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:
Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. 2022. DirtyCred: Escalating Privilege in Linux Kernel. In *Proceedings of Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA., November 7–11, 2022 (CCS '22),* 15 pages.
https://doi.org/10.1145/3548606.3560585

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560585

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 credential object with a high privileged one. This practice confuses the Linux kernel into thinking that an unprivileged user could gain permission to operate on high privileged files or processes. As such, we name this exploitation method after DirtyCred.

To perform exploitation, DirtyCred confronts three critical technical challenges. First, it needs to pivot a vulnerability's capability to the one useful for credential object swap because vulnerabilities in different types provide different capabilities in memory corruption, which may not look sufficient for a credential object swap at first glance. Second, DirtyCred needs to control the time window to launch object swap strictly. As we will discuss in Section 3, the time window valuable for DirtyCred is short. Without a practical mechanism to extend the time window, the exploitation would be unstable. Third, DirtyCred needs to find an effective mechanism

that allows an unprivileged user to allocate privileged credentials in an active fashion because failing to have this ability would make the credential object swap ineffectual.

To address the technical challenges above, we first introduce a series of vulnerability pivoting schemes that allow us to convert any heap-based vulnerability into an ability to free credential objects in an invalid manner. Second, we leverage three different kernel features – userfaultfd [38], FUSE [37], and lock in filesystem – to extend the time window needed for object swap and thus stabilize exploitation. Last but not least, we employ various kernel mechanisms to spawn high privileged threads from both userspace and kernel space and thus actively allocate privileged objects. In this work, we evaluate DirtyCred's exploitability by using 24 real-world kernel vulnerabilities. We surprisingly discovered that DirtyCred could demonstrate privilege escalation on 16 vulnerabilities and container escape. We shared our newly proposed exploitation method with Google Vulnerability Rewards Program (kCTF VRP [20]) and received their acknowledgment and $20,000 bounty reward.

With the strong exploitability demonstration and the lack of effective defenses, we believe that DirtyCred could soon become a severe threat to Linux if the community does not take immediate action to explore and deploy a new defense mechanism. As a result, following our new exploitation approach, we further proposed a new Linux kernel defense mechanism. The basic idea of this defense is to host high privileged and low privileged objects in non-overlapping memory regions. In this work, we do it by utilizing the vmalloc region to store high privileged objects and the normal region for low privileged ones. We implemented this defense as a Linux kernel prototype and evaluated its performance using a standard benchmark. We show that our defense primarily introduces negligible overhead. For some operations involving file operations, it demonstrates only moderate performance overhead.

Compared with existing kernel exploitation techniques, DirtyCred has many unique characteristics. First, it is a general exploitation approach because it enables privilege escalation for arbitrary heap-based vulnerabilities. Second, it could significantly unload the burden of exploit migration because – following DirtyCred – one could craft an exploit that can transfer from one kernel version or architecture to another without any modification. Third, it could bypass many powerful kernel protection and exploit mitigation mechanisms (e.g., CFI [15], KASLR [14], SMEP/SMAP [7, 29], KPTI [8], etc.). Last, it could go beyond privilege escalation, leading to more severe security problems, such as rooting Android and escaping a container.

In summary, this paper makes the following contributions.

- We propose a new, general exploitation method – DirtyCred – that could circumvent widely adopted kernel protections and exploit mitigation and thus fulfill privilege escalation in the Linux system.
- We demonstrate that DirtyCred could manifest strong exploitability on many real-world Linux kernel vulnerabilities. We also show that the exploitable objects useful for DirtyCred are diverse and abundant.
- We analyze existing kernel defenses' limitations and propose a new defense mechanism. We implemented this defense as a prototype on Linux Kernel, showing it introduces negligible and moderate performance overhead.

The rest of this paper is organized as follows. Section 2 introduces the background needed for this research and discusses the threat model. Section 3 introduces the high-level idea of DirtyCred and summarizes the technical challenges that DirtyCred confronts. Section 4, 5, and 6 presents various techniques methods to handle the technical challenges. Section 7 evaluates the effectiveness of the proposed exploitation approach on real-world Linux kernel vulnerabilities. Section 8 introduces a new defense mechanism and evaluates its performance on standard benchmarks. Section 9 provides the discussion of related work, followed by the discussion of some related issues and future work in Section 10. Finally, we conclude the work in Section 11.

2 BACKGROUND & THREAT MODEL

This section introduces some technical background necessary for understanding our newly proposed exploitation method. Besides, we discuss our threat model and assumptions.

2.1 Credentials in Linux kernel

As is defined in [26], credentials refer to some kernel properties that contain privilege information. Through these properties, the Linux kernel could examine users' access privileges. In the Linux kernel, credentials are implemented as kernel objects which carry privilege information. To the best of our knowledge, those objects include "cred", "file", and "inode". In this paper, we designed our exploitation methods by using only "cred" and "file" objects. We excluded the "inode" object because it can only be allocated when a new file is created on the filesystem, which does not provide sufficient flexibility for memory manipulation (a critical operation in a successful program exploitation). We provide some necessary background for "cred", "file", and "inode", objects in the following.

Every Linux task contains a pointer referencing a 'cred' object. The 'cred' object contains the UID field, indicating the task privilege. For example, `GLOBAL_ROOT_UID` indicates the task has the root privilege. When a task attempts to access a resource (e.g., a file), the kernel checks the UID in the task's 'cred' object, determining whether the access could be granted. In addition to UID, the 'cred' object also contains capability. The capability specifies the task's fine-grained privilege. For example, `CAP_NET_BIND_SERVICE` indicates the task could bind a socket to an internet domain privileged port. For each task, their credential is configurable. When altering the task credential, the kernel follows the copy-and-replace principle. It first copies the credential. Second, it modifies the copy. Finally, it changes that cred pointer in the task to the newly modified copy. In Linux, each task may alter only its own credentials.

In the Linux kernel, every file comes with its owner's UID and GID, other users' access permission, and capability. For executable files, they also have SUID/SGID flags indicating special permission that allows other users to run with the owner's privileges. In the Linux kernel implementation, each file is tied to an 'inode' object linking to the credentials. When a task attempts to open a file, the kernel invokes function `inode_permision`, checking the inode and the corresponding permission before granting the file access. After a file is opened, the kernel delinks the credentials from the 'inode' object

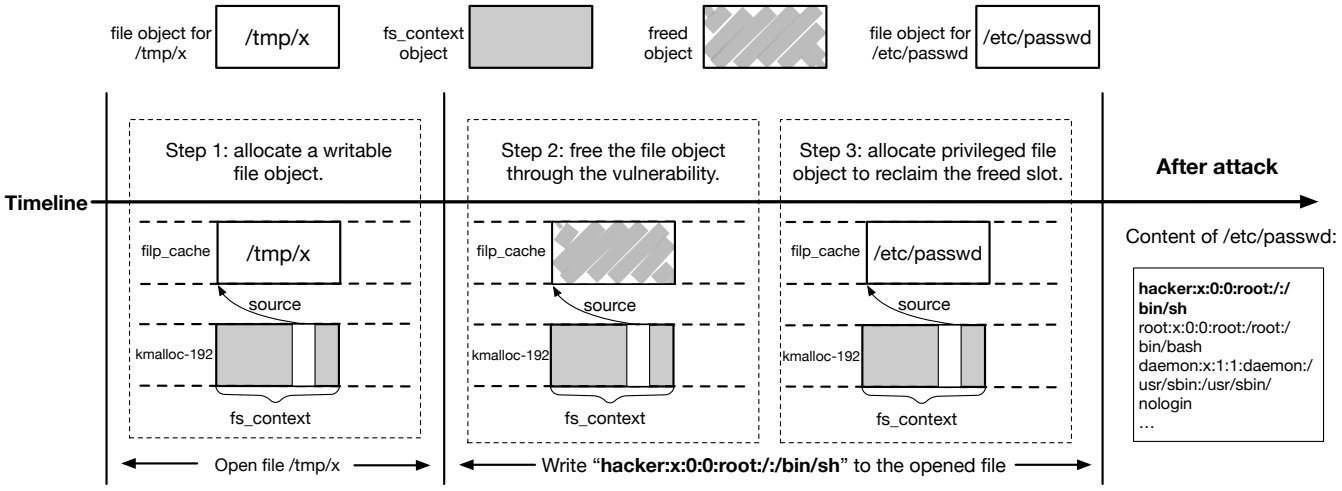

Figure 1: The overview of exploiting CVE-2021-4154, the write operation to the opened file starts between step 1 and step 2 and finishes after step 3.

and attaches them to the 'file' object. In addition to maintaining the credentials, the 'file' object also contains the file's read/write permission. Through the 'file' object, the kernel could index to the cred object and thus examine the privilege. Besides, it could check read/write permission and thus ensure that a task does not write data to a file opened in a read-only mode.

2.2 Kernel Heap Memory Management

Linux kernel designs memory allocators to manage small memory allocation to improve performance and prevent fragmentation. Although there are three different memory allocators in the Linux kernel, they follow the same high-level design. To be specific, they all use caches to maintain the same size memory. For each cache, the kernel allocates memory pages and divides the memory into multiple pieces of the same size, and each piece is a memory slot used for hosting an object. When the memory page for a cache is exhausted, the kernel allocates new pages for the cache. If a cache no longer uses the memory page, i.e., all the objects on the memory page are freed, the kernel recycles the memory page accordingly. There are two main kinds of caches in the Linux kernel, as is briefly described below.

Generic Caches. Linux kernel has different generic caches to allocate different-sized memory. When allocating memory from the generic caches, the kernel will first round up the requested size and find the cache matching the size request. Then, it allocates a memory slot from the corresponding cache. In the Linux kernel, if an allocation request does not specify which kinds of caches it allocates from, the allocation by default occurs at generic caches. For allocations that fall into the same generic cache, they may share the same memory address as they may be maintained on the same memory page.

Dedicated Caches. Linux kernel creates dedicated caches for performance and security purposes. As some objects are used frequently in the kernel, dedicating caches for these objects could reduce the time spent on their allocation and thus improve system performance. Allocations that fall into dedicated caches do not share the same memory page with general allocations. As a result, objects allocated in the generic cache are not adjacent to objects in dedicated caches. It could be viewed as cache-level isolation, which mitigates the overflow threat from objects in general caches.

2.3 Threat Model

In our threat model, we assume an unprivileged user has local access to the Linux system, aiming to exploit a heap memory corruption vulnerability in the kernel and thus escalate his/her privilege. Besides, we assume that Linux enables all the exploit mitigation and kernel protection mechanisms available in the upstream kernel (version 5.15). These mechanisms include KASLR, SMAP, SMEP, CFI [7, 14, 15, 29], KPTI [8], etc. With these mitigation and protections, the kernel address is randomized, the kernel cannot access user-space memory directly during execution, and its control-flow integrity is guaranteed. Last but not least, we do not assume there is a hardware side channel that could facilitate kernel exploitation.

3 TECHNICAL OVERVIEW & CHALLENGES

In this section, we first introduce the high-level idea of DirtyCred by using a real-world example. Then, we analyze and discuss the technical challenges that DirtyCred needs to address.

3.1 Overview

We take a real-world Linux kernel vulnerability (CVE-2021-4154 [9]) as an example showcasing how DirtyCred works at a high level. CVE-2021-4154 is due to a type confusion error that a file object is referenced by the source field of `fs_context` object incorrectly. In the Linux kernel, the lifetime of a file object is maintained through reference count mechanism. The file object will be free automatically when the reference count becomes zero, meaning that the file object is no longer being used. However, by triggering the vulnerability, the kernel will free the file object invalidly even if the file is still in use.

As is depicted in Figure 1, DirtyCred first opens a writable file "/tmp/x", which will allocate a writable file object in kernel. By triggering the vulnerability, the source pointer will reference the file object in the corresponding cache. Then, DirtyCred attempts to write content to the opened file "/tmp/x". Prior to the actual content write, the Linux kernel checks whether the current file has permission to write and whether the position is writable, etc. After passing the check, DirtyCred holds on to this actual file writing action and enters the second step. In this step, DirtyCred triggers the free site of the `fs_context` object to deallocate the file object, which leaves the file object as a freed memory spot.

Then, in the third step, DirtyCred opens a read-only file "/etc/passwd", which triggers the kernel to allocate the file object for "/etc/passwd". As is illustrated in Figure 1, the newly allocated file object takes over the freed spot. After this setup, DirtyCred will release its on-hold write action, and the kernel will perform the actual content writing. Since the file object has been swapped, the content on hold will be redirected to the read-only file "/etc/passwd". Assuming that content written to "/etc/password" is "hacker:x:0:0:root:/:/bin/sh", a bad actor could use this scheme to inject a privileged account and thus fulfill privilege escalation.

The example above is just a demonstration indicating how DirtyCred uses file objects for exploitation. As is mentioned in Section 2, in addition to "file" objects, "cred" objects are also considered credential objects. Like the file swap showcased above, a bad actor can also use a similar idea to swap cred objects and thus fulfill privilege escalation. Due to the space limit, we do not elaborate. Readers interested in cred object exploitation could refer to our exploitation demo published at [2].

From the real-world example described above, we can observe that DirtyCred does not alter the control flow but exploits kernel memory management's nature to manipulate objects in the memory. As a result, many existing defenses that prevent control flow tampering do not affect DirtyCred's exploitation. While some recent research works enable kernel defense by re-designing the memory management (e.g. AUTOSLAB [34]), they are also ineffective in blocking DirtyCred. As we will discuss in Section 8, the newly proposed memory management methods are still in coarse-granularity, not sufficient for hindering our memory manipulation.

3.2 Technical Challenges

While the example above illustrates how DirtyCred performs exploitation and thus fulfills privilege escalation, there are still many technical details that need to be further clarified and many technical challenges that need to be tackled.

- As is mentioned above, DirtyCred needs an invalid-free capability to deallocate a low-privileged object (e.g., a file object with write permission) and then reallocate a high-privileged one (e.g., a file object with read-only permission). In practice, a kernel vulnerability may not always provide us with such a capability. For example, a vulnerability may provide only out-of-bound overwriting capability instead of an invalid-free directly against a credential object. Therefore, DirtyCred needs corresponding approaches to pivot vulnerability's capability for vulnerabilities with different capabilities. In Section 4, we, therefore, describe how to pivot capability for different types of kernel vulnerabilities.
- As described in the example above, DirtyCred needs to hold on to the actual file writing after completing the permission check and prior to the file object swap. However, holding on to the actual writing is challenging. In the Linux kernel, the permission check and the actual content writing happen back-to-back quickly. Without a practical scheme to accurately control the occurrence of file object swap, the exploitation would be inevitably unstable. In Section 5, we introduce a series of effective mechanisms that guarantee the file object swap could occur at the desired time window.
- As is discussed above, one of the most critical steps in DirtyCred is to use a high-privilege credentials to replace the low-privilege one. To do that, DirtyCred allocates high-privileged objects, taking over the freed memory spot. However, it is challenging for a low-privilege user to allocate high-privileged credentials. While simply waiting for the activities from privileged users may potentially resolve the problem, such a passive strategy greatly influences exploitation stability. First, DirtyCred has no clue when the desired memory spot could be reclaimed and thus continues its consecutive exploitation. Second, DirtyCred has no control over the newly allocated objects. Therefore, it is possible that the object that takes over the desired memory slot does not have the expected privilege level. In Section 6, we introduce a userspace mechanism and a kernel space scheme to address this problem.

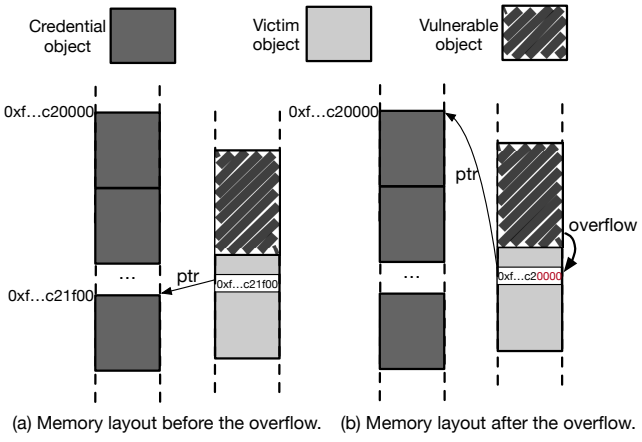

Figure 2: The memory layout before and after converting a heap overflow capability into the ability to deallocate a credential object.

4 PIVOTING VULNERABILITY CAPABILITY

As is demonstrated in the example depicted in Figure 1, the kernel vulnerability cataloged as CVE-2021-4154 provides DirtyCred with the ability to deallocate the file object in an invalid fashion. However, in practice, a vulnerability may not demonstrate such a capability. For example, a double-free (DF) or use-after-free (UAF) capability may not directly tie to a credential object. Some vulnerabilities, like out-of-bound (OOB) access, do not have invalid free capability. To this end, DirtyCred needs to pivot a vulnerability's capability. In

the following, we describe how we design DirtyCred for capability pivot.

4.1 Pivoting OOB & UAF Write

Given an OOB vulnerability or a UAF vulnerability with the capability of overwriting data in a cache, DirtyCred first identifies an object (i.e., victim object) that shares the same cache and encloses a pointer referencing a credential object. Then, it utilizes heap manipulation techniques [6, 49] to allocate the object at the memory region where the overwriting happens. As shown in Figure 2 (a), to pivot an OOB vulnerability, the victim object is just right after the vulnerable object. Using the overwriting capability, DirtyCred further modifies the object-enclosed pointer. More specifically, DirtyCred uses the overwriting capability to write zero to the last two bytes of the pointer referencing the credential object (see Figure 2 (b)).

Recall that a cache is organized on contiguous pages. In the Linux kernel, the address of a memory page is always in a format where the last byte is zero. When allocating objects in a new cache, the object starts from the beginning of the memory page. As a result, the zero-byte overwriting above would force the pointer to reference the beginning of a memory page. For example, as is illustrated in Figure 2 (b), after nullifying the last two bytes of the pointer referencing a credential object, the pointer references to the beginning of a memory page in which another credential object resides.

As is shown in Figure 2 (b), after the pointer manipulation, DirtyCred obtains an additional reference to the first object of the memory page. We argue that this additional object reference means success in capability pivot. The reason is that kernel could free the object normally, leaving the pointer in the victim object as a dangling pointer. Then, following the similar procedure described in Section 3, DirtyCred could perform a heap spray, occupy that freed spot with a high-privileged credential object, and thus fulfill privilege escalation.

4.2 Pivoting DF

In the Linux kernel, general caches (e.g., `kmalloc-96`) and dedicated caches (e.g., `cred_jar`) are isolated. The objects enclosed in these caches have no overlap. However, the Linux kernel has a recycling mechanism. When destroying a memory cache, it recycles the corresponding unused memory pages and then assigns the recycled pages to the caches that need more space. This characteristic enables cross-cache memory manipulation, providing DirtyCred with the ability to pivot capability for double-free vulnerabilities.

Figure 3 shows the procedure of how DirtyCred converts a double-free capability to the capability needed for privileged object swap. First, DirtyCred allocates many objects in the cache where the vulnerability occurs. Among these newly allocated objects, there is a vulnerable object. Using two different pointers, DirtyCred could deallocate the vulnerable object twice in an invalid fashion. Since the number of allocations is large, DirtyCred could ensure that a cache is full of newly allocated objects after the massive object allocation (see Figure 3 (a)).

Following the massive allocation, DirtyCred utilizes the first pointer to deallocate the vulnerable object in an invalid fashion,

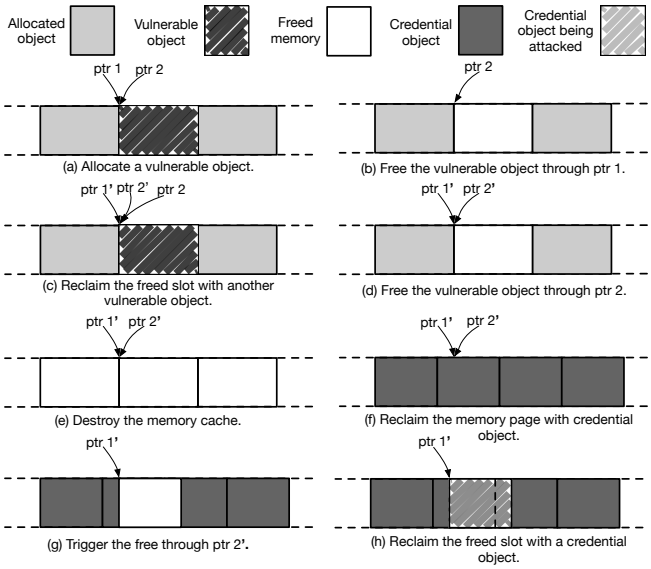

Figure 3: The step-by-step example demonstrating converting a double-free capability into the ability to deallocate a credential object.

```
struct iovec
{
    void __user *iov_base; /* BSD uses caddr_t (1003.1g requires
    ↪  void *) */
    __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
};

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
}
```

Listing 1: The code snippet of vfs_writev function in kernel before 4.13.

leaving the second pointer behind (see Figure 3 (b)). Then, it reallocates the vulnerable object, taking over the freed memory spot. As is shown in Figure 3 (c), after the reallocation, there are three pointers referencing the vulnerable object. One is the pointer left behind by the first vulnerable object. The other two are those tied to the double-free capability against the newly allocated vulnerable object.

Using one of the three-pointers referencing the vulnerable object, DirtyCred further deallocates the newly allocated vulnerable object, leaving a freed memory spot referenced by two dangling pointers (see Figure 3 (d)). As mentioned above, the Linux kernel recycles the memory page and assigns it to another cache if a cache contains no allocated objects. Therefore, after the deallocation of the vulnerable

object, DirtyCred further deallocates other objects in the cache and thus frees up the cache accordingly (see Figure 3 (e)).

On the recycled memory page, the kernel creates a new cache that stores credential objects. The new cache divides the page memory into slots. As is depicted in Figure 3 (f), if the size of the vulnerable object is different from that of the credential object, the address of the credential object will not align with that of the vulnerable object, making the two remaining pointers reference the middle of the credential object. In this memory status, DirtyCred cannot follow the exploitation procedure described in Section 3 because the success of exploitation requires the ability to deallocate a credential object.

To address this problem, DirtyCred first uses one of the remaining pointers to deallocate the credential object in the middle. As is illustrated in Figure 3 (g), after this deallocation, the kernel creates a freed memory spot. This freed spot is the size as same as a credential object. Therefore, when DirtyCred allocates a new credential object, the kernel fills that freed spot with the new credential object. As we can observe in Figure 3 (h), after the freed spot is taken over, the last remaining pointer references the newly allocated credential object. It implies the success of the capability pivot. The reason is that DirtyCred could utilize the remaining pointer to deallocate the credential object in an invalid fashion and then perform an object swap for privilege escalation.

5 EXTENDING TIME WINDOW

Recall that the Linux kernel needs to check file permission before performing a file write operation. DirtyCred needs to perform a file object swap between the permission check and the actual file writing. However, this window is too short to perform a successful exploitation since the swapping need to trigger the vulnerability and do heap layout manipulation, which might take a few seconds. To address this problem, DirtyCred utilizes several techniques to extend this time window to ensure it is larger than the time spent for the swapping process. Here, we describe these techniques and discuss how they could facilitate exploitation.

5.1 Exploitation of Userfaultfd & FUSE

Userfaultfd [38] and FUSE [37] are two critical features in the Linux kernel. The userfaultfd feature allows userspace to handle page faults. When a page fault is triggered on the memory registered in userfaultfd, the user registered page fault handler will get notified to handle the page fault. Unlike userfaultfd, FUSE is a userspace filesystem framework, allowing users to implement the userspace filesystem. Users could register their handler for the implemented userspace filesystem to specify how to respond to file operating requests. Both userfaultfd and FUSE could be utilized to pause Linux kernel execution as long as the user want. For userfaultfd, an adversary could register a page fault handler for a memory page. When the kernel attempts to access that memory and triggers a page fault, the registered handler will be invoked, allowing the adversary to pause kernel execution. For FUSE, an adversary could allocate the memory from the userspace filesystem. When the kernel accesses this memory, it invokes the pre-defined file access handler and thus pauses kernel execution.

```
1  ssize_t vfs_writev(...)
2  {
3      ...
4      // import iovec to kernel, where kernel would be paused
5      // using userfaultfd
6      res = import_iovec(type, uvector, nr_segs,
7                         ARRAY_SIZE(iovstack), &iov, &iter);
8      ...
9      // permission checks
10     if (!(file->f_mode & FMODE_WRITE))
11         return -EBADF;
12     if (!(file->f_mode & FMODE_CAN_WRITE))
13         return -EINVAL;
14     ...
15     // do file writev
16 }
```

Listing 2: The code snippet of vfs_writev function in kernel after 4.13.

In this work, DirtyCred utilizes these features to pause kernel execution after the file permission check is completed. In the following, we take userfaultfd as an example to describe how DirtyCred fulfills the kernel pause and extends the exploitation time window. For FUSE, the kernel pause procedure is similar. Readers could refer to the exploit sample we developed [2].

When performing file write, DirtyCred invokes syscall `writev`, the implementation of vectored I/O. Unlike syscall `write`, this syscall uses structure `iovec` to pass data from userspace to kernel space. List 1 in Line 1~5 defines the structure `iovec`. As we can observe, it contains a userspace address and a size field indicating the amount of data that will be transferred. In the Linux kernel space, in order to copy the data enclosed in `iovec`, the kernel needs to first import the `iovec` to the kernel space. Therefore, before Linux kernel version v4.13, as is shown in List 1, the implementation of `writev` first checks the file object, ensuring that the current file is in open status and with write permission. Once the check passes, it then imports the `iovec` from userspace and writes user data to the corresponding file. In this implementation, the import of `iovec` is in between the permission check and data write. DirtyCred can simply utilize the aforementioned userfaultfd feature to pause kernel execution right after completing the permission check and thus win sufficient time to swap the file object. To the best of our knowledge, this technique was firstly used by Jann Horn's exploitation for CVE-2016-4557 [25], but it is no longer available after kernel v4.13.

5.2 Alternative Exploitation of Userfaultfd & FUSE

After Linux kernel version v4.13, the kernel implementation gets changed. The import of `iovec` was moved ahead of permission check (see List 2). In this new implementation, DirtyCred could still use the userfaultfd feature to pause kernel execution at the site of `iovec` import. However, it no longer gives DirtyCred the ability to extend the time window between permission check and actual file write. To address this problem, DirtyCred exploits the design of the Linux filesystem.

In Linux, the filesystem design follows a strict hierarchy in which the high-level interface is common for file write operations, whereas the low-level interface varies across filesystems. When writing a file, the kernel first invokes the high-level interface. As is shown in List 3, `generic_perform_write` is a high-level interface for file write

```
1  ssize_t generic_perform_write(struct file *file,
2                  struct iov_iter *i, loff_t pos)
3  {
4      /*
5       * Bring in the user page that we will copy from _first_.
6       * Otherwise there's a nasty deadlock on copying from the
7       * same page as we're writing to, without it being marked
8       * up-to-date.
9       */
10     if (unlikely(iov_iter_fault_in_readable(i, bytes))) {
11         status = -EFAULT;
12         break;
13     }
14     ...
15     // call the write operation of the file system
16     status = a_ops->write_begin(file, mapping, pos, bytes, flags,
17                         &page, &fsdata);
18     ...
19 }
```

Listing 3: The code snippet of generic_perform_write function in the Linux kernel.

```
1  static ssize_t ext4_buffered_write_iter(struct kiocb *iocb,
2                      struct iov_iter *from)
3  {
4      ssize_t ret;
5      struct inode *inode = file_inode(iocb->ki_filp);
6      inode_lock(inode);
7      ...
8      ret = generic_perform_write(iocb->ki_filp, from,
       ↪  iocb->ki_pos);
9      ...
10     inode_unlock(inode);
11     return ret;
12 }
```

Listing 4: The code snippet of ext4 filesystem in the Linux kernel.

operation. As we can observe, in Line 15~17, `generic_perform_write` invokes the write operation of the filesystem and writes data to the file. To guarantee performance and compatibility, just before the write operation, the kernel triggers a page fault for the userspace data enclosed in `iovec`. As a result, using the userfaultfd feature in Line 10, DirtyCred could pause kernel execution prior to actual file writing and thus obtain a sufficient time window for privileged file object swap.

Compared with pausing kernel execution at the site of `iovec` import, we argue that exploiting the filesystem's design is more challenging to mitigate. First, as is described in the Linux code comment, removing page fault in `iovec` could potentially cause a deadlock issue (see List 3). Some filesystems will inevitably encounter trouble if the page is not pre-faulted. Second, while moving the page fault prior to the permission check could potentially resolve the problem, this straightforward defense reaction scarifies kernel performance and, more importantly, suffers from potential circumvention. For example, DirtyCred could remove the page right after triggering the first-page fault. In this way, the kernel inevitably triggers the page fault again and thus pauses the kernel execution right after the permission check.

5.3 Exploitation of Lock in Filesystem

In order to avoid messing up the content of a file, a filesystem does not allow two processes to write a file at the same time. In Linux, its filesystem enforces this practice by using a lock mechanism. To illustrate this, List 4 shows a simplified code snippet that performs a write operation in the ext4 filesystem. As we can observe, the filesystem first tries to obtain the inode lock in Line 6. If the inode is under the operation of another file (i.e., others hold the lock), the filesystem will wait until the lock is released. After obtaining the lock, the filesystem calls `generic_perform_write` to write the data to the file. When it completes the write, the filesystem will release the lock and return from the function.

The lock mechanism above could ensure the write operation does not go wrong. Unfortunately, it leaves an opportunity for DirtyCred to extend the time window and thus perform an object swap. To be specific, DirtyCred could spawn two processes – process A and process B – to write data on the same file simultaneously. Assume that process A holds the lock, writing a massive amount of data. When process A writes the file, process B would have to wait for an extended period until the lock is released in Line 10. Since prior to the invocation of `generic_perform_write`, process B has already completed the file permission check, the time spent on lock waiting provides DirtyCred with a sufficiently large time window to complete file object swap without worrying about the block of permission check. Based on our observation, the hold-on time could elapse to dozens of seconds when writing a 4GB file to a hard disk drive. Within this time window, triggering the vulnerability and performing memory manipulation could be completed without incurring any instability issue in exploitation.

6 ALLOCATING PRIVILEGED OBJECT

As is mentioned in Section 3.2, DirtyCred cannot passively wait for privileged users' activities and expect that these activities could result in a privileged object taking over the desired freed spot and thus fulfill privilege escalation. Therefore, DirtyCred has to take active action to trigger a privileged object allocation in the kernel space. This section discusses how DirtyCred – running as a low-privileged user – performs privileged object allocation.

6.1 Allocation from Userspace

In the Linux kernel, the "cred" objects represent the privilege level of corresponding kernel tasks. The root user has a privileged cred object, representing the highest privilege. Therefore, if DirtyCred can actively trigger a root user's activities, the kernel could allocate the privileged cred objects accordingly. In Linux, when a binary has SUID permission, it could be executed as if it is executed by the owner, regardless of whoever executes the binary. Using this characteristic, a low-privileged user could spawn a root process when he/she executes a root-user-owned binary with a SUID permission set.

In the past, attackers focused on exploiting a vulnerability in a privileged binary and thus fulfilled privilege escalation. In this work, DirtyCred does not rely on the vulnerabilities residing in privileged binaries. Instead, it abuses the aforementioned feature to spawn SUID-set binaries owned by root users, allocating the privileged cred object to occupy the free memory spot. In Linux, the binaries that match the feature are many, including the executables such as su, ping, sudo, mount, pkexec, etc.

As discussed earlier, in addition to cred objects, DirtyCred also could swap file objects for privilege escalation. Unlike cred objects, file object allocation is relatively easy. Recall that DirtyCred replaces a write-permitted file's object with a read-only file's object when swapping file objects. To allocate file objects specifying read-only permission, DirtyCred could open multiple target files with only the read permission. In this way, the kernel would allocate many corresponding file objects in the corresponding kernel memory.

6.2 Allocation from Kernel Space

The method described above indicates a way that allocates privileged objects from userspace. In fact, DirtyCred can also perform privileged object allocation from kernel space. When the Linux kernel starts a new kernel thread, it duplicates its current running process. Together with the process duplication, the kernel allocates a copied cred object on the kernel heap accordingly. In the Linux kernel, most of the kernel threads have a privileged cred object. As a result, the copied cred object is also in high privilege. Using the ability to spawn privileged kernel threads, DirtyCred could actively allocate privileged cred objects.

To the best of our knowledge, there are two major approaches to allocating high privileged credential objects. The first is to interact with the kernel code snippets, triggering the kernel to spawn a privileged thread internally. For example, creating workers for kernel workqueue can also be used to spawn kernel threads. In the Linux kernel, the work queue is designed for handling deferred functions. A work queue comes with a number of work pools. Each work pool contains workers. A worker is the underlying execution unit that runs the work committed to the workqueue. The number of the workers in each work pool is, at most, the number of the CPUs. Initially, the kernel creates only one worker for each work pool. When there is a need for more workers or, in other words, more works are committed to the work queue, the kernel will create workers dynamically. Each worker is a kernel thread. As a result, by adjusting the works committed to the kernel work queue, one could control the activities of kernel thread spawning accordingly.

In addition to the method above, the second approach to spawning kernel threads is to invoke the usermode helper. Usermode helper is a mechanism that allows the kernel to create a user-mode process. One of the most straightforward applications of the usermode helper is loading kernel modules to kernel space. When loading a kernel module, the kernel calls the usermode-helper API, which further executes the userspace program – modprobe in high privileged mode and thus creates a high privileged credential objects in the kernel. Part of modprobe functionality is to search through the standard installed module directories to find the necessary drivers. During the search, the kernel needs to continue its execution. As a result, to avoid modprobe blocking kernel execution, when invoking a usermode-helper API, the kernel also spawns a new kernel thread.

7 EVALUATION

In this section, we design two experiments to evaluate the exploitability of DirtyCred on real-world kernel vulnerabilities.

Memory Cache	Structure	Offset
kmalloc-16	vdpa_map_file	0 ★
kmalloc-32	binder_task_work_cb	16 ★
	binder_txn_fd_fixup	16 ★
	coda_file_info	8 ★
	shm_file_data	16 ★
kmalloc-64	fuse_fs_context	8 ★
	ovl_dir_file	24★ 32★
	bpf_event_entry	8 ★
kmalloc-96	gntdev_dmabuf_priv	80 ★
	nfs_access_entry	40 †
	request_key_auth	32 †
	watch	64 †
	bpf_perf_link	64 ★
kmalloc-128	async	32 †
	nfs_delegation	16 †
kmalloc-192	fs_context	88 †
	sync_file	0 ★
	vmci_ctx	144 †
	coda_vm_ops	8 ★
	nfs_open_context	80 †
	nfs_unlinkdata	144 †
	nfs_renamedata	152 †
	nfs4_layoutreturn	144 †
	ovl_fs	112 †
kmalloc-256	usb_dev_state	152 †
	autofs_sb_info	8 ★
	shmid_kernel	128 ★
	bsd_acct_struct	144 ★
kmalloc-512	linux_binprm	48 ★, 64 ★
	loop_device	96 ★
	dma_buf	8 ★
	nvmet_ns	24 ★
	ksmbd_file	0 ★
	rpc_clnt	440 ★
	nfs4_state_owner	56 †
	nfs4_ff_layout_mirror	96 †, 104 †
	p9_trans_fd	0 ★, 8 ★
kmalloc-1k	sock	600 †
	binder_proc	80 †
	kfd_process_device	256 ★
	send_ctx	0 ★
	nlm_host	520 †
	nfs4_layoutcommit_data	472 †
kmalloc-2k	vsock_sock	864 †
	io_ring_ctx	408 †
kmalloc-4k	vduse_iova_domain	3824 ★
vm_area_cachep	vm_area_struct	160 ★
ashmem_area_cache	ashmem_area	288 ★
client_slab	nfs4_client	736 †
nfsd_file_slab	nfsd_file	48 ★, 56†
kioctx_cachep	kioctx	512 ★

Table 1: Exploitable objects identified in the Linux kernel. Note that the symbol ★ indicates an object tied to "file" credential whereas the symbol † represents an object associated with "cred" object. The column "Memory Cache" specifies the caches storing kernel objects. The column "Structure" represents the exploitable objects' types. The column "Offset" describes where the credential object's reference is located in the exploitable object. Note that some exploitable objects contain two credential-object references (e.g., `ovl_dir_file` and `linux_binprm`).

CVE-ID	Observed Capability	DirtyCred
CVE-2022-27666	OOB	✔
CVE-2022-25636	Double Free ⋆	✔
CVE-2022-24122	UAF	✗
CVE-2022-0995	OOB	✔
CVE-2022-0185	OOB	✔
CVE-2021-22600	Double Free	✔
CVE-2021-4154	UAF	✔
CVE-2021-43267	OOB	✔
CVE-2021-41073	Double Free ⋆	✔
CVE-2021-34866	OOB †	✗
CVE-2021-33909	OOB †	✗
CVE-2021-42008	OOB	✔
CVE-2021-3492	Double Free	✔
CVE-2021-27365	OOB	✔
CVE-2021-26708	Double Free ⋆	✔
CVE-2021-23134	Double Free ⋆	✔
CVE-2021-22555	Double Free	✔
CVE-2021-3490	OOB †	✗
CVE-2020-14386	OOB	✔
CVE-2020-16119	Double Free ⋆	✔
CVE-2020-27194	OOB †	✗
CVE-2020-8835	OOB †	✗
CVE-2019-2215	UAF	✗
CVE-2019-1566	UAF	✗

Table 2: Exploitability demonstrated on real-world vulnerabilities. Note that some CVEs provide both use-after-free and double-free capabilities. Here, we categorize such vulnerabilities into double-free and mark them with a ⋆ symbol. Note that the symbol † indicates the vulnerabilities that could corrupt only data in virtual memory area.

7.1 Experiment Design & Setup

As is mentioned above, DirtyCred utilizes exploitable objects (i.e., the ones that enclose credential objects) to perform memory manipulation particularly for vulnerabilities like out-of-bound access and use-after-free. This manipulation is one of the critical steps for DirtyCred to fulfill privilege escalation. When performing memory manipulation, DirtyCred allocates the exploitable object to the cache where the vulnerability locates. For different vulnerabilities, they demonstrate memory corruption capability on different caches. Therefore, the success of DirtyCred highly depends on whether it could successfully identify exploitable objects that could fit into the corresponding cache. With this in mind, we first identify the unique exploitable objects available for each cache.

In order to point out the objects, one instinct reaction is to seek through the Linux kernel code manually, pinpoint those exploitable objects, and figure out the input that could trigger the corresponding allocation. However, Linux kernel code space is large and sophisticated, making code examination impractical. Therefore, to address this problem, we introduce an automated method to track down exploitable objects and the corresponding input to trigger their allocation. In our evaluation, we apply the automated method to the latest stable kernel (i.e., version 5.16.15 at the time of this paper writing). We consider an object as an exploitable object only if the automated method can find an object that encloses the credential object and can demonstrate an input to allocate that object on the kernel heap. Due to the space limit, we detail the design and implementation of our automated method in the Appendix A.

In addition to identifying exploitable kernel objects, our experiment also explores DirtyCred's exploitability against real-world vulnerabilities. Recall that DirtyCred needs to pivot a vulnerability capability if the vulnerability does not provide DirtyCred with the ability to swap credential objects directly. As is discussed in Section 4.1, when performing vulnerability pivoting, DirtyCred might need to overwrite some critical data in the exploitable object. For different vulnerabilities, their overwriting capability could vary significantly, further impacting the success of privilege escalation. As a result, we evaluate DirtyCred's effectiveness by using it to exploit many real-world vulnerabilities and studying how well it could perform exploitation against these vulnerabilities.

We assume a Linux kernel is armed with state-of-the-art exploit mitigation techniques available in kernel when performing the exploitation. As a result, we need to select vulnerabilities identified in the kernel developed in recent years. In our evaluation, we selected only Linux kernel CVEs reported after 2019. In our CVE selection process, we filtered out those vulnerabilities that do not corrupt data on the kernel heap. Besides, we ruled out those vulnerabilities for which we cannot reproduce the corresponding kernel panic. Last but not least, we also eliminated those vulnerabilities, the trigger of which requires the installation of specific hardware. Following these CVE selection criteria, we obtained a data set with 24 unique CVEs. In Table 2, we listed these CVEs' IDs and the corresponding vulnerability types. As we can observe, our selected test cases cover nearly all types of vulnerabilities on the kernel heap.

7.2 Experiment Result

Exploitable objects. Table 1 shows the exploitable objects identified in each kernel cache. As we can observe, the exploitable objects cover nearly all the general caches except for kmalloc-8, which is rarely used in the Linux kernel. For most of memory caches, there are more than one exploitable object potentially useful for DirtyCred's privilege escalation. In each exploitable object, the offset of the field referencing a credential object is also present in Table 1. As we can observe, the offsets for different exploitable objects vary. It indicates that DirtyCred has a higher chance of finding a suitable object to match a vulnerability's capability and perform successful exploitation. For example, if a vulnerability demonstrates the capability of overwriting 8 bytes to an adjacent object at its offset of the 8-th byte, an exploitable object with the critical data at the 8-th byte would greatly facilitate DirtyCred's privilege escalation.

From Table 1, we also discover 5 objects in 5 general caches. They enclose the reference to the credential object at the beginning of the objects. It implies that even if the attackers only obtain a very limited memory corruption capability (e.g., overwriting two bytes of zero at the beginning of a victim object), they are still capable of leveraging the identified exploitable object to launch a DirtyCred attack. It should be noted that Table 1 also distinguishes exploitable objects referencing cred and file using different symbols. As we will discuss in Section 10, a cred object could provide better support for

container escape. Therefore, adequate exploitable objects with cred object linkage indicate more substantial support in docker escape.

Exploitability. Table 2 shows the exploitability of DirtyCred across different vulnerabilities. As we can observe, DirtyCred successfully demonstrates kernel defense bypassing and privilege escalation on 16 out of 24 vulnerabilities when the underlying Linux kernel enables all the exploit mitigation mechanisms discussed in Section 2.3. This observation implies that DirtyCred could be used as a powerful, general exploitation method for kernel vulnerability exploitation tasks. Of the 16 exploitation-successful test cases, 8 are out-of-bound or use-after-free vulnerabilities, and the other 8 are double-free. DirtyCred succeeds on all double-free test cases as a double-free capability could always be pivoted to freeing a credential object invalidly.

The failure cases are primarily from out-of-bound and use-after-free. For OOB vulnerabilities, the failure cases demonstrated memory corruption in the virtual memory area. To use DirtyCred, we need to find kernel objects with credential information. These objects are usually allocated at the kmalloc'ed memory region but not virtual memory. As a result, DirtyCred fails to find necessary objects for a successful exploitation. We annotate these cases with a † symbol in Table 2. As we will discuss in Section 10, the failure of exploiting those cases does not mean DirtyCred cannot exploit vulnerabilities on virtual memory. The memory corruption capabilities on virtual memory could still be pivoted to capabilities useful for DirtyCred if there are suitable exploitable objects or using other capability pivoting techniques.

For the UAF failure case CVE-2022-24122, it does not demonstrate the overwriting capability through the dangling pointer but manifests merely an over-reading ability. As is discussed in Section 4, DirtyCred relies on either invalid write capability or invalid free capability. The over-reading capability of CVE-2022-24122 limits DirtyCred to perform a successful capability pivoting, thus fails the attack. For CVE-2019-2215 and CVE-2019-1566, they manifest the overwriting capability. However, the overwriting ability does not happen in the critical field of exploitable objects. Without such a capability, DirtyCred cannot manipulate necessary fields in the kernel objects to free a credential object, thus fails the attack.

8 DEFENSE AGAINST DIRTYCRED

Given the exploitability demonstrated in the section above, we argue that DirtyCred is a severe threat to the existing Linux system. While the technique of abusing lock mechanism could be mitigated by reengineering the filesystem, it is still not enough to block DirtyCred as it could be launched from another path – swapping cred object. Therefore, an effective approach is to prevent the swap of credentials with different privilege level. From one perspective, userspace heap defenses are not adequate for DirtyCred. The kernel wants the memory allocation/free/access to be as fast as possible. Otherwise, it will slow down user space programs and the entire system. Therefore, the memory allocator in the kernel is much simpler than that in userspace (e.g., ptmalloc). This fact makes userspace heap defenses not applicable to kernel space. From another viewpoint, even if the Linux kernel has introduced many defense mechanisms (e.g., CFI, SMEP, SMAP, and KASLR, etc.), none of the existing kernel defenses are effective to DirtyCred for the following reasons.

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%

Table 3: The performance evaluation results of the proposed defense on two different benchmarks – Phoronix and LMBench.

First, DirtyCred does not violate any control-flow integrity, making the effort of safeguarding kernel control flow futile. Second, DirtyCred does not rely on a single exploitation component for exploitation. As is shown in Section 7, valuable objects for exploitation are spread across nearly all general caches. Therefore, defending against DirtyCred by eliminating exploitable objects is nearly infeasible. Third, DirtyCred fulfills its exploitation goal by placing a legit credential object to an illegitimate memory spot but not tampering with a credential object's content. This exploitation practice makes existing credential integrity protection techniques (e.g., Samsung Knox's Real-time Kernel Protection [43]) less likely to be effective. Last but not least, DirtyCred performs privilege escalation by swapping high and low privileged credential objects between each other. This exploitation method fails many kernel object isolation schemes (e.g., AUTOSLAB [34] and xMP [44]) because they separate critical kernel objects in their own memory regions based on the type of the objects but not their privilege.

To this end, we argue that one effective defense solution against DirtyCred would be to isolate high privileged and low privileged objects, forcing them not to share the same memory space. In this way, DirtyCred will no longer be able to overlap objects with different privileges for privilege escalation. To achieve the goal above, a straightforward reaction is to create two different caches. One is

used for high privileged object storage. The other is used to hold low privileged objects. Since caches are naturally isolated, this design could ensure that objects with different privileges have no overlap. However, as we discussed in Section 4.2, once a memory cache is destroyed, the Linux's buddy allocator recycles the underlying memory page. Therefore, DirtyCred could still launch its attack by abusing this memory-page recycle feature.

Design. With the analysis above in mind, we propose a practical defense solution that creates a cache for high privileged objects in the virtual memory region and leaves the low privileged objects in the normal memory area (i.e. direct-mapped memory region). The virtual memory region refers to dynamic allocations of virtually contiguous memory within the kernel. It resides in the memory area defined by `VMALLOC_START` through to `VMALLOC_END`. Since it is separated from the direct-mapped memory region, the regions designated to high privileged and low privileged objects would never overlap even after the caches are destroyed and the underlying memory pages are recycled.

Implementation. In this work, we implemented our proposed defense against DirtyCred on the Linux kernel v5.16.15. In our implementation, we manually modified the ways of allocating `cred` objects and `file` object in kernel. If the allocation is for privileged ones, we allocate them using virtual memory. To be specific, when allocating `cred` objects, we examine the privilege based on the `UID` of the object. If the `UID` matches `GLOBAL_ROOT_UID`, which means the allocation is for privileged cred objects, we use `vmalloc` as the allocator to allocate virtual memory for the object. For `file` objects, we examines the file's mode. If the file is opened with write permission, we will allocate the `file` object with `vmalloc` accordingly. Our implementation is available at [2].

Technical discussion. Our proposed defense protects the Linux kernel by enforcing memory isolation for credential objects. As is mentioned above, our implementation determines the privilege at the time of object allocation. However, the privilege could be altered by changing the `UID` during runtime (e.g., changing a low privileged credential object to a high privileged one through 'setuid' syscall). When this occurs, our proposed defense above will encounter security issues because we perform object isolation only at the time of allocation. To address this problem, we manually modify the way of altering kernel credential objects in our implementation. Specifically, if the kernel changes `UID` of a credential object to `GLOBAL_ROOT_UID`, we will copy the high privileged credential object to the 'vmalloc' region rather than altering the original one. However, we think some issues might be raised if the future kernel development does not follow the same pattern. As a result, we leave the exploration of alternative solutions as part of our future work.

Performance Evaluation. To evaluate our defense mechanism's performance, we ran two benchmarks against the vanilla Linux kernel and our defense-enabled kernel on a bare-mental machine (with an Intel 4-Core CPU, 16GB RAM, and 1000GB HHD). Our benchmarks include a micro-benchmark from LMbench v3.0 [41] and a macro-benchmark from Phoronix Test Suite [42]. The LMbench evaluates the latency and bandwidth of syscalls and system I/O, whereas the Phoronix Test Suite examines the performance of real-world applications on two Linux kernels. For LMbench, We ran the benchmark 10 times to avoid randomness and took the average as the observed performance. For Phoronix Test Suite, we ran the test with batch mode, which will run the test 50 times and output the average values.

Table 3 shows our evaluation results. First, we can observe that our proposed method mostly introduces negligible performance overhead, indicating that our defense is lightweight. Second, we can observe that there is some moderate performance decrease for the test cases – "10k File Create" and "10k File Delete" – in the LMBench. As is shown in Table 3, our proposed defense introduces an overhead of over 4%. The reason behind this performance decrease is that file objects were allocated to virtual memory region through `vmalloc` rather than the normal memory region through `kmalloc`. In comparison with `kmalloc`, `vmalloc` is relatively slow because virtual memory have to re-map the buffer space into a virtually contiguous range, whereas 'kmalloc' never re-maps.

It should be noted that the file deletion involves a lower performance downgrade than file creation (4.25% vs. 7.17%). The reason behind the difference is that the free of file object is done through RCU, which is executed asynchronously to the file deletion process. While the moderate overhead may raise the concern of some production systems, it dramatically improves the kernel protection against DirtyCred. In this work, our primary objective is to raise the Linux community's awareness but not to build a secure, efficient defense solution. We leave the exploration of the alternative defense solution as our future research effort. Last, it should also be noted that some cases demonstrated slight performance improvement after we introduced defense to the Linux kernel. This is mainly due to the noise of our experiment although we tried to minimize the noise as much as we could by running the benchmark multiple times, and disabling CPU boost on a bare-metal machine.

9 RELATED WORK

This work introduces a new kernel exploitation method and the corresponding defense to mitigate the threat. As a result, the works most relevant to ours include kernel exploitation and kernel exploitation mitigation. In the following, we summarize the works on these two topics and discuss how they differ from our proposed techniques.

Kernel exploitation. The kernel exploitation techniques evolved with the development of kernel defense. Prior to the introduction of Supervisor Mode Execution Prevention (SMEP) [29], the technique – ret2usr [32] – exploits the Linux kernel by pivoting kernel execution to the userland. After the broad deployment of SMEP in Linux, this technique no longer works because SMEP prevents kernel execution in the userspace. Following SMEP, Supervisor Mode Access Prevention (SMAP) [7] was further proposed to block direct userspace access, which further enhances the separation of the kernel and userspace's access. To bypass the protection enforced by SMEP/SMAP, researchers proposed a series of new exploitation methods. For example, Kemerlis et al. proposed the ret2dir technique [31] that enables an adversary to mirror userspace data within the kernel address space. Wu et al. introduce KEPLER [48] that utilizes a particular kernel code gadget to transform the PC control to stack overflow and thus enables a long ROP chain.

To prevent ROP attacks against the Linux kernel, Linux introduces KASLR, which increases exploitation difficulty by randomizing the kernel memory address layout. However, following the adoption of this kernel defense, security experts then proposed many practical methods [5, 16, 23, 28] to circumvent KASLR. For example, using elastic objects in the kernel. Chen et al. proposed ELOISE [5] that could disclose sensitive kernel information after overwriting the length field of elastic objects. Gruss et al. proposed a hardware side-channel attack that leverages pre-fetch instructions to bypass KASLR. Recently, some security experts even proposed to utilize vulnerabilities in processors to launch Meltdown [39] and Spectre [33] attacks and thus bypass KASLR's protection on Linux.

In addition to memory address space randomization, security researchers also proposed techniques to randomize the heap memory layout in the Linux kernel [17, 47]. With the heap randomization enabled, the heap layout is no longer linear, which increases the difficulty for adversaries in performing heap layout manipulation. However, following the success of heap layout randomization, Xu et al. proposed a memory collision technique [49]. This technique uses the memory reuse mechanism to exploit kernel use-after-free vulnerabilities without being hindered by heap randomization.

Unlike the above exploitation techniques, aiming to the circumvention of some specific kernel protection and exploit mitigation but not an ultimate exploitation goal (e.g., privilege escalation), our work focuses on end-to-end exploitation without the headache of bypassing broadly deployed kernel defenses. As such, our exploitation method is more general and consequential. As we will discuss in Section 10, DirtyCred can even facilitate the ability to escape containers and root Android devices.

Kernel defense. In addition to the kernel defenses introduced together with the existing exploitation methods above, there are also many other kernel protection and exploit mitigation mechanisms. These defenses are proposed by academia and industry, receiving significant attention from the security community. Here, we briefly introduced some recently proposed or mostly adopted in practice.

To thwart side-channel attacks against the Linux kernel, Gruss et al. proposed a strict kernel and userspace isolation mechanism – KAISER [22]. This mechanism could ensure that the hardware does not hold any information about kernel addresses while running in user mode. To improve KASLR, Linux kernel developers introduce Function Granular Kernel Address Space Layout Randomization (FGKASLR [1]). By randomizing the layout down to a code function level, FGKASLR makes the code-reuse attack more challenging. To hinder control-flow hijacking, researchers also proposed a variety of defense mechanisms to enforce control-flow integrity in the Linux kernel [11, 15, 18, 50]. For example, Yoo *et al.* proposed to implement an in-kernel, control-flow integrity protection by using ARM's Pointer Authentication [50].

In addition to kernel control-flow protection above, there are also a series of defense techniques that focus on providing protection for critical kernel data [4, 13, 27, 40, 46]. For example, AUTOSLAB [34] and xMP [44] are such kernel defenses. As we discussed in Section 8, AUTOSLAB isolates different types of objects into different memory caches, which reduces the objects useful for kernel heap memory manipulation. xMP employs virtualization techniques to isolate sensitive data and thus prevents them from being tampered with by unlawful actors.

From the defense philosophy viewpoint, our defense mechanism is different from the works safeguarding kernel control-flow integrity. It is similar to the works that isolate critical kernel data. However, our defense is entirely different from these works from the technical perspective. Rather than isolating objects based on the types and sensitivity, our defense performs memory isolation based on the privilege of kernel objects. As such, it is more effective in defending against the threat of DirtyCred.

10 DISCUSSION & FUTURE WORK

In this section, we discuss some other issues we have not yet discuss and present our future effort.

Escaping container. Going beyond privilege escalation on Linux, DirtyCred can facilitate container escape passively and actively. As mentioned earlier, DirtyCred performs exploitation by swapping either file objects or cred objects. Using file objects for exploitation, DirtyCred could overwrite a high privileged file. However, no file in a container provides the privilege to switch the namespace. In order to address this problem, a recent work [12] shows that an attacker could passively wait for the runC process and thus execute root commands on the host by overwriting the process. Motivated by this idea, DirtyCred could use the file object swap mechanism to overwrite the runC process and thus fulfill container escape.

Unlike the method above, using cred objects to perform container escape does not need passive wait. To do so, DirtyCred could first trigger a Linux kernel vulnerability, swap cred objects, and thus escalate the attacker's privilege to `SYS_ADMIN`. With this `SYS_ADMIN` ↪ privilege in hand, the attacker could then follow a previously proposed docker escape method [3] that mounts a cgroup and then utilizes the `notify_no_release` mechanism to execute the root command on the host system. To demonstrate DirtyCred's ability to escape the container, we provide a working exploit at [2]. Reviewers could download the exploit and see more docker escape details.

Rooting Android. In addition to container escape, DirtyCred is also capable of rooting Android. The Android kernel is developed based on the generic Linux kernel. In practice, the Android kernel is more difficult to exploit compared to the generic kernel because of the more strict access control and newly shipped defenses [19]. DirtyCred can root Android with two paths of attacks discussed in this paper. On the one hand, DirtyCred could swap the task credentials directly, which grants attackers privileged task credentials, thus root privilege. On the other hand, DirtyCred could first utilize its file manipulation capability to overwrite shared system library, which allows a privilege escalation from the restrict sandbox. Then, it could overwrite kernel modules with malicious code, fulfilling arbitrary read and write, and eventually disable SELinux on Android. We demonstrated DirtyCred's capability of rooting Android with zero-day vulnerabilities. By the time of writing this paper, we reported the vulnerabilities to Google and received their acknowledgement.

Cross version / architecture exploitation. When crafting an exploit under the guidance of DirtyCred, one could expect the same exploit code could work across different kernel versions or architectures without any modification for the following reasons. First, unlike other exploitation methods that need the leakage of kernel

base address to bypass KASLR, DirtyCred does not need to handle KASLR. As a result, the exploitation code does not include any data specific for kernel versions or underlying architectures. Second, many previous kernel exploitation methods (e.g., KEPLER [48]) heavily rely on ROP to escalate privilege. When migrating such exploits to a different architecture, one needs to modify the ROP chain and thus preserve its exploitability. DirtyCred does not use any architecture-specific data as discussed throughout the paper. Therefore, once a piece of exploit code is developed for a vulnerability, the exploit could work on other vulnerable kernels, regardless of their versions and the underlying architectures.

Other ways to pivot capability. In Section 4, we proposed some techniques to pivot a memory corruption capability to the capability useful for DirtyCred. In our evaluation, we found vulnerabilities happened on virtual memory are harder to exploit with DirtyCred. The reason is that there are less exploitable objects on virtual memory, which limits pivoting capability from the original memory corruption to ones that are useful for DirtyCred. We argue that this does not mean vulnerabilities on virtual memory cannot be exploited with DirtyCred. For example, CVE-2021-34866 provides an out-of-bound capability that demonstrates an overwrite on vmalloc'ed memory. Using our pivoting approach, we cannot pivot this capability to deallocate a credential object. However, a recent writeup [24] shows a sophisticated method that could convert this overwrite capability on vmalloc into an arbitrary read and write and thus enable a double-free capability. As is discussed and shown in Section 4.2, using a double-free capability, DirtyCred has a high chance of fulfilling its privilege escalation. In addition to pivoting vulnerability capability, a recent work [35] demonstrates an approach to exploring the vulnerability's different capability. We argue that these methods are complementary to the DirtyCred attack, as is shown in Section 3.1, DirtyCred could still be launched without pivoting capability. In this work, we leave the exploration of other pivoting methods as part of our future research.

Stability. DirtyCred's exploitation stability could be impacted by two critical factors like all kernel exploitation methods. First, when pivoting vulnerability capability, DirtyCred has to manipulate the memory layout, taking over the target memory spot. At this moment, the exploitation stability might vary if the memory layout manipulation is affected by system activities. Second, when exploiting a kernel vulnerability, the way to trigger the vulnerability could also greatly influence the stability of the exploitation. To improve exploitation stability, a recent work [51] proposes a series of methods to stabilize kernel exploitation. In this work, our goal is to assess the exploitability of DirtyCred in a real-world setting. We conclude DirtyCred could exploit a vulnerability successfully as long as it demonstrates exploitability. In the future, we will explore how to utilize existing exploitation stabilization techniques to improve DirtyCred's exploitation success rate.

TOCTOU. As discussed above, DirtyCred swaps credential objects in a critical time window. Intuition suggests that the existing TOCTOU defense mechanisms may hinder our proposed exploitation method. According to a recent research article [45], TOCTOU defense could be divided into source code detection, postmortem detection, system call interposition, intra/inter-process memory consistency, transactional system calls, and sandbox filesystem. Source code detection analyzes the source code of the target program. It is obvious that this kind of defense method could not be applied to defending against our exploitation because there is no identified source code pattern in DirtyCred. Postmortem detection detects TOCTOU vulnerabilities after the attack is actually carried out. This kind of method needs happens-before analysis, which does not affect our exploitation. The reason is that we use unexpected free operations, which could not be seen in the analysis process. System call interposition monitors the system call sequence and thus determines attacks. Our exploitation method does not use a malicious system call sequence. As a result, system call interposition would not block us either. Intra/inter-process memory consistency protects the shared variables in multiple threads by recording the operations on the variables. Our exploitation applies unexpected operations which could not be recorded. Transactional system calls and sandbox filesystem target the race condition between file read and write. Our method does not require that condition.

11 CONCLUSION

The Linux kernel has been armed with various protection and exploitation mitigation schemes, making kernel exploitation challenging to succeed. To bypass kernel defenses, a bad actor has to leverage a vulnerability with a solid capability to disable protection and mitigation. In this work, we show a new exploitation method – DirtyCred. It can enable exploitation and defense circumvention without a vital requirement for kernel vulnerability. Using DirtyCred, we demonstrate that a bad actor could employ nearly arbitrary heap-based kernel vulnerability to swap credential objects. The credential swap could confuse the Linux kernel into believing a high privileged file or task is in a low privileged mode. Thus, an unprivileged user could escalate his or her privilege for malicious purposes. With this discovery, we conclude that DirtyCred sabotages the existing Linux defense architecture. If our security community does not take immediate action in response to this new exploitation method, the Linux-driven systems will soon be in danger. Their users will suffer from significant personal loss. Following this conclusion, this work also proposed a privilege-object isolation mechanism as a defense suggestion. In this work, we implemented this defense mechanism as a Linux kernel prototype and found it could secure Linux at a negligible and sometimes moderate cost. With this new discovery, we further conclude that isolating memory based on object privilege might be necessary for defending against the threat of DirtyCred. We have reported part of our research findings to software and hardware vendors that could potentially be affected by DirtyCred and are actively working with them to help them understand and mitigate the threat.

12 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful feedback. This work was supported by ONR N00014-20-1-2008 and NSF 1954466. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] Kristen Carlson Accardi. 2020. Function Granular KASLR. (2020). https://lwn.net/Articles/824307/

[2] Anonymous. 2022. DirtyCred Exploit. (2022). https://hackmd.io/giRE2P2oQHektZzOG053IQ

[3] Alex Chapman. 2020. Privileged Container Escape Control Groups release_agent. (2020). https://ajxchapman.github.io/containers/2020/11/19/privileged-container-escape.html

[4] Quan Chen, Ahmed M Azab, Guruprasad Ganesh, and Peng Ning. 2017. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*.

[5] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. 2020. A systematic study of elastic objects in Kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.

[6] Yueqi Chen and Xinyu Xing. 2019. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[7] Jonathan Corbet. 2012. Supervisor mode access prevention. (2012). https://lwn.net/Articles/517475/

[8] Jonathan Corbet. 2017. The current state of kernel page-table isolation. (2017). https://lwn.net/Articles/741878/

[9] The MITRE Corporation. 2021. CVE-2021-4154. (2021). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4154

[10] The MITRE Corporation. 2022. CVE-2022-0847. (2022). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0847

[11] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*.

[12] Datadog. 2022. Using the Dirty Pipe Vulnerability to Break Out from Containers. (2022). https://www.datadoghq.com/blog/engineering/dirty-pipe-container-escape-poc/

[13] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables.. In *Proceedings of the 2017 Network and Distributed Systems Security Symposium*.

[14] Jake Edge. 2013. Kernel address space layout randomization. (2013). https://lwn.net/Articles/569635/

[15] Jake Edge. 2020. Control-flow integrity for the kernel. (2020). https://lwn.net/Articles/569635/

[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*.

[17] Thomas Garnier. 2016. mm: SLAB freelist randomization. (2016). https://lwn.net/Articles/685047/

[18] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy*.

[19] google. 2022. Kernel Control Flow Integrity. (2022). https://source.android.com/devices/tech/debug/kcfi

[20] Google. 2022. Roses are red, Violets are blue, Giving leets more sweets. All of 2022! (2022). https://security.googleblog.com/2022/02/roses-are-red-violets-are-blue-giving.html

[21] Google. 2022. syzkaller kernel fuzzer. (2022). https://github.com/google/syzkaller

[22] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*. 161–176.

[23] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*.

[24] HexRabbit. 2021. CVE-2021-34866 Writeup. (2021). https://github.com/HexRabbit/CVE-writeup/tree/master/CVE-2021-34866

[25] Jann Horn. 2022. Linux: UAF via double-fdput. (2022). https://bugs.chromium.org/p/project-zero/issues/detail?id=808

[26] David Howells. 2022. CREDENTIALS IN LINUX. (2022). https://www.kernel.org/doc/Documentation/security/credentials.txt

[27] Kaiming Huang, Yongzhe Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2022. The Taming of the Stack: Isolating Stack Data from Memory Errors. In *Proceedings of the 2022 Network and Distributed Systems Security Symposium*.

[28] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*.

[29] Mateusz Jurczyk. 2011. SMEP: What is it, and how to beat it on Windows. (2011). https://j00ru.vexillium.org/2011/06/smep-what-is-it-and-how-to-beat-it-on-windows/

[30] Max Kellermann. 2022. The Dirty Pipe Vulnerability. (2022). https://dirtypipe.cm4all.com/

[31] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*.

[32] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. 2012. {kGuard}: Lightweight Kernel Protection against {Return-to-User} Attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium*.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*.

[34] Zhenpeng Lin. 2021. How AUTOSLAB Changes the Memory Unsafety Game. (2021). https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game/

[35] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Chensheng Yu, Dongliang Mu, Xinyu Xing, and Kang Li. 2022. GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy*.

[36] Linux. 2022. File management in the Linux kernel. (2022). https://www.kernel.org/doc/Documentation/security/credentials.txt

[37] Linux. 2022. FUSE's introduction in the Linux kernel user's and administrator's guide. (2022). https://www.kernel.org/doc/html/latest/filesystems/fuse.html

[38] Linux. 2022. Userfaultfd's introduction in the Linux kernel user's and administrator's guide. (2022). https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium*.

[40] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing Kernel Hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium*.

[41] Larry McVoy and Carl Staelin. 2022. LMbench - Tools for Performance Analysis. (2022). http://lmbench.sourceforge.net/

[42] Phoronix Media. 2022. Open-Source, Automated Benchmarking. (2022). https://www.phoronix-test-suite.com/

[43] Samsung Knox News. 2016. Real-time Kernel Protection (RKP). (2016). https://www.samsungknox.com/en/blog/real-time-kernel-protection-rkp

[44] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. 2020. xmp: Selective memory protection for kernel and user space. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*.

[45] Razvan Raducu, Ricardo J. Rodríguez, and Pedro Álvarez. 2022. Defense and Attack Techniques Against File-Based TOCTOU Vulnerabilities: A Systematic Review. *IEEE Access* 10 (2022), 21742–21758.

[46] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity.. In *Proceedings 2016 Network and Distributed System Security Symposium*.

[47] Dan Williams. 2018. Randomize free memory. (2018). https://lwn.net/Articles/767614/

[48] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Conference on Security Symposium*.

[49] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[50] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2021. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. *arXiv preprint arXiv:2112.07213* (2021).

[51] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. 2022. Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability. In *Proceedings of the 31st USENIX Conference on Security Symposium*.

A APPENDIX

As is mentioned in Section 7.1, we designed and implemented an automated tool to facilitate the identification of those exploitable objects (*i.e.,* the objects enclosing a reference to credential objects). Here, we discuss how we design this automated tool step-by-step and then describe our implementation in detail.

```
struct key *request_key_auth_new(...)
{
    struct request_key_auth *rka;

    ...

    /* allocate a auth record */
    rka = kzalloc(sizeof(*rka), GFP_KERNEL);

    ...
    rka->cred = get_cred(irka->cred);
    ...
}

static void free_request_key_auth(struct request_key_auth *rka)
{
    ...
    if (rka->cred)
        put_cred(rka->cred);
    ...
    kfree(rka);
}
```

Listing 5: The allocation and deallocation sites for the object in the type of "struct request_key_auth".

A.1 Design

Step 1: analyzing structure definition. Recall that an exploitable object should include a pointer referencing a credential object. Therefore, the first step in identifying an exploitable object is to analyze the definition of kernel data structures. This analysis could bound our consecutive analysis in a scope, avoiding unnecessary analysis of irrelevant data structures in the following steps.

Given a data structure, we go through each field based on the definition. If a field is a nested structure or union type, we also extract its fields and examine them accordingly. In this work, we follow this procedure recursively until all the structure fields are thoroughly analyzed. Along with this analysis, we also examine the type information of pointers. If a field pointer references a credential structure type (e.g., file or cred), we record the field's offset in the enclosed structure and mark the structure type as a candidate.

Step 2: identifying allocation sites. As is described in the main text, DirtyCred performs privilege escalation by exploiting heap-based memory corruption vulnerabilities. Therefore, we need to ensure that the objects in the identified structure type can be allocated on the kernel heap. To do it, we first pinpoint each code site that allocates an object on the kernel heap (see the object allocation example code snippet in List 5). Second, we examine the return value of the heap (de)allocation function. Following the data flow of the allocated object, we extract the object's type information and check if it matches the structure candidates identified in the first step. We record the allocation site for the corresponding object if a match exists.

Step 3: pinpointing free sites. Recall that DirtyCred needs to deallocate credential objects. As a result, we also need to guarantee that when deallocating an exploitable object candidate, the kernel could also free the corresponding credential object along the way. In the Linux kernel, credential objects have their unique properties. Their deallocation is carried out through dedicated, standard kernel API functions [26, 36]. In this work, we summarize these deallocation functions manually and then use these functions as the starting point to perform our analysis. Specifically, for each code site where the credential-object deallocation function is invoked, we taint corresponding arguments and then perform a backward data-flow analysis. Our backward analysis terminates until it identifies a site where the credential object is initialized. We examine the initialization site and extract the type information of the initialized object. If the object type matches our structure candidate, we record the deallocation site and conclude that the object candidate has the potential to facilitate DirtyCred's privilege escalation.

Step 4: tracking down reachable objects. Note that the objects identified above may not be the ones under the user's control. For example, the objects might be capable of allocation only at the booting phase of the Linux. In this sense, DirtyCred cannot use these objects for memory manipulation and thus pivot a vulnerability's capability. Therefore, the last step is to examine whether the candidate objects can be allocated and freed through user-permitted system calls. To do it, we leverage a kernel fuzzer to explore the reachability of candidate objects' (de)allocation sites. Given a candidate object identified in the first three steps, if the fuzzer could trigger its allocation and deallocation sites using permitted system calls, we mark it as a valuable object for DirtyCred. For those that kernel fuzzer cannot reach out to the (de)allocation sites, we rely on our manual effort to analyze their reachability.

A.2 Implementation

To enable the first three analysis steps, we implemented a static analysis tool on top of LLVM. The analysis tool takes as input the kernel bitcode. To prevent the bitcode from being optimized, which might lose type information and make the data flow complex, we used a customized clang to generate bitcode before any code optimization is invoked. The tool contains 3,382 lines of C++ code in total. Our implementation is available at [2].

To complete the last step of the analysis, we utilized the state-of-the-art kernel fuzzer – Syzkaller [21]. To enable Syzkaller to report (de)allocation sites' reachability, we inserted a panic function at each site where a candidate object is (de)allocated. Once the Syzkaller generates an input that reaches the site, the kernel will experience panic, informing Syzkaller that the site has been reached. Benefited from the advance of kernel fuzzing, Syzkaller will also output a minimized input reachable to the corresponding (de)allocation sites if it triggers the corresponding panic functions. It should be noted that Syzkaller relies on syscall templates to generate the input and thus dynamically test the kernel. For some kernel modules, the templates have not yet been supported by Syzkaller. In this situation, we confirm the existence of (de)allocation manually.