Track 2
Kernel Exploitation With A File System Fuzzer

Donghee Kim, Seungpyo Hong, Wonyoung Jung, Heoungjin Jo

Best of the Best

HITB+CyberWeek UAE
Virtual Edition, 15-19 November 2020

INDEX

- Who we are?
- Abstract
- Introduction
- Background
- Evaluated Various OS
- Triage & monitor
- Analysis Crashes
- Kernel exploitation
- Demo

Who we are?

[Mentor]
Sangsup Lee

[PL]
Sungjun Cho

[Mentor]
Chunsung Park

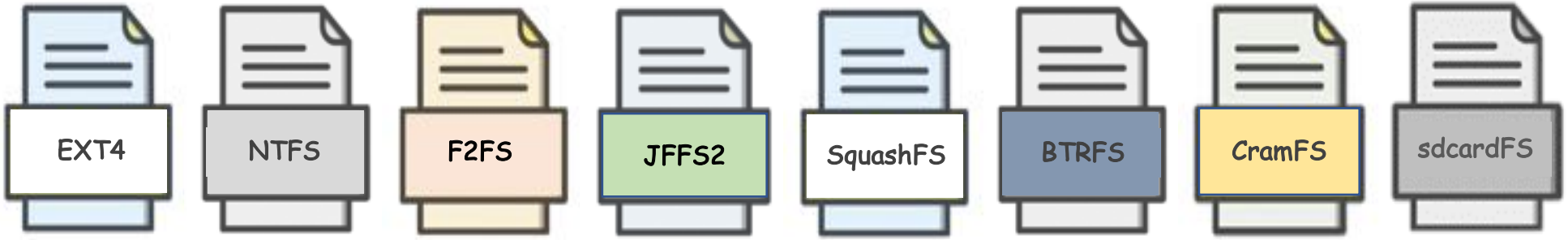
PM / Researcher
Donghee Kim

Researcher
Heoungjin Cho

Researcher
Seoungpyo Hong

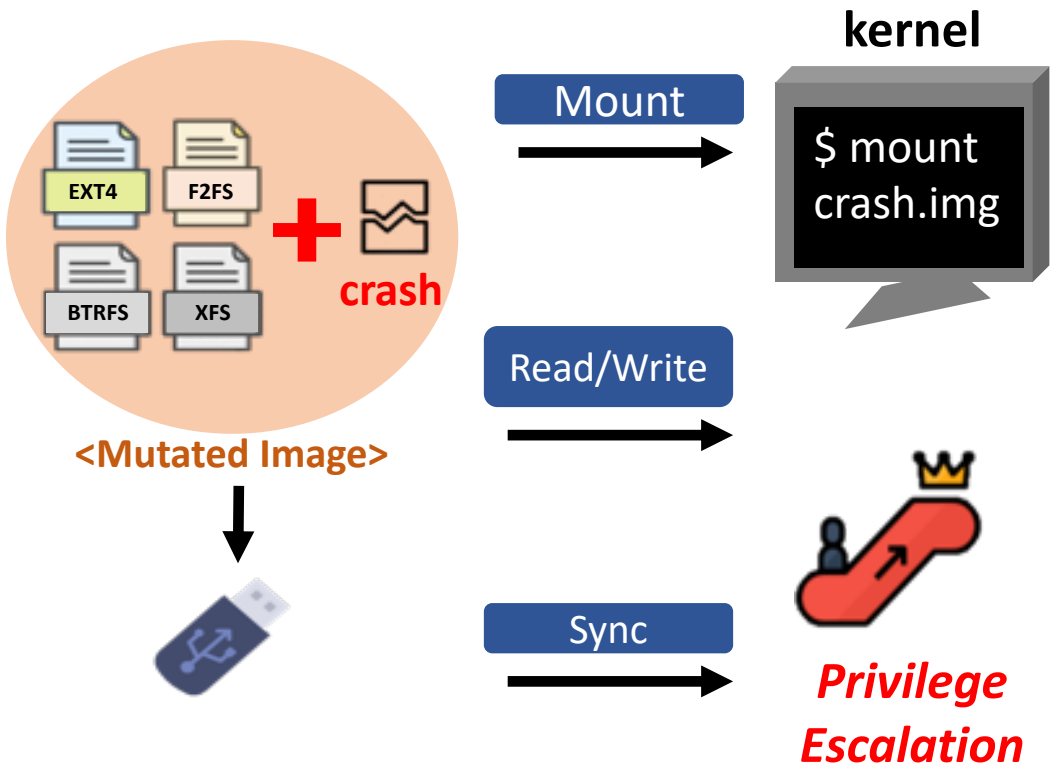
Researcher
Wonyoung Jung

Abstract

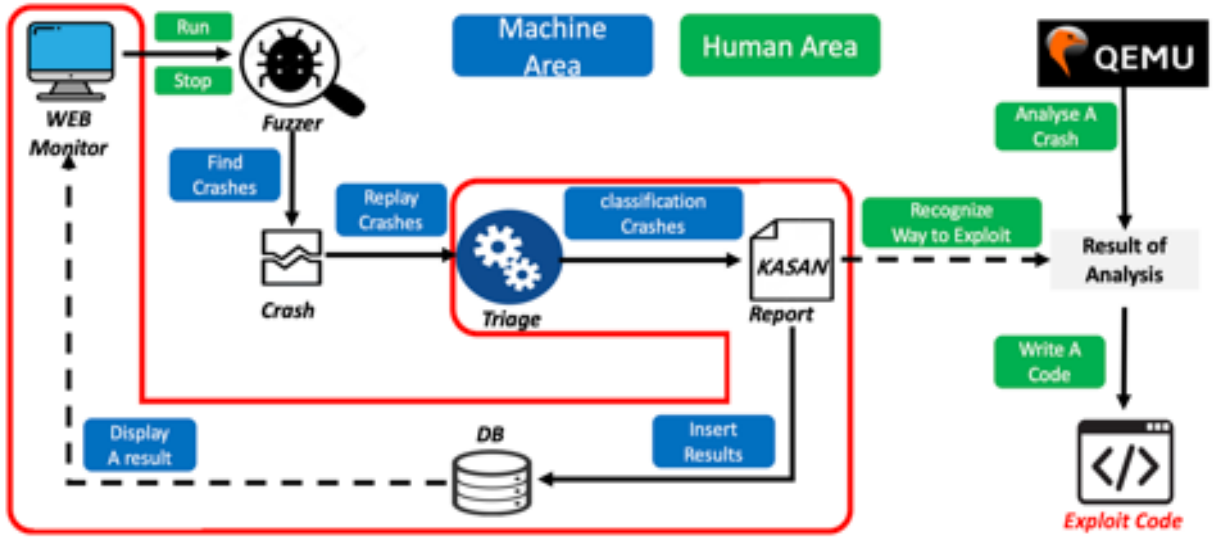


Abstract

Attack Model



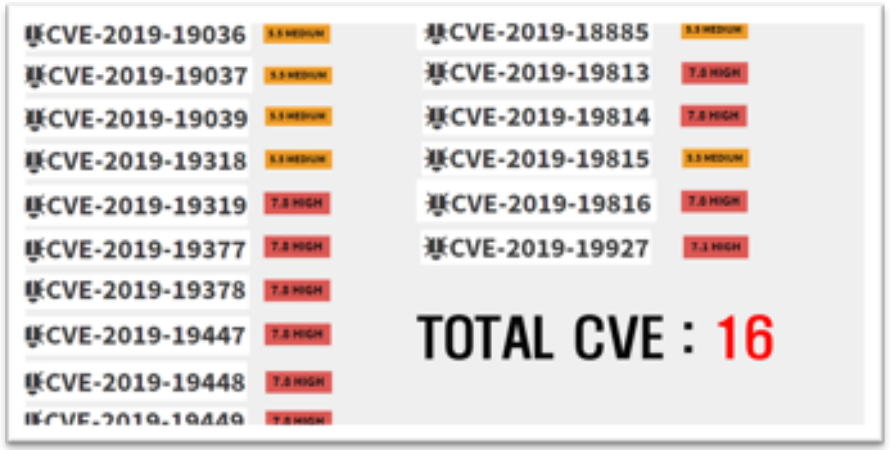
Fuzzing & Monitoring



Abstract

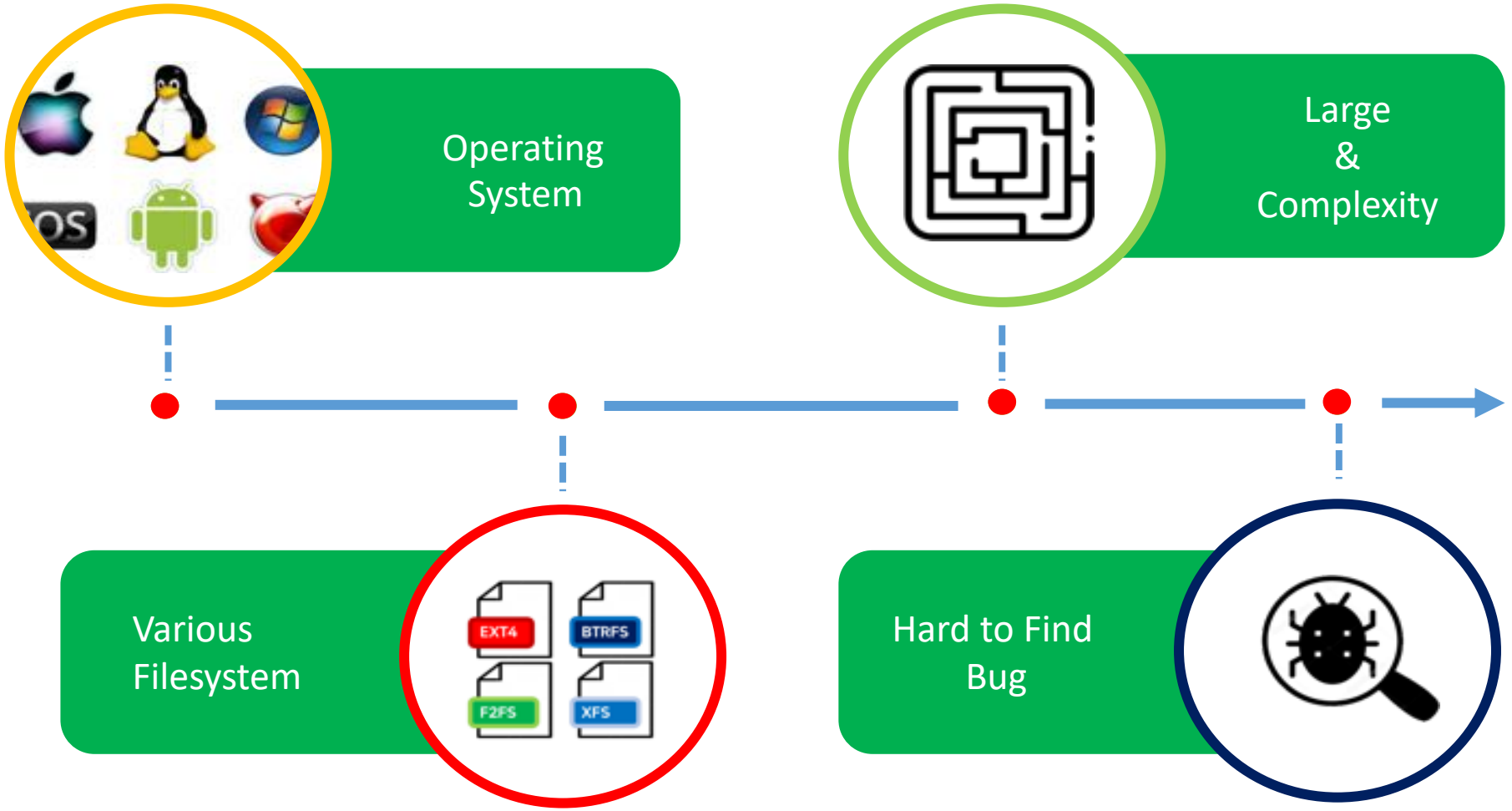

Dashboard


Arbitrary Write


CVE-2019-19036
CVE-2019-19037
CVE-2019-19039
CVE-2019-19318
CVE-2019-19319
CVE-2019-19377
CVE-2019-19378
CVE-2019-19447
CVE-2019-19448
CVE-2019-18885
CVE-2019-19813
CVE-2019-19814
CVE-2019-19815
CVE-2019-19816
CVE-2019-19927
TOTAL CVE : 16


CVE-2019-19377 Demo Video (Ubuntu 19.04 latest release)

Introduction



Introduction

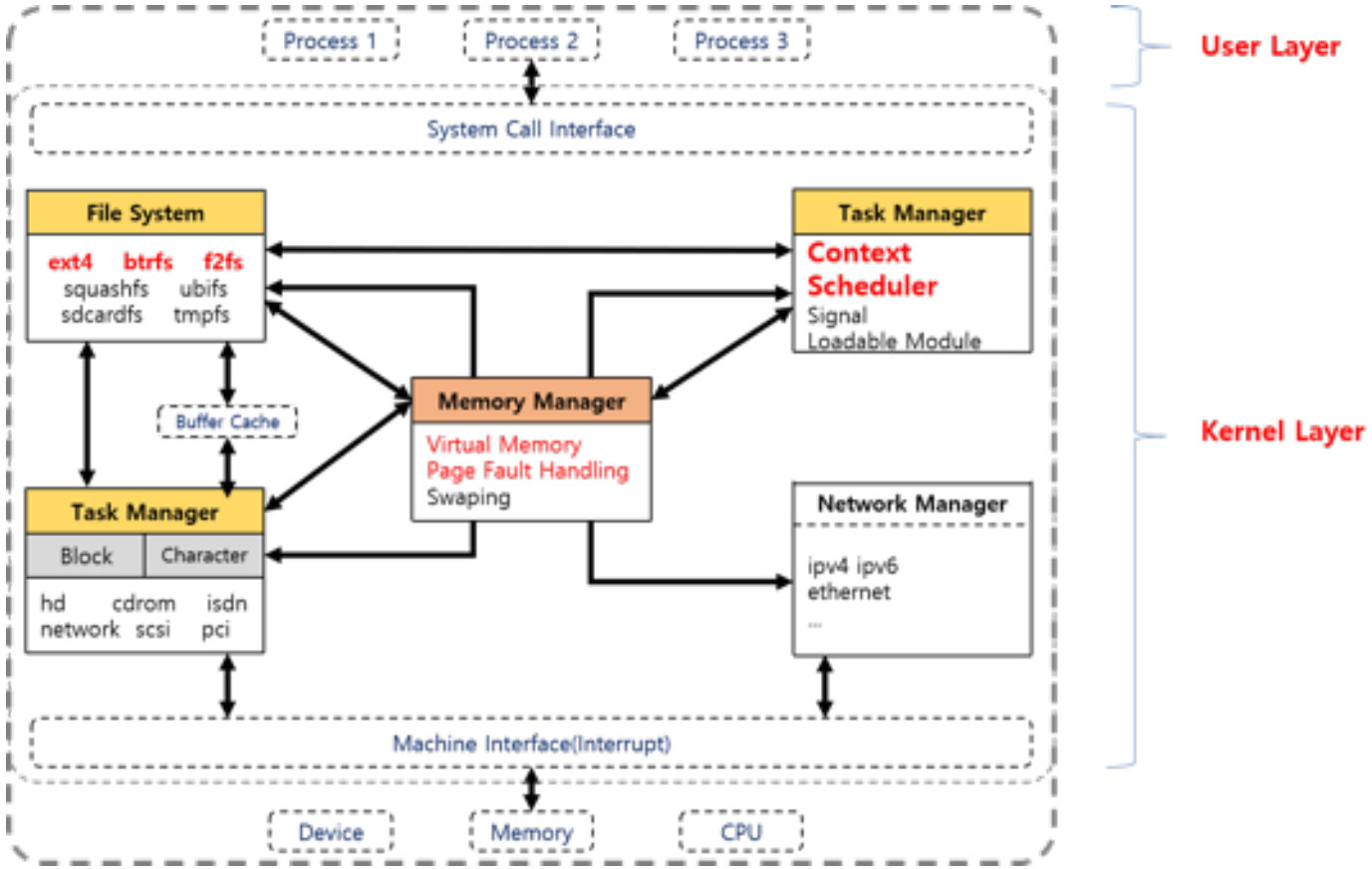
COMFLEX FILE SYSTEMS

File System	LoC	Kernel Use
EXT4	50,000	✔
BTRFS	130,000	✔
XFS	140,000	✔

File Systems are Hard to be bug-free!

Introduction

- OS file system structure

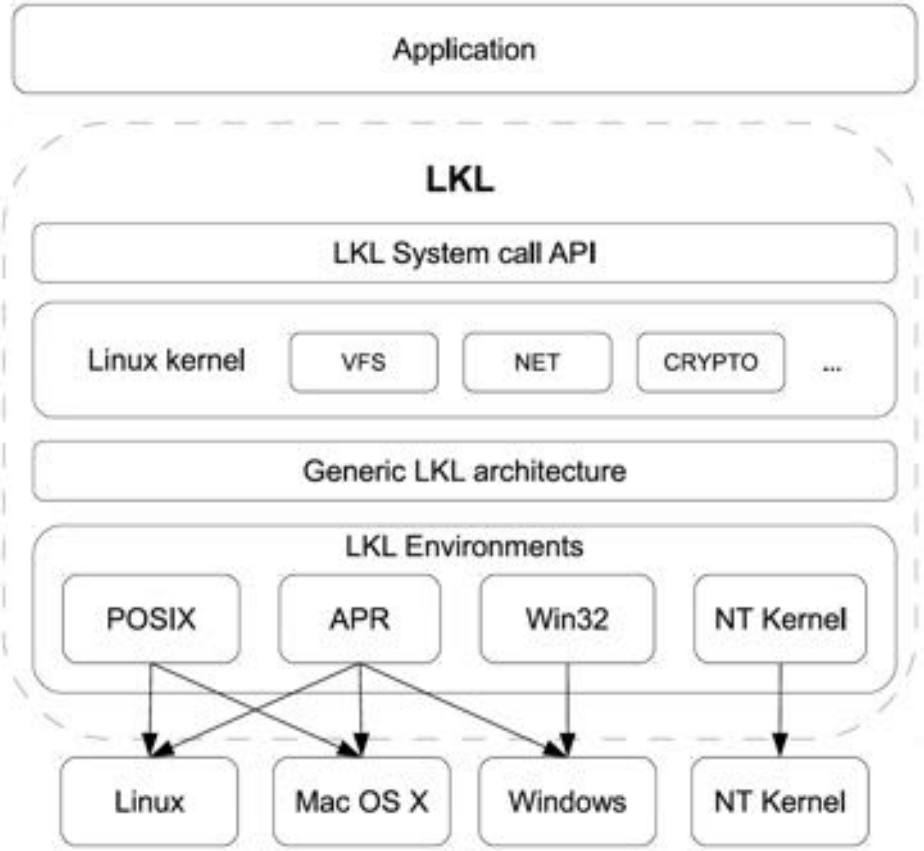


Background

- Linux Kernel Library

1. What is LKL? (IEEE 2010)

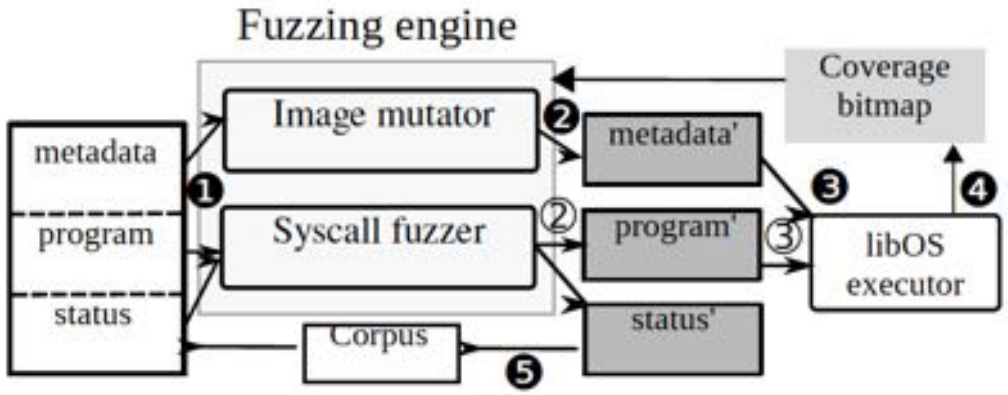
The main goal of LKL is to provide a simple and maintainable way in which applications running in environments as diverse as possible can reuse Linux kernel code



Background

- Janus file system fuzzer

1. What is Janus(IEEE 2019)



Fuzzing File Systems via
Two-Dimensional Input Space Exploration

Wen Xu Hyungon Moon† Sanidhya Kashyap Po-Ning Tseng Taesoo Kim

Georgia Institute of Technology
†*UIsan National Institute of Science and Technology*

Abstract—File systems, a basic building block of an OS, are too big and too complex to be bug free. Nevertheless, file systems rely on regular stress-testing tools and formal checkers to find bugs, which are limited due to the ever-increasing complexity of both file systems and OSes. Thus, fuzzing, proven to be an effective and a practical approach, becomes a preferable choice, as it does not need much knowledge about a target. However, three main challenges exist in fuzzing file systems: mutating a large image blob that degrades overall performance, generating image-dependent file operations, and reproducing found bugs, which is difficult for existing OS fuzzers.

Hence, we present JANUS, the first feedback-driven fuzzer that explores the two-dimensional input space of a file system, i.e., mutating metadata on a large image, while emitting image-directed file operations. In addition, JANUS relies on a library OS rather than on traditional VMs for fuzzing, which enables JANUS to load a fresh copy of the OS, thereby leading to better reproducibility of bugs. We evaluate JANUS on eight file systems and found 90 bugs in the upstream Linux kernel, 62 of which have been acknowledged. Forty-three bugs have been fixed with 32 CVEs assigned. In addition, JANUS achieves higher code coverage on all the file systems after fuzzing 12 hours, when compared with the state-of-the-art fuzzer Syzkaller for fuzzing file systems. JANUS visits 4.19× and 2.01× more code paths in Btrfs and ext4, respectively. Moreover, JANUS is able to reproduce 88–100% of the crashes, while Syzkaller fails on all of them.

I. Introduction

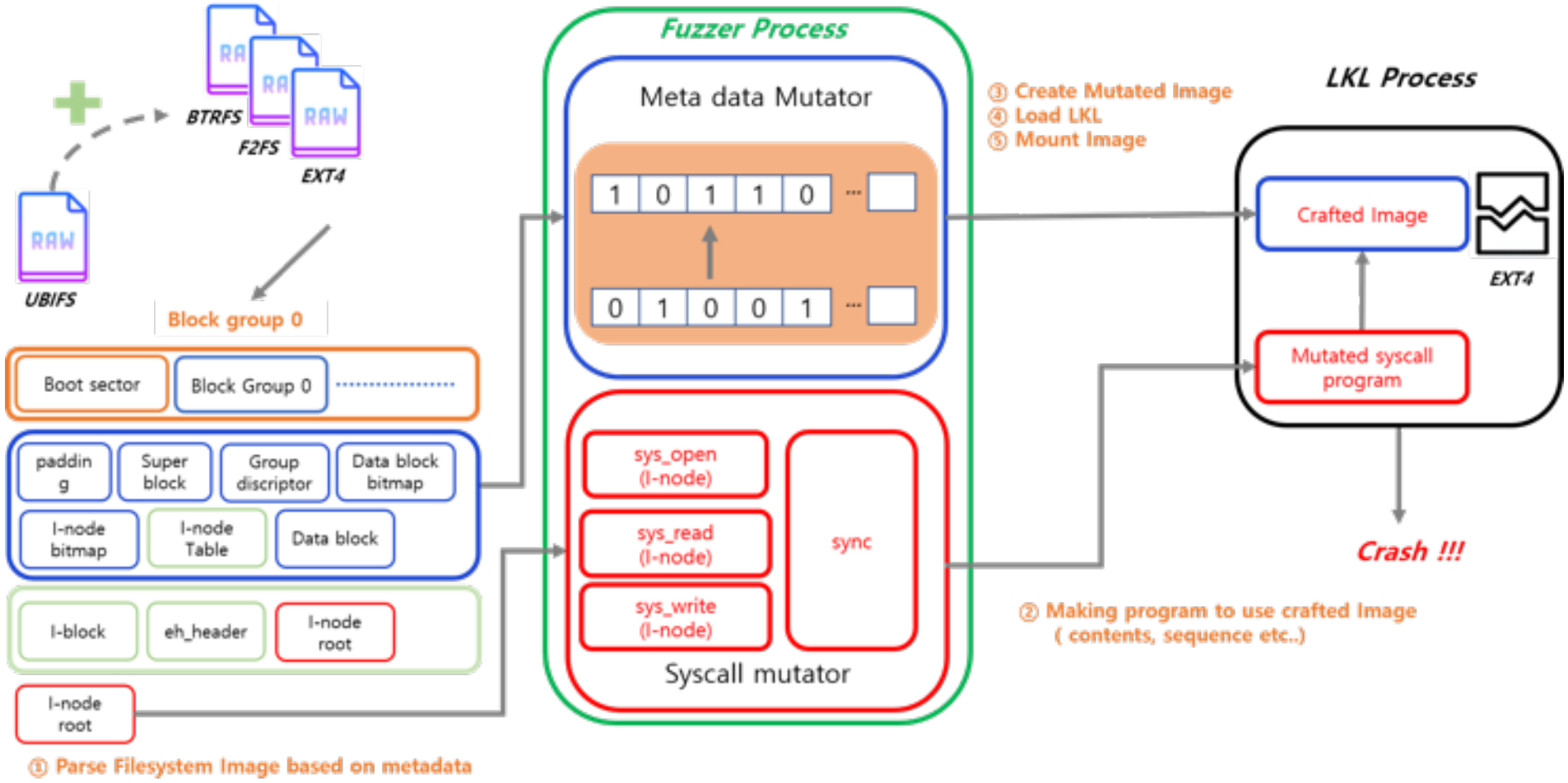
File systems are one of the most basic system services of an operating system that play an important role in managing the files of users and tolerating system crashes without losing data consistency. Currently, most of the conventional file systems, such as ext4 [8], XFS [64], Btrfs [59], and F2FS [30], run in the OS kernel. Hence, bugs in file systems cause devastating errors, such as system reboots, OS deadlock, and unrecoverable errors of the whole file system image. In addition, they also pose severe security threats. For instance, attackers exploit various file system issues by mounting a crafted disk image [28] or invoking vulnerable file system-specific operations [37] to achieve code execution or privilege escalation on victim machines. However, manually eliminating every bug in a file system that has sheer complexity is a challenge, even for an expert. For example, the latest implementation of ext4 in Linux v4.18 comprises 50K lines of code, while that of Btrfs is nearly 130K LoC. At the same time, many widely used file systems are still under active development. File system developers consistently optimize performance [72] and add new features [11, 29], meanwhile introducing new bugs [26, 27, 40].

To automatically discover these potential bugs, most file systems in development rely on the known stress-testing frameworks (xfstests [63], fsck [56, 69], Linux Test Project [62], etc.) that mostly focus on the regression of file systems with minimal integrity checks. For example, one of the bugs we found in ext4 (i.e., CVE-2018-10880) crashes the kernel by moving a critical extended attribute out of the inode structure. We trigger this bug by mounting a normal ext4 image formatted with `inline_data`, which bypasses integrity checks in both xfstests and fsck. In addition, some prior works have applied model checking [73, 74] to find file system bugs, which requires a deep understanding of both the file system and OS states. This is now impractical due to the increasing complexity of modern OSes [2, 35, 68]. On the other hand, most of the verified file systems [6, 9] are too immature to adopt in practice.

Another approach—fuzzing—is gaining traction. Fuzzing not only requires minimal knowledge about the target software, but also is an *effective* and a *practical* approach that has found thousands of vulnerabilities [15, 18, 34, 76]. Hence, fuzzing is a viable approach to automatically discover bugs in a wide range of file systems (e.g., 54 in the Linux kernel). However, unlike other ordinary targets, fuzzing file systems is dependent on two inputs: a mounted disk image and a sequence of file operations (i.e., system calls) that are executed on the mounted image. Existing fuzzers either focus on mutating images as ordinary binary inputs [21, 48, 57, 61] or generating random sets of file operation-specific system calls [20, 25, 46]. Unfortunately, they all fail to efficiently and comprehensively test file systems because of the following three challenges.

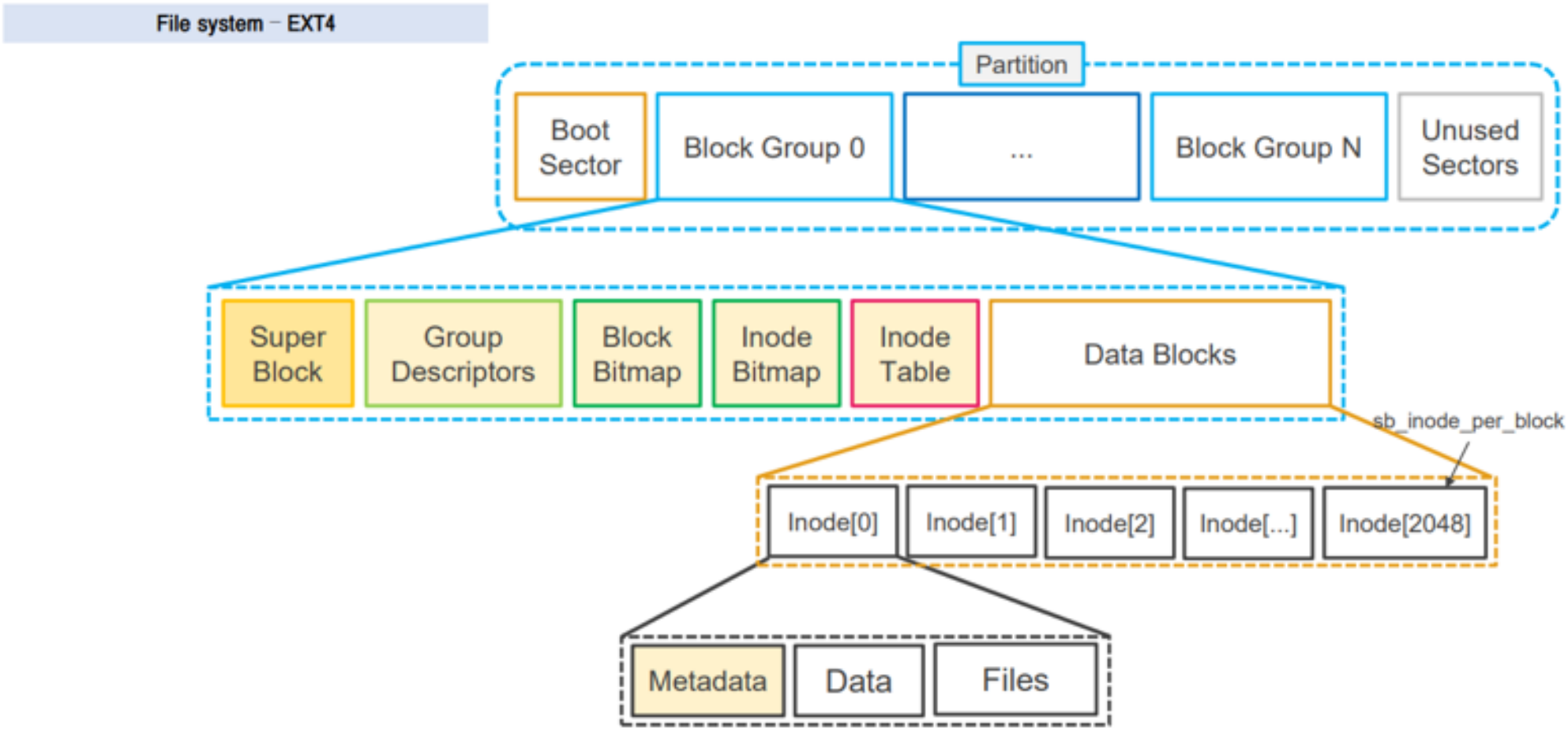
First, a disk image is a large binary blob that is structured but complicated, and the *minimum* size can be almost 100× larger than the *maximum* preferred size of general fuzzers [76], which dramatically degrades the fuzzing throughput [21, 57, 61] due to the heavy I/O involved in mutating images. Another issue related to blob mutation is that existing fuzzers [20, 48] mutate only non-zero chunks in an image. This approach is unsound because these fuzzers do not exploit the properties of structured data, i.e., file system layout, in which mutating metadata blocks is more effective than mutating data blocks. In addition, without any knowledge about the file system layout, existing fuzzers also fail to fix any metadata checksum after corrupting metadata blocks. The second challenge is that file operations are context-aware workloads, i.e., a dependence exists between an image and the file operations executed on

Background

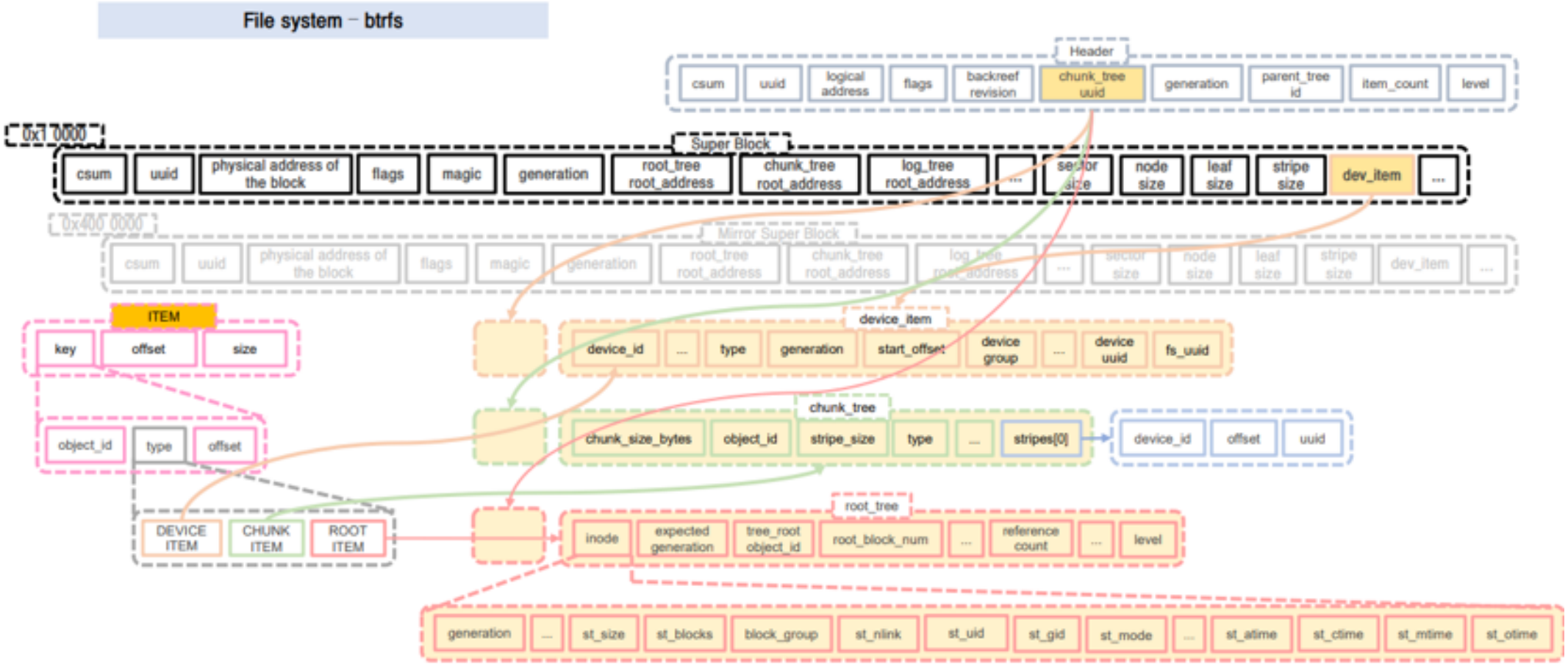

RA
RA
RAW
BTRFS
F2FS
EXT4
RAW
UBIFS
Block group 0
Boot sector
Block Group 0
paddin g
Super block
Group discriptor
Data block bitmap
I-node bitmap
I-node Table
Data block
I-block
eh_header
I-node root
I-node root
① Parse Filesystem Image based on metadata
Fuzzer Process
Meta data Mutator
1 0 1 1 0 ...
0 1 0 0 1 ...
sys_open (I-node)
sys_read (I-node)
sys_write (I-node)
sync
Syscall mutator
③ Create Mutated Image
④ Load LKL
⑤ Mount Image
LKL Process
Crafted Image
EXT4
Mutated syscall program
Crash !!!
② Making program to use crafted Image
(contents, sequence etc..)

Background

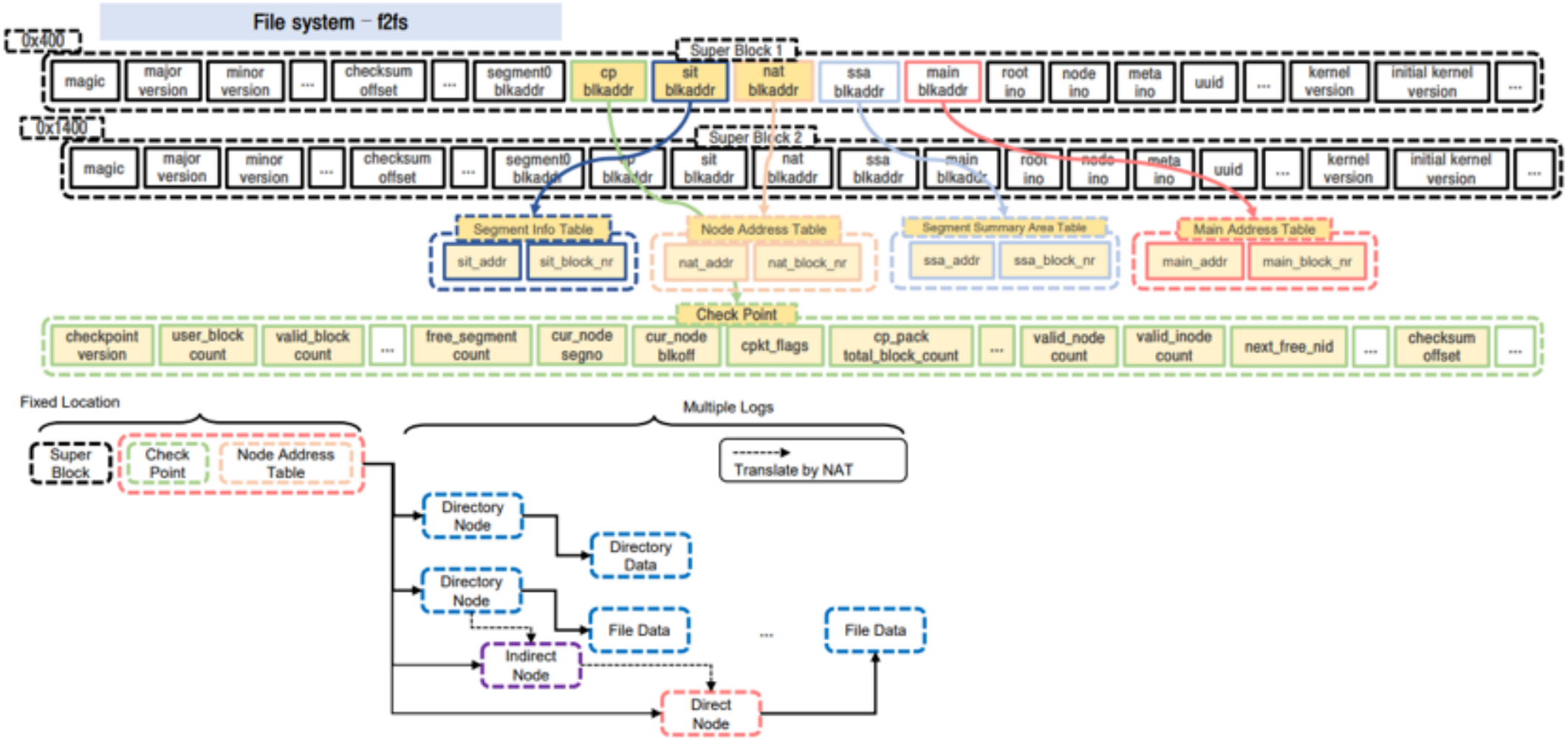
Background


File system – EXT4
Partition
Boot Sector
Block Group 0
...
Block Group N
Unused Sectors
Super Block
Group Descriptors
Block Bitmap
Inode Bitmap
Inode Table
Data Blocks
sb_inode_per_block
Inode[0]
Inode[1]
Inode[2]
Inode[...]
Inode[2048]
Metadata
Data
Files

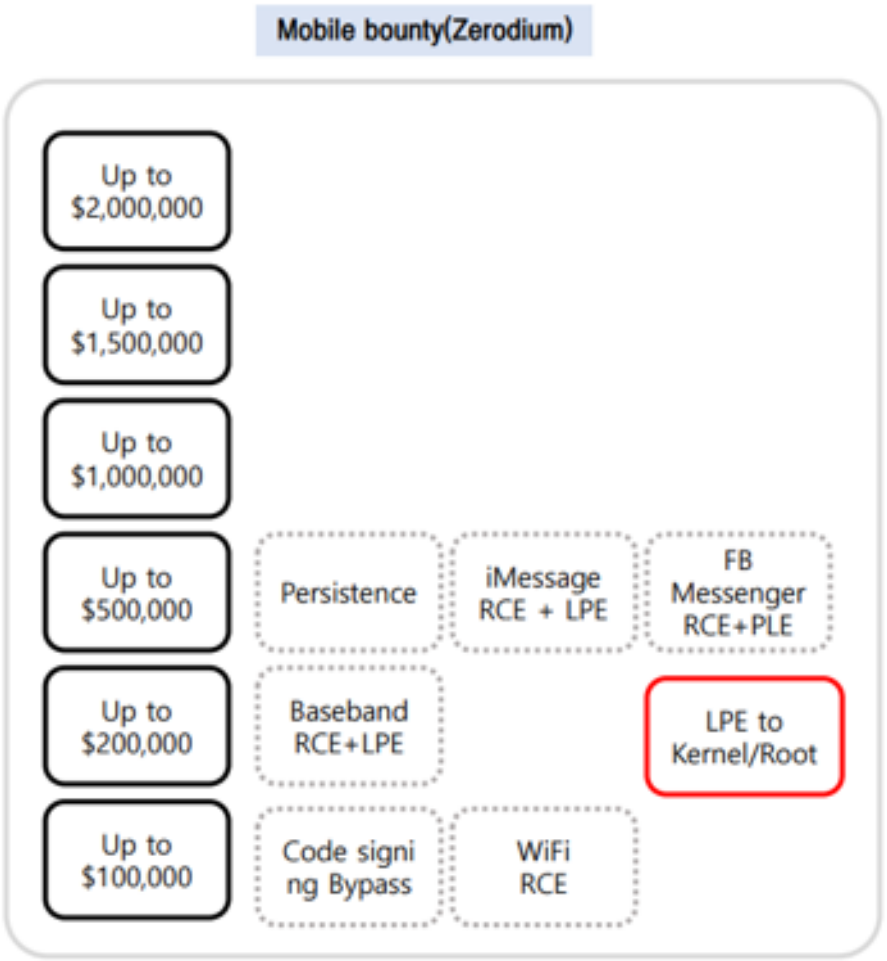
Background

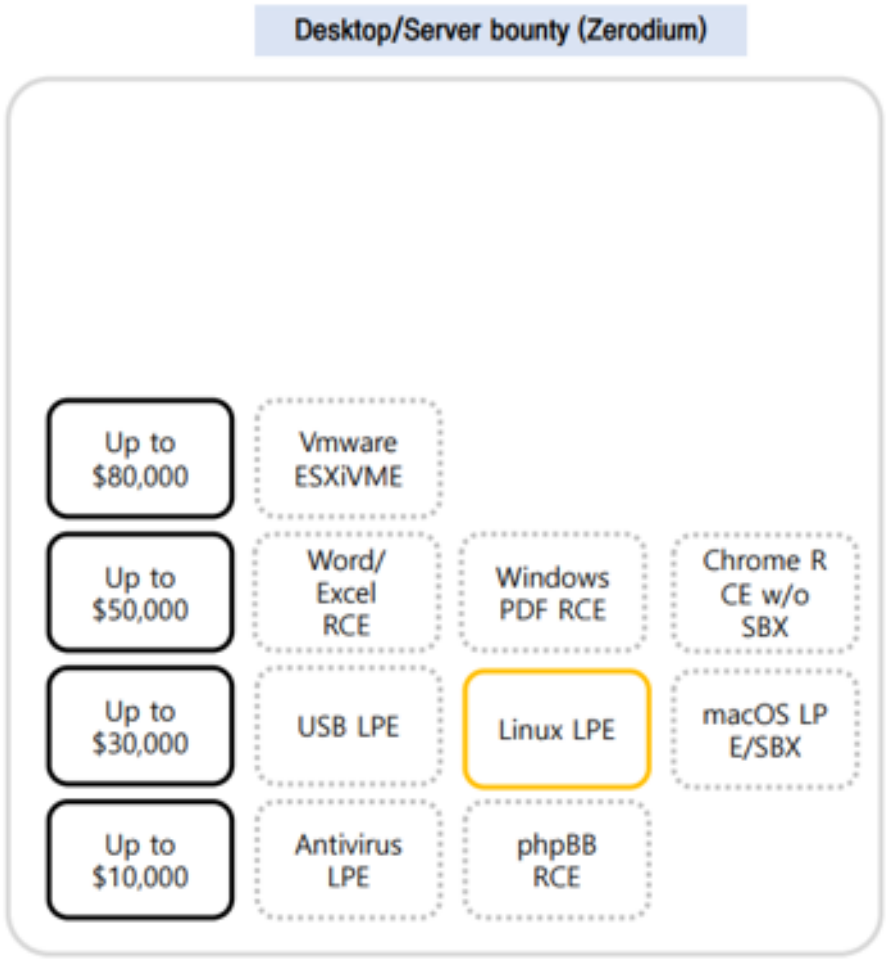

File system – btrfs
Header
csum
uuid
logical address
flags
backreef revision
chunk_tree uuid
generation
parent_tree id
item_count
level
0x1 0000
Super Block
csum
uuid
physical address of the block
flags
magic
generation
root_tree root_address
chunk_tree root_address
log_tree root_address
...
sector size
node size
leaf size
stripe size
dev_item
...
0x400 0000
Mirror Super Block 1
ITEM
key
offset
size
object_id
type
offset
DEVICE ITEM
CHUNK ITEM
ROOT ITEM
device_item
device_id
...
type
generation
start_offset
device group
...
device uuid
fs_uuid
chunk_tree
chunk_size_bytes
object_id
stripe_size
type
...
stripes[0]
device_id
offset
uuid
root_tree
inode
expected generation
tree_root object_id
root_block_num
...
reference count
...
level
generation
...
st_size
st_blocks
block_group
st_nlink
st_uid
st_gid
st_mode
...
st_atime
st_ctime
st_mtime
st_otime

Background

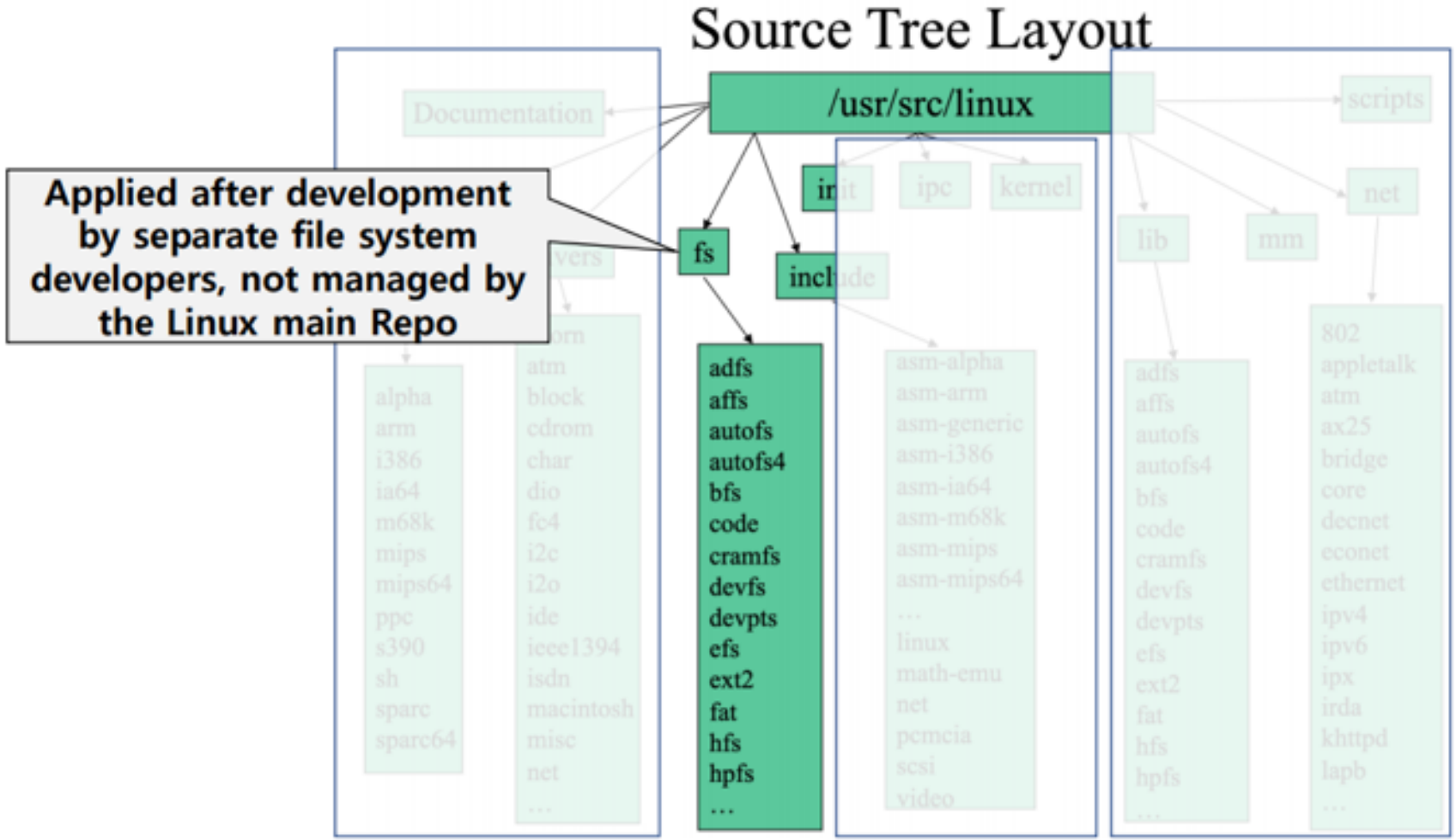

File system – f2fs
0x400
Super Block 1
magic
major version
minor version
checksum offset
segment0 blkaddr
cp blkaddr
sit blkaddr
nat blkaddr
ssa blkaddr
main blkaddr
root ino
node ino
meta ino
uuid
kernel version
initial kernel version
0x1400
Super Block 2
Segment Info Table
sit_addr
sit_block_nr
Node Address Table
nat_addr
nat_block_nr
Segment Summary Area Table
ssa_addr
ssa_block_nr
Main Address Table
main_addr
main_block_nr
Check Point
checkpoint version
user_block count
valid_block count
free_segment count
cur_node segno
cur_node blkoff
ckpt_flags
cp_pack total_block_count
valid_node count
valid_inode count
next_free_nid
checksum offset
Fixed Location
Multiple Logs
Super Block
Check Point
Node Address Table
Translate by NAT
Directory Node
Directory Data
Directory Node
File Data
File Data
Indirect Node
Direct Node

Background


Mobile bounty(Zerodium)
Up to $2,000,000
Up to $1,500,000
Up to $1,000,000
Up to $500,000
Up to $200,000
Up to $100,000
Persistence
iMessage RCE + LPE
FB Messenger RCE+PLE
Baseband RCE+LPE
LPE to Kernel/Root
Code signi ng Bypass
WiFi RCE


Desktop/Server bounty (Zerodium)
Up to $80,000
Up to $50,000
Up to $30,000
Up to $10,000
Vmware ESXiVME
Word/ Excel RCE
Windows PDF RCE
Chrome R CE w/o SBX
USB LPE
Linux LPE
macOS LP E/SBX
Antivirus LPE
phpBB RCE

Background


Source Tree Layout
/usr/src/linux
Documentation
scripts
ipc
kernel
net
lib
mm
fs
Applied after development by separate file system developers, not managed by the Linux main Repo
adfs
affs
autofs
autofs4
bfs
code
cramfs
devfs
devpts
efs
ext2
fat
hfs
hpfs
…

Evaluated Various OS

File System / OS	XFS	F2FS	EXT4	BTRFS	TOTAL
Ubuntu OS – V 18.04	1	2	–	–	3 Total
Gooroom OS – Dev	3	1	–	–	4 Total
HamoniKR OS – V 1.4	6	4	11	4	25 Total
Red Star OS –V 3.0	–	–	2	–	2 Total

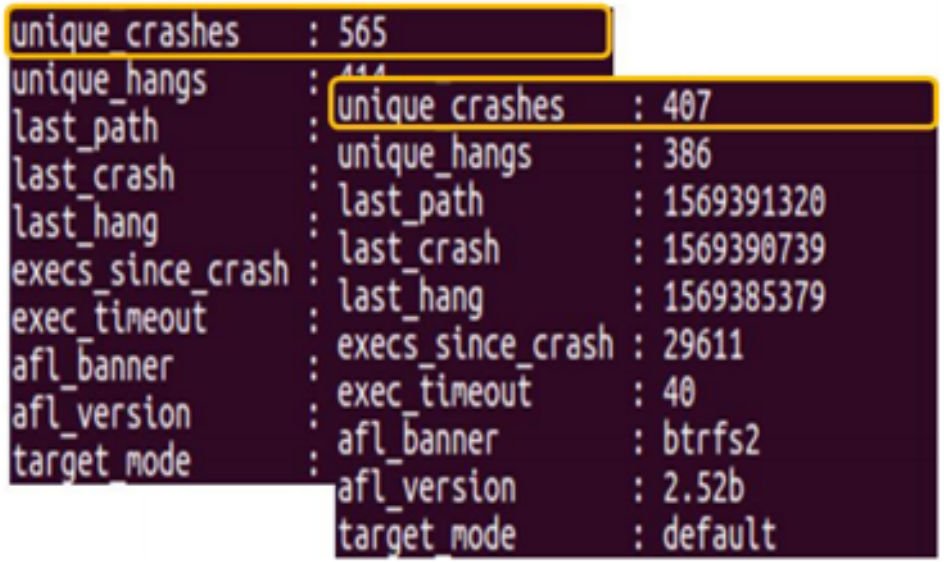
Evaluated Various OS DEMO(Smart TV)

Triage & Monitor

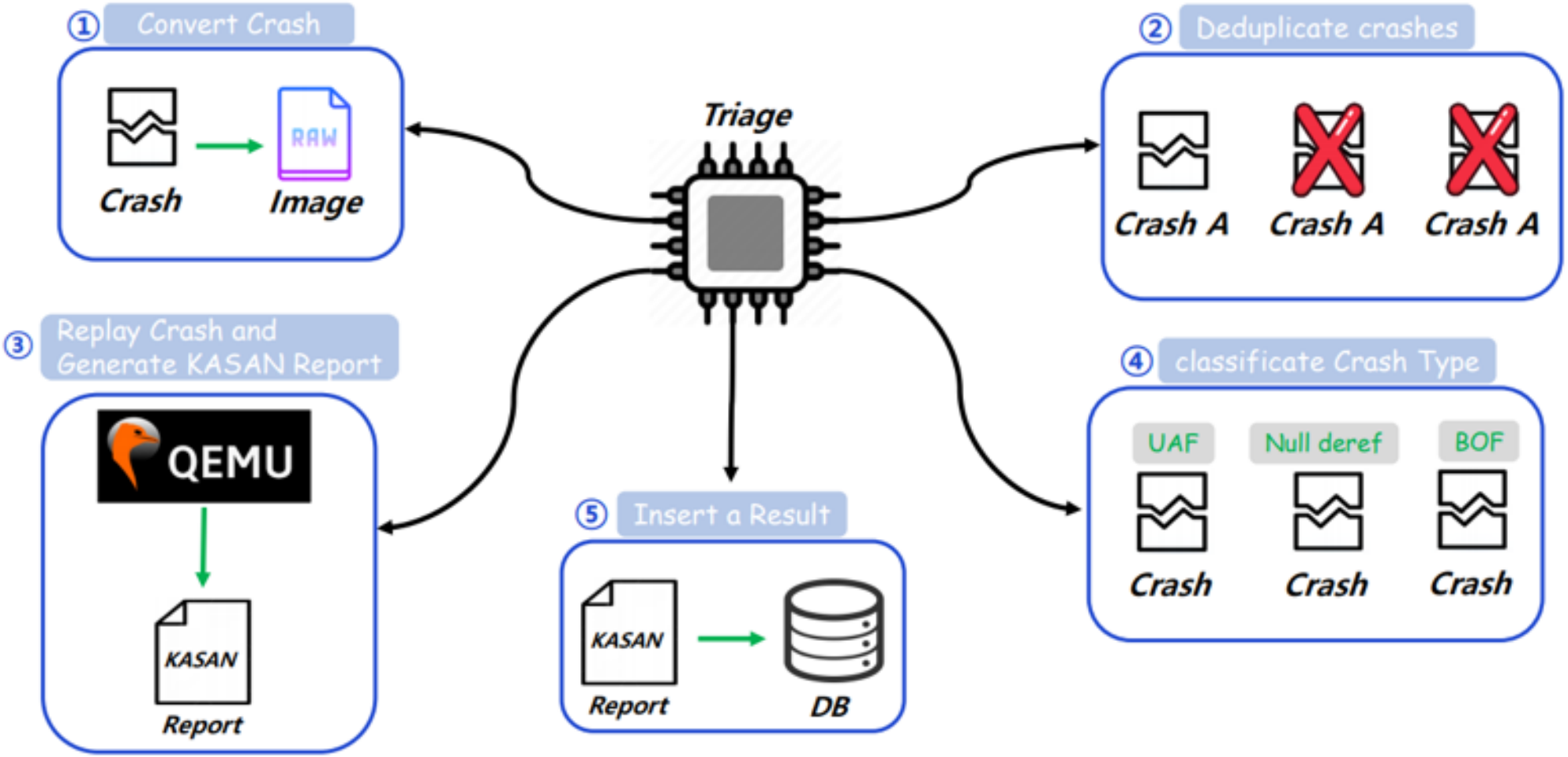
Problems with JANUS

- Since the LKL internal source modification is used to optimize and configure fuzzing, the source modification should be performed whenever the lkl version is changed.

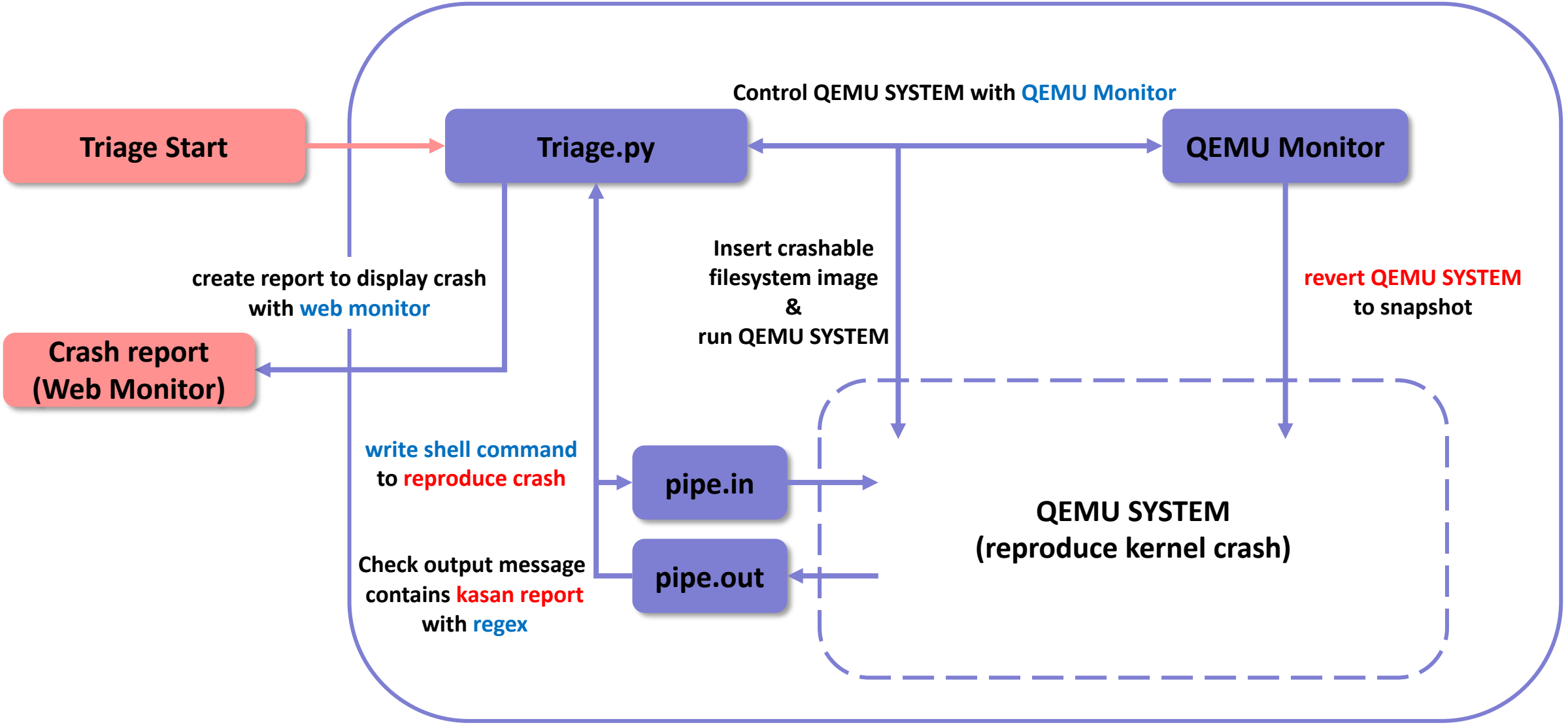
- As a result of fuzzing, too many duplicate crashes were classified as unique crashes.



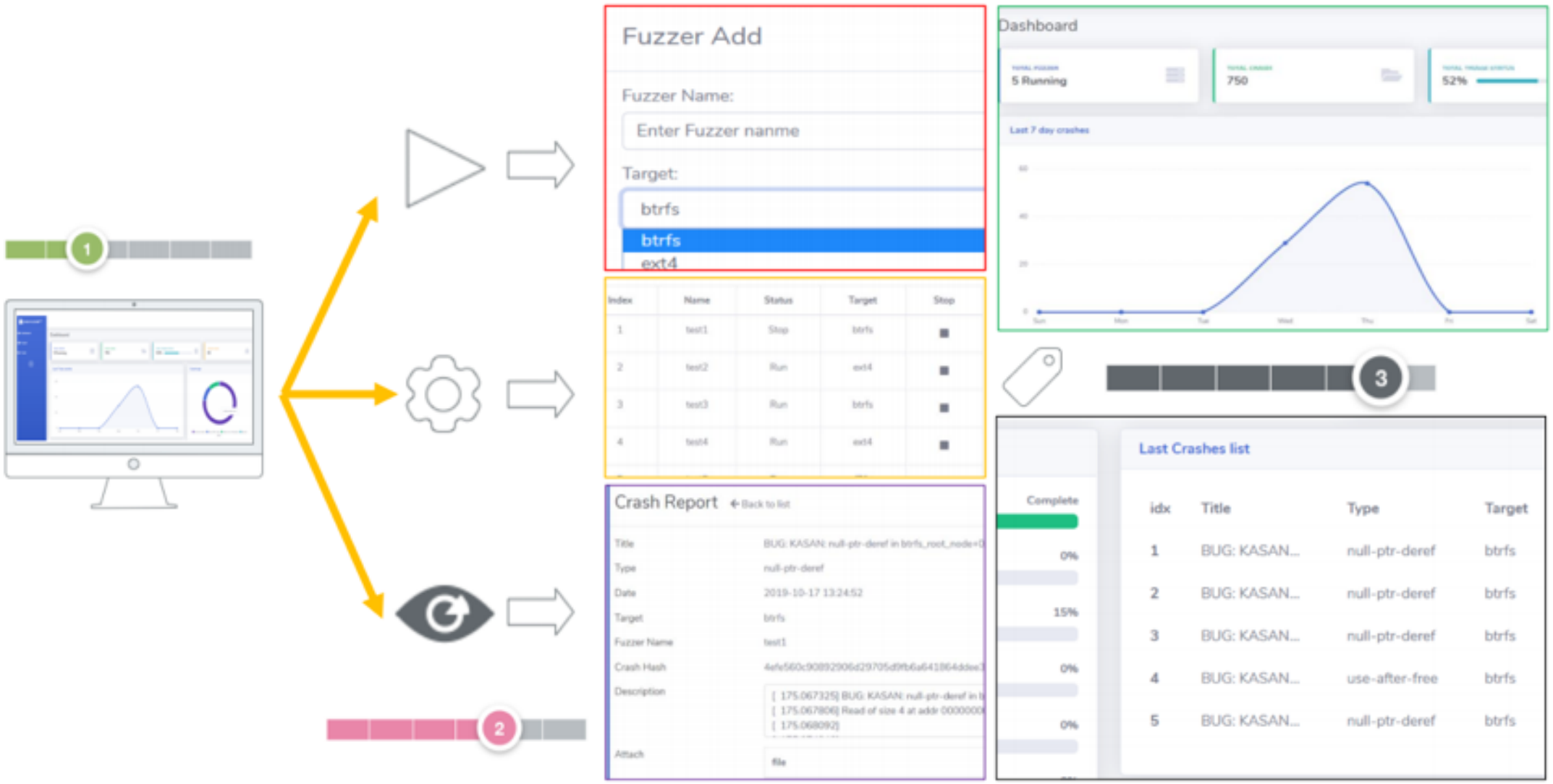
Triage & Monitor



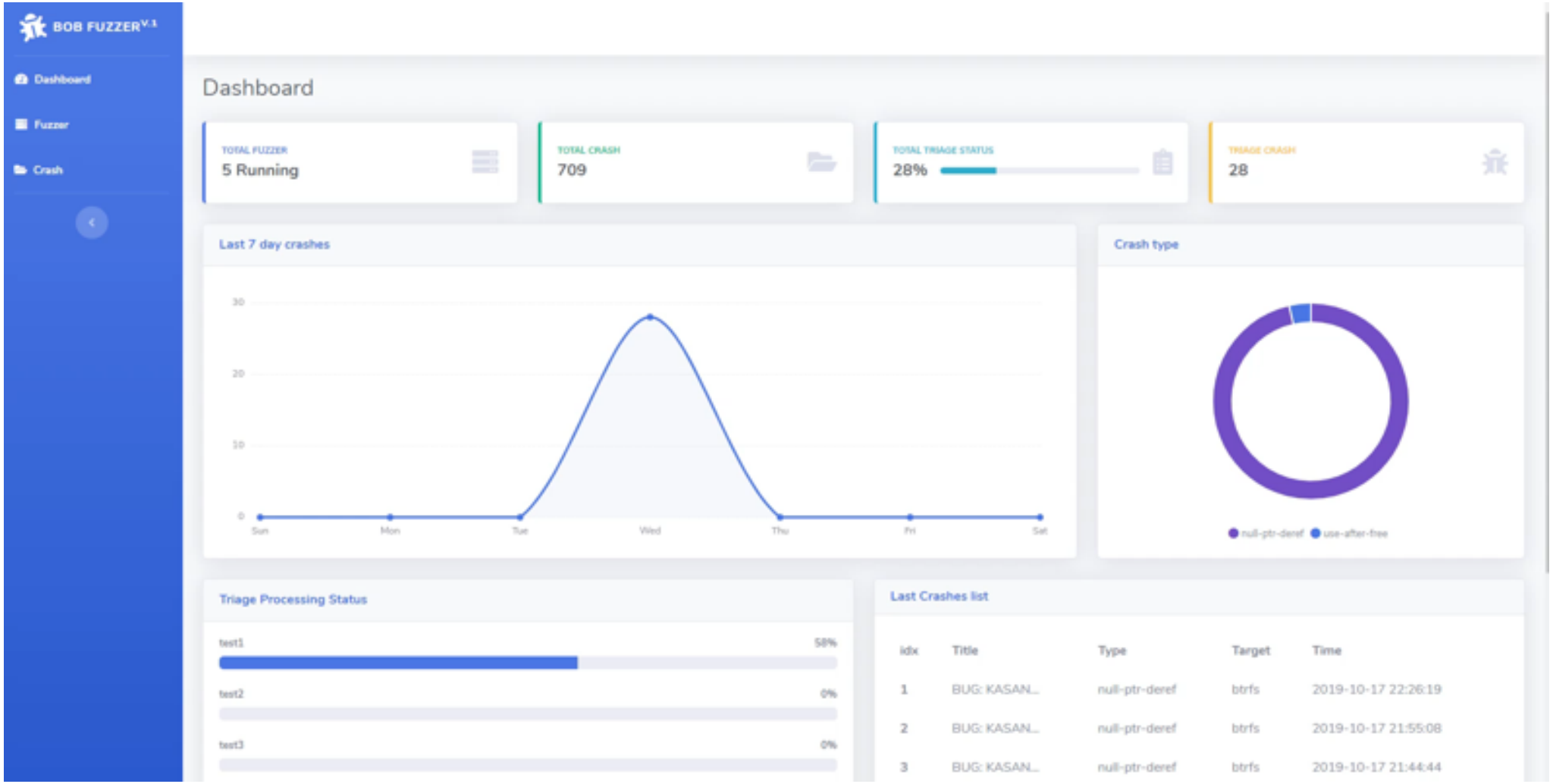
Triage & Monitor



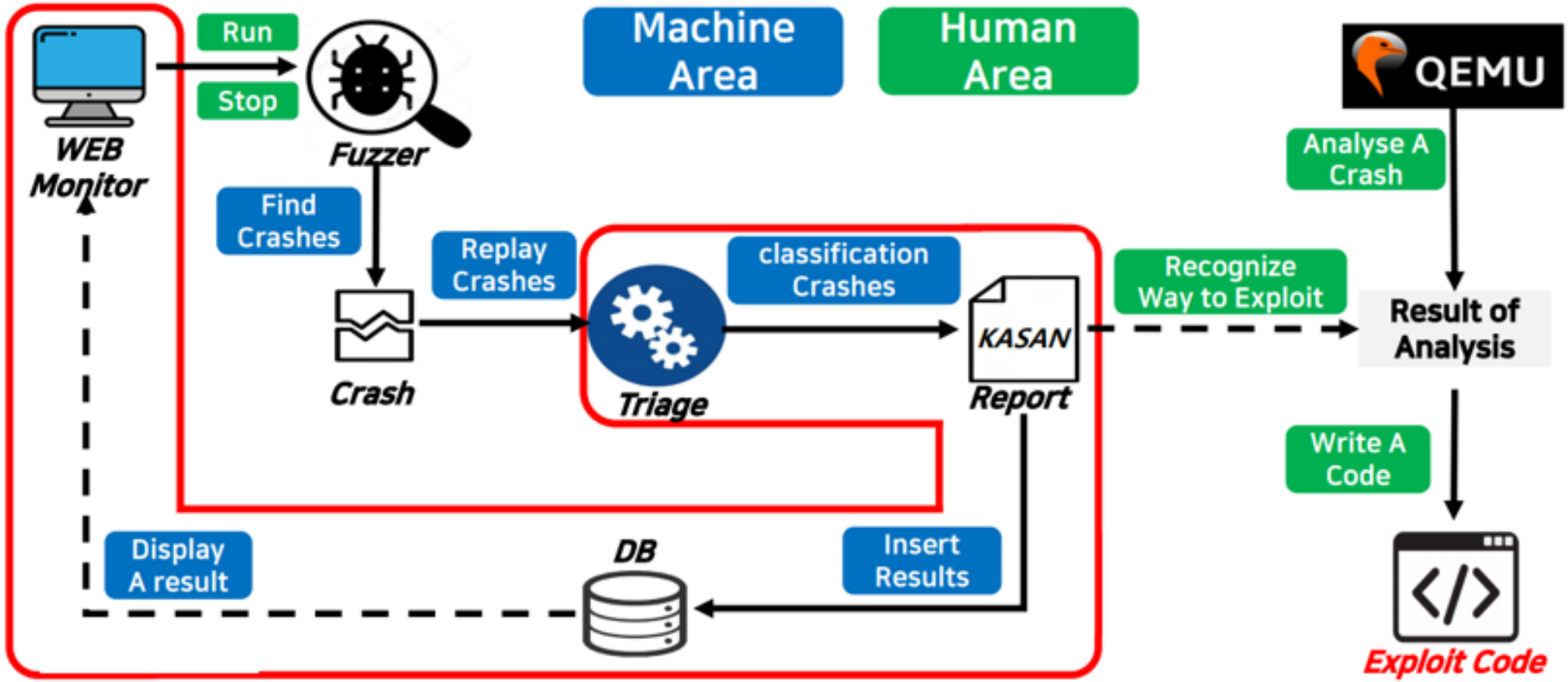
Triage & Monitor



Triage & Monitor



Triage & Monitor



Crash Analysis

- What we found?

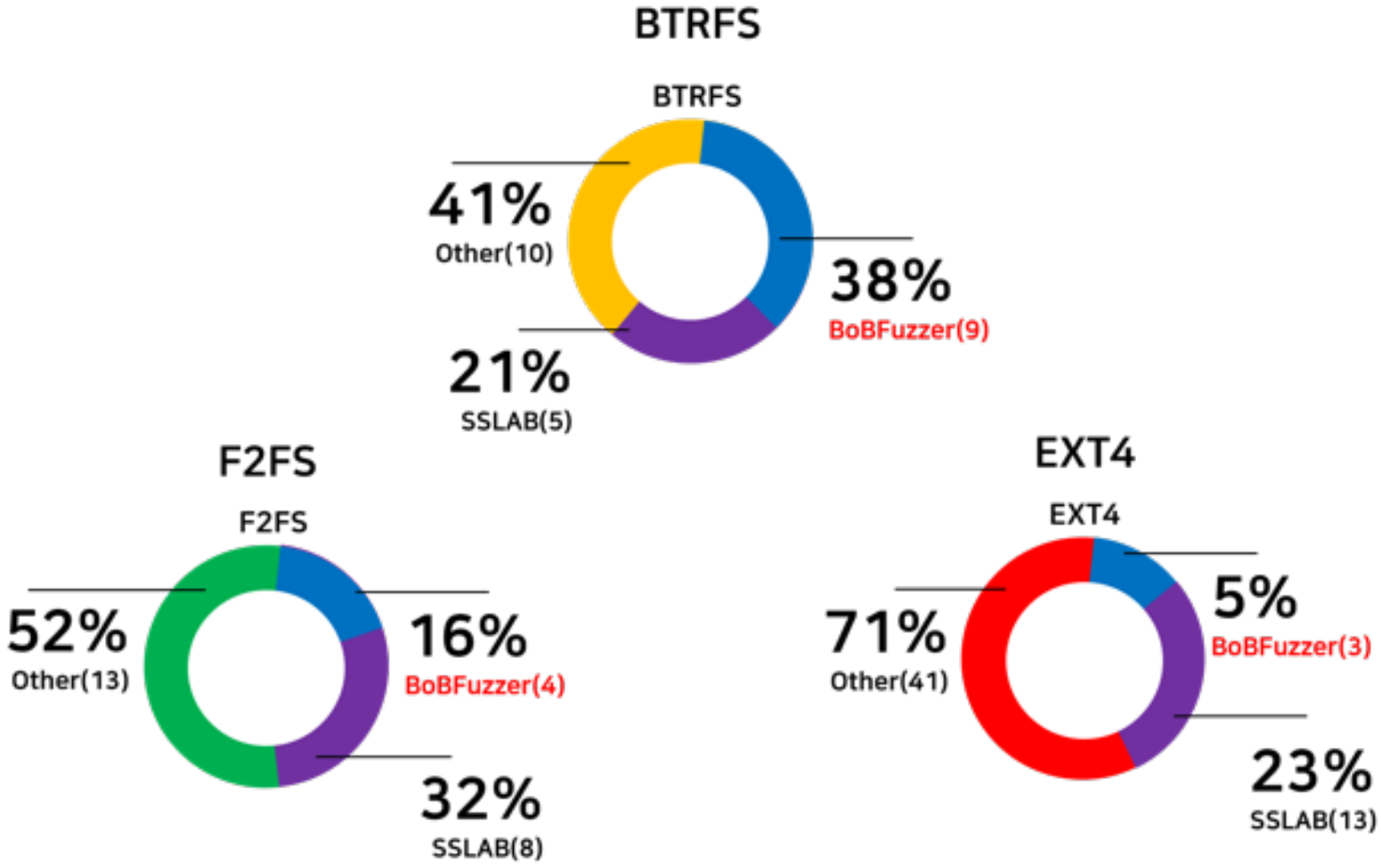
CVE-2019-19036	5.5 MEDIUM
CVE-2019-19037	5.5 MEDIUM
CVE-2019-19039	5.5 MEDIUM
CVE-2019-19318	5.5 MEDIUM
CVE-2019-19319	7.8 HIGH
CVE-2019-19377	7.8 HIGH
CVE-2019-19378	7.8 HIGH
CVE-2019-19447	7.8 HIGH
CVE-2019-19448	7.8 HIGH
CVE-2019-19449	7.8 HIGH
CVE-2019-18885	5.5 MEDIUM
CVE-2019-19813	7.8 HIGH
CVE-2019-19814	7.8 HIGH
CVE-2019-19815	5.5 MEDIUM
CVE-2019-19816	7.8 HIGH
CVE-2019-19927	7.1 HIGH

TOTAL CVE : 16

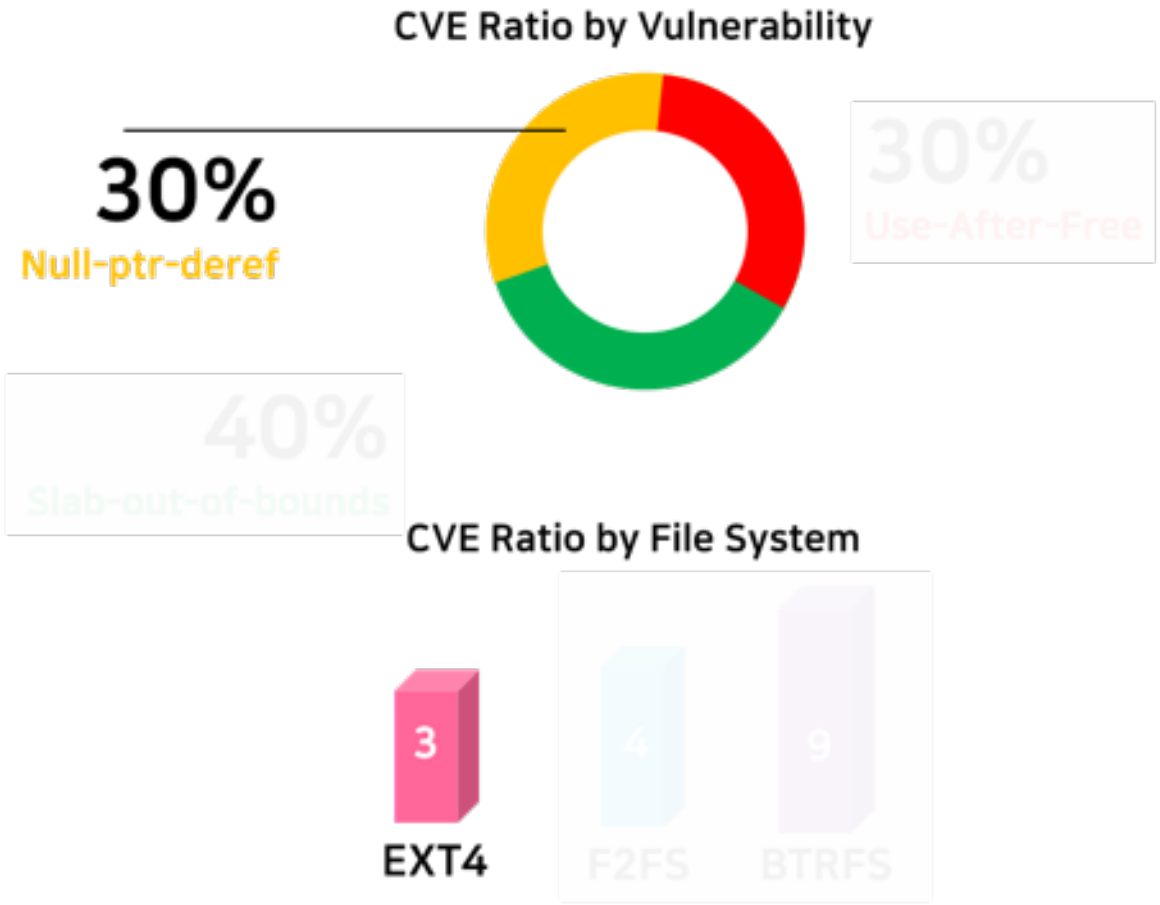
0-Day crashes analysis

Crash Analysis

- Percentage of CVE by Filesystems



Crash Analysis



Null pointer dereference

CVEs

- CVE-2019-19037
- CVE-2019-19036

Crash Analysis

- **Null Pointer dereference - CVE-2019-19037**

Cause

The function ext4_empty_dir is a routine that verifies that the specified directory is empty.

For example, if you run the rmdir command, this function is called when checking for internal files. At this point, **'buffer head *bh'** is used to map a single block within the page, a vulnerability that is caused by returning null as the return value of ext4_read_dirblock.

```
bool ext4_empty_dir(struct inode *inode)
{
        unsigned int offset;
        struct buffer_head *bh;
        struct ext4_dir_entry_2 *de, *de1;
        struct super_block *sb;
```

```
        de = ext4_next_entry(de1, sb->s_blocksize);
        while (offset < inode->i_size) {
                if ((void *) de >= (void *) (bh->b_data+sb->s_blocksize)) {
                        unsigned int lblock;
                        brelse(bh);
                        lblock = offset >> EXT4_BLOCK_SIZE_BITS(sb);
                        bh = ext4_read_dirblock(inode, lblock, EITHER);
                        if (bh == NULL) {
                                offset += sb->s_blocksize;
                                continue;
                        }
```

Crash Analysis

- **Null Pointer dereference - CVE-2019-19037**

Cause

The function ext4_empty_dir is a routine that verifies that the specified directory is empty.

For example, if you run the rmdir command, this function is called when checking for internal files. At this point, **'buffer head *bh'** is used to map a single block within the page, a vulnerability that is caused by returning null as the return value of ext4_read_dirblock.

```
bool ext4_empty_dir(struct inode *inode)
{
        unsigned int offset;
        struct buffer_head *bh;
        struct ext4_dir_entry_2 *de, *de1;
        struct super_block *sb;
```

```
        de = ext4_next_entry(de1, sb->s_blocksize);
        while (offset < inode->i_size) {
                if ((void *) de >= (void *) (bh->b_data+sb->s_blocksize)) {
                        unsigned int lblock;
                        brelse(bh);
                        lblock = offset >> EXT4_BLOCK_SIZE_BITS(sb);
                        bh = ext4_read_dirblock(inode, lblock, EITHER);
                        if (bh == NULL) {
                                offset += sb->s_blocksize;
                                continue;
                        }
```

Crash Analysis

- Null Pointer dereference - CVE-2019-19037

Exploitability

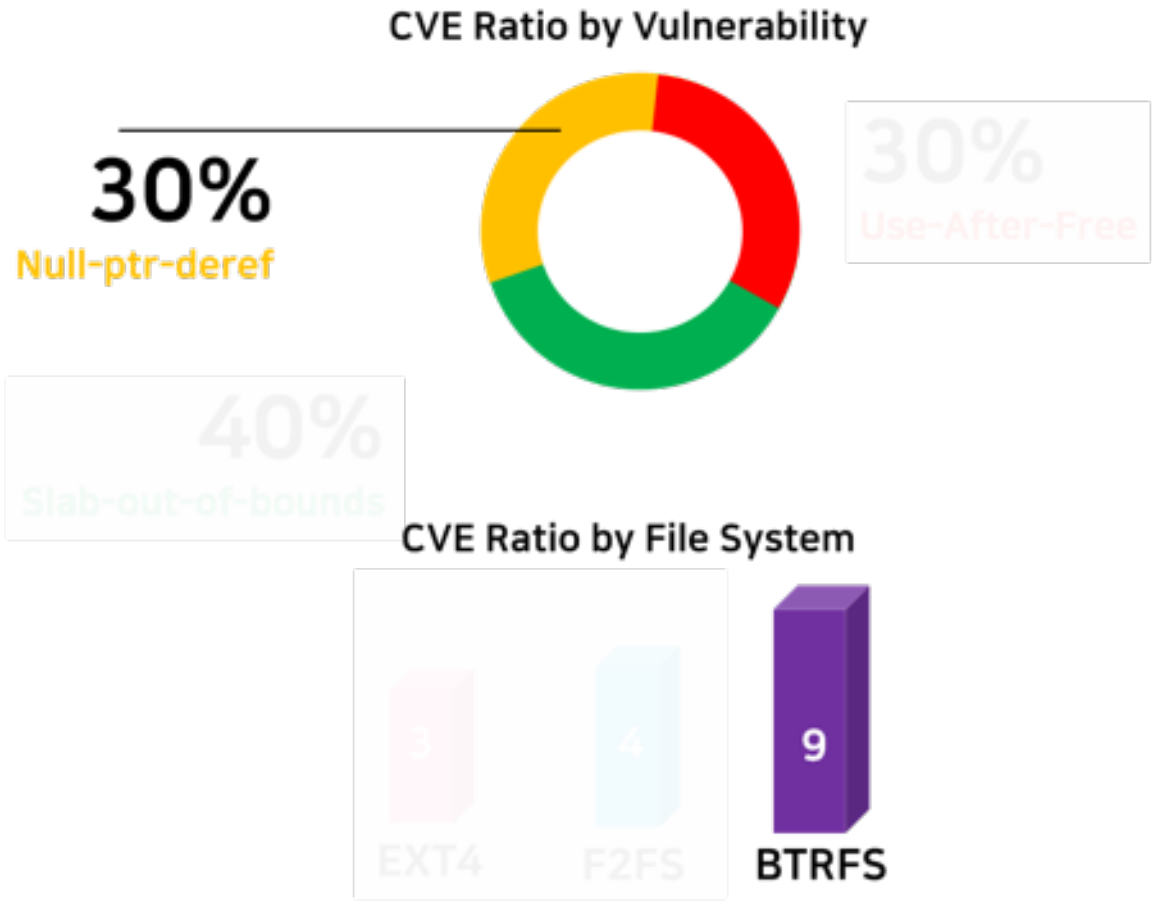
NULL pointer reference is not significant if you assume an attacker who can insert USB or mount the file system in a kernel.

```
static int ext4_empty_dir(struct inode *inode)
{
        unsigned int offset;
        struct buffer_head *bh;
        struct ext4_dir_entry_2 *de, *de1;
        struct super_block *sb;
```

```
        de = ext4_next_entry(de1, sb->s_blocksize);
        while (offset < inode->i_size) {
                if ((void *) de >= (void *) (bh->b_data+sb->s_blocksize)) {
                        unsigned int lblock;
                        brelse(bh);
                        lblock = offset >> EXT4_BLOCK_SIZE_BITS(sb);
                        bh = ext4_read_dirblock(inode, lblock, EITHER);
                        if (bh == NULL) {
                                offset += sb->s_blocksize;
                                continue;
                        }
```

** Internal exploitable score: 1

Crash Analysis



Null pointer dereference

CVEs

- CVE-2019-19037
- CVE-2019-19036

Crash Analysis

- **Null Pointer dereference - CVE-2019-19036**

Cause

Btrfs has buffers that are used to track block-meta-data in memory called extent_buffer.

For this purpose, the function btrfs_root_node() does not perform a null check in the function that obtains root-node block information, resulting in a null pointer dereference vulnerability.

```
struct extent_buffer *btrfs_root_node(struct btrfs_root *root)
{
        struct extent_buffer *eb;

        while (1) {
                rcu_read_lock();
[1]             eb = rcu_dereference(root->node);

                /*
                 * RCU really hurts here, we could free up the root node becaus
                 * it was COWed but we may not get the new root node yet so do
                 * the inc_not_zero dance and if it doesn't work then
                 * synchronize_rcu and try again.
                 */
[2]             if (atomic_inc_not_zero(&eb->refs)) {
                        rcu_read_aunlock();
                        break;
                }
                rcu_read_unlock();
                synchronize_rcu();
        }
        return eb;
}
```

Crash Analysis

- **Null Pointer dereference - CVE-2019-19036**

Cause

Btrfs has buffers that are used to track block-meta-data in memory called extent_buffer.

For this purpose, the function btrfs_root_node() does not perform a null check in the function that obtains root-node block information, resulting in a null pointer dereference vulnerability.

```
struct extent_buffer *btrfs_root_node(struct btrfs_root *root)
{
        struct extent_buffer *eb;

        while (1) {
                rcu_read_lock();
[1]             eb = rcu_dereference(root->node);

                /*
                 * RCU really hurts here, we could free up the root node becaus
                 * it was COWed but we may not get the new root node yet so do
                 * the inc_not_zero dance and if it doesn't work then
                 * synchronize_rcu and try again.
                 */
[2]             if (atomic_inc_not_zero(&eb->refs)) {
                        rcu_read_aunlock();
                        break;
                }
                rcu_read_unlock();
                synchronize_rcu();
        }
        return eb;
}
```

Crash Analysis

- **Null Pointer dereference - CVE-2019-19036**

Cause

Btrfs has buffers that are used to track block-meta-data in memory called extent_buffer.

For this purpose, the function btrfs_root_node() does not perform a null check in the function that obtains root-node block information, resulting in a null pointer dereference vulnerability.

```
struct extent_buffer *btrfs_root_node(struct btrfs_root *root)
{
        struct extent_buffer *eb;

        while (1) {
                rcu_read_lock();
[1]             eb = rcu_dereference(root->node);

                /*
                 * RCU really hurts here, we could free up the root node becaus
                 * it was COWed but we may not get the new root node yet so do
                 * the inc_not_zero dance and if it doesn't work then
                 * synchronize_rcu and try again.
                 */
[2]             if (atomic_inc_not_zero(&eb->refs)) {
                        rcu_read_aunlock();
                        break;
                }
                rcu_read_unlock();
                synchronize_rcu();
        }
        return eb;
}
```

Crash Analysis

- Null Pointer dereference - CVE-2019-19036

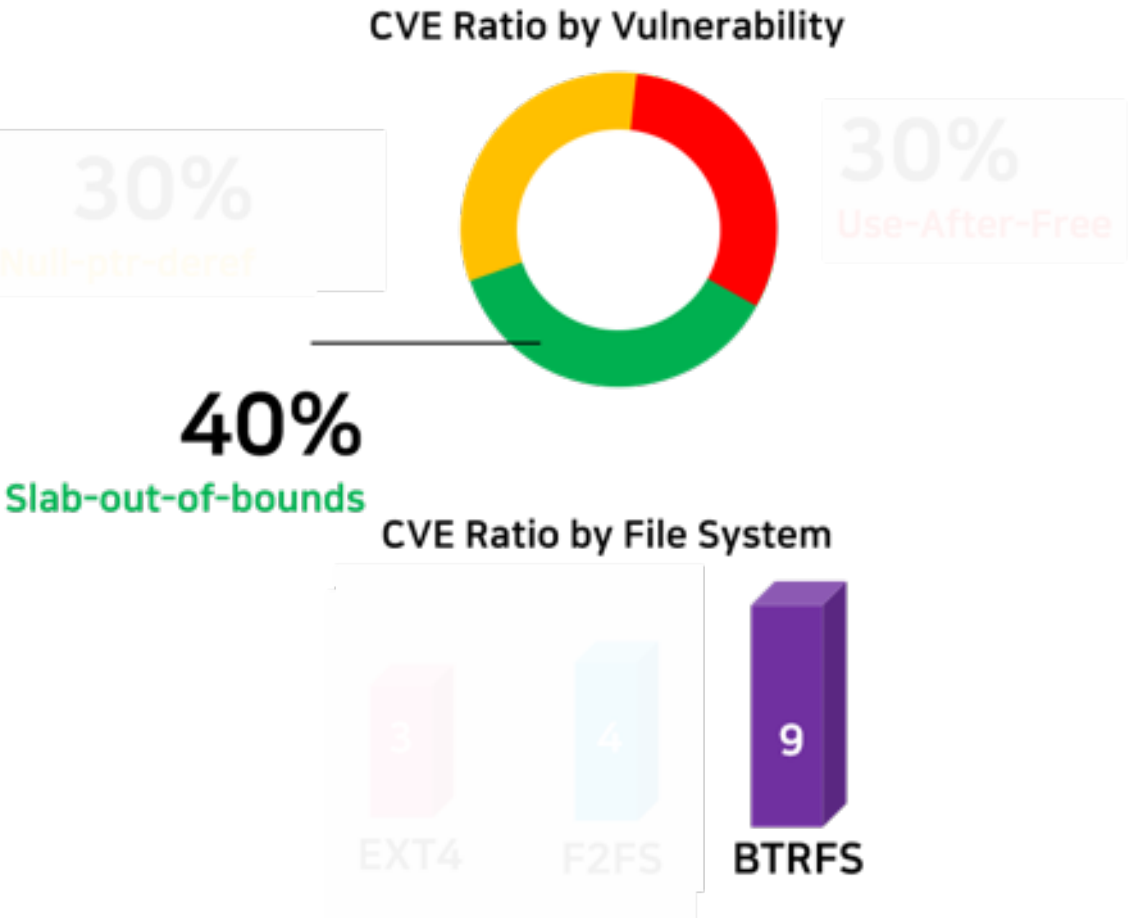
Exploitability

NULL pointer reference is not significant if you assume an attacker who can insert USB or mount the file system in a kernel.

```
struct extent_buffer *btrfs_root_node(struct btrfs_root *root)
{
        struct extent_buffer *eb;

        while (1) {
                rcu_read_lock();
[1]             eb = rcu_dereference(root->node);

                /*
                 * RCU really hurts here, we could free up the root node becaus
                 * it was COWed but we may not get the new root node yet so do
                 * the inc_not_zero dance and if it doesn't work then
                 * synchronize_rcu and try again.
                 */
[2]             if (atomic_inc_not_zero(&eb->refs)) {
                        rcu_read_aunlock();
                        break;
                }
                rcu_read_unlock();
                synchronize_rcu();
        }
        return eb;
}
```

** Internal exploitable score: 1

Crash Analysis



Slab Out-Of-Bounds

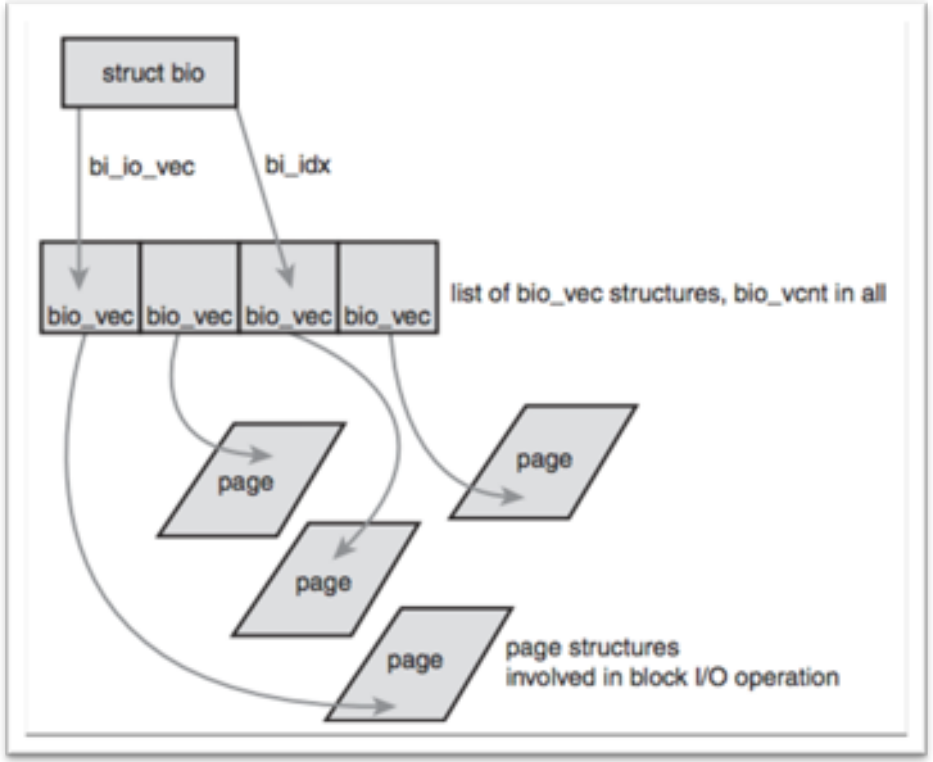
CVEs

- CVE-2019-19378
- CVE-2019-19449
- CVE-2019-19319
- CVE-2019-19927
- CVE-2019-19816

Crash Analysis

- **Slab-out-of-bounds - CVE-2019-19378**

RAID(Redundant array of inexpensive Disk)
The ability to use multiple storage devices as one big storage device. Low-cost devices can be used like expensive, high-capacity devices.
block->raid_map[5]
Each number has a different address to map a block that manages the virtual disk



Crash Analysis

- **Slab-out-of-bounds - CVE-2019-19378**

Cause

'page_index' returns the value of 'raid_map', which is a problem caused by the mutated value being entered here.
There is a possibility that it is related to other bugs besides OOB.
OOB occurs when a 'page_index' value greater than the array value of 'rbio->bip_pages' is entered.
This is a problem caused by touching the KASAN Red Zone while writing 'bvec.bv_page' value in the OOB area.

```
static void index_rbio_pages(struct btrfs_raid_bio *rbio)
{
        struct bio *bio;
        u64 start;
        unsigned long stripe_offset;
        unsigned long page_index;

        spin_lock_irq(&rbio->bio_list_lock);
        bio_list_for_each(bio, &rbio->bio_list) {
                struct bio_vec bvec;
                struct bvec_iter iter;
                int i = 0;

                start = (u64)bio->bi_iter.bi_sector << 9;
                stripe_offset = start - rbio->bbio->raid_map[0];
[1]             page_index = stripe_offset >> PAGE_SHIFT;

                if (bio_flagged(bio, BIO_CLONED))
                        bio->bi_iter = btrfs_io_bio(bio)->iter;

                bio_for_each_segment(bvec, bio, iter) {
[2]                     rbio->bio_pages[page_index + i] = bvec.bv_page;
                        i++;
                }
        }
        spin_unlock_irq(&rbio->bio_list_lock);
}
```

Crash Analysis

- **Slab-out-of-bounds - CVE-2019-19378**

Cause

'page_index' returns the value of 'raid_map', which is a problem caused by the mutated value being entered here.
There is a possibility that it is related to other bugs besides OOB.
OOB occurs when a 'page_index' value greater than the array value of 'rbio->bip_pages' is entered.
This is a problem caused by touching the KASAN Red Zone while writing 'bvec.bv_page' value in the OOB area.

```
static void index_rbio_pages(struct btrfs_raid_bio *rbio)
{
        struct bio *bio;
        u64 start;
        unsigned long stripe_offset;
        unsigned long page_index;

        spin_lock_irq(&rbio->bio_list_lock);
        bio_list_for_each(bio, &rbio->bio_list) {
                struct bio_vec bvec;
                struct bvec_iter iter;
                int i = 0;

                start = (u64)bio->bi_iter.bi_sector << 9;
                stripe_offset = start - rbio->bbio->raid_map[0];
[1]             page_index = stripe_offset >> PAGE_SHIFT;

                if (bio_flagged(bio, BIO_CLONED))
                        bio->bi_iter = btrfs_io_bio(bio)->iter;

                bio_for_each_segment(bvec, bio, iter) {
[2]                     rbio->bio_pages[page_index + i] = bvec.bv_page;
                        i++;
                }
        }
        spin_unlock_irq(&rbio->bio_list_lock);
}
```

Crash Analysis

- **Slab-out-of-bounds - CVE-2019-19378**

Cause
'page_index' returns the value of 'raid_map', which is a problem caused by the mutated value being entered here. There is a possibility that it is related to other bugs besides OOB. OOB occurs when a 'page_index' value greater than the array value of 'rbio->bip_pages' is entered. This is a problem caused by touching the KASAN Red Zone while writing 'bvec.bv_page' value in the OOB area.

```
static void index_rbio_pages(struct btrfs_raid_bio *rbio)
{
        struct bio *bio;
        u64 start;
        unsigned long stripe_offset;
        unsigned long page_index;

        spin_lock_irq(&rbio->bio_list_lock);
        bio_list_for_each(bio, &rbio->bio_list) {
                struct bio_vec bvec;
                struct bvec_iter iter;
                int i = 0;

                start = (u64)bio->bi_iter.bi_sector << 9;
                stripe_offset = start - rbio->bbio->raid_map[0];
[1]             page_index = stripe_offset >> PAGE_SHIFT;

                if (bio_flagged(bio, BIO_CLONED))
                        bio->bi_iter = btrfs_io_bio(bio)->iter;

                bio_for_each_segment(bvec, bio, iter) {
[2]                     rbio->bio_pages[page_index + i] = bvec.bv_page;
                        i++;
                }
        }
        spin_unlock_irq(&rbio->bio_list_lock);
}
```

Crash Analysis

- Slab-out-of-bounds - CVE-2019-19378

Exploitability

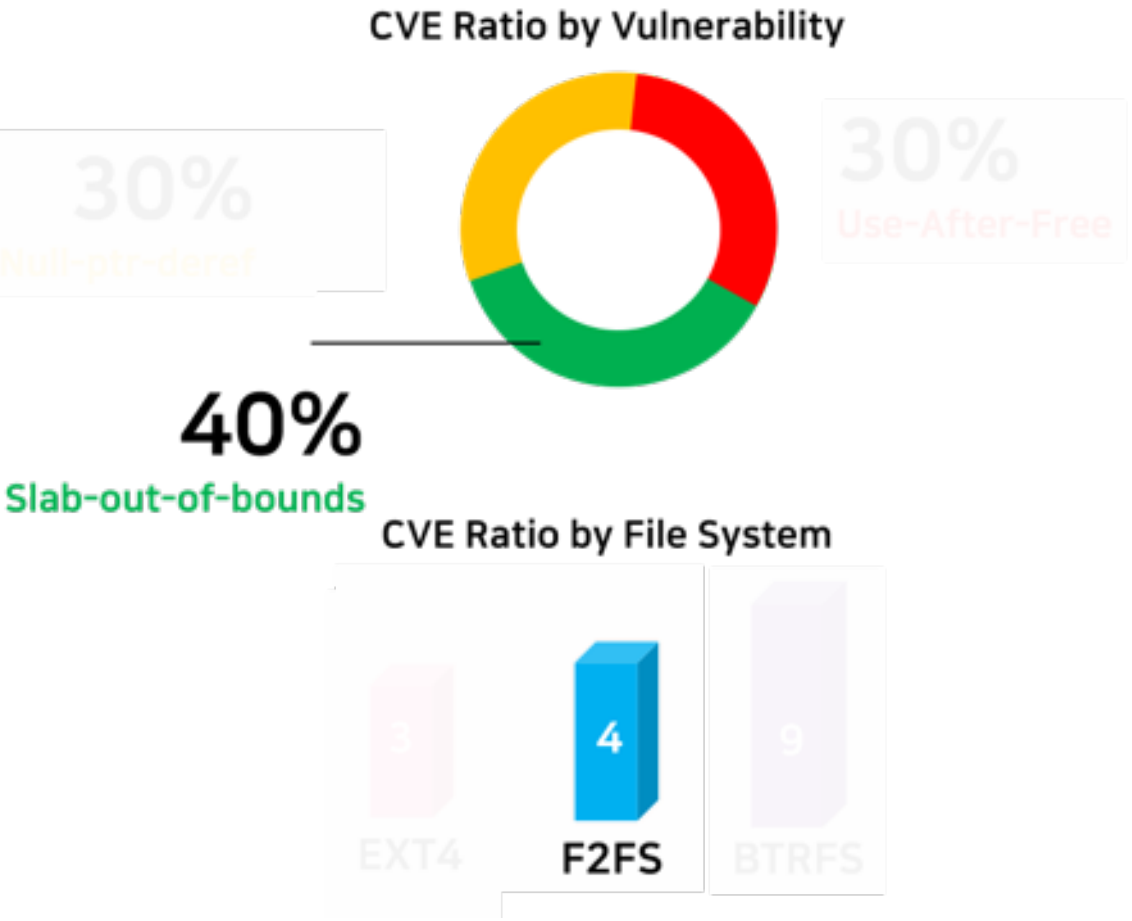
Control flow hijacking is possible only when all three problems are solved.

1. You need to know how the 'crafted value' entered the raid_map[0] comes in.
2. Need 'Heap feng shui' to place the target object in adjacent memory of 'rbio-bbio_page[]'.
3. The 'bvec.bv_page' value to be overwritten as the hacker needs

```
static void index_rbio_pages(struct btrfs_raid_bio *rbio)
{
        struct bio *bio;
        u64 start;
        unsigned long stripe_offset;
        unsigned long page_index;

        spin_lock_irq(&rbio->bio_list_lock);
        bio_list_for_each(bio, &rbio->bio_list) {
                struct bio_vec bvec;
                struct bvec_iter iter;
                int i = 0;

                start = (u64)bio->bi_iter.bi_sector << 9;
                stripe_offset = start - rbio->bbio->raid_map[0];
[1]             page_index = stripe_offset >> PAGE_SHIFT;

                if (bio_flagged(bio, BIO_CLONED))
                        bio->bi_iter = btrfs_io_bio(bio)->iter;

                bio_for_each_segment(bvec, bio, iter) {
[2]                     rbio->bio_pages[page_index + i] = bvec.bv_page;
                        i++;
                }
        }
        spin_unlock_irq(&rbio->bio_list_lock);
}
```

**** Internal exploitable score: 3**

Crash Analysis



Slab Out-Of-Bounds

CVEs

- CVE-2019-19378
- CVE-2019-19449
- CVE-2019-19319
- CVE-2019-19927
- CVE-2019-19816

Crash Analysis

- **Slab-out-of-bounds - CVE-2019-19449**

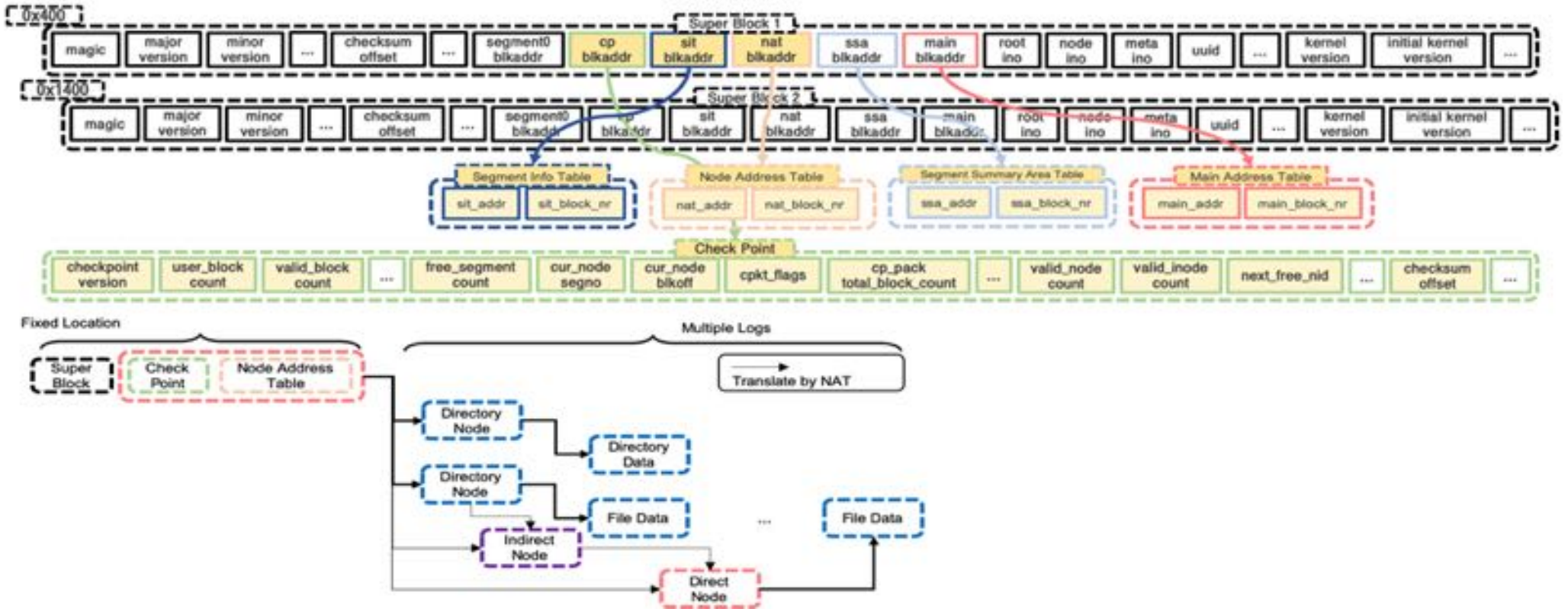
CP	File system info., bitmaps for valid NAT/SIT sets, and summary of current active segments
SIT	Segment info. such as valid block count and bitmap for the validity of all the blocks
NAT	Block address table for all the node blocks stored in the main area
SSA	Summary entries which contains the owner info. Of all the data and node blocks

```
struct sit_info {
        const struct segment_allocation *s_ops;

        block_t sit_base_addr;          /* start block address of SIT area */
        block_t sit_blocks;             /* # of blocks used by SIT area */
        block_t written_valid_blocks;   /* # of valid blocks in main area */
        char *sit_bitmap;               /* SIT bitmap pointer */
#ifdef CONFIG_F2FS_CHECK_FS
        char *sit_bitmap_mir;           /* SIT bitmap mirror */
#endif
        unsigned int bitmap_size;       /* SIT bitmap size */

        unsigned long *tmp_map;                 /* bitmap for temporal use */
        unsigned long *dirty_sentries_bitmap;   /* bitmap for dirty sentries */
        unsigned int dirty_sentries;            /* # of dirty sentries */
        unsigned int sents_per_block;           /* # of SIT entries per block */
        struct rw_semaphore sentry_lock;        /* to protect SIT cache */
        struct seg_entry *sentries;             /* SIT segment-level cache */
        struct sec_entry *sec_entries;          /* SIT section-level cache */

        /* for cost-benefit algorithm in cleaning procedure */
```

```
static inline struct seg_entry *get_seg_entry(struct f2fs_sb_info *sbi,
                                              unsigned int segno)
{
        struct sit_info *sit_i = SIT_I(sbi);
        return &sit_i->sentries[segno];
}
```

Crash Analysis

- Slab-out-of-bounds - CVE-2019-19449



Crash Analysis

- Slab-out-of-bounds - CVE-2019-19449

Cause

There is a logic that checks the dirtyness of a section within the structure that manages the segments of the f2fs file system.
mtime is required for logic to periodically inspect segments to be divided or added by the GC algorithm of the f2fs file system.
In these test logic, the value of 0x19 is entered because the segno or i value is incorrectly calculated, which is greater than the array previously allocated.
This causes problems when reading mtime in out of bound memory area.

```
        for (segno = 0; segno < MAIN_SEGS(sbi); segno += sbi->segs_per_sec) {
                unsigned int i;
                unsigned long long mtime = 0;

                for (i = 0; i < sbi->segs_per_sec; i++)
(1)               mtime += get_seg_entry(sbi, segno + i)->mtime;

                mtime = div_u64(mtime, sbi->segs_per_sec);

                if (sit_i->min_mtime > mtime)
                        sit_i->min_mtime = mtime;
        }
        sit_i->max_mtime = get_mtime(sbi, false);
```

```
static inline struct seg_entry *get_seg_entry(struct f2fs_sb_info *sbi,
                                              unsigned int segno)
{
        struct sit_info *sit_i = SIT_I(sbi);
        return &sit_i->sentries[segno];
}
```

Crash Analysis

- Slab-out-of-bounds - CVE-2019-19449

Cause

There is a logic that checks the dirtyness of a section within the structure that manages the segments of the f2fs file system.
mtime is required for logic to periodically inspect segments to be divided or added by the GC algorithm of the f2fs file system.
In these test logic, the value of 0x19 is entered because the segno or i value is incorrectly calculated, which is greater than the array previously allocated.
This causes problems when reading mtime in out of bound memory area.

```
        for (segno = 0; segno < MAIN_SEGS(sbi); segno += sbi->segs_per_sec) {
                unsigned int i;
                unsigned long long mtime = 0;

                for (i = 0; i < sbi->segs_per_sec; i++)
(1)               mtime += get_seg_entry(sbi, segno + i)->mtime;

                mtime = div_u64(mtime, sbi->segs_per_sec);

                if (sit_i->min_mtime > mtime)
                        sit_i->min_mtime = mtime;
        }
        sit_i->max_mtime = get_mtime(sbi, false);
```

```
static inline struct seg_entry *get_seg_entry(struct f2fs_sb_info *sbi,
                                               unsigned int segno)
{
        struct sit_info *sit_i = SIT_I(sbi);
        return &sit_i->entries[segno];
}
```

Crash Analysis

- Slab-out-of-bounds - CVE-2019-19449

Exploitability

'Exploit' is easier when arbitrary R/W is available.

Currently, OOB read value is stored in a long long long type variable called 'mtime' and used elsewhere.

It is meaningful if 'mtime' is printed directly or the side channel attack can be performed with 'mtime'.

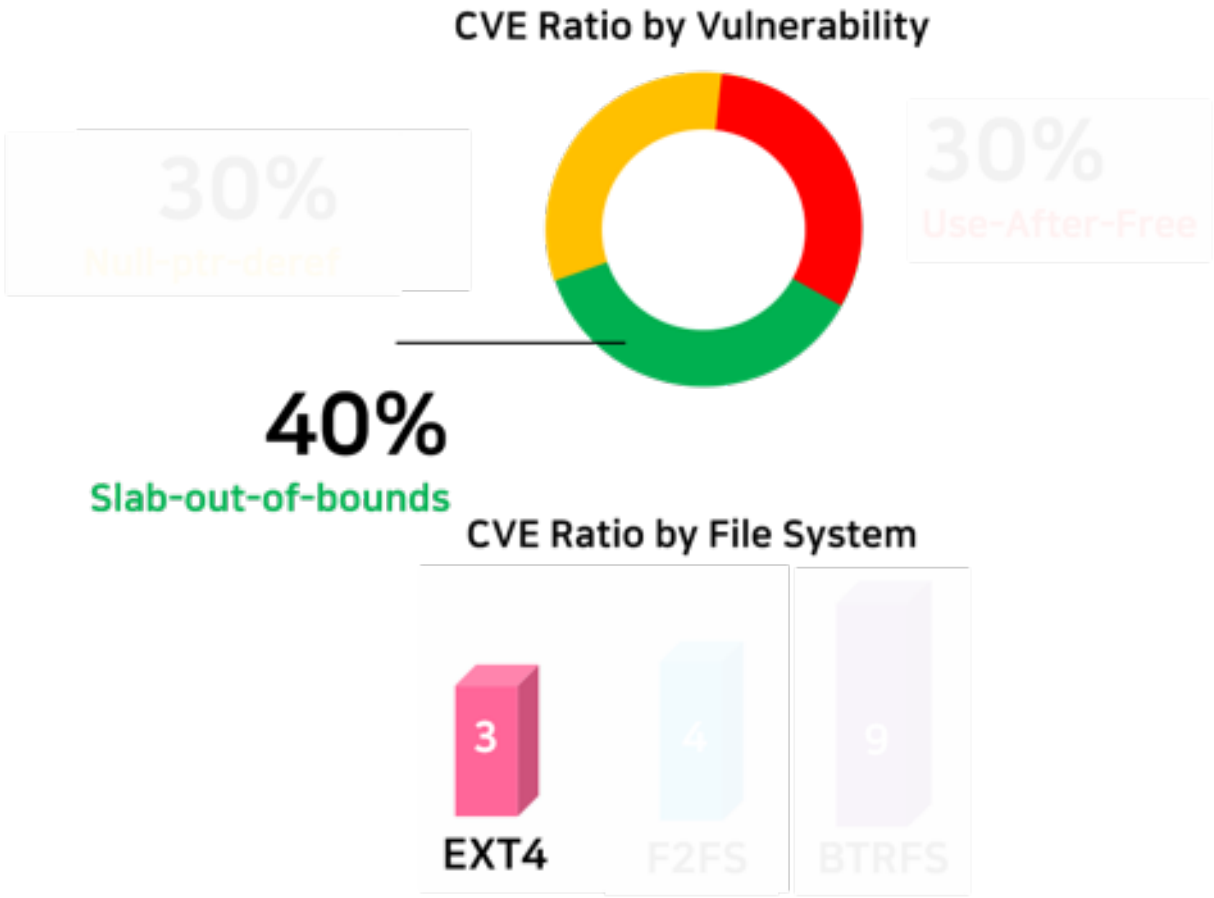
However, it is considered difficult to directly control flow hijacking.

```
        for (segno = 0; segno < MAIN_SEGS(sbi); segno += sbi->segs_per_sec) {
                unsigned int i;
                unsigned long long mtime = 0;

                for (i = 0; i < sbi->segs_per_sec; i++)
(1)               mtime += get_seg_entry(sbi, segno + i)->mtime;

                mtime = div_u64(mtime, sbi->segs_per_sec);

                if (sit_i->min_mtime > mtime)
                        sit_i->min_mtime = mtime;
        }
        sit_i->max_mtime = get_mtime(sbi, false);
```

```
static inline struct seg_entry *get_seg_entry(struct f2fs_sb_info *sbi,
                                              unsigned int segno)
{
        struct sit_info *sit_i = SIT_I(sbi);
        return &sit_i->sentries[segno];
}
```

**** Internal exploitable score: 3**

Crash Analysis



Slab Out-Of-Bounds

CVEs

- CVE-2019-19378
- CVE-2019-19449
- CVE-2019-19319
- CVE-2019-19927
- CVE-2019-19816

Crash Analysis

- Slab-out-of-bounds - CVE-2019-19449

Cause

old_size is set in the process of setting up an entry in the ext4 filesystem.(Returns the value of EXT4_XATTR_SIZE.)
That's when old_size is created using le32_to_cpu, Cause in large unexpected values in a 64-bit OS.
An unexpected large value is set to the size argument of memset. This is a vulnerability caused by initializing the area beyond the memory of first_val to 0.

```
static int ext4_xattr_set_entry(struct ext4_xattr_info *i,
                                struct ext4_xattr_search *s,
                                handle_t *handle, struct inode *inode,
                                bool is_block)
{
        struct ext4_xattr_entry *last, *next;
        struct ext4_xattr_entry *here = s->here;
        size_t min_offs = s->end - s->base, name_len = strlen(i->name);
        int in_inode = i->in_inode;
        struct inode *old_ea_inode = NULL;
        struct inode *new_ea_inode = NULL;
        size_t old_size, new_size;
        int ret;

        /* Space used by old and new values. */
        old_size = (!s->not_found && !here->e_value_inum) ?
[1]             EXT4_XATTR_SIZE(le32_to_cpu(here->e_value_size)) : 0;
        new_size = (i->value && !in_inode) ? EXT4_XATTR_SIZE(i->value_len) : 0;
```

```
memmove(first_val + old_size, first_val, val - first_val);
memset(first_val, 0, old_size);
min_offs += old_size;
```

Crash Analysis

- Slab-out-of-bounds - CVE-2019-19449

Exploitability

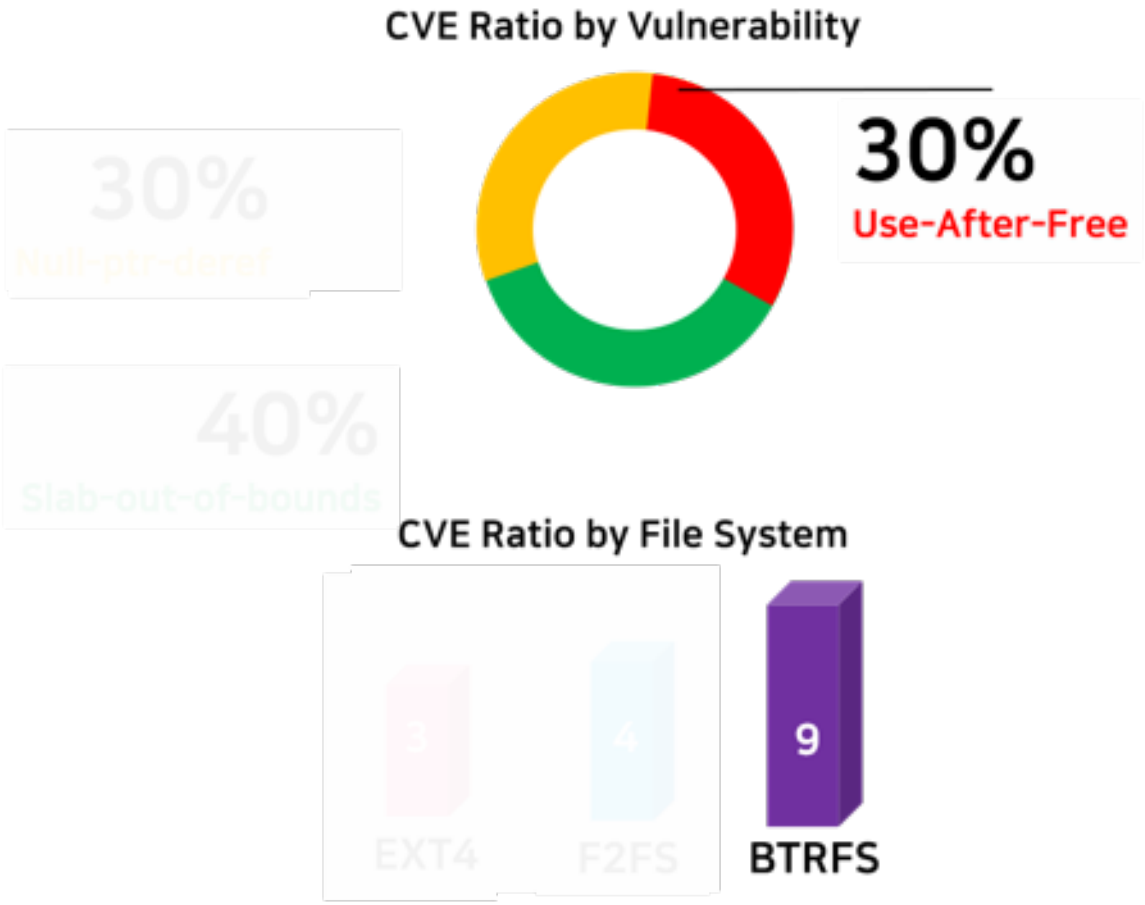
EXT_4_XATTR_SIZE return value cannot be manipulated arbitrarily.
The data that can be written is also not a value that can be arbitrarily written and was hard coded to 0. Only 0 can write memory.
An exploit can be exploited by placing an object with a credential method that raises the privilege when memset to 0 after the first_val variable. However, you should consider the side effects that occur as the memory memsets to zero.

```
static int ext4_xattr_set_entry(struct ext4_xattr_info *i,
                                struct ext4_xattr_search *s,
                                handle_t *handle, struct inode *inode,
                                bool is_block)
{
        struct ext4_xattr_entry *last, *next;
        struct ext4_xattr_entry *here = s->here;
        size_t min_offs = s->end - s->base, name_len = strlen(i->name);
        int in_inode = i->in_inode;
        struct inode *old_ea_inode = NULL;
        struct inode *new_ea_inode = NULL;
        size_t old_size, new_size;
        int ret;

        /* Space used by old and new values. */
        old_size = (!s->not_found && !here->e_value_inum) ?
[1]                     EXT4_XATTR_SIZE(le32_to_cpu(here->e_value_size)) : 0;
        new_size = (i->value && !in_inode) ? EXT4_XATTR_SIZE(i->value_len) : 0;
```

```
memmove(first_val + old_size, first_val, val - first_val);
memset(first_val, 0, old_size);
min_offs += old_size;
```

** Internal exploitable score: 4

Crash Analysis



Use-After-Free
CVEs • CVE-2019-19318 • CVE-2019-19447 • CVE-2019-19448 • CVE-2019-19377

Crash Analysis

- Use-After-Free - CVE-2019-19318

Cause

The rwsem_can_spin_on_owner () function calls rcu_read_lock() an internal function that sets the spinlock, and executes the owner_on_cpu(owner) function to find out who the owner of the current CPU is.
If the owner is write_mode, all the data in the object is deleted when the spinlock is deallocated, but in read_mode, the data in the object is not deleted even if the spinlock is deallocated.
The owner_on_cpu() function takes a pointer and processes it, causing a malfunction if an deallocated object enters. That is, once mounted, no problem, but when mounted twice, the remaining pointer causes problems.

```
        rcu_read_lock();
[1]     owner = rwsem_owner_flags(sem, &flags);
        /*
         * Don't check the read-owner as the entry may be stale.
         */
        if ((flags & nonspinnable) ||
[2]         (owner && !(flags & RWSEM_READER_OWNED) && !owner_on_cpu(owner)))
                ret = false;
        rcu_read_unlock();
        preempt_enable();
```

Crash Analysis

- Use-After-Free - CVE-2019-19318

Cause

The rwsem_can_spin_on_owner () function calls rcu_read_lock(), an internal function that sets the spinlock, and executes the owner_on_cpu(owner) function to find out who the owner of the current CPU is.
If the owner is write_mode, all the data in the object is deleted when the spinlock is deallocated, but in read_mode, the data in the object is not deleted even if the spinlock is deallocated.
The owner_on_cpu() function takes a pointer and processes it, causing a malfunction if an deallocated object enters. That is, once mounted, no problem, but when mounted twice, the remaining pointer causes problems.

```
        rcu_read_lock();
[1]     owner = rwsem_owner_flags(sem, &flags);
        /*
         * Don't check the read-owner as the entry may be stale.
         */
        if ((flags & nonspinnable) ||
[2]         (owner && !(flags & RWSEM_READER_OWNED) && !owner_on_cpu(owner)))
                ret = false;
        rcu_read_unlock();
        preempt_enable();
```

Crash Analysis

- **Use-After-Free - CVE-2019-19318**

Cause

When the craft image is first mounted, the owner in this semaphore is set to read permission and deallocated upon unlock.

The owner of the deallocated semaphore task is deallocated, and the dummy value is placed in this object at the second mount.

owner->flag value contains a dummy value but does not map to RW_SEM_READER, so it is entered as an argument of the owner_on_cpu() function.

```
        rcu_read_lock();
[1]     owner = rwsem_owner_flags(sem, &flags);
        /*
         * Don't check the read-owner as the entry may be stale.
         */
        if ((flags & nonspinnable) ||
[2]         (owner && !(flags & RWSEM_READER_OWNED) && !owner_on_cpu(owner)))
                ret = false;
        rcu_read_unlock();
        preempt_enable();
```

Crash Analysis

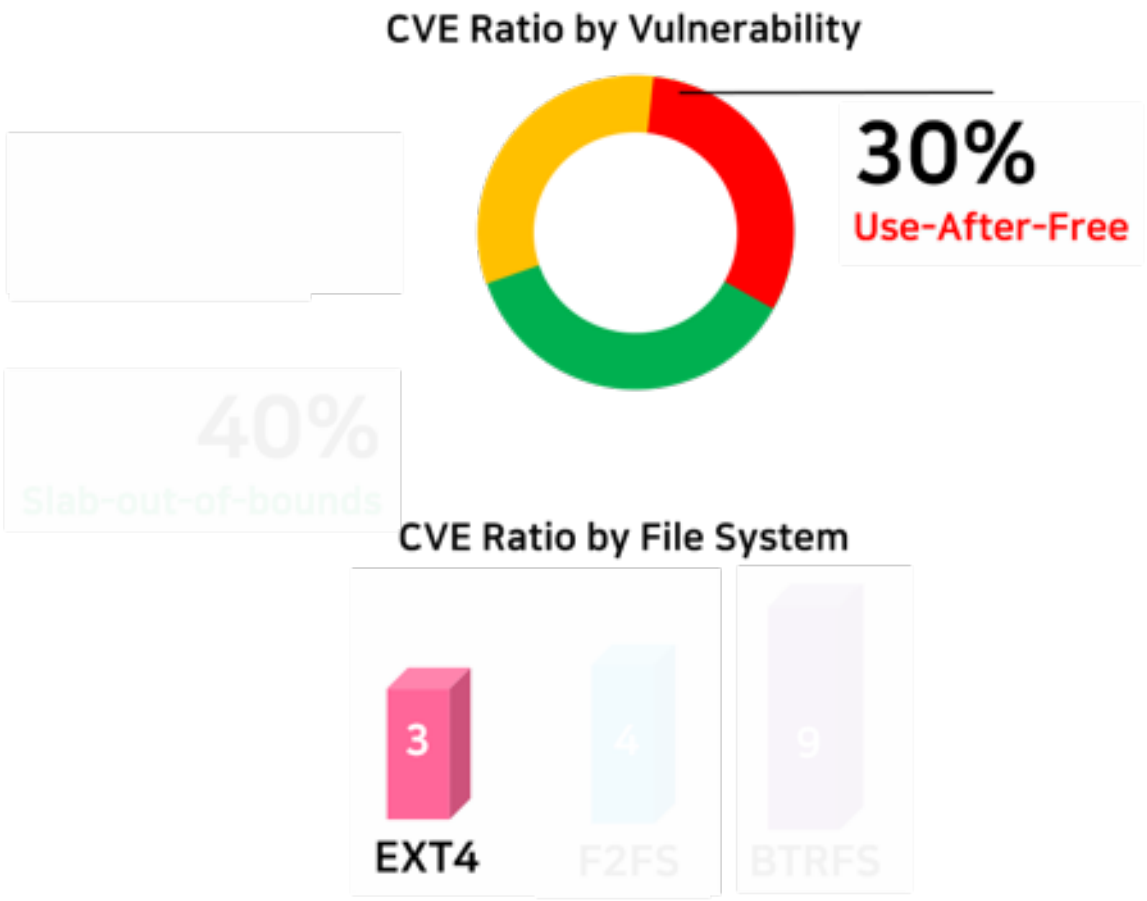
- Use-After-Free - CVE-2019-19318

Exploitability
If you properly message the heap before the owner object is reallocated, you can adjust the owner value. owner value manages a lot of data for locking, but there is a difficulty in control flow hijacking because there is no variable for function pointer or related struct execution.

```
        rcu_read_lock();
[1]     owner = rwsem_owner_flags(sem, &flags);
        /*
         * Don't check the read-owner as the entry may be stale.
         */
        if ((flags & nonspinnable) ||
[2]         (owner && !(flags & RWSEM_READER_OWNED) && !owner_on_cpu(owner)))
                ret = false;
        rcu_read_unlock();
        preempt_enable();
```

**** Internal exploitable score: 6**

Crash Analysis



Use-After-Free

CVEs

- CVE-2019-19318
- CVE-2019-19447
- CVE-2019-19448
- CVE-2019-19377

Crash Analysis

- Use-After-Free - CVE-2019-19447

Cause
Use-After-Free vulnerability that occurs when the dump_orphan_list() function outputs a list of orphaned inodes from the Ext4 filesystem without checking the deallocated inodes. Importing an orphan list from a super block can sometimes lead to deallocated objects.

```
static void dump_orphan_list(struct super_block *sb, struct ext4_sb_info *sbi)
{
        struct list_head *l;

        ext4_msg(sb, KERN_ERR, "sb orphan head is %d",
                 le32_to_cpu(sbi->s_es->s_last_orphan));

        printk(KERN_ERR "sb_info orphan list:\n");
        list_for_each(l, &sbi->s_orphan) {
                struct inode *inode = orphan_list_entry(l);
                printk(KERN_ERR "  "
[1]                     "inode %s:%lu at %p: mode %o, nlink %d, next %d\n",
                        inode->i_sb->s_id, inode->i_ino, inode,
                        inode->i_mode, inode->i_nlink,
                        NEXT_ORPHAN(inode));
        }
}
```

Crash Analysis

- Use-After-Free - CVE-2019-19447

Cause

Use-After-Free vulnerability that occurs when the dump_orphan_list() function outputs a list of orphaned inodes from the Ext4 filesystem without checking the deallocated inodes.

Importing an orphan list from a super block can sometimes lead to deallocated objects.

```
static void dump_orphan_list(struct super_block *sb, struct ext4_sb_info *sbi)
{
        struct list_head *l;

        ext4_msg(sb, KERN_ERR, "sb orphan head is %d",
                 le32_to_cpu(sbi->s_es->s_last_orphan));

        printk(KERN_ERR "sb_info orphan list:\n");
        list_for_each(l, &sbi->s_orphan) {
                struct inode *inode = orphan_list_entry(l);
                printk(KERN_ERR "  "
[1]                     "inode %s:%lu at %p: mode %o, nlink %d, next %d\n",
                        inode->i_sb->s_id, inode->i_ino, inode,
                        inode->i_mode, inode->i_nlink,
                        NEXT_ORPHAN(inode));
        }
}
```

Crash Analysis

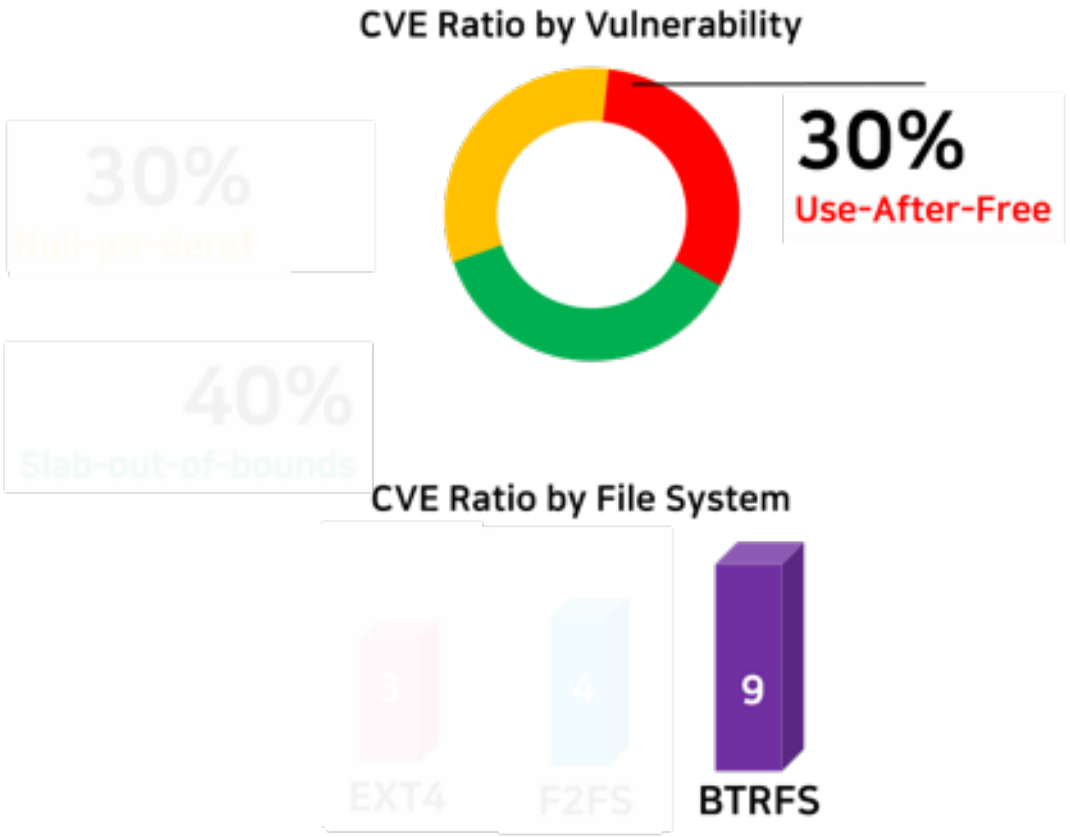
- Use-After-Free - CVE-2019-19447

Exploitability
Use-After-Free vulnerability that occurs when the dump_orphan_list() function outputs a list of orphaned inodes from the Ext4 filesystem without checking the deallocated inodes. Importing an orphan list from a super block can sometimes lead to deallocated objects.

```
static void dump_orphan_list(struct super_block *sb, struct ext4_sb_info *sbi)
{
        struct list_head *l;

        ext4_msg(sb, KERN_ERR, "sb orphan head is %d",
                 le32_to_cpu(sbi->s_es->s_last_orphan));

        printk(KERN_ERR "sb_info orphan list:\n");
        list_for_each(l, &sbi->s_orphan) {
                struct inode *inode = orphan_list_entry(l);
                printk(KERN_ERR "  "
[1]                     "inode %s:%lu at %p: mode %o, nlink %d, next %d\n",
                        inode->i_sb->s_id, inode->i_ino, inode,
                        inode->i_mode, inode->i_nlink,
                        NEXT_ORPHAN(inode));
        }
}
```

** Internal exploitable score: 3

Crash Analysis



Use-After-Free

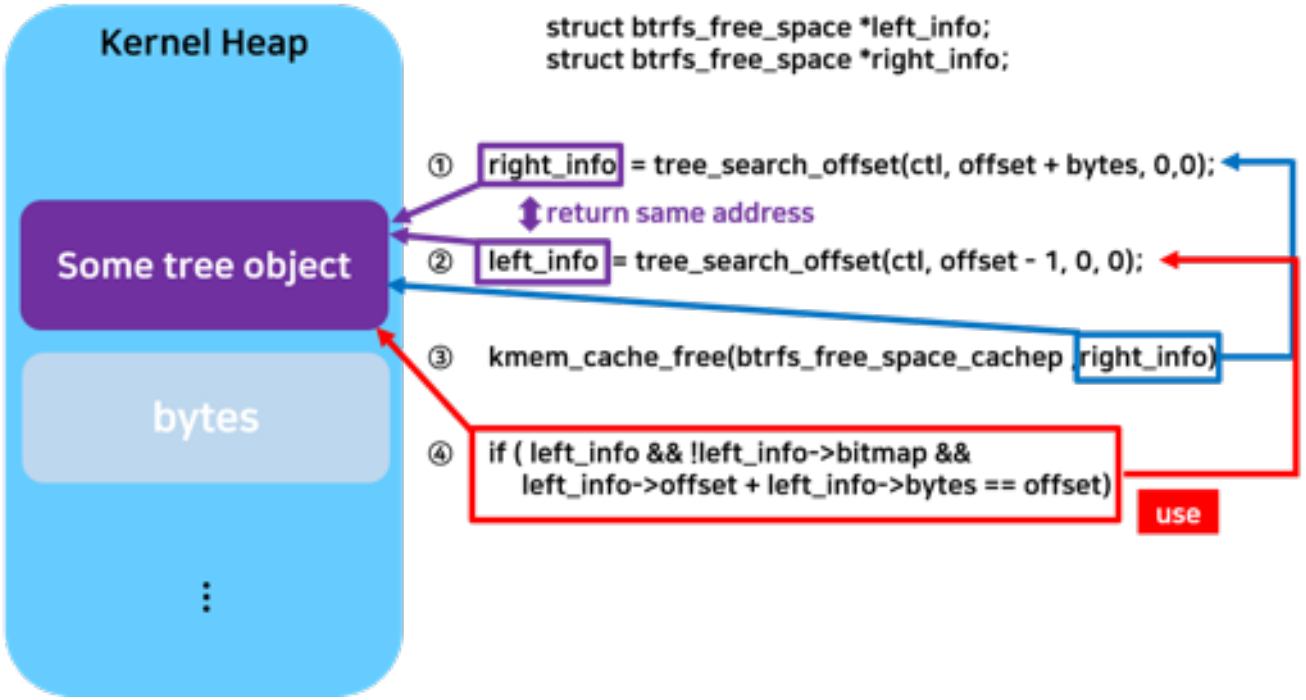
CVEs

- CVE-2019-19318
- CVE-2019-19447
- CVE-2019-19448
- CVE-2019-19377

Crash Analysis

- Use-After-Free - CVE-2019-19448

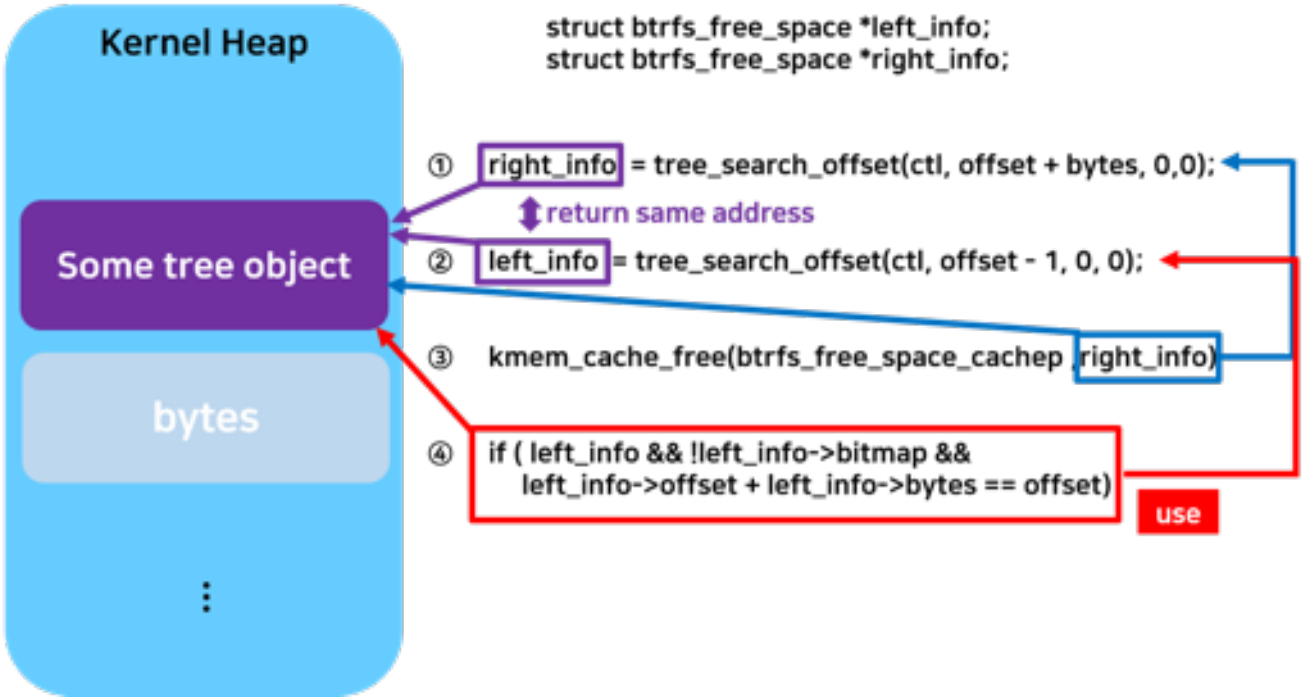
Cause
This problem occurs when the try_merge_free_space () function points to the same object with the left_info pointer and the right_info pointer, which are pointers to the double linked list of the structure for managing btrfs.



Crash Analysis

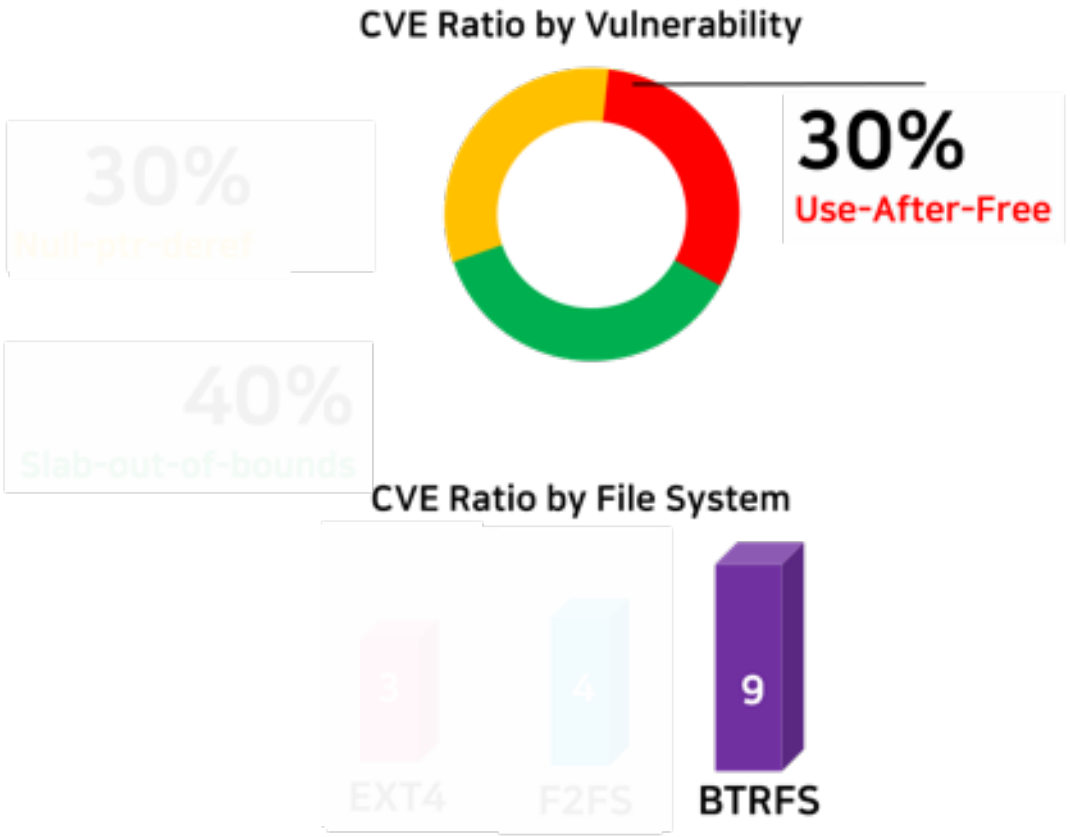
- Use-After-Free - CVE-2019-19448

Exploitability
Control Flow Hijacking is possible if you can push other objects at the timing of being re-referenced to deallocated objects! (TOCTOU). However, the code interval is so short that it is difficult to race-condition.



** Internal exploitable score: 7

Crash Analysis



Use-After-Free

CVEs

- CVE-2019-19318
- CVE-2019-19447
- CVE-2019-19448
- CVE-2019-19377

Crash Analysis

- Use-After-Free - CVE-2019-19377

Cause

BTRFS has a btrfs_workqueue * wq structure that manages files in action.

Wq freed from btrfs_queue_work () function comes in and occurs when referencing
wq-> high in if statement

```
struct btrfs_workqueue *btrfs_alloc_workqueue(struct btrfs_fs_info *fs_info,
                                              const char *name,
                                              unsigned int flags,
                                              int limit_active,
                                              int thresh);
```

```
void btrfs_queue_work(struct btrfs_workqueue *wq,
                      struct btrfs_work *work)
{
  struct __btrfs_workqueue *dest_wq;

>>> if (test_bit(WORK_HIGH_PRIO_BIT, &work->flags) && wq->high)
    dest_wq = wq->high;
  else
    dest_wq = wq->normal;
  __btrfs_queue_work(dest_wq, work);
}
```

Crash Analysis

- Use-After-Free - CVE-2019-19377

Cause

BTRFS has a btrfs_workqueue * wq structure that manages files in action.

Wq freed from btrfs_queue_work () function comes in and occurs when referencing
wq-> high in if statement

```
struct btrfs_workqueue *btrfs_alloc_workqueue(struct btrfs_fs_info *fs_info,
                                              const char *name,
                                              unsigned int flags,
                                              int limit_active,
                                              int thresh);
```

```
void btrfs_queue_work(struct btrfs_workqueue *wq,
                      struct btrfs_work *work)
{
  struct __btrfs_workqueue *dest_wq;

>>> if (test_bit(WORK_HIGH_PRIO_BIT, &work->flags) && wq->high)
    dest_wq = wq->high;
  else
    dest_wq = wq->normal;
  __btrfs_queue_work(dest_wq, work);
}
```

Crash Analysis

- Use-After-Free - CVE-2019-19377

Cause
BTRFS has a btrfs_workqueue * wq structure that manages files in action. Wq freed from btrfs_queue_work () function comes in and occurs when referencing wq-> high in if statement

```
struct btrfs_workqueue *btrfs_alloc_workqueue(struct btrfs_fs_info *fs_info,
                                              const char *name,
                                              unsigned int flags,
                                              int limit_active,
                                              int thresh);
```

```
void btrfs_queue_work(struct btrfs_workqueue *wq,
                      struct btrfs_work *work)
{
  struct __btrfs_workqueue *dest_wq;

>>> if (test_bit(WORK_HIGH_PRIO_BIT, &work->flags) && wq->high)
    dest_wq = wq->high;
  else
    dest_wq = wq->normal;
  __btrfs_queue_work(dest_wq, work);
}
```

Crash Analysis

- Use-After-Free - CVE-2019-19377

Exploitability
If we know the timing at which the wq is freed, then UAF can easily do so by inserting another object to replace it. Control flow hijacking is also possible if there is a meaningful function pointer in the __btrfs_workqueue structure and the logic that triggers the function pointer is present. Requires logic to trigger function pointers.

```
22  struct __btrfs_workqueue {
23          struct workqueue_struct *normal_wq;
24
25          /* File system this workqueue services */
26          struct btrfs_fs_info *fs_info;
27
28          /* List head pointing to ordered work list */
29          struct list_head ordered_list;
30
31          /* Spinlock for ordered_list */
32          spinlock_t list_lock;
33
34          /* Thresholding related variants */
35          atomic_t pending;
```

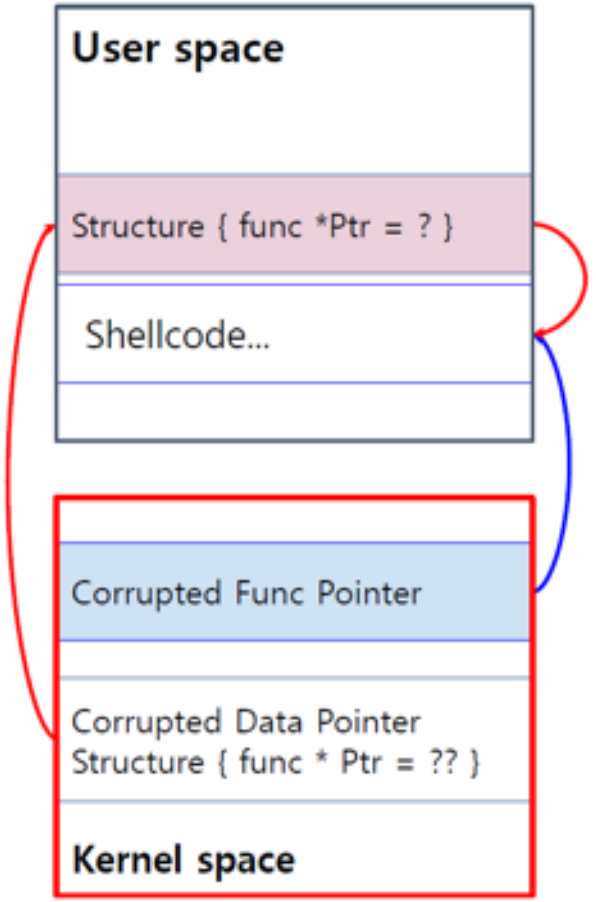
** Internal exploitable score: 7

General Kernel Exploitation

General Kernel Exploitation

- ret2usr

- **Exploit code is created in the user area and the code is executed with Kernel authority**
  - Function pointer tampering in kernel space
    - Direct exploit code execution through altered function pointer calls
  - Modifies structure pointers in kernel space
    - Indirect exploit code execution through function pointers in a modulated structure

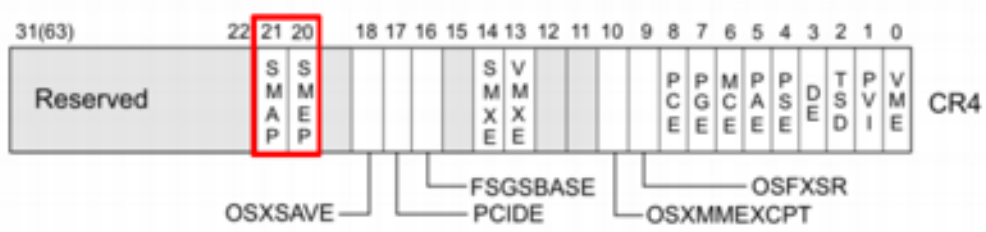


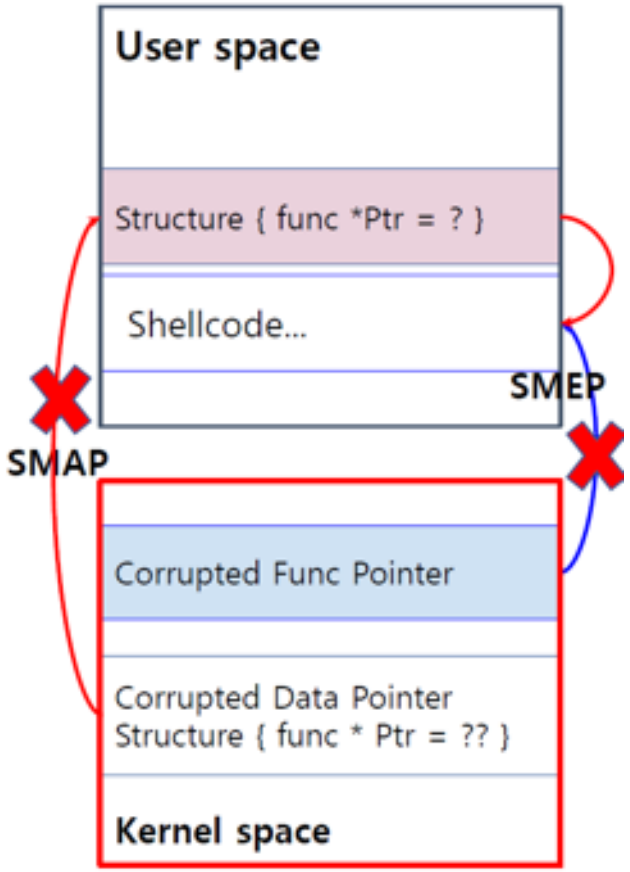
General Kernel Exploitation

- SMAP / SMEP

- **SMAP(Supervisor Mode Access Prevention)**
  - Access Violation occurs when referring to user-space address with kernel privileges
    - Managed via bit 21 of CR4 register

- **SMEP(Supervisor Mode Execution Prevention)**
  - Access Violation occurs when running user-space code with kernel privileges
    - Managed via bit 20 of CR4 register

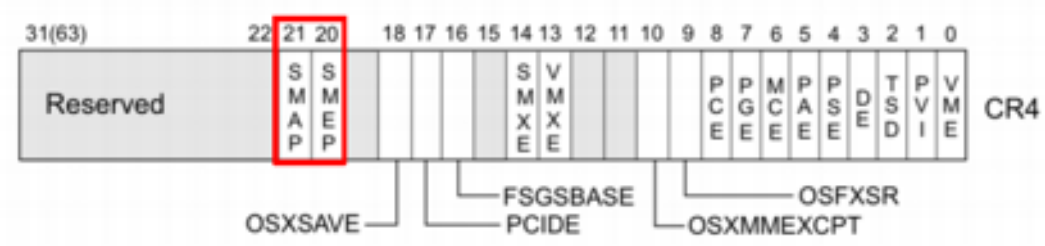


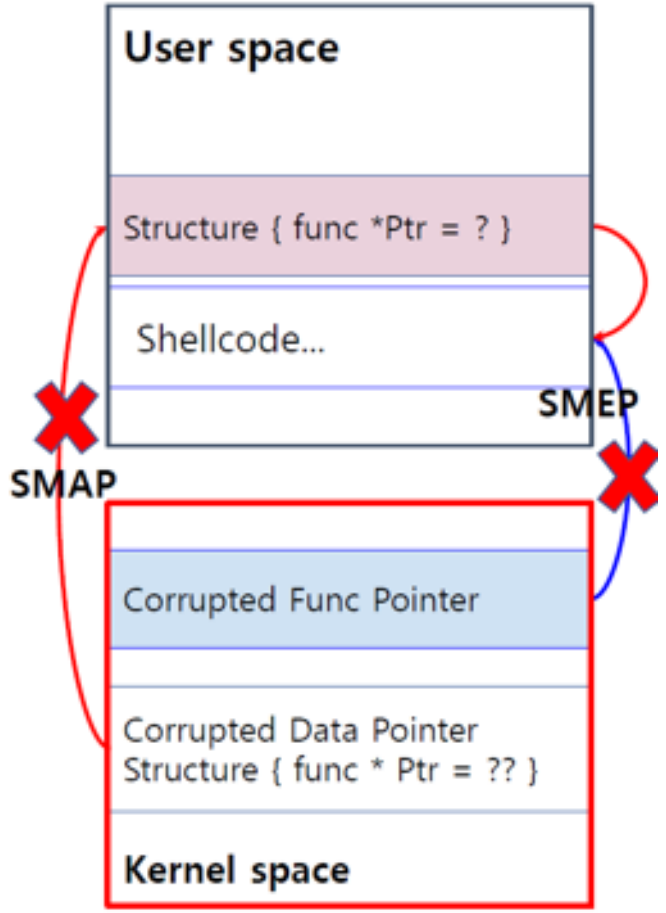


General Kernel Exploitation

- SMAP / SMEP bypass

- **How to bypass SMAP/SMEP?**
  - The value of the CR4 register can be changed!
  - Register value change through kernel ROP
  - Return-to-dir Technique

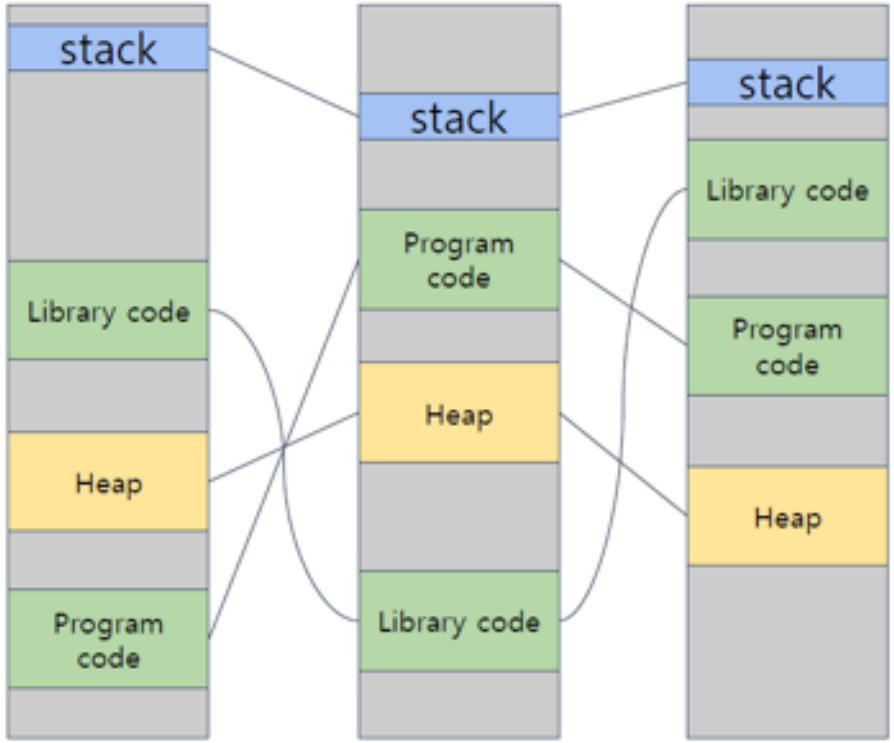




General Kernel Exploitation

- KASLR

- **ASLR(Address Space Layout Randomization)**
  - The ASLR protection scheme randomly places the address of the memory region to which the shared library, stack, and heap are mapped. The reason is that when attackers write attack code, they are not able to predict in advance where the payload should be inserted.
- **KASLR (Kernel Address space layout Randomization)**
  - A defensive technique that changes the base address at which the kernel is loaded at every boot, making it impossible to predict in advance where the payload should be inserted when writing attack code in kernel memory.



General Kernel Exploitation

- **Linux kernel heap**

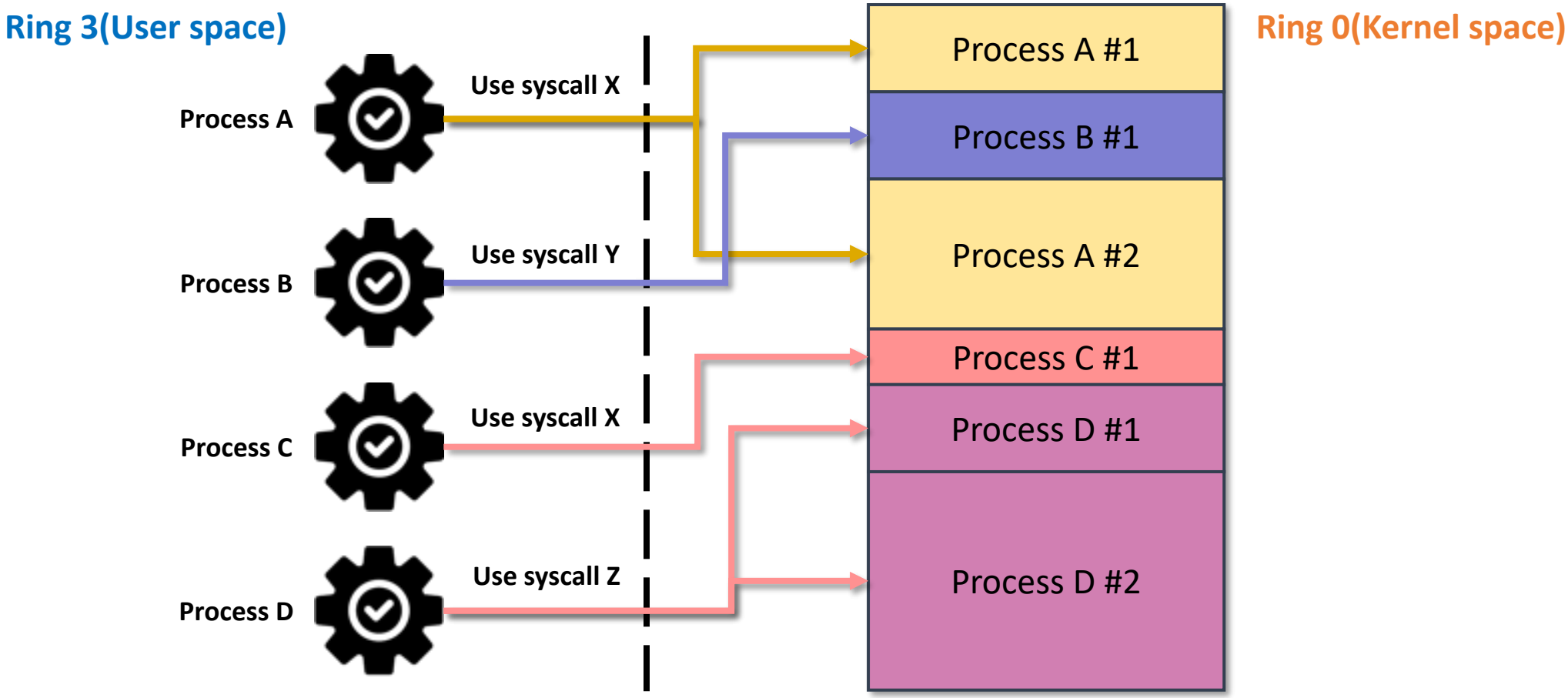
- User can’t allocate kernel heap directly
  - So, we must use specific **syscall** to allocate/free kernel heap

- Allocatable min/max size of heap block is **limited**
  - Each syscall has different min/max allocatable size
  - Possibility of heap spray failure

- All process uses kernel heap, shares kernel heap memory
  - Possibility of heap manipulation failure

General Kernel Exploitation

- Linux kernel heap allocation



General Kernel Exploitation

- **Linux kernel heap spraying**

- Three rules for kernel spray
  - User callable syscall (with low privilege)
  - Flexible size of kernel heap block
  - Controllable payload of heap block

- Matched syscalls
  - send[m]msg
  - msgsnd
  - add_key

General Kernel Exploitation

- Kernel Heap Spraying

- **Heap Spraying – msgsnd**
  - Header presence in heap memory
    - 48 byte header in front of allocated memory
    - Kernel heap pointer exists in Header

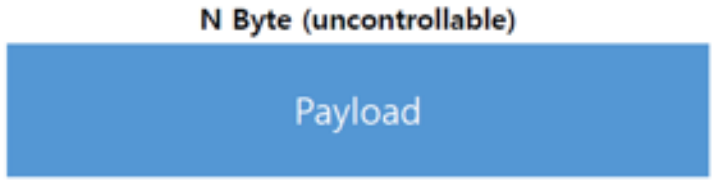
```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    int err = -EFAULT;
    size_t alen;

    msg = alloc_msg(len);
    if (msg == NULL)
        return ERR_PTR(-ENOMEM);

    alen = min(len, DATALEN_MSG);
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }

    err = security_msg_msg_alloc(msg);
    if (err)
        goto out_err;

    return msg;
```

General Kernel Exploitation

- Kernel Heap Spraying

- **Heap Spraying – send[m]msg**
  - Header presence in heap memory
    - Header before allocated memory does not exist.
    - Each socket type has a fixed size.
    - 16 bytes or more default size.

```
    ctl_buf = msg_sys->msg_control;
    ctl_len = msg_sys->msg_controllen;
} else if (ctl_len) {
    BUILD_BUG_ON(sizeof(struct cmsghdr) !=
                 CMSG_ALIGN(sizeof(struct cmsghdr)));
    if (ctl_len > sizeof(ctl)) {
        ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);
        if (ctl_buf == NULL)
            goto out_freeiov;
    }
    err = -EFAULT;
    /*
     * Careful! Before this, msg_sys->msg_control contains a user pointer.
     * Afterwards, it will be a kernel pointer. Thus the compiler-assisted
     * checking falls down on this.
     */
    if (copy_from_user(ctl_buf,
                       (void __user __force *)msg_sys->msg_control,
                       ctl_len))
        goto out_freectl;
    msg_sys->msg_control = ctl_buf;
} /* end if ctl_len */
msg_sys->msg_flags = flags;

if (sock->file->f_flags & O_NONBLOCK)
    msg_sys->msg_flags |= MSG_DONTWAIT;
```



General Kernel Exploitation

- Kernel Heap Spraying

  - **Heap Spraying – add_key**

    - Header presence in heap memory

      - 18 Byte Header in front of allocated memory
      - Kernel heap pointer exists in Header

```
/* pull the payload in if one was supplied */
payload = NULL;

if (plen) {
    ret = -ENOMEM;
    payload = kvmalloc(plen, GFP_KERNEL);
    if (!payload)
        goto error2;

    ret = -EFAULT;
    if (copy_from_user(payload, _payload, plen) != 0)
        goto error3;
}

/* find the target keyring (which must be writable) */
keyring_ref = lookup_user_key(ringid, KEY_LOOKUP_CREATE, KEY_NEED_WRITE);
if (IS_ERR(keyring_ref)) {
    ret = PTR_ERR(keyring_ref);
    goto error3;
}
```

Exploitation Practice!

Use-After-Free in EXT4 (CVE-2018-10879)


gef➤
root@exploit:~/2018-10879#
[0] 0:root@localhost* 1:qemu-system-x86_64-
"pwn" 11:17 18-10-19

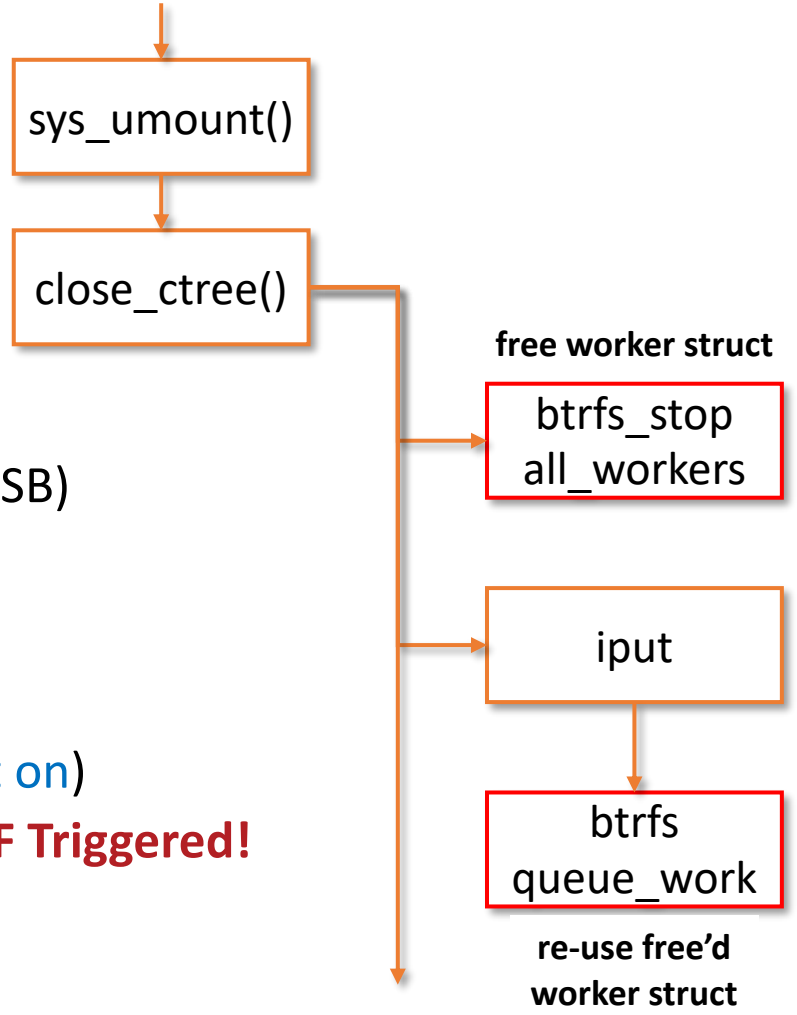
CVE-2019-19377 Exploitation

CVE-2019-19377 Exploitation

- Root cause of CVE-2019-19377

- BTRFS filesystem uses kernel's workqueue feature for async operations

1. umount starts (user call umount syscall or unplug USB)
2. close_ctree() on umount syscall
3. free workqueue in btrfs_stop_all_workers()
4. continue umount routine
5. if async option is on, follow specific routine (default on)
6. reference workqueue on btrfs_queue_work() **UAF Triggered!**



CVE-2019-19377 Exploitation

- **Exploit plan for CVE-2019-19377**

- Just re-allocate worker struct & follow the routine!

- But, life is never easy.
  - Free & Reuse of target memory happens in a flash.
  - Target structure is too small (**16byte** size)

- So, how can we exploit this?
  - Race-condition-like target memory allocation
  - Find another heap spraying technic!

```
struct btrfs_workqueue {
    struct __btrfs_workqueue *high;
    struct __btrfs_workqueue *normal;
}
```

CVE-2019-19377 Exploitation

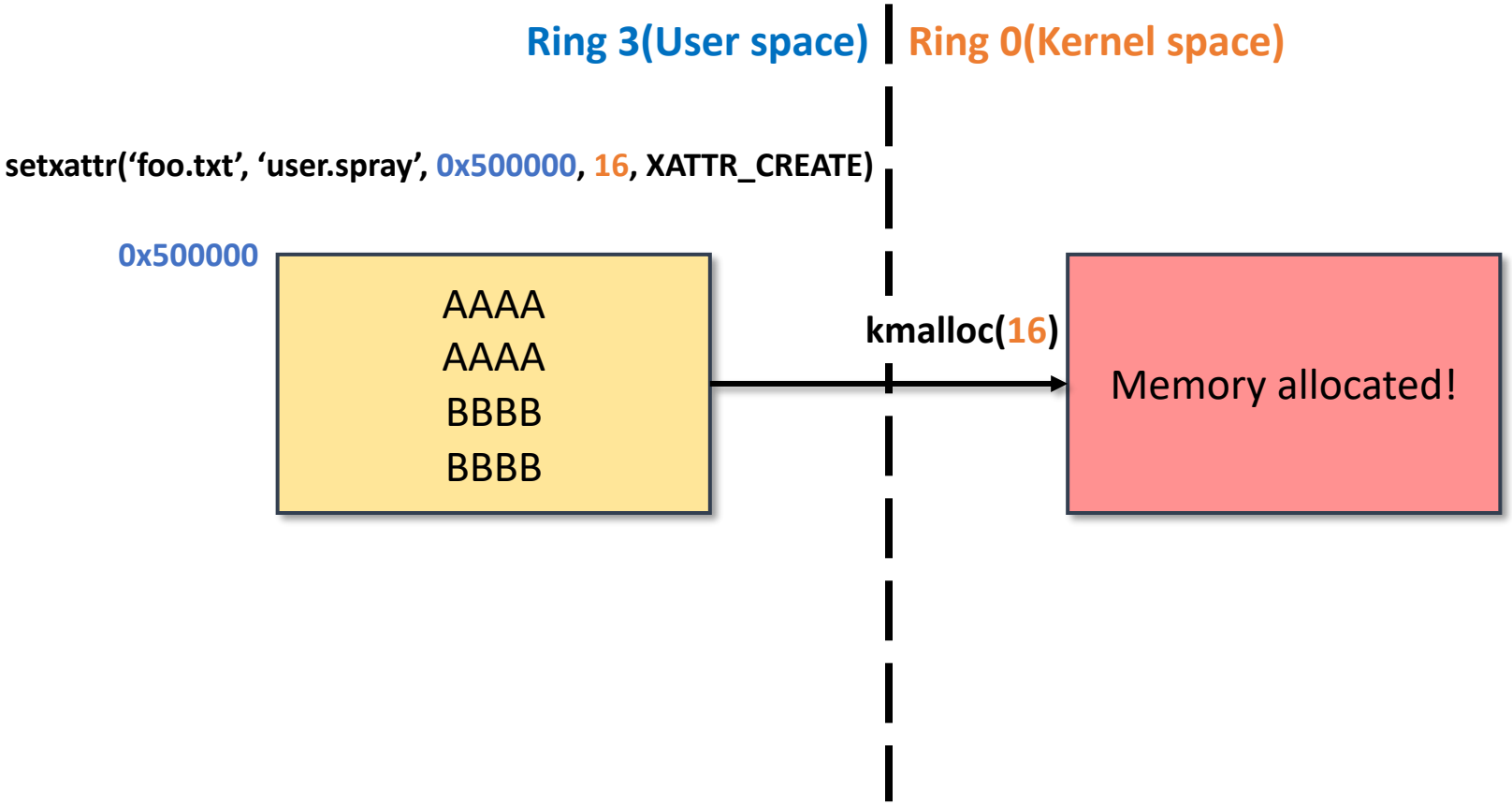
- **Find another heap spraying technic!**

- Vitaly Nikolenko saved us!
  - Dissecting a 17-years-old kernel bug(CVE-2018-6555)
  - A new heap spraying technic
    - Use **setxattr** syscall for heap spraying
    - User callable, flexible size, controllable payload, no header (Awesome!)

    - But, syscall destroys heap block on exit

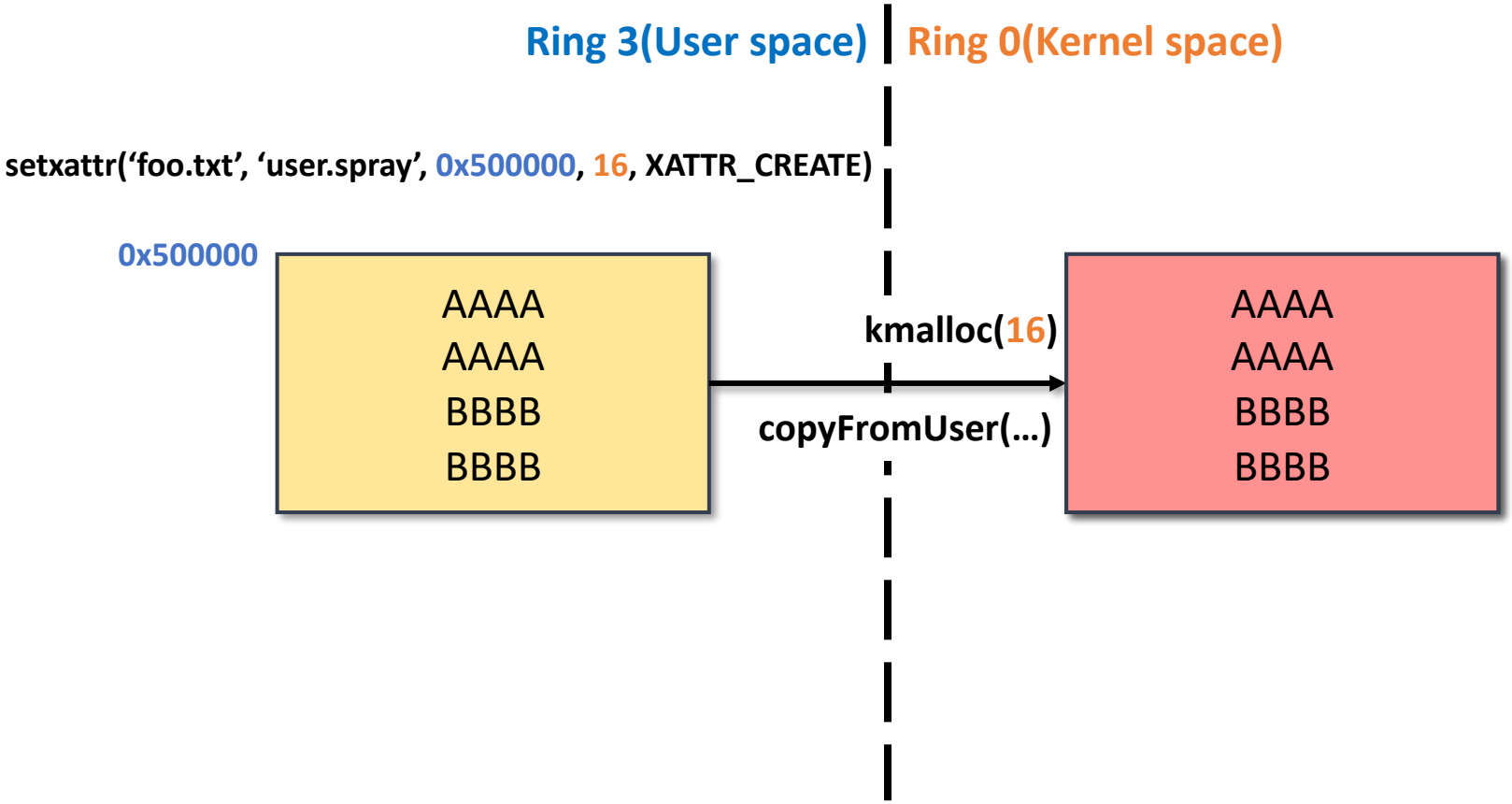
CVE-2019-19377 Exploitation

- Find another heap spraying technic!



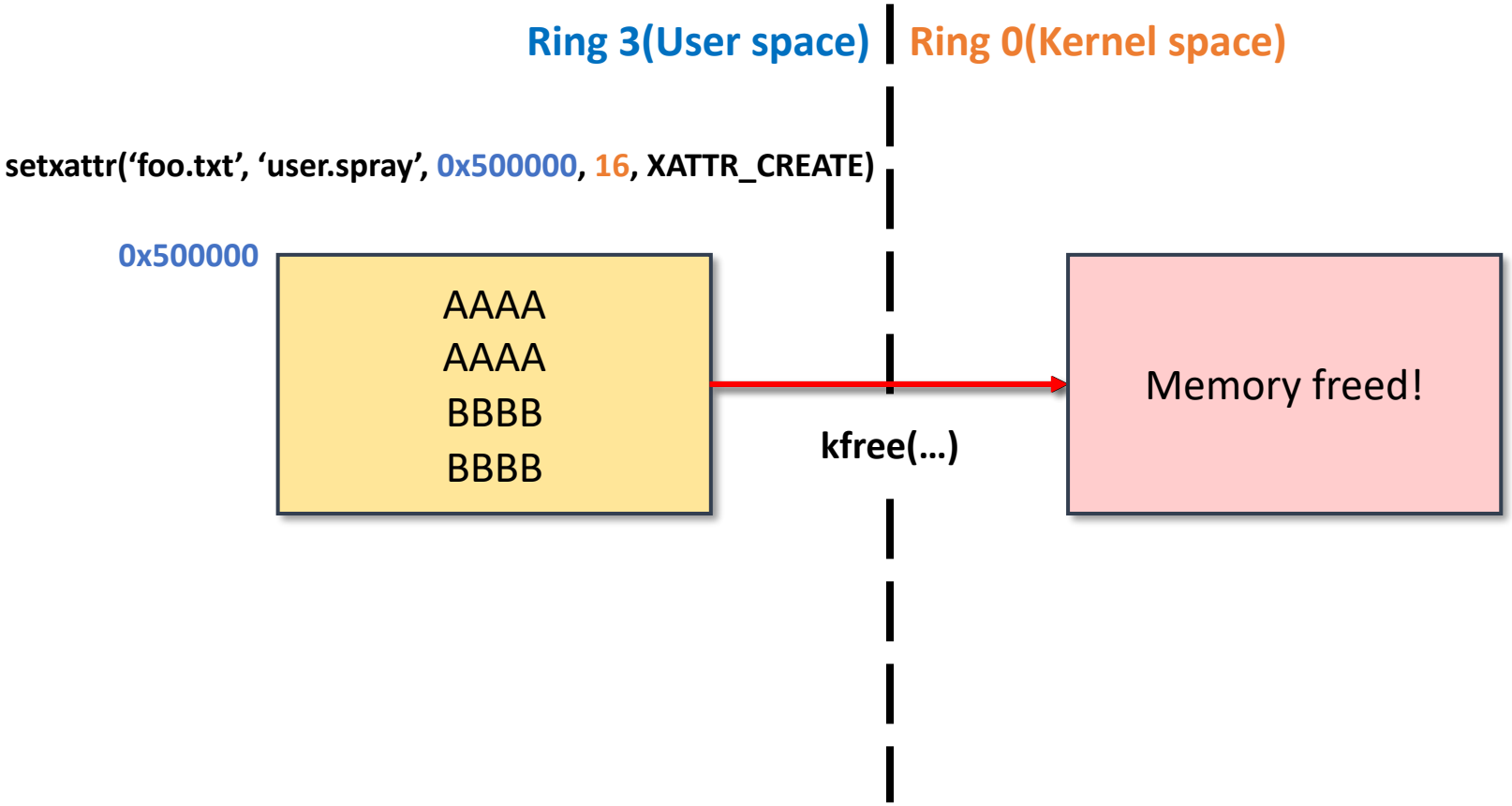
CVE-2019-19377 Exploitation

- Find another heap spraying technic!



CVE-2019-19377 Exploitation

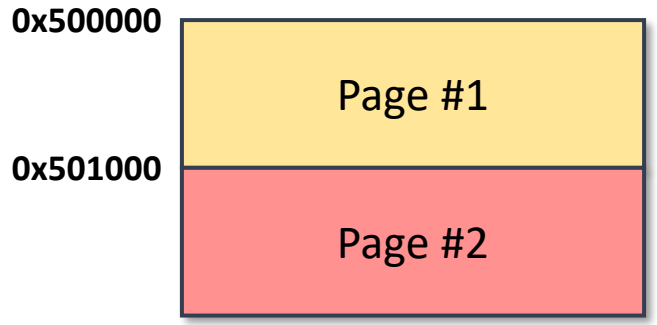
- Find another heap spraying technic!



CVE-2019-19377 Exploitation

- **Find another heap spraying technic!**

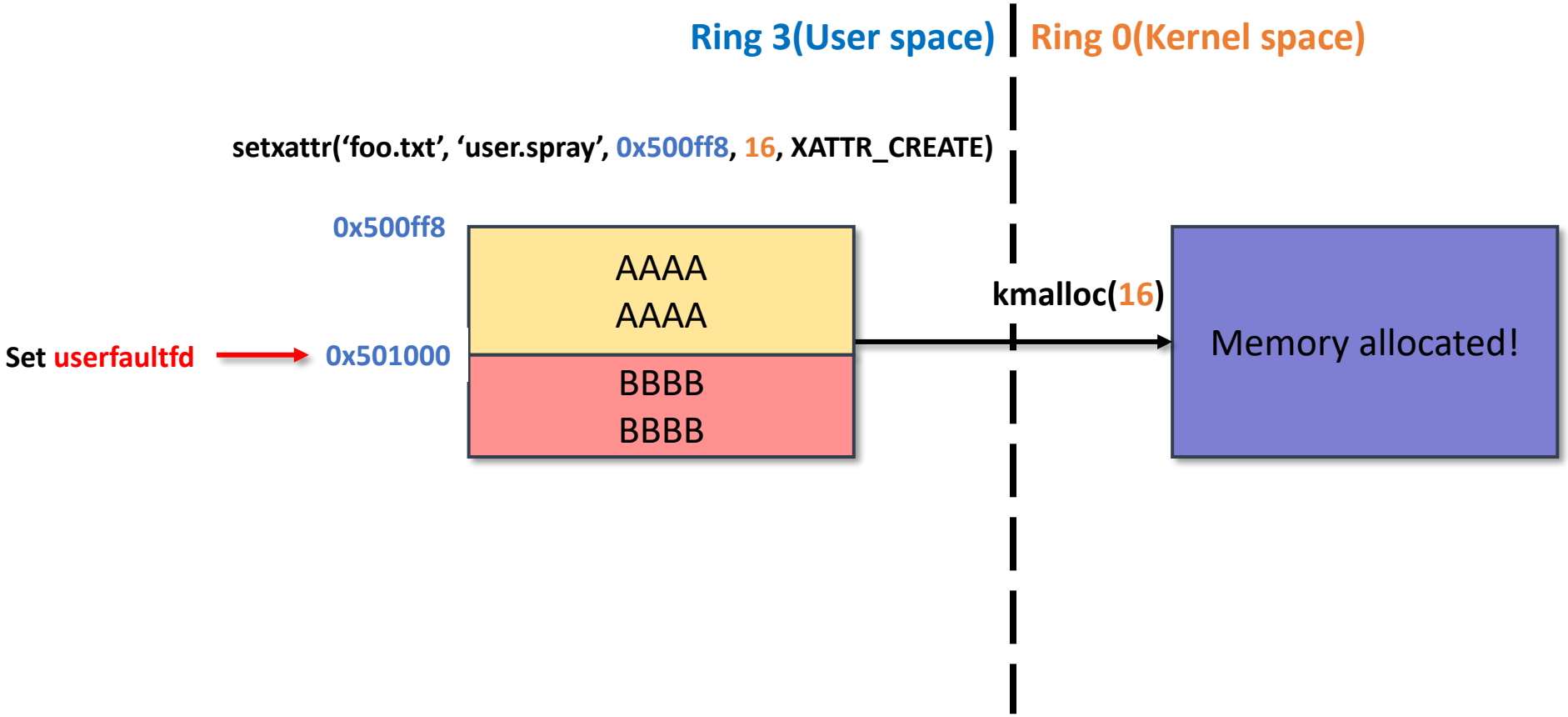
mmap(0x500000, 0x2000, …);



- setxattr + userfaultfd
  - Set userfaultfd on Page #2
    - memory read on Page #2 invokes page fault,
      call pre-registered handling function by userfaultfd
  - Call setxattr with address 0x501000 - 0xC
  - setxattr syscall reads Page #2(0x510000) and invokes page fault
  - call sleep() or use while(1) inside handling function.
    - It **delays memory free**!

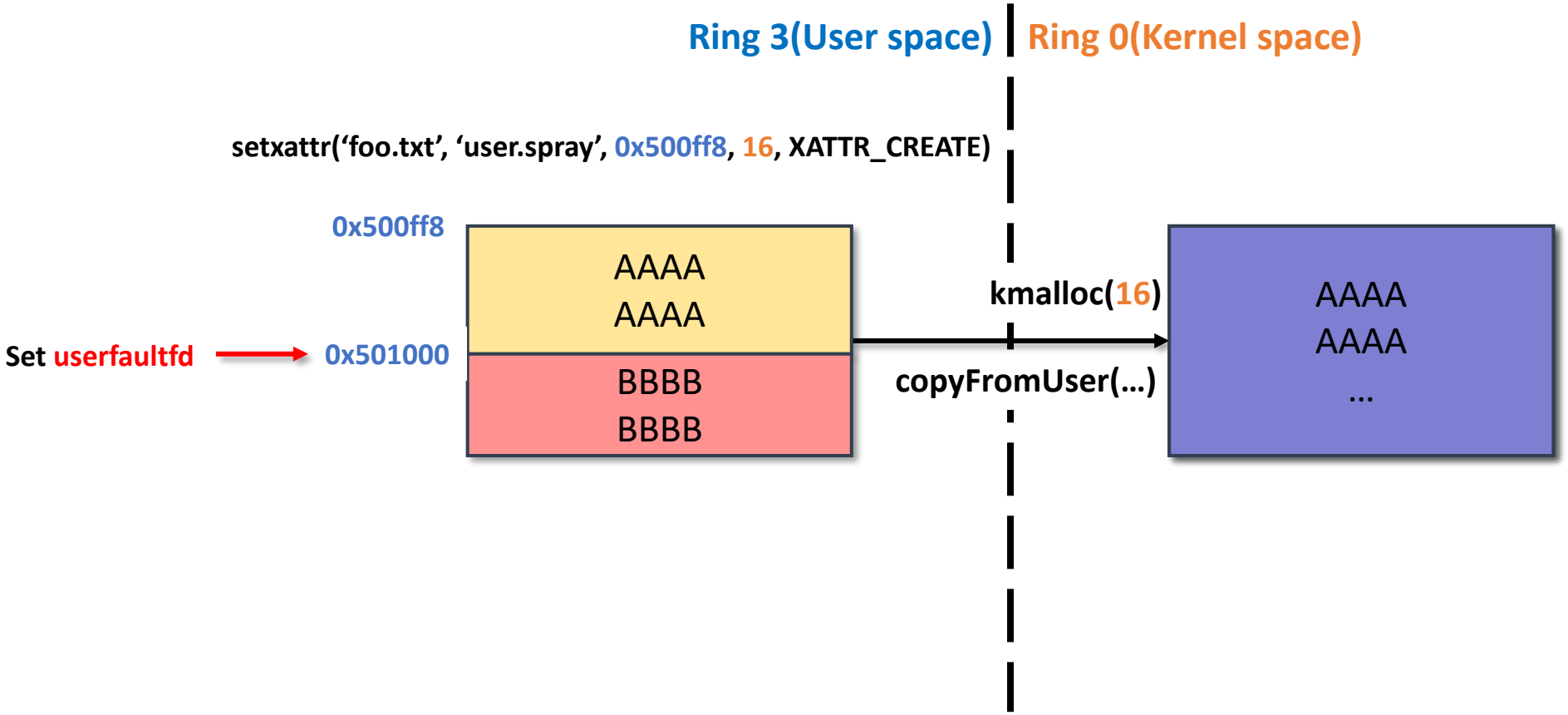
CVE-2019-19377 Exploitation

- Find another heap spraying technic!



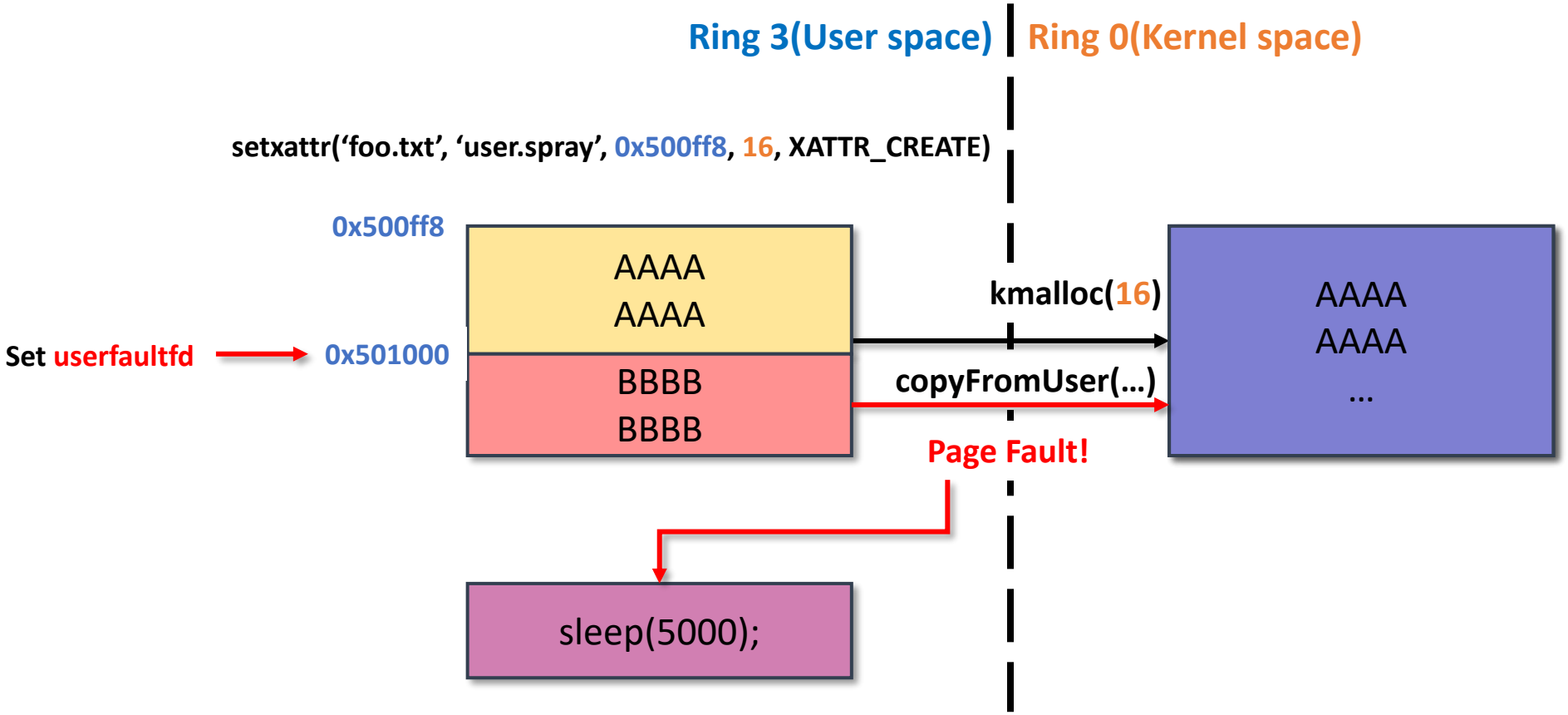
CVE-2019-19377 Exploitation

- Find another heap spraying technic!



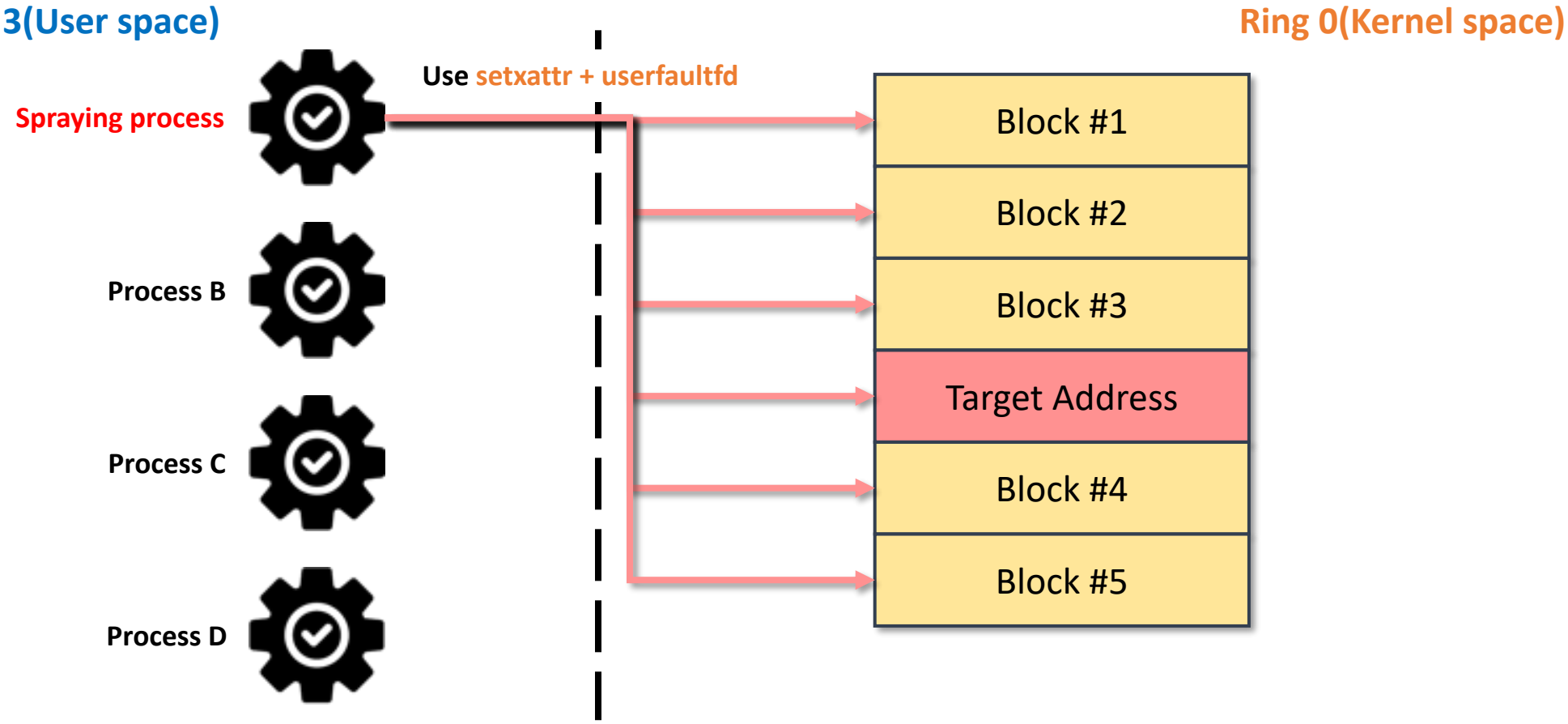
CVE-2019-19377 Exploitation

- Find another heap spraying technic!



CVE-2019-19377 Exploitation

- Find another heap spraying technic!



CVE-2019-19377 Exploitation

- RIP Handling

- Analyze following kernel routines with corrupted structure pointer

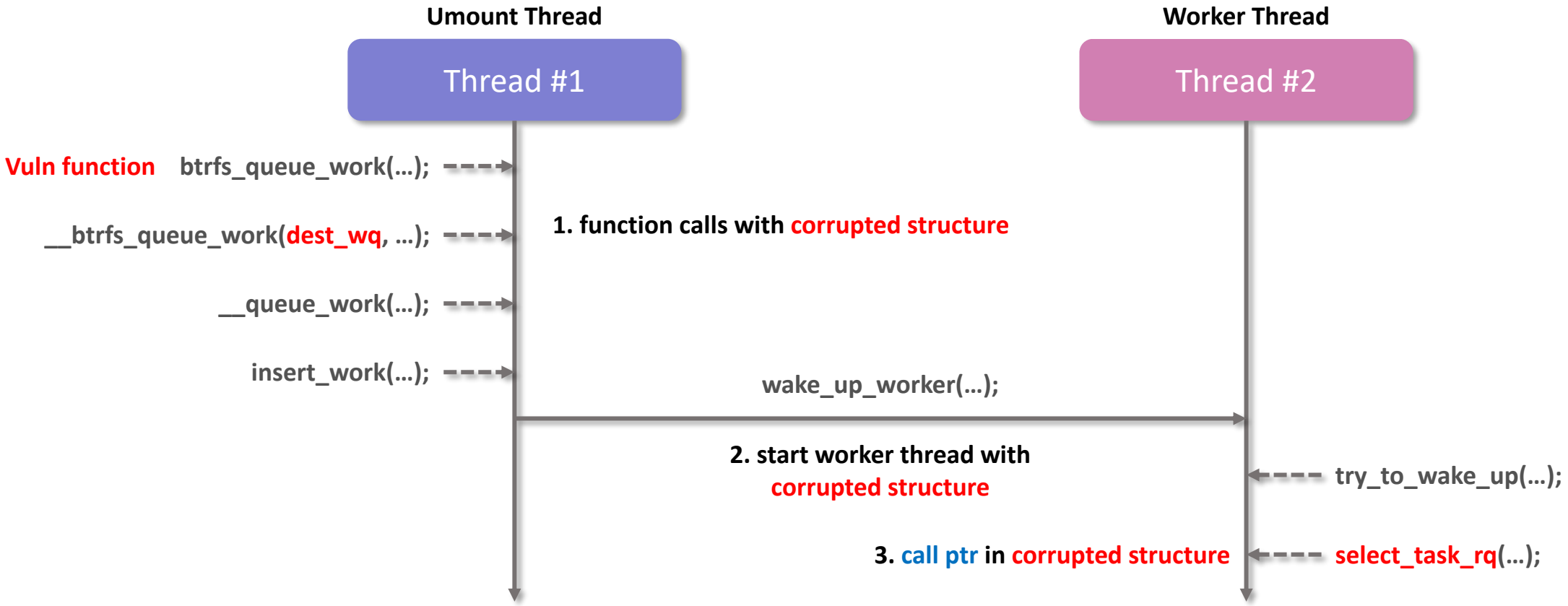
dest_wp is corrupted

Call __btrfs_queue_work with dest_wp

```
void btrfs_queue_work(struct btrfs_workqueue *wq,
                      struct btrfs_work *work)
{
  struct __btrfs_workqueue *dest_wq;

>>> if (test_bit(WORK_HIGH_PRIO_BIT, &work->flags) && wq->high)
      dest_wq = wq->high;
  else
      dest_wq = wq->normal;
  __btrfs_queue_work(dest_wq, work);
}
```

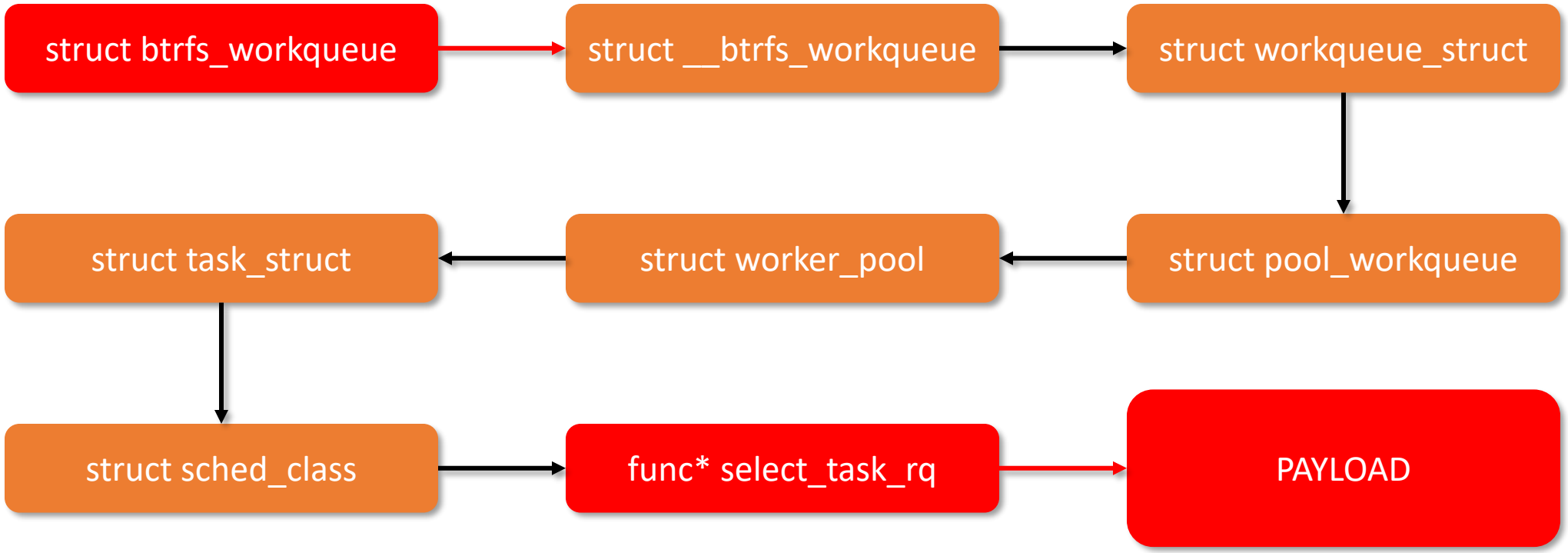
CVE-2019-19377 Exploitation

- RIP Handling (function trace)



CVE-2019-19377 Exploitation

- RIP Handling (structure trace)



CVE-2019-19377 Exploitation

- RIP Handling

```
[   59.086086] RIP: 0010:0x4242424242424242
[   59.086086] Code: Bad RIP value.
[   59.086086] RSP: 0018:ffffc900007cb968 EFLAGS: 00000002
[   59.086086] RAX: 4242424242424242 RBX: 0000000010001100 RCX: 0000000000
000000
[   59.086086] RDX: 0000000000000010 RSI: 0000000042424242 RDI: 0000000010
001100
[   59.086086] RBP: ffffc900007cb9c0 R08: 0000000000000001 R09: ffff88807a
cfaed8
[   59.086086] R10: ffff88807acfacf8 R11: 0000000000000000 R12: 0000000010
001c34
[   59.086086] R13: 0000000000000000 R14: 0000000000000046 R15: 0000000000
022600
[   59.086086] FS:  00007fef42fd9440(0000) GS:ffff88807da00000(0000) knlGS
:0000000000000000
[   59.086086] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   59.086086] CR2: 00007f392eea6020 CR3: 000000007abf6000 CR4: 0000000000
0006f0
Segmentation fault
user@exploit:~/images/exploit$
```

CVE-2019-19377 LPE DEMO

on specific environment, SM(E/A)P off

```
user@exploit:~/exploit$
```

Thank You

Donghee Kim, 20133019@kookmin.ac.kr
Wonyoung Jung, exploit@kakao.com
SeungPyo Hong, newbiepwner@kakao.com
HeoungJin Jo, gudwls10290@gmail.com

See you at HITB's Discord channel
for questions & answers!