Practical Firmware Reversing 
and Exploit Development for 
AVR-based Embedded Devices

Alexander @dark_k3y Bolshev
Boris @dukeBarman Ryutin
Agenda

Part 1: Quick **RJMP** to AVR + Introduction example
Part 2: Pre-exploitation
Part 3: Exploitation and ROP-chains building
Part 4: Post-exploitation and tricks
Thus: If you have a question, please interrupt and ask immediately.

Disclaimer:

1) Training is **VERY** fast-paced
2) Training is highly-practical
3) You may encounter information overflow
4) My English is far from perfect
Part 1: What is AVR?
AVR

- Alf (Egil Bogen) and Vegard (Wollan)’s RISC processor
- Modified Harvard architecture 8-bit RISC single-chip microcontroller
- Developed by Atmel in 1996 (now Dialog/Atmel)
AVR is almost everywhere

- Industrial PLCs and gateways
- Home electronics: kettles, irons, weather stations, etc
- IoT
- HID devices (ex.: Xbox hand controllers)
- Automotive applications: security, safety, powertrain and entertainment systems.
- Radio applications (and also Xbee and Zwave)
- Arduino platform
- WirelessHART transmitters and sensors
- Your new shiny IoE fridge ;-)

AVR inside industrial gateway
Synapse IoT module with Atmega128RFA1 inside
Philips Hue Bulb

AVR inside home automation dimmer
Harvard Architecture
Harvard Architecture

• Physically separated storage and signal pathways for instructions and data
• Originated from the Harvard Mark I relay-based computer
Modified Harvard architecture...

...allows the contents of the instruction memory to be accessed as if it were data\(^1\) but not the data as code!
Introduction example:
We’re still able to exploit!
AVR “features”
AVR-8

• MCU (MicroController Unit) -- single computer chip designed for embedded applications
• Low-power
• Integrated RAM and ROM (SRAM + EEPROM + Flash)
• Some models could work with external SRAM
• 8-bit, word size is 16 bit (2 bytes)
• Higher integration
• Single core/Interrupts
• Low-freq (<20MHz in most cases)
Higher Integration

- Built-in SRAM, EEPROM an Flash
- GPIO (discrete I/O pins)
- UART(s)
- I²C, SPI, CAN, ...
- ADC
- PWM or DAC
- Timers
- Watchdog
- Clock generator and divider(s)
- Comparator(s)
- In-circuit programming and debugging support
AVRs are very different

• AtTiny13
• Up to 20 MIPS Throughput at 20 MHz
• 64 SRAM/64 EEPROM/1k Flash
• Timer, ADC, 2 PWMs, Comparator, internal oscillator
• 0.24mA in active mode, 0.0001mA in sleep mode
AVRs are very different

- Atmega32U4
- 2.5k SRAM/1k EEPROM/32k Flash
- JTAG
- USB
- PLL, Timers, PWMs, Comparators, ADCs, UARTs, Temperatures sensors, SPI, I²C, ... => tons of stuff
AVRs are very different

- Atmega128
- 4k SRAM/4k EEPROM/128k Flash
- JTAG
- Tons of stuff...

The workshop focuses on this chip
Why Atmega128?

• Old, but very widespread chip
• At90can128 – popular analogue for CAN buses in automotive application
• **Cheap JTAG programmer**
• Much SRAM == ideal for ROP-chain construction training

Let’s look to the architecture of Atmega128...
OH SHIT
Ok, ok, let’s simplify a bit 😊
Note: code is **separated** from data
Memory map

Figure 9. Data Memory Map
Memory: registers

• R0-R25 – GPR
• X,Y,Z – pair “working” registers, e.g. for memory addressing operations
• I/O registers – for accessing different “hardware”
Memory: special registers

- **PC** – program counter, 16-bit register
- **SP** – stack pointer, 16-bit register (SPH:SPL)
- **SREG** – status register (8-bit)
Memory addressing

• SRAM/EEPROM – 16-bit addressing, 8-bit element

• Flash – 16(8)-bit addressing, 16-bit element
Memory addressing directions

• Direct to register
• Direct to I/O
• SRAM **direct**
• SRAM **indirect** (pre- and post- increment)
• Flash direct
Datasheets are your best friends!
Interrupts

- Interrupts interrupt normal process of code execution for handling something or reacting to some event.
- Interrupt handler is a procedure to be executed after interrupt; address stored in the interrupt vector.
- Examples of interrupts:
  - Timers
  - Hardware events
  - Reset
AVR assembly
Instruction types

- Arithmetic and logic
- Bit manipulation/test
- Memory manipulation
- Unconditional jump/call
- Branch commands
- SREG manipulation
- Special (watchdog, etc)
Instruction mnemonics

```assembly
mov    r16, r0 ; Copy r0 to r16
out    PORTA, r16 ; Write r16 to PORTA
```

16-bit long
“Intel syntax” (destination before source)
A bit more about architecture
Fuses and Lock Bits

• Several bytes of permanent storage
• Set internal hardware and features configuration, including oscillator (int or ext), bootloader, pins, ability to debug/program, etc.
• 2 lock bits controls programming protection.
AVR bootloader – what is it?

• Part of code that starts **BEFORE** RESET interrupt.
• Could be used for self-programmable (i.e. without external device) systems, in case you need update the firmware of your IoT device.
• Bootloader address and behavior configured via FUSEs.
• BLB lock bits controls bootloader ability to update application and/or bootloader parts of flash.
AVR bootloaders

• Arduino bootloader
• USB bootloaders (AVRUSBBoot)
• Serial programmer bootloaders (STK500-compatible)
• Cryptobootloaders
• ...
• Tons of them!
Watchdog

• Timer that could be used to interrupt or reset device.
• Cleared with **WDR** instruction.
Development for AVR
Atmel studio
AVR-GCC

• Main compiler/debugger kit for the platform
• Used by Atmel studio
• Use “AVR libc” -- http://www.nongnu.org/avr-libc/
• Several optimization options, several memory models
Other tools

- Arduino
- CodeVision AVR
- IAR Embedded workbench
Debugging AVR
JTAG

• Joint Test Action Group (JTAG)
• Special debugging interface added to a chip
• Allows testing, debugging, firmware manipulation and boundary scanning.
• Requires external hardware
JTAG for AVRs

AVR JTAG mkI

AVR JTAG mkII

AVR Dragon

AVR JTAGIce3

Atmel ICE3
Avarice

• Open-source interface between AVR JTAG and GDB
• Also allow to flash/write EEPROM, manipulate fuse and lock bits.
• Could *capture* the execution flow to restore the firmware
• Example usage:

  avarice --program --file test.elf --part atmega128 --jtag /dev/ttyUSB0 -d :4242
AVR-GDB

- Part of “nongnu” AVR gcc kit.
- Roughly ported standard gdb to AVR platform
- Doesn’t understand Harvard architecture
  - You will need to resolve memory address by reference of $pc to read the flash

(gdb) x/10b $pc + 100
Simulators

• Atmel Studio simulator
• Proteus simulator
• Simavr
• Simulavr
Training kit content

- Arduino
- AVR JTAG mkI
- Atmega128 custom devboard
- ESP8266 “WiFi to serial”
VM access:

Login: radare
Password: radare
Ex 1.1: Hello world!

cd /home/radare/workshop/ex1.1
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file hello.hex

Communication: CuteCom or Cscreen /dev/ttyUSB1 9600

For debugging:
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file hello.hex -d :4242

In new terminal window:
avr-gdb
(gdb) target remote :4242
Ex 1.1_simulator: Hello world!

Simulator

cd /home/radare/workshop/ex1.1_simulator
simulavr -d atmega128 -f hello.elf -F 16000000 -x -,E1,9600 -y -,E0,9600

For debugging:
simulavr -d atmega128 -f hello.elf -F 16000000 -x -,E1,9600 -y -,E0,9600 -g
avr-gdb
(gdb) target remote :1212
Ex 1.2: Blink!

```
cd /home/radare/workshop/ex1.2
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file blink.hex

For debugging:
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file blink.hex -d :4242
avr-gdb
(gdb) target remote :4242
```
Part 2: Pre-exploitation
You have a device. First steps?

- Decide what you want
- Determine target platform
- Search for I/O point(s)
- Search for debug point(s)
- Acquire the firmware
- Fuzz and/or static analysis
Let’s start with a REAL example

• Let’s use training kit board as an example
• Imagine that you know nothing about it
• We will go through all steps, one by one
What we want?

To start with, decide what you want:
• Abuse of functionality
• Read something from EEPROM/Flash/SRAM
• Stay persistent
Determine target platform

- Look at the board and search for all ICs...

ICs: Atmega128 16AU, CP2102, ESP8266EX
Digikey/Octopart/Google...
Search for I/O(s)

- USB
- UART
- Antenna

External connectors
Search for I/O(s): tools

- Bus pirate
- Saleae logic analyzer
- Jtagulator
- Arduino
Search for debug interface(s)
Search for debug interface(s): tools

Jtagulator

Arduino + JTAGEnum

Or cheaper
JTAGEnum against Atmega128 demoboard

- Connect Arduino to Atmega128 demoboard
- Connect Arduino to PC with USB cable

```
cd ~/workshop/JTAGEnum
make upload  (click reset on arduino just before it)
screen /dev/ttyACM0 115200
```

- Press “s”
Search for debug & I/O: real device

- Connector
- Ethernet
- Button
- LEDs
- 2 JTAGs
- ICS bus
- ISPs
Acquire the firmware

• From vendor web-site 😊
• Sniffing the firmware update session
• From device itself
Acquiring the firmware: sniff it!
Acquiring the firmware: JTAG or ISP

• Use JTAG or ISP programmer to connect to the board debug ports

• Use:
  – Atmel Studio
  – AVRDude
  – Programmer-specific software to read flash

  $ avrdude -p m128 -c jtag1 -P /dev/ttyUSB0 \
      -U flash:r:"/home/avr/flash.bin":r
Acquiring the firmware: lock bits

• AVR has lock bits that protect device from extracting flash

<table>
<thead>
<tr>
<th>Mode</th>
<th>LB1</th>
<th>LB2</th>
<th>Protection Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Unprogrammed, no protection enabled</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>Further Programming disabled, Read back possible</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Further programming and read back is disabled</td>
</tr>
</tbody>
</table>

• Clearing these lockbits will erase the entire device
• If you have them set you’re not lucky --> try to get firmware from other sources
• However, if you have lock bits set but JTAG is enabled you could try partial restoration of firmware with avarice –capture (rare case)
Exercise 2.0: Fuses

Read fuses and lock bits using

`
avarice --mkI --jtag /dev/ttyUSB0 -r -l
`
Firmware reversing: formats

• Raw binary format
• ELF format for AVRs
• Intel HEX format (often used by programmers)

• Could be easily converted between with avr-objcopy, e.g.:
  
  ```
  avr-objcopy -I ihex -O binary blink.hex blink.bin
  ```
AVR RE
Reverse engineering AVR binaries

Pure disassemblers:
• avr-objdump – gcc kit standard tool
• Vavrdisasm -- https://github.com/vsergeev/vavrdisasm
• ODAweb -- https://www.onlinedisassembler.com/odaweb/

“Normal” disassemblers:
• IDA Pro
• Radare
IDA PRO: AVR specifics

- Incorrect AVR elf-handling
- Incorrect LPM command behavior
- Addressing issues
- Sometimes strange output
- Still usable, but “with care”
Radare2

- Opensource reverse engineering framework (RE, debugger, forensics)
- Crossplatform (Linux, Mac, Windows, QNX, Android, iOS, ...)
- Scripting
- A lot of architectures / file-formats
- ...
- Without “habitual” GUI (c) pancake
Radare2: Tools

• radare2
• rabin2
• radiff2
• rafind2
• rasm2
• r2pm

• rarun2
• rax2
• r2agent
• ragg2
• rahash2
• rassign2
Radare2: Usage

• Install from git
  # git clone https://github.com/radare/radare2
  # cd radare2
  # sys/install.sh

• Packages (yara, retdec / radeco decompilers, ...):
  # r2pm -i radare2

• Console commands
  # r2 -d /bin/ls – debugging
  # r2 –a avr sample.bin – architecture
  # r2 –b 16 sample.bin – specify register size in bits
  # r2 sample.bin –i script – include script
Radare2: Basic commands

- aaa – analyze
- axt – xrefs
- s – seek
- p – disassemble
- ~ – grep
- ! – run shell commands
- / – search
- /R – search ROP
- /c – search instruction
- ? – help
Radare2: Disassembling

• p?
• pd/pD - disassemble
• pi/pl – print instructions
• Examples:
  > pd 35 @ function
Radare2: Options

- ~/.radarerc
- e asm.describe=true
- e scr.utf8=true
- e asm.midflags=true
- e asm.emu=true
- eco solarized
Radare2: Interfaces

- ASCII – VV
- Visual panels – V! (vim like controls)
- Web-server – r2 -c=H file
- Bokken
Best combinations for AVR RE

• Both Radare2 and IDA Pro have pitfalls when working with AVR
• That’s why I am using the following combination

    IDA Pro 6.6+ + Radare2 + GDB + avr-objdump

*Here we will focus on Radare2 + GDB, because not everyone can afford latest IDA Pro 😞*
Ex 2.1: Hello! RE

cd /home/radare/workshop/ex2.1
avr-objcopy -I ihex -O binary hello.hex hello.bin
r2 -a avr hello.bin
Now we will scrutinize every line of disassembled code. Boring, but is required for further understanding
Interrupts vector && init section

Program halt
Memory manipulation: stack push

```
push    r14     ; save r14 on the Stack
```

```
SP = SP - 1
```

```
r14 = 0x10
```

```
                   Some value
                   Some value

                   Some value
                   Some value

0x10
```
Memory manipulation: stack pop

push r14 ; save r14 on the Stack

SP = SP - 1

r14 = 0x10

pop r15 ; pop top of Stack to r15

SP = SP + 1

r15 ← 0x10
Unconditional jump/call

jmp 0xABC1 ; PC = 0xABC1
rjmp 5 ; PC = PC + 5 + 1
call 0xABC1 ; “push PC+2”
            ; jmp 0xABC1
ret ; “pop PC”
Harvard architecture? But PC goes to **DATA** memory

**STORE PC ON STACK?**

**ROP IS POSSIBLE!**
Arithmetic instructions

add    r1, r2    ; r1 = r1 + r2
add    r28, r28 ; r28 = r28 + r28
and    r2, r3   ; r2 = r2 & r3
clr    r1       ; r1 = 0
ser    r28      ; r28 = 0xFF
inc    r0       ; r0 = r0 + 1
neg    r0       ; r0 = -r0
Memory manipulation: immediate values

```
ldi r29, 0x10 ; r29 = 0x10
```
Memory manipulation: ports

in        r15, $16 ; r15 = PORTB
out       $16, r0 ; PORTB = r0

What is the 0x3f, 0x3e, 0x3d and where to find them?
Datasheets are your best friends! (2)

P. 366 of Atmega128L datasheet

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(61)$</td>
<td>DORF</td>
<td>DDF7</td>
<td>DDF6</td>
<td>DDF5</td>
<td>DDF4</td>
<td>DDF3</td>
<td>DDF2</td>
<td>DDF1</td>
<td>DDF0</td>
<td>88</td>
</tr>
<tr>
<td>$(60)$</td>
<td>Reserved</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>11</td>
</tr>
<tr>
<td>$3F $(S5F)$</td>
<td>SREG</td>
<td>I</td>
<td>T</td>
<td>H</td>
<td>S</td>
<td>V</td>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>14</td>
</tr>
<tr>
<td>$3E $(S5E)$</td>
<td>SPH</td>
<td>SP15</td>
<td>SP14</td>
<td>SP13</td>
<td>SP12</td>
<td>SP11</td>
<td>SP10</td>
<td>SP9</td>
<td>SP8</td>
<td>14</td>
</tr>
<tr>
<td>$3D $(S5D)$</td>
<td>SPL</td>
<td>SP7</td>
<td>SP6</td>
<td>SP5</td>
<td>SP4</td>
<td>SP3</td>
<td>SP2</td>
<td>SP1</td>
<td>SP0</td>
<td>14</td>
</tr>
<tr>
<td>$3C $(S5C)$</td>
<td>XDIV</td>
<td>XDIVEN</td>
<td>XDIV6</td>
<td>XDIV5</td>
<td>XDIV4</td>
<td>XDIV3</td>
<td>XDIV2</td>
<td>XDIV1</td>
<td>XDIV0</td>
<td>43</td>
</tr>
<tr>
<td>$3B $(S5B)$</td>
<td>RAMPZ</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>14</td>
</tr>
<tr>
<td>$3A $(S5A)$</td>
<td>EICRB</td>
<td>ISC71</td>
<td>ISC70</td>
<td>ISC61</td>
<td>ISC60</td>
<td>ISC51</td>
<td>ISC50</td>
<td>ISC41</td>
<td>ISC40</td>
<td>90</td>
</tr>
<tr>
<td>$39 $(S59)$</td>
<td>EIMSK</td>
<td>INT7</td>
<td>INT6</td>
<td>INT5</td>
<td>INT4</td>
<td>INT3</td>
<td>INT2</td>
<td>INT1</td>
<td>INT0</td>
<td>91</td>
</tr>
<tr>
<td>$38 $(S58)$</td>
<td>EIFR</td>
<td>INTF7</td>
<td>INTF6</td>
<td>INTF5</td>
<td>INTF4</td>
<td>INTF3</td>
<td>INTF2</td>
<td>INTF1</td>
<td>INTF0</td>
<td>91</td>
</tr>
<tr>
<td>$37 $(S57)$</td>
<td>TIMSK</td>
<td>OCIE2</td>
<td>TOIE2</td>
<td>TIE1</td>
<td>OCIE1A</td>
<td>OCIE1B</td>
<td>TOIE1</td>
<td>OCIE0</td>
<td>TOIE0</td>
<td>108, 140, 160</td>
</tr>
<tr>
<td>$36 $(S56)$</td>
<td>TIFR</td>
<td>OCF2</td>
<td>TOV2</td>
<td>TCF1</td>
<td>OCF1A</td>
<td>OCF1B</td>
<td>TOV1</td>
<td>OCF0</td>
<td>TOV0</td>
<td>108, 141, 161</td>
</tr>
<tr>
<td>$35 $(S55)$</td>
<td>MCUCR</td>
<td>SRE</td>
<td>SRW10</td>
<td>SE</td>
<td>SM1</td>
<td>SM0</td>
<td>IVSEL</td>
<td>IVCE</td>
<td>–</td>
<td>31, 44, 63</td>
</tr>
<tr>
<td>$34 $(S54)$</td>
<td>MCUCSR</td>
<td>JTD</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>JTRF</td>
<td>WDRF</td>
<td>BORF</td>
<td>EXTRF</td>
<td>PORF</td>
</tr>
<tr>
<td>$33 $(S53)$</td>
<td>TCCTR0</td>
<td>FOC0</td>
<td>WGM00</td>
<td>COM01</td>
<td>COM00</td>
<td>WGM01</td>
<td>CS02</td>
<td>CS01</td>
<td>CS00</td>
<td>103</td>
</tr>
<tr>
<td>$32 $(S52)$</td>
<td>TCNT0</td>
<td>Timer/Counter0 (8 Bit)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>105</td>
</tr>
</tbody>
</table>
So, what’s going on here?

1. SREG (Status REGister) is cleared (set to r1 value, which is 0x00)
2. SPL ← r28 (0xFF)
3. SPH ← r29 (0x10)

After init:
SREG = 0, SP = 0x10FF = 4351(SRAM limit)
Going further
Memory manipulation: lds/sts

```assembly
lds       r2,0xFA00 ; r2 = *0xFA00
sts       0xFA00,r0 ; *0xFA00 = r0
```

Here:

1. \(0x00\) \(\rightarrow\) r1 (0x00)
2. r24 \(\rightarrow\) 0x67
3. \(0x67\) \(\rightarrow\) r24 (0x67)
Datasheets are your best friends! (3)

P. 365 of Atmega128L datasheet

### Register Summary

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>($FF)</td>
<td>Reserved</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>..</td>
<td>Reserved</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>($9E)</td>
<td>Reserved</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>($9D)</td>
<td>UCSR1C</td>
<td>–</td>
<td>UMSEL1</td>
<td>UPM11</td>
<td>UPM10</td>
<td>USBS1</td>
<td>UCSZ11</td>
<td>UCSZ10</td>
<td>UCPOL1</td>
<td>192</td>
</tr>
<tr>
<td>($9C)</td>
<td>UDR1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>192</td>
</tr>
<tr>
<td>($9B)</td>
<td>UCSR1A</td>
<td>RXC1</td>
<td>TXC1</td>
<td>UDRE1</td>
<td>FE1</td>
<td>DOR1</td>
<td>UPE1</td>
<td>U2X1</td>
<td>MPCM1</td>
<td>190</td>
</tr>
<tr>
<td>($9A)</td>
<td>UCSR1B</td>
<td>RXIE1</td>
<td>TXCIE1</td>
<td>UDRIE1</td>
<td>RXEN1</td>
<td>TXEN1</td>
<td>UCSZ12</td>
<td>RXB81</td>
<td>TXB81</td>
<td>191</td>
</tr>
<tr>
<td>($99)</td>
<td>UBRR1L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>194</td>
</tr>
<tr>
<td>($98)</td>
<td>UBRR1H</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td>194</td>
</tr>
</tbody>
</table>
What is it all about and why \texttt{sts} and not \texttt{out}?

\begin{verbatim}
UBRR1H = (BAUD_PRESCALE >> 8);
    10929800  sts     0x0098, r1
UBRR1L = (BAUD_PRESCALE);
    87e6      ldi     r24, 0x67
    80939900  sts     0x0099, r24
\end{verbatim}

- Registers are also part of the RAM
- Common rule:
  - Every IO/RAM address is reachable with \texttt{sts/ldi}s while \texttt{in/out} are used for (0x00 - 0x3F range)
Why 0x0067?

• USART is clocking from internal generator (16MHz in our case)
• We selected baud speed of 9600
• The common formula of USART frequency divider for AVR (see datasheet for USART section, p.194+):

\[
 \text{BAUD\_PRESCALE} = \left( \frac{F\_\text{CPU}}{\text{USART\_BAUDRATE} \times 16} \right) - 1 = 16\,000\,000 \div (9600 \times 16) - 1 = 104.1666666(6) - 1 \approx 103 = 0x0067
\]
More arithmetic instructions

\texttt{andi r2, 0x10 \quad ; r2 = r2 \& 0x10}

\texttt{ori r24, 0x98 \quad ; r24 = r24 | 0x98}

\texttt{...}

\texttt{UCSR1B | = (1 << RXEN1) | (1 << TXEN1) | (1 << RXCIE1)};

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000bc</td>
<td>80919a00</td>
<td>lds r24, 0x9a</td>
</tr>
<tr>
<td>0x000000c0</td>
<td>8869</td>
<td>ori r24, 0x98</td>
</tr>
<tr>
<td>0x000000c2</td>
<td>80939a00</td>
<td>sts 0x9a, r24</td>
</tr>
</tbody>
</table>

Enables RX and TX lines, enable RX interrupt.

\texttt{UCSR1C | = (1<<USBS1) | (3<<UCSZ10)};

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000c6</td>
<td>80919d00</td>
<td>lds r24, 0x9d</td>
</tr>
<tr>
<td>0x000000ca</td>
<td>8e60</td>
<td>ori r24, 0x0e</td>
</tr>
<tr>
<td>0x000000cc</td>
<td>80939d00</td>
<td>sts 0x9d, r24</td>
</tr>
</tbody>
</table>

Set stop bit and character size
Functions & Calling conventions

send_byte('H');

Typical AVR calling convention for arguments

- Call-used: **R18–R27, R30, R31**
- Call-saved: **R2–R17, R28, R29**
- **R29:R28** used as frame pointer

We will discuss it in more details later.
_delay_ms(1000);

This “function” is inlined as:

```
subi r18, 0x01 ; r18 = r18 - 1
sbci r24, 0x00 ; r24 = r24 - 0 - C

; C – Carry flag from arithmetic operations (SREG)
```
Conditional jump

cpse    r1, r0 ; r1 == r2 ?
        \[ \text{PC} \leftarrow \text{PC} + 2 : \text{PC} \leftarrow \text{PC} + 3 \]

breq   10 ; Z ? PC \leftarrow PC + 1 + 10

brne   -4 ; !Z ? PC \leftarrow PC + 1 - 4

...
Why -4?

Operation:
(i) If Rd \neq Rr (Z = 0) then \text{PC} \leftarrow \text{PC} + k + 1, else \text{PC} \leftarrow \text{PC} + 1

Syntax:
(i) BRNE \ k

Operands:
-64 \leq k \leq +63

Program Counter:
\text{PC} \leftarrow \text{PC} + k + 1
\text{PC} \leftarrow \text{PC} + 1, if condition is false

16-bit Opcode:

| 1111 | 01kk | kkkk | k001 |

f7e1 = 1111 0111 1110 0001
11 1110 0 in two’s complement form == -4
Special

- `break` – debugger break
- `nop` – no operation
- `sleep` – enter sleep mode
- `wdr` – watchdog timer reset
void send_byte(uint8_t byte)

while((UCSR1A &(1<<UDRE1)) == 0);

UDR1 = byte;
Conditional “skip”

sbrc r0, 7 ; skip if bit 7 in r0 cleared

cpse r4,r0 ; skip if r4 == r0

sbrs r25,5 ; skip if bit 5 in r25 set
More things to know
Comparison

cp r4,r19 ; Compare r4 with r19
brne label1 ; jump if r19 != r4

; Compare r3:r2 with r1:r0

cp r2,r0 ; Compare low byte
cpc r3,r1 ; Compare high byte
brne label2 ; jump if r3:r2 != r1:r0


cpi r19,3 ; Compare r19 with 3
brne label3 ; jump if r19 != 3
SREG – 8-bit status register

C – Carry flag
Z – Zero flag
N – Negative flag
V – two’s complement overflow indicator
S – \( N \oplus V \), for Signed tests
H – Half carry flag
T – Transfer bit (BLD/BST)
I – global Interrupt enable/disable flag
SREG manipulations

- `sec/clc` – set/clear carry
- `sei/cli` – set/clear global interruption flag
- `se*/cl*` – set/clear * flag in SREG
More memory manipulation

```
mov    r1, r2        ; r1 = r2
st     Z, r0        ; *Z(r31:r30) = r0
st     -Z, r1       ; *Z-- = r1
std    Z+5, r2      ; *(Z+5) = r2
...
```

Direct

Indirect

Same
for LD*
Memory manipulation: flash

lpm r16, Z ; r16 = *(r31:r30), but from flash

Note: code is separated from data
Bit manipulation instructions

sbr r16, 3 ; set bits 0 and 1 in r16
lsl r0 ; r0 << 2
lsr r1 ; r1 >> 2
rol r15 ; cyclic shift r16 bits to the left
ror r16 ; cyclic shift r16 bits to the right
cbr r18, 1 ; clear bit 1 in r18
cbi $16, 1 ; PORTB[1] = 0
Ex 2.2: Blink! RE

cd /home/radare/workshop/ex2.2
avr-objcopy -I ihex -O binary blink.hex blink.bin
r2 -a avr blink.bin

Questions:
1. Identify main() function, define and rename it
2. Find the LED switching command
3. What type of delay is used and why accuracy of MCU frequency important?
4. Locate interrupt vector and init code, explain what happens inside init code
Reversing: function szignatures

- Majority of firmware contains zero or little strings.
- How to start?
- Use function signatures.
- However, in AVR world signatures may be to vary.
- Be prepared to guess target compiler/library/RTOS and options... or bruteforce it.
- In R2, signatures are called zignatures.
Working with zignatures
Embedded code priorities

• Size
• Speed
• Hardware limits
• Redundancy
• ...
• ...
• ...
• ...
• ...
• ...
• Security
Fuzzing specifics

• Fuzzing is a fuzzing. Everywhere.
• But... we’re in embedded world
• Sometimes you **can** detect crash through test/debug UART or pins
• In most cases, you can detect crash only by noticing that device is no longer response
• Moreover, **watchdog timer** can limit your detection capabilities by resetting the device
• So how to detect crash?
Fuzzing: ways to detect crash

• JTAG debugger – break on RESET
• External analysis of functionality – detect execution pauses
• Detect bootloader/initialization code (e.g. for SRAM) behavior with logic analyzer and/or FPGA
• Detect power consumption change with oscilloscope/DAQ
Sometimes Arduino is enough to detect

- \textsuperscript{1}\textsuperscript{2}C and SPI init sequences could be captured by Arduino GPIOs
- In case bootloader is slow and has \(\sim\)1 second loading delay, this power consumption reduction could be reliably detected with cheap current sensor, e.g.:

SparkFun Low Current Sensor Breakout - ACS712
https://www.sparkfun.com/products/8883
Let’s proof it.
Part 3: Exploitation
Quick intro to ROP-chains

• **Return Oriented Programming**
• Series of function returns
• We are searching for primitives ("gadgets") ending with `ret` that could be chained into a useful code sequence
• SP is our new PC
Notice: Arduino

• The next examples/exercises will be based upon Arduino ‘libc’ (in fact, Non-GNU AVR libc + Arduino wiring libs)

• We’re using Arduino because it is sufficiently complex, full of gadgets and free (vs. IAR or CV which are also complex and full of gadgets)

• Also, Arduino is fairly popular today due to enormous number of libraries and “quick start” (and quick bugs)
Ex 3.1 – 3.3

cd /home/radare/workshop/ex3.1
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file build-crumbuino128/ex3.1.hex -d :4242

In the new terminal window:
avr-gdb
(gdb) target remote :4242
Example 3.1
Abusing functionality: ret to function
Internal-SRAM only memory map

Overflowing the heap => Rewriting the stack!

How to connect data(string/binary) to code?

Standard model: with .data variables
- Determine data offset in flash
- Find init code/firmware prologue where .data is copied to SRAM
- Using debugging or own brain calculate offset of data in SRAM
- Search code for this address

Economy model: direct read with lpm/elpm
- Determine data offset in flash
- Search code with *lpm addressing to this offset
ABI, Types and frame layouts (GCC)

• Types: standard (short == int == 2, long == 4, except for double (4))
• Int could be 8bit if -mint8 option is enforced
• Call-used: R18–R27, R30, R31
• Call-saved: R2–R17, R28, R29
• R29:R28 used as frame pointer
• Frame layout after function prologue:

<table>
<thead>
<tr>
<th>incoming arguments</th>
<th>return address</th>
<th>saved registers</th>
<th>stack slots, Y+1 points at the bottom</th>
</tr>
</thead>
</table>
Calling convention: arguments

• An argument is passed either completely in registers or completely in memory

• To find the register where a function argument is passed, initialize the register number $R_n$ with R26 and follow the procedure:
  1. If the argument size is an odd number of bytes, round up the size to the next even number.
  2. Subtract the rounded number from the register number $R_n$.
  3. If the new $R_n$ is at least R18 and the size of the object is non-zero, then the low-byte of the argument is passed in $R_n$. Other bytes will be passed in $R_{n+1}$, $R_{n+2}$, etc.
  4. If the new register number $R_n$ is smaller than R18 or the size of the argument is zero, the argument will be passed in memory.
  5. If the current argument is passed in memory, stop the procedure: All subsequent arguments will also be passed in memory.
  6. If there are arguments left, goto 1. and proceed with the next argument.

• Varargs are passed on the stack
Calling conventions: returns

• Return values of size 1 byte up to 8 bytes (including) will be returned in registers
• For example, an 8-bit value is returned in R24 and an 32-bit value is returned R22...R25
• Return values whose size is outside that range will be returned in memory
Example

For

```c
int func (char a, long b);
```

- a will be passed in R24
- b will be passed in R20, R21, R22 and R23 with the LSB in R20 and the MSB in R23
- The result is returned in R24 (LSB) and R25 (MSB)
Example 3.2
Abusing functionality: simple ROP
ROP gadget sources

• User functions
• “Standard” or RTOS functions
• Data segment 😊
• Bootloader section

More code => more gadgets
ROP chain size

- It is MCU
- SRAM is small
- SRAM is divided between register file, heap and stack
- Stack size is small
- We are limited in chain size
- Obviously, you will be constrained to 20-40 bytes (~15-30 gadgets)
- However it all depends on compiler and memory model

Memory maps – external SRAM/separated stack
Memory maps – external SRAM/mixed stack
Detecting “standard” functions

• In AVR world there are a lot of different compilers, libraries and even RToSes
• Thus, “standard” function could vary
• More bad news: memory model and optimization options can change function
• The best approach is to try to detect functions like malloc/str(n)cpy and then find the exact compiler/options that generates this code
• After that, use function signatures to restore the rest of the code
• In Radare2, you could use zignatures or Yara
Example 3.3
More complex ROP
Exercise 3.1
ret 2 function

Build exploit that starts with ABC but calls switchgreen() function
Exercise 3.3
Print something else

3.3.1 Build exploit that prints “a few seconds…”
3.3.2 (homework) Build exploit that prints “blink a few seconds…”
Ex 3.4

cd /home/radare/workshop/ex3.1

• In Blink.ino change APNAME constant from “esp_123” to “esp_<your3digitnumber>”
make
avr-objdump -I ihex -O binary build-crumbuino128/ex3.4.hex ex3.4.bin
avarice --mkI --jtag /dev/ttyUSB0 -p -e --file build-crumbuino128/ex3.4.hex -g :4242
avr-gdb

• Connect to WiFi “esp_<your3digitnumber>” (password: 1234567890) and type http://192.168.4.1 in your browser
Example 3.4
Blink using HTTP GET
Exercise 3.4
UARTing using HTTP query
Exercise 3.5
Blink using HTTP Post
(homework)
It is possible to construct ROP with a debugger...
...But if you don’t have one, how could you determine the overflow point?

• Reverse firmware and use an external analysis to find function that overflows
• Bruteforce it!
Arduino blink (ROP without debugger)

- Connect Arduino board using MicroUSB cable

```bash
cd /home/radare/workshop/ex_arduino
make upload  # (click reset on arduino just before it)
```

- Run cutecom and connect to /dev/ttyACM0 using speed 9600
Arduino blink
(ROP without debugger)

Modify ROP chain to generate another blinking pattern
Part 4: Post-exploitation && Tricks
What do we want? (again)

• Evade watchdog
• Work with persistent memory (EEPROM and Flash)
• Stay persistent in device
• Control device over long time
Evade the watchdog

In most cases, there three ways:

1. Find a ROP with **WDR** and periodically jump on it
2. Find watchdog disabling code and try to jump on it
3. Construct watchdog disabling code using watchdog enabling code

```
set r18 to 0 and JMP here
```
Fun and scary things to do with memory...

- Read/write EEPROM (and extract cryptographic keys)
- Read parts of flash (e.g., read locked bootloader section)
  - Could be more useful than it seems
- Staying persistent (writing flash)
Reading EEPROM/Flash

• In most cases it is easy to find gadget(s) that reads byte from EEPROM or flash and stores it somewhere
• We could send this byte back over UART or any external channel gadgets
• Not always possible, but there are good chances
Writing flash

• Writing flash is locked during normal program execution

• However, if you use “jump-to-bootloader” trick, you could write flash from bootloader sections

• To do this, you need bootloader which has enough gadgets

• Modern bootloaders are large and you may be lucky quite often (e.g. Arduino bootloader)

• Remember to disable interrupts before jumping to bootloader
“Infinite-ROP” trick*

1. Set array to some “upper” stack address (A1) and N to some value (128/256/etc) and JMP to read(..)
2. Output ROP-chain from UART to A1.
3. Set SPH/SPL to A1 (gadgets could be got from init code)
4. JMP to RET.
5. ???
6. Profit!

Don’t forget to include 1 and 3-4 gadgets in the ROP-chain that you are sending by UART.

*Possible on firmwares with read(array, N) from UART functions and complex init code
Mitigations
Mitigations (software)

• Safe coding/Don’t trust external data (read 24 deadly sins of computer security)
• Reduce code size (less code -> less ROP gadgets)
• Use \texttt{rjmp/jmp} instead of \texttt{call/ret} (but it won’t save you from ret2 function)
• Use “inconvenient” memory models with small stack
• Use stack canaries in your RTOS
• Limit external libraries
• Use watchdogs
• Periodically check stack limits (to avoid stack expansion tricks)
Mitigations (hardware)

- Disable JTAG/debuggers/etc, remove pins/wires of JTAG/ISP/UART
- Write lock bits to 0/0
- Use multilayered PCBs
- Use external/hardware watchdogs
- Use modern MCUs (more secure against various hardware attacks)
- Use external safety controls/processors

And last, but not least:
- Beware of Dmitry Nedospasov ;-)

Conclusions

• RCE on embedded systems isn’t so hard as it seems.
• Abuse of functionality is the main consequence of such attacks
• However, more scary things like extracting cipherkeys or rewriting the flash are possible
• When developing embedded system remember that security also should be part of the software DLC process
Books/links

Radare2. Links

- http://radare.org
- https://github.com/pwntester/cheatsheets/blob/master/radare2.md
- https://github.com/radare/radare2ida
Any Questions?

@dark_k3y
@dukeBarman

http://radare.org/r/