Saudi Arabia:

r14 is incremented until it gets to the first byte after the password

r15 = password begin

r11 = password size

Username allocation = 17 bytes

Password is checked for 16 bytes (45e0-45ec)

45e0-4610- ensure password is between 8-16 characters

What if I overwrote the byte count?

464c- Test against 00... strcpy...

Overview:

The lock now accepts a username and a password. There are bounds checks to ensure the password is within bounds (which isn’t really properly implemented- it can only accept 15 bytes instead of 16, but it doesn’t matter). Even if you do somehow manage to get past the bounds checks, there’s a stack canary that’s set to 00 so you can’t just put the proper value in your code- strcpy will see to that.

However the username is not tested at all. We can first break the bounds checks on the password by overwriting the minimum and maximum lengths on the stack. Then we can get all the way to the return address and overwrite it.

But there’s still that damn stack canary, which we overwrote with the username. The solution is thanks to us breaking the bounds checks on the password. We can just write a 00 to the proper position with the password. Then we get to ret and boom there we go.

Username:

code 08 min FF max padding ret

**3740017f178387100712b012c446**FFFFFF**08FF**FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF**a243**

stack canary

Password: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF**00**

**Addis Ababa**

The exploit is definitely in printf

@4476 flag is pushed to the stack before printf, can we target it? If so, well, I suppose we can just overwrite the return address

4408 is return address at 349E

R4 points just before the return address

Begins by looping and checking for %’s (25)

This loop seems to check for the positions of %’s, throughout the entire string

It… Just checks for where %’s stop happening.

It looks like we can’t somehow overflow the string: the return address is before the string. So is the “password set” flag.

With the password entirely being %’s, it looks like the length gets overflowed by one

At 45E0 sp is moved to the position of sp, so, it’s really mov sp, 0x0(sp)

The entire string is copied before the return address. Presumably this is used for the printf args.

@4638 the current byte is checked for a % and if it’s not it’s printed

@4658 it’s checked for a %s

@456e then @466c is \*really\* interesting… It’s reading from memory… an address we control…?

@4648 appears to checking for “%%”

So… It allows for %s and %x.

%s moves the next two bytes that was somehow copied before the return address to r11 where it will later be read from.

@467e-4672 it just wants the upper bits.

Overall %x just… doesn’t show much promise. Although it’s really funky with pointer addressing it just doesn’t do anything other than putchar. Blah. Let’s look at that copy mechanism.

This has got to be it. @46b2, %n, which \*writes the current count of characters to a pointer\*.

What would happen if I wrote to 0x10?

@4624: if r14 < r13

@461a-4626: Copy loop

So we have an arbitrary write to wherever we want… But the value written needs to be between 0x2 ­­and 0x13, as it’ll be the size that’s written.

Maybe we can… replace a mov to a register with pc?

Or… We could not be stupid and overwrite the unlock flag. Nope. We can’t say that we’ve written 0 bytes.

And we can’t even write to pc due to it being address 00.

The perfect candidate is to overwrite the tst instruction of the conditional flag. We need to write a 0x3.

@4600 we check for %’s.

@4606 We step back sp by the number of %’s \* 2 + 2. Is this for a copy?

@461a-4626 is the copy loop. It copies the amount of %’s in the string. Does it have to be all %’s, or can it be other things with the %’s later in the string?

We can copy as many bytes as there are %’s in the string.

For some reason, r9 points directly before our string. No matter, we can simply %n twice.

AddrPadd256e256e

After all this time… It’s complete.

**Writeup- fuck printf**

This will be terse because I just want to be done with this level.

printf has the %n function, which takes as an argument a pointer to write how many bytes have been printed so far. LockItAll’s implementation uses the string itself, copying as many bytes for the argument buffer as there are %’s, which makes sense as it only needs as many bytes as there are arguments. Well, this is a great recipe for an arbitrary write, eh? It’s *extremely limited*- you can write a value between 0x2 and 0x11, as that’s the amount of bytes you can use. Exploiting this is almost as difficult as finding the bug in the first place, which is why the next level, a printf HSM-2, ought to be quite interesting. I decided on overwriting the tst instruction to test the conditional flag on whether the passcode is correct or not. It was overwritten with 0006, which decodes to push pc, which does not modify the state register. Since sr did not have the zero flag set at this point, the jump instruction to skip unlocking does not occur, and the lock is unlocked.

Oh, and the address for arguments is right before the actual arguments for some reason. No matter. Just %n again.

8a44FFFFFFFF256e256e

**Jakarta**

The unlock function is left in even though it’s an HSM-1 level. Huh.

Username + password testing seems to be typical.

It accepts 0xFF bytes for the username and password. However a size check immediately follows.

If the username is greater than or equal to 0x21 bytes it stops. 32 bytes, as promised.

31 bytes is subtracted by the username count and AND’ed by 0x1ff to get how big the password should be. As promised, no more than 32 characters.

0x1FF passes through the two bytes and makes sure the 3rd is not greater than 1. Is this a poor attempt to ensure the size is proper? I suck at bitwise operations

The password is moved to sp + username size

The password + username are then tested to ensure the size doesn’t exceed 31 bytes.

Can we load 0xFF bytes with no null terminator to get 0x100 bytes? This wouldn’t help- it would still set the carry flag. Actually… it’s a cmp.b. Would it compare 00 to 21 or 01 to 21?

Nope- the getsn interrupt appears to set a null terminator at the end “for you”.

**0x45cc is the key problem!**

**We’re given 0x20 bytes as input for the username. Then 0x1f – size = password input size. 0x1f – 0x20 = 0xffff!** (Then this is and’ed by 0x1ff so we actually get 0x1ff bytes.)

We’re not out of the woods yet. There’s another cmp.b 0x21. But this time we get as many as 0x1ff bytes. Can we overflow like I theorized earlier?

We need to make it so username size + password size = 0x100. So we need 0xE0 bytes.

YES IT WORKED!!!

Username:

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Password:

4c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c444c44

**Novosibersk**

Well, this was easy.

printf was implemented in the HSM-2. Arbwrite works without issues, except the weirdness with r9 being right before our string arguments is gone. We’re given 0x1ff bytes to work with. ( ͡° ͜ʖ ͡°)

So we simply overwrite what gets pushed to the stack when INT is called. This could work anywhere with a push that’s not of a register, so I used the conditional\_door\_unlock function’s INT call. Replace 0x7e with 0x7f and ( ͡° ͜ʖ ͡°) happens.

c844FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF256e

**Algiers**

This is a basic *unlink* exploit for Microcorruption’s free(). 0x30 bytes are inputted into a 0x10 allocated space, so we can overwrite the next header. In this exploit, free is *extremely conveniently* placed right before the unlock\_door function. So let’s just overwrite free’s ret with something else ( ͡° ͜ʖ ͡°)

Microcorruption’s mchunk headers look like this:

4500 4500 0020

Previous chunk, next chunk, and… Some kind of magic, I think, I didn’t bother reversing it enough to find out.

Anyway, free manipulates the headers so they point to different things through unlink. If we set previous and next equal to each other, pointing to the same address (455e- 6 bytes before the ret necessary) it will end up doing typical unlink things, so:

Set P->fd->bk = P->bk.

Set P->bk->fd = P->fd.

Then it will for some reason manipulate the magic value.

This is analogous to the following:

i: \*ii = i;

ii: \*(i+2) = ii;

iii: \*(i + 4) = rand();

This ends up writing 6 bytes, a complete header, which ends up looking in our case like:

5e45 5e45 \*\*\*\*

Keeping in line with what we expect: fd and bk are written to with the fd and bk from our header, and the magic comes from somewhere, who knows.

We’ll simply set our pointer 6 bytes before what we actually want to remove- the ret.

Fortunately for us, these decode to valid instructions that don’t destroy program execution before the door gets unlocked. (They, in fact, write to values in memory far outside of the used range)

Without a ret to run, program execution falls through to the unlock door. ( ͡° ͜ʖ ͡°)

Password:

ffffffffffffffffffffffffffffffff5e455e45

^ padding ^ header overwrite

**Lagos**

I pulled out quite an unorthodox approach for this one…

We get the following bytes- 41-5a + 61-7a

We somehow need to jump to 4650

Perhaps a ROP chain of conditional\_unlock->getsn->0 could do the trick.

cond\_unlock- Gets r14 to non-zero (7e).

getsn- Get our shellcode.

0- Run our shellcode.

Some padding may be required, as we can't write a zero, but we can use the zero later on the stack. It's at 4410, we are at 43fe.

cond\_unlock and getsn are perfectly addressed for ASCII. What else is?

464e is a great ret.

cond\_unlock @ 4446

getsn @ 4650

Writing to 0 is absolutely OK :D

Code exec.

*Writeup:*

Well, this one is neat. We’re given almost unlimited bytes to work with and can overwrite code to gain control flow (0x200) BUT we can only use capital A-Z and lowercase a-z. This essentially cripples us as we only get bytes 41-5a and bytes 61-7a. Anything other than that discards it and the rest of the input as it’s copied on to the stack. The original heap is also memset after the fact so we can’t just jump to it- not like we can write a jump or branch instruction anyway. I ended up taking what I think was an unorthodox approach to this- I wrote a ROP chain that set registers accordingly and called gets, allowing us to actually enter a payload. We then jump to rets to sled along our chain until it ends at a 0000, jumping to our payload.

ROP Chain-

(r15 is set to 0 at the beginning of the chain. Notice how all these addresses are perfectly positioned to fit in ASCII.)

4464- conditional\_unlock\_door: Sets r14 (size of gets) to 7e.

4650- getsn- Grabs r14 bytes and places them at r15.

464e- ret. Simply a sled to keep going up the stack until we reach a 0000 that’s returned to.

Stage 1- ROP Chain

DDDDDDDDDDDDDDDDDFDPFNFNFNFNFNFNFNF (ASCII rep)

4444444444444444444444444444444444464450464e464e464e464e464e464e464e46 (hex rep)

^ Padding ^ ROP Chain

Stage 2- Chained gets (payload)

3740017f178387100712b012fc45

**Vladivostok**

ASLR!

#define aslrfix let r11 = 4400

This will allow me to continue debugging, although it will help to allow it to run through without it as well.

It doesn't seem to call functions...

4492 clears out the original so we can't ROP to it. Let's just fix that really quickly:

#define aslrfixb let r14 = 0FFF

44a0-455a: Print "Username (8 char max):"

455c-45b8: Prints a newline then a >>

45ba-45dc: Gets 0x8 bytes of input at address 0x2426... We have a constant!

r12 holds the location of printf in ASLR... Good... It would be insane to copy/paste printf like they've done everything else here...

r12 is 0x36a from ASLR base... If we can read it, we may have a pointer leak...

45de, uh, moves a guaranteed null byte into r14. Yay.

45e4 calls printf on our input... Arbitrary write between 0x2 and 0x6. Not too helpful. (Actually not, later I discover it doesn’t take arguments from the string)

45ec-45f6... Zeroes out the username, without copying it on to the stack. So the username doesn't matter in authentication, and we no longer have a constant... But we still have that write...

45f8-4686: Print "Password: (newline)"

Start at: 4688

4688-468a: Get the password and place it on the stack; we can overwrite the return address with this, as we get 0x14 bytes- the return address is 8 bytes from the start of the password

46ae-46ce: Call HSM-2, with the flag set at current SP.

46d0-474a: Print "Wrong!" (newline)

And ret.

printf is perfect for this. Let's take a look at the stack when it's called, because we're doing to \*have\* to have a pointer leak here or this isn't exploitable at all...

OK, this time around LockItAll printf is a bit more compliant with the usual. It takes arguments from sp.

We have a pointer leak! We get printf's starting position, but no stack pointer, as we don't have enough %x's to get that far.

%x%x. I have no clue why, but printf handily places its starting address for us right after sp. Funnily enough, their arguments system in this version is terribly broken- they're taking arguments from printf's data, not the user's.

We don't necessarily \*have\* to use gets... We could use any of the bajillion interrupt calls in here...

496c is quite interesting... We get an arbitrary r15 write! Simply the byte value right after the ROP call (sign extended)... We can make this a 0.

r15 write offset = 0x56c

gets offset = 0x574

FFFFFFFFFFFFFFFFadra0000adrb0000

printf offset: 0x36a

Phase 1 (pointer leak): 25782578

Phase 2 (ROP chain): Get from Python script

Phase 3 (Shellcode): 324000ffb0121000

**Writeup:**

Well, this one was quite interesting!

To defeat ASLR, you need to find either a constant or a pointer leak. This time, I found the latter.

LockItAll’s printf has been changed- it now takes arguments from the stack, as usual. Funnily enough, they accidentally took arguments from printf’s stack (current sp) instead of from the pre-call stack. For some reason, sp + 0x2 contains the starting address of printf. I don’t know why, but %x%x allows us to leak the base pointer of the code when our username is printed! Later on, we can overwrite the return address with the password. With this, we can craft a ROP chain to gets our payload and call it, using a Python script to generate the addresses. We can’t simply call our password, as the stack pointer is still unknown to us.

**Bangalore**

DEP!

OK, so we need to map the stack as executable...

We have two approaches to this. r15 is cleared in this state, so we can map 0x0 as writable, gets to it, mark it as executable and jump to it.

Or, we can mark the stack as executable. This involves writing to r11 (which we have a gadget for) and jumping in to the set\_up\_protection code, which uses r11.

We have an r11 pop gadget at 4508.

We would need to jump in to 44f6.

The stack is in page 43.

Aaaand that's not going to work, because the set\_up\_protection loop still writes to the stack, after we've marked it executable... To plan B!

We somehow need to bypass the problem of mark\_writeable moving r15 to r14, zeroing it out...

448e is the perfect gadget, it sets r14 to 0xa from a putchar and zeroes out r15.

And Plan B fails as well! When the interrupt to mark page 0 returns, it cannot, as it as attempting to execute a writeable page.

4512 is an interesting gadget, but it's at the very beginning of login... It subtracts sp by 0x10.

How far can we take this chain, and are 3ff0 and 4000 in different pages...?

We have pretty good wiggle room for this chain.

We can jump in to the middle of functions! There's no need at all for moving to r15!

So, let's move sp to the middle of the heap in our shellcode so we're not writing to the stack, which is executable.

The page we need to map is 0x3f, and the mode is 0x0 (execute).

We need to jump to 44ba.

**Writeup:**

LockItAll has introduced DEP to the MSP430! Wow! Let’s break it!

This one took me a really long time to figure out, but in the end the solution was simple. You’re immediately given a return address overwrite so you can start a ROP chain. Gadgets suck in this level, you’re not going to do much of anything with them. It took me the longest time to realize that I did not need the nonexistent gadgets to do register writes and call map\_executable- that function simply pushed its arguments on to the stack, so we can jump in with our arguments already in place!

Well, that gets us a mapping of the stack as executable, but that means we can no longer write to it. To call our interrupt (which pushes the return address on to the stack), we’ll simply place the stack pointer at a random page that’s writeable- I picked 0x3000.

31400030324000ffb0121000FFFFFFFFbe440000000000003f000000ee3f

^Shellcode ^Padding ^map\_e ^Arguments ^Stack jump