

Vulnerabilities in the proc Component of the CAN BCM Protocol in the Linux kernel

Analysis and vulnerability discovery

Anderson Nascimento, 307 Temporary Security Conference.

Agenda

- About me
- Introduction to the CAN BCM protocol
- Vulnerabilities
 - Race condition in `bcm_connect()` leading to use-after-free read
 - Use-after-free read in `bcm_proc_show()` (CVE-2023-52922)
 - Use-after-free read in `bcm_proc_show()` due to missing rcu read protection (CVE-2025-38003) and out-of-bounds read in `bcm_can_tx()` (CVE-2025-38004)
- Conclusion

About me

About me

- Anderson Nascimento
 - Director and Principal Security Researcher at Allele Security Intelligence
 - Linux and Android security researcher with over 10 years of experience in vulnerability research.
 - I have discovered and exploited several vulnerabilities in the past.
 - Instructor of Linux binary and kernel exploitation training.

Summary

- This presentation covers four vulnerabilities affecting the proc component of the CAN BCM networking protocol in the Linux kernel.
- I wrote an exploit for the first one in 2020. In 2023, I noticed there was another public vulnerability in the same component. While working on a Proof-of-Concept for that second issue, I found that the patch was incomplete. Besides being able to exploit it on the latest linux kernel version earlier this year, I also found a different vulnerability.
- I will talk about the exploits and proofs of concept I wrote for them, a technique used to speed up one of the exploits and the experience of working with the developers to fix the vulnerabilities I discovered.

Introduction to the CAN BCM protocol in the Linux kernel

Introduction to the Linux CAN BCM protocol

- I have been researching this protocol for some time, primarily focusing on security rather than operational use. My main goal is vulnerability discovery and exploitation. Don't ask me much.
- It's used mostly on vehicles and industrial systems, including defense and military applications. There are several protocols:
 - BCM
 - RAW
 - ISO-TP
 - GW
 - J1939

Vulnerabilities in the proc component of the CAN BCM protocol

First vulnerability

Race condition in `bcm_connect()` leading to use-after-free read

Race condition in `bcm_connect()` leading to use-after-free read

- Vulnerability patched in October 2016 in the Linux kernel upstream, discovered by Andrey Konovalov.
- The CAN BCM protocol was added to the Red Hat Enterprise Linux 7 in the 7.3 release on November 2016, but didn't contain the patch that fixes the vulnerability.
- I noticed in the second semester of 2020 that RHEL 7 and its derivatives missed the patch and wrote an exploit for it that dumps the content of the file `/etc/shadow` and the hash of the root user of the MySQL/MariaDB databases.

Commit deb507f



hartkopp authored and **marckleinebudde** committed on Oct 31, 2016

can: bcm: fix warning in bcm_connect/proc_register

Andrey Konovalov reported an issue with proc_register in bcm.c.
As suggested by Cong Wang this patch adds a lock_sock() protection and
a check for unsuccessful proc_create_data() in bcm_connect().

Reference: <http://marc.info/?l=linux-netdev&m=147732648731237>

Reported-by: Andrey Konovalov <andreyknvl@google.com>

Suggested-by: Cong Wang <xiyou.wangcong@gmail.com>

Signed-off-by: Oliver Hartkopp <socketcan@hartkopp.net>

Acked-by: Cong Wang <xiyou.wangcong@gmail.com>

Tested-by: Andrey Konovalov <andreyknvl@google.com>

Cc: linux-stable <stable@vger.kernel.org>

Signed-off-by: Marc Kleine-Budde <mkl@pengutronix.de>



master



v6.18 ··· v4.9-rc6

The bug

- The protocol doesn't protect against two threads executing the connect() system call on the same socket simultaneously.
- The connect() system call is used to define the address of the destination, just as the bind() system call is used to define the local address.
- While one of the threads succeeds and creates the entry at /proc/net/can-bcm/ENTRY, the other will fail because the entry has already been created. This results in the bo->bcm_proc_read pointer within the socket being set to NULL.
- When the socket is closed and freed, due to bo->bcm_proc_read being NULL, the entry on the proc filesystem is not removed, allowing access to the freed object.

Conditions to abuse it

- As the freed object is part of a linked-list and used inside `list_for_each_entry()`, those characteristics make it a less interesting vulnerability. Too many conditions.
- We would need to know the address of the list (head) and find an object that would have an interesting content at the offsets printed by `bcm_proc_show()` and still allowing to control the field `obj->next`.
- But we can turn this vulnerability into an arbitrary read in an unclean way by abusing the oops generated and the fact that access to the kernel console is allowed to unprivileged users on RHEL 7 and its derivatives.

Abusing Linux kernel oops

- This approach is not viable for a professional exploit, but might be useful in other scenarios.
 - To demonstrate impact of known and unfixed vulnerabilities in most distributions.
 - To validate system configurations and implement hardening improvements.
 - Red team, penetration tests and advanced threat simulation engagements.
 - To learn Linux kernel internals.

KASLR bypass

- To abuse it, we still need a KASLR bypass. Otherwise, even if we can read any kernel address, we don't know where the interesting content is located. Fortunately, there were plenty of kernel addresses leaked on Red Hat Enterprise Linux 7 and its derivatives. I implemented this one below in the exploit.
- Create a process, put it to sleep and read `/proc/PID/stat`. The `wchan` value returned contains the address of a kernel function.

Exploiting a Linux Kernel Infoleak to bypass Linux KASLR

<https://marcograss.github.io/security/linux/2016/01/24/exploiting-infoleak-linux-kaslr-bypass.html>

KASLR bypass

- In fact, KASLR should not be considered a strong barrier and is especially bypassable due to micro-architectural design.
- KASLR: An Exercise in Cargo Cult Security
https://grsecurity.net/kaslr_an_exercise_in_cargo_cult_security
- Exploiting CVE-2022-42703 - Bringing back the stack attack
<https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html>

KASLR bypass

- A long-standing information leak vulnerability existed in the Linux kernel that became an immediate KASLR bypass upon its introduction. This vulnerability was finally patched in 2024.

Linux Kernel GSM Multiplexing Race Condition Local Privilege Escalation Vulnerability (CVE-2023-6546)

<https://github.com/zerozenlabs/ZDI-24-020>

CVE-2024-26816: x86, relocs: Ignore relocations in .notes section

<https://lore.kernel.org/linux-cve-announce/2024041039-CVE-2024-26816-5054@gregkh/>

Commit deb507f

```
diff --git a/net/can/bcm.c b/net/can/bcm.c
index 8e999ffdf28b..8af9d25ff988 100644
--- a/net/can/bcm.c
+++ b/net/can/bcm.c
@@ -1549,24 +1549,31 @@ static int bcm_connect(struct socket *sock, struct sockaddr *uaddr, int len,
     struct sockaddr_can *addr = (struct sockaddr_can *)uaddr;
     struct sock *sk = sock->sk;
     struct bcm_sock *bo = bcm_sk(sk);
+ int ret = 0;
...
- if (bo->bound)
-     return -EISCONN;
+ lock_sock(sk);
+
+ if (bo->bound) {
+     ret = -EISCONN;
+     goto fail;
+ }
```

Commit deb507f

```
@@ -1577,17 +1584,24 @@ static int bcm_connect(struct socket *sock, struct sockaddr *uaddr, int len,
...
- bo->bound = 1;
-
...
    bo->bcm_proc_read = proc_create_data(bo->procname, 0644,
...
+   if (!bo->bcm_proc_read) {
+       ret = -ENOMEM;
+       goto fail;
+   }
...
- return 0;
+ bo->bound = 1;
+
+fail:
+ release_sock(sk);
+
+ return ret;
}
```

Function bcm_proc_show()

```
list_for_each_entry(op, &bo->rx_ops, list) {  
    ...  
    if (op->flags & CAN_FD_FRAME)  
        seq_printf(m, "(%u)", op->nframes);  
    else  
        seq_printf(m, "[%u]", op->nframes);  
    ...  
    if (op->kt_ival1.tv64)  
        seq_printf(m, "timeo=%lld ",  
            (long long)ktime_to_us(op->kt_ival1));  
  
    if (op->kt_ival2.tv64)  
        seq_printf(m, "thr=%lld ",  
            (long long)ktime_to_us(op->kt_ival2));  
    ...  
}
```

How can we abuse it?

- We can successfully reallocate the freed object and populate it with arbitrary content.
- By defining the next pointer to an address we would like to read that is not a pointer and accessing the dangling proc entry, the content of the address will be interpreted as an object and the access results in an oops. The content of the target address can be seen via the kernel console (dmesg) on the R14 register.
- If the address we would like to read is a pointer, it's possible to read it by making it an invalid address (e.g: adding and subtracting 1). This requires at least two accesses to read the content.

How can we abuse it?

```
$ sudo cat /proc/kallsyms | grep -w page_offset_base  
ffffffff81c45c00 D page_offset_base  
$
```

How can we abuse it?

```
(gdb) x/gx 0xffffffff81c45c00
```

```
0xffffffff81c45c00: 0xffff880000000000
```

```
(gdb) x/gx 0xffffffff81c45c00 + 1
```

```
0xffffffff81c45c01: 0x00ffff8800000000
```

```
(gdb) x/gx 0xffffffff81c45c00 - 1
```

```
0xffffffff81c45bff: 0xff880000000000ff
```

```
(gdb)
```

How can we abuse it?

```
$ ./exploit  
[*] PID: 2707  
[*] KASLR disabled  
[*] Socket address: 0xffff88006cce1080  
[*] CAN BCM entry: 28650  
[*] page_offset_base: 0xffff880000000000
```

```
[ 112.500541] R13: ffff88007818d400 R14: 00ffff8800000000 R15: 000000000000000064  
[ 112.515984] R13: ffff88007818d400 R14: ff88000000000000ff R15: 000000000000000064
```

How can we abuse it?

```
$ ./exploit
```

```
[*] PID: 2707
```

```
[*] KASLR disabled
```

```
[*] Socket address: 0xffff88006cce1080
```

```
[*] CAN BCM entry: 28650
```

```
[*] page_offset_base: 0xffff880000000000
```

```
[*] vmemmap_base: 0xffffea0000000000
```

```
[ 112.535092] R13: ffff88007818d400 R14: 00ffffea00000000 R15: 000000000000000064
```

```
[ 112.553935] R13: ffff88007818d400 R14: ffea000000000000 R15: 000000000000000064
```

Optimizing it

- We can read arbitrary kernel content by abusing the default configuration on RHEL 7 and its derivatives. This configuration includes settings that allow an unprivileged user to read the kernel console, do not enable `panic_on_oops` by default, and provide other weaknesses.
- But we would still need to trigger the vulnerability for any read we need to do. The race condition is easy to trigger, but it adds complexity to the final exploit.
- It also has a side effect of creating many more entries at `/proc/net/can-bcm/` than needed.

Optimizing it

- We can use a technique to speed up the exploit.
 - We need a kernel address that we can update at will. Then, when we want to read a target address, we just update the content of the kernel address. But where can we find one?
 - TCP sockets!
 - I wrote an exploit in the past that abused the network address of a TCP socket. The addresses (kernel address) of the TCP sockets were leaked at `/proc/net/tcp`.
 - Instead of filling the `->next` field of freed object with the target address we want to read, we populate it with the address of the TCP socket that holds the interface address. Then, when we want to read a different target address, we bind the desired target address as IPv4/IPv6 address of the TCP socket.

Optimizing it

- The optimization implemented allowed us to use only a dangling CAN BCM entry at `/proc/net/can-bcm/` to read as many addresses we want.
- This also makes the exploit much faster, allowing it to dump the content of the `/etc/shadow` in less than 30 seconds.
- This technique could still be used nowadays in case the address of a TCP socket is obtained.
 - We can't by default bind an IP address that doesn't exist on the interface.
 - But this can be bypassed by `sysctl net.ipv6.ip_nonlocal_bind` on an unprivileged network namespace or the `setsockopt` option `IP_FREEBIND` that applies to the initial network namespace.

Demonstration

<https://www.youtube.com/watch?v=znTLHc2mXIs>

Second vulnerability

Use-after-free read in bcm_proc_show()
(CVE-2023-52922)

Use-after-free read in `bcm_proc_show()` (CVE-2023-52922)

- Vulnerability patched in July 2023 by YueHaibing (Huawei). The objects accessible via the `/proc/net/can-bcm/` entry were being freed before the entry was removed, leading to use-after-free reads in the same `bcm_proc_show()` function.
 - We reported it to Red Hat on July 2024 and it was finally fixed on March 2025.
- The function `bcm_remove_op()`, which is called when the socket is closed, frees the socket's objects before the function `remove_proc_entry()` is executed. If `bcm_remove_op()` is called while `bcm_proc_show()` executes, it can access a freed object.
- The patch changes the order of the functions.

Commit 55c3b96

 YueHaibing authored and marckleinebudde committed on Jul 17, 2023

can: bcm: Fix UAF in bcm_proc_show()

BUG: KASAN: slab-use-after-free in bcm_proc_show+0x969/0xa80
Read of size 8 at addr ffff888155846230 by task cat/7862

CPU: 1 PID: 7862 Comm: cat Not tainted 6.5.0-rc1-00153-gc8746099c197 [#230](#)
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.15.0-1 04/01/2014

Call Trace:

<TASK>

```
dump_stack_lvl+0xd5/0x150
print_report+0xc1/0x5e0
kasan_report+0xba/0xf0
bcm_proc_show+0x969/0xa80
seq_read_iter+0x4f6/0x1260
seq_read+0x165/0x210
proc_reg_read+0x227/0x300
vfs_read+0x1d5/0x8d0
ksys_read+0x11e/0x240
do_syscall_64+0x35/0xb0
entry_SYSCALL_64_after_hwframe+0x63/0xcd
```

Allocated by task 7846:

```
kasan_save_stack+0x1e/0x40
kasan_set_track+0x21/0x30
__kasan_kmalloc+0x9e/0xa0
bcm_sendmsg+0x264b/0x44e0
sock_sendmsg+0xda/0x180
__sys_sendmsg+0x735/0x920
__sys_sendmsg+0x11d/0x1b0
__sys_sendmsg+0xfa/0x1d0
do_syscall_64+0x35/0xb0
entry_SYSCALL_64_after_hwframe+0x63/0xcd
```

Freed by task 7846:

```
kasan_save_stack+0x1e/0x40
kasan_set_track+0x21/0x30
kasan_save_free_info+0x27/0x40
__kasan_slab_free+0x161/0x1c0
slab_free_freelist_hook+0x119/0x220
__kmem_cache_free+0xb4/0x2e0
rcu_core+0x809/0x1bd0
```

bcm_op is freed before procfs entry be removed in bcm_release(), this lead to bcm_proc_show() may read the freed bcm_op.

Fixes: [ffd980f](#) ("[CAN]: Add broadcast manager (bcm) protocol")

Signed-off-by: YueHaibing <yuehaibing@huawei.com>

Reviewed-by: Oliver Hartkopp <socketcan@hartkopp.net>

Acked-by: Oliver Hartkopp <socketcan@hartkopp.net>

Link: <https://lore.kernel.org/all/20230715092543.15548-1-yuehaibing@huawei.com>

Cc: stable@vger.kernel.org

Signed-off-by: Marc Kleine-Budde <mkl@pengutronix.de>

 [master](#) ·  v6.18 ·  v6.5-rc3

Commit 55c3b96

```
diff --git a/net/can/bcm.c b/net/can/bcm.c
index 32db244ba2c012..0a082726395af3 100644
--- a/net/can/bcm.c
+++ b/net/can/bcm.c
@@ -1572,6 +1572,12 @@ static int bcm_release(struct socket *sock)

    lock_sock(sk);

+#if IS_ENABLED(CONFIG_PROC_FS)
+ /* remove procfs entry */
+ if (net->can.bcmproc_dir && bo->bcm_proc_read)
+   remove_proc_entry(bo->procname, net->can.bcmproc_dir);
+#endif /* CONFIG_PROC_FS */
+
    list_for_each_entry_safe(op, next, &bo->tx_ops, list)
      bcm_remove_op(op);
```

Commit 55c3b96

```
@@ -1607,12 +1613,6 @@ static int bcm_release(struct socket *sock)
    list_for_each_entry_safe(op, next, &bo->rx_ops, list)
        bcm_remove_op(op);

-#if IS_ENABLED(CONFIG_PROC_FS)
- /* remove procfs entry */
- if (net->can.bcmproc_dir && bo->bcm_proc_read)
-     remove_proc_entry(bo->procname, net->can.bcmproc_dir);
-#endif /* CONFIG_PROC_FS */
-
    /* remove device reference */
    if (bo->bound) {
        bo->bound = 0;
```

Heap grooming

- We explored some ideas by controlling the `->next` field of freed object, but they didn't pan out.
- The `->next` field is at offset 0 and some heap grooming techniques doesn't allow to control it. We came up with a mix of using the VSOCK protocol + `sendmsg()`.
- A problem with `sendmsg()` heap grooming technique is that the system call frees the object when returning to user mode. We circumvent this by filling up the send/receive buffer of the socket, so the next `sendmsg()` call sleeps. The sleeping threads woke up when a read is done, consuming the send/receive buffer.
- The VSOCK is an interesting protocol because its `sendmsg()` implementation doesn't validate the offset 0, different from other networking protocols. The KCM protocol also works for this purpose, but it wasn't available in this case.

Abusing CVE-2023-52922

- We developed two proofs of concept for it. None of them need to reclaim the object.
 - The first one leaks the encoded freelist pointer.
 - This can help attackers to bypass a security mitigation
 - The second one dumps the addresses of a slab.
 - This can also help attackers to bypass a security mitigation

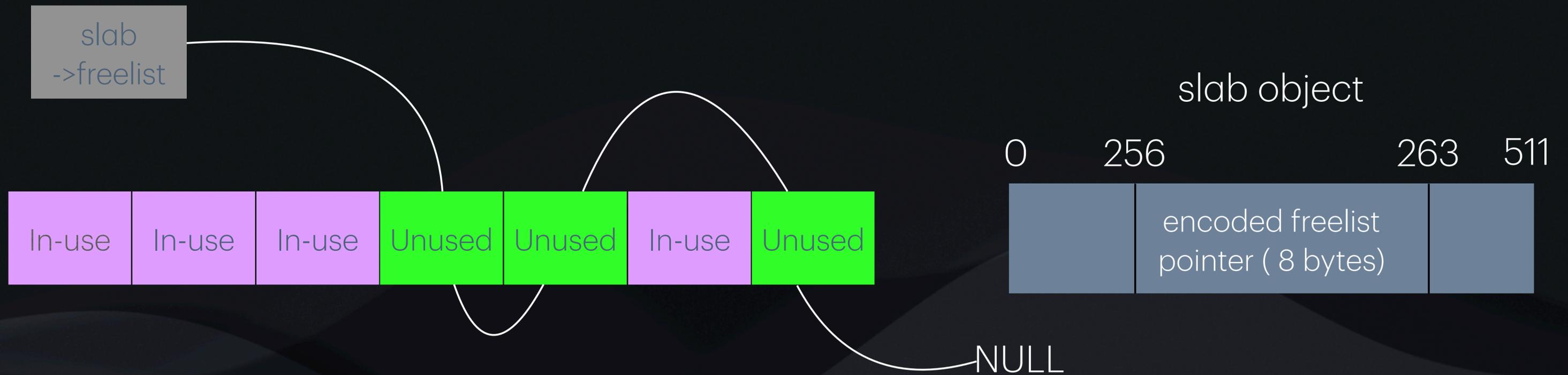
Leaking encoded freelist pointer

- The first result
 - In the Linux dynamic memory management, the unused objects are reused to store the addresses of the next unused objects. This is the freelist pointer.
 - In the past, in some conditions, the freelist pointer were placed at the beginning of the object, but it was easier for an out-of-bounds write to overwrite it. To mitigate this attack, the freelist was moved to the middle of the object. Also, it's encoded in modern Linux kernel to reduce the impact even when it's leaked.

Leaking encoded freelist pointer

- The first result
 - Coincidentally, if the vulnerability is triggered and the freed object is not reclaimed, the data leaked is at the same offset of the encoded freelist pointer (the middle of the object). As the object ->next pointer is not overwritten even after being freed, it still points to the head of the list. Meeting one of the conditions to abuse it.
 - This fact makes the `list_for_each_entry()` on `bcm_proc_show()` to exit gracefully. This results in the vulnerability to be used to leak encoded freelist pointer in a reliable and clean way. That value could be used by an attacker to craft an arbitrary encoded freelist pointer.

slab in-use and unused objects



Defeating the randomization of slabs objects

- The second result
 - We noticed we can dump the encoded freelist pointer of all objects from a slab. We then tried to derive another attack.
 - We discovered if we XOR the encoded freelist pointer of the last object with the others encoded freelist pointers, we obtain a result that is very similar to the addresses of the objects being used in the operation.
 - We still believe it is possible to leak the SLUB's cache freelist random value, which is used to encode the freelist pointers. If this is possible and combined with the two attacks, the vulnerability would have a greater impact.

Extracting the maximum impact

- Improvements
 - We still believe it is possible to leak the SLUB's cache freelist random value, which is used to encode the freelist pointers. If this is possible and combined with the two attacks, the vulnerability would have a greater impact.
 - The leaked value suffers from truncation/rounding, reducing the precision of the leaked address. However, under specific conditions, this imprecision is neutralized, resulting in the original, untruncated value being leaked.

Demonstration

<https://www.youtube.com/watch?v=XQ3QlXqn6pI>

That's the question.

- A common pattern of vulnerabilities?
 - Even though the characteristics of the vulnerabilities were different, they were in the same component of the same protocol. I also noticed there were other patches fixing issues in the /proc component of the CAN BCM protocol, but I didn't take time to analyze those ones.
 - We wrote a blog post detailing our results and mentioned in the post that looks like the CAN BCM protocol was having a common pattern of vulnerabilities affecting the proc component.

USE-AFTER-FREE VULNERABILITY IN THE CAN BCM SUBSYSTEM LEADING TO INFORMATION DISCLOSURE (CVE-2023-52922)

<https://allelesecurity.com/uaf-can-bcm/>

Third vulnerability

Use-after-free read in `bcm_proc_show()` due to missing rcu read protection.

Use-after-free read in `bcm_proc_show()` due to missing rcu read protection.

- After we had finished the work on the vulnerability CVE-2023-52922 and reviewing the blog post, I remembered there are other paths on the CAN BCM protocol that free the struct `bcm_op` objects, the commands: `TX_DELETE` and `RX_DELETE`.
- As the fix for CVE-2023-52922 focused on the struct `bcm_op` objects being freed when the socket is released, the function `bcm_release()`, I imagined it could still be triggered via the delete commands, as deletion also frees the object.
- I wrote a proof of concept and confirmed I had discovered a 0-day vulnerability, which we reported upstream.
 - As anticipated, we noted in the blog post that the proc implementation of the CAN BCM protocol seemed to have a pattern of vulnerabilities, which we then confirmed by finding a new one. The blog post was delayed while the vulnerability was being fixed.

Use-after-free read in `bcm_proc_show()` due to missing rcu read protection.

- Besides the problem in the order of the operation during the release of the socket, the protocol implemented Read-Copy Update (RCU) locking, but it wasn't being used properly.
 - RCU allows an object that is about to be freed to be used in a safe way. In simple terms, the reader notifies the system it is using the object and then the freeing is delayed.
 - The reader of those objects needs to notify the kernel by using `rcu_read_lock()` and `rcu_read_unlock()`. Even though the objects were being freed via RCU, the reader weren't using it.
 - To make things worse, the implementations using RCU need to have some properties that are enforced by the right API functions. This wasn't happening with the CAN BCM protocol.
 - E.g: `list_for_each_entry_rcu()` rather than `list_for_each_entry()`.

Commit dac5e62

 **hartkopp** authored and **marckleinebudde** committed on May 19

can: bcm: add missing rcu read protection for procfs content

When the procfs content is generated for a bcm_op which is in the process to be removed the procfs output might show unreliable data (UAF).

As the removal of bcm_op's is already implemented with rcu handling this patch adds the missing rcu_read_lock() and makes sure the list entries are properly removed under rcu protection.

Fixes: [f1b4e32](#) ("can: bcm: use call_rcu() instead of costly synchronize_rcu()")

Reported-by: Anderson Nascimento <anderson@allelesecurity.com>

Suggested-by: Anderson Nascimento <anderson@allelesecurity.com>

Tested-by: Anderson Nascimento <anderson@allelesecurity.com>

Signed-off-by: Oliver Hartkopp <socketcan@hartkopp.net>

Link: <https://patch.msgid.link/20250519125027.11900-2-socketcan@hartkopp.net>

Cc: stable@vger.kernel.org # >= 5.4

Signed-off-by: Marc Kleine-Budde <mkl@pengutronix.de>

 **master** ·  v6.18 ··· v6.15

Using the proper RCU functions

```
diff --git a/net/can/bcm.c b/net/can/bcm.c
index 871707dab7db..6bc1cc4c94c5 100644
--- a/net/can/bcm.c
+++ b/net/can/bcm.c
@@ -217,11 +217,13 @@ static int bcm_proc_show(struct seq_file *m, void *v)
...
- list_for_each_entry(op, &bo->rx_ops, list) {
+ rcu_read_lock();
+
+ list_for_each_entry_rcu(op, &bo->rx_ops, list) {
...
@@ -273,10 +275,13 @@ static int bcm_proc_show(struct seq_file *m, void *v)
...
+
+ rcu_read_unlock();
+
return 0;
}
```

Using the proper RCU functions

```
@@ -856,11 +861,11 @@ static int bcm_delete_rx_op(..., struct bcm_msg_head *mh,  
...  
- list_del(&op->list);  
+ list_del_rcu(&op->list);  
bcm_remove_op(op);  
...  
@@ -876,11 +881,11 @@ static int bcm_delete_tx_op(..., struct bcm_msg_head *mh,  
list_for_each_entry_safe(op, n, ops, list) {  
...  
- list_del(&op->list);  
+ list_del_rcu(&op->list);  
bcm_remove_op(op);  
...  
@@ -1294,11 +1299,11 @@ static int bcm_rx_setup(..., struct msghdr *msg,  
...  
- list_del(&op->list);  
+ list_del_rcu(&op->list);  
bcm_remove_op(op);
```

Trying a proper fix (let's hope so)

- One of the developers asked if only the RCU lock pair is enough, but after testing a patch, UAF still happened. There were other bugs. The function `list_del()` executing concurrently with `list_for_each_entry()`. After changing all the functions to their RCU equivalent, no issues happened even after testing the proof of concept for more than 30.000 times.
- Fortunately, I had time available to test their patch and elaborate my approach that fixed the problem. I wrote a reliable proof of concept that I could use to validate the patches.
- But this takes time! I had to maintain my own Linux kernel source based on the release candidate at the time, as the Linux kernel is actively being developed and changes so much and so fast. The patch needs to be developed based on the latest Linux kernel version available to avoid conflicts.

But then, I found another
vulnerability.

Fourth vulnerability

Race condition in `bcm_can_tx()` leading to out-of-bounds read (CVE-2025-38004)

Race condition in `bcm_can_tx()` leading to out-of-bounds read (CVE-2025-38004)

- While working on CVE-2023-52922, I had to do heap grooming. I usually do that with the subsystem's own objects. So, I played with the sending and deleting operations of the CAN BCM protocol.
- Through the delete operations I noticed a UAF on `bcm_proc_show()` could still be triggered, bypassing the patch for CVE-2023-52922.
- Through the send operations (TX_SETUP) I learned some details of the protocol and noticed the function `bcm_can_tx()` uses a variable and increment it later on. That function doesn't use lock mechanism, so it seemed suspicious.
- It can be executed directly (at a user wishes) and via a scheduled timer. So, it could race making the second execution to read the variable beyond the value it should.

op->currframe unprotected accesses

```
static void bcm_can_tx(struct bcm_op *op)
{
...
    struct canfd_frame *cf = op->frames + op->cfsiz * op->currframe;
...
    skb_put_data(skb, cf, op->cfsiz);
...
    skb->dev = dev;
    can_skb_set_owner(skb, op->sk);
    err = can_send(skb, 1);
...
    op->currframe++;
...
    if (op->currframe >= op->nframes)
        op->currframe = 0;
...
}
```

Consequences and the protection

- An out-of-bounds is triggered and the data read is returned to the user.
- It's possible to reach several different caches. The user controls the size of the object being allocated.
- It leaks struct pages, dynamic objects (heap de-randomization) and the vmlinux addresses, leading to KASLR bypass.
- The fix was to protect the race in the accesses of `op->currframe` in `bcm_can_tx()`
 - I suggested using `spin_lock()` to guard the variable access. The developers decided to try a different approach first, but it didn't work and my suggestion was implemented in the end. Different from my suggestion, they implemented the lock per-object.

Commit c2aba69

 **hartkopp** authored and **marckleinebudde** committed on May 19

can: bcm: add locking for bcm_op runtime updates

The CAN broadcast manager (CAN BCM) can send a sequence of CAN frames via hrtimer. The content and also the length of the sequence can be changed resp reduced at runtime where the 'currframe' counter is then set to zero.

Although this appeared to be a safe operation the updates of 'currframe' can be triggered from user space and hrtimer context in bcm_can_tx(). Anderson Nascimento created a proof of concept that triggered a KASAN slab-out-of-bounds read access which can be prevented with a spin_lock_bh.

At the rework of bcm_can_tx() the 'count' variable has been moved into the protected section as this variable can be modified from both contexts too.

Fixes: [ffd980f](#) ("[CAN]: Add broadcast manager (bcm) protocol")

Reported-by: Anderson Nascimento <anderson@allelesecurity.com>

Tested-by: Anderson Nascimento <anderson@allelesecurity.com>

Reviewed-by: Marc Kleine-Budde <mkl@pengutronix.de>

Signed-off-by: Oliver Hartkopp <socketcan@hartkopp.net>

Link: <https://patch.msgid.link/20250519125027.11900-1-socketcan@hartkopp.net>

Cc: stable@vger.kernel.org

Signed-off-by: Marc Kleine-Budde <mkl@pengutronix.de>

 **master** ·  v6.18 ... v6.15

Implementing lock per-object

```
@@ -122,6 +123,7 @@ struct bcm_op {  
122 123     struct canfd_frame last_sframe;  
123 124     struct sock *sk;  
124 125     struct net_device *rx_reg_dev;  
126 +     spinlock_t bcm_tx_lock; /* protect currframe/count in runtime updates */  
125 127     };
```

Guarding the variable accesses

```
net/can/bcm.c  @@ -285,13 +287,18 @@ static void bcm_can_tx(struct bcm_op *op)
285 287 {
286 288     struct sk_buff *skb;
287 289     struct net_device *dev;
288 - struct canfd_frame *cf = op->frames + op->cfsiz * op->currframe;
290 + struct canfd_frame *cf;
289 291     int err;
290 292
291 293     /* no target device? => exit */
292 294     if (!op->ifindex)
293 295         return;
294 296
297 + /* read currframe under lock protection */
298 + spin_lock_bh(&op->bcm_tx_lock);
299 + cf = op->frames + op->cfsiz * op->currframe;
300 + spin_unlock_bh(&op->bcm_tx_lock);
301 +
```

Guarding the variable accesses

```
net/can/bcm.c  [copy] [toggle]
322 +
323 + /* update currframe and count under lock protection */
324 + spin_lock_bh(&op->bcm_tx_lock);
325 +
315 326 if (!err)
316 327     op->frames_abs++;
317 328
318 329     op->currframe++;
319 330
320 331     /* reached last frame? */
321 332     if (op->currframe >= op->nframes)
322 333         op->currframe = 0;
334 +
335 +     if (op->count > 0)
336 +         op->count--;
337 +
338 +     spin_unlock_bh(&op->bcm_tx_lock);
323 339 out:
324 340     dev_put(dev);
325 341 }
```

Leaking dynamic objects (0xffff8880...)

```
[root@almalinux95research tmp]# for ((i=0; i<1000; i++)); do ./exploit 3  
2 0 0 0 0 1 0; done
```

```
[*] - Success!
```

```
can_id: 1
```

```
len: 0
```

```
__pad: 0
```

```
__res0: 0
```

```
len8_dlc: 0
```

```
data:
```

```
0x0 0x21 0x53 0xc 0x80 0x88 0xff 0xff
```

Leaking struct page objects (0xffffea...)

```
[root@almalinux95research tmp]# for ((i=0; i<1000; i++)); do ./exploit 2 0 0 0 0 1 0;
done
[*] - Success!
can_id: 2465024
len: 0
__pad: 234
__res0: 255
len8_dlc: 255
data:

0x0 0x82 0x35 0x0 0x0 0xea 0xff 0xff
[*] - Success!
can_id: 2232320
len: 0
__pad: 234
__res0: 255
len8_dlc: 255
data:

0xc0 0xf5 0xd0 0x0 0x0 0xea 0xff 0xff
```

Leaking vmlinux addresses (0xffffffff8265...)

```
[root@almalinux95research tmp]# for ((i=0; i<1000; i++)); do ./exploit 32 0 0 0 0 1 0
; done
[*] - Success!
can_id: 32
len: 0
__pad: 0
__res0: 0
len8_dlc: 0
data:

0x6 0x24 0x41 0xed 0x0 0x0 0x0 0x0
[*] - Success!
can_id: 5
len: 0
__pad: 0
__res0: 0
len8_dlc: 0
data:

0xb3 0xc3 0x65 0x82 0xff 0xff 0xff 0xff
```

Conclusion

Conclusion

- Some vulnerabilities are harder to fix.
 - A proper auditing of the whole subsystem is recommended.
- Use-after-free reads are also interesting.
 - It leads to arbitrary file reading and leakage of processes memory content, even across namespaces/containers.
- The security of the Linux ecosystem is messy.
 - Many public bugs and probable vulnerabilities are unpatched on your systems right now.
- Auditing well-known subsystems might still be worthwhile.
 - The out-of-bounds read (CVE-2025-38004) is 17 years old. It's not part of the core, but it leaks interesting objects and as far as I know, is enabled by default on most distributions.

THANK YOU!

Questions?