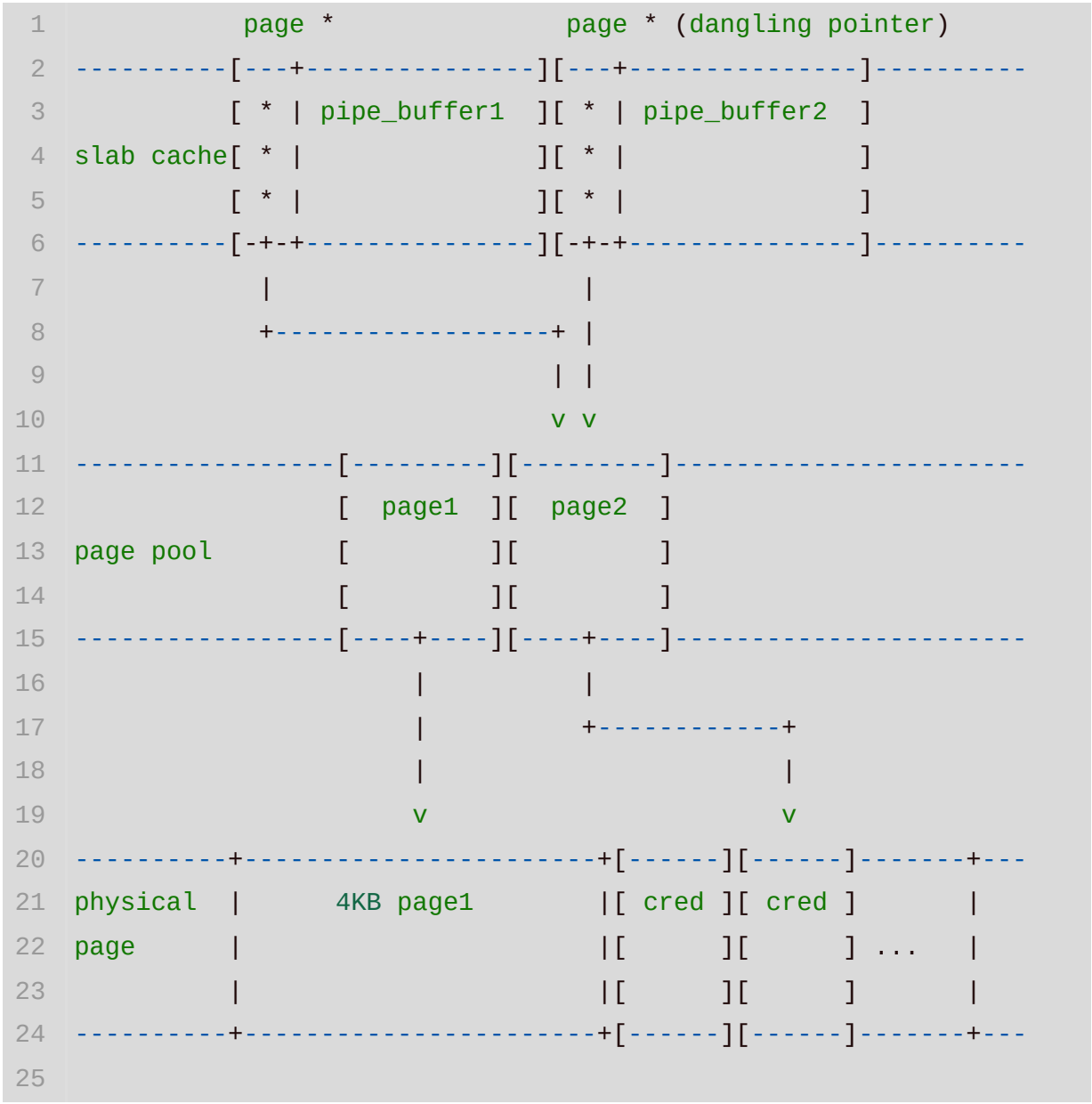
Now, we corrupt the page pointer within pipe_buffer1 to cause it to point to page2 – this can be achieved by overwriting the lower bits of the pointer field, similar to what DirtyCred requires [4]. This makes the pointers in both pipe_buffer1 and pipe_buffer2 point to the same page object (as shown below). At this point, we can release the page2 object (along with its corresponding 4KB physical page) through the pointer in pipe_buffer1. As a result, the page pointer within pipe_buffer2 becomes a dangling pointer pointing to the freed page. The memory layout after triggering corruption is shown in the figure below:



3.1 - Critical object corruption through page-level UAF

Now that we have a freed physical page and a dangling pointer that can read/write it. It is fairly easy to then allocate and corrupt critical heap objects. This is because the critical heap objects are eventually allocated through the buddy allocator at the page granularity. Therefore, attackers can spay the heap objects into the freed page as long as they have

already exhausted all existing slab caches. For example, we can overwrite the cred object through the dangling pointer, as shown in the following figure.



4 - Evaluation results

We analyzed the bridge objects in the Linux kernel v5.14. We sample 26 recent Linux kernel CVEs that are either OOB, UAF, or double-free from 2020 to 2023. This includes 24 vulnerabilities from prior work [4], and 2 additional OOB ones missed by the prior work.

4.0 - Bridge objects

We found many objects containing page pointers, which reside in various standard slab caches of different sizes. The Linux kernel has interfaces to read/write the physical pages through the page pointers, such as copy_page_from_iter and copy_page_to_iter (iter usually represents the user buffer). We use the struct types with inside fields to show the

bridge objects and the slab caches used to allocate them as follows:

```
address_space->i_pages (adix_tree_node_cachep)
configfs_buffer->page (kmalloc-128)
pipe_buffer->page (variable size)
st_buffer->reserved_pages (variable size)
bio_vec->bv_page (variable size)
wait_page_queue->page  (variable size)
xfrm_state->xfrag->page (kmalloc-1k)
pipe_inode_info->tmp_page (kmalloc-192)
lbuf->l_page (kmalloc-128)
skb_shared_info->frags->bv_page (variable size)
orangefs_bufmap_desc ->page_array (variable size)
```

We denote some objects with "variable size" whose sizes could be different at runtime due to different allocation paths. Some objects are allocated as arrays with variable sizes using standard kmalloc caches, such as pipe_buffer, which can be general to many bugs having memory corruption on the standard caches (e.g., kmalloc-192).

In addition, certain functions, such as process_vm_rw_core(), allocate page pointers into the heap and store them in a page array, which can also be used in the page-UAF strategy.

4.1 - Real-World Exploitation Experiments

We confirm that 18 out of the 26 CVEs are exploitable. Furthermore, we develop end-to-end exploits against CVE-2023-5345, CVE-2022-0995, CVE-2022-0185, CVE-2021-22555, and CVE-2021-22600. The exploits spray bridge objects to construct page UAF – we specifically target pipe_buffer->page and configfs_buffer->page. These exploits are generally easier and more stable because there is no need for infoleak, cross-cache attacks, and page-level fengshui. We ran our exploits to test the stability. Each exploit is run 10 times, and 9 or 10 out of 10 are successful.

4.2 - Comparison of our page-UAF with previous methods

As we can see, our page-UAF exploit strategy can achieve privilege escalation without requiring infoleaks or bypassing KASLR. But it offers the capability to leak information, via the read of physical page memory. It can also help derive arbitrary write primitives via writing to objects that have pointers in the physical page. Overall, the exploit strategy reduces the requirements for memory layout manipulation to use object-level heap fengshui – only the initial fengshui is required to derive the page-level UAF.

To our knowledge, we have not seen widespread use of bridge objects to achieve page-level UAF in real-world exploits. The only example we are aware of is a CTF competition [3] that uses the struct pipe_buffer as a bridge object. However, we note that struct pipe_buffer is being isolated into kmalloc-cg slab using flag GFP_ACCOUNT after Linux kernel v5.14. Therefore, other bridge objects, such as the ones we showed, would be required to launch the generalized page-level exploits.

Exploits using our page-UAF strategy are relatively stable due to the provided information leakage primitive and avoidance of page-level heap fengshui. We ran our exploit of five end-to-end exploits using CVE-2022-0995 and CVE-2022-0185. Each exploit is run 10 times, and 9 or 10 out of 10 are successful without any crashes. This is attributed to the fact that the exploits have a built-in feedback mechanism (i.e., read of the physical page) that helps make sure the final write is performed on the right target. Using the feedback mechanism, we can restart the page-level UAF if the expected target is not detected.

5 - References

[1] CVE-2022-27666: Exploit esp6 modules in Linux kernel. https://etenal.me/archives/1825

[2] Reviving Exploits Against Cred Structs - Six Byte Cross Cache Overflow to Leakless Data-Oriented Kernel Pwnage. https://www.willsroot.io/2022/08/reviving-exploits-against-cred-struct.html

[3] [D^3CTF 2023] d3kcache: From null-byte cross-cache overflow to infinite arbitrary read & write in physical memory space. https://github.com/arttnba3/D3CTF2023_d3kcache

[4] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. Dirtycred: escalating privilege in Linux kernel. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security

[5] FUSE for Linux exploitation 101. https://exploiter.dev/blog/2022/FUSE-exploit.html.

[6] Understanding Dirty Pagetable - m0leCon Finals 2023 CTF Writeup https://ptr-yudai.hatenablog.com/entry/2023/12/08/093606