Date: Thu, 10 Feb 2022 14:16:41 +0000
From: Samuel Page <samuel.page@...gate.com>
To: "oss-security@...ts.openwall.com" <oss-security@...ts.openwall.com>
Subject: CVE-2022-0435: Remote Stack Overflow in Linux Kernel TIPC Module since 4.8 (net/tipc)

Immunity Security Advisory

CVE-2022-0435: A Remote Stack Overflow in The Linux Kernel

Contents

Summary

We discovered a remotely & locally reachable stack overflow in the Linux kernel networking module for the Transparent Inter-Process Communication (TIPC) protocol.

While the module can be found in most major distributions, it must be loaded in order to be exploited. Furthermore, for remote exploitation the target would need to have a TIPC bearer set up already i.e. vulnerability extends to systems using TIPC.

Exploitation is trivial and can lead to denial of service via kernel panic. In the absence, or bypass, of stack canaries/KASLR the vulnerability can lead to control flow hijacking with an arbitrary payload.

The vulnerability has been present since the introduction of the TIPC monitoring framework in kernel version 4.8.

- Introduced: commit 35c55c9877f8 ("tipc: add neighbor monitoring framework")
- Fixed: https://github.com/torvalds/linux/commit/9aa422ad326634b76309e8ff342c246800621216

Analysis

Transparent Inter Process Communication (TIPC) is an IPC mechanism designed for intra-cluster communication. Cluster topology is managed around the concept of nodes and the links between these nodes.

One of the many features of the TIPC module is its monitoring framework. Introduced into the kernel in June 2016 (commit 35c55c9), the framework allows nodes to monitor network topology and share their view with other nodes in the same domain.

Peer state is tracked via `struct tipc_peer`:

...

```
    /* struct tipc_peer: state of a peer node and its domain
     * @addr: tipc node identity of peer
     * @head_map: shows which other nodes currently consider peer 'up'
     * @domain: most recent domain record from peer
     * @hash: position in hashed lookup list
     * @list: position in linked list, in circular ascending order by 'addr'
     * @applied: number of reported domain members applied on this monitor list
     * @is_up: peer is up as seen from this node
     * @is_head: peer is assigned domain head as seen from this node
     * @is_local: peer is in local domain and should be continuously monitored
     * @down_cnt: - numbers of other peers which have reported this on lost
     */
    struct tipc_peer {
        u32 addr;
        struct tipc_mon_domain *domain;
        struct hlist_node hash;
        struct list_head list;
        u8 applied;
        u8 down_cnt;
        bool is_up;
        bool is_head;
        bool is_local;
    };
...
```

`struct tipc_mon_domain` referends a domain record, used to define that peers view of the TIPC topology:

```
...
    #define MAX_MON_DOMAIN       64
    ...

    /* struct tipc_mon_domain: domain record to be transferred between peers
     * @len: actual size of domain record
     * @gen: current generation of sender's domain
     * @ack_gen: most recent generation of self's domain acked by peer
     * @member_cnt: number of domain member nodes described in this record
     * @up_map: bit map indicating which of the members the sender considers up
     * @members: identity of the domain members
     */
    struct tipc_mon_domain {
        u16 len;
        u16 gen;
        u16 ack_gen;
        u16 member_cnt;
        u64 up_map;
        u32 members[MAX_MON_DOMAIN];
    };
...
```

These records are transferred between peers, with each node keeping a copy of the most up-to-date domain record received from each of its peers in the `tipc_peer->domain` field.

Records are processed by the function `tipc_mon_rcv`, which check `STATE_MSG` received from peers, to see if the message body contains a valid `struct tipc_mon_domain`:

```
...
    /* tipc_mon_rcv - process monitor domain event message
     *
     * @data: STATE_MSG body
     * @dlen: STATE_MSG body size (taken from TIPC header)
     */
    void tipc_mon_rcv(struct net *net, void *data, u16 dlen, u32 addr,
        struct tipc_mon_state *state, int bearer_id)
    {
        ...
        struct tipc_mon_domain *arrv_dom = data;
        struct tipc_mon_domain dom_bef;
        ...

        /* Sanity check received domain record */                       [0]
        if (dlen < dom_rec_len(arrv_dom, 0))                            [1]
            return;
```

```
        if (dlen != dom_rec_len(arrv_dom, new_member_cnt))              [2]
            return;
        if (dlen < new_dlen || arrv_dlen != new_dlen)                   [3]
            return;
        ...

        /* Drop duplicate unless we are waiting for a probe response */
        if (!more(new_gen, state->peer_gen) && !probing)                [4]
            return;
        ...

        /* Cache current domain record for later use */
        dom_bef.member_cnt = 0;
        dom = peer->domain;
        if (dom)                                                        [5]
            memcpy(&dom_bef, dom, dom->len);                            [6]

        /* Transform and store received domain record */
        if (!dom || (dom->len < new_dlen)) {
            kfree(dom);
            dom = kmalloc(new_dlen, GFP_ATOMIC);                        [7]
            peer->domain = dom;
            if (!dom)
                goto exit;
        }
...
```

The function does some basic sanity checks [0] to make sure that a) the message body actually contains a domain record and b) does it contain a valid `struct tipc_mon_domain`.

Where `data` is the message body and `dlen` is the length of the `data` taken from the message header, the function checks:

- the length of `data` is enough to at least hold an empty record [1]
- the length of `data` matches the expected size of a domain record
  given the provided `member_cnt` field [2]
- the length of `data` matches the provided `len` field [3]

Later we fetch the sending peers `struct peer` to see if we've already received a domain record from them [5]. If we have, we want to temporarily cache a copy of the old record to do a comparison later [6].

Then, satisfied it's a new and valid record, we'll update the `struct peer->domain` field with the new info. If it's the first domain record, we'll make a new `kmalloc`ation for this [7], or if it's larger than the last one we'll reallocate it.

The Vulnerability

The vulnerability lies in the fact that during the initial sanity checks, the function doesn't check that `member_cnt` is below MAX_MON_DOMAIN which defines the maximum size of the `members` array.

By pretending to be a peer node and establishing a link with the target, locally or remotely, we're able to first submit a malicious domain record containing an arbitrary payload; so long as the len/member_cnt fields match up for the sanity checks, this will be kmallocated fine.

Next, we can send a newer domain record which will cause the previous malicious record to be memcpy'd into a 272 bytes local `struct tipc_mon_domain` &dom_bef [6] triggering a stack overflow.

This allows us to overwrite the contents of the stack following &dom_bef with our arbitrary members buffer from the malicious domain record submitted first; the size of which is constrained by the media MTU (Ethernet, UDP, Inifiband)

Further Information

For a more in-depth analysis, including additional TIPC background, we
will be posting up a follow-up blog to compliment this advisory.

Regarding exploitation, after time has been given for patching and
remediation we will also release a further post discussing exploitation.

Further information will be posted over at https://twitter.com/immunityinc/

Remediation

This vulnerability has been present since the monitoring framework was
first introduced in June 2016, impacting versions 4.8 forward.

The patch below was introduced in commit 9aa422ad3266, so updating your
system to include this patch is the best way to mitigate CVE-2022-0435,
including an additional u16 overflow spotted by Eric Dumazet.

The TIPC module must be loaded for the system to be vulnerable,
furthermore to be targeted remotely the system needs to have a TIPC
bearer enabled. If you don't need to use TIPC or are unsure if you are,
you can take the following steps:

- `$ lsmod | grep tipc` will let you know if the module is currently
  loaded,
- `modprobe -r tipc` may allow you unload the module if loaded,
  however you may need to reboot your system
- `$ echo "install tipc /bin/true" >> /etc/modprobe.d/disable-tipc.conf`
  will prevent the module from being loaded, which is a good idea if you
  have no reason to use it

If you need to use TIPC and can't immediately patch your system, look to
enforce any configurtions that prevent or limit the ability for
attackers to imitate nodes in your cluster. Options include TIPC
protocol level encryption, IPSec/MACSec, network seperation etc.

It's also worth noting that the current `CONFIG_FORTIFY_SRC=y` is a hard
mitigation to leveraging CVE-2022-0435 for control-flow hijacking, as it
does a bounds check on the size of the offending memcpy and causes a
kernel panic.

Acknowledgements

I'd like to thank the maintainers of the TIPC module, as well as members
of the security@...nel.org and linux-distros@...openwall.org for their
help and work throughout the disclosure process.

I would also like to mention the vulnerability research done on TIPC
previously by SentinelLabs with their work on CVE-2021-43267 and other
security researchers for publishing their findings on TIPC exploitation.

- https://www.sentinelone.com/labs/tipc-remote-linux-kernel-heap-overflow-allows-arbitrary-code-execution/
- https://haxx.in/posts/pwning-tipc/

Disclosure Timeline

- 27/01/2022: Vulnerability & fix suggestion reported
- 05/02/2022: The patch is finalised
- 10/02/2022: Coordinated release date (14:00 GMT)

Please check out the Open Source Software Security Wiki, which is counterpart to this mailing list.

Confused about mailing lists and their use? Read about mailing lists on Wikipedia and check out these guidelines on proper formatting of your messages.