pipe_buffer arbitrary read write

by Jayden R

Introduction

In this post we will look at an arbitrary read/write technique that can be used to achieve privilege escalation in a variety of Linux kernel builds. This has been used practically against Ubuntu builds, but the technique is amenable to other targets such as Android. It is particularly useful in cases where ARM64 User Access Override mitigates the common technique of setting `addr_limit` to ULONG_MAX.

The `pipe_buffer` technique was discovered independently by the author, but a recent Blackhat talk suggested the technique is being used in the wild. The technique provides an intuitive way to gain arbitrary read/write, so we suspect that it's been used widely for a long time.

The technique

The technique targets the page pointer of a pipe buffer. In ordinary pipe operations, this page stores data which was written through a pipe. The data may then be loaded from the page into userspace through a read operation. By overwriting said page pointer we're able to read from and write to arbitrary locations in the physical address space. This includes the kernel heap, targeting sensitive objects such as task descriptors and credentials, as well the writeable pages of the kernel image itself.

`struct pipe_buffer`

The `pipe_buffer` array is a common heap-allocated structure targeted in Linux kernel exploitation. By leaking a `pipe_buffer` element, we are able to deduce the kernel (virtual) base address. In overwriting the `pipe_buffer` we're typically able to gain code execution.

Each pipe is managed by the `pipe_inode_info` data structure. The `pipe_buffer` array comes automatically with every pipe. It is pointed to by the field `pipe_inode_info::bufs` and is treated as a ring through the `pipe_inode_info::tail` and `pipe_inode_info::head` indices.

The array is allocated from the memcg slab caches. It may be a variety of sizes. In particular we can have our `pipe_buffer` array be of size `n * sizeof(struct pipe_buffer)` where `n` is a power of 2. With `fcntl(pipe_fd, F_SETPIPE_SZ, PAGE_SIZE * n)` we can alter the pipe ring size.

A single element in the `pipe_buffer` array is structured as follows:

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

The typical leak and overwrite target is the `pipe_buf_operations` pointer. This is great if a ROP chain is sufficient to gain full privileges, however on some platforms such as Android this is not sufficient. For arbitrary read/write we will use the `page` pointer instead.

`struct page`

The kernel represents physical memory using `struct page` objects. These are all stored in a single array which we will call `vmemmap_base` (its symbol on some common platforms).

When we write to a pipe, the kernel reserves a new page to store our data. This page can then be spliced onto other pipes or have its contents copied back out from the read-side of the pipe.

```
static ssize_t pipe_write(struct kiocb *iocb, struct iov_iter *from)
{
. . .
       for (;;) {
. . .
              if (!pipe_full(head, pipe->tail, pipe->max_usage)) {
. . .
                     struct page *page = pipe->tmp_page;
. . .
                     if (!page) {
                            page = alloc_page(GFP_HIGHUSER | __GFP_ACCOUNT);
                            if (unlikely(!page)) {
                                   ret = ret ? : -ENOMEM;
                                   break;
                            }
                            pipe->tmp_page = 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;
. . .
copied = copy_page_from_iter(page, 0, PAGE_SIZE, from);
```

As we can see here, `alloc_page()` returns a new page from the page allocator. A `pipe_buffer` is then initialised to encapsulate the page. The user-supplied contents are copied into it. The exact mechanics of the page allocator is outside the scope of this post, but just think of it as a free-list of available physical pages.

The central question for this technique is: assuming we can corrupt a `pipe_buffer`, are we able to set `pipe_buffer::page` to the struct page representing a sensitive region of memory? We will look at two applications. The first targets heap memory and involves straightforward arithmetic. The second targets the kernel image itself and may require some additional brute-forcing.

Read and write into the kernel heap

We will further split this into two cases. In the first case, we assume that we know where a target object is in the heap. In the second case, we assume that we don't know where a target object is in memory and we need to find it.

With only a leaked `struct page` pointer we can:

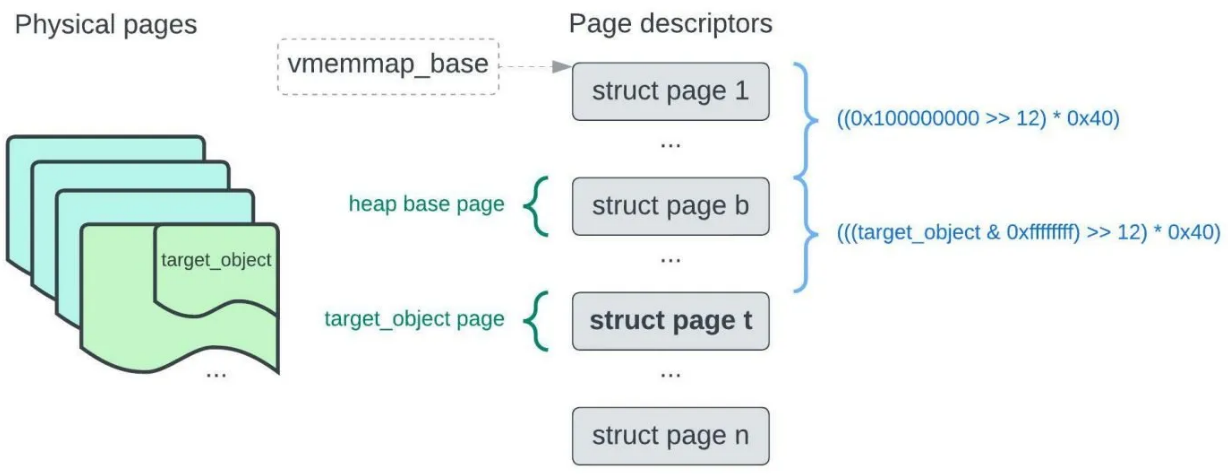
- Deduce the `vmemmap_base` address
- Calculate the physical page loadpoint of the heap base
- Repeatedly increment, scale, and rewrite the page pointer to seek across the heap

Suppose we're targeting the object with virtual address `0xffff98784d431d00` and we've leaked the `struct page` address `0xffffebea044d9f00`. Both are randomized with KASLR.

Through the mask `0xfffffffff0000000 & 0xffffebea044d9f00` we get `0xffffebea00000000` for `vmemmap_base`.

First, we ask the question: how can we choose the `struct page` which corresponds to the target object in the heap? Clearly, this target `struct page` will be `vmemmap + offset`. But what offset? Since the `vmemmap` array corresponds directly to physical memory and since the heap base is not typically (physically) randomized, we can use the simple formula:

```
vmemmap_base
+ ((0x100000000 >> 12) * 0x40)
+ (((target_object & 0xffffffff) >> 12) * 0x40)
```



Indexing the target object's page

The result of this formula gives the virtual address for the `struct page` element of the `vmemmap` array corresponding to the physical page which underlies our target object. A few questions remain: what is that `0x100000000`? Why shift by 12? Why scale by 0x40?

Luckily for us, the physical heap base is not randomized. It starts at physical address `0x100000000`. The 12-shift returns the "index" of the page in memory. For example, the address `0x100000000` corresponds to the `0x100000000 >> 12` page of memory. Finally, the 0x40-scale corrects the bytes offset in the `vmemmap_base` array according to the size of the elements. In other words, 0x40 is the size of a single `struct page`. The analogous operation is:
`int x[N]; int y = x[3];` which retrieves the value at `&x + (3 * sizeof(int))`.

So in plain words, the formula says:
Take the `vmmemap` base address and displace up to the `struct page` of the first physical page of heap memory. Then displace by the number of pages between the bottom of the heap and the target object.

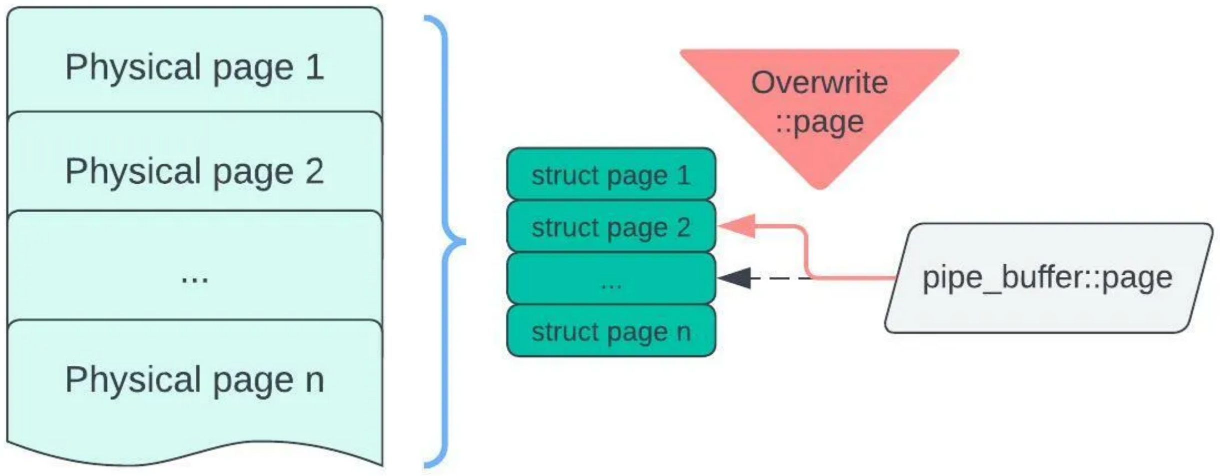
If we set the `pipe_buffer::page` to the result then we will be able to read/write to the page of the target object. Note that objects can lie over multiple pages. So it's important to determine the page relative to the target field(s) of an object rather than just from the beginning of the object.

This can be used to set the `pipe_buffer` fields as follows:

```
uint64_t cred_page = virt_to_page(target_obj, vbase);
uint64_t cred_off = (target_obj & 0xfff);


pbuf->page = (long *)cred_page;
pbuf->offset = cred_off + 0x4;
```

```
pbuf->len = 0;

pbuf->ops = (long *)FAKE_OPS;

pbuf->flags = PIPE_BUF_FLAG_CAN_MERGE;

pbuf->private = 0;

...

write(dest_pipe.wr, zeroes, 0x20);
```

where `virt_to_page()` is the implementation of the formula. As the reader can see, we targeted the cred object of our task, overwriting the *id fields with zeroes to escalate privileges. This assumes we already know the virtual address of our task credentials.



Manipulating the page pointer of a pipe buffer

On the other hand, we might not yet know the address. In this case we would need to seek through the heap to identify our `task_struct` and then our `struct cred`. One way to do this is to use `prctl()` to change our task's name just before searching for it. Since `prctl()` changes the `task_struct::comm` field to the new name we can use this, as well as some other determinants, to confirm that we've found the `task_struct`.

To do this, we loop with `i` over `(vmemmap_base + ((0x100000000 >> 12) * 0x40)) + (0x40 * i)`, writing it back as the `pipe_buffer::page`. We can then repeatedly leak heap memory, halting when we find our `task_struct`. Once we've read out this final leak we'll have our `cred` virtual address. From here, we are in the first case again as shown above. This likely means we need to retrigger the vulnerability a substantial number of times.

Avoiding reallocation

One possible scenario is when we know the address of our object replacement in memory but not the address of our target object. For example, we know the address of a `msg_msgseg` which has overlaid a `pipe_buffer` array but not the address of the credentials which we ultimately need to overwrite.

If this is the case, then we can repeatedly overwrite the seeker `pipe_buffer` by setting the page pointer of an overwriter pipe to the seeker `pipe_buffer` page. This works as follows:

1. Calculate the struct page address of the seeker `pipe_buffer` page.
2. Create another pipe – our overwriter.
3. Trigger the use-after-free and write to the seeker `pipe_buffer::page` its own page.
4. Call `tee()` with the seeker pipe as source and the overwrite pipe as destination.

Now we have a reliable way to overwrite the seeker `pipe_buffer` without reallocating it. We can use this in the following way:

```
void set_new_pipe_bufs_overwrite(char *buf, struct pipe_struct *overwrite,
                                 char *obj_in_page, struct pipe_buffer *pbuf,
                                 uint64_t new_page, uint32_t len, int *tail)
{
    if(read(overwrite->read, buf, PAGE_SIZE) != PAGE_SIZE)
        error_out("read overwriter_pipe[0]");

    struct pipe_buffer *setpbuf = (struct pipe_buffer *) obj_in_page;
    setpbuf += (*tail) % 8;
    (*tail)++;
    *setpbuf = *pbuf;
    setpbuf->page = (void *) new_page;
    setpbuf->len = len;

    if (write(overwrite->write, buf, PAGE_SIZE) != PAGE_SIZE)
        error_out("write overwriter_pipe[1]");
}
```

The `read()` sets the overwriter's `pipe_inode_info::tmp_page` to the seeker `pipe_buffer` object's underlying page. This temporary page can then we written to directly. After this, we construct the new `pipe_buffer` in our buffer. Finally, we write out the new seeker `pipe_buffer` with the overwriter pipe. Behind the scenes, the kernel copies the given contents into the physical page represented by the overwriter pipe's `pipe_inode_info::tmp_page`. This circumvents repeated reallocation of the use-after-free object.

Reading and writing to the kernel image

Let's suppose that we need to target a variable in the kernel image itself. Or that we don't want to seek through the whole heap, instead opting to traverse a list such as via `init_task` to find our task and then our `cred`. Can we just leak a kernel image virtual address (e.g. `pipe_buffer::ops`) and then use the `virt_to_page()` formula as before?

On x86 systems, with KASLR enabled, this is not possible. The option `CONFIG_RANDOMIZE_BASE` for this architecture randomizes the physical load address and the virtual base address separately. That is, one cannot be used to derive the other.

To discover the kernel image base we need to have leaked a `struct page` pointer (or else have a partial overwrite primitive of the first qword of a `pipe_buffer`). We also need to know a byte pattern in the kernel image, at some offset, to confirm when we've found our target page.

Let's take the first qword of Ubuntu 22.04: `0x4801e03f51258d48` which introduces the `startup_64` function. We'll seek from some offset in `vmemmap` until we find this byte pattern at the first leaked qword read from our corrupted `pipe_buffer`. But doesn't this mean we need to seek across every single page a hard slog in 0x40 byte increments?

Luckily, the kernel can't be loaded at any arbitrary physical address. It's constrained by `CONFIG_PHYSICAL_ALIGN`. It will also be randomized above `CONFIG_PHYSICAL_START`. So we need only check in `CONFIG_PHYSICAL_ALIGN` increments.

Further, for ARM64 systems Kconfig has something different to say:

> `CONFIG_RANDOMIZE_BASE` randomizes the virtual address at which the kernel image is loaded, as a security feature that deters exploit attempts relying on knowledge of the location of kernel internals.

This is most interesting for Android and means that physical base randomization needs to be implemented by third-party vendors.

Regardless, it's demonstrably feasible to brute-force the randomized physical base without any optimization. However, one method to speed things up is to search by increments of `(N * CONFIG_PHYSICAL_ALIGN)` and in a single leak check that any of the qwords of the N offsets of the kernel image, are present.

For example, in Ubuntu's case we have the alignment `0x200000`. But we don't want to check every `0x200000`th physical address for the `startup_64` qword. So we seek by `(0x200000 * 8)` at a time and check for any of 8 known qword at offsets `(0x200000 * (0 < n < 9))` in the kernel image. Once we find one, we displace backwards by the right offset and we've got the physical base.

```
bool search_phys_base(const char *buf, int64_t x,
                          uint64_t *scroll_page)
{
#define QWORD2 0x95e8e58948f63155
#define QWORD3 0x6548478b4c48ec83
#define QWORD4 0x000004c8908b0000
#define QWORD5 0xb0458948ffffff60
#define QWORD6 0x04ba550000441f0f
#define QWORD7 0x4500000233840f40
#define QWORD8 0x2524894865f8010f
#define scale_offset(i) (((((0x200000) * i) >> 12) * 0x40))

    uint64_t first_qword = ((uint64_t *)buf)[0];

    switch (first_qword) {
        case STARTUP_QWORD_5_15:
            break;
        case QWORD2:
            *scroll_page -= scale_offset(1);
            break;
        case QWORD3:
            *scroll_page -= scale_offset(2);
            break;
. . .
```

Once we've got the struct page representing the physical base, we can easily derive the `struct page` for the target object in the kernel image as the kernel image pages are ordered to be physically contiguous. For example, `uint64_t init_task_page = kbase_page + ((INIT_TASK_OFF >> 12) * 0x40);` where `INIT_TASK_OFF` is the known offset of the `init_task` in the kernel image.

Additional considerations

Limiting factors

As outlined above, we need to leak (or partially overwrite) a struct page pointer. We may also need to brute-force up to the physical base page for the kernel image. This latter factor can increase the running time of an exploit which uses this technique.

It's also not possible to write to read-only kernel memory through this method. We can't just alter some system call's implementation to run our own shellcode. An extension of the technique might, however, target page tables directly in order to switch permission bits to then write out to read-only memory.

Grace factors

The Linux memory model sees physical page frames as a substratum of raw memory – ready to be linked with virtual addresses, or used directly, when storing and loading data. This allows us to use kernel pages in pipes where there really ought to only be user pages. Further, we may be able to target other process' user pages to leak secrets or corrupt internal data anyway. So armed with knowledge of kernel page dynamics, as well as with a `pipe_buffer` corruption primitive, it is possible to do very interesting things in the physical address space.

Search for...

Recent Posts

pipe_buffer arbitrary read write

Converting IDA DB to VxWorks .sym

Game Hacking with Binary Ninja

Dissection of a Payment Terminal

So you want to work in Vulnerability Research?