Attacking Microcontrollers From a Software Perspective

Don A. Bailey
(donb@isecpartners.com)
A copy of this presentation...

- http://pa-ri.sc/uC/
- Including all example software
whois donb?
Ok, donb. What’s this all about?
How did I get here?

- “Device Profiling” from Carmen Sandiego
- Tracking “types” of devices
  - Telephone network
- Make assumptions about capability
- Attack remotely based on profile
uC are Everywhere

- Common household items
  - Utilities
  - Toys
- Access control system
  - RFID
  - Biometrics
  - Tracking devices
- Sensor networks
  - WBAN
  - SCADA
  - UTC
- Basebands
Um. I’m still not convinced, donb.
Substantial Growth in this Market!

• “The sales of [uC] is expected to reach $22.99 billion in 2020 from 2009, at a compound annual growth rate (CAGR) of 5.6%...”
  ▫ Research and Markets via Business Wire (Nov 30, 2010)

• Included sales data from
  ▫ Renesas, NEC, Freescale, Infineon, Microchip

• Didn’t include data from
  ▫ Atmel, Texas Instruments, everyone else?
Speaking of Atmel...

• They beat predicted 2010 Q4 Sales predictions
• Estimated 436.1 million USD
• Resulted in 457.8 million USD
• Atmel: “Largely due to [uC] sales…”
• Claims uC sales will be the primary driver of Q1 2011
• (Reuters, Feb 8th, 2011)
In other words?
**Invest!**

- Atmel stock in mid-2009?
- Atmel stock today?
- Japanese earthquake/tsunami/nuclear
  - Causing a slight hit
  - Buy while low!
Driving Investments

- Increasing versatility
- More robust than ever
- Decreasing cost of manufacturing
- More consumers, less cost
- Pass savings on to bulk consumers
Engagement Building

- Whole new arena for security work
- Interesting security problems
- Unmanned devices = difficult to protect
Now with More Networking!

- Bluetooth
- ANT+
- USB
- CAN
- 802.11
- 802.15.4
- RFID
- DECT
- GSM
And yet... What isn’t changing?
Security?

- Some tamper resistance / Hardware security

- Firmware security?
  - Upgrade
  - Debug
  - Read

- From a software point of view?
  - Crypto support
  - …?
Upgrade Capabilities?

- Field upgrades are less rare
  - FOTA
  - Mesh firmware distribution

- Personalization
  - ST M24LR64 Dual EEPROM (Leet!!)

- Most firmware is legacy code
  - Msp430-libc partly based on avr-libc

- Spot updates for new functionality / peripherals

- Mostly written in C, C++, and/or ASM
  - Lightweight JVMs in some cases
Why wouldn’t you PWN an uC?
Prior work? Hardware/Proto Focused

- Travis Goodspeed
  - GoodFET, neighbor!

- Josh Wright
  - Killerbee!

- Mike Ossmann
  - Ubertooth!
Picking on Atmel AVR8 and MSP430/x
Why AVR8 and MSP430/x?

- Both are RISC
- Simple instruction set
- Common peripherals
- Similarly developed/deployed
- But, architecturally very different
- They represent two primary types of uC
  - I’ll elaborate later
Lots of uC out there, but...

- Popular with hackers and engineers
- Free tool chain (gcc based)
- Free IDE (AVR Studio 4)
- Minimal (MSP430) or no (AVR) Soldering necessary
- Relatively cheap dev tools
  - GoodFET (free)
  - AVRISP mkII (~30 USD)
  - AVR JTAGICE mkII (good deals from Arrow Electronics)
Let's talk Hardware
MSP430/x Hardware
Typically included in MSP430/x

- ALU
- Flash
- SRAM
- Peripheral support (USART, IrDA SPI, I2C, TWI, etc)
That’s right, it’s von Neumann

- Unified memory
- CPU accepts
  - Data from Code regions
  - Code from Data regions
  - Yay!
- Flash can be written
  - Via JTAG
  - Via Bootstrap Loader (BSL) in memory
  - Via “Custom Solution”???
Point?

• Attack data, not instructions
• Return-to-whatever is unnecessary
• Straight forward!
• Shellcode
• Can be owned remotely very easily
AVR8 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
MSP430/x Software
Typical MSP430/x Stuff?

- Interrupts
- Atomic Execution (sort of ;-) 
- Stack/heap/bss/data 
- 16 16-bit registers (20-bit on MSP430x) 
- LSB 
- 8/16/32/64-bit integer support 
- Access to peripheral via mem 
- RISC
What doesn’t MSP430/x have?

- Security boundaries
- Contexts (multiple stacks)
- Concurrency
- Segmentation/Paging
- No atomic instructions (cmpxchg?)
- Exceptions
  - Where’s the Page Fault, yo?!
AVR8 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
MSP430/x Program Flow
On MSP430/x startup

- Retrieves address at 0xFFFE in Flash
  - Reset Vector
- Branch to Reset (init)
- Init does stuff...
- Call main
- Do main stuff...
- Call somefunc
- Do more stuff...
From RESET -> main()
Function Call
Frame Setup / Tear Down
00003462 <dummy>:
04 12      push r4
31 80 10 00 sub #16, r1 ;#0x0010
04 41      mov r1, r4
0d 12      push r13
04 12      push r4
0f 12      push r15
30 12 5a 34 push #13402 ;#0x345a
30 12 10 00 push #16 ;#0x0010
04 12      push r4
b0 12 ca 35 call #0x35ca
31 50 0c 00 add #12, r1 ;#0x000c
31 50 10 00 add #16, r1 ;#0x0010
34 41      pop r4
30 41      ret

0000348a <main>:
31 40 00 21 mov #8448, r1 ;#0x2100
3d 40 33 00 mov #51, r13 ;#0x0033
3e 40 22 00 mov #34, r14 ;#0x0022
3f 40 11 00 mov #17, r15 ;#0x0011
b0 12 62 34 call #0x3462
Four Main Points Demonstrated...

• Function conventions are typical
  ▫ Optimization may minimize this

• Code Layout

• Data Layout

• No Atomic Code Sections
Memory Layout

- Special Function Registers (SFR) at OxOO
- Peripheral Memory at Ox1O
- Bootstrap Loader
- Data (copied from Flash) at Ox11O2
- BSS
- Heap
- Stack
- Code
- Interrupt vectors (OxFFC0 – OxFFFF)
- Extended Flash (MSP430x)
AVR8 Program Flow
On AVR8 startup

- AVR sets PC to Ox00 in Flash
- Ox00 = Reset Vector
- JMP to init in crt0
- Init does stuff...
- Call main
- Do stuff...
- Call somefunc
- Do more stuff...
From RESET -> main()
0: File not found
+00000000: 940C000E JMP 0x00000003E Jump
+00000002: 940C0053 JMP 0x000000053 Jump
+00000004: 940C0053 JMP 0x000000053 Jump
+00000006: 940C0053 JMP 0x000000053 Jump
+00000008: 940C0053 JMP 0x000000053 Jump
+0000000A: 940C0053 JMP 0x000000053 Jump
+0000000C: 940C0053 JMP 0x000000053 Jump
+0000000E: 940C0053 JMP 0x000000053 Jump
+00000010: 940C0053 JMP 0x000000053 Jump
+00000012: 940C0053 JMP 0x000000053 Jump
+00000014: 940C0053 JMP 0x000000053 Jump
+00000016: 940C0053 JMP 0x000000053 Jump
+00000018: 940C0053 JMP 0x000000053 Jump
+0000001A: 940C0053 JMP 0x000000053 Jump
+0000001C: 940C0053 JMP 0x000000053 Jump
+0000001E: 940C0053 JMP 0x000000053 Jump
+00000020: 940C0053 JMP 0x000000053 Jump
+00000022: 940C0053 JMP 0x000000053 Jump
+00000024: 940C0053 JMP 0x000000053 Jump
+00000026: 940C0053 JMP 0x000000053 Jump
+00000028: 940C0053 JMP 0x000000053 Jump
+0000002A: 940C0053 JMP 0x000000053 Jump
+0000002C: 940C0053 JMP 0x000000053 Jump
+0000002E: 940C0053 JMP 0x000000053 Jump
+00000030: 940C0053 JMP 0x000000053 Jump
+00000032: 940C0053 JMP 0x000000053 Jump
+00000034: 940C0053 JMP 0x000000053 Jump
+00000036: 940C0053 JMP 0x000000053 Jump
+00000038: 940C0053 JMP 0x000000053 Jump
+0000003A: 940C0053 JMP 0x000000053 Jump
+0000003C: 2411 CLR R1 Clear Register
+0000003E: 241F OUT 0x3F.R1 Out to I/O location
+00000040: ECF0 SER R28 Set Register
+00000041: E1D0 LDI R29, 0x10 Load immediate
+00000042: BFDE OUT 0x3E.R29 Out to I/O location
+00000043: BFCD OUT 0x3D.R28 Out to I/O location
+00000044: E011 LDI R17, 0x01 Load immediate
+00000045: E0A0 LDI R26, 0x00 Load immediate
+00000046: E0B1 LDI R27, 0x01 Load immediate
+00000047: EFE0 LDI R30, 0xF0 Load immediate
+00000048: E0F8 LDI R31, 0x08 Load immediate
+00000049: C002 RJMP PC+0x0003 Relative jump
+0000004A: 9005 LPM R0.Z+ Load program memory and postincrement
+0000004B: 920D ST X+.R0 Store indirect and postincrement
+0000004C: 38A6 CPI R26, 0x86 Compare with immediate
+0000004D: 07B1 CPC R27, R17 Compare with carry
+0000004E: F7D9 BRNE PC-0x04 Branch if not equal
+0000004F: 940E0098 CALL 0x00000098 Call subroutine
+00000050: 940C0476 JMP 0x00000476 Jump
+00000051: 940C0000 JMP 0x00000000 Jump
crt0 Copy of .rodata
<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00100</td>
<td>25 64 2E 25 64 2E 25 64 %d %d %d</td>
</tr>
<tr>
<td>0x00108</td>
<td>00 25 73 00 64 6F 6E 62 .%s.donb</td>
</tr>
<tr>
<td>0x00110</td>
<td>27 73 20 6D 65 6D 64 75 's memdu</td>
</tr>
<tr>
<td>0x00118</td>
<td>6D 70 20 73 74 61 72 74 mp start</td>
</tr>
<tr>
<td>0x00120</td>
<td>69 6E 67 20 75 70 2E 2E ing up..</td>
</tr>
<tr>
<td>0x00128</td>
<td>2E 0D 00 30 31 32 33 34 ...01234</td>
</tr>
<tr>
<td>0x00130</td>
<td>35 36 37 38 39 61 62 63 56789abc</td>
</tr>
<tr>
<td>0x00138</td>
<td>64 65 66 0D 00 2D 2D 2D def....--</td>
</tr>
<tr>
<td>0x00140</td>
<td>2D 2D 2D 2D 2D 2D 2D 2D ---------</td>
</tr>
<tr>
<td>0x00148</td>
<td>2D 2D 2D 2D 2D 2D 2D 2D ---------</td>
</tr>
<tr>
<td>0x00150</td>
<td>2D 2D 2D 2D 2D 2D 2D 2D ---------</td>
</tr>
<tr>
<td>0x00158</td>
<td>2D 2D 2D 2D 2D 2D 2D 2D ---------</td>
</tr>
<tr>
<td>0x00160</td>
<td>69 6E 67 20 30 20 2D 3E ing 0 -&gt;</td>
</tr>
<tr>
<td>0x00168</td>
<td>20 66 66 66 66 20 77 68 ffff wh</td>
</tr>
<tr>
<td>0x00170</td>
<td>65 72 65 20 52 41 4D 45 ere RAME</td>
</tr>
<tr>
<td>0x00178</td>
<td>4E 44 3D 25 70 0D 00 25 ND=%p..%</td>
</tr>
<tr>
<td>0x00180</td>
<td>63 00 0D 25 2E 30 34 78 c..%.04x</td>
</tr>
<tr>
<td>0x00188</td>
<td>20 00 25 2E 30 32 78 20 .%.02x</td>
</tr>
<tr>
<td>0x00190</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>0x00198</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>0x001A0</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>0x001A8</td>
<td>00 00 00 00 00 00 00 00 ........</td>
</tr>
<tr>
<td>0x001B0</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>0000</td>
</tr>
<tr>
<td>0x0008</td>
<td>0000</td>
</tr>
<tr>
<td>0x0010</td>
<td>0000</td>
</tr>
<tr>
<td>0x0018</td>
<td>0000</td>
</tr>
<tr>
<td>0x0020</td>
<td>0000</td>
</tr>
<tr>
<td>0x0028</td>
<td>0000</td>
</tr>
<tr>
<td>0x0030</td>
<td>0000</td>
</tr>
<tr>
<td>0x0038</td>
<td>0000</td>
</tr>
<tr>
<td>0x0040</td>
<td>0000</td>
</tr>
<tr>
<td>0x0048</td>
<td>0000</td>
</tr>
<tr>
<td>0x0050</td>
<td>0000</td>
</tr>
<tr>
<td>0x0058</td>
<td>0000</td>
</tr>
<tr>
<td>0x0060</td>
<td>0000</td>
</tr>
<tr>
<td>0x0068</td>
<td>0000</td>
</tr>
<tr>
<td>0x0070</td>
<td>0000</td>
</tr>
<tr>
<td>0x0078</td>
<td>0000</td>
</tr>
<tr>
<td>0x0080</td>
<td>0000</td>
</tr>
<tr>
<td>0x0088</td>
<td>0000</td>
</tr>
<tr>
<td>0x0090</td>
<td>0000</td>
</tr>
<tr>
<td>0x0098</td>
<td>0000</td>
</tr>
<tr>
<td>0x00A0</td>
<td>0000</td>
</tr>
<tr>
<td>0x00A8</td>
<td>0000</td>
</tr>
<tr>
<td>0x00B0</td>
<td>0000</td>
</tr>
<tr>
<td>0x00B8</td>
<td>0000</td>
</tr>
<tr>
<td>0x00C0</td>
<td>0000</td>
</tr>
<tr>
<td>0x00C8</td>
<td>0000</td>
</tr>
<tr>
<td>0x00D0</td>
<td>0000</td>
</tr>
<tr>
<td>0x00D8</td>
<td>0000</td>
</tr>
<tr>
<td>0x00E0</td>
<td>0000</td>
</tr>
<tr>
<td>0x00E8</td>
<td>0000</td>
</tr>
<tr>
<td>0x00F0</td>
<td>0000</td>
</tr>
<tr>
<td>0x00F8</td>
<td>0000</td>
</tr>
</tbody>
</table>

Notice: The file size is 51 bytes.
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+0\x0001  Relative call subroutine
+000000E8:  B7CD  IN  R28.0\x3D  In from I/O location
+000000E9:  B7DE  IN  R29.0\x3E  In from I/O location
58:     dummy(0\x7f, 0\x8f, 0\x9f): 
+000000EA:  E78F  LDI  R24.0\x7F  Load immediate
+000000EB:  E090  LDI  R25.0\x00  Load immediate
+000000EC:  E86F  LDI  R22.0\x8F  Load immediate
+000000ED:  E070  LDI  R23.0\x00  Load immediate
+000000EE:  E94F  LDI  R20.0\x9F  Load immediate
+000000EF:  E050  LDI  R21.0\x00  Load immediate
+000000F0:  940E0082  CALL  0\x000000082  Call subroutine
60:     return run() ? True : False ;
Frame Setup/Teardown
42: stropy(x, o):
+00000A0: 01CE  MOVW  R24.R28  Copy register pair
+00000A1: 9601  ADIW  R24.0x01  Add immediate to word
+00000A2: 940E0214  CALL  0x00000214  Call subroutine
+00000A4: 81BB  LDD  R24,Y+3  Load indirect with displacement
+00000A5: 819C  LDD  R25,Y+4  Load indirect with displacement
+00000A6: 2789  EOR  R24.R25  Exclusive OR
+00000A7: 2F28  MOV  R18.R24  Copy register
+00000A8: 0E03  LDI  R19.0x00  Load immediate
+00000A9: E040  LDI  R20.0x00  Load immediate
+00000AA: E050  LDI  R21.0x00  Load immediate
+00000AB: 6D0F  LDD  R24,Y+31  Load indirect with displacement
+00000AC: 2388  TST  R24  Test for Zero or Minus
+00000AD: F411  BRNE  PC+0x03  Branch if not equal
+00000AE: E041  LDI  R20.0x01  Load immediate
+00000AF: E050  LDI  R21.0x00  Load immediate
+00000B0: 0F24  ADD  R18.R20  Add without carry
+00000B1: 1F35  ADC  R19.R21  Add with carry
+00000B2: 01C9  MOVW  R24.R18  Copy register pair
+00000B3: 96A0  ADIW  R23.0x20  Add immediate to word
+00000B4: B60F  IN  R0.0x3F  In from I/O location
+00000B5: 94F8  CLI  Global Interrupt Disable
+00000B6: BFDE  OUT  0x3E.R29  Out to I/O location
+00000B7: BEOF  OUT  0x3F.R0  Out to I/O location
+00000B8: BFCD  OUT  0x3D.R28  Out to I/O location
+00000BA: 91CF  POP  R22  Pop register from stack
+00000BB: 91DF  POP  R29  Pop register from stack
+00000BC: 9508  RET  Subroutine return

@000000BC: dummy

49:
+00000BC: 93DE  PUSH  R12  Push register on stack
+00000BD: 92CF  PUSH  R13  Push register on stack
+00000BE: 92EF  PUSH  R14  Push register on stack
+00000BF: 92FF  PUSH  R15  Push register on stack
+00000C0: 930F  PUSH  R16  Push register on stack
+00000C1: 931F  PUSH  R17  Push register on stack
+00000C2: 93CF  PUSH  R23  Push register on stack
+00000C3: 93DF  PUSH  R29  Push register on stack
+00000C4: B7CD  IN  R23.0x3D  In from I/O location
+00000C5: B7DE  IN  R29.0x3E  In from I/O location
+00000C6: 97A0  SBIV  R20.0x20  Subtract immediate from word
+00000C7: B60F  IN  R0.0x3F  In from I/O location
+00000C8: 94F8  CLI  Global Interrupt Disable
+00000C9: BFDE  OUT  0x3E.R29  Out to I/O location
+00000CA: BEOF  OUT  0x3F.R0  Out to I/O location
+00000CB: BFCD  OUT  0x3D.R28  Out to I/O location
+00000CC: 018C  MOVW  R15.R24  Copy register pair
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
A note from Texas Instruments

Use VLO and DCO
  - Count number of DCO cycles per VLO cycle
  - Even/Odd is unpredictable
    - Bit generation
  - Both internal clocks

DCO current can be sourced through
  - External resistor
  - Change the resistor
  - Get more predictable values
AVR8 Entropy? What entropy?

• Randomness is very weak
• Crypto hurt as a result
• However, Pools can be accumulated!
  ▫ “True Random Number Generator On an Atmel uC” – IEEE Paper
• 8 Random Bits using RC oscillator
  ▫ *Per second***!!
MSP430/x Race Conditions

- Duh, there’s no atomicity!
- There are multiple places for enabling/disabling interrupts
  ▫ SFR (NMIE, etc)
  ▫ SR (GIE)
- Multiple places = multiple steps
  ▫ Not optimal!
- Attack by forcibly enabling interrupts
AVR8 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 on Both

• Snprintf returning $\leq 0$ or $\geq \text{sizeof buf}$?
• Logic Issue
• Always a problem
• May be *more* of a problem in Firmware
  ▫ Less sanity checking to minimize code size!
memcpy and Friends

• Latest avr-libc and msp430-libc
• Does not test for negative size values
• No option to “secure” with atomicity
  ▫ Can be interrupted
  ▫ Oops...Where’d my SP go?! ;-)
MSP430/x Buffer Overflows

- Easy as pie
- Instruction address in mem is LSB
- Inject a Code Section reader
- Write out code to PORT
- Read altered code from PORT
- Instant Flash update from RAM (simulate field update)
AVR8 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 for Both

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

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

- Work as expected
- 8-bit registers
- 16-bit native instructions
- Easy to wrap OxFFFFF
Integer Promotion same for Both

- 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 \textit{int} during packet ingestion
  - Oops!!
MSP430/x Heap Overflows

- Heap Struct consists of { Size|Busy, Data }
- Next* is generated from &Current[Size+1]
- No function pointers 😞
- Easily mangle data
- Heap data isn’t zeroed on free()
- Easy way to create pseudo stack frames
- ROP Helper!
AVR8 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!
MSP430/x Double Free

• Latest msp430-libc free() doesn’t check at all
• Any address can be used (including NULL)
• Free() will *(data – 1) &= OxFFFE
  ▫ Useful to disable Watchdog Interrupts
  ▫ SFR at OxOO
• Can easily force malloc() to return (void*)OxOO
• Write to SFRs!
• ROP Helper!!
AVR8 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*)0x00 
- Write direct to Registers, I/O memory, etc 
- ROP Helper!!
“Segment” Collision for Both!

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

- Allocate a large Heap chunk
- Spray with OxAABB
- Stack decends into Heap
- Bewm!
- Example AVR8 code at:
  - http://pa-ri.sc/uC/dangle.tar.bz2
Format Strings on Both

- Current libcs have no %n support
- No fun
  - But, kind of reasonable
NULL Pointer Dereferences on Both

- There are no privilege rings, but still useful
- Functions like malloc() still return NULL
- (void*)0x00 points to
  - Registers in SRAM on AVR8
  - SFRs on MSP430
- NULL deref is a very good thing
- Like free() bug, instant access to Regs, I/O Mem
Beyond Memory on MSP430/x

- Dereferencing beyond valid memory addresses
  - Returns the LSB of the address
  - Or trash

- Unprogrammed (but, valid) addresses
  - Return 0xFF

- Typically does not cause a Reset, unless
  - The PC points to 0x00 – 0x1FF (peripheral mem)
Beyond Memory on AVR8

- Deref beyond physical memory addresses?
- Example: ATmega644P
  - 4096 bytes SRAM
  - Total 4196 addressable bytes
    - With registers, I/O memory
- 0x11OFF should be highest addressible address
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*)0x00
- Overwriting past end of PHYSMEM = start of PHYSMEM
- i.e. 0x1100 = 0x0100
- 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
Flashing MSP430/x from RAM
Security model

- Nothing to bypass
  - Only BSL and JTAG have restrictions
- The Clock is the problem
  - Device dependent
  - *Can* get around this issue
- *Don’t* have to Erase before Writing
  - Programming bits to zero won’t get you *
  - But it can get you enough
- Program chars to ‘@’ (Ox40) or ‘\’ (Ox60)
  - Or, simply NULL
Program even Far Memory

- Memory past OxFFFF requires MSP430x ASM
  - Build shellcode
  - Load into memory
  - Execute
- Execute Writes until Programmed
  - Simple Word sized Write operations
  - Drop passwords/hashes to NULL
- [http://pa-ri.sc/uC/msp430-release.tar.bz2](http://pa-ri.sc/uC/msp430-release.tar.bz2)
  - Example (verified) Far Write shellcode
  - Full Memory Dump example
What does your Heart Rate Monitor read now?
Summary?

- MSP430x can be easily hacked to
  - Read Flash from RAM
  - Write Flash from RAM

- AVR8 can be easily hacked to
  - Easily own any memory address on the device
  - Leads to simplified ROP

- You can’t NOT own a uC application!
Did I mention the TPM?

- Thinkpad TPMs (and others)
  - Based on AVR8/16 platform

- The Tracking Device I’ll be talking about in April
  - Uses a uC to process SMS and Network data

- Too large a threat surface
  - Too many important devices
OPANA

• OPcode ANAlysis Project
  ▫ More of an Architecture Analysis Project
  ▫ Define opcodes
  ▫ And memory regions
  ▫ And architectural “features”
  ▫ Make decisions about ROP / shellcode generation

• Catalog vulnerabilities and “features” of many architectures
  ▫ ARM Cortex-M3
  ▫ 8051 variants (CC1110 isn’t a typical 8051)
  ▫ PIC
  ▫ etc
Special thanks...

- iSEC Partners, Inc. and NCC Group
- Jim Geovedi
- Y3dips
- Dhillon Kannabhiran
- Nick DePetrillo
- Mike Kershaw
- Travis Goodspeed
- Josh Wright
- Mike Ossmann
- Jared Davies
Thank you!

@DonAndrewBailey
donb@isecpartners.com