The quantum state of Linux kernel garbage collection CVE-2021-0920 (Part I)

googleprojectzero.blogspot.com/2022/08/the-quantum-state-of-linux-kernel.html

A deep dive into an in-the-wild Android exploit

Guest Post by Xingyu Jin, Android Security Research

This is part one of a two-part guest blog post, where first we'll look at the root cause of the CVE-2021-0920 vulnerability. In the second post, we'll dive into the in-the-wild 0-day exploitation of the vulnerability and post-compromise modules.

Overview of in-the-wild CVE-2021-0920 exploits

A surveillance vendor named Wintego has developed an exploit for Linux socket syscall 0-day, CVE-2021-0920, and used it in the wild since at least November 2020 based on the earliest captured sample, until the issue was fixed in November 2021. Combined with Chrome and Samsung browser exploits, the vendor was able to remotely root Samsung devices. The fix was released with the November 2021 Android Security Bulletin, and applied to Samsung devices in Samsung's December 2021 security update.

Google's Threat Analysis Group (TAG) discovered Samsung browser exploit chains being used in the wild. TAG then performed root cause analysis and discovered that this vulnerability, CVE-2021-0920, was being used to escape the sandbox and elevate privileges. CVE-2021-0920 was reported to Linux/Android anonymously. The Google Android Security Team performed the full deep-dive analysis of the exploit.

This issue was initially discovered in 2016 by a RedHat kernel developer and disclosed in a public email thread, but the Linux kernel community did not patch the issue until it was re-reported in 2021.

Various Samsung devices were targeted, including the Samsung S10 and S20. By abusing an ephemeral race condition in Linux kernel garbage collection, the exploit code was able to obtain a use-after-free (UAF) in a kernel sk_buff object. The in-the-wild sample could effectively circumvent CONFIG_ARM64_UAO, achieve arbitrary read / write primitives and bypass Samsung RKP to elevate to root. Other Android devices were also vulnerable, but we did not find any exploit samples against them.

Text extracted from captured samples dubbed the vulnerability “quantum Linux kernel garbage collection”, which appears to be a fitting title for this blogpost.

Introduction

CVE-2021-0920 is a use-after-free (UAF) due to a race condition in the garbage collection system for SCM_RIGHTS. SCM_RIGHTS is a control message that allows unix-domain sockets to transmit an open file descriptor from one process to another. In other words, the sender transmits a file descriptor and the receiver then obtains a file descriptor from the sender. This passing of file descriptors adds complexity to reference-counting file structs. To account for this, the Linux kernel community designed a special garbage collection system. CVE-2021-0920 is a vulnerability within this garbage collection system. By winning a race condition during the garbage collection process, an adversary can exploit the UAF on the socket buffer, sk_buff object. In the following sections, we'll explain the SCM_RIGHTS garbage collection system and the details of the vulnerability. The analysis is based on the Linux 4.14 kernel.

What is SCM_RIGHTS?

Linux developers can share file descriptors (fd) from one process to another using the SCM_RIGHTS datagram with the sendmsg syscall. When a process passes a file descriptor to another process, SCM_RIGHTS will add a reference to the underlying file struct. This means that the process that is sending the file descriptors can immediately close the file descriptor on their end, even if the receiving process has not yet accepted and taken ownership of the file descriptors. When the file descriptors are in the "queued" state (meaning the sender has passed the fd and then closed it, but the receiver has not yet accepted the fd and taken ownership), specialized garbage collection is needed. To track this "queued" state, this LWN article does a great job explaining SCM_RIGHTS reference counting, and it's recommended reading before continuing on with this blogpost.

Sending

As stated previously, a unix domain socket uses the syscall sendmsg to send a file descriptor to another socket. To explain the reference counting that occurs during SCM_RIGHTS, we'll start from the sender's point of view. We start with the kernel function unix_stream_sendmsg, which implements the sendmsg syscall. To implement the SCM_RIGHTS functionality, the kernel uses the structure scm_fp_list for managing all the transmitted file structures. scm_fp_list stores the list of file pointers to be passed.

```
struct scm_fp_list {
        short                   count;
        short                   max;
        struct user_struct      *user;
        struct file             *fp[SCM_MAX_FD];
};
```

unix_stream_sendmsg invokes scm_send (af_unix.c#L1886) to initialize the scm_fp_list structure, linked by the scm_cookie structure on the stack.

```
struct scm_cookie {
        struct pid              *pid;           /* Skb credentials */
        struct scm_fp_list      *fp;            /* Passed files        */
        struct scm_creds        creds;          /* Skb credentials      */
#ifdef CONFIG_SECURITY_NETWORK
        u32                     secid;          /* Passed security ID   */
#endif
};
```

To be more specific, scm_send → __scm_send → scm_fp_copy (scm.c#L68) reads the file descriptors from the userspace and initializes scm_cookie->fp which can contain SCM_MAX_FD file structures.

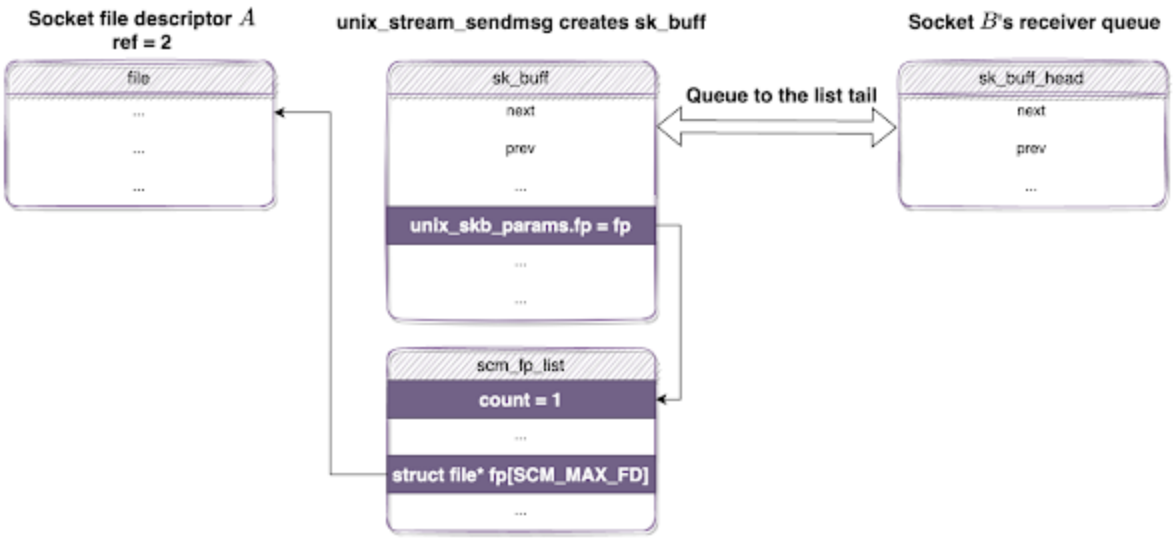
Since the Linux kernel uses the sk_buff (also known as socket buffers or skb) object to manage all types of socket datagrams, the kernel also needs to invoke the unix_scm_to_skb function to link the scm_cookie->fp to a corresponding skb object. This occurs in unix_attach_fds (scm.c#L103):

```
…
/*
 * Need to duplicate file references for the sake of garbage
 * collection.  Otherwise a socket in the fps might become a
 * candidate for GC while the skb is not yet queued.
 */
UNIXCB(skb).fp = scm_fp_dup(scm->fp);
if (!UNIXCB(skb).fp)
        return -ENOMEM;
…
```

The scm_fp_dup call in unix_attach_fds increases the reference count of the file descriptor that's being passed so the file is still valid even after the sender closes the transmitted file descriptor later:

```
struct scm_fp_list *scm_fp_dup(struct scm_fp_list *fpl)
{
        struct scm_fp_list *new_fpl;
        int i;
        if (!fpl)
                return NULL;
        new_fpl = kmemdup(fpl, offsetof(struct scm_fp_list, fp[fpl->count]),
                          GFP_KERNEL);
        if (new_fpl) {
                for (i = 0; i < fpl->count; i++)
                        get_file(fpl->fp[i]);
                new_fpl->max = new_fpl->count;
                new_fpl->user = get_uid(fpl->user);
        }
        return new_fpl;
}
```

Let's examine a concrete example. Assume we have sockets A and B. The A attempts to pass itself to B. After the SCM_RIGHTS datagram is sent, the newly allocated skb from the sender will be appended to the B's sk_receive_queue which stores received datagrams:



sk_buff carries scm_fp_list structure

The reference count of A is incremented to 2 and the reference count of B is still 1.

Receiving

Now, let's take a look at the receiver side unix_stream_read_generic (we will not discuss the MSG_PEEK flag yet, and focus on the normal routine). First of all, the kernel grabs the current skb from sk_receive_queue using skb_peek. Secondly, since scm_fp_list is attached to the skb, the kernel will call unix_detach_fds (link) to parse the transmitted file structures from skb and clear the skb from sk_receive_queue:

```
/* Mark read part of skb as used */
if (!(flags & MSG_PEEK)) {
        UNIXCB(skb).consumed += chunk;
        sk_peek_offset_bwd(sk, chunk);
        if (UNIXCB(skb).fp)
                unix_detach_fds(&scm, skb);
        if (unix_skb_len(skb))
                break;
        skb_unlink(skb, &sk->sk_receive_queue);
        consume_skb(skb);
        if (scm.fp)
                break;
```

The function scm_detach_fds iterates over the list of passed file descriptors (scm->fp) and installs the new file descriptors accordingly for the receiver:

```
for (i=0, cmfptr=(__force int __user *)CMSG_DATA(cm); i<fdmax;
    i++, cmfptr++)
{
        struct socket *sock;
        int new_fd;
```

```
        err = security_file_receive(fp[i]);
        if (err)
                break;
        err = get_unused_fd_flags(MSG_CMSG_CLOEXEC & msg->msg_flags
                                  ? O_CLOEXEC : 0);
        if (err < 0)
                break;
        new_fd = err;
        err = put_user(new_fd, cmfptr);
        if (err) {
                put_unused_fd(new_fd);
                break;
        }
        /* Bump the usage count and install the file. */
        sock = sock_from_file(fp[i], &err);
        if (sock) {
                sock_update_netprioidx(&sock->sk->sk_cgrp_data);
                sock_update_classid(&sock->sk->sk_cgrp_data);
        }
        fd_install(new_fd, get_file(fp[i]));
}
…
/*
 * All of the files that fit in the message have had their
 * usage counts incremented, so we just free the list.
 */
__scm_destroy(scm);
```

Once the file descriptors have been installed, __scm_destroy (link) cleans up the allocated scm->fp and decrements the file reference count for every transmitted file structure:

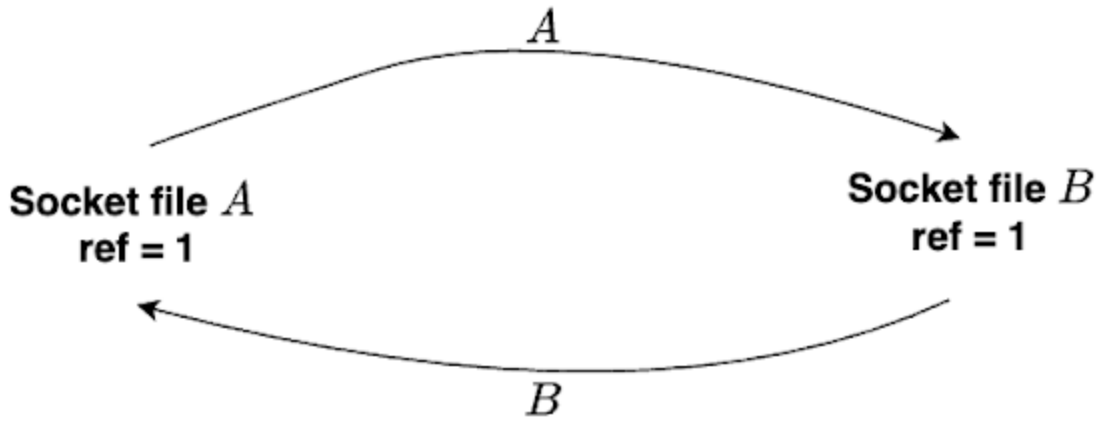
```
void __scm_destroy(struct scm_cookie *scm)
{
        struct scm_fp_list *fpl = scm->fp;
        int i;
        if (fpl) {
                scm->fp = NULL;
                for (i=fpl->count-1; i>=0; i--)
                        fput(fpl->fp[i]);
                free_uid(fpl->user);
                kfree(fpl);
        }
}
```

Reference Counting and Inflight Counting

As mentioned above, when a file descriptor is passed using SCM_RIGHTS, its reference count is immediately incremented. Once the recipient socket has accepted and installed the passed file descriptor, the reference count is then decremented. The complication comes from the “middle” of this operation: after the file descriptor has been sent, but before the receiver has accepted and installed the file descriptor.

Let’s consider the following scenario:

1. The process creates sockets A and B.
2. A sends socket A to socket B.
3. B sends socket B to socket A.
4. Close A.
5. Close B.



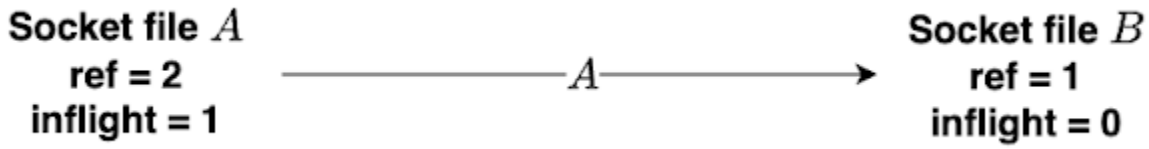
Scenario for reference count cycle

Both sockets are closed prior to accepting the passed file descriptors.The reference counts of A and B are both 1 and can't be further decremented because they were removed from the kernel fd table when the respective processes closed them. Therefore the kernel is unable to release the two skbs and sock structures and an unbreakable cycle is formed. The Linux kernel garbage collection system is designed to prevent memory exhaustion in this particular scenario. The inflight count was implemented to identify potential garbage. Each time the reference count is increased due to an SCM_RIGHTS datagram being sent, the inflight count will also be incremented.

When a file descriptor is sent by SCM_RIGHTS datagram, the Linux kernel puts its unix_sock into a global list gc_inflight_list. The kernel increments unix_tot_inflight which counts the total number of inflight sockets. Then, the kernel increments u->inflight which tracks the inflight count for each individual file descriptor in the unix_inflight function (scm.c#L45) invoked from unix_attach_fds:

```
void unix_inflight(struct user_struct *user, struct file *fp)
{
        struct sock *s = unix_get_socket(fp);
        spin_lock(&unix_gc_lock);
        if (s) {
                struct unix_sock *u = unix_sk(s);
                if (atomic_long_inc_return(&u->inflight) == 1) {
                        BUG_ON(!list_empty(&u->link));
                        list_add_tail(&u->link, &gc_inflight_list);
                } else {
                        BUG_ON(list_empty(&u->link));
                }
                unix_tot_inflight++;
        }
        user->unix_inflight++;
        spin_unlock(&unix_gc_lock);
}
```

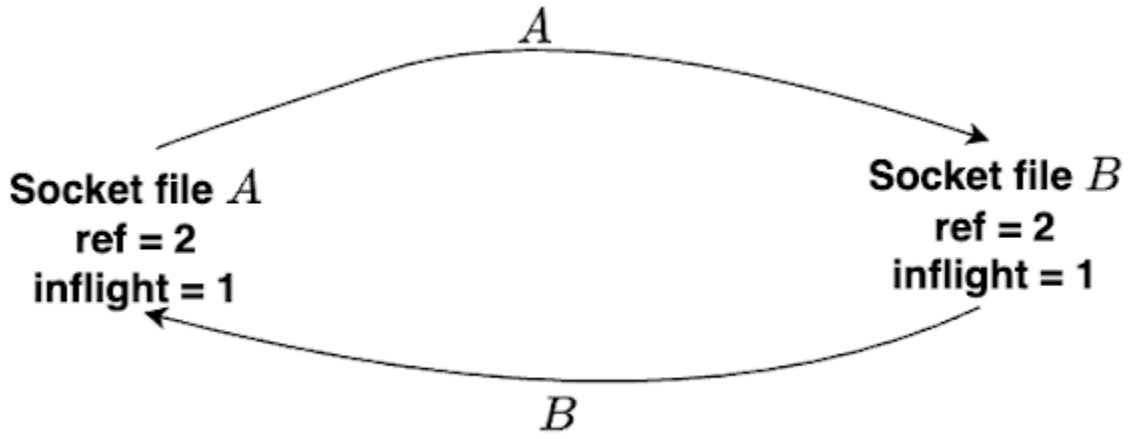
Thus, here is what the sk_buff looks like when transferring a file descriptor within sockets A and B:



The inflight count of A is incremented

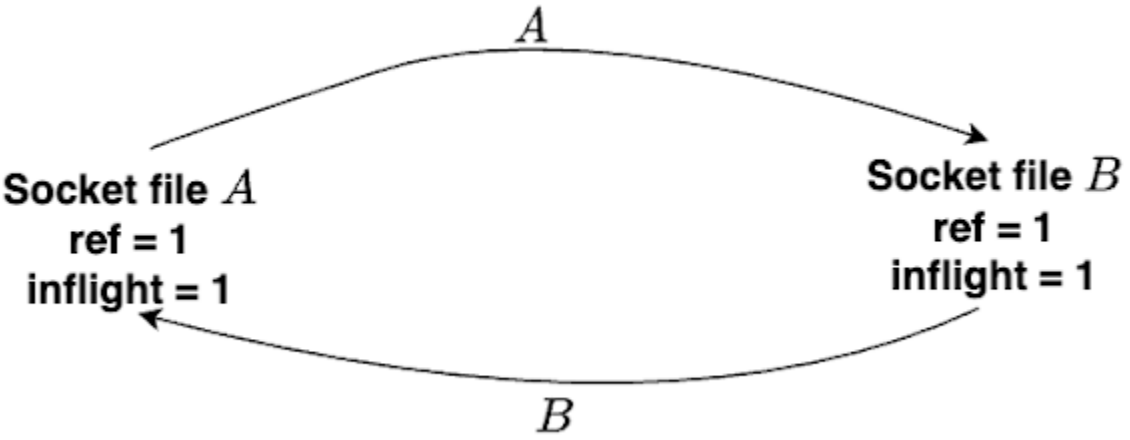
When the socket file descriptor is received from the other side, the unix_sock.inflight count will be decremented.

Let’s revisit the reference count cycle scenario before the close syscall. This cycle is breakable because any socket files can receive the transmitted file and break the reference cycle:



Breakable cycle before close A and B

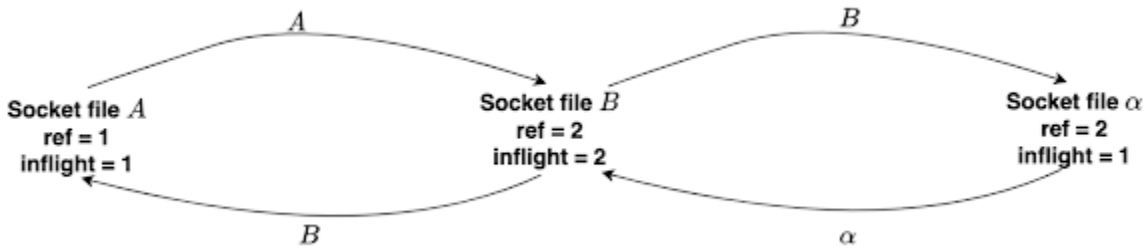
After closing both of the file descriptors, the reference count equals the inflight count for each of the socket file descriptors, which is a sign of possible garbage:



Unbreakable cycle after close A and B

Now, let’s check another example. Assume we have sockets A, B and α:

1. A sends socket A to socket B.
2. B sends socket B to socket A.
3. B sends socket B to socket α.
4. α sends socket α to socket B.
5. Close A.
6. Close B.



Breakable cycle for A, B and α

The cycle is breakable, because we can get newly installed file descriptor B’ from the socket file descriptor α and newly installed file descriptor A' from B’.

Garbage Collection

A high level view of garbage collection is available from lwn.net:

"If, instead, the two counts are equal, that file structure might be part of an unreachable cycle. To determine whether that is the case, the kernel finds the set of all in-flight Unix-domain sockets for which all references are contained in SCM_RIGHTS datagrams (for which f_count and inflight are equal, in other words). It then counts how many references to each of those sockets come from SCM_RIGHTS datagrams attached to sockets in this set. Any socket that has references coming from outside the set is reachable and can be removed from the set. If it is reachable, and if there are any SCM_RIGHTS datagrams waiting to be consumed attached to it, the files contained within that datagram are also reachable and can be removed from the set.

At the end of an iterative process, the kernel may find itself with a set of in-flight Unix-domain sockets that are only referenced by unconsumed (and unconsumable) SCM_RIGHTS datagrams; at this point, it has a cycle of file structures holding the only references to each other. Removing those datagrams from the queue, releasing the references they hold, and discarding them will break the cycle."

To be more specific, the SCM_RIGHTS garbage collection system was developed in order to handle the unbreakable reference cycles. To identify which file descriptors are a part of unbreakable cycles:

1. Add any unix_sock objects whose reference count equals its inflight count to the gc_candidates list.
2. Determine if the socket is referenced by any sockets outside of the gc_candidates list. If it is then it is reachable, remove it and any sockets it references from the gc_candidates list. Repeat until no more reachable sockets are found.
3. After this iterative process, only sockets who are solely referenced by other sockets within the gc_candidates list are left.

Let's take a closer look at how this garbage collection process works. First, the kernel finds all the unix_sock objects whose reference counts equals their inflight count and puts them into the gc_candidates list (garbage.c#L242):

```
list_for_each_entry_safe(u, next, &gc_inflight_list, link) {
        long total_refs;
        long inflight_refs;
        total_refs = file_count(u->sk.sk_socket->file);
        inflight_refs = atomic_long_read(&u->inflight);
        BUG_ON(inflight_refs < 1);
        BUG_ON(total_refs < inflight_refs);
        if (total_refs == inflight_refs) {
                list_move_tail(&u->link, &gc_candidates);
                __set_bit(UNIX_GC_CANDIDATE, &u->gc_flags);
                __set_bit(UNIX_GC_MAYBE_CYCLE, &u->gc_flags);
        }
}
```

Next, the kernel removes any sockets that are referenced by other sockets outside of the current gc_candidates list. To do this, the kernel invokes scan_children (garbage.c#138) along with the function pointer dec_inflight to iterate through each candidate's sk->receive_queue. It decreases the inflight count for each of the passed file descriptors that are themselves candidates for garbage collection (garbage.c#L261):

```
/* Now remove all internal in-flight reference to children of
 * the candidates.
 */
list_for_each_entry(u, &gc_candidates, link)
        scan_children(&u->sk, dec_inflight, NULL);
```

After iterating through all the candidates, if a gc candidate still has a positive inflight count it means that it is referenced by objects outside of the gc_candidates list and therefore is reachable. These candidates should not be included in the gc_candidates list so the related inflight counts need to be restored.

To do this, the kernel will put the candidate to not_cycle_list instead and iterates through its receiver queue of each transmitted file in the gc_candidates list (garbage.c#L281) and increments the inflight count back. The entire process is done recursively, in order for the garbage collection to avoid purging reachable sockets:

```
 /* Restore the references for children of all candidates,
  * which have remaining references.  Do this recursively, so
  * only those remain, which form cyclic references.
  *
  * Use a "cursor" link, to make the list traversal safe, even
  * though elements might be moved about.
  */
 list_add(&cursor, &gc_candidates);
 while (cursor.next != &gc_candidates) {
        u = list_entry(cursor.next, struct unix_sock, link);
        /* Move cursor to after the current position. */
        list_move(&cursor, &u->link);
        if (atomic_long_read(&u->inflight) > 0) {
                list_move_tail(&u->link, &not_cycle_list);
                __clear_bit(UNIX_GC_MAYBE_CYCLE, &u->gc_flags);
                scan_children(&u->sk, inc_inflight_move_tail, NULL);
        }
 }
 list_del(&cursor);
```

Now gc_candidates contains only “garbage”. The kernel restores original inflight counts from gc_candidates, moves candidates from not_cycle_list back to gc_inflight_list and invokes __skb_queue_purge for cleaning up garbage (garbage.c#L306).

```
 /* Now gc_candidates contains only garbage.  Restore original
  * inflight counters for these as well, and remove the skbuffs
  * which are creating the cycle(s).
  */
 skb_queue_head_init(&hitlist);
 list_for_each_entry(u, &gc_candidates, link)
        scan_children(&u->sk, inc_inflight, &hitlist);
 /* not_cycle_list contains those sockets which do not make up a
  * cycle.  Restore these to the inflight list.
  */
 while (!list_empty(&not_cycle_list)) {
        u = list_entry(not_cycle_list.next, struct unix_sock, link);
        __clear_bit(UNIX_GC_CANDIDATE, &u->gc_flags);
        list_move_tail(&u->link, &gc_inflight_list);
 }
 spin_unlock(&unix_gc_lock);
 /* Here we are. Hitlist is filled. Die. */
 __skb_queue_purge(&hitlist);
 spin_lock(&unix_gc_lock);
```

__skb_queue_purge clears every skb from the receiver queue:

```
/**
 *      __skb_queue_purge - empty a list
 *      @list: list to empty
 *
 *      Delete all buffers on an &sk_buff list. Each buffer is removed from
 *      the list and one reference dropped. This function does not take the
 *      list lock and the caller must hold the relevant locks to use it.
 */
void skb_queue_purge(struct sk_buff_head *list);
static inline void __skb_queue_purge(struct sk_buff_head *list)
{
        struct sk_buff *skb;
        while ((skb = __skb_dequeue(list)) != NULL)
                kfree_skb(skb);
}
```

There are two ways to trigger the garbage collection process:

1. wait_for_unix_gc is invoked at the beginning of the sendmsg function if there are more than 16,000 inflight sockets
2. When a socket file is released by the kernel (i.e., a file descriptor is closed), the kernel will directly invoke unix_gc.

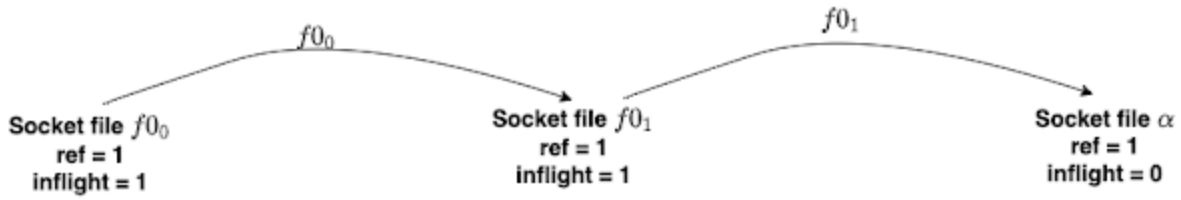
Note that unix_gc is not preemptive. If garbage collection is already in process, the kernel will not perform another unix_gc invocation.

Now, let's check this example (a breakable cycle) with a pair of sockets f00 and f01, and a single socket α:

1. Socket f00 sends socket f00 to socket f01.
2. Socket f01 sends socket f01 to socket α.
3. Close f00.
4. Close f01.

Before starting the garbage collection process, the status of socket file descriptors are:

- f 00: ref = 1, inflight = 1
- f 01: ref = 1, inflight = 1
- α: ref = 1, inflight = 0



Breakable cycle by f 00, f 01 and α

During the garbage collection process, f 00 and f 01 are considered garbage candidates. The inflight count of f 00 is dropped to zero, but the count of f 01 is still 1 because α is not a candidate. Thus, the kernel will restore the inflight count from f 01's receive queue. As a result, f 00 and f 01 are not treated as garbage anymore.

CVE-2021-0920 Root Cause Analysis

When a user receives SCM_RIGHTS message from recvmsg without the MSG_PEEK flag, the kernel will wait until the garbage collection process finishes if it is in progress. However, if the MSG_PEEK flag is on, the kernel will increment the reference count of the transmitted file structures without synchronizing with any ongoing garbage collection process. This may lead to inconsistency of the internal garbage collection state, making the garbage collector mark a non-garbage sock object as garbage to purge.

recvmsg without MSG_PEEK flag

The kernel function unix_stream_read_generic (af_unix.c#L2290) parses the SCM_RIGHTS message and manages the file inflight count when the MSG_PEEK flag is NOT set. Then, the function unix_stream_read_generic calls unix_detach_fds to decrement the inflight count. Then, unix_detach_fds clears the list of passed file descriptors (scm_fp_list) from the skb:

```
static void unix_detach_fds(struct scm_cookie *scm, struct sk_buff *skb)
{
        int i;
        scm->fp = UNIXCB(skb).fp;
        UNIXCB(skb).fp = NULL;
        for (i = scm->fp->count-1; i >= 0; i--)
                unix_notinflight(scm->fp->user, scm->fp->fp[i]);
}
```

The unix_notinflight from unix_detach_fds will reverse the effect of unix_inflight by decrementing the inflight count:

```
void unix_notinflight(struct user_struct *user, struct file *fp)
{
        struct sock *s = unix_get_socket(fp);
        spin_lock(&unix_gc_lock);
        if (s) {
                struct unix_sock *u = unix_sk(s);
                BUG_ON(!atomic_long_read(&u->inflight));
                BUG_ON(list_empty(&u->link));
                if (atomic_long_dec_and_test(&u->inflight))
                        list_del_init(&u->link);
                unix_tot_inflight--;
        }
        user->unix_inflight--;
        spin_unlock(&unix_gc_lock);
}
```

Later skb_unlink and consume_skb are invoked from unix_stream_read_generic (af_unix.c#2451) to destroy the current skb. Following the call chain kfree(skb)->__kfree_skb, the kernel will invoke the function pointer skb->destructor (code) which redirects to unix_destruct_scm:

```
static void unix_destruct_scm(struct sk_buff *skb)
{
        struct scm_cookie scm;
        memset(&scm, 0, sizeof(scm));
        scm.pid  = UNIXCB(skb).pid;
        if (UNIXCB(skb).fp)
                unix_detach_fds(&scm, skb);
        /* Alas, it calls VFS */
        /* So fscking what? fput() had been SMP-safe since the last Summer */
        scm_destroy(&scm);
        sock_wfree(skb);
}
```

In fact, the unix_detach_fds will not be invoked again here from unix_destruct_scm because UNIXCB(skb).fp is already cleared by unix_detach_fds. Finally, fd_install(new_fd, get_file(fp[i])) from scm_detach_fds is invoked for installing a new file descriptor.

recvmsg with MSG_PEEK flag

The recvmsg process is different if the MSG_PEEK flag is set. The MSG_PEEK flag is used during receive to “peek” at the message, but the data is treated as unread. unix_stream_read_generic will invoke scm_fp_dup instead of unix_detach_fds. This increases the reference count of the inflight file (af_unix.c#2149):

```
/* It is questionable, see note in unix_dgram_recvmsg.
 */
if (UNIXCB(skb).fp)
        scm.fp = scm_fp_dup(UNIXCB(skb).fp);
sk_peek_offset_fwd(sk, chunk);
if (UNIXCB(skb).fp)
        break;
```

Because the data should be treated as unread, the skb is not unlinked and consumed when the MSG_PEEK flag is set. However, the receiver will still get a new file descriptor for the inflight socket.

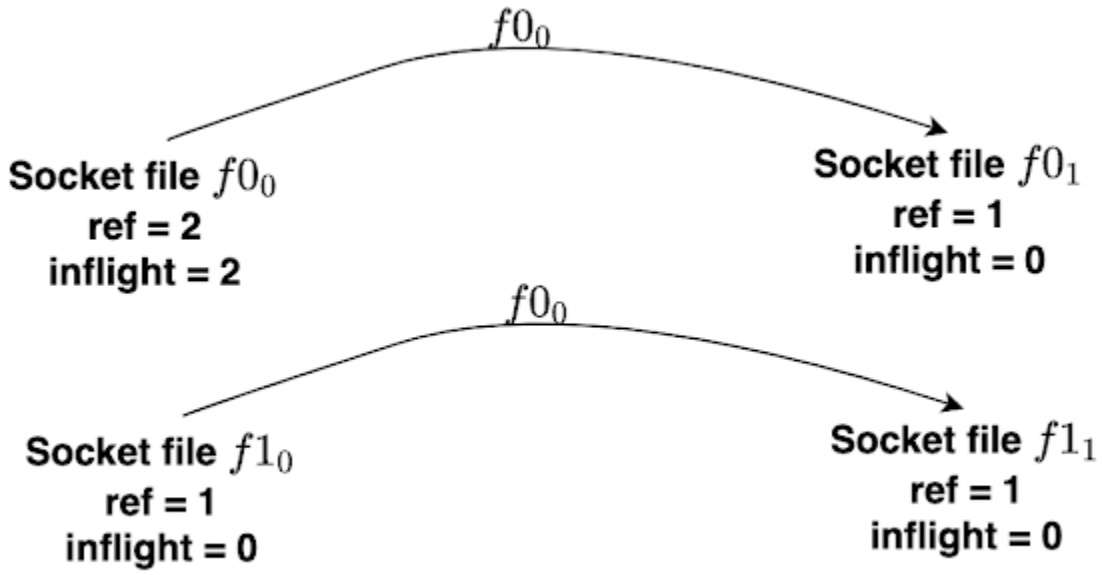
recvmsg Examples

Let's see a concrete example. Assume there are the following socket pairs:

- f 00, f 01
- f 10, f 11

Now, the program does the following operations:

- f 00 → [f 00] → f 01 (means f 00 sends [f 00] to f 01)
- f 10 → [f 00] → f 11
- Close(f 00)



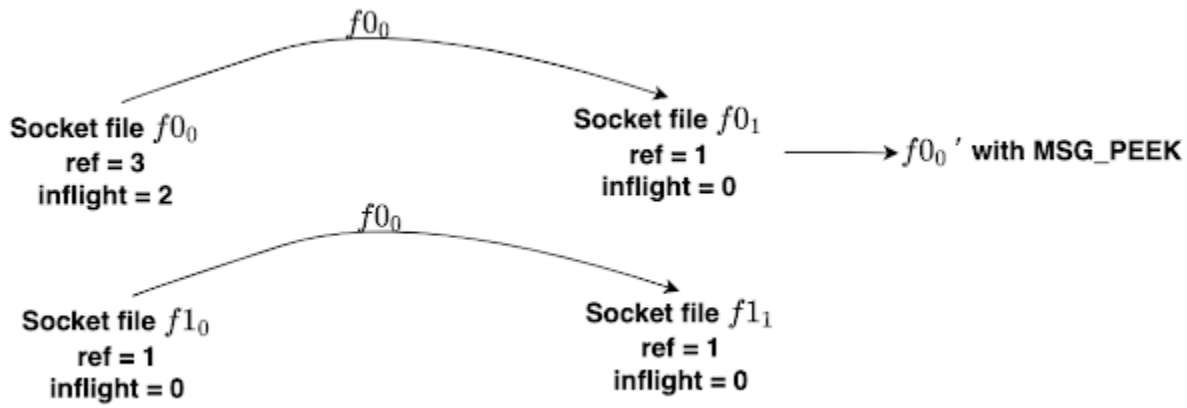
Breakable cycle by f 00, f 01, f 10 and f 11

Here is the status:

- inflight(f 00) = 2, ref(f 00) = 2
- inflight(f 01) = 0, ref(f 01) = 1
- inflight(f 10) = 0, ref(f 10) = 1
- inflight(f 11) = 0, ref(f 11) = 1

If the garbage collection process happens now, before any recvmsg calls, the kernel will choose f 00 as the garbage candidate. However, f 00 will not have the inflight count altered and the kernel will not purge any garbage.

If f 01 then calls recvmsg with MSG_PEEK flag, the receive queue doesn't change and the inflight counts are not decremented. f 01 gets a new file descriptor f 00' which increments the reference count on f 00:

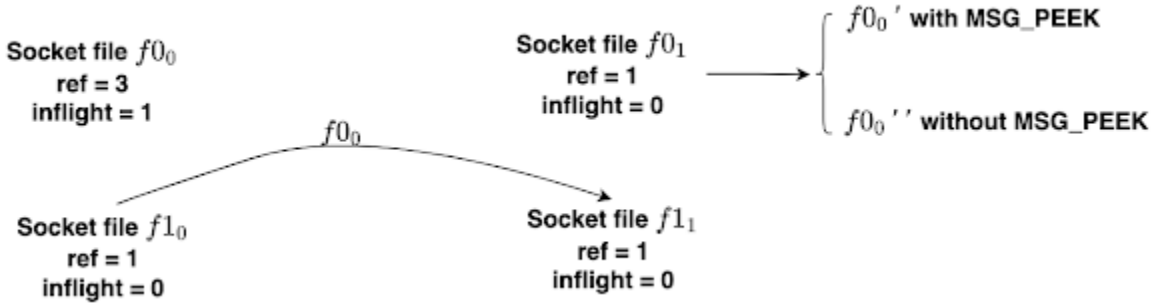


MSG_PEEK increment the reference count of f 00 while the receive queue is not cleared

Status:

- inflight(f 00) = 2, ref(f 00) = 3
- inflight(f 01) = 0, ref(f 01) = 1
- inflight(f 10) = 0, ref(f 10) = 1
- inflight(f 11) = 0, ref(f 11) = 1

Then, f 01 calls recvmsg without MSG_PEEK flag, f 01's receive queue is removed. f 01 also fetches a new file descriptor f 00'':



The receive queue of f 01 is cleared and f 01'' is obtained from f 01

Status:

- inflight(f 00) = 1, ref(f 00) = 3
- inflight(f 01) = 0, ref(f 01) = 1
- inflight(f 10) = 0, ref(f 10) = 1
- inflight(f 11) = 0, ref(f 11) = 1

UAF Scenario

From a very high level perspective, the internal state of Linux garbage collection can be non-deterministic because MSG_PEEK is not synchronized with the garbage collector. There is a race condition where the garbage collector can treat an inflight socket as a garbage candidate while the file reference is incremented at the same time during the MSG_PEEK receive. As a consequence, the garbage collector may purge the candidate, freeing the socket buffer, while a receiver may install the file descriptor, leading to a UAF on the skb object.

Let's see how the captured 0-day sample triggers the bug step by step (simplified version, in reality you may need more threads working together, but it should demonstrate the core idea). First of all, the sample allocates the following socket pairs and single socket α:

- f 00, f 01

- f 10, f 11
- f 20, f 21
- f 30, f 31
- sock α (actually there might be even thousands of α for protracting the garbage collection process in order to evade a BUG_ON check which will be introduced later).

Now, the program does the below operations:

- $f2_0 \rightarrow [f1_1] \rightarrow f2_1$
- $f1_0 \rightarrow [f1_0] \rightarrow f1_1$
- $f0_0 \rightarrow [f1_0] \rightarrow f0_1$
- $f0_1 \rightarrow [f1_0] \rightarrow f0_0$
- $f1_0 \rightarrow \left[\sum_0^N f0_0\right] \rightarrow f1_0\ (Sending\ N\ f0_0)$
- $f1_1 \rightarrow [f0_1] \rightarrow f1_0$
- $f0_1 \rightarrow [f0_0] \rightarrow f0_0$
- $f0_0 \rightarrow [f0_1] \rightarrow f0_1$
- $f1_1 \rightarrow [f3_1] \rightarrow f0_1$
- $f3_0 \rightarrow [\alpha] \rightarrow f3_1$

Close the following file descriptors prior to any recvmsg calls:

- Close(f 00)
- Close(f 01)
- Close(f 11)
- Close(f 10)
- Close(f 30)
- Close(f 31)
- Close(α)

Here is the status:

- inflight(f 00) = N + 1, ref(f 00) = N + 1
- inflight(f 01) = 2, ref(f 01) = 2
- inflight(f 10) = 3, ref(f 10) = 3
- inflight(f 11) = 1, ref(f 11) = 1
- inflight(f 20) = 0, ref(f 20) = 1
- inflight(f 21) = 0, ref(f 21) = 1
- inflight(f 31) = 1, ref(f 31) = 1
- inflight(α) = 1, ref(α) = 1

If the garbage collection process happens now, the kernel will do the following scrutiny:

- List f 00, f 01, f 10, f 11, f 31, α as garbage candidates. Decrease inflight count for the candidate children in each receive queue.
- Since f 21 is not considered a candidate, f 11's inflight count is still above zero.
- Recursively restore the inflight count.
- Nothing is considered garbage.

A potential skb UAF by race condition can be triggered by:

1. Call recvmsg with MSG_PEEK flag from f 21 to get f 11'.
2. Call recvmsg with MSG_PEEK flag from f 11 to get f 10'.
3. Concurrently do the following operations:

1. Call recvmsg without MSG_PEEK flag from f 11 to get f 10''.
2. Call recvmsg with MSG_PEEK flag from f 10'

How is it possible? Let's see a case where the race condition is not hit so there is no UAF:

Thread 0	Thread 1	Thread 2
Call unix_gc		
Stage0: List f 00, f 01, f 10, f 11, f 31, α as garbage candidates.		
	Call recvmsg with MSG_PEEK flag from f 21 to get f 11'	

	Increase reference count: scm.fp = scm_fp_dup(UNIXCB(skb).fp);	
Stage0: decrease inflight count from the child of every garbage candidate		
Status after stage 0: inflight(f 00) = 0 inflight(f 01) = 0 inflight(f 10) = 0 inflight(f 11) = 1 inflight(f 31) = 0 inflight(α) = 0		
Stage1: Recursively restore inflight count if a candidate still has inflight count.		
Stage1: All inflight counts have been restored.		
Stage2: No garbage, return.		
	Call recvmsg with MSG_PEEK flag from f 11 to get f 10’	
		Call recvmsg without MSG_PEEK flag from f 11 to get f 10’’
	Call recvmsg with MSG_PEEK flag from f 10’	
Everyone is happy	Everyone is happy	Everyone is happy

However, if the second recvmsg occurs just after stage 1 of the garbage collection process, the UAF is triggered:

Thread 0	Thread 1	Thread 2
Call unix_gc		
Stage0: List f 00, f 01, f 10, f 11, f 31, α as garbage candidates.		
	Call recvmsg with MSG_PEEK flag from f 21 to get f 11’	
	Increase reference count: scm.fp = scm_fp_dup(UNIXCB(skb).fp);	
Stage0: decrease inflight count from the child of every garbage candidates		
Status after stage 0: inflight(f 00) = 0 inflight(f 01) = 0 inflight(f 10) = 0 inflight(f 11) = 1 inflight(f 31) = 0 inflight(α) = 0		
Stage1: Start restoring inflight count.		
	Call recvmsg with MSG_PEEK flag from f 11 to get f 10’	

		Call recvmsg without MSG_PEEK flag from f 11 to get f 10''
		unix_detach_fds: UNIXCB(skb).fp = NULL
		Blocked by spin_lock(&unix_gc_lock)
Stage1: scan_inflight cannot find candidate children from f 11. Thus, the inflight count accidentally remains the same.		
Stage2: f 00, f 01, f 10, f 31, α are garbage.		
Stage2: start purging garbage.		
	Start calling recvmsg with MSG_PEEK flag from f 10', which would expect to receive f 00'	
	Get skb = skb_peek(&sk->sk_receive_queue), skb is going to be freed by thread 0.	
Stage2: for $f1_0 \rightarrow \left[\sum_0^N f0_0\right] \rightarrow f1_0$, calls __skb_unlink and kfree_skb later.		
	state->recv_actor(skb, skip, chunk, state) UAF	

GC finished.
Start garbage collection.
Get f 10”

Therefore, the race condition causes a UAF of the skb object. At first glance, we should blame the second recvmsg syscall because it clears skb.fp, the passed file list. However, if the first recvmsg syscall doesn’t set the MSG_PEEK flag, the UAF can be avoided because unix_notinflight is serialized with the garbage collection. In other words, the kernel makes sure the garbage collection is either not processed or finished before decrementing the inflight count and removing the skb. After unix_notinflight, the receiver obtains f11' and inflight sockets don't form an unbreakable cycle.

Since MSG_PEEK is not serialized with the garbage collection, when recvmsg is called with MSG_PEEK set, the kernel still considers f 11 as a garbage candidate. For this reason, the following next recvmsg will eventually trigger the bug due to the inconsistent state of the garbage collection process.

Patch Analysis

CVE-2021-0920 was found in 2016

The vulnerability was initially reported to the Linux kernel community in 2016. The researcher also provided the correct patch advice but it was not accepted by the Linux kernel community:

David Miller

```
From: Nikolay Borisov <kernel@kyup.com>
Date: Tue, 27 Sep 2016 17:16:27 +0300

> What's the status of https://patchwork.ozlabs.org/patch/664062/ , is
> this going to be picked up ?

Why would I apply a patch that's an RFC, doesn't have a proper commit
message, lacks a proper signoff, and also lacks ACK's and feedback
from other knowledgable developers?
```

Patch was not applied in 2016

In theory, anyone who saw this patch might come up with an exploit against the faulty garbage collector.

Patch in 2021

Let's check the official patch for CVE-2021-0920. For the MSG_PEEK branch, it requests the garbage collection lock unix_gc_lock before performing sensitive actions and immediately releases it afterwards:

```
…
+       spin_lock(&unix_gc_lock);
+       spin_unlock(&unix_gc_lock);
…
```

The patch is confusing - it's rare to see such lock usage in software development. Regardless, the MSG_PEEK flag now waits for the completion of the garbage collector, so the UAF issue is resolved.

BUG_ON Added in 2017

Andrey Ulanov from Google in 2017 found another issue in unix_gc and provided a fix commit. Additionally, the patch added a BUG_ON for the inflight count:

```
void unix_notinflight(struct user_struct *user, struct file *fp)
        if (s) {
                struct unix_sock *u = unix_sk(s);

+                BUG_ON(!atomic_long_read(&u->inflight));
                BUG_ON(list_empty(&u->link));

                if (atomic_long_dec_and_test(&u->inflight))
```

At first glance, it seems that the BUG_ON can prevent CVE-2021-0920 from being exploitable. However, if the exploit code can delay garbage collection by crafting a large amount of fake garbage, it can waive the BUG_ON check by heap spray.

New Garbage Collection Discovered in 2021

CVE-2021-4083 deserves an honorable mention: when I discussed CVE-2021-0920 with Jann Horn and Ben Hawkes, Jann found another issue in the garbage collection, described in the Project Zero blog post Racing against the clock -- hitting a tiny kernel race window.

\

Part I Conclusion

To recap, we have discussed the kernel internals of SCM_RIGHTS and the designs and implementations of the Linux kernel garbage collector. Besides, we have analyzed the behavior of MSG_PEEK flag with the recvmsg syscall and how it leads to a kernel UAF by a subtle and arcane race condition.

The bug was spotted in 2016 publicly, but unfortunately the Linux kernel community did not accept the patch at that time. Any threat actors who saw the public email thread may have a chance to develop an LPE exploit against the Linux kernel.

In part two, we'll look at how the vulnerability was exploited and the functionalities of the post compromise modules.