HAW HAMBURG

Bachelor Thesis

Marcel Delissen

Design and Implementation of a Demonstrator Bypassing the Readout Protection in an Embedded System

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Marcel Delissen

Design and Implementation of a Demonstrator Bypassing the Readout Protection in an Embedded System

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Bachelor of Science Elektro- und Informationstechnik
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr. Heike Neumann
Second examiner: Prof. Dr. Paweł Buczek

Day of delivery: 13. January 2022

Marcel Delissen

Title of Thesis

Design and Implementation of a Demonstrator Bypassing the Readout Protection in an Embedded System

Keywords

Readout Protection (RDP), microcontroller, hardware security, flash memory, Serial Wire Debug (SWD)

Abstract

This thesis describes the development and implementation of a demonstrator that shows a vulnerability in the readout protection of a microcontroller. For this purpose, the vulnerability in the readout protection is presented, a use case is designed and a demonstrator is developed.

Marcel Delissen

Thema der Arbeit

Entwurf und Implementierung eines Demonstrators zur Umgehung des Ausleseschutzes in einem eingebetteten System

Stichworte

Ausleseschutz, Mikrocontroller, Hardwaresicherheit, Flash-Speicher, Serial Wire Debug (SWD)

Kurzzusammenfassung

Diese Thesis beschreibt die Entwicklung und Implementierung eines Demonstrators, der eine Schwachstelle im Ausleseschutz eines Mikrocontrollers aufzeigt. Zu diesem Zweck wird die Schwachstelle im Ausleseschutz vorgestellt, ein Anwendungsfall entworfen und ein Demonstrator entwickelt.

Contents

List of Figures

List of Tables

Abbreviations

ADI	ARM Debug Interface.
AES	Advanced Encryption Standard (a symmetric-key cryptography algorithm).
AHB	Advanced High-performance Bus.
AMBA	Advanced Microcontroller Bus Architecture.
AP	Access Port.
APB	Advanced Peripheral Bus.
ASCII	American Standard Code for Information Interchange (character encoding standard).
CDC	Communication Device Class.
CPU	Central Processing Unit.
DAP	Debug Access Port.
DP	Debug Port.
ECB	Electronic Codebook Modus (a block cipher operation mode).
FAT12	File Allocation Table (12-bit File Allocation Table).
GUI	Graphical User Interface.
IDE	Integrated Development Environment.
IRQ	Interrupt Request.

JTAG Joint Test Action Group (a standard of test/debug interfaces).

MCU Microcontroller Unit.

MEM-AP Memory Access Port.

MSC Mass Storage Class.

RDP Readout Protection.

ROM Read-only memory (type of non-volatile memory).

SD Secure Digital (memory card).

SDK Software Development Kit.

SRAM Static Random Access Memory.

SW-DP Serial Wire Debug Port.

SWD Serial Wire Debug.

UART Universal Asynchronous Receiver/Transmitter (a standard for asynchronous serial communication.

USB Universal Serial Bus.

VCP Virtual COM port.

Number Systems

Binary numbers are preceded by *0b*. Example: 0b101010
Hexadecimal numbers are preceded by *0x*. Example: 0x2a
All other values are decimal. Example: 42

1 Introduction

Embedded systems, consisting of several components such as microcontrollers, are increasingly used in everyday devices. For example, there are about 50 microcontrollers in an average car [8]. Almost every technical device uses a microcontroller for control. Therefore, security in everyday devices is also becoming increasingly important. These devices are often physically easily accessible and therefore need special protection. Attackers can access internal interfaces through physical access, such as debug interfaces or other direct accesses. Therefore, the microcontroller, especially the memory, should be protected accordingly, as otherwise sensitive data such as passwords or cryptographic keys can be stolen. Additionally, the theft of software also causes considerable economic damage [10].

This readout protection protects the firmware stored in the microcontroller's memory from being readout. Unfortunately, this security mechanism is not error-free or may contain vulnerabilities, such as design flaws to bypass the readout protection. Another possibility is the incorrect configuration of the readout protection [11] [13].

A paper shows that readout protection can be bypassed called "Shedding too much Light on a Microcontroller's Firmware Protection" [13]. The security researchers of this paper found and described a logical vulnerability in the readout protection.

This thesis is about the development of a demonstrator that shows one of the vulnerabilities of the paper. A demonstrator shows how such an attack is possible. The target audience of the demonstrator is primarily non-technical, and it has to be understandable for a large audience. It demonstrates an attack, preferably live, to convince the majority of the need for security mechanisms in such products.

This demonstrator is being developed for NXP Semiconductors Germany GmbH (NXP), which wants to use this demonstrator to introduce customers and visitors.

This way of demonstrating security vulnerabilities or bypassing security measures is not new at NXP. Several demonstrators are maintained by NXP and, in this case, newly developed, such as in the bachelor thesis entitled "A Demonstrator for Optical Fault Injection Attacks" [24], in which a demonstrator is revised and renewed.

This thesis is divided into the following sections. First, Chapter 2 describes the necessary technical basics and technologies. Chapter 3 presents the paper from which knowledge of this vulnerability originated. Then, in Chapter 4, the requirements that NXP places on this demonstrator are presented. Based on these requirements, a concept is presented in Chapter 5, implemented in Chapter 6 . Chapter 7, the conclusion, gives a summary and an outlook for the demonstrator.

2 Basics

This chapter explains the technologies required for the attack and the implementation of the demonstrator. In the first section, the architecture of an ARM Cortex-M0-based microcontroller is described since such a microcontroller is being attacked. The attack is carried out via the debugging interface of the microcontroller and is therefore covered in the Section 2.2. The last Section 2.3 of this chapter covers the Serial Wire Debug (SWD) transmission protocol used by the debugging interface.

2.1 Architecture of ARM

ARM Limited, or ARM for short, is a company that does not manufacture microcontrollers itself but has revolutionised the market in a lasting way. Before ARM launched its Cortex series, the market for microcontrollers was very diverse. Many companies developed and marketed their own controller families with their own instruction sets and their architectures. The design approaches were also very different. As a result, products from the same manufacturer were sometimes incompatible with each other. Among other things, this also affected the debug interfaces. Manufacturers had to provide tools such as compilers, linkers, debuggers and sometimes IDEs in different variants. ARM has unified the market by developing processors and different components. ARM only sells the design of the components. The sale of the designs is called "intellectual property" (IP), and the business model is IP licensing. [4, pp. 21ff.] and [23, pp. 5ff.]

In the following, one of these IPs - the ARM Cortex-M0 MCU - is presented as part of the device under attack.

The ARM Cortex-M0 Microcontroller Unit (MCU) contains all the essential elements such as:

- a Central Processing Unit (CPU) called a processor for short, for control and computing tasks,
- memory systems like flash memory for program ROM and Static Random Access Memory (SRAM) and
- interface hardware, such as a debug interface and other peripherals.

There are many different versions of these microcontrollers, which can be equipped with different memory sizes and peripherals and come in different sized packages. The Cortex-M processors are designed for the mainstream microcontroller market where processing requirements are less critical than power consumption. Their design is suitable for general-purpose applications, which means they can be used in a wide range of applications. [23, pp. 1, 7, 98]

A Cortex-M0 MCU is often packed with many different components. These components communicate via a system bus in which messages are sent over a shared transmission path.
A simplified block diagram with typical elements of a Cortex-M0-based microcontroller is shown in Figure 2.1. The acronyms in the diagram are explained in Table 2.1.
[23, pp. 18f.]

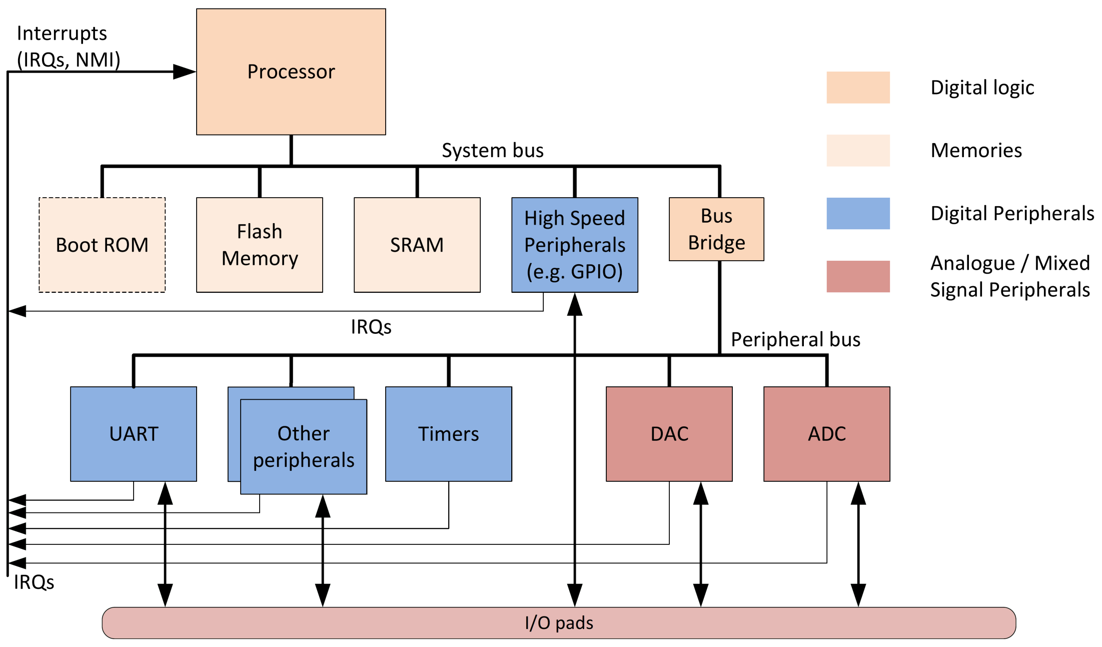


Figure 2.1: Simplified Block Diagram of a Cortex M0-based Microcontroller modified from [23, pp. 19, 35]

Table 2.1: Components in a Microcontroller modified from [23, p. 19]

Item	**Description**
ROM	Read-only memory: Nonvolatile memory storage for program code.
Flash memory	A special type of ROM, which can be reprogrammed many times, typically for storing program code.
SRAM	Static Random Access Memory for data storage (volatile)
GPIO	General Purpose Input/Output: a peripheral with parallel data interface to control external devices and to read back external signals status.
UART	Universal Asynchronous Receiver/Transmitter: a peripheral to handle data transfers in a simple serial data protocol.
DAC	Digital to Analog Converter: a peripheral to convert data values into analog signal level.
ADC	Analog to Digital Converter: a peripheral to convert analog signal-level information into digital form.
Timers	Timers (or Counters) are modules that can count events, measure time intervals and periodically execute parts of the program.

All these components are memory-mapped, and the communication interface is based on read and write operations to particular addresses within this memory mapping. Therefore, the memory and the peripherals are mapped to distinct areas in the address space. All ARM Cortex-M processors have a 4 GB memory address space. This partitioning is hard-wired, and each region has a recommended use to facilitate porting software between different devices. The Cortex-M memory partitioning is shown in Figure 2.2. [23, pp. 65, 97, 167] [4, pp. 28ff.]

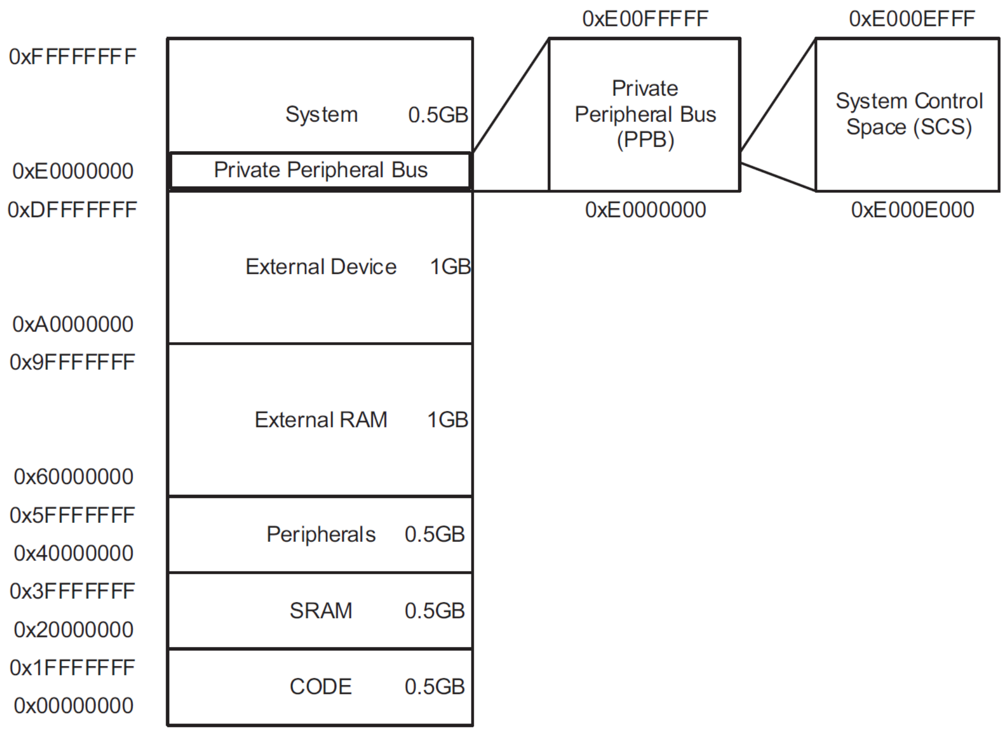


Figure 2.2: ARM Cortex-M0 Memory Map modified from [23, p. 98]

The memory mapped peripherals are available between 0x40000000 and 0x5FFFFFFF, in this range the software can access peripherals (e.g. the I/O lines or serial interfaces). Since this area is used differently on each Cortex depending on the peripherals, the respective Technical Reference Manual must be consulted. [4, p. 28]

An important point is that microcontrollers, even if they have the same peripheral characteristics on paper, can have completely different peripherals and therefore different programming models (e.g. peripheral register definitions), and this varies from chip manufacturer to chip manufacturer. [23, p. 20]

As shown in Figure 2.1, a Cortex-M0 processor is typically divided into two bus systems:

- One is the system bus, which communicates with the memories including ROM, Flash memory and SRAM, and with a few high speed peripherals.
- And secondly, the peripheral bus to which the other peripherals are connected. This can have a different operating frequency than the system bus.

It is quite common for some of the peripherals to be connected to a separate peripheral bus, which is connected to the main system bus via a bus bridge. This bus protocol for the peripheral bus is usually based on Advanced Peripheral Bus (APB), a bus protocol described in the Advanced Microcontroller Bus Architecture (AMBA) [1], an open standard specification for proprietary ARM buses. [23, p. 35]

The system bus is based on a bus protocol called Advanced High-performance Bus Lite (AHB-Lite), which, like the data path in the processor core, is 32-bit wide and is also defined in the AMBA standard. The AMBA standard is developed by ARM and is widely used in semiconductor industry. [23, pp. 29, 32f., 166]

The Cortex-M0 is based on the Von Neumann bus architecture. This means that an instruction fetch and a data operation share the same bus and therefore cannot occur simultaneously. [23, p. 12] [21, pp. 18f.]

In addition to the standard described for the hardware components, the next section introduces another ARM standard, namely the ARM Debug Interface, which has standardised the debug interfaces.

2.2 ARM Debug Interface

This section explains the ARM Debug Interface (ADI), which is needed for the attack. The ADI provides access to debug functionality that is provided by debug components in an embedded System. An implementation of the ARM Debug Interface is called a Debug Access Port (DAP). A DAP provides a debugger with a standard interface to access debug resources in systems that use resource-specific methods to provide their debug information. To access a debug resource, the debugger passes the appropriate resource address information to the Debug Access Port, which executes the request

by selecting the appropriate resource and then accessing resource-specific transport methods that are presented by the system to be debugged.
The block diagram in Figure 2.3 shows how an ADI implementation is connected between a debugger and the system to be debugged.

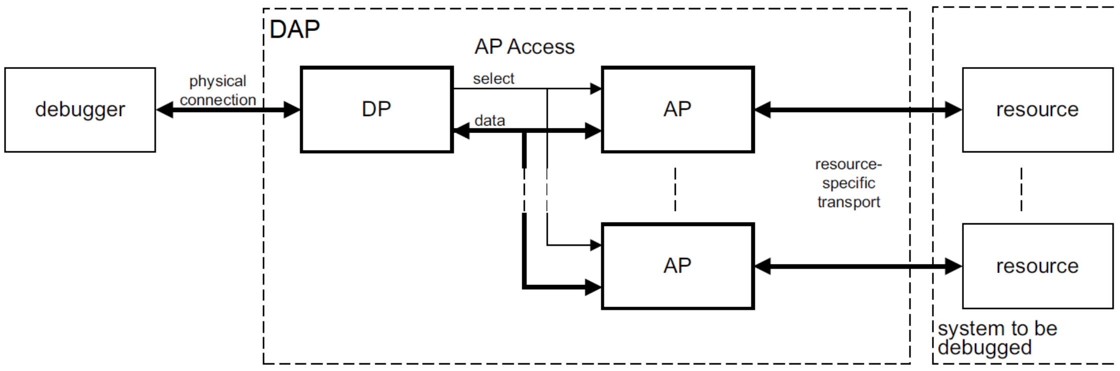


Figure 2.3: Simplified Block diagram of an ADI implementation modified from [2, p. 25]

The Debug Access Port (DAP) shown in Figure 2.3 consists of the following elements:

Access Port
An Access Port (AP) accesses debug information from the system being debugged and passes the information to the Debug Port. Examples of debug resources are the debug registers of the core processor, ROM Table and flash memory.

Debug Port
The Debug Port (DP) provides a debugger with a common interface to access the information that is held in the APs. The Debug Port includes the following elements:

- A physical connection to the debugger. The ARM Debug Interface supports the following physical connection types, with only one interface type occurring at a time:
  - Serial Wire Debug Port (SW-DP), which is covered in Section 2.3
  - JTAG debug port (JTAG-DP)
  - Serial Wire JTAG debug port (SWJ-DP), a combination of the previous two interfaces.

  The JTAG-DP and the SWJ-DP are not discussed below and are listed only for completeness.

- DP registers, hold the required information to support the transport mechanism that is implemented by the Debug Access Port.

The connection between the Debug Port and the Access Ports selects the appropriate debug resource based on the address information provided by the debugger and transports the data between the Access Ports and the Debug Port. [2, pp. 23-26]
In the next section, a protocol that the debug interface uses is explained in more detail, as it is needed for Chapter 3.

2.3 Serial Wire Debug

This section explains the protocol used for the attack.
Serial Wire Debug (SWD) uses an ARM standard protocol, which is described in the ARM Debug Interface Architecture Specification [2] and is a Debug Port for small package microcontrollers. Serial Wire Debug is a standard interface for ARM processor-based devices. SWD replaces the 5-pin JTAG port with a clock pin SWDCLK and a bi-directional data pin SWDIO and provides all JTAG debug and test functions, as well as real-time access to system memory without stopping the processor. [3]

The connection from the debug host to the processor goes through a Debug Port (DP) interface called Serial Wire Debug Port (SW-DP) via an AHB-Access Port (AP), more specifically the Memory Access Port (MEM-AP). The Memory Access Port is connected to the system bus and has access to the memory, such as the flash memory. The memory of the target can be read or written by using the MEM-AP. An overview of the connection from the debug host to the processor is shown in the following Figure 2.4. [22, p. 453]

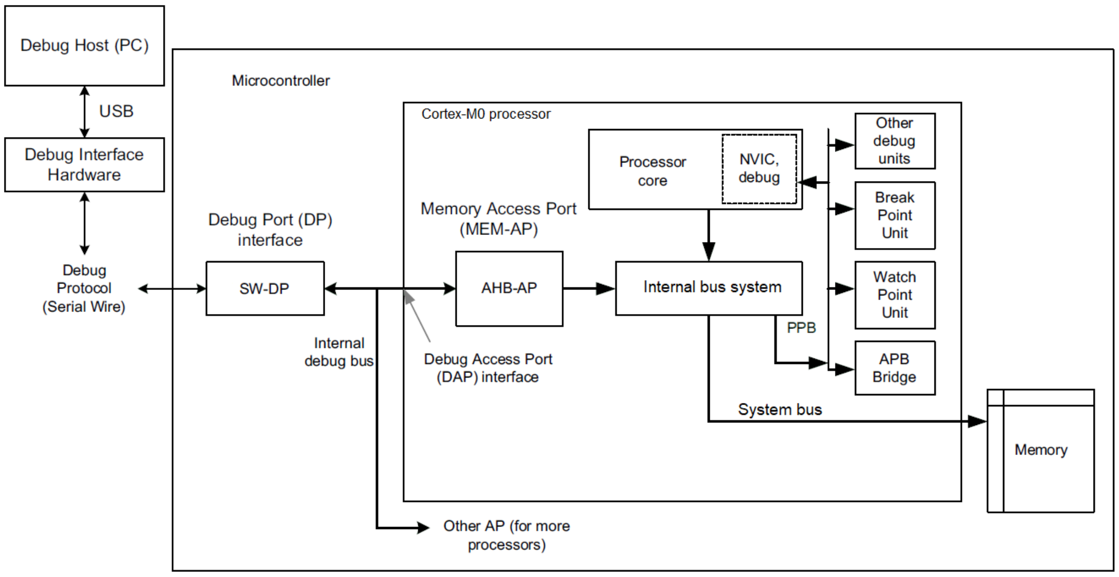


Figure 2.4: Overview of the Connection from the Debug Host to the Processor modified from [22, p. 453] and [20, p. 911ff.]

The next section describes the different phases of data transmission of the SWD protocol.

2.3.1 Basic SWD Operation

The ARM Serial Wire Debug interface uses a single bidirectional data connection and a separate clock to transfer data synchronously. A successful operation on the wire consists of three phases:

1. Packet request
   The external host debugger issues a request to the Serial Wire Debug Port of the microcontroller. The SW-DP is always the target of the request and works as a bridge between the APs and the Host.

2. Acknowledge response
   The target sends an acknowledge response to the host.

3. Data transfer phase
   This phase is only present when a data read or data write request is followed by a valid "OK" acknowledge response. The data transfer is one of:
   - Target to host, following a read request called "RDATA".
   - Host to target, following a write request called "WDATA".

To prevent conflicts, a turnaround period called "Trn" is required when the device driving the wire changes. The default setting is a turnaround period of one clock cycle. For the turnaround period, neither the host nor the target drives the wire. There is a turnaround period between these phases, because the transmission direction changes. When the SW-DP receives a packet request from the debug host, it responds immediately by entering the acknowledge phase. For a write request, there is a turnaround period between the acknowledge phase and the WDATA data transfer phase. Following the WDATA data transfer phase the host continues to drive the wire. There is no additional turnaround period. For a read request, there is no turnaround period between the acknowledge phase and the data transfer phase. There is a turnaround period following the RDATA data transfer phase. [2, p. 106]
All data values in SWD operations are transferred least significant bit (LSB) first. For example, the "OK" Acknowledge response of 0b001 appears on the wire as 1, followed by 0, followed by 0. [2, p. 108]

2.3.2 SWD Protocol Operation

This section explains which phases each SWD operation consists of, which information bits are exchanged in these phases and then describes the transmission of read and write operations.

1. Packet Request Phase

The host starts the transmission with the bits listed in Table 2.2.

Table 2.2: Bits of the Packet Request Phase (modified from [2, p. 110])

Bits	Description
Start	start bit, with value 0b1
APnDP	Access to AP (0b1) or DP (0b0) (alternative name AP$\overline{\text{DP}}$)
RnW	Read (0b1) or Write (0b0) request (alternative name R$\overline{\text{W}}$)
A[2:3]	AP or DP register address bits
Parity	Even parity over the four bits APnDP, RnW and A[2:3]
Stop	single stop bit, this bit is always 0b0.
Park	A single bit. The host must drive the Park bit HIGH to park the line before tristating it for the turnaround period.
Trn	The turnaround period. The target reads this as 0b1.

APnDP is used to select the AP or DP to be accessed. The two bits A[2:3] are used to address the register of the selected AP or DP, and RnW is used to read from or write to the selected register.

2. Acknowledge Response Phase

The acknowledge response is a three-bit target-to-host response and listed in Table 2.3.

Table 2.3: Bits of the Acknowledge Response Phase (modified from [2, pp. 110f.])

ACK[0:2] encoding	Response
0b100	OK
0b010	WAIT
0b001	FAULT

3. Data Transfer Phase

Each data transfer ends with an even parity bit. The Bits of the Frame are listed in Table 2.3.

Table 2.4: Bits of the Data Transfer Phase modified from [2, p. 111]

Bits	Description
WDATA[0:31]	32 bits of write data, from host to target.
RDATA[0:31]	32 bits of read data, from target to host.
Parity	A single parity bit for the data packet.

Diagram of the Read Operation

A successful read operation consists of three phases:

1. An eight-bit read packet request, from the host to the target.
   Followed by a turnaround period "Trn".
2. A three-bit OK acknowledge response, from the target to the host.
3. A total 33-bit data transfer phase, where data "RDATA" is transferred
   from the target to the host.
   Followed by a "Trn".

Figure 2.5 shows the Diagram of a successful read operation.

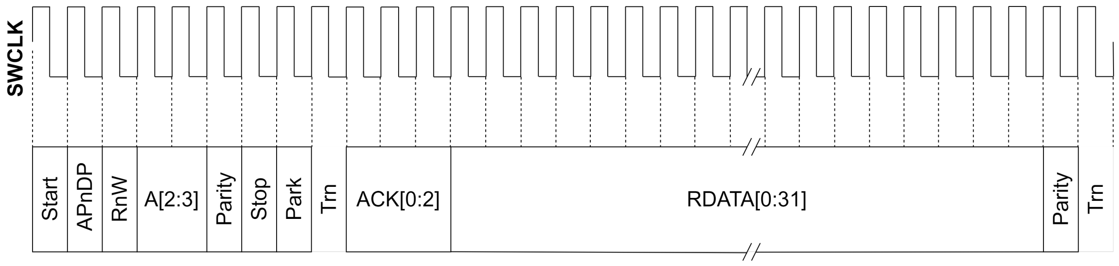


Figure 2.5: SWD read operation modified from [2, p. 112]

Diagram of the Write Operation

A successful write operation consists of three phases and is shown in Figure 2.6

1. An eight-bit write packet request, from the host to the target.
   Followed by a turnaround period "Trn".
2. A three-bit OK acknowledge response, from the target to the host.
   Followed by a "Trn".
3. A total 33-bit data transfer phase, from the host to the target,
   containing the data "WDATA" and a parity bit.
   And also followed by a turnaround period.

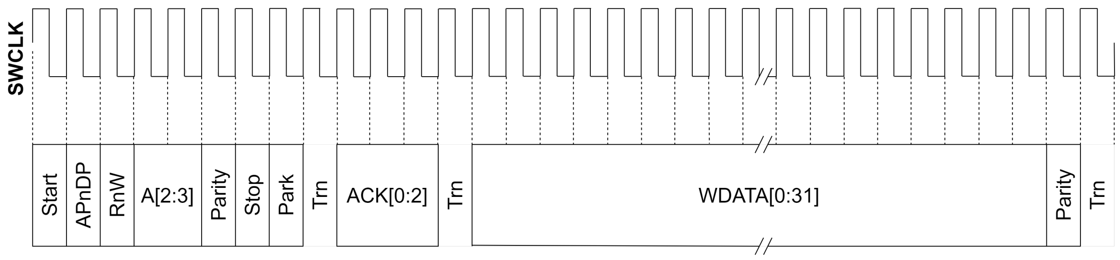


Figure 2.6: SWD Write Operation modified from [2, p. 112]

The information from this section is taken from source [2, pp. 110-114], where further information, e.g. on protocol errors, can be found. Errors are not dealt with in this section, as this section only gives an overview of the functionality of the SWD protocol and is not needed for the following chapters.

2.3.3 SW-DP and MEM-AP Register

Each ADI contains a single DP and can implement multiple APs that are conform with one of the DP architecture versions. This section briefly describes the SW-DP and MEM-AP registers for the DP architecture versions DPv1. For details about how the register is implemented in a specific architecture version, see [2, pp. 48f.].

SW-DP Register

Table 2.5 shows the SW-DP registers and the corresponding addresses, which will be described in more detail below.

Table 2.5: SW-DP Registers and Addresses modified from [2, p. 50]

Address	**Register name**	
	Read	Write
0x00	DPIDR	ABORT
0x04	CTRL/STAT	CTRL/STAT
0x08	RESEND	SELECT
0x0C	RDBUFF	N/A

Debug Port Identification Register

The Debug Port Identification Register (DPIDR) provides information about the DP, such as a part number, revision code, and version of the implemented DP architecture.

Abort Register

The ABORT Register forces an AP transaction abort and on a SW-DP it is also used to clear error and sticky flag conditions.

Control/Status Register

The Control and Status Register (CTRL/STAT) is a 32-bit register that is used to control and obtains status information about the DP and is shown in Figure 2.7.

Only the bits CDBGPWRUPREQ and CSYSPWRUPREQ are given special attention, as these bits are necessary for initialising the debugger.

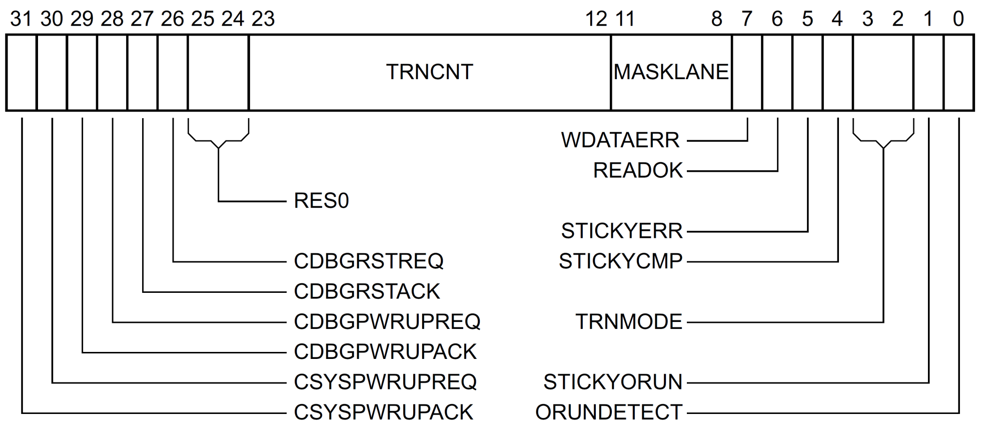


Figure 2.7: SW-DP Control and Status Register (CTRL/STAT) from [2, p. 56]

- Bit [28] CDBGPWRUPREQ:
  Debug powerup request. This bit controls the CDBGPWRUPREQ signal. This signal is required for the power controller and is used to start up and activate the clocks in the debug power domain.

- Bit [30] CSYSPWRUPREQ:
  System powerup request. This bit controls the CSYSPWRUPREQ signal. This signal is required for the power controller and is used to start up and activate the system power domain.

It is necessary to generate a debug and a system power-on request, otherwise the DAP will not answer any requests.

Resend Register
This register does not capture new data from the AP while a read operation is performed on "RESEND" but returns the AP's last read or DP RDBUFF read.

Select Register
SELECT is a 32-bit DP architecture register and selects

- an Access Port and the active register banks within that AP or
- the DP address bank.

Figure 2.8 shows the SELECT register.

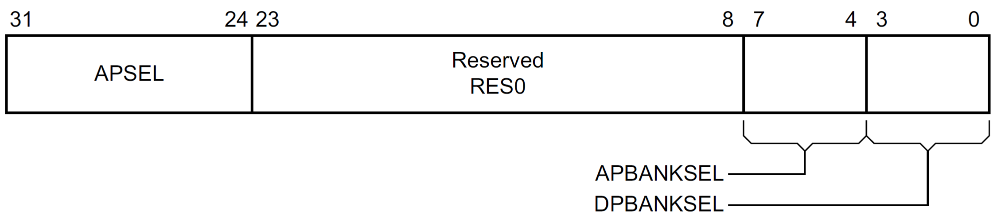


Figure 2.8: SW-DP Select Register from [2, p. 70]

- Bit [31:24] APSEL Selects the AP with the ID number APSEL
- Bit [7:4] APBANKSEL Selects the active four-word register bank on the current AP.
- Bit[3:0] DPBANKSEL Debug Port address bank select.

Read Buffer Register
The Read Buffer (RDBUFF) presents data (typically 32-bit) captured during the previous AP read and allows the value to be returned repeatedly without generating a new AP access.

MEM-AP Register

The Memory Access Port module is connected to the internal bus system of the processor and has access to the memory. This allows the debugger to access all the memories components, peripherals, debug components, and debug registers of the processor. Table 2.6 shows the MEM-AP registers and the corresponding addresses.

Table 2.6: MEM-AP Registers and Addresses modified from [2, p. 151]

Bank	Address	Register name
0x0	0x00	Control/Status Word (CSW)
	0x04	Transfer Adress (TAR)
	0x08	RESERVED
	0x0C	Data Read/Write (DRW)
0x1	0x10	Banked Data 0 (BD0)
	0x14	Banked Data 1 (BD1)
	0x18	Banked Data 2 (BD2)
	0x1C	Banked Data 3 (BD3)
⋮		
0xF	0xF4	RESERVED
	0xF0	Configuration Reg. (CFG)
	0xF8	Debug Base Addr. (BASE)
	0xFC	Identification Register (IDR)

Identification Register
Every Access Port must implement an Identification Register (IDR) which identifies the AP and provides information about the AP, such as a class type and revision code. An IDR value of zero indicates that there is no AP present.

Control/Status Word register
The Control/Status Word register CSW holds control and status information for the MEM-AP.

Transfer Address Register
The Transfer Address Register TAR holds the address for the next access to the memory system, or set of debug resources, which are connected to the MEM-AP. The MEM-AP can be configured to automatically incremented the TAR value after each memory access. Reading or writing to the TAR does not cause a memory access.

Data Read/Write register
The Data Read/Write register DRW is used for memory accesses:

- Writing to the DRW initiates a write to the address specified by the TAR.
- Reading from the DRW triggers a read request from the address stored in the TAR. When the read access completes, the value is returned from the DRW.

Banked Data Registers
The Banked Data Registers, BD0-BD3, provide direct read or write access to a block of four words of memory, starting at the address that is specified in the TAR.

Configuration register
The Configuration register CFG register hold information about the configuration of the MEM-AP.

Debug Base Address register
The Debug Base Address register BASE is a pointer into the connected memory system. It points to one of:

- The start of a set of debug registers for the single connected debug component.
- The start of a ROM Table that describes the connected debug components.

This brief summary of the MEM-AP registers does not include cross-references to the detailed register descriptions. For more information about these registers, see on page 170 and following in source [2].

2.3.4 Example of Reading from the SW-DP

This subsection describes which bit sequence must be sent to read the the Debug Port Identification Register (DPIDR) located in the SW-DP. This register provides information about the Debug Port, such as a part number and revision code. Table 2.7 shows the sequence of bits that must be sent to read the register entry. Reading this register is a good way to check if the initialisation was successful and a connection exists.

Table 2.7: Example of Reading from DPIDR

Bit	Start	APnDP	RnW	A[2:3]	Parity	Stop	Park	Trn
Value	1	0	1	0 0	1	0	1	

If 32-bit data are received after the OK acknowledge response, the connection to the SW-DP works properly. For a Cortex-M0 MCU, the default response is 0x0bb11477. [20, p. 911ff.]

2.3.5 Example of Reading from Flash Memory via the MEM-AP

This section shows which steps are necessary to read data from the flash memory with a debugger via SWD. These steps are essential for the attack and will be described later as minimal communication and are therefore listed briefly.

1. System reset (optional): Initially, a power cycle is performed to reset the system.
2. Initialising the debug interface: The steps described in the Arm Debug Interface Specification [2, pp. 123ff.] are followed.
    - First, the SWD interface is reset by applying the reset pattern to SWDIO and SWCLK.
    - Secondly, reading the IDCODE from the Serial Wire Debug Port is recommended; see Section 2.3.4.
    - Thirdly, set the System powerup request (CSYSPWRUPREQ) and Debug powerup request (CDBGPWRUPREQ) in the CTRL/STAT Register of the SW-DP to fully initialise the debug interface, see Section 2.3.3.
3. Setting the access width to 32 bits (optional): Although this is optional, it is recommended to switch to 32-bit mode, as this allows a whole word of 32 bits to be read. To switch to 32-bit mode, set the value of the size field in the Control/Status Word Register of the AHB Access Port, here the MEM-AP, to 0x02.
4. Setting the read address of the flash: The address to be read is written to the Transfer Address Register of the MEM-AP.

5. Triggering the flash read access: A read access to the Data Read/Write Register of the MEM-AP triggers the read access of the MEM-AP via the system bus, which reads the flash memory.
6. Reading the data: The result of the previous access is read from the DP Read Buffer Register. An "SWD OK" response is returned if flash memory access was granted on success. In case of an error, "SWD ERROR" is returned. In the case of "SWD OK", the word was correctly read from the flash memory and stored in the DP read buffer register, where it can be read by the debugger.

Reading the DP Read Buffer Register is also possible if the CPU switches off the system bus in the meantime. This note is important for the following chapter. [13, pp. 9f.]

3 Basis of the Attack

The vulnerability used for the demonstrator is based on a paper of the Fraunhofer Institute for Applied and Integrated Security, Fraunhofer AISEC for short. Fraunhofer AISEC supports companies in securing their systems and products and researches in the field of embedded systems and hardware components, among other things. The paper is entitled ”Shedding too much Light on a Microcontroller's Firmware Protection” [13] and investigates the safety concept of a specific microcontroller family.

The paper reports on the growing interest in testing the security mechanisms of newer microcontrollers, including the ARM Cortex-M-based STM32 series. The widespread use of these devices has finally sparked in their security mechanisms. Also, by the time the paper was published, there were no publicly available penetration test results for STM32 devices. Therefore, the paper's authors conducted a thorough security analysis of the STM32F0 sub-series, paying special attention to firmware readout protection.

The readout protection protects the memory in which the firmware is stored. This protection is intended to protect against unauthorized access. Flash readout protection is the root of system security because reading the firmware protects against access to the system's security configuration. Bypassing the readout protection can completely compromise other security mechanisms.

The paper begins by introducing the security system, which consists of three Levels. The different Readout Protection (RDP) Levels differ as follows.

RDP Level 0 is the default configuration and is not restricted in any way. The debug interface is active and allows full access to the device. Normally this level is only used for development. This protection should be set by the developer after the firmware is loaded to the embedded Flash memory.

RDP Level 1 keeps the debug interface active, but restricts the access to the flash memory. As soon as a debugger is connected, the flash memory is locked. A read access generates a Hard Fault interrupt and a bus error. The protection can be increased to RDP Level 2, but can also be downgraded to level 0, in which case the entire flash memory content is erased.

RDP Level 2 is the most restricted and offers the highest security. Debug access is completely disabled by permanently shutting down the interface. This level is irreversible and cannot be downgraded. Although RDP Level 2 offers the best protection, RDP Level 1 is preferred. According to the paper, experience shows that companies do not like to lock down their devices completely as this makes it difficult to troubleshoot errors and failures. In addition, the manufacturer cannot analyse defective devices with RDP Level 2. Overall, this leads to most devices being set to RDP Level 1.

The readout protection is part of the system configuration of the microcontroller and is stored in a dedicated "option byte" section. In it, the three available RDP levels are encoded with 16 bits of non-volatile memory. The 16 bits are implemented as two consecutive bytes called "RDP" and "nRDP". In any intended configuration, nRDP represents the bitwise complement of RDP. Table 3.1 shows the mapping of each RDP and nRDP setting to the configured RDP level.

Table 3.1: Readout Protection Options of the STM32F0 Series according to the Datasheet [20, p. 64]

nRDP	RDP	Resulting protection
0x55	0xAA	RDP Level 0
Any other combination except the values of levels 0 and 2		RDP Level 1
0x33	0xCC	RDP Level 2

RDP Level 0 and RDP Level 2 are each represented by exactly one complementary byte pair. Any other configuration, including non-complementary byte pairs, is represented by RDP Level 1 by default. The factory setting is RDP Level 0.
This description is based on sources [13, pp. 2f.] and [20, p. 64]

Fraunhofer AISEC has demonstrated that this readout protection can be bypassed. This approach is now explained and the setup is shown in Figure 3.1.

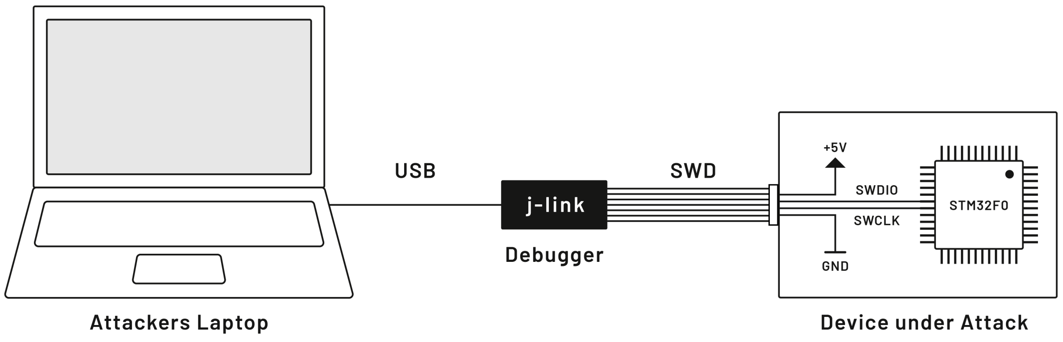


Figure 3.1: Setup of the Attack

An ARM M0-based STM32F0 microcontroller was analyzed, in which protection against flash access is given by RDP level 1. As soon as a debugger tries to read from the flash memory via SWD, this triggers a Hard Fault interrupt, and the CPU cannot process any further commands. The flash protection logic must be reset by a power cycle.

A closer investigation shows that the readout protection is not triggered for all SWD requests but only for the system bus's access. Access to SWD interface-internal registers such as the Debug Port Identification Register (see Section 2.3.3) remains inconsequential, and the system continues to run. However, the flash is deactivated when the debugger accesses another module such as peripherals, SRAM, or flash memory via the system bus. The debugger triggers a transfer on the system bus by reading from or writing to the SWD Memory Access Port Data Read/Write Register (see Section 2.3.3). A more detailed analysis shows that the AHB transaction and thus the lockdown is triggered by the last rising SWCLK edge of the corresponding SWD packet transmission.

However, the paper shows that reading from memory is still possible with RDP activated. A minimum initialization of the debug interface is required to mount the attack, and the SWD communication needs to be reduced to a bare minimum. Experiments show that the default software shipped with SWD Debugger (onboard Debugger and the Segger J-Link Debugger) triggers the RDP protection while connecting. By reducing the communication, the SWD read request can be answered without triggering the flash protection. The readout protection triggers the Hard Fault interrupt too late, which means the request can be answered. It depends on the system busload whether the read request succeeds or fails. The paper further investigated the relationship between a successful attack and the busload and came to the following conclusion. CPU instruction and data fetches have priority over debugging access during the arbitration. Thus, the debugger has to wait for a free cycle on the bus to place the request. If the debugger gains instant access to the bus, the access takes place before the flash protection locking is triggered. The flash protection logic is triggered two cycles too late. Thus, the firmware can be extracted at an average rate of 45 bytes per second. The largest STM32F0 microcontrollers with up to 256 kByte of flash memory can be read entirely in less than two hours. Further data and the methodology used for these results can be found in the paper from the Frauenhofer AISEC by Obermaier and Tatschner [13].

An assumption about the timing of the attack is shown in Figure 3.2. It shows how the debugger makes a read request to the flash memory via the system bus, received and processed by the RDP. While the flash memory starts processing the request, the RDP sends an IRQ to the CPU via the interrupt line and triggers a bus fault exception. However, the flash memory could respond before the exception and thus fully answer the request.

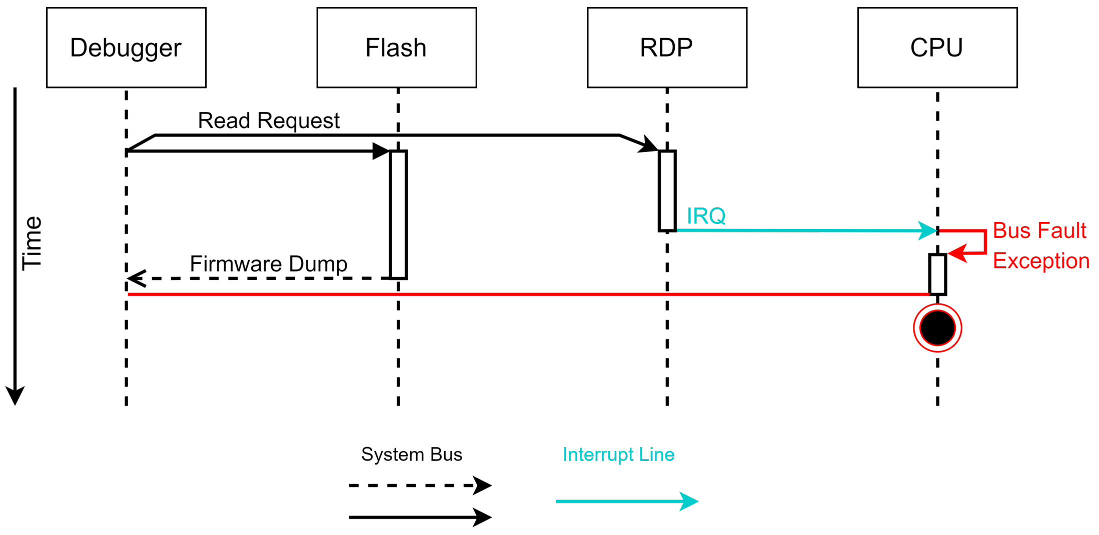


Figure 3.2: An Assumption about the Timing of the Attack

4 Requirements Specification

This chapter lists the requirements that NXP places on a demonstrator. The basic requirements for a demonstrator are as follows. A demonstrator is intended to be a live demo showing an attack in real-time without preparations. The attack should be understandable for a broad audience, and the consequences should be impressive. The demonstrator intends to show how important security measures are and that it is not enough to trust existing security mechanisms because they can possibly be bypassed or hacked. Therefore, the explanation of the attack must be greatly simplified, as the audience does not necessarily have the required knowledge. A short and straightforward explanation of the demonstrator and presentation of the attack is required for this.

The requirements are divided into three sections. At the beginning, the use case of the demonstrator is described. Which is followed by the hardware and software requirements.

4.1 Use Case Requirements

The requirements for the use case, as already partially described, are as follows:

- The demonstrator and the attack should be easy to understand and follow. Only widely known technology should be assumed.
- It should be impressively shown how essential firmware protection is.
- It should be possible to demonstrate it live and therefore require little time and effort, ideally, in a ca. 30-minute presentation

4.2 Software Requirements

The requirements for the software are also that it should be easy to understand. The software should be executable on a modern and widely used operating system such as Windows 10. The demonstrator should communicate via a straightforward Graphical User Interface (GUI). The operation of the software should also be intuitive, and any errors or system failures should be self-correctable and remediable, if possible. Another requirement is that the user interface is resizable and the content displayed be scalable. This requirement is necessary to change the window size for large or high-resolution display devices.

4.3 Hardware Requirements

The requirements for the hardware are to keep a simple structure. This means that it must only be limited to the connection of plugs. In the best case, these are also protected against incorrect connection, like USB. This also means that, as with software, errors and system failures can be corrected and fixed by themselves. Again, the hardware should also be easy and intuitive to use as with the software. Therefore, if possible, no additional hardware should be needed other than that included with the demonstrator. The only exception, of course, is a computer or laptop. Another requirement for the demonstrator is that it needs to be easily and cost-effectively reproducible in case the demonstrator breaks down.

This also includes that any errors or system failures can be corrected and dealt with by themselves, if possible. Also, the handling of the demonstrator and implementation should be intuitive.

4.4 Specifications - Summary

In this section, the requirements already specified are summarised in a list.

- The attack must be demonstrated live in an about 30-minute presentation, including setup.
- The use case should be as comprehensible and convincing as possible
- Explanations should be simple and easy to understand
- The consequences of the attack clearly show that further or better protective measures are necessary
- Communication should take place via a clear Graphical User Interface
- The demonstrator should be simple in design and assembly
- The demonstrator should be easy to transport
- The demonstrator must be easily and cost-effectively duplicable
- The demonstrator must support Windows 10 as it is very well known.
- Errors or system failures should be self-correctable and remediable
- Reverse polarity protected connections

In the next section, a concept is developed based on these specifications.

5 Concept

This chapter describes the concept of the demonstrator and is divided into five sections. In the beginning, the use case is presented, which is necessary for understanding the following overview of the demonstrator. In the Use Case section, the password-protected hard disk is introduced. Then the concept of the password-protected hard disk is described, followed by the section describing the presentation. In the last section, the hardware necessary for the attack is described.

5.1 Use Case

In the use case, sensitive data must be read from the microcontroller's flash memory by exploiting the vulnerability described in Chapter 3. Therefore, a password-protected storage medium, such as a USB hard drive, is chosen for the demonstrator. This technology and its handling are well known to the broad public, and the consequences of bypassing the protection are severe. This Use Case meets the requirements of designing an easy-to-understand use case and is also easy to handle, as USB hard drives usually only require a USB cable to be connected. Password protection is also easy to explain, as this procedure is widely known. The presentation of the password-protected hard disk will only take a few minutes so that there is still enough time for the live demonstration of the attack, and thus all the requirements are met. It is essential to mention that the password evaluation occurs on the hard disk, i.e. on the microcontroller. With this design, it is possible that the password stored on the hard disk can be read out for decryption. In this way, the password can be read from memory, as this must be available for checking against an entered password.

5.1.1 Use Case Description

This section describes how to use the password-protected hard disk, and an overview of the necessary components can be seen in Figure 5.1. This overview shows that the hard disk is connected to a computer via USB and that no additional hardware is required, which corresponds to the requirements. A straightforward GUI is displayed on the computer screen, which is required to unlock and mount the hard drive. The password is entered into this software and sends entered passwords to the hard drive.

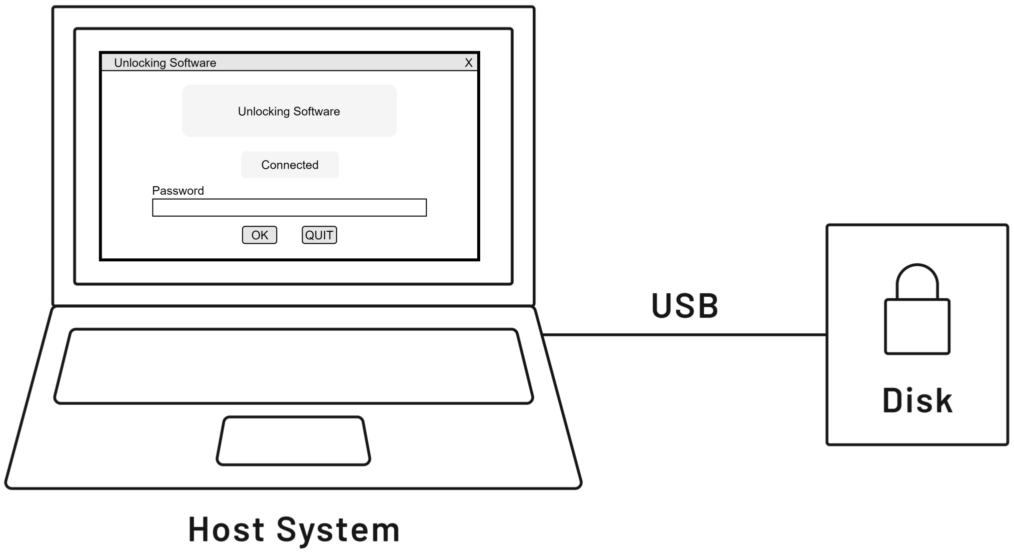


Figure 5.1: Setup Demo Disk

Prerequisites: The hard disk is initially encrypted so that the data cannot be read, and no hard disk can be mounted at this time.

Post-conditions: After the hard disk is disconnected, it is encrypted again so that data cannot be read.

Primary way: The hard disk is connected to the computer via a USB cable. The unlocking software is then started and shows the successful connection to the hard disk, and the password for unlocking can be entered. If an incorrect password was entered, the software displays this, and the password can be entered again. The hard disk remains encrypted at all times and cannot be mounted. There is no limit to the number of incorrect passwords that can be entered, and there are no consequences. The unlocking software indicates that the correct password has been entered and then starts Windows File Explorer. Only at this point the hard is disk decrypted and accessible to the operating system. The File Explorer is started at the mount point of the hard disk to allow immediate access to the hard disk and increase user-friendliness. The unlocking software exits shortly afterwards.

Alternatively, the unlocking software can be started before the hard disk is connected. The software indicates that the hard disk is not yet connected and signals that it is ready to receive the password when the hard disk is connected. This method also leads to successful completion.

5.2 Concept Overview

This overview presents the different components of the demonstrator. A distinction is made between the following parts.

- The password-protected hard disk, which includes the unlocking software.
- The presentation in which the hard disk is introduced, the attack is vividly described and visualised, and the attack is demonstrated live.

The hard disk, especially the host application needed to communicate with the hard disk, is developed as a stand-alone program and is not integrated into the presentation. This separation increases authenticity, as the hard disk could be used independently of the presentation. This separation also allows the scenario of a fictitious developer of this password-protected hard disk, who only relies on the readout protection of the hardware used during development and does not take further security measures. As a result, the password used for encryption is stored in plaintext in the code and thus in the microcontroller's flash memory. Exploiting the vulnerability shows that the readout protection is insufficient, and the password can be read out, making the encryption of the data obsolete. This problem only exists because no further security measures have been taken to make it difficult to reconstruct the password.

5.3 The Password-Protected Hard Disk

This section is divided into the hard disk hardware, the software running on it, and the software to unlock the hard disk. The unlocking software is presented first. The firmware and the hardware follow.

5.3.1 Unlocking GUI

The main task of the unlocking GUI is to send the entered password to the hard disk for evaluation. This software is a GUI because the user can enter data, i.e. the password, and receive feedback on whether this is correct or incorrect. The following points are necessary for this communication, particularly for program intuitiveness. These points are shown in a concept sketch in Figure 5.2.

1. For simplicity, the GUI consists only of a description text field (1 a) and an input field (1 b) for the password.

2 a. The connection status is be displayed immediately after the program starts and regularly checked for up-to-dateness and changed if necessary.

2 b. The software independently establishes a connection to the hard disk and displays the connection's current status.

3. If the hard disk is not connected, entering a password is impossible, indicated by a grayed-out input field.
4. The input is hidden by using a password field.
5. The entry process is completed by pressing the Enter key, and the password will be automatically transferred to the hard disk.
6. Feedback is given as to whether the password was entered correctly.
7. The GUI window automatically moves to the foreground and is centered.

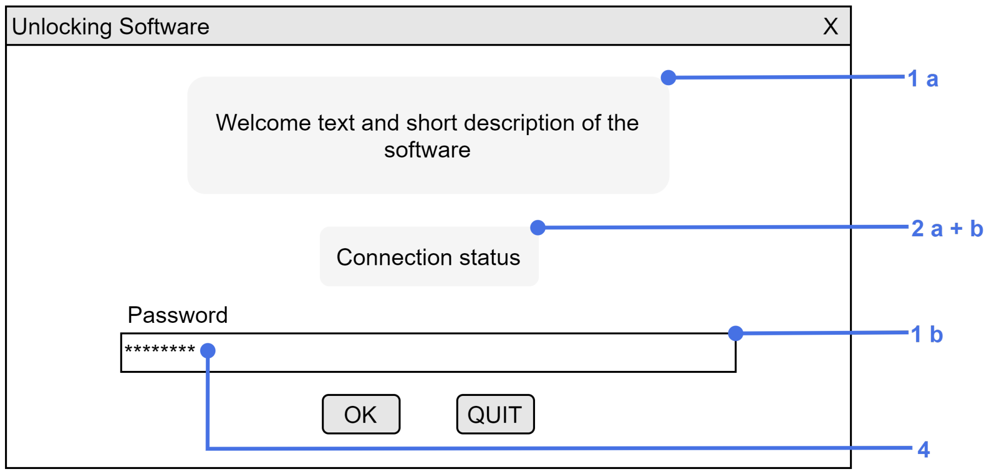


Figure 5.2: Sketch of the Unlocking GUI

5.3.2 Firmware of the Hard Disk

This section introduces the concept of firmware. The firmware is the software executed on the microcontroller and implements the functionalities of a password-protected hard disk. Since the password evaluation takes place on the microcontroller, it must communicate with the host system. In order to make this particularly user-friendly and to imitate a real hard disk as closely as possible, communication via USB is necessary. This approach is in line with the requirements. So there is only one interface and one cable for communication between the host and the hard disk. Everything necessary is realised in software. Therefore, the software has two states. The initial state after start-up is the communication state, in which it remains until the correct password has been received. In this state, the user data are encrypted and cannot be accessed because no hard disk can be mounted yet. The second state is the mass storage state, which can only be reached with the correct password and makes the hard disk available to the host as a USB storage medium and can be mounted as a hard disk. The program flow is shown in Figure 5.3, which shows the states with the corresponding USB class of the Communication Device Class (CDC) and the Mass Storage Class (MSC). More about the different USB classes can be found at [5, pp. 169-189].

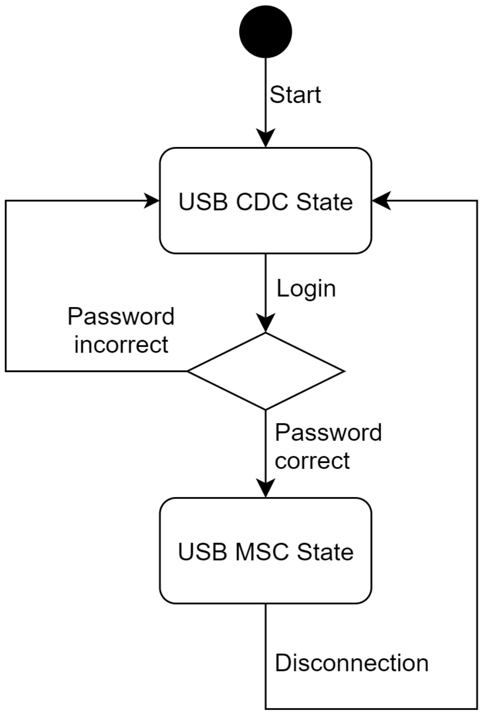


Figure 5.3: Conceptualised Program Flow of the Firmware

Intermediate steps such as decrypting the data between the two states have been omitted for the sake of clarity. A modern encryption standard such as AES presents the security authentically. A full disk encryption (FDE) is remarkably authentic and is therefore conceptualised.

A communication protocol is required to send the password to the hard disk. The following commands with corresponding responses need to be implemented to meet the use case description and are listed in Table 5.1.

Table 5.1: Protocol for Password Evaluation

Message from Hostsystem	**Response from Microcontoller**	**Explanation**
"hello\n" or "HELLO\n"	"HELLO WORLD!!!"	To test the communication
"PASSWORD****\n" or "password****\n"	"WRONG_PW!\r\n" "RIGHT_PW!\r\n"	**** represents the password

Since the protocol is not overly complex, it is implemented using serial communication supported by USB's CDC class. Using USB CDC has the advantage that all communication requires only the USB cable. A message containing the keyword "password" and the password to be evaluated must be sent by the host to verify a password by the microcontroller. The keyword is not separated from the actual password and is sent together as one word. The non-printable character Newline (or Line Feed) "\n" ends the string and thus shows a unique end.

The firmware must first connect to the host as a USB communication device and process received messages. If an incorrect password was sent, the message "WRONG_PW!\r\n" must be sent to the host, otherwise "RIGHT_PW!\r\n" must be sent, and the USB class must be changed from the CDC to the USB MSC. As soon as the device is disconnected, everything must be reset, and the password must be entered again.

5.3.3 Hardware of the Hard Disk

Only ready-to-use development boards were used to reduce the development effort to a minimum. The choice of hardware is limited to the STM32F0 series, as the security vulnerability is restricted to this series. Since the use case is a portable password-protected USB hard drive, the board must have a USB port that supports USB device mode. The microcontroller can change the USB device classes in this mode, implementing the communication protocol described in the previous section. For the STM32F0 series, only one compatible development board is offered, namely the "32F072BDISCOVERY" with an STM32F072RB MCU. This development board is shown in Figure 5.4 and more information about the development board and the MCU can be found on the manufacturer's website [16].
Since this board is provided with unnecessary equipment such as sensors, buttons and an onboard debugger, a smaller board equipped with the same Microcontroller Unit was chosen. In particular, the second USB port, which is for the onboard debugger, could lead to irritation; therefore, the smaller board with only one USB port and no onboard debugger was chosen and is shown in Figure 5.5.

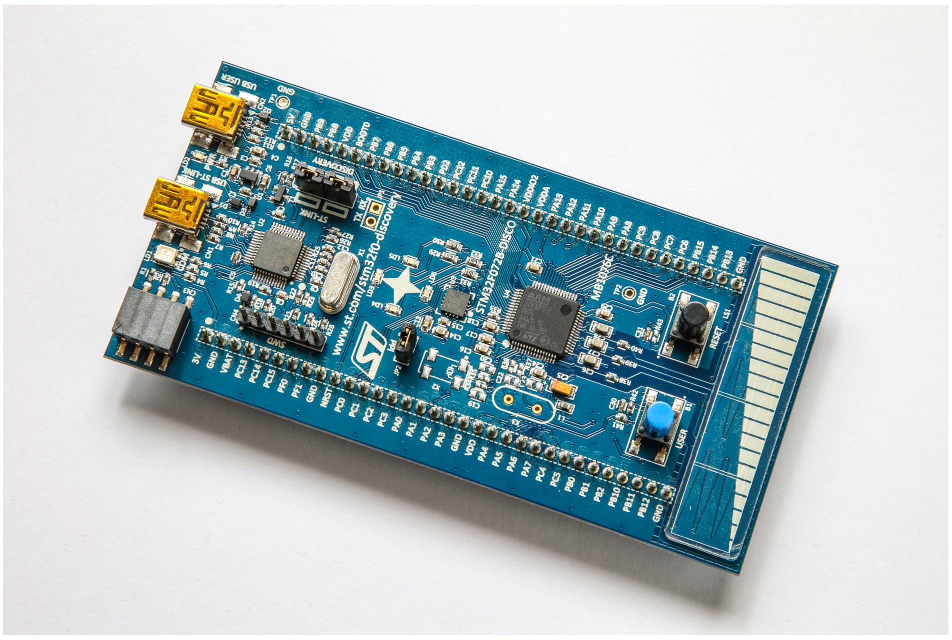


Figure 5.4: STM32F0 Development Board 32F072BDISCOVERY

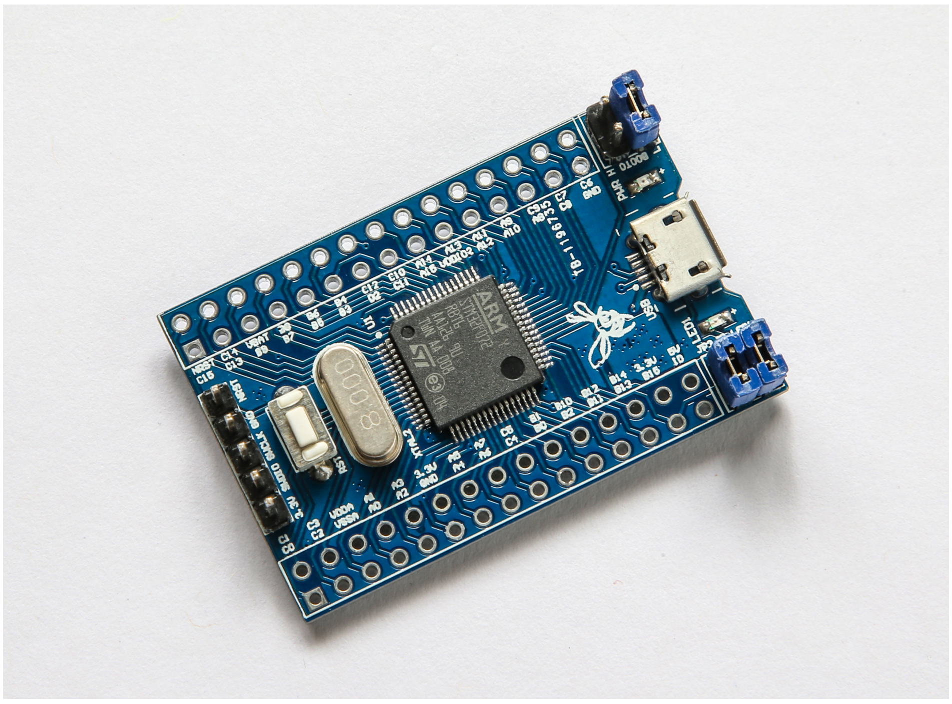


Figure 5.5: Demonstrator Board with an STM32F072RB

5.4 Presentation of the Demonstrator

The main task of the presentation is to introduce the password-protected hard disk, explain the attack and demonstrate it live. For this purpose, the presentation will consist of slides shown one after the other. These slides are high-resolution graphics so that the presentation can be displayed well on high-resolution and large display devices, as specified in the requirements. The presentation is divided into five parts, explained and conceptualised in the following sections.

1. Presentation of the hard disk
2. Demonstration of the hard disk
3. Setup of the attack
4. Explanation of the attack
5. Demonstration of the attack

The sequence of the individual parts is explained in the following and why it is chosen in this way. First, it is shown that it is a hard disk, and it is explained that a password protects it and that this password must be transmitted to the hard disk so that the evaluation takes place on the hard disk (part 1). This introduction is made because the password of this hard disk will be stolen later. The demonstration of the hard disk is followed, which makes the explanation clearer and easier to follow, as the steps are followed once (part 2). After the hard disk has been established, it is pointed out that the password is stored on the hard disk for the evaluation, thus making it a target for attack. The presentation of the attack setup (part 3) builds the bridge to the explanation of the attack (part 4). The attack process is then explained simplified, followed by a live demonstration (part 5).

All slides are uniformly structured, and Figure 5.6 shows the concept sketch of the master page from which all others are derived.

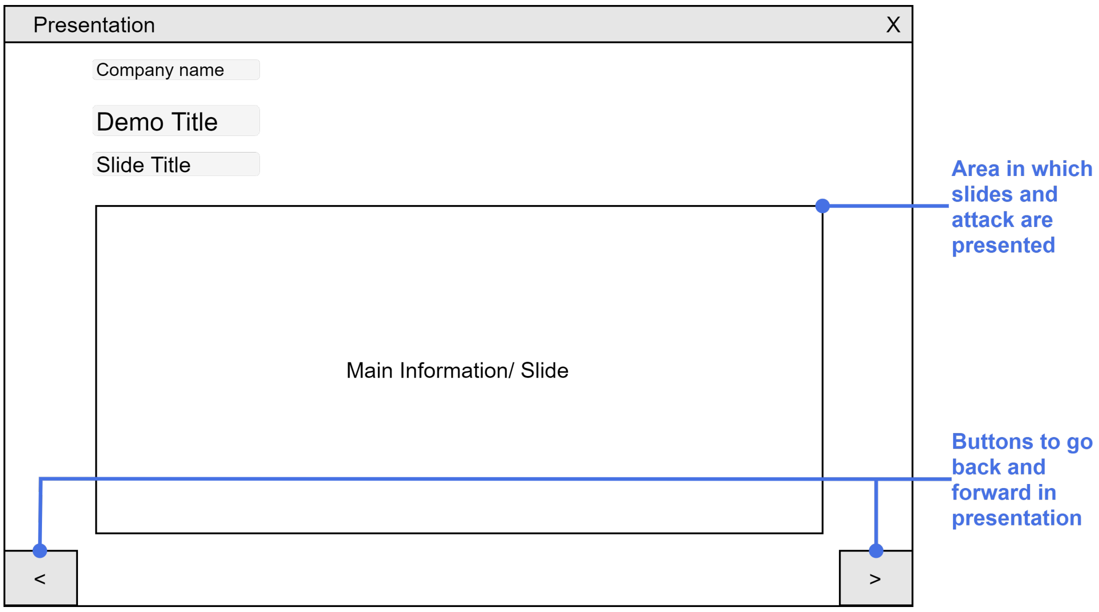

Presentation
X
Company name
Demo Title
Slide Title
Main Information/ Slide
<
>
Area in which slides and attack are presented
Buttons to go back and forward in presentation

Figure 5.6: Sketch of the Presentation GUI - Master Slide

It is possible to use both the buttons and the arrow keys to make the operation of the presentation as intuitive as possible. This GUI is structured like a classic presentation with slides, in which the live demonstration is integrated.

5.4.1 The Hard Disk Part

Parts 1 and 2 of the presentation are conceptualised in this section. For Part 2, a variation of Figure 5.1 (Subsection 5.1.1) is used, and it is shown that the password has to be sent to the hard drive to unlock it, which is illustrated by sending a password and then opening the lock. For the second part, there will be a slide with a button to start the unlocking software. After starting, the hard disk and software functionality is demonstrated once. First, a wrong password is entered, showing that no hard disk is available. Then the correct password is entered, showing that the hard disk is available.

5.4.2 The Attack Part

This section explains and conceptualises parts 3 and 4 of the presentation. For part 3, a simplified variation of Figure 3.1 from Chapter 3 is used. Part 4, the explanation of the attack must be designed for a broad audience and therefore be very descriptive. Since this vulnerability within the system is a race condition, a visualisation is required. The designed visualisation shows which components are present and that they are interconnected. The main components are the CPU, the debugger (partially housed in the CPU), the RDP and the flash memory. Unfortunately, as described in Chapter 3, the components' connections are not entirely clear and thus are not described in detail. The result is the simplified representation in Figure 5.7, which reflects the most likely assumption as shown in Figure 3.2 in Chapter 3. Figure 5.9 illustrates how the race condition is visualised. It is essential to show that the RDP is a little too slow or that the flash memory lock message arrives too late, as shown in Figure 5.8.

This visualisation lends itself well to a presentation, and if the slides are shown one after the other, the flow of information is more vivid. This visualisation is easier to understand for an audience without technical expertise than the sequence diagram from Chapter 3 (Figure 3.2).

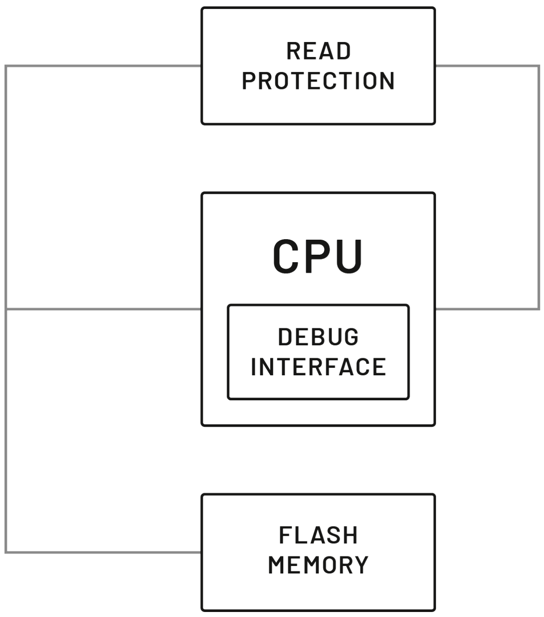


Figure 5.7: Sketch of the Attack Visualisation

Messages sent over these connections are shown as thicker lines from the sender to the receiver, as shown in Figure 5.7.

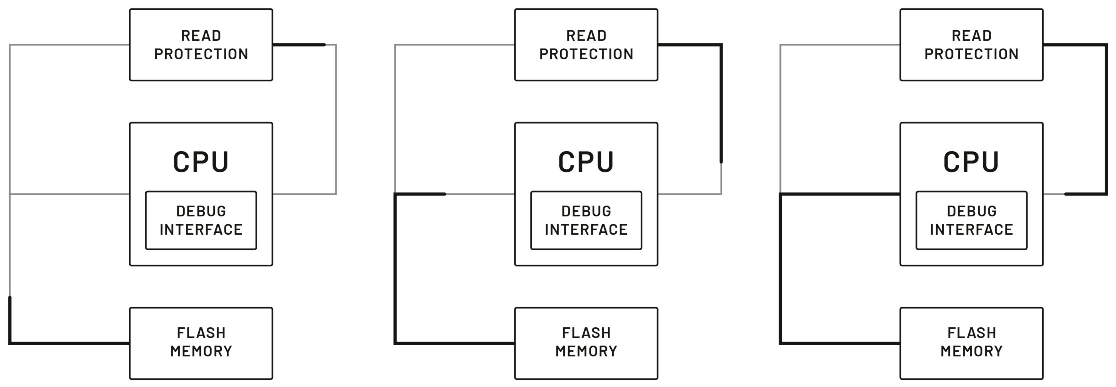
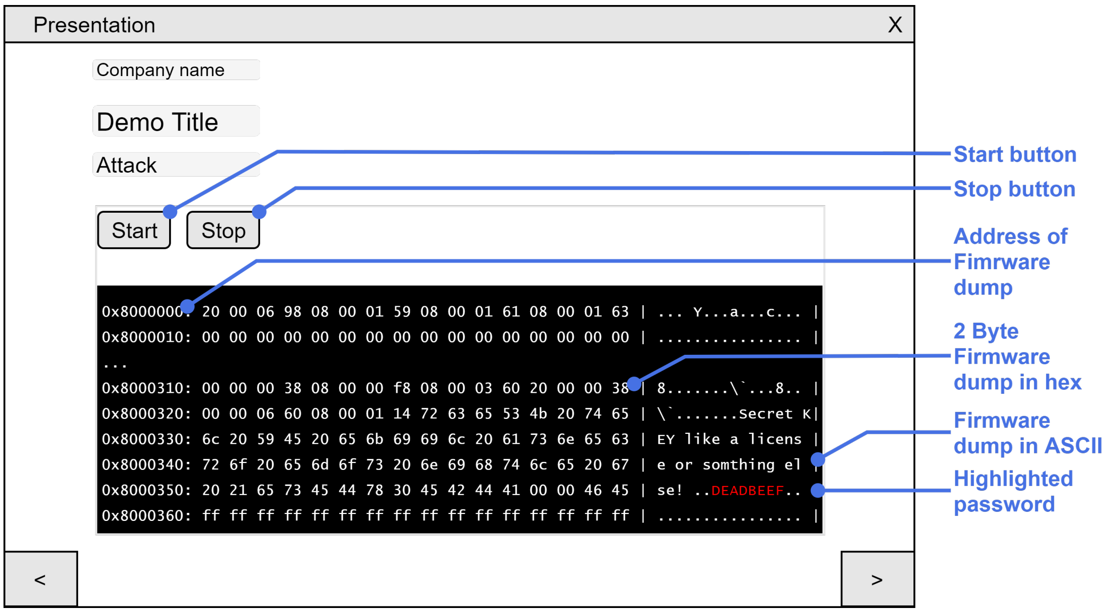


Figure 5.8: Message Flow of the Attack Visualisation

The primary purpose of this visualisation is to show that the message from the RDP arrives too late. This message tells the CPU that the debugger is not allowed to read from the flash. However, the flash has answered the read request in the meantime.

5.4.3 Live Demonstration of the Attack

The attack slide, part 5 of the presentation, looks like the sketch shown in Figure 5.9 and can be started with the "Start" button and, if necessary, stopped with the "Stop" button. During the attack, the firmware dump is output line by line. A line consists of the address where the read operation was started, the firmware dump in hexadecimal representation and the firmware dump interpreted in ASCII characters. After reading out the firmware, the password that was the target of the reading is highlighted. The attack occurs in the background and is executed by the presentation.



```
0x8000000: 20 00 06 98 08 00 01 59 08 00 01 61 08 00 01 63 | ... Y...a...c... |
0x8000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
...
0x8000310: 00 00 00 38 08 00 00 f8 08 00 03 60 20 00 00 38 | 8.......\`...8.. |
0x8000320: 00 00 06 60 08 00 01 14 72 63 65 53 4b 20 74 65 | \`.......Secret K|
0x8000330: 6c 20 59 45 20 65 6b 69 69 6c 20 61 73 6e 65 63 | EY like a licens |
0x8000340: 72 6f 20 65 6d 6f 73 20 6e 69 68 74 6c 65 20 67 | e or somthing el |
0x8000350: 20 21 65 73 45 44 78 30 45 42 44 41 00 00 46 45 | se! ..DEADBEEF.. |
0x8000360: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff | ................ |
```

Figure 5.9: Sketch of the Presentation GUI - Attack Slide

5.5 Hardware for the Attack

As described in Chapter 3, a debugger is also needed for the attack. Since the SEGGER J-Link is commonly used at NXP and software from NXP is already available for this device, it was chosen for the demonstrator. The J-Link supports SWD at the required low level, i.e. reading and writing registers needed for the attack. Tests in advance have also shown that the J-Link is suitable. The SEGGER J-Link can be seen in Figure 5.10.

Figure 5.10: SEGGER J-Link Debugger from [15]

6 Implementation

As with the concept, a distinction is made between the two following parts.

- The password-protected hard disk, which includes the unlocking GUI and the firmware.
- The presentation, with the explanation and live demo of the attack and the hard drive demonstration.

In the beginning, the implementation of the unlocking GUI and the hard disk firmware is introduced. Then the implementation of the presentation is presented.

6.1 The Password-Protected Hard Disk

This section is divided into the software for unlocking the hard disk, which runs on the host system, and the hard disk firmware, which runs on the microcontroller and hardware for the demonstrator.

6.1.1 Unlocking GUI

The unlocking software is implemented with Python and with the help of the GUI toolkit Tkinter. Tkinter was the first GUI toolkit for Python, which is why it is now part of the standard Python package. [14]

In this way, it was possible to create a simple GUI as specified by the concept. The GUI works with two threads. The main thread creates the visible window that the user operates and updates the visual components. The second thread runs in the background, checks the connection status of the hard disk, passes changes and other data via message queues to the main thread, which then updates the window. The second thread also manages the communication with the hard disk, sends entered passwords to the microcontroller and passes the received response to the main thread. In this way, the GUI remains responsive as soon as communication with the hard disk occurs. The second thread, the communication thread, is described first, followed by the main thread.

The Communication Thread

This section describes the thread that works in the background, the communication thread. This worker thread automatically searches for the hard drive after the software is started to increase usability. It searches for the unique USB identifiers, such as the Vendor ID (VID) and the Product ID (PID), and thus identifies the hard disk. The software also handles the connection initialisation, so the user only has to enter the password. Communication occurs via three message queues. One queue syncs the connection status from the communication thread to the main thread, another one receives the password from the main thread, and the last queue forwards the response from the hard disk to the main thread. These three threads are described in Table 6.1.

Table 6.1: Message Queues for Communication between Threads

Name of the Queue	**Task of the Queue**
connectionQueue	The current connection status is transmitted from the communication thread to the main thread.
passwordQueue	The main thread sends the password to the communication thread.
answerQueue	The response from the microcontroller is sent from the communications thread to the main thread.

A flowchart of the communication thread can be seen in Figure 6.1. This flowchart shows how an attempt is first made to connect to the hard disk. Whether a connection could be established is saved and passed on to the main thread via the connection queue. The following steps are repeated continuously until the thread is terminated. This termination occurs when the microcontroller has received the correct password and the main thread has received this message via the response queue. After this, the loop and thus the communication thread is terminated. Within this loop, a check is made to see if the connection status has changed since the last run. If so, it is forwarded to the main thread. Otherwise, if a connection is established, it is checked whether a password has been received from the main thread via the password queue. If a password is received, this password is sent to the microcontroller. The microcontroller's response is forwarded to the main thread via the answer queue.

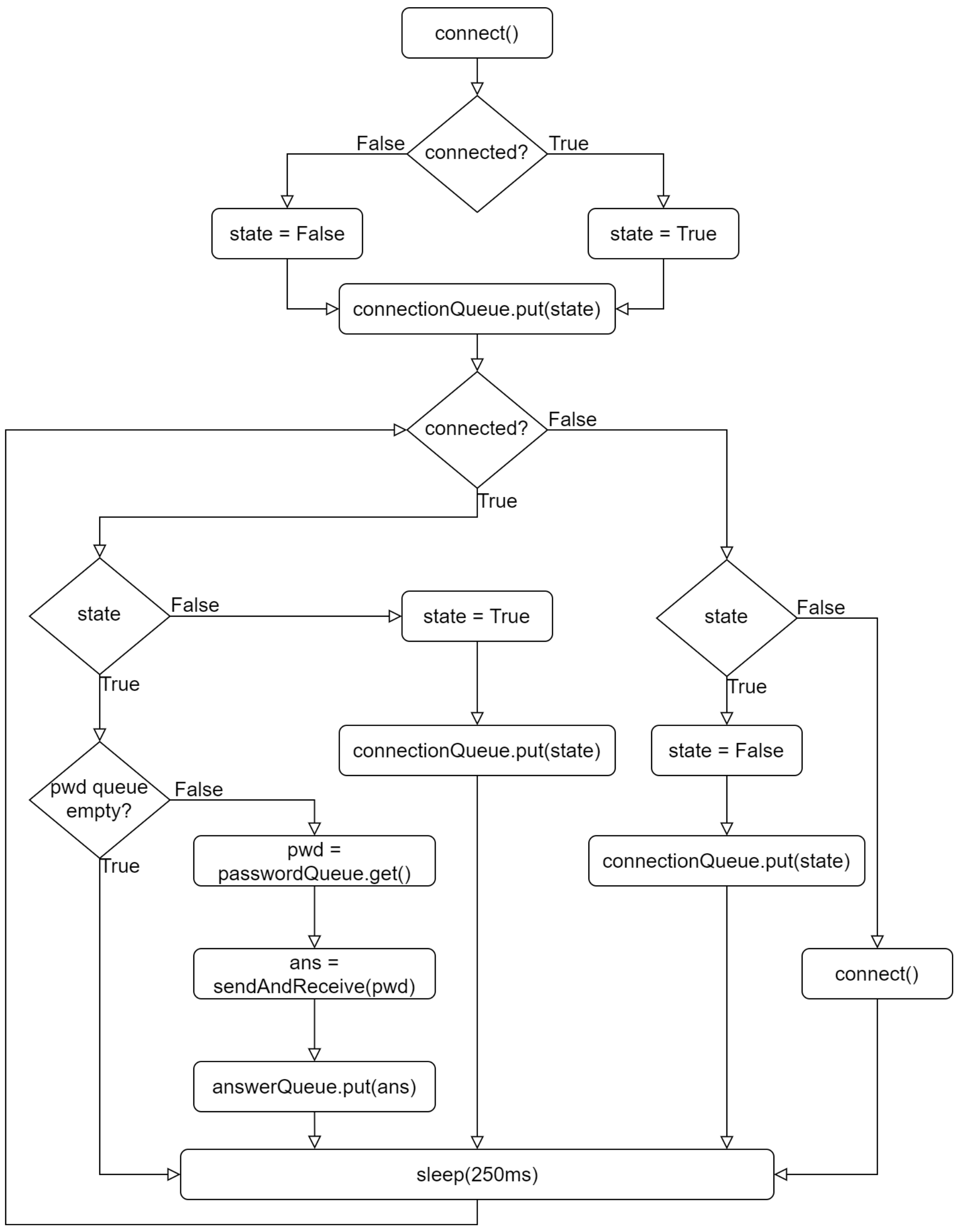


Figure 6.1: Flowchart of the Communication Thread

Figure 6.2 shows a sequence diagram of the main thread and the communication thread, illustrating the communication between the two threads through the queues.

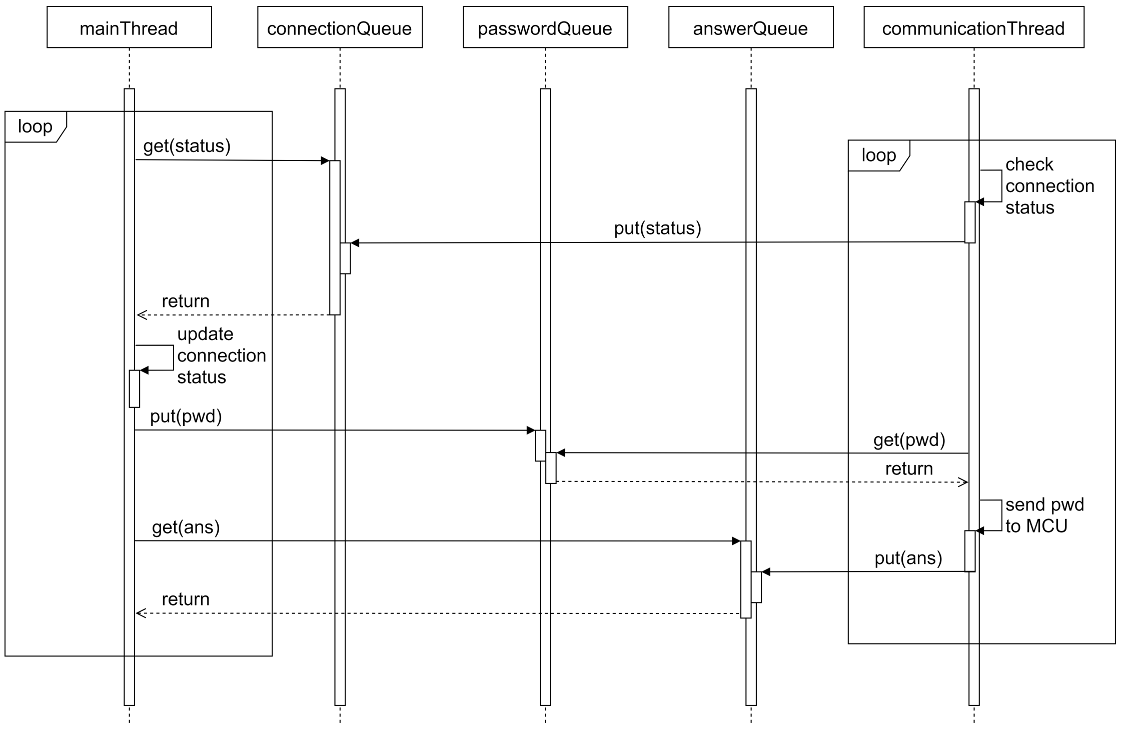
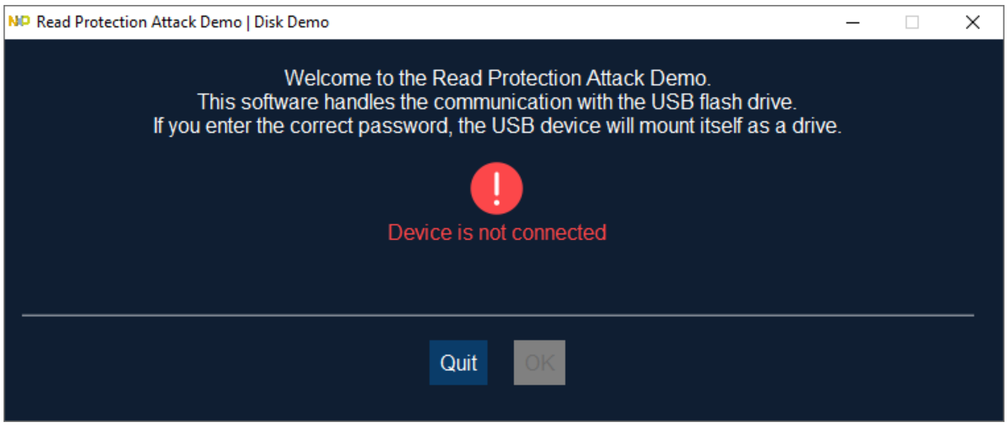


Figure 6.2: Sequence Diagram of the Unlocking GUI

The Main Thread

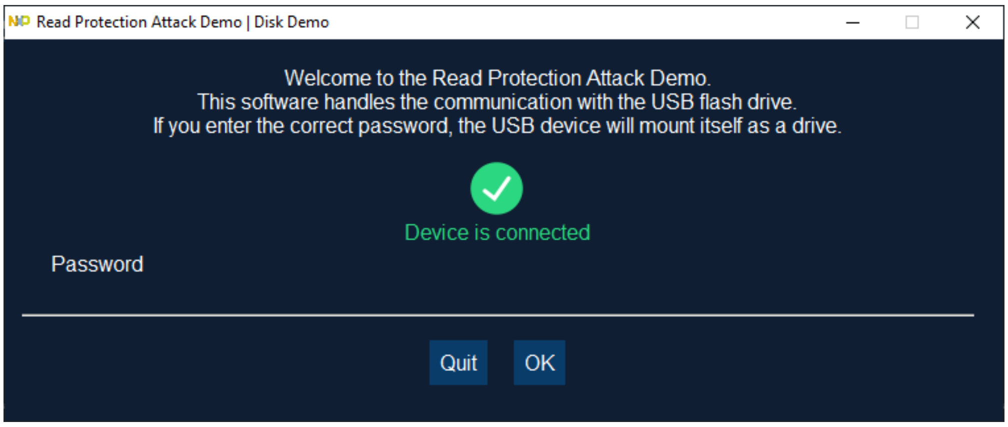
As described at the beginning of the section, the main thread is responsible for the visible window and manages the displayed components. The concept chapter describes these components as the welcome text and the description, the connection status, and the password input field. Furthermore, there are two buttons in the GUI. The "Quit" button can be used to end the program, and the "OK" button can be used to send the entered password. This button can only be pressed when the hard disk is connected and ready to receive. The window is shown in Figure 6.3 with an unconnected hard disk and Figure 6.4 with a connected hard disk.


Read Protection Attack Demo | Disk Demo

Welcome to the Read Protection Attack Demo.
This software handles the communication with the USB flash drive.
If you enter the correct password, the USB device will mount itself as a drive.

Device is not connected

Figure 6.3: Unlocking GUI with an unconnected Hard Disk


Read Protection Attack Demo | Disk Demo

Welcome to the Read Protection Attack Demo.
This software handles the communication with the USB flash drive.
If you enter the correct password, the USB device will mount itself as a drive.

Device is connected

Password

Figure 6.4: Unlocking GUI with a connected Hard Disk

As required by the concept, a password field has been implemented. The entered characters cannot be read because they are displayed as an asterisk '*'. In order to increase the user-friendliness requested in the requirements, the window automatically centers itself after start-up and automatically lifts itself into the foreground when changes are made. Thus, all requirements for the software are fulfilled.

6.1.2 Firmware of the Hard Disk

The hard disk's firmware is written in the C programming language. The basis for the firmware is the Software Development Kit (SDK) from the microcontroller's manufacturer [17]. This SDK served as the basis and had to be adapted in some places. The SDK is chosen because it already offers executable projects that can easily be adapted and provides many required functionalities. The communication functionalities via USB and the hard disk are initially developed separately and later merged.

1. The USB Communication Device Class (CDC) sample application from the SDK is adapted in the next subsection Communications Part of the Firmware.
2. The USB Mass Storage Class (MSC) sample application is adapted in the following subsection Mass Storage Part of the Firmware.
3. These applications are merged with the required functionality, in subsection The merged Firmware.

Communications Part of the Firmware

The USB-CDC sample application uses the USB device and UART peripherals. With this application the MCU acts as a USB-to-UART bridge. This bridge is implemented as a Virtual COM port, which means that on one side, the microcontroller exchanges data with the host system via USB interface in device mode and on the other side, the microcontroller exchanges data with other devices via UART interface. In this USB example, three USB endpoints (separate communication channels) are declared in the CDC class.

These three endpoints are:

1. The Bulk IN endpoint.
   This endpoint is for messages received via UART, which must be forwarded to the host system from the microcontroller.
2. The Bulk OUT endpoint.
   This endpoint is for messages received via USB from the host system, which must be forwarded via UART.
3. The interrupt IN endpoint for setting and getting serial-port parameters.

The interrupt IN endpoint and is taken over unchanged, as it has all the necessary functions. Only the callback functions of the Bulk IN and OUT endpoints for receiving and transmitting are adapted.

Since the UART functionality is not needed, the functions are adapted so that messages received via USB are not forwarded via the UART Interface but instead copied into a buffer. Furthermore, messages can be sent to the host via USB without receiving via the UART Interface. The status of the input buffer is periodically queried. If messages are present, they are processed, and the request is answered.

The following had to be changed in the USB-CDC example for this functionality, as listed in Table 6.2.

Table 6.2: Modified Functions in the USB-CDC Example Application

File	Function	Explanation
usbd_cdc_interface.c	`CDC_Itf_Receive()`	Data received over USB OUT end-point are sent over CDC inter-face through this function and copied into the buffer instead of forwarding via the UART Interface
usbd_cdc_interface.c	`CDC_Transmit_FS()`	Data to send over USB IN end-point are sent over CDC inter-face from this function.

Mass Storage Part of the Firmware

The USB-MSC sample application is originally intended for more complex development board with the same processor. This board has an internal SD card slot where the sample application uses the microcontroller as a bridge and acts as a USB SD card reader. In this way, the host system can address and access the SD card via the microcontroller.

The functionality of the SD card is not adopted to keep the demonstrator as simple as required. This way, no parts of the demonstrator, such as an SD card or the external SD card holder, can be lost. Instead of the SD card, a file system for the hard disk is implemented in the flash memory. The host system accesses the file system stored in the flash memory block by block via USB. The file system in the flash memory replaces the SD card and behaves like a real hard disk. This functionality is taken from the open-source project TinyUSB. TinyUSB is a cross-platform open-source USB host/device stack for embedded systems, and more information can be found on the project website [9]. In this project, an array can be found that represents a complete FAT12 file system. This file system contains a sample text file whose contents can be customized. Thus the array is adopted for the hard disk firmware. Changes made by the host via USB to the text files are initially buffered on the host side and are thus volatile since the changed file system is not written to the flash memory after use. Therefore, the system is in the same state with each new demonstration. A part of the FAT12 array is shown in Figure 6.5.

```
uint8_t msc_disk[STORAGE_BLK_NBR * STORAGE_BLK_SIZ] =
{ //------------- Block0: Boot Sector -------------//
  // byte_per_sector                   = STORAGE_BLK_SIZ;
  // fat12_sector_num_16               = STORAGE_BLK_NBR;
  // sector_per_cluster                = 1;
  // reserved_sectors                  = 1;
  // fat_num                           = 1;
  // fat12_root_entry_num              = 16;
  // sector_per_fat                    = 1;
  // sector_per_track                  = 1;
  // head_num                          = 1;
  // hidden_sectors                    = 0;
  // drive_number                      = 0x80;
  // media_type                        = 0xf8;
  // extended_boot_signature           = 0x29;
  // filesystem_type                   = "FAT12   ";
  // volume_serial_number              = 0x1234;
  // volume_label                      = "NXP-USB MSC";
  // FAT magic code at offset 510-511
  {
        0xEB, 0x3C, 0x90, 0x4D, 0x53, 0x44, 0x4F, 0x53,
        0x35, 0x2E, 0x30, 0x00, 0x02, 0x01, 0x01, 0x00,
        0x01, 0x10, 0x00, 0x10, 0x00, 0xF8, 0x01, 0x00,
        0x01, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x29, 0x34,
        0x12, 0x00, 0x00, 'N' , 'X' , 'P' , '-' , 'U' ,
        'S' , 'B' , '␣' , 'M' , 'S' , 'C' , 0x46, 0x41,
        0x54, 0x31, 0x32, 0x20, 0x20, 0x20, 0x00, 0x00,
        // Zero up to 2 last bytes of FAT magic code
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        ...
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xAA
  },
  //------------- Block1: FAT12 Table -------------//
  { 0xF8, 0xFF, 0xFF, 0xFF, 0x0F // first 2 entries must
   // be F8FF, third entry is cluster end of text file
  },
  //------------- Block2: Root Directory ----------//
  {     // first entry is volume label
        'N' , 'X' , 'P' , '-' , 'U' , 'S' , 'B' , '␣' ,
        'M' , 'S' , 'C' , 0x08, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4F, 0x6D,
        0x65, 0x43, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        // second entry is text file
        'S' , 'E' , 'C' , 'R' , 'E' , 'T' , '␣' , '␣' ,
        'T' , 'X' , 'T' , 0x20, 0x00, 0xC6, 0x52, 0x6D,
        0x65, 0x43, 0x65, 0x43, 0x00, 0x00, 0x88, 0x6D,
        0x65, 0x43, 0x02, 0x00,
        sizeof(FILE_CONTENT)-1, // file size
        0x00, 0x00, 0x00
  },
  //------------- Block3: Text File Content -------//
  FILE_CONTENT
};
```

Figure 6.5: FAT12 File System Array

Building on this, encryption is applied to the entire file system residing in the flash memory. The file system stored as an array is encrypted with Tiny AES. Tiny AES is a lightweight implementation of the AES crypto algorithm in C, which supports the standard key sizes (128/192/256-Bit) and the common block cipher operation modes, like the Electronic Codebook Modus (ECB). The implementation of Tiny AES is based on the recommendations of the National Institute of Standards and Technology (NIST). [12] The file system is encrypted with an AES in ECB mode with a key length of 256 bits. Information on this can be found in NIST's Specification for the AES [7] and in NIST's Paper "Recommendation for Block Cipher Modes of Operation" [6]. A part of the encrypted FAT12 array is shown in Figure 6.6.

The following had to be changed in the USB-MSC example for this functionality, as listed in Table 6.3.

Table 6.3: Modified Functions in the USB-MSC Example Application

File	**Function**	**Explanation**
stm32072b_eval_sd.c (This file is changed in usbd_storage.c)	`STORAGE_GetCapacity()`	Originally returns the capacity of the SD card. Instead, the hard coded size of the array is returned.
stm32072b_eval_sd.c	`STORAGE_IsReady()`	Since no SD card connection needs to be initiated or checked, "ready" is always returned.
stm32072b_eval_sd.c	`STORAGE_Read()`	Instead of reading from the SD card and writing to a buffer, the requested block is read from the flash, decrypted and written to the buffer.

```
        uint8_t msc_disk_enc[STORAGE_BLK_NBR*STORAGE_BLK_SIZ] =
        {
                0xba, 0x59, 0x4a, 0x8a, 0x9d, 0x90, 0xd9, 0x19,
                0x83, 0xaa, 0x32, 0x63, 0x90, 0xfb, 0x27, 0xcc,
                0x45, 0x61, 0x86, 0xe1, 0x90, 0xad, 0xc3, 0x49,
                0xb3, 0x87, 0xdf, 0xcd, 0x8d, 0xd2, 0x17, 0xe9,
                0xb7, 0xef, 0x9e, 0xf5, 0x34, 0x19, 0xdb, 0x13,
                0xd7, 0x04, 0x81, 0x7a, 0x71, 0x02, 0x9c, 0xd6,
                0x0c, 0xd1, 0x1a, 0x40, 0xbf, 0x86, 0xac, 0x4a,
                0x76, 0x58, 0x12, 0xc1, 0x0f, 0xda, 0xe6, 0xb8,
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                ...
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                0xc3, 0x62, 0x63, 0x4c, 0xe6, 0x30, 0xc1, 0x23,
                0x70, 0xde, 0x6d, 0xbc, 0x27, 0x93, 0x5b, 0xb9,
                0xf3, 0xdb, 0x3f, 0xe1, 0xc5, 0x4b, 0xb8, 0x34,
                0xc1, 0x56, 0x06, 0x22, 0x4c, 0xe6, 0xbe, 0x9e,
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                ...
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                0xda, 0xa3, 0xaf, 0xb5, 0xf1, 0x4a, 0xfb, 0xb5,
                0xe3, 0x21, 0xfc, 0x87, 0x68, 0xc2, 0x66, 0x92,
                0x76, 0x7b, 0xfb, 0x91, 0x38, 0xf1, 0xb7, 0xed,
                0x11, 0xd4, 0x6b, 0x8b, 0xcb, 0x70, 0x68, 0xe0,
                0x44, 0x52, 0xc7, 0x0f, 0x17, 0xc1, 0x38, 0x3c,
                0x8c, 0xed, 0x76, 0x20, 0x0f, 0x62, 0x43, 0xd0,
                0xe7, 0x0f, 0x97, 0xa5, 0xe9, 0xaa, 0x45, 0x72,
                0xc2, 0xd2, 0x52, 0x3a, 0x7a, 0xc9, 0x31, 0x1f,
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                ...
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                0x53, 0x68, 0x15, 0xa8, 0xb6, 0xb6, 0x48, 0x53,
                0xe5, 0x3a, 0x51, 0x1a, 0x21, 0x85, 0x58, 0xb7,
                0x11, 0xfb, 0x83, 0x44, 0x73, 0xf7, 0xcf, 0xc7,
                0x7d, 0x8f, 0xe1, 0x52, 0xb0, 0x15, 0x18, 0x64,
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                ...
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77,
                0xf3, 0x95, 0x06, 0x91, 0x37, 0x2f, 0xe3, 0xca,
                0xe4, 0xf7, 0xaf, 0x76, 0xdd, 0xd4, 0x93, 0x77
        };
```

Figure 6.6: Encrypted FAT12 File System Array

In summary, the following is implemented. An encrypted array is stored in the flash memory. After decryption, this array becomes a FAT12 file system, which fulfils the required Full Disk Encryption (FDE). The block requested by the host system via USB is read from the flash memory, decrypted and then passed on to the host system. Since changes are not saved, it is strictly speaking a read-only disk.

The merged Firmware

This section describes how to merge the customised USB sample applications into the password-protected hard disk firmware.

Since the previous applications can both communicate, i.e. receive a password via USB and act like an encrypted hard disk, the following points need to be implemented for the merge:

1. Change the USB class from CDC to MSC class after correctly entering the password.
2. Return to the starting point after disconnecting the USB connection.
3. Make sure that the password used to unlock the device can be read as a string from the flash memory, i.e., it is stored there.

A simplified flowchart of the merged program is shown in Figure 6.7 and the steps required are described below.

For point 1, as soon as the correct password is entered, the USB interrupts are deactivated and then the USB interface is stopped and deinitialised. Afterwards, the USB interface is initialised again, but as USB MSC device. To achieve this functionality, the documentation "STM32Cube USB device library" [18] and the "Description of STM32F0xx HAL drivers" [19, p. 21-31, 33ff., 48ff.] had to be consulted.

For point 2, a reset function (`NVIC_SystemReset()`) is called as soon as the connection is disconnected within the USB MSC state. This resets the microcontroller and the program is restarted, returning the microcontroller to the USB CDC state. This function requests a system reset by setting the system reset request flag (SYSRESETREQ) in the Application Interrupt and Reset Control Register (AIRCR).

Finally, for item 3, the password is stored in plaintext as a character array in the main function and used for comparison with the received password during the program run. This way it is stored in the flash memory and can be read out.

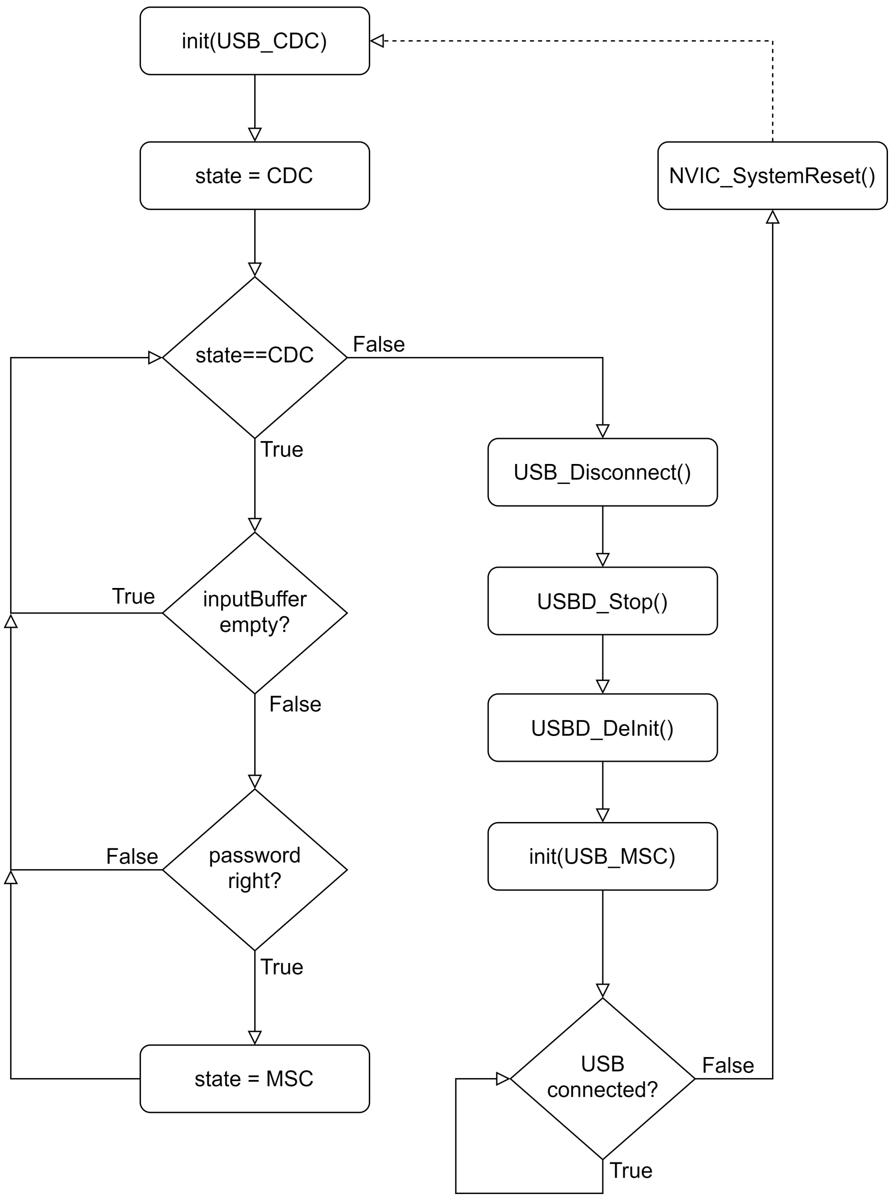


Figure 6.7: Flowchart of the password-protected Hard Disk's Firmware

6.1.3 Hardware of the Disk

Only reverse polarity-protected cables have to be connected for the assembly, as specified in the requirements. This connection is shown in Figure 6.9 and ensures that the cable is not misused. For this purpose, a connector is soldered to the development board, and the board is provided with housing. The housing for the development board of the hard disk is made with a 3D printer. The cover was made of transparent Plexiglas to allow a view inside, seen in Figure 6.8 and 6.10.

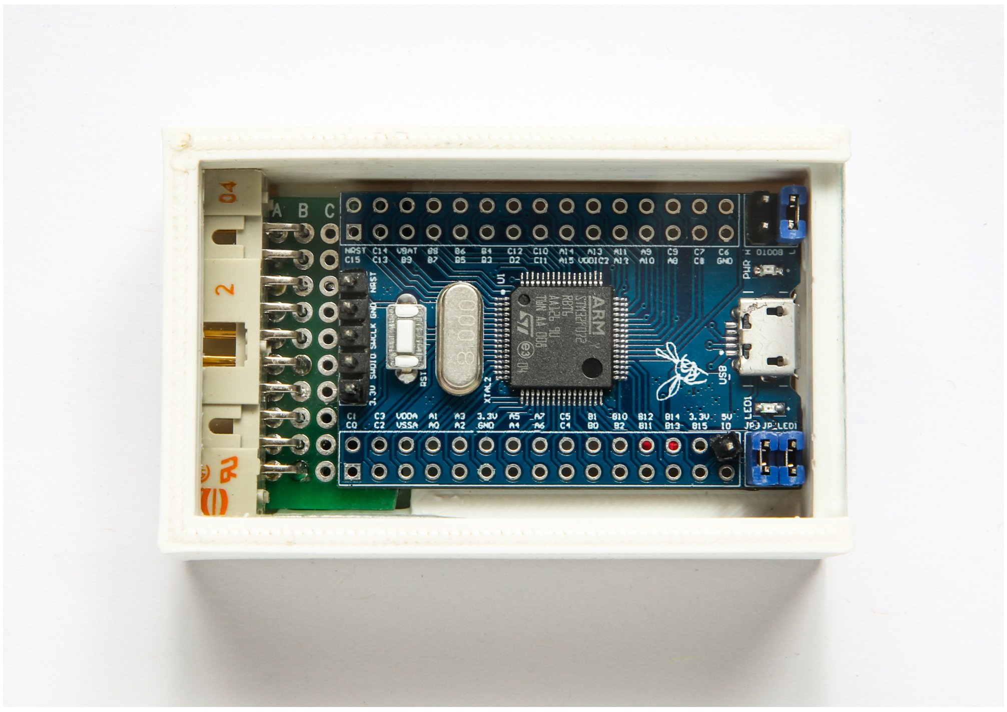


Figure 6.8: Device under Attack in Housing - Topview

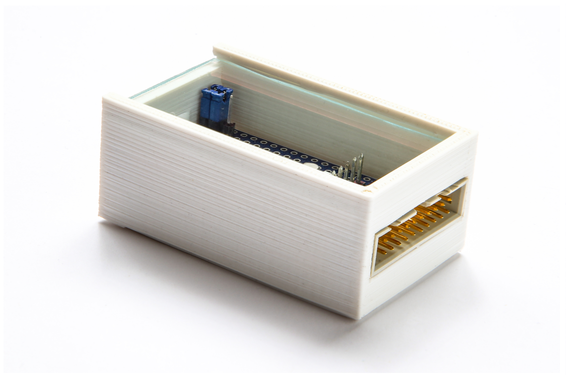


Figure 6.9: Device under Attack in Housing - Connector Front

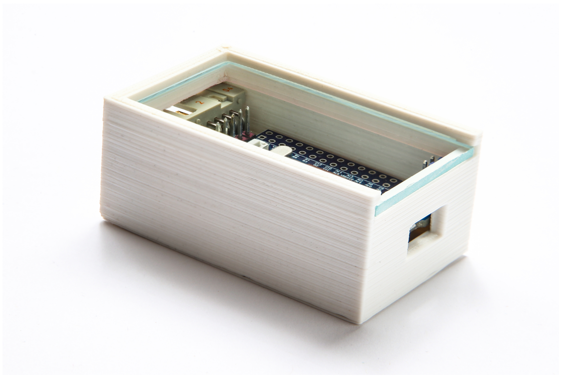


Figure 6.10: Device under Attack in Housing - USB Front

6.2 Presentation of the Demonstrator

This section describes the implementation of the presentation, which introduces the hard drive, explains the attack and demonstrates it live. As with the unlocking GUI, Python is used together with Tkinter for the implementation. A presentation developed and used for other demonstrators is used as the basis for the implementation. In this section, the structure of the GUI, the different classes and the implementation of the attack are explained.

6.2.1 Structure of the GUI

The presentation consists of the following classes, which can also be seen in the class diagram in Figure 6.11.

- MainView
- SlideShow
- Page-Classes (Page1 to Page5)
- DebugJLink

The classes will be explained in this order.

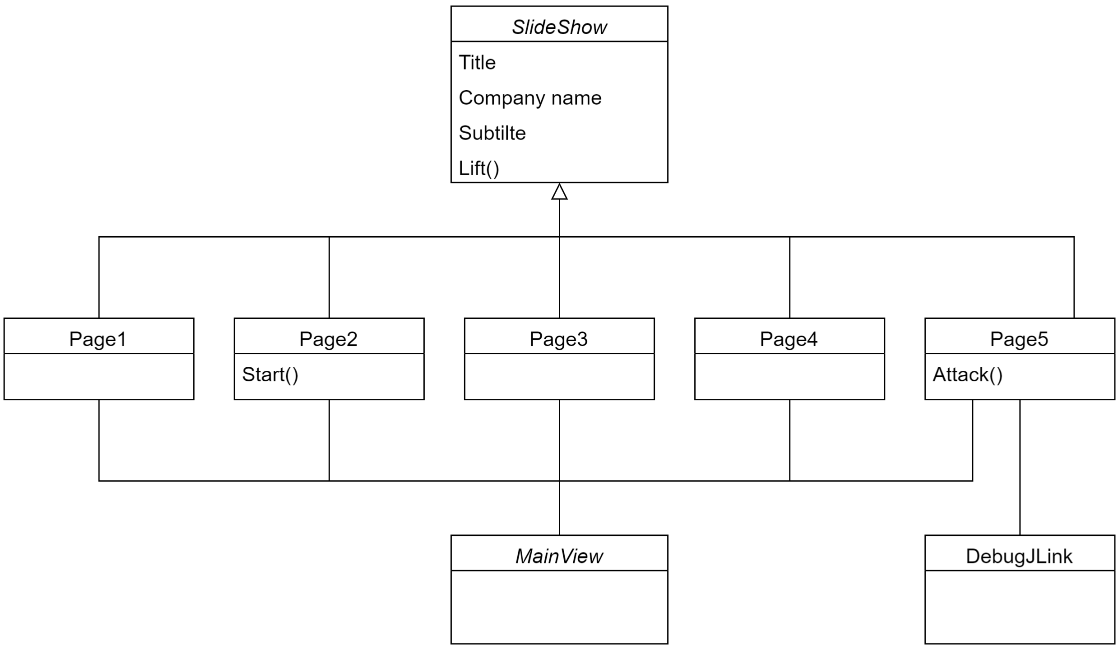


Figure 6.11: Class Diagram of the Presentation

MainView

The MainView class is responsible for creating the visible window and managing it, i.e., creating and displaying various pages. The buttons for scrolling forwards and backwards are also managed in the Main-View class. These buttons call the `lift()` function of Tkinter for the corresponding page, which lifts this page into the foreground and thus makes it visible. All other pages lie behind it and can be displayed in order. After the last page, the first page is displayed and the other way around.

A loading bar is displayed at the beginning to inform the user that the GUI has been started. This feedback increases user-friendliness. This loading bar is necessary because the first reading, copying and pre-scaling of the graphics take about 5 seconds. This time is necessary because the graphics are high resolution and can be used on large and high-resolution display devices as specified in the requirements.

After the graphics have been processed, the pages are created. These inherit from the class SlideShow, described in the next section. As soon as all pages have been created, the loading bar is 100%. After that, the first page, Page1, is displayed, and the loading bar is closed. The window is brought to the foreground of the main screen and displayed in full screen. In summary, the following happens from the user's point of view. After starting the presentation, the user sees only a loading bar displayed in the center of the main screen in the foreground. This loading bar reaches 100% within a few seconds, and the next moment, the first page of the presentation is full screen on the main screen. The presentation can be operated using the buttons at the bottom or with the arrow keys on the keyboard. As required, starting the software is very simple and intuitive.

SlideShow

A base class is implemented from which all parts are inherit. This base class represents the master slide shown in the concept (Figure 5.6). The base class contains the title of the demonstrator, the name of the company and an area for slides. It is also possible to display graphics that can be automatically scaled to the size of the window. The scaling functionality has been adopted mainly from the existing presentation. The scaling functionality saves the high-resolution original graphic and creates a copy used for scaling. The original is saved to ensure the best possible quality. If the window is first reduced and then enlarged again, this could lead to a loss of quality. Because when the window is reduced, information is removed that is then missing when the window is enlarged. In this case, the original is used, and the copy is replaced by the original. The renewed copy can then be resized without loss of quality. The possibility of a slide show is also implemented in the base class. This slide show makes it possible to display several graphics one after the other, as is necessary, for example, when explaining the attack and is described in the next section, among the other Pages.

Page-Classes

This section describes the page classes. There are five Page classes, each representing a part of the presentation and inherits from the basis-class SlideShow. Table 6.4 lists all five classes.

Table 6.4: Parts of Presentation with the corresponding Page-Class

Name of the Page-Class	Part of the Presentation
Page1	Presentation of the hard disk
Page2	Demonstration of the hard disk
Page3	Setup of the attack
Page4	Explanation of the attack
Page5	Demonstration of the attack

Page1

Page1 shows the password-protected hard disk and that the password must be sent to the hard disk. By inheritance, only a subtitle is assigned to the class, i.e. the title of the slide "Explanation Disk" and the associated graphic for the slide show.

Page2

As with any page class, the subtitle must still be added to page2. This subtitle is called "Demo Disk", and this page only has a button for starting the unlocking GUI, with which the hard disk is presented and demonstrated live.

Page3

Page3 is titled "Attack" and shows the setup of the attack. Figure 6.12 shows this setup visualisation.

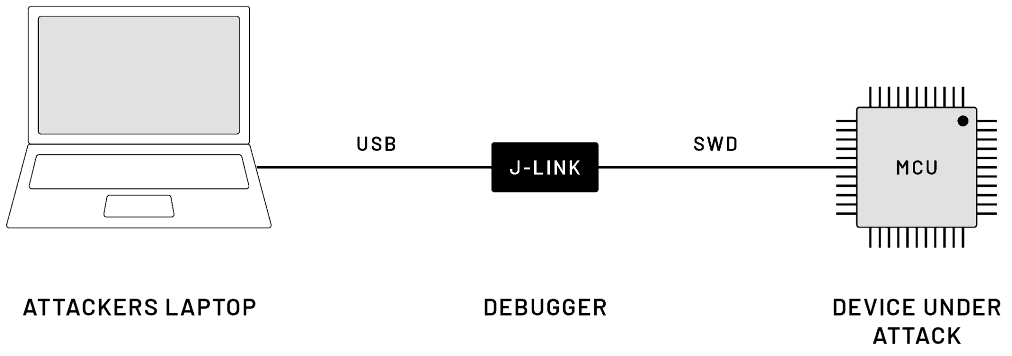
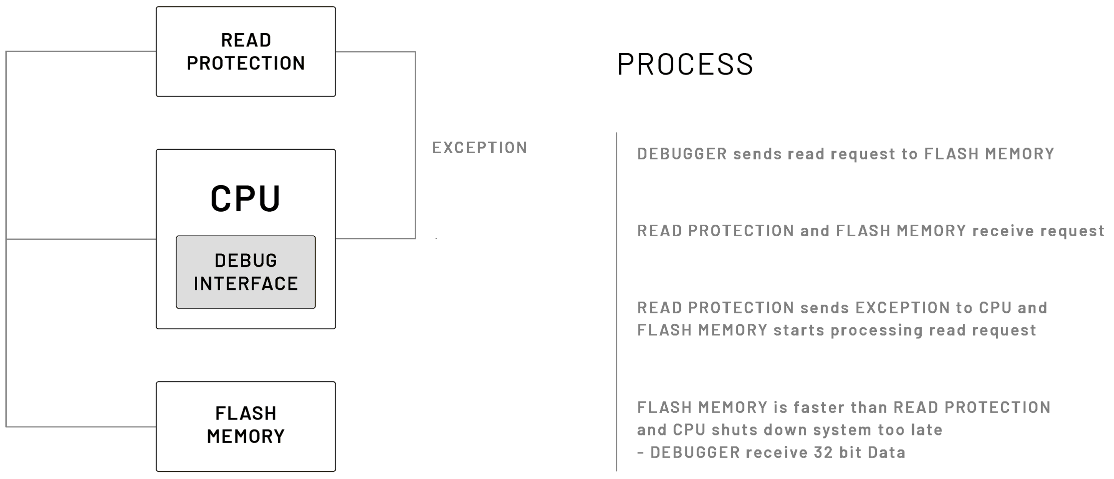


Figure 6.12: Visualisation of the Attack's Setup

Page4

Page4 describes the attack and is also named "Attack". For this, the graphics shown in the concept are displayed and shown with stick point explanations. In Figure 6.13, the first graphic is shown, and the corresponding explanation is highlighted. This graphic gives an overview of all components.



PROCESS

DEBUGGER sends read request to FLASH MEMORY

READ PROTECTION and FLASH MEMORY receive request

READ PROTECTION sends EXCEPTION to CPU and FLASH MEMORY starts processing read request

FLASH MEMORY is faster than READ PROTECTION and CPU shuts down system too late
– DEBUGGER receive 32 bit Data

Figure 6.13: Slide Explaining the Attack in the Presentation

Page5

Page 5 is the slide of the live attack and is called "Attack". There is only a start button and a black field in the beginning. If the hard disk is still connected to the computer via USB, it is pointed out that the USB cable must be disconnected. This notification is necessary because the microcontroller is supplied with power via USB, and a restart (power cycle) via the debugger is not possible. The attack, described in Chapter 3, begins as soon as the Start button is pressed. After the start, another button is displayed - the Stop button - with which the attack can be stopped. The read firmware is output during the attack as described in the concept, and the password is highlighted in the end. The class of the attack is described next.

DebugJLink

The class DebugJLink controls the J-Link debugger and thus executes the attack. NXP's software library is used to control the J-link. This library, Lab Control, contains functions to communicate with the microcontroller via the minimal implementation of the SWD protocol using the J-Link, which is necessary for the attack.

After the attack is started, it is checked whether the microcontroller is configured correctly. It is checked whether the read protection is set to RDP level 1. If this is not the case, the user can decide whether the protection should be switched on. If this option is chosen, the current firmware is loaded into the flash memory and then RDP level 1 is set. This procedure fulfils the requirement that the software can correct errors itself. The attack then begins with a restart (power cycle) of the microcontroller. Power cycle means that the J-Link briefly switches off the microcontroller's power supply. For reading out the firmware, the steps described in section 4 are performed, and if the access is successful, the address to be read is incremented. The password is searched and marked if the firmware is completely read out. In addition, the focus is set to the position of the password. The procedure is shown in a flowchart of the attack in Figure 6.14.

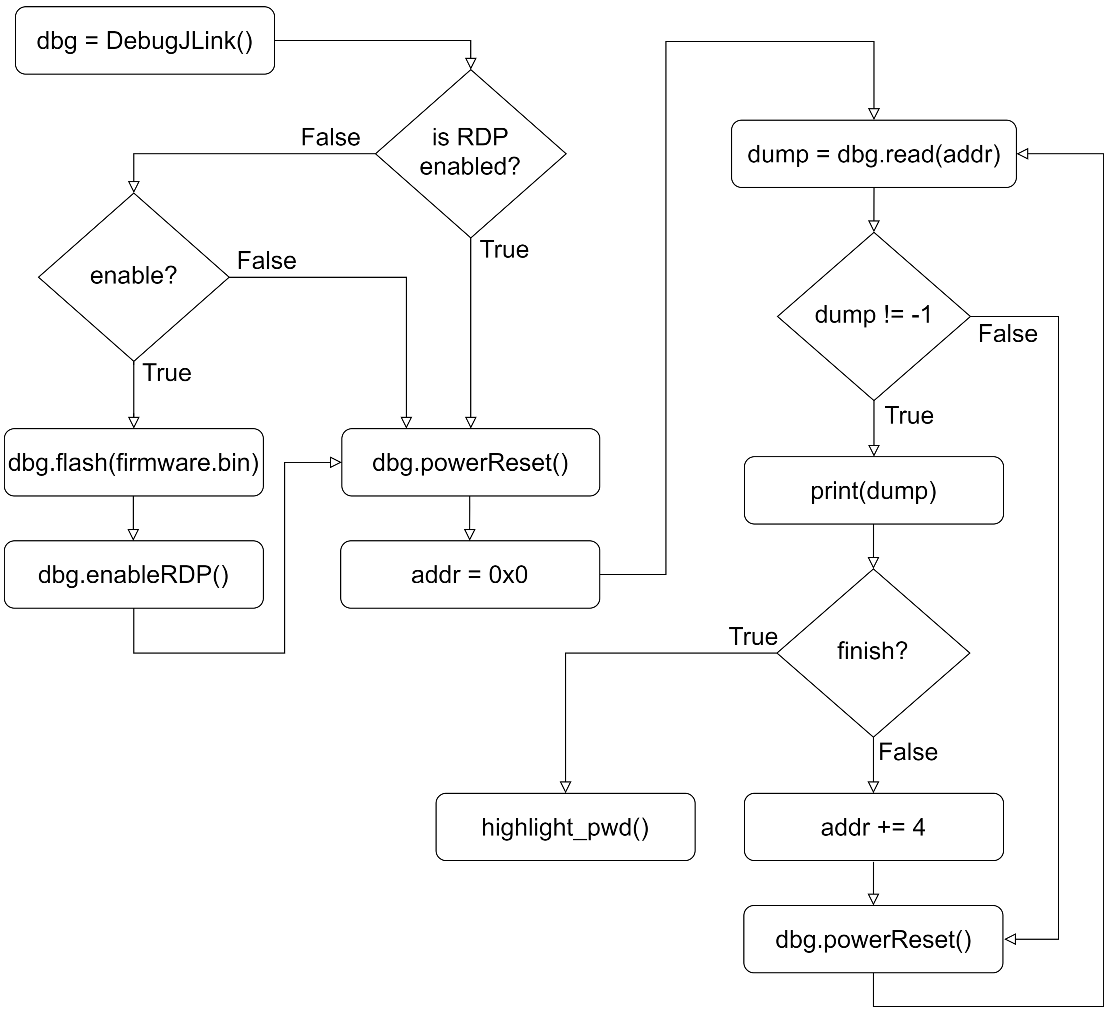


Figure 6.14: Flowchart of the Attack

6.3 Limitations and Opportunities

The demonstrator is in an operational state. Several test runs have shown that the complete firmware with a size of nearly 32 kByte can be read in approximately 10 minutes. That means one of the essential requirements, the live demonstration, has been successfully implemented. Furthermore, typical errors are detected by the software. For example, the presentation points out that the USB cable must be disconnected before the attack begins. Otherwise, the attack is not possible. The presentation can also check the status of the microcontroller and, if necessary, renew the firmware and switch on the RDP.

Nevertheless, there is still potential for improvement. For one thing, the hard drive's locking mechanism could be reworked. A desirable feature would be the ability to change the password. The password is currently hardcoded into the program and cannot be changed. For this, the communication protocol would have to be extended to include the possibility of changing the password, and a function would have to overwrite the old password in the flash memory with the new one. On the other hand, a different operating mode of the AES would be desirable since the ECB mode encrypts identical plaintext blocks into identical ciphertext blocks and thus does not hide data patterns. Because the hard disk consists primarily of "0x00" entries, and these blocks are uncomplicated to recognise. For this reason, the ECB is unsuitable for full disk encryption, even if it is straightforward to implement.

Another essential point is the performance of the presentation. The presentation must be revised, whereby the revision will focus on the performance of the graphics scaling. In this area, optimising with a second thread with message queues could be helpful and reduce the loading time in the beginning.

Further work will replace the J-Link debugger with a smaller, less expensive alternative to better meet the requirement of duplicating quick and inexpensive this demonstrator. During development, it was detected that the J-Link debugger could easily be replaced by a debugger that supports SWD at all levels, especially the low level. Support for all SWD levels means that the debugger supports flashing of the MCU and operation with low-level functions, such as reading and writing registers, as required for the attack. The debugger should be able to supply the MCU with power. This power supply should be switchable since the microcontroller must be restarted via a power cycle. However, this functionality could also be easily implemented with a transistor, which turns the power supply on and off. It would be conceivable to develop a debugger with a development board and implement the SWD protocol. For this purpose, the development board used for the hard disk could be used with the USB communications functionality already developed. Also, the attack's performance could possibly be increased by implementing a self-made debugger. In the course of this, it could also be tested whether reducing the capacity of the capacitor that stabilises the voltage supply influences the restart duration. In other words, whether there is a significant correlation between the total capacity of the voltage supply and the restart time.

The creation of a manual is still obligatory. This manual is included with the demonstrator and describes the setup. This manual is provided on a USB flash drive. The demonstrator will later be packed in a suitcase to make it handier and easier to transport. This case contains the microcontroller, i.e. the hard disk, the debugger and the USB flash drive with the manual, the presentation and the unlocking GUI.

7 Conlusion

In this bachelor thesis, a demonstrator is built that bypasses the readout protection and thus reads out a microcontroller's firmware. For this purpose, a suitable and authentic use case was developed and implemented with corresponding hardware. The use case consists of a portable password-protected USB hard disk, which can only be decrypted and mounted with the correct password. The password is evaluated on the hard disk. For this purpose, the password is sent to the hard disk via USB. Since the evaluation takes place on the hard disk, the password must also be stored there for comparison, making the hard disk the target of the attack. The attack shows how the readout protection is bypassed, and the password stored in the firmware is readout. The impact of password theft is significant as it compromises the entire encryption. A presentation was developed to demonstrate the attack that introduces the typical working hard disk to the user and gives a live demonstration. Afterwards, the attack is explained and also demonstrated live. The entire firmware, including the password, is read out in about 10 minutes. After reading out, the password is highlighted and presented, ending the presentation.

This demonstrator fulfills almost all the requirements that were set for it. Thus, the most important goals were achieved, which say that the hard drive and the attack have to be live demos, and especially the attack is shown in real-time without preparations. It also tries to explain the attack in an understandable way to a broad audience and show the consequences as impressively as possible. The demonstrator shows the importance of security measures and that it is not enough to rely on single security mechanisms, as they may be bypassed or hacked.

One possibility to extend the demonstrator even further would be to implement the much more secure RDP level 2 in this demonstrator so that the microcontroller is first downgraded from level 2 to 1 and then read out. RDP Level 2 is more secure because it permanently shuts down the entire debugging interface. However, possibilities for successfully attacking the RDP level 2 were also presented in the Frauenhofer AISEC paper by Obermaier and Tatschner [13]. Alternatively, also found in this presentation [3]. In this presentation, one of the cryptocurrency hardware wallets will be cracked. The attacks carried out on the hardware wallets range from leveraging the proprietary bootloader protection to leveraging the web interfaces to physical attacks, including glitches, to bypass the security implemented in the microcontroller of the wallet. These attacks include glitching a microcontroller protected with RDP level 2 to 1. For this extension, the microcontroller secured with RDP Level 2 could be downgraded from Level 2 to Level 1 by attacking the power supply, which is shown in the presentation, with a so-called voltage glitch, in which the power supply is briefly switched off, thereby inducing an error in the digital logic of the microcontroller. Then the firmware can be read out with the debugger.

Bibliography

[1] ARM LTD.: *AMBA® AHB Protocol Specification IHI 0033C*. https://developer.arm.com/documentation/ihi0033/latest/, Last accessed: 04. Nov 2021. – Rev. C

[2] ARM LTD.: *Arm Debug Interface Architecture Specification v5.1 IHI 0031F*. https://developer.arm.com/documentation/ihi0031/latest/, Last accessed: 25 Jun 2021. – Rev. F

[3] ARM LTD.: *CoreSight Architecture: Serial Wire Debug*. https://developer.arm.com/architectures/cpu-architecture/debug-visibility-and-trace/coresight-architecture/serial-wire-debug, Last accessed: 25 Jun 2021. 2021

[4] ASCHE, Rüdiger R.: *Embedded Controller - Grundlagen und praktische Umsetzung für industrielle Anwendungen*. Berlin Heidelberg New York : Springer-Verlag, 2016. – ISBN 978-3-658-14850-8

[5] AXELSON, Jan: *USB Complete - The Developer's Guide*. Madison WI 53704 : Lakeview Research, 2009. – ISBN 978-1-931-44808-6

[6] DWORKIN, Morris: Recommendation for Block Cipher Modes of Operation / National Institute of Standards and Technology. Washington, D.C., 2001 (NIST Special Publication 800-38A 2001 Edition). – Forschungsbericht

[7] DWORKIN, Morris ; BARKER, Elaine ; NECHVATAL, James ; FOTI, James ; BASSHAM, Lawrence ; ROBACK, E. ; DRAY, James: *Advanced Encryption Standard (AES)*. https:///doi.org/10.6028/NIST.FIPS.197, Last accessed: 04. Jan 2022. 2001-11-26 2001

[8] FLEMING, Bill: Microcontroller Units in Automobiles [Automotive Electronics]. In: *IEEE Vehicular Technology Magazine* 6 (2011), Nr. 3, S. 4–8

[9] HA THACH: *TinyUSB*. https://docs.tinyusb.org/en/latest/index.html, Last accessed: 04. Jan 2022

[10] INTEGRIERTE SICHERHEIT AISEC, Fraunhofer-Institut für Angewandte und: *Produktschutz-Infografik.pdf*. https://www.aisec.fraunhofer.de/content/dam/aisec/Dokumente/Publikationen/Sonstige/Produktschutz-Infografik.pdf, Last accessed: 10. Jan 2022

[11] KHAN, Sultan Q.: *Microcontroller Readback Protection: Bypasses and Defenses*. https://research.nccgroup.com/wp-content/uploads/2020/02/NCC-Group-Whitepaper-Microcontroller-Readback-Protection-1.pdf, Last accessed: 10. Jan 2022

[12] KOKKE: *tiny-AES-c*. https://github.com/kokke/tiny-AES-c, Last accessed: 04. Jan 2022

[13] OBERMAIER, Johannes ; TATSCHNER, Stefan: Shedding too much Light on a Microcontroller's Firmware Protection. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC : USENIX Association, 2017. – URL https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier. – Last accessed: 04. Oct 2021

[14] PYTHON SOFTWARE FOUNDATION: *tkinter — Python interface to Tcl/Tk*. https://docs.python.org/3/library/tkinter.html, Last accessed: 07. Dec 2021

[15] SEGGER MICROCONTROLLER GMBH: *Product Page J-Link Base*. https://www.segger.com/products/debug-probes/j-link/models/j-link-base/, Last accessed: 07. Dec 2021

[16] STMICROELECTRONICS: *Product overview: 32F072BDISCOVERY*. https://www.st.com/en/evaluation-tools/32f072bdiscovery.html, Last accessed: 07. Dec 2021

[17] STMICROELECTRONICS: *STM32Cube MCU Package for STM32F0 series*. https://www.st.com/en/embedded-software/stm32cubef0.html, Last accessed: 07. Dec 2021

[18] STMICROELECTRONICS: *UM1734 — STM32Cube USB device library*. http://www.st.com/st-web-ui/static/active/en/resource/

technical/document/user_manual/DM00105256.pdf, Last accessed: 01. Dec 2021

[19] STMICROELECTRONICS: *UM1785 — Description of STM32F0xx HAL drivers*. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00122015.pdf, Last accessed: 01. Dec 2021

[20] STMICROELECTRONICS: *RM0091 Reference manual*. https://www.st.com/en/microcontrollers-microprocessors/stm32f0-series.html#documentation, Last accessed: 08. Oct 2021. 2021. – Rev 9

[21] TANENBAUM, Andrew S.: *Structured Computer Organization* -. 5th Edition. London : Prentice Hall, 2006. – ISBN 978-0-131-48521-1

[22] YIU, Joseph: *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Third Edition. London : Newnes, 2013. – ISBN 978-0-124-07918-2

[23] YIU, Joseph: *The Definitive Guide to ARM® Cortex®-M0 and Cortex®-M0+ Processors*. Second Edition. London : Newnes, 2015. – ISBN 978-0-12-803277-0

[24] ZÜRNER, Dominik B.: *A Demonstrator for Optical Fault Injection Attacks*. 2018. – URL https://reposit.haw-hamburg.de/handle/20.500.12738/8371

A Appendix

The appendix is located on a CD-ROM.
This CD-ROM can be inspected at the supervisors:
Prof. Dr. Heike Neumann <heike.neumann@haw-hamburg.de> and
Prof. Dr. Paweł Buczek <pawel.buczek@haw-hamburg.de>.

Declaration

I declare that this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used.

Hamburg	13. January 2022	
City	Date	Signature