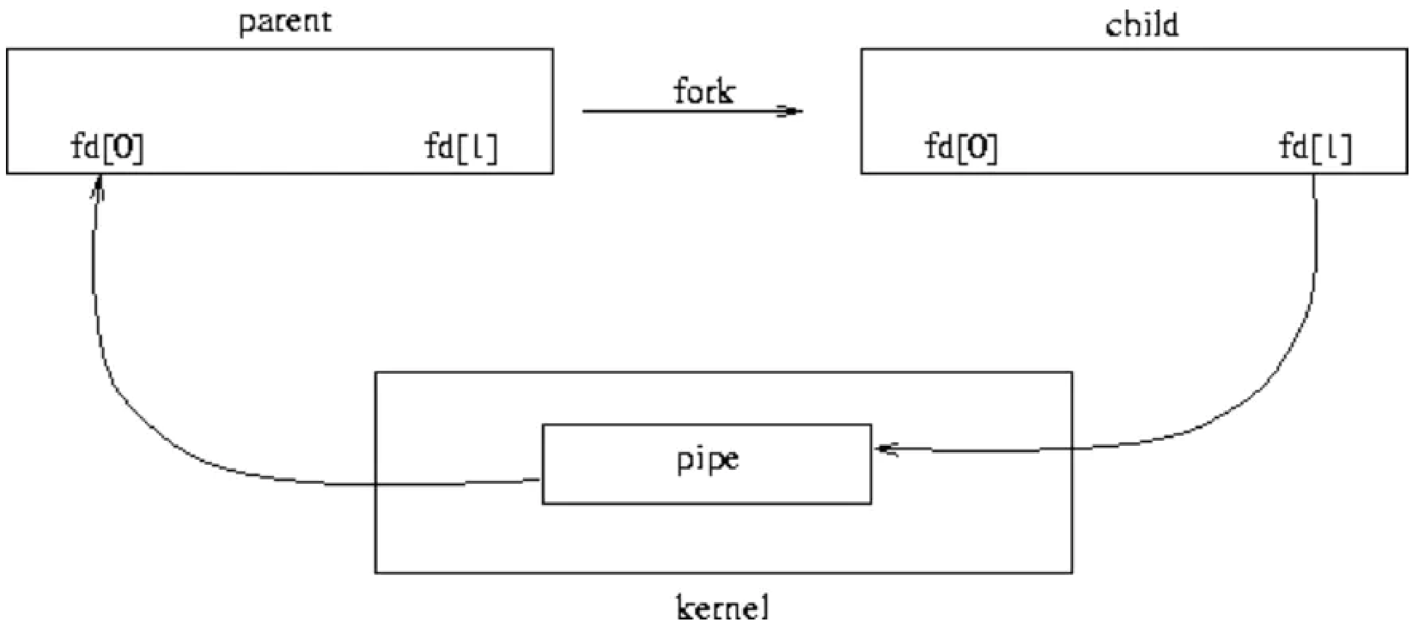
Dirty Pipe
CVE-2022-0847

保密2001
林俊烨

pipe 机制

- 管道（pipe）是Linux系统中重要的进程间通信（IPC）机制，又分为匿名管道（anonymous pipe）和命名管道（named pipe/FIFO）两种.
- 匿名管道在两个**有亲缘关系**的进程（即存在父子或兄弟关系的进程）之间创建，本质上是由内核管理的一小块内存缓冲区，默认大小由系统中的PIPE_BUF常量指定（默认为一页，即4096字节）。



pipe 相关结构体

```
struct pipe_inode_info {
    struct mutex mutex;
    wait_queue_head_t rd_wait, wr_wait;
    unsigned int head;
    unsigned int tail;
    unsigned int max_usage;
    unsigned int ring_size;
    unsigned int readers;
    unsigned int writers;
    unsigned int files;
    unsigned int r_counter;
    unsigned int w_counter;
    struct page *tmp_page;
    struct fasync_struct *fasync_readers;
    struct fasync_struct *fasync_writers;
    struct pipe_buffer *bufs;
    struct user_struct *user;
};
```

```
low addr                                high
addr
+---------------------------------------------+
|  |  |  |  |  |  |  | >|//|//|//|> |  |  |  |
+---------------------------------------------+
                       A    ---->   A
                       |            |
                      tail         head
```

pipe 相关结构体

```
/**
 *  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;
};

// include/linux/pipe_fs_i.h
#define PIPE_BUF_FLAG_LRU       0x01    /* page is on the LRU */
#define PIPE_BUF_FLAG_ATOMIC    0x02    /* was atomically mapped */
#define PIPE_BUF_FLAG_GIFT      0x04    /* page is a gift */
#define PIPE_BUF_FLAG_PACKET    0x08    /* read() as a packet */
#define PIPE_BUF_FLAG_CAN_MERGE 0x10    /* can merge buffers */
```

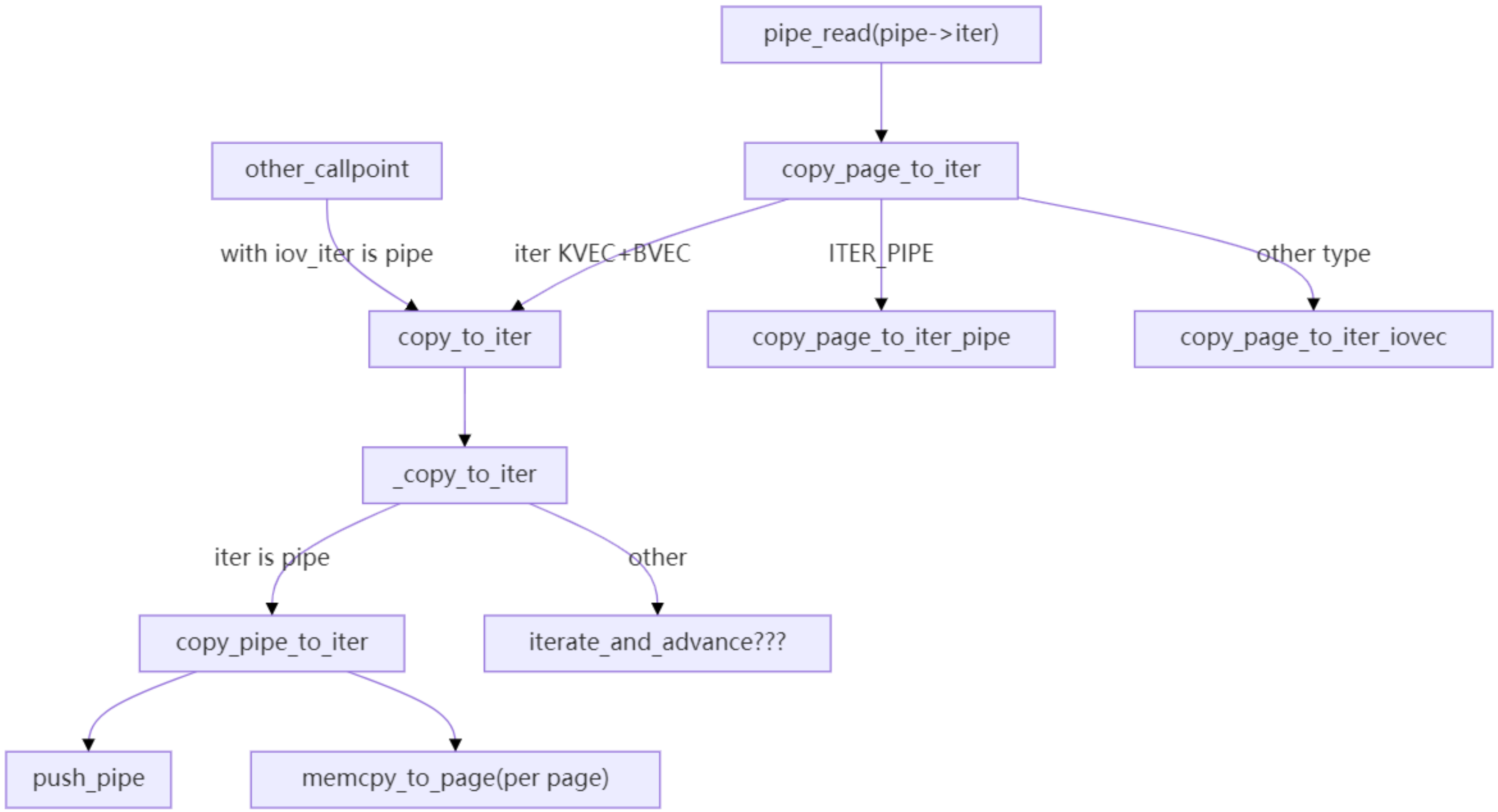
该结构体将用于迭代一个个Page

```
enum iter_type {
    /* iter types */
    ITER_IOVEC = 4,
    ITER_KVEC = 8,
    ITER_BVEC = 16,
    ITER_PIPE = 32,     // 表示正在迭代的数据是位于 pipe 中的
    ITER_DISCARD = 64,
};

struct iov_iter {
    /*
     * Bit 0 is the read/write bit, set if we're writing.
     * Bit 1 is the BVEC_FLAG_NO_REF bit, set if type is a bvec and
     * the caller isn't expecting to drop a page reference when done.
     */
    unsigned int type;
    size_t iov_offset;
    size_t count;
    union {
        const struct iovec *iov;
        const struct kvec *kvec;
        const struct bio_vec *bvec;
        struct pipe_inode_info *pipe;
    };
    union {
        unsigned long nr_segs;
        struct {
            unsigned int head;
            unsigned int start_head;
        };
    };
};
```

pipe_read(struct kiocb *iocb, struct iov_iter *to)

- iocb：中存放着获取当前 pipe 结构体的指针
- to：从管道读出来的数据将要写入的地方，iov_iter 迭代器类型。

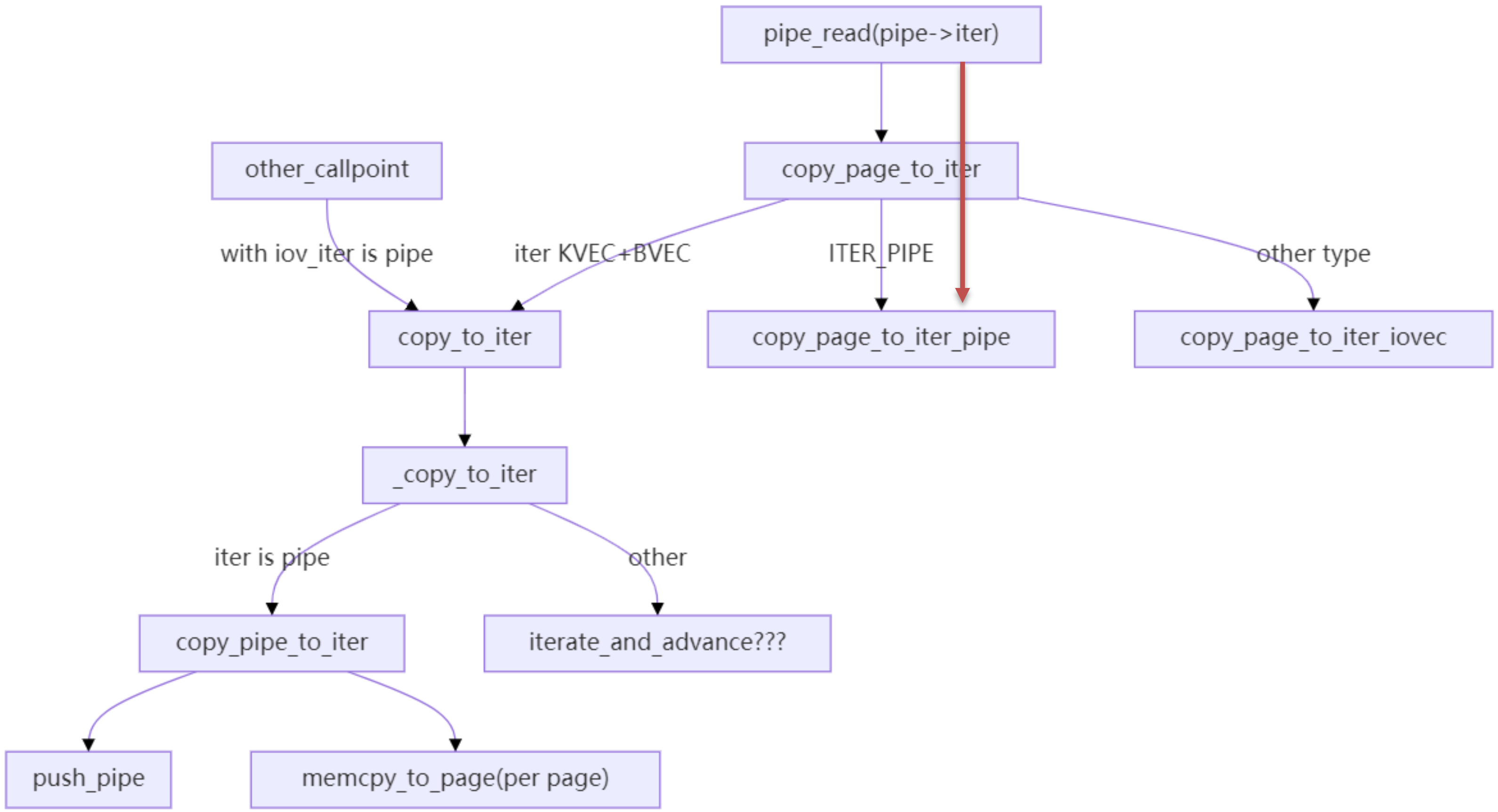
大致流程： 循环遍历pipe->bufs数组，使用*copy_page_to_iter*将buf中的一整个page复制到iter中，如果iter是pipe，则不复制直接引用，如此循环再顾及到截断等问题就结束读取.



pipe_read(struct kiocb *iocb, struct iov_iter *to)

- iocb: 中存放着获取当前 pipe 结构体的指针
- to: 从管道读出来的数据将要写入的地方，iov_iter 迭代器类型。

大致流程： 循环遍历pipe->bufs数组，使用*copy_page_to_iter*将buf中的一整个page复制到iter中，如果iter是pipe，则不复制直接引用，如此循环再顾及到截断等问题就结束读取.



copy_page_to_iter_pipe

由于*copy_page_to_iter_pipe*中pipe buf 是**直接引用其他页**，因此在修改buf的地方必须确保新传来的数据不会写入这样的页面中，而这种保证就依赖于 MERGE 标志位。然而可以看到虽然 recv pipe buf 结构体上的众多字段都被重新赋值，**但有一个字段却被遗漏了，那就是 flags 字段！**

```
static size_t copy_page_to_iter_pipe(struct page *page, size_t offset, size_t bytes,
             struct iov_iter *i)
{
    ......
    buf->ops = &page_cache_pipe_buf_ops;
    // 增加该页的 refcount
    get_page(page);
    buf->page = page;   // 直接引用已有的页
    buf->offset = offset;
    buf->len = bytes;
    /* !!! 需要注意的是，这里没有对 buf 的 flag 字段初始化！ */

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

pipe_write: 把数据从iter复制到pipe中

函数第一段

```
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) &&
        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;
    }
}
```

如果说当前 pipe buf 中已经存在数据，

- 并且**数据总长度不是页大小的整数倍**
- pipe buf的起始位置+
  pipe已有数据长度+
  iter总长度mod页大小 < PAGE_SIZE,

那么直接先把iter<u>开头一段</u>填充到pipe buf中进行数据合并。

pipe_write: 把数据从iter复制到pipe中

函数第一段

```
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) &&
        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;
    }
}
```

如果说当前 pipe buf 中已经存在数据，

- 并且**数据总长度不是页大小的整数倍**
- pipe buf的起始位置+

pipe已有数据长度+

iter总长度mod页大小 < PAGE_SIZE，

那么直接先把iter<u>开头一段</u>填充到pipe buf中进行数据合并。

pipe_write: 把数据从iter复制到pipe中

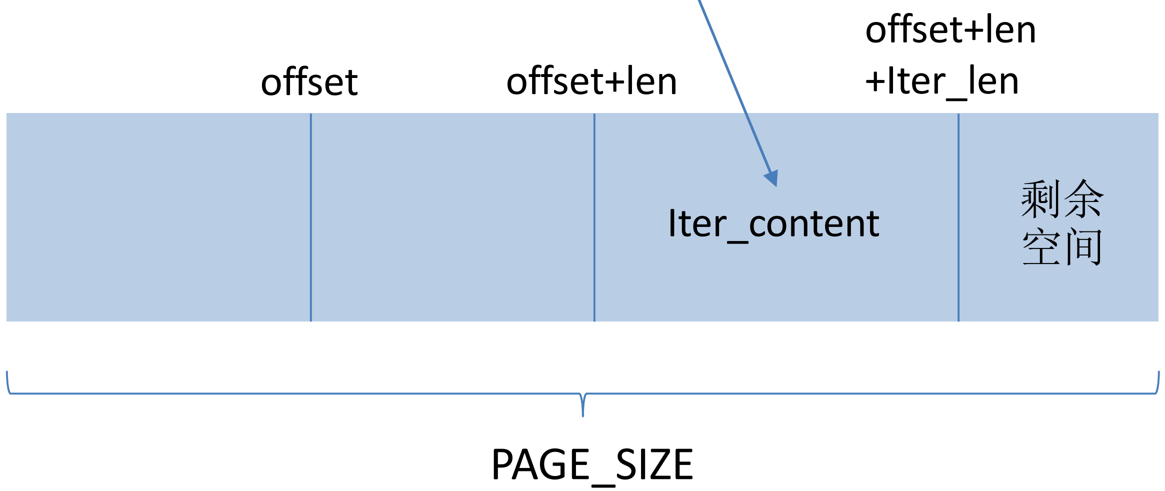
函数第一段

```
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) &&
        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;
    }
}
```

如果说当前 pipe buf 中已经存在数据，

- 并且**数据总长度不是页大小的整数倍**
- pipe buf的起始位置+pipe已有数据长度+iter总长度mod页大小 < PAGE_SIZE，

那么直接先把iter<u>开头一段</u>填充到pipe buf中进行数据合并。



do_splice():将某个 fd 的数据直接拷贝进另一个 fd 中

```
/*
 * Determine where to splice to/from.
 */
long do_splice(struct file *in, loff_t __user *off_in,
        struct file *out, loff_t __user *off_out,
        size_t len, unsigned int flags)
{
    struct pipe_inode_info *ipipe;
    struct pipe_inode_info *opipe;
    ...;
    ipipe = get_pipe_info(in);
    opipe = get_pipe_info(out);
    ...;

    // 当数据从文件复制给管道时
    if (opipe) {
        ...
        ret = wait_for_space(opipe, flags);
        // 如果等到 pipe 存在空闲空间后
        if (!ret) {
            unsigned int p_space;
             // 获取待传递数据大小
            /* Don't try to read more the pipe has space for. */
            p_space = opipe->max_usage - pipe_occupancy(opipe->head, opipe->tail);
            len = min_t(size_t, len, p_space << PAGE_SHIFT);
            // 执行真正的传递操作
            ret = do_splice_to(in, &offset, opipe, len, flags);
        }
        ...
        return ret;
```

只关注From-fd为file，To-fd为pipe，
即数据**从文件传递至管道**的情况

代码浅析

do_splice_to ()

```
/*
 * Attempt to initiate a splice from a file to a pipe.
 */
static long do_splice_to(struct file *in, loff_t *ppos,
            struct pipe_inode_info *pipe, size_t len,
            unsigned int flags)
{
    ... //some security check
    // 调用 splice_read 函数
    if (in->f_op->splice_read)
        return in->f_op->splice_read(in, ppos, pipe, len, flags);
    return default_file_splice_read(in, ppos, pipe, len, flags);
}


// fs/ext4/file.c
const struct file_operations ext4_file_operations = {
    ...
    .read_iter    = ext4_file_read_iter,
    ...
    .splice_read  = generic_file_splice_read,
    ...
};
```

只关注From-fd为file，To-fd为pipe，
即数据**从文件传递至管道**的情况

```
ssize_t generic_file_splice_read(struct file *in, loff_t *ppos,
                struct pipe_inode_info *pipe, size_t len,
                unsigned int flags)
{
    ...
    // 根据 pipe 结构体，创建 iov_iter 结构
    iov_iter_pipe(&to, READ, pipe, len);
    i_head = to.head;
    // 创建 kiocb 结构
    init_sync_kiocb(&kiocb, in);
    kiocb.ki_pos = *ppos;
    // 调用 call_read_iter 执行实际的数据传输操作 ！！！
    ret = call_read_iter(in, &kiocb, &to);
    ...
}
```

代码浅析

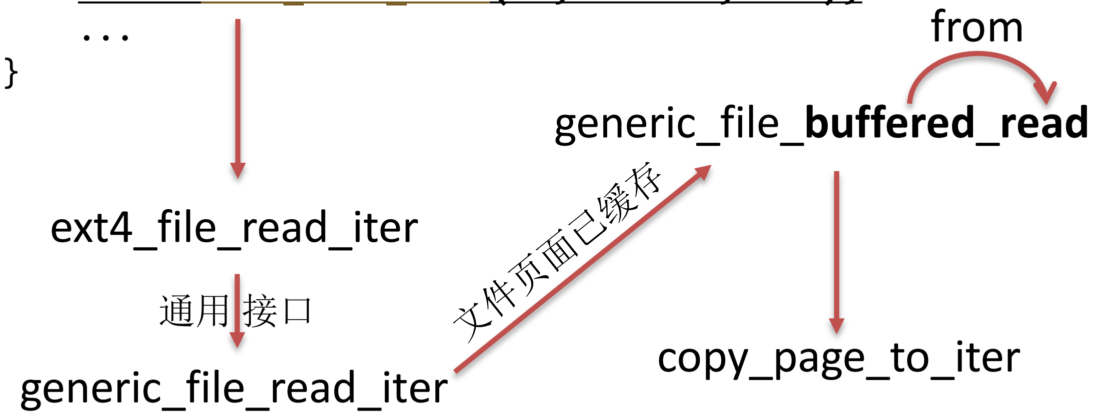
do_splice_to ()

```
/*
 * Attempt to initiate a splice from a file to a pipe.
 */
static long do_splice_to(struct file *in, loff_t *ppos,
            struct pipe_inode_info *pipe, size_t len,
            unsigned int flags)
{
    ... //some security check
    // 调用 splice_read 函数
    if (in->f_op->splice_read)
        return in->f_op->splice_read(in, ppos, pipe, len, flags);
    return default_file_splice_read(in, ppos, pipe, len, flags);
}
```

```
// fs/ext4/file.c
const struct file_operations ext4_file_operations = {
    ...
    .read_iter    = ext4_file_read_iter,
    ...
    .splice_read  = generic_file_splice_read,
    ...
};
```

只关注From-fd为file，To-fd为pipe，
即数据**从文件传递至管道**的情况

```
ssize_t generic_file_splice_read(struct file *in, loff_t *ppos,
            struct pipe_inode_info *pipe, size_t len,
            unsigned int flags)
{
    ...
    // 根据 pipe 结构体，创建 iov_iter 结构
    iov_iter_pipe(&to, READ, pipe, len);
    i_head = to.head;
    // 创建 kiocb 结构
    init_sync_kiocb(&kiocb, in);
    kiocb.ki_pos = *ppos;
    // 调用 call_read_iter 执行实际的数据传输操作 ！！！
    ret = call_read_iter(in, &kiocb, &to);
    ...
}
```



发现者的Exploit

1. 创建管道（务必不要带上 O_DIRECT）
2. 往管道中直接写入大量数据，使得 pipe 结构体中所有 page buf 的 flag 全部都设置了 PIPE_BUF_FLAG_CAN_MERGE 标志。
3. 从该管道中将数据全部读取出来，释放所有 page buf。
4. 调用 splice，将**数据长度不与页大小对齐**的**可读**文件数据，传递至该管道中。这样在管道的 head 位置，势必会有一个 page buf，其中 **page 指向文件缓存**，**flags 为 PIPE_BUF_FLAG_CAN_MERGE**。
5. 因为 page buf 在重分配时不会初始化 flags，因此这里的 flags 将仍然保留为 PIPE_BUF_FLAG_CAN_MERGE。
6. 直接继续往该管道中写入目标数据，这样由于 PIPE_BUF_FLAG_CAN_MERGE 标志仍然存在，新写入的数据将会直接与 page buf 所指向的文件缓存合并。
7. 此时访问该文件，则内核会将被修改后的文件缓存中的数据返回，这样便可达到在内核层面任意文件写的目的。

漏洞复现

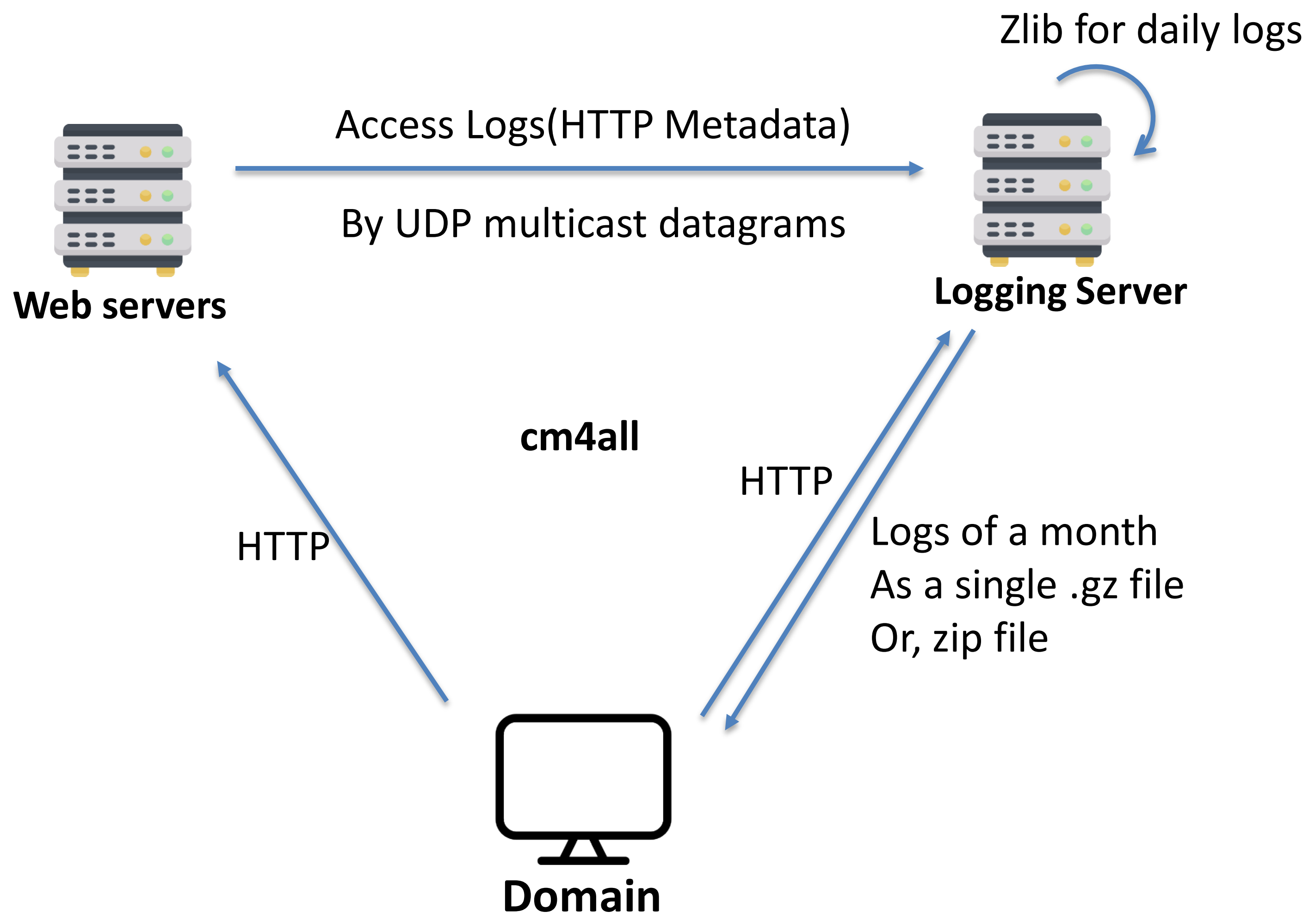
测试环境: Kali Linux 2022
Linux commit id: f6dd975583bd
接下来是实际运行.

1. 下载对应linux
2. 设置并编译linux: menu or manual.
3. 解决编译中的问题
4. 下载编译busybox
5. 编译exp
6. 设置虚拟linux环境: init script, /etc/passwd, launch.sh.
7. qemu启动!
8. 看两眼passwd有什么变化

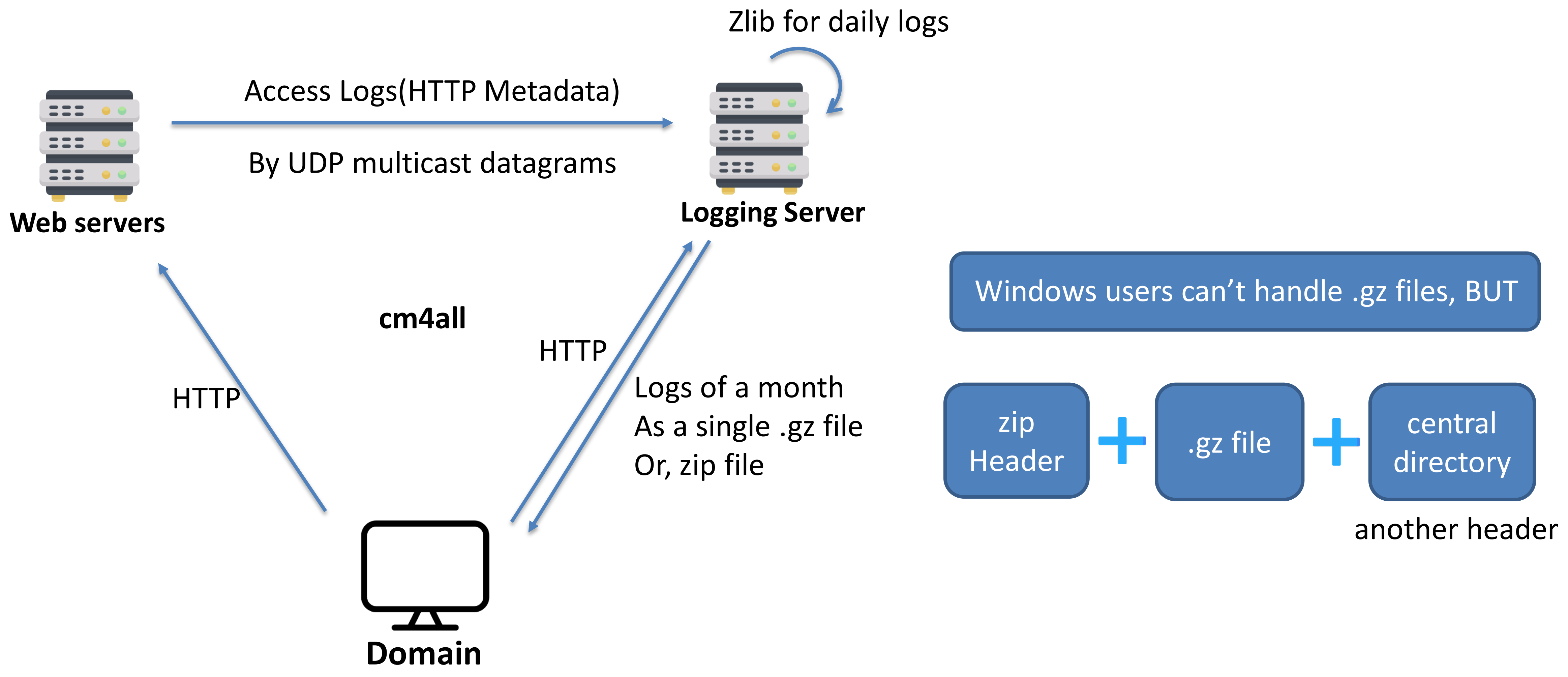
漏洞历程

- Long ago, struct pipe_buf_operations had **a field 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 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**.
- 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.**

漏洞场景


Zlib for daily logs
Access Logs(HTTP Metadata)
By UDP multicast datagrams
Web servers
Logging Server
cm4all
HTTP
HTTP
Logs of a month
As a single .gz file
Or, zip file
Domain

漏洞场景


Zlib for daily logs
Access Logs(HTTP Metadata)
By UDP multicast datagrams
Web servers
Logging Server
cm4all
HTTP
HTTP
Logs of a month
As a single .gz file
Or, zip file
Domain
Windows users can't handle .gz files, BUT
zip Header
.gz file
central directory
another header

发现异常

Normal end of a proper daily file(.gz file)

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 9c 12 0b f5 f7 4a 00 00
```

Corrupted file end

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 50 4b 01 02 1e 03 14 00
```

Tips:

- 00 00 ff ff 结束标志位
- 03 00 empty “final” block
- 9c 12 0b f5 CRC32
- f7 4a 00 00 未压缩文件大小

发现异常

Normal end of a proper daily file(.gz file)

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 9c 12 0b f5 f7 4a 00 00
```

Corrupted file end

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 50 4b 01 02 1e 03 14 00
```

Tips:

- 00 00 ff ff 结束标志位
- 03 00 empty “final” block
- 9c 12 0b f5 CRC32
- f7 4a 00 00 未压缩文件大小

all of them had the **same** CRC32 and the **same** “file length” value.

Why?

发现异常

Normal end of a proper daily file(.gz file)

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 9c 12 0b f5 f7 4a 00 00
```

Corrupted file end

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 50 4b 01 02 1e 03 14 00
```

Tips:

- 00 00 ff ff 结束标志位
- 03 00 empty "final" block
- 9c 12 0b f5 CRC32
- f7 4a 00 00 未压缩文件大小

all of them had the **same** CRC32 and the **same** "file length" value.

Why?

stared at these 8 bytes

50 4b 01 02 1e 03 14 00

- 50 4b is "PK"
- 01 02 is the code for central directory file header.
- "Version made by" = 1e 03; 0x1e = 30 (3.0); 0x03 = UNIX
- "Version needed to extract" = 14 00; 0x0014 = 20 (2.0)

发现异常

Normal end of a proper daily file(.gz file)

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 9c 12 0b f5 f7 4a 00 00
```

Corrupted file end

```
000005f0 81 d6 94 39 8a 05 b0 ed e9 c0 fd 07 00 00 ff ff
00000600 03 00 50 4b 01 02 1e 03 14 00
```

Tips:
- 00 00 ff ff 结束标志位
- 03 00 empty “final” block
- 9c 12 0b f5 CRC32
- f7 4a 00 00 未压缩文件大小

all of them had the **same** CRC32 and the **same** “file length” value.

Why?

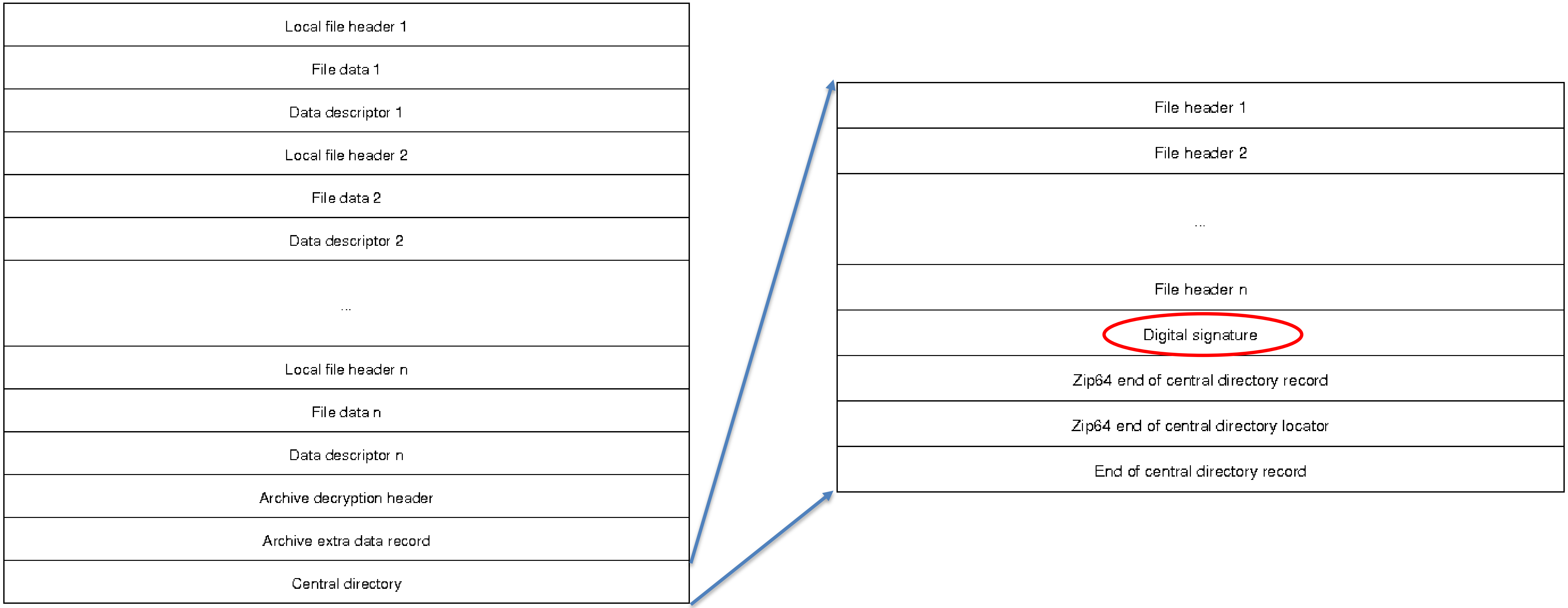
stared at these 8 bytes

50 4b 01 02 1e 03 14 00

- 50 4b is “PK”
- 01 02 is the code for central directory file header.
- “Version made by” = 1e 03; 0x1e = 30 (3.0); 0x03 = UNIX
- “Version needed to extract” = 14 00; 0x0014 = 20 (2.0)

There is one process which generates “PK” headers, though; it’s the web service **which constructs ZIP files** on-the-fly. But this process runs as a different user which doesn’t have write permissions on these files. It cannot possibly be that process.

插叙—zip格式


Local file header 1
File data 1
Data descriptor 1
Local file header 2
File data 2
Data descriptor 2
...
Local file header n
File data n
Data descriptor n
Archive decryption header
Archive extra data record
Central directory
File header 1
File header 2
...
File header n
Digital signature
Zip64 end of central directory record
Zip64 end of central directory locator
End of central directory record

继续收集信息

- there were 37 corrupt files within the past 3 months
- they occurred on 22 unique days
- 18 of those days have 1 corruption
- 1 day has 2 corruptions (2021-11-21)
- 1 day has 7 corruptions (2021-11-**30**)
- 1 day has 6 corruptions (2021-12-**31**)
- 1 day has 4 corruptions (2022-01-**31**)

- Only the **primary** log server had corruptions (the one which served HTTP connections and constructed ZIP files).
- The **standby** server (HTTP inactive but same log extraction process) had zero corruptions.

the web service writes a ZIP header:

- Read from .gz file
- uses *splice*() to send all compressed files
- finally uses *write*() again for the "central directory file header", which begins with 50 4b 01 02 1e 03 14 00, exactly the corruption.

继续收集信息

- there were 37 corrupt files within the past 3 months
- they occurred on 22 unique days
- 18 of those days have 1 corruption
- 1 day has 2 corruptions (2021-11-21)
- 1 day has 7 corruptions (2021-11-**30**)
- 1 day has 6 corruptions (2021-12-**31**)
- 1 day has 4 corruptions (2022-01-**31**)

- Only the **primary** log server had corruptions (the one which served HTTP connections and constructed ZIP files).
- The **standby** server (HTTP inactive but same log extraction process) had zero corruptions.

the web service writes a ZIP header:

- Read from .gz file
- uses *splice*() to send all compressed files
- finally uses *write*() again for the "central directory file header", which begins with 50 4b 01 02 1e 03 14 00, exactly the corruption.

The last day of the month is always **followed by the "PK" header.** That's why it's more likely to corrupt the last day.

思考过程……?

After being stuck for more hours, after **eliminating everything** that was definitely impossible (in my opinion), I drew a conclusion: this must be **a kernel bug**.

思考过程……?

After being stuck for more hours, after **eliminating everything** that was definitely impossible (in my opinion), I drew a conclusion: this must be **a kernel bug**.

In a moment of **extraordinary clarity**, I hacked two C programs.

蹦出来的两段程序

```c
#include <unistd.h>
int main(int argc, char **argv) {
  for (;;) write(1, "AAAAA", 5);
}
// ./writer >foo
```

log splitter

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char **argv) {
  for (;;) {
    splice(0, 0, 1, 0, 2, 0);
    write(1, "BBBBB", 5);
  }
}
// ./splicer <foo | cat >/dev/null
```

ZIP generator

蹦出来的两段程序

```c
#include <unistd.h>
int main(int argc, char **argv) {
  for (;;) write(1, "AAAAA", 5);
}
// ./writer >foo
```

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char **argv) {
  for (;;) {
    splice(0, 0, 1, 0, 2, 0);
    write(1, "BBBBB", 5);
  }
}
// ./splicer <foo |cat >/dev/null
```

- All bugs become shallow once they can be **reproduced**.
- A quick check verified that this bug affects Linux 5.10 (Debian Bullseye) but not Linux 4.19 (Debian Buster).
- There are **185 011** git commits between v4.19 and v5.10, but thanks to **git bisect**, it takes **just 17 steps** to locate the faulty commit.

Binary Search

Truth

the write() call that writes the central directory file header will be written to the **page cache** of the last compressed file

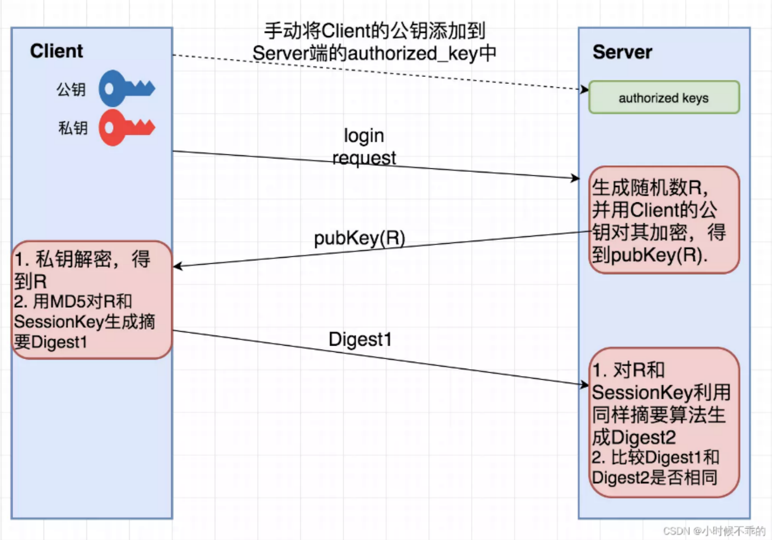
Why only the first 8 bytes of that header? Actually, this operation does not increase the file size. The original file had **only 8 bytes of “unspliced” space at the end**

the page cache is **always writable** (by the kernel), and writing to a pipe never checks any permissions.

还能修改什么?

1. **Authorized Keys**
2. **Setuid file**
3. **Cron Job**
4. **……**

[Dirty Pipe Exploit CVE-2022-0847 — Raxis]



参考资料

- Linux Dirty Pipe CVE-2022-0847 漏洞分析 | Kiprey's Blog
- The Dirty Pipe Vulnerability — The Dirty Pipe Vulnerability documentation
- Dirty Pipe Exploit CVE-2022-0847 — Raxis
- AlexisAhmed/CVE-2022-0847-DirtyPipe-Exploits: A collection of exploits and documentation that can be used to exploit the Linux Dirty Pipe vulnerability.
- pwncollege/pwnkernel: Kernel development & exploitation practice environment.
- ZIP (file format) – Wikipedia
- Zlib Flush Modes
- filemap.c - mm/filemap.c - Linux source code (v5.4) – Bootlin
- https://tryhackme.com/room/dirtypipe -- 非常详细的讲解加实践

感谢观看