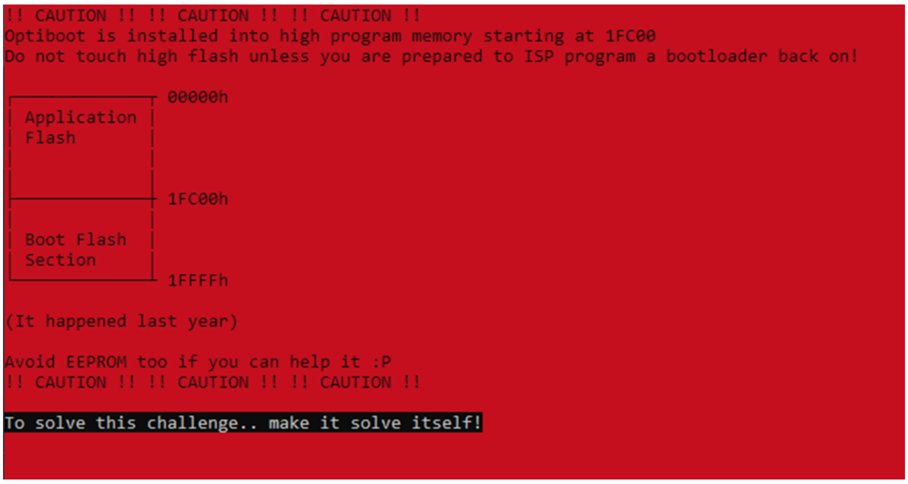
# Hackable Badge Challenge Walkthrough For SANS EMEA & NCSC UK’s CyberThreat24

## Solution For *“A Nice Edit”* By Badge Challenge Author, Secure Impact’s Security Engineer, Nathan Taylor

Just like in the previous challenge, we’re prompted to open a serial console to proceed.

```
[BOOT] Firmware OK
[BOOT] Complete. Welcome!
There are currently 7 unsolved challenges
This challenge carries a high risk of soft-bricking your badge!
It's strongly recommended to solve all the others first :)
Send a single LF to start the challenge.
```

On the test badge I’m using here I haven’t completed all previous challenges and I’m receiving a rather apt warning that I should finish those first. Accepting that warning a rather more in one’s face warning appears.



```
!! CAUTION !! !! CAUTION !! !! CAUTION !!
Optiboot is installed into high program memory starting at 1FC00
Do not touch high flash unless you are prepared to ISP program a bootloader back on!

 ________________ 00000h
| Application    |
| Flash          |
|                |
|                |
|________________ 1FC00h
|                |
| Boot Flash     |
| Section        |
|________________ 1FFFFh

(It happened last year)

Avoid EEPROM too if you can help it :P
!! CAUTION !! !! CAUTION !! !! CAUTION !!

To solve this challenge.. make it solve itself!
```

The processors used on the hackable badges are ATmega1284Ps. Like many in the ATmega series of parts this chip lacks any USB functionality. Instead, an FTDI serial converter chip is being used, connected to one of the hardware UARTs. This is how we’re able to get a serial console to the badge over USB.

There are a handful of ways to program an ATmega chip, however we’re particularly interested in two of them. The first is In System Programming, or ISP. This operates over SPI and allows a programmer to write to flash. The MOSI, MISO, RESET and SCK pins are used when performing ISP. This is notably *not* USB, and delegates were not expected to bring an ISP programmer with them to the event!

To facilitate programming over USB, a small bootloader is provisioned onto the microcontroller that can then use self-programming. This bootloader checks if a computer is trying to enter programming mode, otherwise it hands over to the normal user code. When we dumped flash in the previous challenge this is what we were doing.

But why the warning and long-winded explanation? This challenge is going to require us to *write* to flash. If we erase all of flash and write our new program, we get exactly one try because in the process we would have removed the bootloader! While the bootloader can be reinstalled using ISP it's game-over for USB programming.

avrdude provides the `-D` flag which instructs the programmer to erase the minimum amount of flash required for the new firmware, rather than a complete chip erase. You **must** use this flag at all times unless you have an ISP programmer available for recovery—I had one with me during the event in anticipation of people disregarding this warning.

With all that said, shall we get back to the actual challenge?

From the previous challenge I trust you're now familiar with the process of finding string usages. Let's have a look for that warning message:

```
void FUN_code_006863(void)

{
  HardwareSerial::write(0x1b);
  Print::println(&Serial);
  HardwareSerial::write(0x1b);
  Print::println(&Serial);
  HardwareSerial::write(0x1b);
  Print::println(&Serial);
  Print::println(&Serial,s_!!_CAUTION_!!_!!_CAUTION_!!_!!_C_mem_07b5);
  Print::println(&Serial,s_Optiboot_is_installed_into_high_p_mem_07df);
  Print::println(&Serial,s_Do_not_touch_high_flash_unless_y_mem_0820);
  Print::println(&Serial);
  Print::println(&Serial,s_00000h_mem_0875);
  Print::println(&Serial,s_|_Application_|_mem_08aa);
  Print::println(&Serial,s_|_Flash_|_mem_08ba);
  Print::println(&Serial,s_|_|_mem_08ca);
  Print::println(&Serial,s_|_|_mem_08ca);
  Print::println(&Serial,s_1FC00h_mem_08da);
  Print::println(&Serial,s_|_|_mem_08ca);
  Print::println(&Serial,s_|_Boot_Flash_|_mem_090f);
  Print::println(&Serial,s_|_Section_|_mem_091f);
  Print::println(&Serial,s_1FFFFh_mem_092f);
  Print::println(&Serial);
  Print::println(&Serial,s_(It_happened_last_year)_mem_0964);
  Print::println(&Serial);
  Print::println(&Serial,s_Avoid_EEPROM_too_if_you_can_help_mem_097c);
  Print::println(&Serial,s_!!_CAUTION_!!_!!_CAUTION_!!_!!_C_mem_07b5);
  HardwareSerial::write(0x1b);
  Print::println(&Serial);
  Print::println(&Serial);
  Print::println(&Serial,s_To_solve_this_challenge.._make_i_mem_09a7);
  Print::println(&Serial);
  return;
}
```

This doesn't look super useful; let's check the calling function instead:

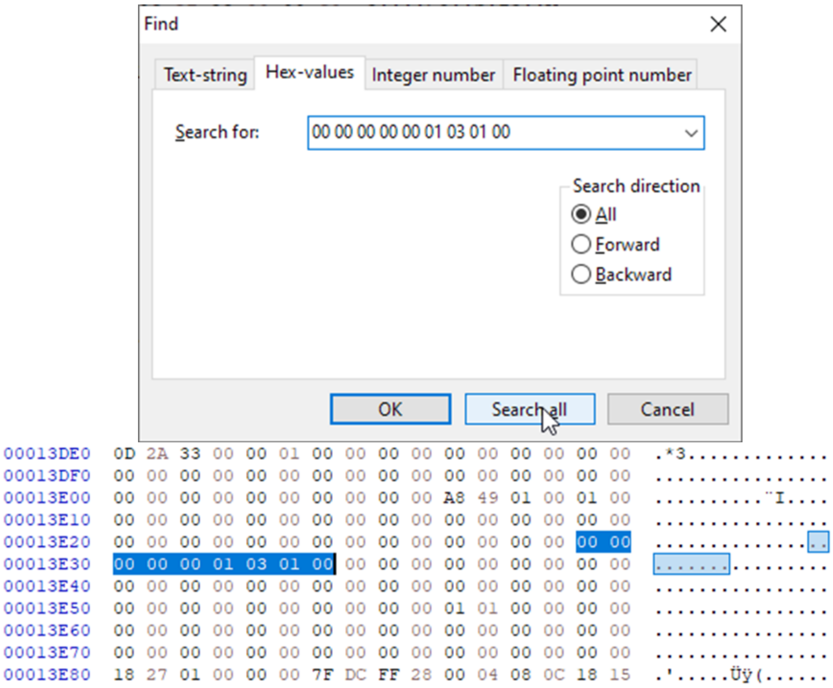
```
void FUN_code_006905(void)

{
  byte bVar1;
  undefined2 uVar2;
  ushort uVar3;

  bVar1 = BYTE_mem_025b;
  if (bVar1 == 0x69) {
    WORD_mem_166f._1_1_ = 0;
    WORD_mem_166f._0_1_ = 8;
    PTR_mem_1672._1_1_ = 0x1b;
    PTR_mem_1672._0_1_ = 7;
  }
  else {
    FUN_code_00520b(&DAT_mem_20cf,2,5,0x3f,0x36,0xd2);
    FUN_code_004265(&DAT_mem_20cf,0x53);
    FUN_code_0044a6(&DAT_mem_20cf,0x48,0x10,0xd7);
    FUN_code_0044a6(&DAT_mem_20cf,0x48,0x1a,0xb2);
    FUN_code_004265(&DAT_mem_20cf,0xa1);
    FUN_code_0044a6(&DAT_mem_20cf,0x48,0x28,0xe2);
    FUN_code_0044a6(&DAT_mem_20cf,0x48,0x32,0xec);
    uVar2 = HardwareSerial::available(&Serial);
    if ((((char)uVar2 != '\0' || (char)((uint)uVar2 >> 8) != '\0') &&
        (uVar3 = HardwareSerial::read(&Serial), uVar3 == 10)) {
                    // Print warning message
      FUN_code_006863();
    }
    if ((DAT_mem_1675 & 2) != 0) {
      PTR_mem_1672._1_1_ = 0x1b;
      PTR_mem_1672._0_1_ = 0x11;
    }
  }
  return;
}
```

What we're seeing here is interesting. There's a check for a specific byte value, and either some code runs, or the challenge prompt is issued. Given the name of the challenge, it's rather safe to assume we're going to need to edit something, and in this case what we need to do is to get to that first code-path.

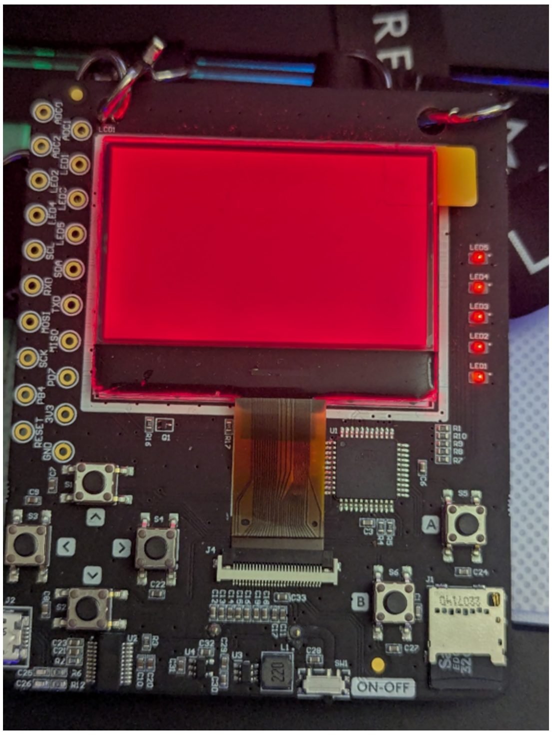
While we could invert the condition itself, it's somewhat simpler to just change that single byte at `mem:025B` to be $69_h$. We know how memory maps to `flash.bin`, so we could find out the offset "properly", or we could just do a search for the surrounding bytes.




We can make a simple edit of that first $01_h$ to $69_h$, save, then run the following command to flash the firmware back to the badge:

```
avrdude -V -v -D -pm1284p -carduino -b115200 -PCOM9 -Uflash:w:flash.bin:r
```

Congratulations. Your badge no longer works!




The good news is this was expected. If you see anything other than a pure red screen at this point it's likely you didn't flash the badge correctly.

Why, though, are we seeing a red screen? In embedded software development, code integrity is often a concern; these hackable badges are no different. Remember that `[BOOT] Firmware OK` we've been seeing every time we connect over serial? That's not just flavour text. The first thing the badges do when they power on is validate their own firmware.

You might already have some ideas of how this is performed, especially if you're familiar with AVR, but let's take a look at the code. Following on from the reset handler we can jump through the initialisation code to land at our `void main(void)` [1] function at `code:905f`.

Before the bulk of setup occurs, we can see a large loop, some conditions, and then finally that output of `Firmware OK` we're looking for. Ignore the `GPIO_...` names; Ghidra is getting a little confused with register names.

```
bVar3 = 0;
while( true ) {
  bVar7 = WORD_mem_0232._0_1_;
  bVar20 = WORD_mem_0232._1_1_;
  bVar2 = WORD_mem_0234._0_1_;
  bVar13 = WORD_mem_0234._1_1_;
  bVar8 = (byte)uVar10;
  bVar11 = (byte)(uVar10 >> 8);
  bVar12 = (byte)uVar14;
  bVar15 = (byte)(uVar14 >> 8);
  if (bVar13 <= bVar15 &&
      (bVar12 < bVar2 ||
      (byte)(bVar12 - bVar2) < (bVar11 < bVar20 || (byte)(bVar11 - bVar20) < (bVar8 < bVar7))) <=
      (byte)(bVar15 - bVar13)) break;
  RAMPZ = (undefined)((uint3)((byte *)CONCAT12(bVar12,uVar10) + 1) >> 0x10);
  bVar20 = bVar5 ^ *(byte *)CONCAT12(bVar12,uVar10);
  bVar7 = bVar20 >> 4 | bVar20 << 4;
  bVar5 = bVar20 >> 3 ^ ((bVar7 ^ bVar20) & 0xf0) + CARRY1(bVar20,bVar20) ^ bVar3;
  bVar22 = bVar11 != 0xff || (byte)(bVar11 + 1) < (bVar8 != 0xff);
  uVar10 = CONCAT11(bVar11 - ((bVar8 != 0xff) + -1),bVar8 + 1);
  uVar14 = CONCAT11(bVar15 - ((bVar12 != 0xff || (byte)(bVar12 + 1) < bVar22) + -1),
                    bVar12 - (bVar22 + -1));
  bVar3 = bVar20 >> 4 ^ bVar20 ^ (bVar20 ^ bVar7) * '\x02' & 0xe0;
}
if (bVar3 != 0 || bVar5 != 0) {
  bVar3 = GPIO_GPIORA;
  GPIO_GPIORA = bVar3 | 0x70;
  uVar16 = GPIO_GPIORB;
  uVar14 = CONCAT11(bVar5,uVar16) & 0xffcf;
  GPIO_GPIORB = (char)uVar14;
  bVar3 = GPIO_GPIORB;
  GPIO_GPIORB = bVar3 | 0x40;
  bVar3 = GPIO_GPIOR1;
  GPIO_GPIOR1 = bVar3 | 0xf8;
  uVar16 = GPIO_GPIOR2;
  uVar14 = CONCAT11((char)(uVar14 >> 8),uVar16) | 0xf8;
  GPIO_GPIOR2 = (char)uVar14;
  *(undefined3 *)(iVar21 + -2) = 0x90e2;
  BOOT(uVar14);
  do {
                    // WARNING: Do nothing block with infinite loop
  } while( true );
}
*(undefined *)CONCAT11(DAT_mem_17b4,DAT_mem_17b3) = 2;
*(undefined *)CONCAT11(DAT_mem_17b0,DAT_mem_17af) = 0;
*(undefined *)CONCAT11(DAT_mem_17b2,DAT_mem_17b1) = 0xcf;
DAT_mem_17bb = 0;
*(undefined *)CONCAT11(DAT_mem_17b8,DAT_mem_17b7) = 6;
*(byte *)CONCAT11(DAT_mem_17b6,DAT_mem_17b5) = *(byte *)CONCAT11(DAT_mem_17b6,DAT_mem_17b5) | 0x98
;
*(byte *)CONCAT11(DAT_mem_17b6,DAT_mem_17b5) = *(byte *)CONCAT11(DAT_mem_17b6,DAT_mem_17b5) & 0xdf
;
uVar16 = 0xc;
*(undefined3 *)(iVar21 + -2) = 0x9110;
Print::println(&Serial,s_[BOOT]_Firmware_OK_mem_0c4b);
bVar3 = GPIO_GPIORA;
```

You would be correct to reason that this is our firmware check. We first have a loop that calculates a checksum, compares it to 0, and errors if it is not zero. Reasoning from this, we need to amend our firmware such that the checksum comes out as $0000_h$. This poses two questions: what checksum is being performed, and how do we ensure its value?

Both are relatively simple questions to answer. The AVR compiler toolchain includes a number of pre-defined CRC implementations in `crc16.h` . If we compare these to the disassembly of the firmware, that loop reveals itself to be repeated calls to `_crc_xmodem_update` .

To make the checksum valid, we *could* just edit any random bytes in the firmware. If that feels like an icky thought it's because it is. This is generally the point to stop and think "what did the original software engineer do here?". There's a very common place for CRCs and corrective bytes to appear in firmware, and that's tacked onto the end of the firmware blob, after the compiler has done its dues.

If we go and look at `flash.bin`, we can see two seemingly random bytes after what's quite clearly a bunch of strings. The especially astute may also have noticed that the initialisation data for `.data` ended at $149A6_h$, meaning these two bytes were tacked on afterwards.

```
000148E0  65 54 72 61 73 68 00 44 6F 6F 6D 00 52 65 74 75  eTrash.Doom.Retu
000148F0  72 6E 00 53 65 74 74 69 6E 67 73 00 53 65 74 20  rn.Settings.Set 
00014900  4E 61 6D 65 00 53 65 74 20 41 6C 69 61 73 00 53  Name.Set Alias.S
00014910  79 73 74 65 6D 00 43 68 61 6C 6C 65 6E 67 65 73  ystem.Challenges
00014920  00 4E 75 6D 65 72 69 63 69 73 6D 00 57 6F 72 6C  .Numericism.Worl
00014930  64 20 31 2D 31 00 50 61 69 6E 74 20 62 79 20 4E  d 1-1.Paint by N
00014940  75 6D 62 65 72 73 00 46 6C 61 73 68 79 00 42 6C  umbers.Flashy.Bl
00014950  69 6E 6B 65 6E 6C 69 67 68 74 73 00 4C 6F 73 74  inkenlights.Lost
00014960  20 53 6F 6D 65 74 68 69 6E 67 00 53 63 61 6E 78   Something.Scanx
00014970  69 65 74 79 00 45 63 68 6F 20 53 65 72 76 69 63  iety.Echo Servic
00014980  65 00 41 20 4E 69 63 65 20 45 64 69 74 00 4D 61  e.A Nice Edit.Ma
00014990  69 6E 20 4D 65 6E 75 00 42 61 63 6B 20 74 6F 20  in Menu.Back to 
000149A0  42 61 64 67 65 00 FB 77                          Badge.ûw
```

A convenient quirk of CRC16 XMODEM (and of many other CRCs) is adding the compute CRC to the end of a block of data causes its CRC to now calculate as 0. We can sanity check our previous assumption by calculating the CRC of the un-edited firmware without the last two bytes, and we receive `FB77` as expected. If we calculate the checksum *with* our edit, we instead receive `D69D`. Let's edit that into our firmware and then re-flash to the badge.

At this point, your badge should now be taking you back to the name display as normal, but the challenge isn't marked as completed. Head into the challenges menu, select A Nice Edit, and you'll see the challenge completed display.

Congratulations! That's all 9 of the challenges for the CyberThreat 2024 badge! Hopefully you learnt some things along the way, or at the very least had as much fun with them as I did writing them 😊.

[1] That's not a familiar signature for main? If you're used to programming for desktop platforms, you can expect to have a C runtime that handles parsing command line arguments, passes them to main, then uses the return value from main as a process exit code. We have none of that in embedded land! The arguments to main are therefore going to be empty. Depending on the specific framework being used it may be acceptable to return from main. For example, this project was compiled using PlatformIO and a Wiring-based framework. In this instance, a return from main would land in _exit which contains an infinite loop. This is often not the case, and it's generally advised to ensure that you never return from main when writing embedded code.

SECURE IMPACT

To highlight this distinction and avoid confusion, I often opt to name my main function something like _entry instead, but the framework used here doesn't play so nice with that.