EXPLOITING CHECKM8 WITH UNKNOWN SECUREROM FOR THE T2 CHIP

Alex Kovrizhnykh
@a1exdandy

whoami

- Reverse engineer and security researcher
- Flare-On 2018-2020 winner (#11, #3, #7 place respectively)
- Articles
  - Edge Browser exploitation writeup
  - Flare-On 2019 writeup
  - checkm8 technical analysis
  - checkm8 for Apple Lightning to VGA Adapter

CHECKM8 RELATED ARTICLES AND WORKS

Technical analysis of the checkm8 exploit

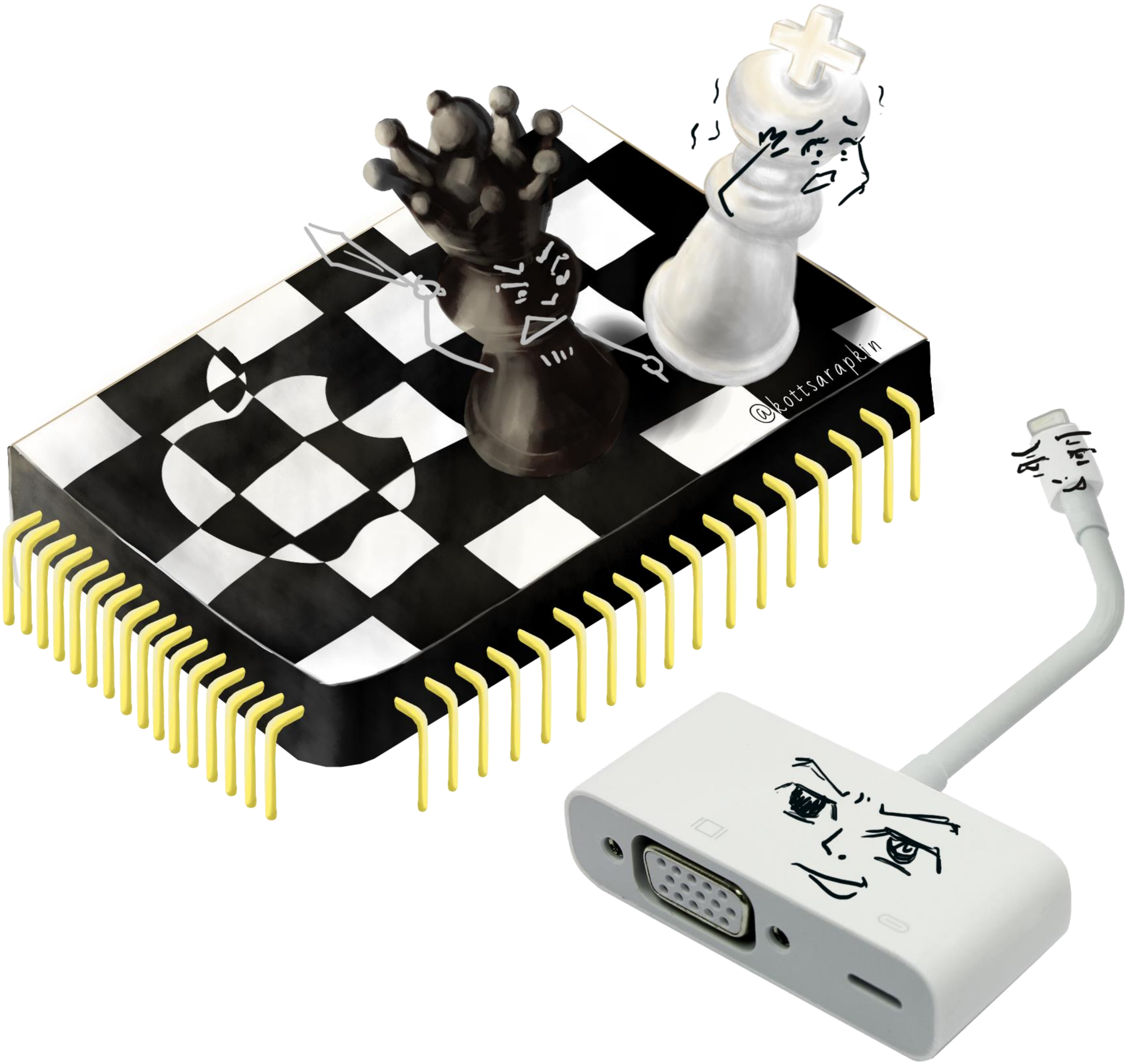
ax🔥🌸mX
@axi0mX

"#checkm8: The iPhone Exploit That Hackers Use to Research Apple's Most Sensitive Code"

This is what the title of this write-up would be if it was a VICE article. This is a detailed write-up of the vulnerability I found and how the exploit really works.

checkm8 for Apple Lightning to VGA Adapter

- S5L8747 has executable SRAM by default
- Implement the code that searches for a standard string USB-descriptor and overwrites it with a SecureROM fragment
- Also works for S7002 - Apple Watch (1st gen.), dumped by @chiptunext
- Both SecureROMs have been added to the securerom.fun after this research
- PoC for S5L8747 and S7002
- Article (RU)



T7000, S8000, S8003

- Adapted heap feng shui as in other devices instead of task structure corruption for iPhone 6s (S8000)
- @moski_dev also checked this on T7000 and S8003
- PoC
- These processors were also added to King (C/C++ port of checkm8 by @Blips_and_Chitz) and were successfully launched on Windows

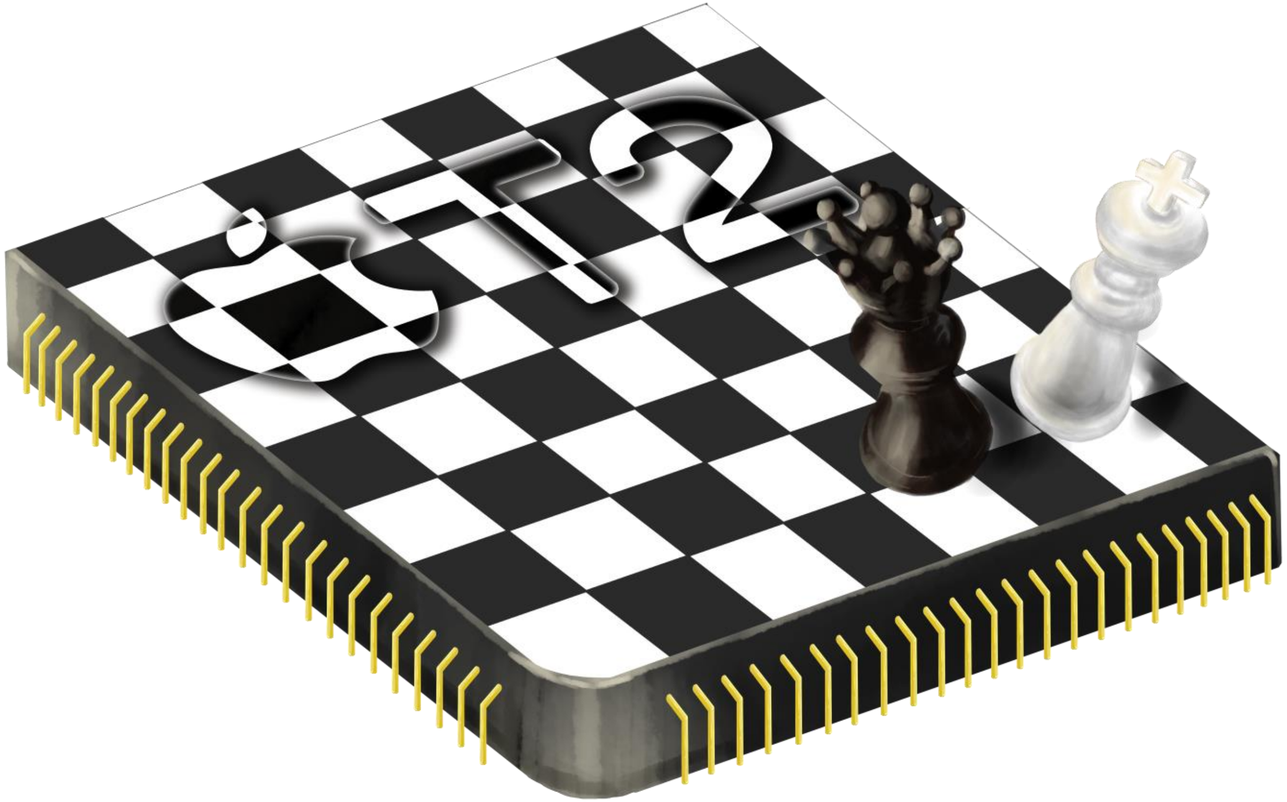
- **T7000**
  - Apple TV (4th generation)
  - HomePod
  - iPad mini 4
  - iPhone 6
  - iPhone 6 Plus
  - iPod touch (6th generation)
- **S8000, S8003**
  - iPad (5th generation)
  - iPhone 6s
  - iPhone 6s Plus
  - iPhone SE

S5L8940X, S5L8942X, S5L8945X

- Together with @nyan_satan, using his iPad mini 1 prototype device, the reason why checkm8 does not work with default PC USB-stack on A5 processors was found
- Using Arduino and MAX3421E-based USB Host Shield, we have successfully ported checkm8 to A5/A5X
- Our research and PoC

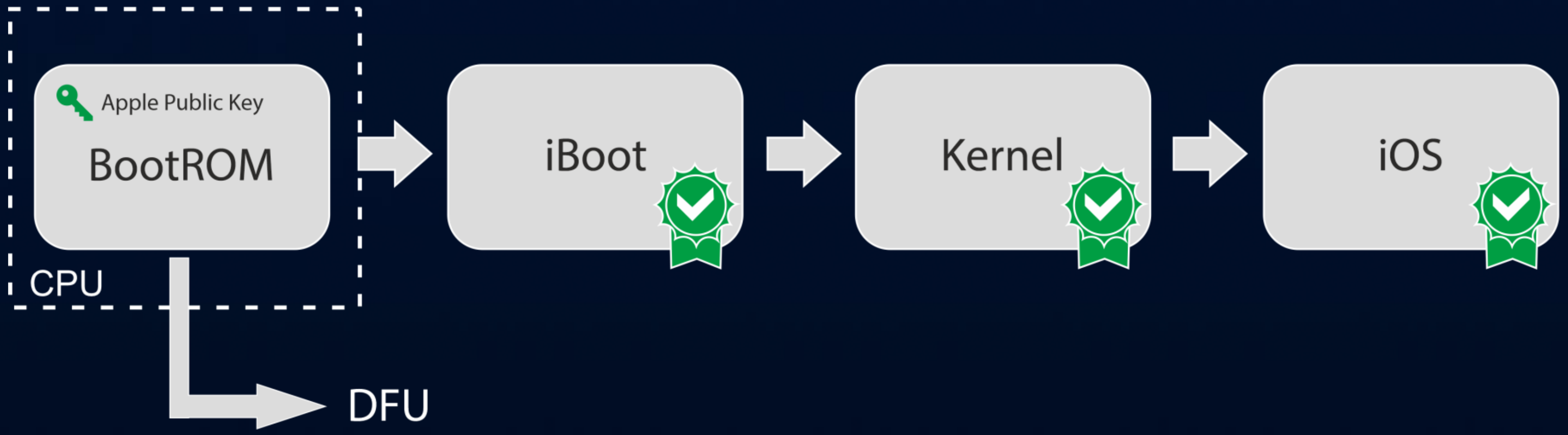
T2

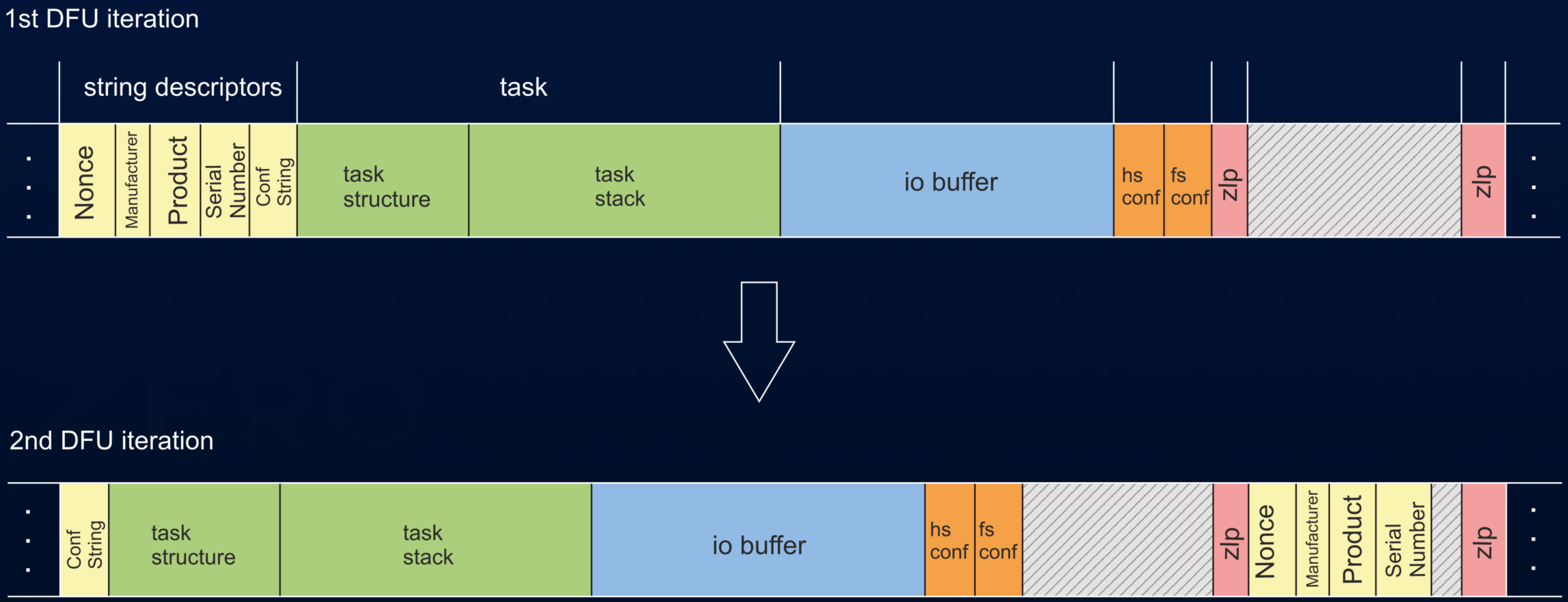
- Was dumped by me on December 3, 2019
- Independently was dumped by T2 Development Team on March 6, 2020
- In both cases, brute-force of the T2 SecureROM offsets for checkm8 was used
- I will tell you my way



checkm8

- Affecting the iPhone 4S (A5 chip) through the iPhone X (A11 chip)
- checkm8 exploits two vulnerabilities
  - use-after-free of USB IO-buffer (**ep0_data_phase_buffer** pointer)
  - memory leak of **usb_device_io_request** object




1st DFU iteration
string descriptors
task
Nonce
Manufacturer
Product
Serial Number
Conf String
task structure
task stack
io buffer
hs conf
fs conf
zlp
zlp
2nd DFU iteration
Conf String
task structure
task stack
io buffer
hs conf
fs conf
zlp
Nonce
Manufacturer
Product
Serial Number
zlp

checkm8 stages (for iPhone 7 as example)

1. Heap feng shui
2. Allocation and deallocation of IO buffer without global state clearing (UAF triggering)
3. Rewriting **usb_device_io_request** on heap using UAF
4. Payload placement
5. Callback-chain execution
6. Shellcode execution

checkm8 details

- To exploit the vulnerability, especially starting with the iPhone 7, you need to know the various offsets in SecureROM, which is why it is unclear how to develop an exploit without having SecureROM access
- What do you need to know to exploit?
  - Starting with iPhone 7, the exploit uses a callback chain to disable the WXN bit and edit translation tables
    - This is achieved by building a fake chain of **usb_device_io_request** using the **"next"** and **"callback"** fields
    - You need to know the addresses of gadgets in SecureROM to build a callback chain

The Chicken-and-Egg Problem 🥚🐣🐥

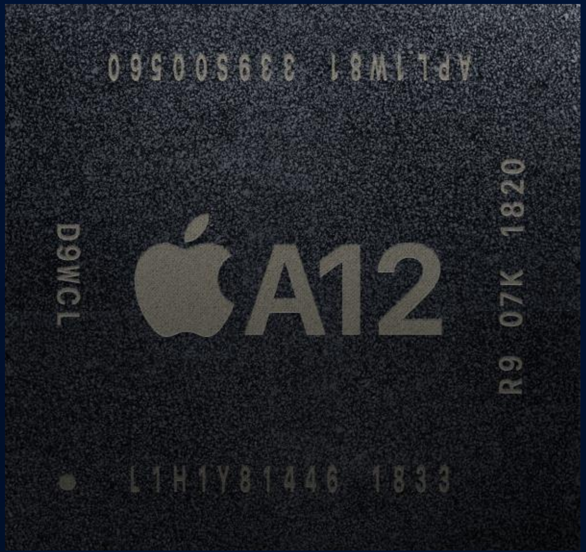
- Possible solutions:
  - Prototype devices (EVT, PVT, DVT, etc)
    - More info about prototypes by @1nsane_dev
  - Other vulnerabilities
    - Maybe at a higher level
  - Hardware ways
  - ...

T2 case

securerom.fun

iBoot-3332.0.0.1.23

iBoot-3401.0.0.1.16



iBoot-3865.0.0.4.6

ZERONIGHTSX

Plan

1. Achieve the ability to dump a small piece of SecureROM
2. Using this, dump the necessary SecureROM fragments
3. Port checkm8

⇩

- We need to find the minimum number of gadgets/functions, with which we can dump the SecureROM fragment

iPhone 7 example

- 9 code offsets
- 7 data offsets

```
constants_usb_t8010 = [
            0x1800B0000, # 1 - LOAD_ADDRESS
    0x6578656365786563, # 2 - EXEC_MAGIC
    0x646F6E65646F6E65, # 3 - DONE_MAGIC
    0x6D656D636D656D63, # 4 - MEMC_MAGIC
    0x6D656D736D656D73, # 5 - MEMS_MAGIC
            0x10000DC98, # 6 - USB_CORE_DO_IO
]
constants_checkm8_t8010 = [
            0x180088A30, # 1 - gUSBDescriptors
            0x180083CF8, # 2 - gUSBSerialNumber
            0x10000D150, # 3 - usb_create_string_descriptor
            0x1800805DA, # 4 - gUSBSRNMStringDescriptor
            0x1800AFC00, # 5 - PAYLOAD_DEST
  PAYLOAD_OFFSET_ARM64, # 6 - PAYLOAD_OFFSET
    PAYLOAD_SIZE_ARM64, # 7 - PAYLOAD_SIZE
            0x180088B48, # 8 - PAYLOAD_PTR
]
t8010_func_gadget              = 0x10000CC4C
t8010_enter_critical_section   = 0x10000A4B8
t8010_exit_critical_section    = 0x10000A514
t8010_dc_civac                 = 0x10000046C
t8010_write_ttbr0              = 0x1000003E4
t8010_tlbi                     = 0x100000434
t8010_dmb                      = 0x100000478
t8010_handle_interface_request = 0x10000DFB8
```

First idea

- There is no ASLR in SecureROM, you can brute some address byte by byte
- In our case, you can brute the callback **standard_device_request_cb** as part of **usb_device_io_request**

usb_device_io_request object

```
struct usb_device_io_request
{
  u_int32_t                        endpoint;
  volatile u_int8_t                *io_buffer;
  int                              status;
  u_int32_t                        io_length;
  u_int32_t                        return_count;
  void (*callback) (struct usb_device_io_request *io_request);
  struct usb_device_io_request    *next;
};
```

ZERONIGHTSX

Call chain on abort

- synopsys_otg_abort_endpoint
  - for each **io_req** in linked list
    - usb_core_complete_endpoint_io(**io_req**)
    - **io_req**->callback(**io_req**)
    - free(**io_req**) <=== problem

Show me true oracle...

Device is still in DFU:

- Hit into a RET gadget with a frame shift by 0x20

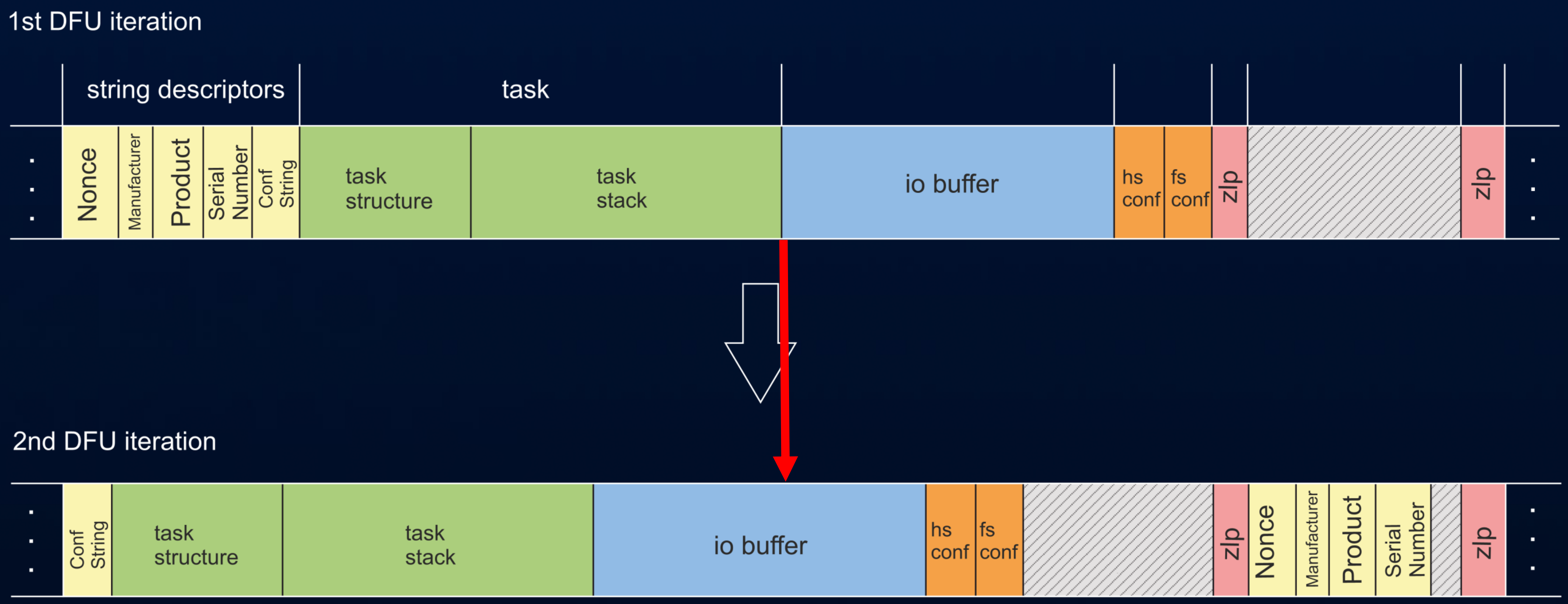
```
LDP               X29, X30, [SP,#0x20+var_10]
LDP               X20, X19, [SP+0x20+var_20],#0x20
RET
```

Device not in DFU:

- Didn't hit the desired gadget or the exploit failed

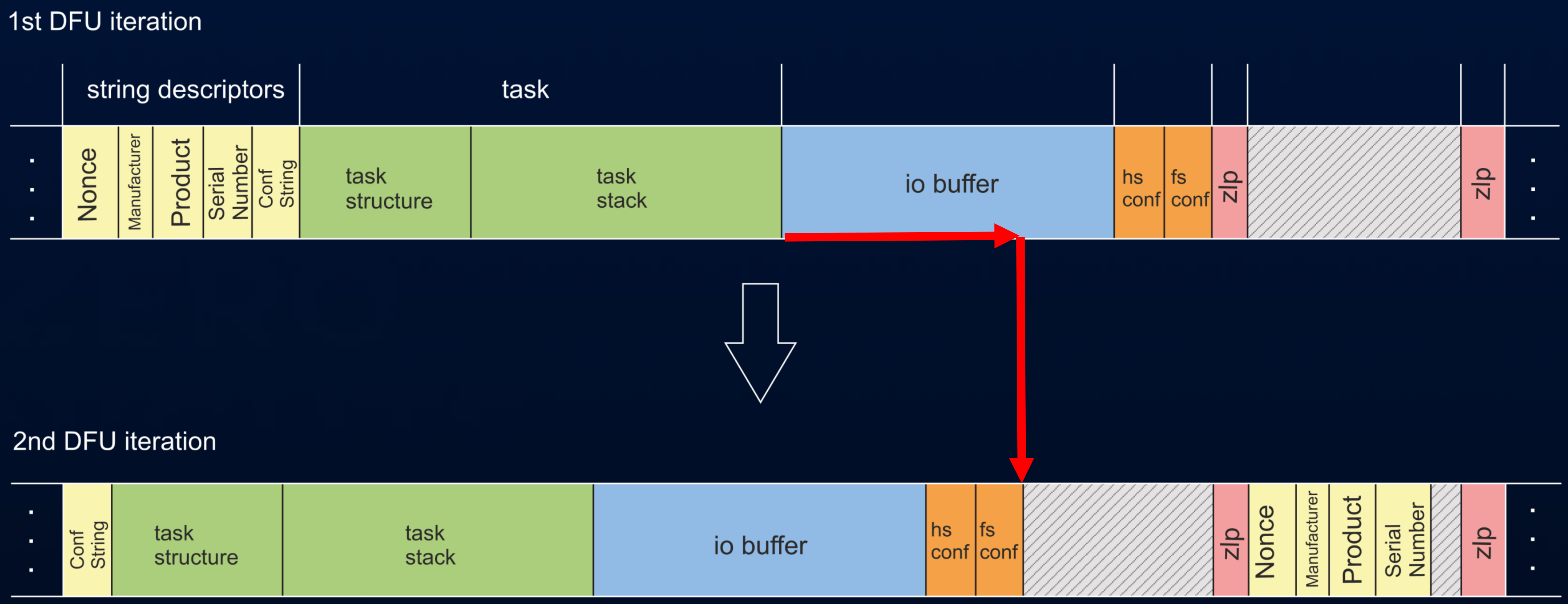
Idea from ipwndfu_public

- We can shift the UAF pointer to a multiple of 0x40 before next DFU iteration so as not to corrupt the heap



Idea from ipwndfu_public

- We can shift the UAF pointer to a multiple of 0x40 before next DFU iteration so as not to corrupt the heap



Call chain on abort

- synopsys_otg_abort_endpoint
  - for each **io_req** in linked list
    - usb_core_complete_endpoint_io(**io_req**)
      - **io_req**->callback(**io_req**)
      - free(**io_req**) <=== not a problem anymore

...I said true oracle...

Device is still in DFU:

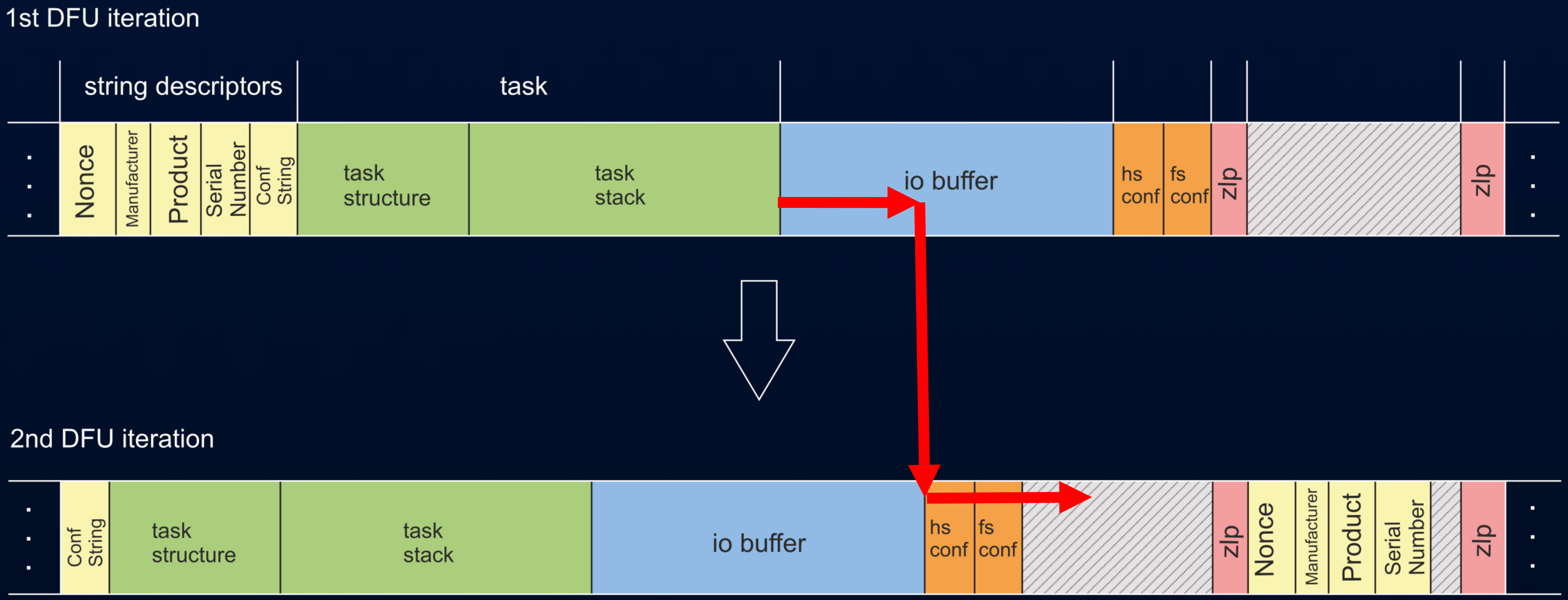
- Some code was executed and control returned correctly (found RET, etc.)

Device not in DFU:

- Executed some bs or exploit failed

Improving the idea from ipwndfu_public and my findings

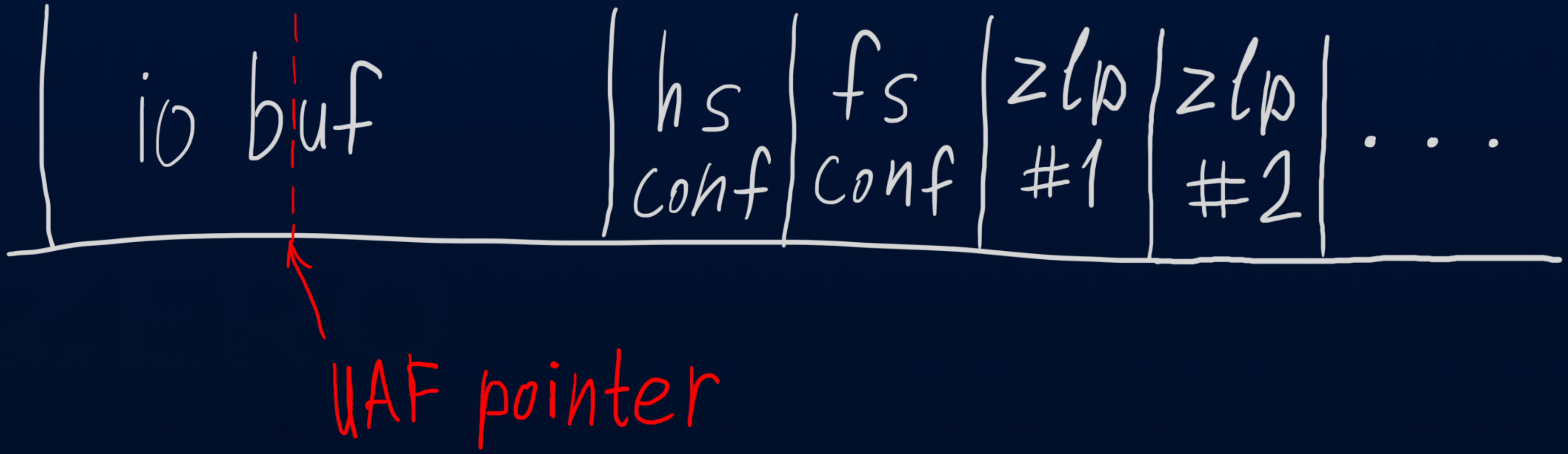
- UAF pointer can be shifted multiple times in 0x40 increments
- We can overflow **hs** and **fs conf.** descriptors and achieve buffer overread



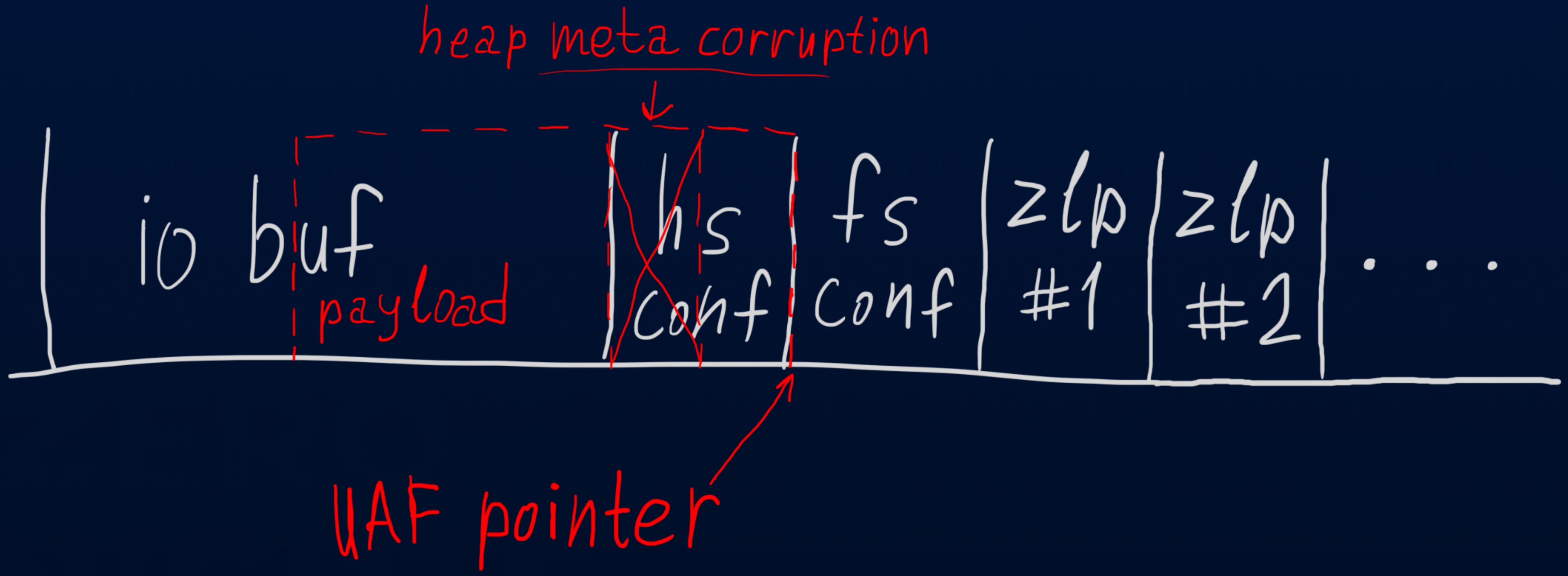
Improving the idea from ipwndfu_public and my findings

1. UAF triggering
2. Memory leak of two USB requests
3. Write payload and overwrite **hs conf.** to achieve buffer overread
4. Read **hs conf.** and get the metadata of the next heap chunk
5. Overwrite metadata and **fs conf.**
6. Read **fs conf.** and get the metadata of the next heap chunk with USB request
7. Building a fake chain of 3 USB requests

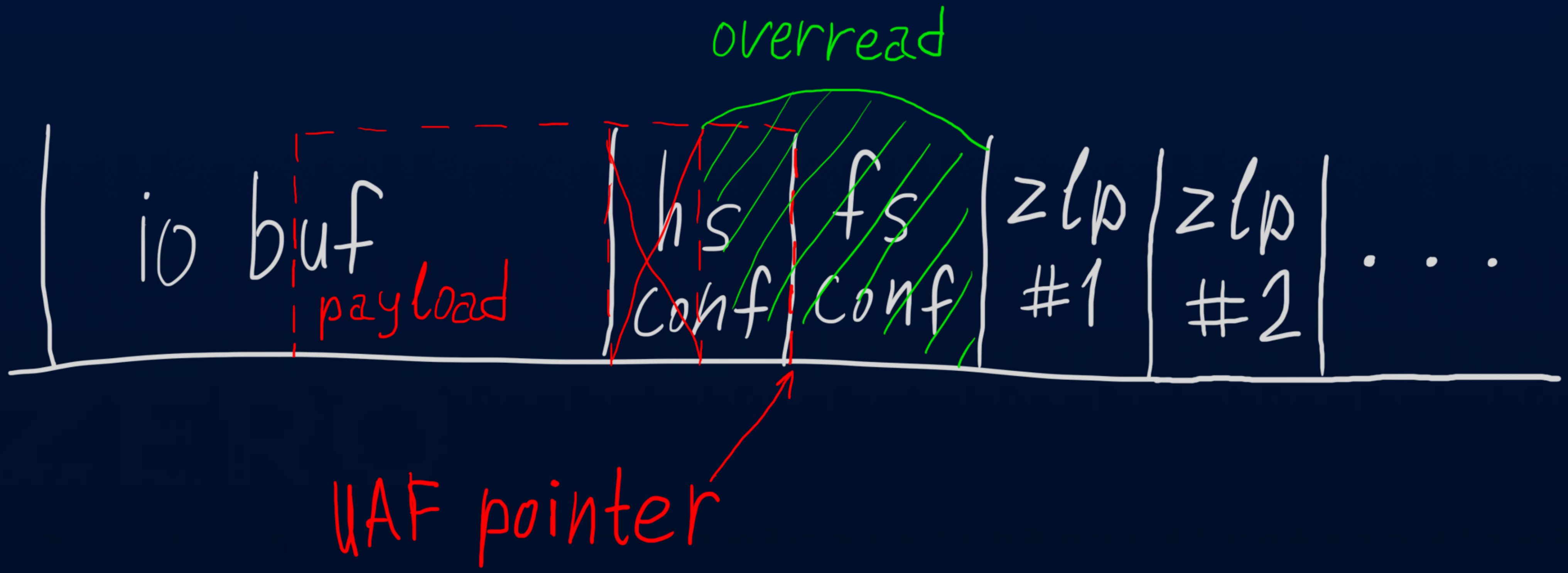

io buf
hs conf
fs conf
...
UAF pointer

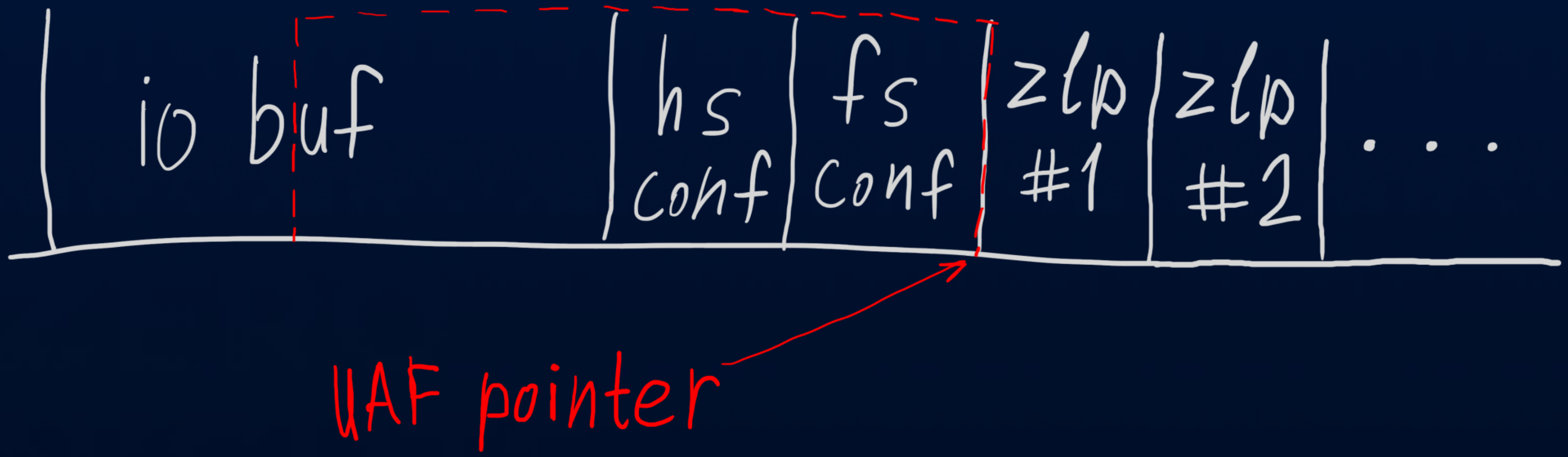


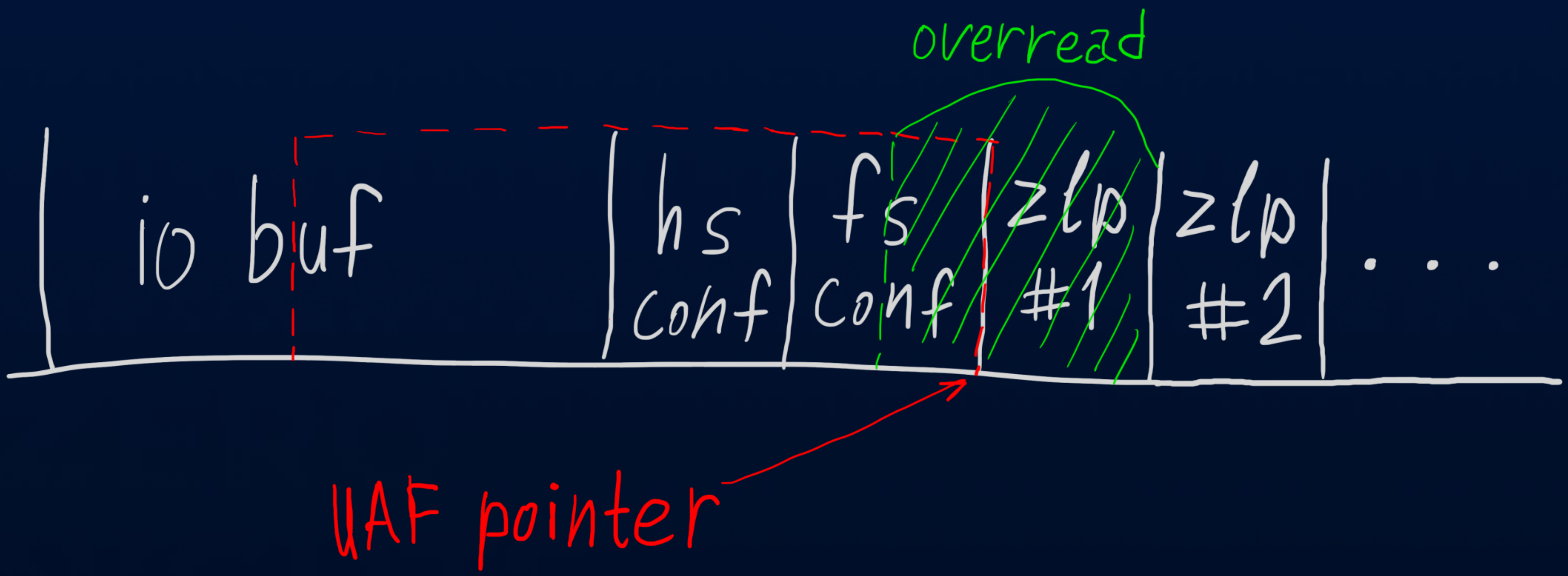
zlp – zero length packet

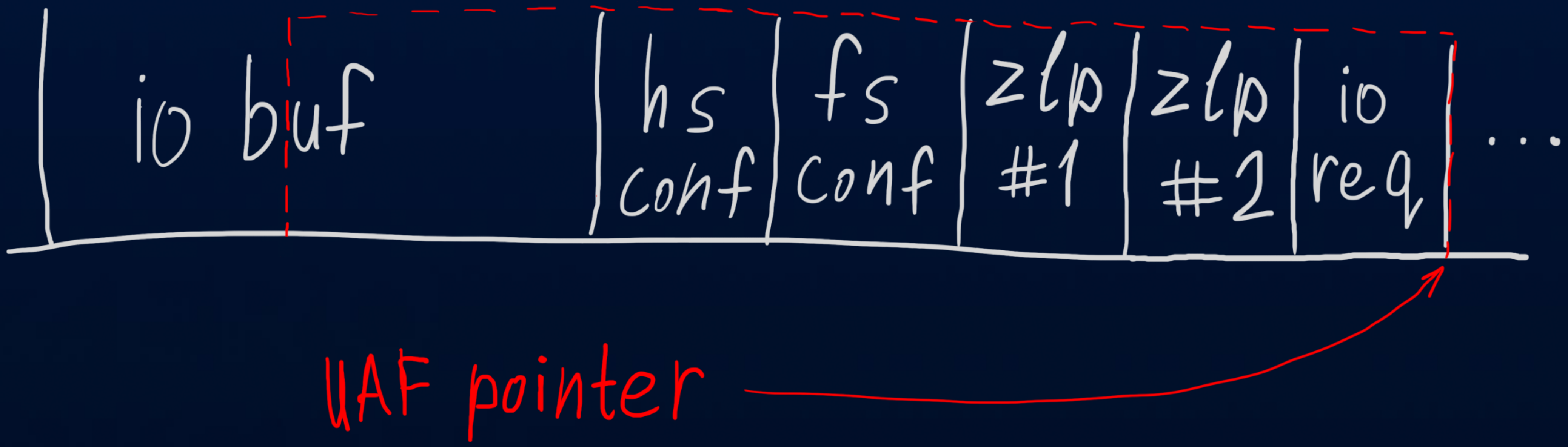


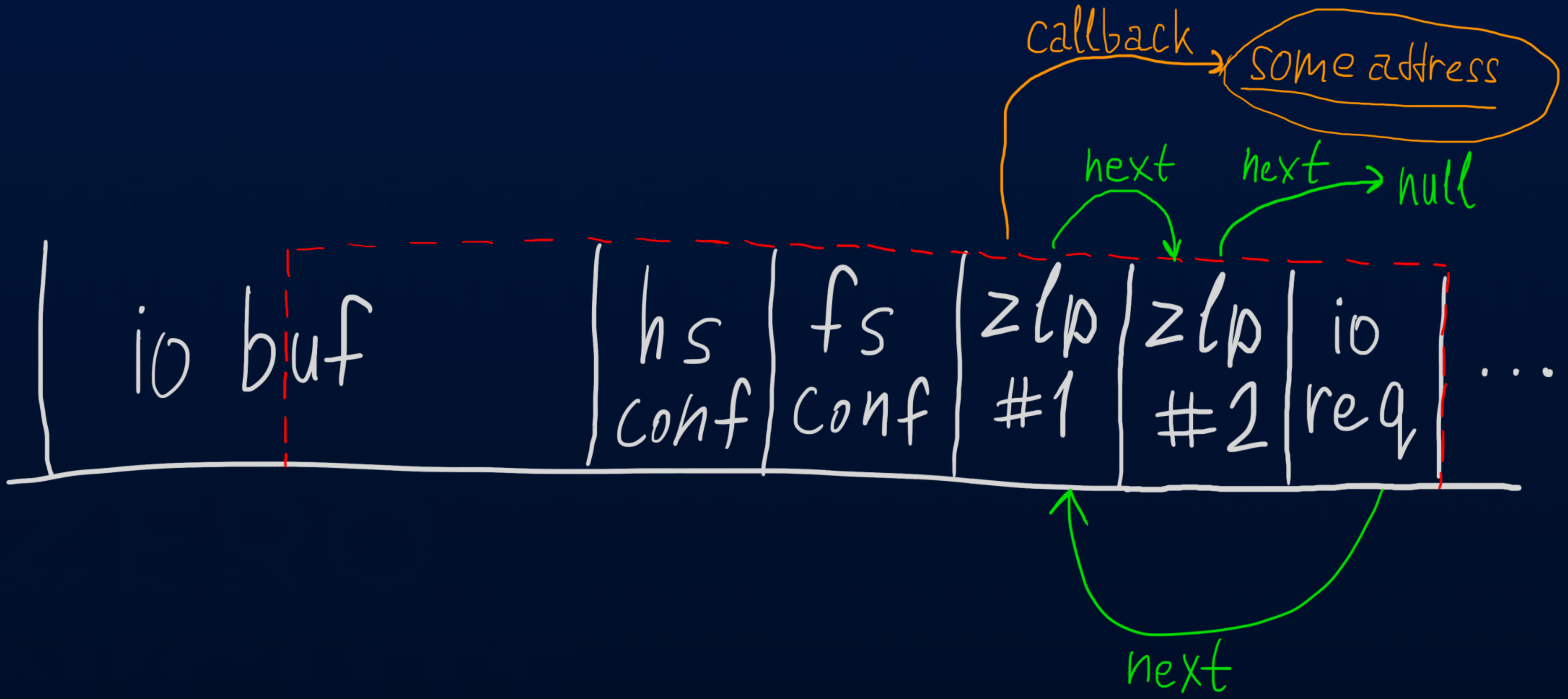
ZERONIGHTSX


overread
io buf
payload
hs conf
fs conf
zlp #1
zlp #2
...
UAF pointer


io buf
hs conf
fs conf
zlp #1
zlp #2
...
UAF pointer


overread
io buf
hs conf
fs conf
zlp #1
zlp #2
. . .
UAF pointer


io buf
hs conf
fs conf
zlp #1
zlp #2
io req
...
UAF pointer


callback
some address
next
next
null
io buf
hs conf
fs conf
zlp #1
zlp #2
io req
...
next

...Perfection

Device is still in DFU, we can read fs conf.:

- If **io_req** is freed, then we hit RET
- If **io_req** is not freed, then we hit RET with a frame shift by 0x20
- You can get other interesting effects on the buffer

Device not in DFU:

- Executed some bs
- Exploit failed

Now we have a clear separation of these two cases

Using Oracle V3, we brute force **standard_device_request_cb**

crash

ret

ret ←

ret

…

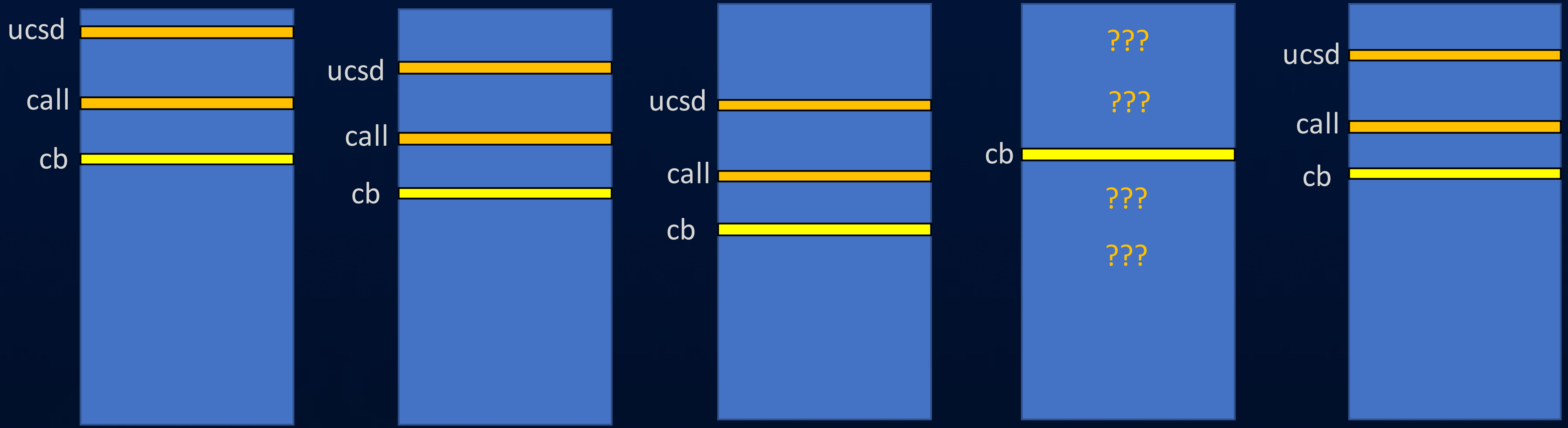
ret

crash

The minimum set of gadgets for dumping

- **usb_create_string_descriptor()**
  - Has some limitations, for example, you cannot dump a sequence of more than 0x80 consecutive non-zero bytes
- **call-gadget** – ***f(x)*** where we control ***f*** and ***x***
  - Used in original checkm8
- How to Catch 'Em All?

Analysis of known SecureROMs



cb - standard_device_request_cb()
ucsd - usb_create_string_descriptor()
call - call-gadget

The analysis showed:

1. The necessary gadgets/functions were present in all SecureROMs
2. The gadgets/functions order is the same in close versions
3. They were at approximately the same distance from each other in different firmware

iBoot-3332.0.0.1.23

- 0x100003E78 – call
- 0x10000AE80 – ucsd
- 0x10000BB5C – cb

iBoot-3401.0.0.1.16

- ???
- ???
- ???

iBoot-3865.0.0.4.6

- 0x10000A404 - call
- 0x10000D390 - cb
- 0x10000D544 - ucsd

ARMA - Advanced Return Map Analyzing

```
crash              ldp x8, x9, [x0, #0x70]
crash              lsl w2, w2, w10
crash              mov x0, x8
crash              blr x9
ret, w/o free      cmp w0, #0
ret, w/o free      csel w0, w0, w19, lt
ret, w/o free      ldp x29, x30, [sp, #0x10]
crash              ldp x20, x19, [sp], #0x20
ret                ret
ret                stp x20, x19, [sp, #-0x20]!
ret, w/o free      stp x29, x30, [sp, #0x10]
ret, w/o free      add x29, sp, #0x10
```

```
ret, w/o free    add x29, sp, #0x10
ret, w/o free    adrp x19, #0x80000000
ret, w/o free    add x19, x19, #0x4f0
ret, w/o free    ldrb w8, [x19, #2]
ret, w/o free    tbnz w8, #0, #0x40                  ; buf[0] = 0x01, buf[2] = 0x01
ret, w/o free    movz w20, #0x200, lsl #16           ; buf[0] = 0x01, buf[2] = 0x01
crash            movk w20, #0x3800
crash            movz w0, #0x200, lsl #16
crash            movk w0, #0x3800
crash            bl func
crash            strb w0, [x19]
crash            orr w0, w20, #0x600
crash            bl func
ret, w/o free    strb w0, [x19, #1]                  ; buf[1] = 0x40, buf[2] = 0x01
ret, w/o free    orr w8, wzr, #1                     ; buf[2] = 0x01
ret, w/o free    strb w8, [x19, #2]                  ; buf[2] = 0xf4
ret, w/o free    ldp x29, x30, [sp, #0x10]
crash            ldp x20, x19, [sp], #0x20
ret              ret
ret              stp x20, x19, [sp, #-0x20]!
```

usb_init_with_controller

```
ret, w/o free    b #0x4c
crash            bl usb_controller_register
crash            adr x0, aAppleMobileDev ; "Apple Mobile Device (DFU Mode)"
crash            nop
crash            bl usb_core_init
reset            cmn w0, #1
reset            b.eq #0x44
reset            bl usb_dfu_init
reset            cmn w0, #1
reset            b.eq #0x44
reset            bl usb_core_start
ret, w/o free    cmn w0, #1
ret, w/o free    b.eq #0x44
```

```
infloop           stp x20, x19, [sp, #-0x20]!
infloop           stp x29, x30, [sp, #0x10]
infloop           add x29, sp, #0x10
infloop           mov x19, x0
infloop           bl  func0
infloop           umull x20, w0, w19
ret, w/o free     bl  time
ret, w/o free     mov x19, x0
                  loop:
ret, w/o free     bl  time
ret, w/o free     sub x8, x0, x19
ret, w/o free     cmp x8, x20
ret, w/o free     b.ls loop
ret, w/o free     ldp x29, x30, [sp, #0x10]
crash             ldp x20, x19, [sp], #0x20
ret               ret
```

Dumping

- Dump our SecureROM using the found:
  - usb_create_string_descriptor()
  - call-gadget from original checkm8
- Each time you try checking the address, you must manually enter the system into a special USB operating mode (DFU)
- Only "strings" can be read (up to the first null byte)
  - It is so slow...
- Cannot read more than 127 bytes (non-zero) at a time
  - There are only two such places in SecureROM and this is not critical
- But it works and allows us to get all addresses from the original checkm8

Results

- checkm8 has been fully ported to T2
- Full dump of SecureROM T2 was received
- Now we can explore T2 at a higher level
- All this without using prototype devices and other "cheats"

Conclusions

- Never give up! Even the impossible at first glance may turn out to be real upon closer examination
- Brute force is still working
- The described method can be useful in other cases

THANKS FOR ATTENTION

QUESTIONS?