FireEye

FLARE

SECURITY REIMAGINED

Flare-On 4: Challenge 9 Solution

Challenge Author: Joshua Homan

Overview

This solution takes a couple of different approaches for solving the challenge. The first section covers how to solve the challenge using only static analysis with `radare2`. The second section covers dynamic analysis with `simavr` and the final section covers solving using an Arduino UNO.

Initial Analysis

This challenge is an ASCII file that each line starts with a colon character followed by HEX characters as shown in Figure 1. The file is an Intel HEX file, a file format that is commonly used to program microcontrollers. A detailed write-up on the Intel HEX file format can be found on Wikipedia[1].

```
:100000000C9462000C948A000C948A000C948A0070
:100010000C948A000C948A000C948A000C948A0038
:100020000C948A000C948A000C948A000C948A0028
:100030000C948A000C948A000C948A000C948A0018
<- truncated ->
```

Figure 1 - Intel HEX of `remorse_09.ino.hex`

Now that we know the challenge is an Intel HEX file, the first thing we'd like to do is convert the file to binary to see if we can extract any additional information. One way to accomplish this is to use `avr-objcopy` from `toolchain-avr`[2]. `Avr-objcopy` is a useful utility that converts between various files types such as binary, elf and Intel Hex. An example command line is shown in Figure 2.

```
      avr-objcopy -I ihex -O binary remorse_09.ino.hex remorse_09.ino.bin
```

Figure 2 - avr-objcopy syntax

Once the challenge is converted to binary, we can run strings and get the results shown in Figure 3.

```
#+$+%+a
/_'1
Correct Pin State:
```

---

[1] https://en.wikipedia.org/wiki/Intel_HEX

[2] https://github.com/arduino/toolchain-avr

```
Flare-On 2017 Adruino UNO Digital Pin state:
```

Figure 3 - Strings output

Taking a look at the strings, we are provided with a hint indicating the challenge is for an Arduino UNO[3] based on the string "`Flare-On 2017 Adruino UNO Digital Pin state:`".

AVR Quick Overview

The Arduino UNO is built around an 8-bit ATmega328p processor that uses the Atmel AVR instruction set. The processor has 32 general purpose registers, labeled `r0-r31`, along with a few special registers we need to be aware of: stack pointer (`SP`), status register (`SREG`) and program counter (`PC`).

Register labels	**Description**
`r0-r31`	General purpose registers
`SP`	Stack pointer
`PC`	Program counter
`SREG`	Status register
`x`	Memory access register based on `r27:r26`
`y`	Memory access register based on `r29:r28`
`z`	Memory access register based on `r31:r30`

Table 1 - AVR registers of interest

The ATmega328p processor equipped with 2KB of SRAM. To overcome the limitation of referencing memory with only an 8-bit register, the processor has three special registers that combine two general purpose registers (`x`, `y` and `z` from Table 1). These registers have an interesting property in which they can be incremented or decremented after accessing. Take for example the instruction "`ld r25, z+`". This instruction loads the value stored at the memory address pointed to by the `z` register (`r31:r30`) into `r25` and then increments the value contained in `z`.

---

[3] https://store.arduino.cc/usa/arduino-uno-rev3

Another important property of the ATmega328p is that it uses a modified Harvard architecture, meaning date and code are stored in different memory locations (Flash for code and SRAM for data). For example, the data stored at memory location `0x500`, is not the same as the code located at address `0x500`.

When calling functions, the arguments to the function are placed in the registers starting with `r25` down through `r8`. The registers containing a function return value depends on the data type. Bytes are stored in `r24`, words in registers `25:24`, 32-bits in `r22-r25` and 64-bit in `r18-r25`[4].

Static Analysis with Radare2

For static analysis, the one way to get started is `radare2`[5]. `Radare2` is an open source reverse engineering framework that supports a wide range of architectures, file formats and operating systems. The challenge can be loaded into `radare2` using the command line `r2 -a avr remorse.bin` and initial analysis can be performed using the `aaaaa` command. The output is shown in Figure 4.

```
[0x000000c4]> aaaaa
[opcode st @de returned 0 cycles.th sym. and entry0 (aa)
[x] Analyze all flags starting with sym. and entry0 (aa)
[ ]
[Value from 0x00000000 to 0x00001156
aav: from 0x0 to 0x1156
[x] Analyze len bytes of instructions for references (aar)
[opcode lds @c12 returned 0 cycles.
opcode std @1b0 returned 0 cycles.
[x] Analyze function calls (aac)
[x] Emulate code to find computed references (aae)
[Cannot find section boundaries in here
[x] Analyze consecutive function (aat)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[x] Type matching analysis for all functions (afta)
```

Figure 4: Initial analysis using r2

The `afl` command lists the functions identified by radare2. The results are shown in Figure 5. We can see that `29` functions are identified and the entry point is labeled `entry0`.

```
[0x000000c4]> afl
0x000000c4   12 3108 -> 84   entry0
0x000001d8    1 24           fcn.000001d8
0x000001f0    3 20           fcn.000001f0
0x000002ce    4 66           fcn.000002ce
0x000003e2    5 20   -> 24   fcn.000003e2
```

---

[4] http://www.atmel.com/webdoc/avrlibcreferencemanual/FAQ_1faq_reg_usage.html
[5] http://rada.re/r/

```
0x000003f6   18 82          fcn.000003f6
0x00000448    7 82          fcn.00000448
0x0000049a    9 96          fcn.0000049a
0x000004fa    9 120         fcn.000004fa
0x00000572    1 26   -> 64   fcn.00000572
0x00000596    3 38          loc.00000596
0x000005bc    6 116         fcn.000005bc
0x00000630    5 6    -> 208  fcn.00000630
0x0000063a    5 42          fcn.0000063a
0x00000664    1 40          fcn.00000664
0x0000068c   11 170         fcn.0000068c
0x00000736    1 4           fcn.00000736
0x0000087e    6 46          fcn.0000087e
0x000008ac    3 58          fcn.000008ac
0x000008e6    6 92          fcn.000008e6
0x00000942    7 40   -> 60   fcn.00000942
0x0000096a    9 118  -> 132  fcn.0000096a
0x000009e0    8 140         fcn.000009e0
0x00000a6c    9 212         fcn.00000a6c
0x00000b40    5 94          fcn.00000b40
0x00000bf8    2 138  -> 148  fcn.00000bf8
0x00000c8c    5 68          fcn.00000c8c
0x00000cd0    1 12          fcn.00000cd0
0x00000cdc    3 12   -> 14   fcn.00000cdc
```

Figure 5: Functions identified by r2

Let's take a closer look at the entry point by disassembling it with the command `pd @ entry0`. In Figure 6, we can see `entry0` initializing memory starting at addresses `0xdc` with the `lpm` instruction. The `lpm` instruction loads a byte from program memory and stores it to data memory.

```
[0x000000c4]> pd @ entry0
/ (fcn) entry0 84
|   entry0 ();
|              ; JMP XREF from 0x00000000 (fcn.000003e2)
|           0x000000c4      1124           clr r1
|           0x000000c6      1fbe           out 0x3f, r1   ;IO SREG: flags
|           0x000000c8      cfef           ser r28
|           0x000000ca      d8e0           ldi r29, 0x08
|           0x000000cc      debf           out 0x3e, r29  ;IO SPH: Stack higher bits SP8-SP10
|           0x000000ce      cdbf           out 0x3d, r28  ;IO SPL: Stack lower bits SP0-SP7
|           0x000000d0      15e0           ldi r17, 0x05
|           0x000000d2      a0e0           ldi r26, 0x00
|           0x000000d4      b1e0           ldi r27, 0x01
|           0x000000d6      eaee           ldi r30, 0xea
|           0x000000d8      fce0           ldi r31, 0x0c
|       ,=< 0x000000da      02c0           rjmp 0xe0
|      .--> 0x000000dc      0590           lpm r0, z+
|      ||   0x000000de      0d92           st x+, r0
|      !|      ; JMP XREF from 0x000000da (entry0)
```

```
|       |`-> 0x000000e0      ac36           cpi r26, 0x6c
|       |    0x000000e2      b107           cpc r27, r17
|       `==< 0x000000e4      d9f7           brne 0xdc
|            0x000000e6      26e0           ldi r18, 0x06
|            0x000000e8      ace6           ldi r26, 0x6c
|            0x000000ea      b5e0           ldi r27, 0x05
|        ,=< 0x000000ec      01c0           rjmp 0xf0
|       .--> 0x000000ee      1d92           st x+, r1
|       !|      ; JMP XREF from 0x000000ec (entry0)
|       |`-> 0x000000f0      ac32           cpi r26, 0x2c
|       |    0x000000f2      b207           cpc r27, r18
|       `==< 0x000000f4      e1f7           brne 0xee
|            0x000000f6      10e0           ldi r17, 0x00
|            0x000000f8      c2e6           ldi r28, 0x62
|            0x000000fa      d0e0           ldi r29, 0x00
|        ,=< 0x000000fc   ~  04c0           rjmp 0x106
|        |   ;-- r30:
|        |   0x000000fd      c0             unaligned
|       .--> 0x000000fe   ~  2197           sbiw r28, 0x01
|       ||   ;-- r1:
|       ||   ;-- r8:
|       ||   0x000000ff      97             unaligned
|       ||   0x00000100      fe01           movw r30, r28
|       ||   0x00000102      0e946806       call fcn.00000cd0
|       !|      ; JMP XREF from 0x000000fc (entry0)
|       |`-> 0x00000106      c136           cpi r28, 0x61
|       |    0x00000108      d107           cpc r29, r17
|       `==< 0x0000010a      c9f7           brne 0xfe
|            0x0000010c      0e94fc05       call fcn.00000bf8
|        ,=< 0x00000110      0c947306       jmp 0xce6
```

Figure 6 – Disassembly of entry point function entry0

Notice two functions are a called from `entry0`: `fcn.00000cd0` and `fcn.00000bf8`. Taking a closer look at `fcn.00000bf8`, disassembly shown in Figure 7, we can see some initial processor setup by configuring timers and at the end of the function there is an infinite loop calling two functions `fcn.00000b40` and `fcn.000003e2`.

```
[0x000000c4]> pd @ fcn.00000bf8
/ (fcn) fcn.00000bf8 148
|   fcn.00000bf8 ();
|              ; CALL XREF from 0x0000010c (entry0)
|            0x00000bf8      7894           sei
|            0x00000bfa      84b5           in r24, 0x24   ; IO TCNT2: Timer/Counter2 (8 bits).
|            0x00000bfc      8260           ori r24, 0x02
|            0x00000bfe      84bd           out 0x24, r24  ; IO TCNT2: Timer/Counter2 (8 bits).
|            0x00000c00      84b5           in r24, 0x24   ; IO TCNT2: Timer/Counter2 (8 bits).
|            0x00000c02      8160           ori r24, 0x01
|            0x00000c04      84bd           out 0x24, r24  ; IO TCNT2: Timer/Counter2 (8 bits).
|            0x00000c06      85b5           in r24, 0x25   ; IO TCCR2: Timer/Counter2 Control
Register (8 bits).
|            0x00000c08      8260           ori r24, 0x02
```

```
|           0x00000c0a      85bd           out 0x25, r24 ; IO TCCR2: Timer/Counter2 Control
Register (8 bits).
|           0x00000c0c      85b5           in r24, 0x25  ; IO TCCR2: Timer/Counter2 Control
Register (8 bits).
|           0x00000c0e      8160           ori r24, 0x01
|           0x00000c10      85bd           out 0x25, r24 ; IO TCCR2: Timer/Counter2 Control
Register (8 bits).

<…>

|           0x00000c76      80937a00       sts 0x7a, r24
|           0x00000c7a      1092c100       sts 0xc1, r1
\           0x00000c7e      0e945604       call fcn.000008ac
|              ; JMP XREF from 0x00000c8a (fcn.00000bf8)
|       .-> 0x00000c82      0e94a005       call fcn.00000b40
|       |   0x00000c86      0e94f101       call fcn.000003e2
|       `=< 0x00000c8a      fbcf           rjmp 0xc82
```

Figure 7 - Disassembly of function fcn.00000bf8

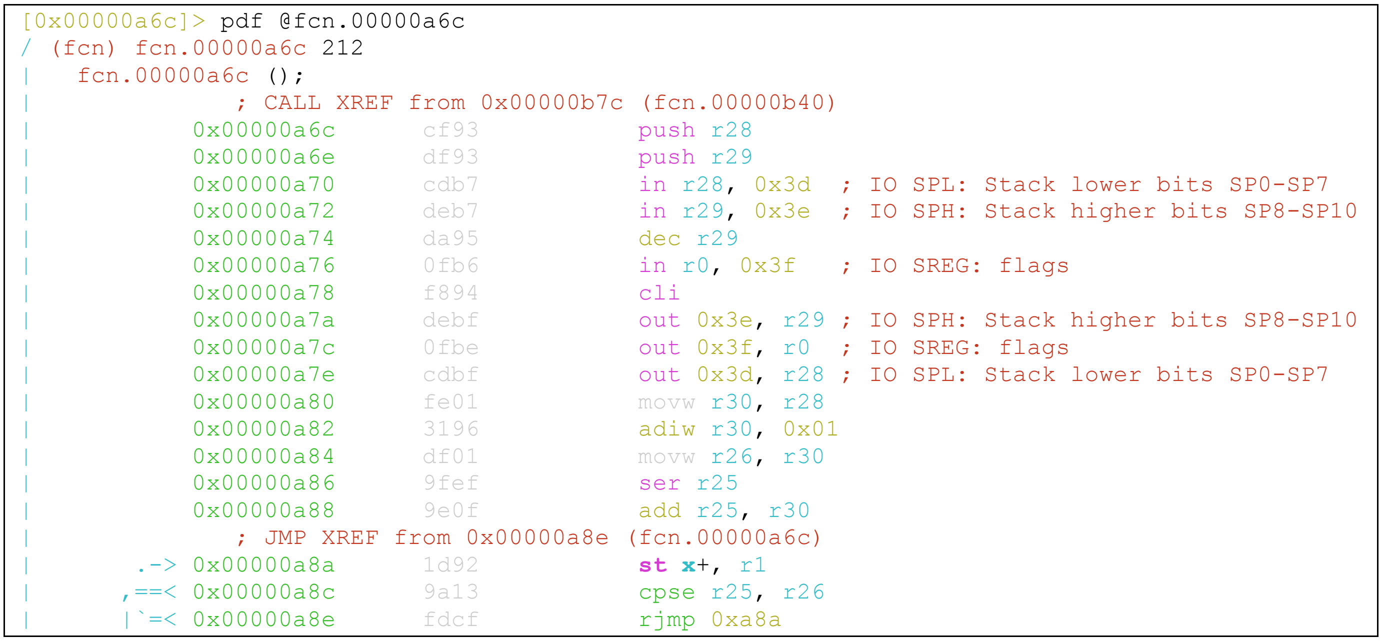
Inspecting the disassembly of `fcn.00000b40`, shown in Figure 8, we can see a call to another function `fcn.0000087e` at address `0xb42`. The code at address `0xb4c` compares the return value from this function with a value stored in memory at address `0x585`. If the values are different, the challenge continues executing through address `0xb50`. At this point we don't know what `fcn.0000087e` does but we do know the return value significantly affects program flow. Looking further down in the function, we can see the return value is passed as a single argument to the function `fcn.00000a6c` at offset `0xb7c`.

```
[0x000000c4]> pd @ fcn.00000b40
/ (fcn) fcn.00000b40 94
|   fcn.00000b40 ();
|       !      ; CALL XREF from 0x00000c82 (fcn.00000bf8)
|       |   0x00000b40      cf93           push r28
|       |   0x00000b42      0e943f04       call fcn.0000087e
|       |   0x00000b46      c82f           mov r28, r24
|       |   0x00000b48      80918505       lds r24, 0x585
|       |   0x00000b4c      c817           cp r28, r24
|      ,==< 0x00000b4e      01f1           breq 0xb90
|      ||   0x00000b50      60910005       lds r22, 0x500
|      ||   0x00000b54      70910105       lds r23, 0x501
|      ||   0x00000b58      8fe8           ldi r24, 0x8f
|      ||   0x00000b5a      95e0           ldi r25, 0x05
|      ||   0x00000b5c      0e949b03       call fcn.00000736
|      ||   0x00000b60      42e0           ldi r20, 0x02
|      ||   0x00000b62      50e0           ldi r21, 0x00
|      ||   0x00000b64      6c2f           mov r22, r28
|      ||   0x00000b66      8fe8           ldi r24, 0x8f
|      ||   0x00000b68      95e0           ldi r25, 0x05
|      ||   0x00000b6a      0e944603       call fcn.0000068c
|      ||   0x00000b6e      64e2           ldi r22, 0x24
|      ||   0x00000b70      75e0           ldi r23, 0x05
```

```
|       ||   0x00000b72      8fe8           ldi r24, 0x8f
|       ||   0x00000b74      95e0           ldi r25, 0x05
|       ||   0x00000b76      0e943203       call fcn.00000664
|       ||   0x00000b7a      8c2f           mov r24, r28
|       ||   0x00000b7c      0e943605       call fcn.00000a6c
|       ||   0x00000b80      0197           sbiw r24, 0x01
|      ,===< 0x00000b82      21f0           breq 0xb8c
|      |||   0x00000b84      60e0           ldi r22, 0x00
|      |||   0x00000b86      8de0           ldi r24, 0x0d
|      |||   0x00000b88      0e944d02       call fcn.0000049a
|      `---> 0x00000b8c      c0938505       sts 0x585, r28
|       `--> 0x00000b90      68ee           ldi r22, 0xe8
|        |   0x00000b92      73e0           ldi r23, 0x03
|        |   0x00000b94      80e0           ldi r24, 0x00
|        |   0x00000b96      90e0           ldi r25, 0x00
|        |   0x00000b98      cf91           pop r28
\        `=< 0x00000b9a      0c94de02       jmp fcn.000005bc
 [0x000000c4]>
```

Figure 8 - Disassembly of function fcn.00000b40

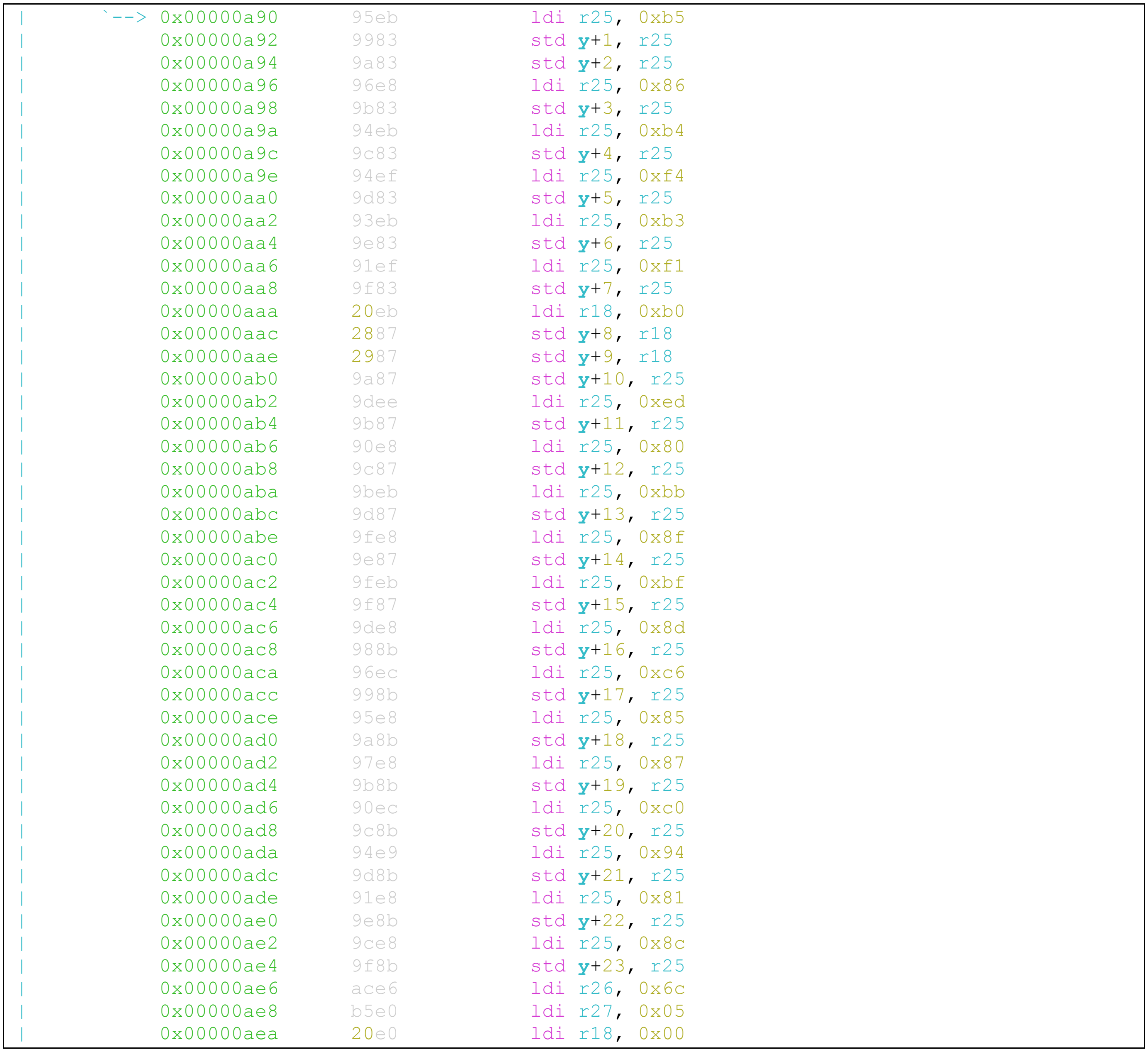
After inspecting function `fcn.00000a6c`, we can see the function starts by initializing the stack at address `0xa74` by decrementing the value obtained from the `SPH` register (the higher `8` bits of the stack pointer) by one and storing the result back with the instruction at address `0xa7a`. The disassembly for `fcn.00000a6c` is shown in Figure 9.



```
[0x00000a6c]> pdf @fcn.00000a6c
/ (fcn) fcn.00000a6c 212
|   fcn.00000a6c ();
|           ; CALL XREF from 0x00000b7c (fcn.00000b40)
|           0x00000a6c      cf93           push r28
|           0x00000a6e      df93           push r29
|           0x00000a70      cdb7           in r28, 0x3d  ; IO SPL: Stack lower bits SP0-SP7
|           0x00000a72      deb7           in r29, 0x3e  ; IO SPH: Stack higher bits SP8-SP10
|           0x00000a74      da95           dec r29
|           0x00000a76      0fb6           in r0, 0x3f   ; IO SREG: flags
|           0x00000a78      f894           cli
|           0x00000a7a      debf           out 0x3e, r29 ; IO SPH: Stack higher bits SP8-SP10
|           0x00000a7c      0fbe           out 0x3f, r0  ; IO SREG: flags
|           0x00000a7e      cdbf           out 0x3d, r28 ; IO SPL: Stack lower bits SP0-SP7
|           0x00000a80      fe01           movw r30, r28
|           0x00000a82      3196           adiw r30, 0x01
|           0x00000a84      df01           movw r26, r30
|           0x00000a86      9fef           ser r25
|           0x00000a88      9e0f           add r25, r30
|           ; JMP XREF from 0x00000a8e (fcn.00000a6c)
|       .-> 0x00000a8a      1d92           st x+, r1
|      ,==< 0x00000a8c      9a13           cpse r25, r26
|      |`=< 0x00000a8e      fdcf           rjmp 0xa8a
```

Figure 9 - Allocating space on the stack

At address `0xa90`, shown in Figure 10, we can see bytes being placed into a local variable in what appears to be populating a stack string.



```
|       `--> 0x00000a90      95eb           ldi r25, 0xb5
|            0x00000a92      9983           std y+1, r25
|            0x00000a94      9a83           std y+2, r25
|            0x00000a96      96e8           ldi r25, 0x86
|            0x00000a98      9b83           std y+3, r25
|            0x00000a9a      94eb           ldi r25, 0xb4
|            0x00000a9c      9c83           std y+4, r25
|            0x00000a9e      94ef           ldi r25, 0xf4
|            0x00000aa0      9d83           std y+5, r25
|            0x00000aa2      93eb           ldi r25, 0xb3
|            0x00000aa4      9e83           std y+6, r25
|            0x00000aa6      91ef           ldi r25, 0xf1
|            0x00000aa8      9f83           std y+7, r25
|            0x00000aaa      20eb           ldi r18, 0xb0
|            0x00000aac      2887           std y+8, r18
|            0x00000aae      2987           std y+9, r18
|            0x00000ab0      9a87           std y+10, r25
|            0x00000ab2      9dee           ldi r25, 0xed
|            0x00000ab4      9b87           std y+11, r25
|            0x00000ab6      90e8           ldi r25, 0x80
|            0x00000ab8      9c87           std y+12, r25
|            0x00000aba      9beb           ldi r25, 0xbb
|            0x00000abc      9d87           std y+13, r25
|            0x00000abe      9fe8           ldi r25, 0x8f
|            0x00000ac0      9e87           std y+14, r25
|            0x00000ac2      9feb           ldi r25, 0xbf
|            0x00000ac4      9f87           std y+15, r25
|            0x00000ac6      9de8           ldi r25, 0x8d
|            0x00000ac8      988b           std y+16, r25
|            0x00000aca      96ec           ldi r25, 0xc6
|            0x00000acc      998b           std y+17, r25
|            0x00000ace      95e8           ldi r25, 0x85
|            0x00000ad0      9a8b           std y+18, r25
|            0x00000ad2      97e8           ldi r25, 0x87
|            0x00000ad4      9b8b           std y+19, r25
|            0x00000ad6      90ec           ldi r25, 0xc0
|            0x00000ad8      9c8b           std y+20, r25
|            0x00000ada      94e9           ldi r25, 0x94
|            0x00000adc      9d8b           std y+21, r25
|            0x00000ade      91e8           ldi r25, 0x81
|            0x00000ae0      9e8b           std y+22, r25
|            0x00000ae2      9ce8           ldi r25, 0x8c
|            0x00000ae4      9f8b           std y+23, r25
|            0x00000ae6      ace6           ldi r26, 0x6c
|            0x00000ae8      b5e0           ldi r27, 0x05
|            0x00000aea      20e0           ldi r18, 0x00
```

Figure 10 - Initializing stack string

After the local variable is initialized, we can see a loop that is loading a byte from the stack variable at address `0xaec`. This loop is shown in Figure 11. The byte is XORed with the function argument stored in register `r24` and the loop index counter is added to it. The result is stored in data memory starting at address `0x56c` (the `x` register is set at address `0xae6`).

```
|           0x00000ae6      ace6           ldi r26, 0x6c
|           0x00000ae8      b5e0           ldi r27, 0x05
|       .-> 0x00000aec      9191           ld r25, z+
|       |   0x00000aee      9827           eor r25, r24
|       |   0x00000af0      920f           add r25, r18
|       |   0x00000af2      9d93           st x+, r25
|       |   0x00000af4      2f5f           subi r18, 0xff
|       |   0x00000af6      2731           cpi r18, 0x17
|       `=< 0x00000af8      c9f7           brne 0xaec
```

Figure 11 - Decode loop

Once the loop completes, the challenge compares the value stored at memory address `0x576` with the byte value `0x40` (‘`@`’). Being this far along in the Flare-On challenge, seeing a check for this character should be very interesting to us.

At this point we’ve identified a stack string, decoding loop and sanity check. The next step is to determine a value for the key, the first argument to the function, that results in a \``@`\` character in memory location `0x576`. After recreating the stack variable, we get the value shown in Figure 12.

```
\xb5\xb5\x86\xb4\xf4\xb3\xf1\xb0\xb0\xf1\xed\x80\xbb\x8f\xbf\x8d\xc6\x85\x87\xc0\x94\x81\x8c
```

Figure 12 - Stack string

We are interested in getting the 10th byte (`0xED`) to equal `0x40`. The offset `10` is calculated by subtracting the start of the string `0x56c` with `0x576`, the address of the sanity check character ‘`@`’. A simple solution is to brute force all values for the key (`r24`) because the key space is limited to only `256` possible keys (8-bit processor). A sample script is shown in Figure 13.

```
ctext =
bytearray("\xb5\xb5\x86\xb4\xf4\xb3\xf1\xb0\xb0\xf1\xed\x80\xbb\x8f\xbf\x8d\xc6\x85\x87\xc0\x
94\x81\x8c")

def decrypt(ctext, key):
       rvalue = bytearray()

       for x in range(len(ctext)):
              rvalue.append(((ctext[x] ^ key) + x) & 0xff)
       return str(rvalue)
```

```
for x in range(255):
       r = decrypt(ctext, x)

       if r[0x0a] == '@':
              print "Decrypt key: 0x%02X" % x
              print "Plaintext: %s" % r
              break
```

Figure 13 - Example brute force script

The output from running the script in Figure 13 shows the decryption key is `0xDB` and the decoded string is `no_r3m0rs3@flare-on.com`. An alternative to brute force would involve subtracting the index (`0x0A`) with the plaintext character `0x40` (‘`@`’) and XORing the encrypted text (`0xED`) to get the key (`0x40 – 0x0A ^ 0xED = 0xDB`).

Dynamic Analysis with Simavr

Another approach to solving the Arduino challenge is to use the GDB functionality included in `simavr`[6]. `Simavr` is an open source AVR simulator that enables us to execute and debug without any of the hardware. `Simavr` also supports executing Intel HEX file. The challenge can be executed with the command line shown in Figure 14.

```
run_avr -m atmega328p -f 160000000 --gdb remorse_09.ino.hex
```

Figure 14 - Example `run_avr` command line

After `simavr` loads, it pauses waiting for a remote debugger to attach to it.

```
GNU gdb (GDB) 7.10.1
(gdb) target remote :1234
Remote debugging using :1234
0x00000000 in ?? ()
(gdb)
```

As seen in Figure 15, we set a breakpoint at `0xb48`, where the return of `fcn.0000087e` is compared with the value stored in memory at `0x585`. One thing to be aware of with `avr-gdb` and `simarvr` is that setting breakpoints using the instruction address (`break *addr`) does not work. This is possibly a result of a bug or configuration issue. There are a couple solutions to this problem. The first option is to set the breakpoint relative to the `pc` register using the command format `break * $pc + <addr>`. This is the easiest solution when starting analysis because the program counter is set to zero. Another workaround is to treat the address as a function pointer using the format `break *(void(*)()) <addr>`.

---

[6] https://github.com/buserror/simavr

```
(gdb) break * $pc + 0xb48
Breakpoint 1 at 0xb48
(gdb) c
Continuing.
```

Figure 15 - Example breakpoint

Once the breakpoint hits, we can inspect the register `r28` to see the return value of `fcn.0000087e` is `0xFF`. We can also show the value stored at address `0x585` that the return value is compared against. An example is shown in Figure 16.

```
Breakpoint 1, 0x00000b48 in ?? ()
(gdb) info reg $r28
r28            0xff
(gdb) x/u 0x585
0x800585: 0
```

Figure 16 – Inspecting the return value of `fcn.0000087e`

Knowing register `r28` and the value stored at memory address `0x585` are different, the branch at address `0xb4e` will not be taken. The next portion of interest are the instructions at addresses `0xb50` and `0xb54` that load bytes from memory addresses `0x500` and `0x501`. These registers are then passed as arguments to the function call `fcn.00000736` at address `0xb5c` as shown in Figure 17.

```
[0x000000c4]> pd @ fcn.00000b40
/ (fcn) fcn.00000b40 94
|   fcn.00000b40 ();
|        !       ; CALL XREF from 0x00000c82 (fcn.00000bf8)
|        |    0x00000b40      cf93           push r28
|        |    0x00000b42      0e943f04       call fcn.0000087e
|        |    0x00000b46      c82f           mov r28, r24
|        |    0x00000b48      80918505       lds r24, 0x585
|        |    0x00000b4c      c817           cp r28, r24            ; Return value (key)
|       ,==< 0x00000b4e      01f1           breq 0xb90
|       ||    0x00000b50      60910005       lds r22, 0x500         ; Loading memory address
|       ||    0x00000b54      70910105       lds r23, 0x501         ; Loading memory address
|       ||    0x00000b58      8fe8           ldi r24, 0x8f
|       ||    0x00000b5a      95e0           ldi r25, 0x05
|       ||    0x00000b5c      0e949b03       call fcn.00000736
```

Figure 17 - Argument for `fcn.00000736`

The two bytes stored in memory at address `0x500` is a pointer to a string at address `0x53e` (shown in Figure 18). The string at address `0x53e` is "`Flare-On 2017 Adruino UNO Digital Pin state:`”. Recall that the ATmega328P is an 8-bit processor that requires two registers to store a pointer.

```
Breakpoint 1, 0x00000b48 in ?? ()
 (gdb) x/2bx 0x500
0x800500: 0x3e 0x05
 (gdb) x/s 0x53e
0x80053e: "Flare-On 2017 Adruino UNO Digital Pin state:"
```

Figure 18 - Identifying argument to fcn.00000736

At this point if we continue execution we will see the emulator displays the output shown in Figure 19.

```
Flare-On 2017 Adruino UNO Digital Pin state:11111111..
```

Figure 19 - Emulator output

Now we have a general idea function `fcn.00000736` is likely responsible for printing the string "`Flare-On 2017 Adruino UNO Digital Pin state:`" to the serial port, the function `fcn.0000068c` prints the digital pin state and the function `fcn.0000087e` obtains the digital pin state. This leads to the indication that `fcn.00000a6c`, which takes the digital pin state as the only argument, is the function we should focus on to solve the challenge.

```
[0x000000c4]> pd @ fcn.00000b40
/ (fcn) fcn.00000b40 94
|   fcn.00000b40 ();
|       !      ; CALL XREF from 0x00000c82 (fcn.00000bf8)
|       |   0x00000b40      cf93           push r28
|       |   0x00000b42      0e943f04       call fcn.0000087e
|       |   0x00000b46      c82f           mov r28, r24       ; Possible as return value
|       |   0x00000b48      80918505       lds r24, 0x585
|       |   0x00000b4c      c817           cp r28, r24
|      ,==< 0x00000b4e      01f1           breq 0xb90
|      ||   0x00000b50      60910005       lds r22, 0x500
|      ||   0x00000b54      70910105       lds r23, 0x501     ; Pointer to “Flare-On 2017...”
|      ||   0x00000b58      8fe8           ldi r24, 0x8f
|      ||   0x00000b5a      95e0           ldi r25, 0x05
|      ||   0x00000b5c      0e949b03       call fcn.00000736  ; print to serial
|      ||   0x00000b60      42e0           ldi r20, 0x02
|      ||   0x00000b62      50e0           ldi r21, 0x00
|      ||   0x00000b64      6c2f           mov r22, r28       ; Possible key passed as arg
|      ||   0x00000b66      8fe8           ldi r24, 0x8f
|      ||   0x00000b68      95e0           ldi r25, 0x05
|      ||   0x00000b6a      0e944603       call fcn.0000068c  ; print pin state
|      ||   0x00000b6e      64e2           ldi r22, 0x24
|      ||   0x00000b70      75e0           ldi r23, 0x05
|      ||   0x00000b72      8fe8           ldi r24, 0x8f
|      ||   0x00000b74      95e0           ldi r25, 0x05
|      ||   0x00000b76      0e943203       call fcn.00000664
|      ||   0x00000b7a      8c2f           mov r24, r28       ; Possible key passed as arg
|      ||   0x00000b7c      0e943605       call fcn.00000a6c  ; Function of interest
|      ||   0x00000b80      0197           sbiw r24, 0x01
|     ,===< 0x00000b82      21f0           breq 0xb8c
```

```
|      |||   0x00000b84      60e0           ldi r22, 0x00
|      |||   0x00000b86      8de0           ldi r24, 0x0d
|      |||   0x00000b88      0e944d02       call fcn.0000049a
|      `---> 0x00000b8c      c0938505       sts 0x585, r28
|       `--> 0x00000b90      68ee           ldi r22, 0xe8
|        |   0x00000b92      73e0           ldi r23, 0x03
|        |   0x00000b94      80e0           ldi r24, 0x00
|        |   0x00000b96      90e0           ldi r25, 0x00
|        |   0x00000b98      cf91           pop r28
\        `=< 0x00000b9a      0c94de02       jmp fcn.000005bc
 [0x000000c4]>
```

Figure 20 - Disassembly with annotations

At this point, we have a couple options to get to a solution. One option is to brute force using the debugger to set the digital pin state and the other is to do static analysis of the function shown in the above section. Since we are reverse engineers, let's assume we chose the latter and determined the key is `0xdb`. We can set another breakpoint at `0xafe` after the decoding loop to verify the results (Figure 21).

```
(gdb) info reg $r28
r28            0xff
(gdb) set $r28=0xdb
(gdb) info reg $r28
r28            0xdb
(gdb) break *(void(*)()) 0xafe
Breakpoint 2 at 0xafe
(gdb) c
Continuing.
```

Figure 21 - Setting breakpoint on decoding validation

Once the breakpoint at `0xafe` hits, we can see the instructions at addresses `0xae6` and `0xae8` are setting the `x` register to `0x56c` in Figure 22.

```
|            0x00000ae6      ace6           ldi r26, 0x6c  ; Low byte of x register
|            0x00000ae8      b5e0           ldi r27, 0x05  ; High byte of x register
|            0x00000aea      20e0           ldi r18, 0x00
|        .-> 0x00000aec      9191           ld r25, z+
|        |   0x00000aee      9827           eor r25, r24
|        |   0x00000af0      920f           add r25, r18
|        |   0x00000af2      9d93           st x+, r25
|        |   0x00000af4      2f5f           subi r18, 0xff
|        |   0x00000af6      2731           cpi r18, 0x17
|        `=< 0x00000af8      c9f7           brne 0xaec
|            0x00000afa      80917605       lds r24, 0x576
|            0x00000afe      8034           cpi r24, 0x40  ; Validate results
```

Figure 22 - Setting the `x` register for decoding loop

Now we can dump the decoded key from memory using the gdb command `x/s 0x56c` and see the key `no_r3m0rs3@flare-on.com`. as shown in Figure 23.

```
Breakpoint 4, 0x00000afe in ?? ()
(gdb) x/s 0x56c
0x80056c: "no_r3m0rs3@flare-on.com"
```

Figure 23 - Displaying key from `simavr`

Executing on Arduino UNO

This section covers how to load and solve the challenge on a genuine Arduino UNO. There are many clones of Arduino hardware and different boot loaders available that may not program correctly using these instructions. If you choose to test this challenge on an Arduino UNO, make sure you are testing on a genuine Arduino UNO with the default bootloader installed.

To program the Arduino, we can use avrdude from the `toolchain-avr`. The easiest method to obtain the `avrdude` command line arguments for your environment is to use the Arduino IDE with verbose output. To enable verbose output, under `Preferences->Settings`, check `upload` for `Show verbose output during`. Then compile and upload a simple script to your Arduino. The output window will show the `avrdude` command line used to upload the script. An example command line is shown in Figure 24.

```
avrdude -C<conf path> -v -patmega328p -carduino -P<Arduino device> -b115200 -D -
Uflash:w:remorse_09.ino.hex:i
```

Figure 24 - Example avrdude to program Arduino UNO

Once the challenge is uploaded and running on the Arduino UNO, we can now use the serial monitor from the Arduino IDE to see the output shown in Figure 25 displayed by the challenge.

```
Flare-On 2017 Adruino UNO Digital Pin state:11111111
```

Figure 25 - Serial output from Arduino UNO

After connecting the digital pins two and five to ground, as shown in Figure 26, we can see the solution output in the serial monitor shown in Figure 27.

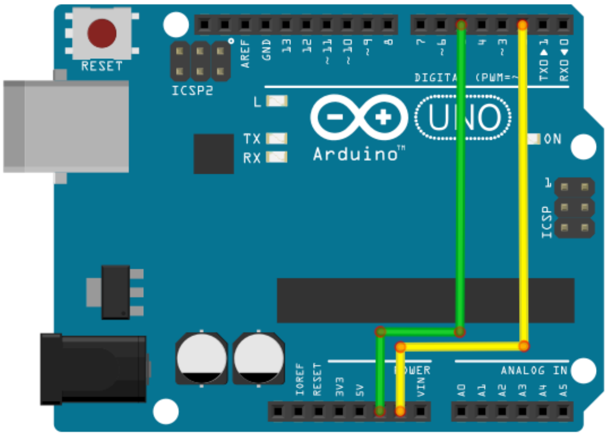


Figure 26 - Arduino UNO solution wiring diagram

```
Flare-On 2017 Adruino UNO Digital Pin state:11111111
Flare-On 2017 Adruino UNO Digital Pin state:11011111
Flare-On 2017 Adruino UNO Digital Pin state:11011011
Correct Pin State:
.. .-- ..--.- .-. ...-- .- .---- .-. ... ...-- .--.-. ..-. .-.. .- .-. . .....- .--
.. .-.-.- ..-. .-- .-
no_r3m0rs3@flare-on.com
```

Figure 27 - Arduino UNO solution serial monitor output