Micro Control
Attacking uC Applications

Don A. Bailey
(donb@isecpartners.com)
whois donb?
What’s this uC thing all about?

- Single integrated computer
- Processor, volatile, non-volatile storage
  - All In One
- Can drive many peripherals
- Easily programmable
- Field update/upgrade capability
- Personalization (EEPROM)
No, really... Why do I care?

- Your car
- Implanted medical devices
  - **WBAN** (Wireless Body Area Network)
- Crops monitoring (hydro/aero/enviro-ponics)
- Infrastructure monitoring (SCADA, etc)
- “Smart Dust”
- Access controls (RFID, biometrics, etc)
Now with More Networking!

- Bluetooth
- USB
- 802.11
- 802.15.4
- RFID
- DECT
- GSM
Security?

- Some tamper resistance
- Hardware security
- From a software point of view?
  - Crypto support
  - ...?
OODA Loop?

- Field upgrades are rare
  - But getting more common
  - ST M24LR64 Dual EEPROM (Leet!!)
- Most firmware is legacy code
- Spot updates for new functionality / peripherals
- Mostly written in C, C++, and/or ASM
Why wouldn’t you PWN an uC?
Prior work?

- Travis Goodspeed
  - GoodFET, neighbor!

- Josh Wright
  - Killerbee!
Picking on Atmel AVR8
Lots of uC out there, but…

• Popular with hackers and engineers
• Free toolchain (gcc based)
• Free IDE (AVR Studio 4)
• No Soldering necessary
• Relatively cheap dev tools
  ▫ AVRISP mkII (~30 USD)
  ▫ AVR JTAGICE mkII
    (good deals from Arrow Electronics)
Let’s Talk Hardware
Typically included in AVR8

- ALU
- Flash
- SRAM
- EEPROM
- Peripheral support (USART, SPI, I2C, TWI, etc)
That’s right, it’s Harvard

- Separate Data and Code lines
- Code always retrieved from Flash
- Data always retrieved from SRAM
- Flash can be written in software
  - Typically Boot Loader Support
  - Fuses determine this
  - Some AVR8 don’t support this
Point?

- Attack data, not instructions
- Return-to-whatever (ROP :-P)
- Easier! Less data to inject (typically)
- Takes longer
- That’s what GoodFET is for
  - Snatch one Smart Dust sensor
  - GoodFET
  - Analyze code
  - Build ROP strategy
  - Own 100 more remotely
Let’s Talk Software
Typical AVR8 Stuff?

- Interrupts
- Atomic Execution (sort of ;-) 
- Stack
- 32 8-bit registers
- LSB
- 8/16/32/64-bit integer support
- Access to I/O mem
- RISC
What doesn’t AVR8 have?

- Security boundaries
- Contexts (multiple stacks)
- Concurrency
- Segmentation/Paging
- No atomic instructions (cmpxchg?)
- Native 32/64-bit integer support
- Exceptions
  - Where’s the Page Fault, yo?!
Let’s Talk Program Flow
Typical programmatic flow

- Reset
- Init
- Main
- somefunc
On startup

• AVR sets PC to OxOO in Flash
• OxOO = Reset Vector
• JMP to init in crtO
• Init does stuff...
• Call main
• Do stuff...
• Call somefunc
• Do more stuff...
From RESET -> main()
+00000000:    940C003E  JMP    0x0000003E    Jump
+00000002:    940C0053  JMP    0x00000053    Jump
+00000004:    940C0053  JMP    0x00000053    Jump
+00000006:    940C0053  JMP    0x00000053    Jump
+00000008:    940C0053  JMP    0x00000053    Jump
+0000000A:    940C0053  JMP    0x00000053    Jump
+0000000C:    940C0053  JMP    0x00000053    Jump
+0000000E:    940C0053  JMP    0x00000053    Jump
+00000010:    940C0053  JMP    0x00000053    Jump
+00000012:    940C0053  JMP    0x00000053    Jump
+00000014:    940C0053  JMP    0x00000053    Jump
+00000016:    940C0053  JMP    0x00000053    Jump
+00000018:    940C0053  JMP    0x00000053    Jump
+0000001A:    940C0053  JMP    0x00000053    Jump
+0000001C:    940C0053  JMP    0x00000053    Jump
+0000001E:    940C0053  JMP    0x00000053    Jump
+00000020:    940C0053  JMP    0x00000053    Jump
+00000022:    940C0053  JMP    0x00000053    Jump
+00000024:    940C0053  JMP    0x00000053    Jump
+00000026:    940C0053  JMP    0x00000053    Jump
+00000028:    940C0053  JMP    0x00000053    Jump
+0000002A:    940C0053  JMP    0x00000053    Jump
+0000002C:    940C0053  JMP    0x00000053    Jump
+0000002E:    940C0053  JMP    0x00000053    Jump
+00000030:    940C0053  JMP    0x00000053    Jump
+00000032:    940C0053  JMP    0x00000053    Jump
+00000034:    940C0053  JMP    0x00000053    Jump
+00000036:    940C0053  JMP    0x00000053    Jump
+00000038:    940C0053  JMP    0x00000053    Jump
+0000003A:    940C0053  JMP    0x00000053    Jump
+0000003C:    940C0053  JMP    0x00000053    Jump
+0000003E:    940C0055  CALL   0x0000003E    Jump
+0000003F:    940C005E  OUT    0x3F,R1    Out to I/O location
+00000040:    940C0063  SER    R28    Set Register
+00000041:    940C0072  E01F    E0D0    Load immediate
+00000042:    940C0072  LDI    R29,0x10    Load immediate
+00000043:    940C0072  E01F    E0E0    Load immediate
+00000044:    940C0072  OUT    0x32,R29    Out to I/O location
+00000045:    940C0072  LDI    R17,0x01    Load immediate
+00000046:    940C0072  LDI    R26,0x00    Load immediate
+00000047:    940C0072  LDI    R27,0x01    Load immediate
+00000048:    940C0072  LDI    R30,0x0F    Load immediate
+00000049:    940C0072  E01F    E0F8    Load immediate
+0000004A:    940C0072  E01F    E0F8    Load immediate
+0000004B:    940C0072  LDI    R31,0x08    Load immediate
+0000004C:    940C0072  RJMP    R0,0x0003    Relative jump
+0000004D:    940C0072  LPM    R0,0x02    Load program memory and postincrement
+0000004E:    940C0072  ST    R0,0x05    Store indirect and postincrement
+0000004F:    940C0072  CPI    R26,0x86    Compare with immediate
+00000050:    940C0072  CPC    R27,0x17    Compare with carry
+00000051:    940C0072  F7D9    BRNE    PC-0x04    Branch if not equal
+00000052:    940C0072  940E0098  CALL   0x00000098    Call subroutine
+00000053:    940C0072  940C0476  JMP    0x00000476    Jump
+00000054:    940C0072  940C0000  JMP    0x00000000    Jump
crt0 Copy of .rodata
<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100</td>
<td>25 64 2E 25 64 2E 25 64 %d %d %d</td>
</tr>
<tr>
<td>000108</td>
<td>00 25 73 00 64 6F 6E 62 .%s.donb</td>
</tr>
<tr>
<td>000110</td>
<td>27 73 20 6D 65 6D 64 75 's memdu</td>
</tr>
<tr>
<td>000118</td>
<td>6D 70 20 73 74 61 72 74 mp start</td>
</tr>
<tr>
<td>000120</td>
<td>69 6E 67 20 75 70 2E 2E ing up..</td>
</tr>
<tr>
<td>000128</td>
<td>2E 0D 00 30 31 32 33 34 ...01234</td>
</tr>
<tr>
<td>000130</td>
<td>35 36 37 38 39 61 62 63 56789abc</td>
</tr>
<tr>
<td>000138</td>
<td>64 65 66 0D 00 2D 2D 2D def----</td>
</tr>
<tr>
<td>000140</td>
<td>2D 2D 2D 2D 2D 2D 2D 2D -----------</td>
</tr>
<tr>
<td>000148</td>
<td>2D 2D 2D 2D 2D 2D 2D 2D -----------</td>
</tr>
<tr>
<td>000150</td>
<td>2D 2D 2D 2D 2D 2D 2D 2D -----------</td>
</tr>
<tr>
<td>000158</td>
<td>2D 2D 0D 00 64 75 6D 70 ---dump</td>
</tr>
<tr>
<td>000160</td>
<td>69 6E 67 20 30 20 2D 3E ing 0 -&gt;</td>
</tr>
<tr>
<td>000168</td>
<td>2D 66 66 66 66 66 20 77 68 ffff wh</td>
</tr>
<tr>
<td>000170</td>
<td>65 72 65 20 52 41 4D 45 ere RAME</td>
</tr>
<tr>
<td>000178</td>
<td>4E 44 3D 25 70 0D 00 25 ND=.%p..%</td>
</tr>
<tr>
<td>000180</td>
<td>63 00 0D 25 2E 30 34 78 c..%.04x</td>
</tr>
<tr>
<td>000188</td>
<td>2D 00 25 2E 30 32 78 20 .% .02x</td>
</tr>
<tr>
<td>000190</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>000198</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>0001A0</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>0001A8</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>0001B0</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
</tbody>
</table>
Stack Dump After Call to main()
<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>001000 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0008</td>
<td>001008 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0010</td>
<td>001010 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0018</td>
<td>001018 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0020</td>
<td>001020 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0028</td>
<td>001028 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0030</td>
<td>001030 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0038</td>
<td>001038 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0040</td>
<td>001040 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0048</td>
<td>001048 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0050</td>
<td>001050 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0058</td>
<td>001058 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0060</td>
<td>001060 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0068</td>
<td>001068 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0070</td>
<td>001070 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0078</td>
<td>001078 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0080</td>
<td>001080 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0088</td>
<td>001088 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0090</td>
<td>001090 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0098</td>
<td>001098 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0100</td>
<td>0010A0 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0108</td>
<td>0010A8 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0110</td>
<td>0010B0 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0118</td>
<td>0010B8 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0120</td>
<td>0010C0 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0128</td>
<td>0010C8 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0130</td>
<td>0010D0 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0138</td>
<td>0010D8 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0140</td>
<td>0010E0 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0148</td>
<td>0010E8 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0150</td>
<td>0010F0 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x0158</td>
<td>0010F8 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
Function Call
@000000E5: main
57:     {    
+000000E5: 93DF    PUSH    R29    Push register on stack
+000000E6: 93CF    PUSH    R28    Push register on stack
+000000E7: D000    RCALL   PC+0x0001 Relative call subroutine
+000000E8: B7CD    IN      R28,0x3D  In from I/O location
+000000E9: B7DE    IN      R29,0x3E  In from I/O location
58:     dummy(0x7f, 0x8f, 0x9f);    
+000000EA: E78F    LDI     R24,0x7F  Load immediate
+000000EB: E090    LDI     R25,0x00  Load immediate
+000000EC: E86F    LDI     R22,0x8F  Load immediate
+000000ED: E070    LDI     R23,0x00  Load immediate
+000000EE: E94F    LDI     R20,0x9F  Load immediate
+000000EF: E050    LDI     R21,0x00  Load immediate
+000000F0: 940E0082 CALL    0x00000082 Call subroutine
60:     return run() ? True : False ;
Frame Setup/Teardown
42:  
   stropy(x, c);  
   01CE    MOVW  R24.R28  
   9601    ADIW  R24.0x01  
   9400    CALL  0x00000214  
  81BB    LDD  R24.Y+3  
  61C9    LDD  R25.Y+4  
  2789    EOR  R24.R25  
  2F28    MOV  R18.R24  
  E030    LDI  R19.0x00  
  E040    LDI  R20.0x00  
  E050    LDI  R21.0x00  
  0D0F    LDD  R24.Y+31  
  2388    TST  R24  
  F411    BRNE  PC+0x03  
  E041    LDI  R20.0x01  
  E050    LDI  R21.0x00  
  0F24    ADD  R18.R20  
  1F35    ADC  R19.R21  
   01C9    MOVW  R24.R18  
  96A0    ADIW  R23.0x20  
  B60F    IN   R0.0x3F  
  94F8    CLI   
  BFDE    OUT  0x3E.R29  
  BE0F    OUT  0x3F.R0  
  BFCD    OUT  0x3D.R28  
   9508    RET   

45:  
    @000000BC: dummy  

49:  
    {  
  92CF    PUSH  R12  
  92DF    PUSH  R13  
  92EF    PUSH  R14  
  92FF    PUSH  R15  
  930F    PUSH  R16.R0  
  931F    PUSH  R17  
  93DF    PUSH  R29  
  93CF    PUSH  R23  
  B7CD    IN   R23.0x3D  
  B7DE    IN   R29.0x3E  
  97A0    SBIW  R23.0x20  
  B60F    IN   R0.0x3F  
  94F8    CLI   
  BFDE    OUT  0x3E.R29  
  BE0F    OUT  0x3F.R0  
  BFCD    OUT  0x3D.R28  
  018C    MOVW  R15.R24  
}
Four Main Points Demonstrated...

- Function conventions are typical
  - Optimization may minimize this
- Code Layout
- Data Layout
- Atomic Code Sections
Code Layout in Flash

- Interrupt Vectors at OxOO
- RESET Vector at OxOO
- Main Application Code
- Data (???)
- Boot Loader Section
  - Can write to Flash (if Fuses allow) for field updates
Data Layout in SRAM

- Registers at Ox00
- I/O Memory at Ox20
- Extended I/O Memory
- Data (copied from Flash) at Ox100
- BSS
- Heap
- Stack
- ??? ;-)

...
Atomicity

- CLI used
- SREG can be accessed via SRAM (I/O memory)
- 1 CPU Cycle to write to SREG
- Flow:
  - Save a copy of SREG
  - Clear Interrupt Bit in SREG
  - Perform uninterrupted action
    - Write to low byte of SP
    - Write to SREG (old state with interrupt bit set)
    - Write to high byte of SP
Now, Let’s Have Some *Real* Fun
Entropy? What entropy?

- Randomness is very weak
- Crypto hurt as a result
- Pools can be accumulated
  - “True Random Number Generator On an Atmel uC” – IEEE Paper
- 8 Random Bits using RC oscillator
  - *Per second***
Race Conditions

• No semblance of context switching
  ▫ TinyOS/Contiki simulate it
• Critical Sections secured through CLI
• Attack these sections
  ▫ Overwrite SREG; enable Interrupts
• Use Interrupts to cause unexpected behavior
Return Value Checks

• Snprintf returning $\leq 0$ or $\geq \text{sizeof buf}$?
• Logic Issue
• Always a problem
memcpy and Friends

- Latest avr-libc
- Don’t test for negative size values
- No option to “secure” with CLI
  - Interruptable
  - Oops...Where’d my SP go?! ;-)


Buffer Overflows

• Easy as pie
• Instruction address in mem is /2
• Return Oriented Programming
  ▫ Get those Registers set up correctly!
• Force a jump to the Boot Loader
• Instant Flash update (simulate field update)
• Can be triggered remotely
• AVR doesn’t know the difference between you and developer
Frame Pointer Overwrite

- Standard FP overwrite
- Point stack to attacker controlled data
- Next frame has the RET
- FP saved LSB first
Setjmp

- Obvious target
- Often used
- Makes up for lack of exceptions
- Saves entire program state
- Overwrite all registers
- Overwrite PC
Integer Overflows

• Work as expected
• 8-bit registers
• 16-bit native instructions
• Easy to wrap 0xFFFF
Integer Promotion

- Normal integer promotion
- Unsigned -> Signed = No Sign Extension
- Signed -> Signed = Sign Extension
- Stop using ‘char’ for everything ;-) 
- Lots of 8-bit networking protocols
  - 8-bit size fields
  - Promoted to int during packet ingestion
  - Oops!!
Heap Overflows

- Heap Struct consists of \{ size, Next* \}
- Next* points to the next free heap chunk
- Adjacent chunks are combined
- No function pointers 😞
- Easily mangle data
- Next* doesn’t have to point to Heap 😊
- Heap data isn’t zeroed on free()
- Easy way to create pseudo stack frames
- ROP Helper!
Double Free

- Latest avr-libc free() doesn’t check
- Any address can be used (except NULL)
- Free() will happily overwrite first 2 bytes with
  - Next*
- Add it to the free list ;-) 
- Can stealthily force malloc() to return (void*)0xOO
- Write direct to Registers, I/O memory, etc
- ROP Helper!!
“Segment” Collision

- Heap is allocated slightly under stack
- Stack is dynamic!!
- BSS is adjacent to Heap
- .rodata isn’t Read Only! Adjacent to BSS
- One big happy family!
Uninitialized Variables

- Allocate a large Heap chunk
- Spray with OxAABB
- Stack decends into Heap
- Bewm!
- Example code at:
  - [http://pa-ri.sc/uC/dangle.tar.bz2](http://pa-ri.sc/uC/dangle.tar.bz2)
Format Strings

- Current avr-libc has no %n support
- No fun 😞
- But, kind of reasonable
NULL Pointer Dereferences

• There are no privilege rings, but still useful
• Functions like malloc() still return NULL
• (void*)OxOO points to Registers in SRAM
• NULL deref is a very good thing
• Like free() bug, instant access to Regs, I/O Mem
• On the flip side...
  ▫ ??? ;-)
Beyond Memory

• Dereference beyond physical memory addresses?
• Example: ATmega644P
  ▫ 4096 bytes SRAM
  ▫ Total 4196 addressable bytes
    • With registers, I/O memory
• Ox1OFF should be highest addressable address
<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>00</td>
</tr>
<tr>
<td>001008</td>
<td>00</td>
</tr>
<tr>
<td>001010</td>
<td>00</td>
</tr>
<tr>
<td>001018</td>
<td>00</td>
</tr>
<tr>
<td>001020</td>
<td>00</td>
</tr>
<tr>
<td>001028</td>
<td>00</td>
</tr>
<tr>
<td>001030</td>
<td>00</td>
</tr>
<tr>
<td>001038</td>
<td>00</td>
</tr>
<tr>
<td>001040</td>
<td>00</td>
</tr>
<tr>
<td>001048</td>
<td>00</td>
</tr>
<tr>
<td>001050</td>
<td>00</td>
</tr>
<tr>
<td>001058</td>
<td>00</td>
</tr>
<tr>
<td>001060</td>
<td>00</td>
</tr>
<tr>
<td>001068</td>
<td>00</td>
</tr>
<tr>
<td>001070</td>
<td>00</td>
</tr>
<tr>
<td>001078</td>
<td>00</td>
</tr>
<tr>
<td>001080</td>
<td>00</td>
</tr>
<tr>
<td>001088</td>
<td>00</td>
</tr>
<tr>
<td>001090</td>
<td>00</td>
</tr>
<tr>
<td>001098</td>
<td>00</td>
</tr>
<tr>
<td>0010A0</td>
<td>00</td>
</tr>
<tr>
<td>0010A8</td>
<td>00</td>
</tr>
<tr>
<td>0010B0</td>
<td>00</td>
</tr>
<tr>
<td>0010B8</td>
<td>00</td>
</tr>
<tr>
<td>0010C0</td>
<td>00</td>
</tr>
<tr>
<td>0010C8</td>
<td>00</td>
</tr>
<tr>
<td>0010D0</td>
<td>00</td>
</tr>
<tr>
<td>0010D8</td>
<td>00</td>
</tr>
<tr>
<td>0010E0</td>
<td>00</td>
</tr>
<tr>
<td>0010E8</td>
<td>00</td>
</tr>
<tr>
<td>0010F0</td>
<td>00</td>
</tr>
<tr>
<td>0010F8</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>51</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Address (0x100)</td>
<td>Data</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
</tr>
<tr>
<td>000100</td>
<td>25   64 2E 25 64 2E 25 64</td>
</tr>
<tr>
<td>000108</td>
<td>00   25 73 00 64 6F 6E 62</td>
</tr>
<tr>
<td>000110</td>
<td>27   73 20 6D 65 6D 64 75</td>
</tr>
<tr>
<td>000118</td>
<td>6D   70 20 73 74 61 72 74</td>
</tr>
<tr>
<td>000120</td>
<td>69   6E 67 20 75 70 2E 2E</td>
</tr>
<tr>
<td>000128</td>
<td>2E   0D 00 30 31 32 33 34</td>
</tr>
<tr>
<td>000130</td>
<td>35   36 37 38 39 61 62 63</td>
</tr>
<tr>
<td>000138</td>
<td>64   65 66 0D 00 2D 2D 2D</td>
</tr>
<tr>
<td>000140</td>
<td>2D   2D 2D 2D 2D 2D 2D 2D</td>
</tr>
<tr>
<td>000148</td>
<td>2D   2D 2D 2D 2D 2D 2D 2D</td>
</tr>
<tr>
<td>000150</td>
<td>2D   2D 2D 2D 2D 2D 2D 2D</td>
</tr>
<tr>
<td>000158</td>
<td>2D   2D 0D 00 64 75 6D 70</td>
</tr>
<tr>
<td>000160</td>
<td>69   6E 67 20 30 20 2D 3E</td>
</tr>
<tr>
<td>000168</td>
<td>20   66 66 66 66 66 20 77 68</td>
</tr>
<tr>
<td>000170</td>
<td>65   72 65 20 52 41 4D 45</td>
</tr>
<tr>
<td>000178</td>
<td>4E   44 3D 25 70 0D 00 25</td>
</tr>
<tr>
<td>000180</td>
<td>63   00 0D 25 2E 30 34 78</td>
</tr>
<tr>
<td>000188</td>
<td>20   00 25 2E 30 32 78 20</td>
</tr>
<tr>
<td>000190</td>
<td>00   00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>000198</td>
<td>00   00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0001A0</td>
<td>00   00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0001A8</td>
<td>00   00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0001B0</td>
<td>00   00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>
There is no Page Fault on AVR8

- Memory faults cannot occur
- For program safety, don’t RESET
- Read AND Write support
- Just wrap addresses back to (void*)OxOO
- Overwriting past end of PHYSMEM = start of PHYSMEM
- i.e. Ox1100 = Ox0100
- How convenient ;-)
- Overwrite EVERYTHING ANYWHERE
Example code?

• See the memdump application
  ▫ Runs on any AVR8 with USART
  ▫ http://pa-ri.sc/uC/memdump.tar.bz2

• Code tested on 10 different uCs in the AVR family
  ▫ ATtiny
  ▫ ATmega
We Pack and Deliver like UPS Trucks
Summary?

- Ripe environment for application vulnerabilities
- Little protection schemes
  - Except solid auditing and a tight SDLC
- Lots of legacy code in the field
- Lots of important devices
Thanks for your support...

- Dhillon Kannabhiran
- iSEC Partners
- Nick DePetrillo
- Mike Kershaw
- Travis Goodspeed
- Josh Wright
- Dennis Brown
- Dan Guido
- David Munson
Terima kasih!

@DonAndrewBailey
donb@isecpartners.com