Reversing and Exploiting Wireless Sensors

[Work in Progress]

Travis Goodspeed
travis@radiantmachines.com

ABSTRACT

Wireless sensors will soon be part of many industrial, military, and home networks. Of the various networking protocols, which include Zigbee, ISA100, Wireless HART, 6LowPAN, and others, none has yet become a definitive standard. Neither have vendors standardized upon a given operating system, compiler, or microcontroller. Users of these sensor networks are often given no command-line, no internal documentation, and no access to the internals of each device. This paper provides a thorough introduction to reverse engineering such devices, both in hardware and in software.

1. INTRODUCTION

That which follows is an introduction to reverse engineering and exploiting low-powered wireless embedded systems, such as those which implement Zigbee and other 802.15.4-based LPAN protocols. Example being taken chiefly from the first-hand experience of the author, most involve the MSP430 microcontroller. The TinyOS operating system and Telos B hardware are used in order to make the exercises repeatable, but the techniques here presented are intended for use on black-box devices without documentation.

2. MSP430STATIC

Many examples in this paper will include examples from TinyOS being analyzed with MSP430static[1], a disassembly analysis tool by the author of this paper. TinyOS examples were chosen because they may be freely distributed and results may be cross-checked with symbols, neither of which can be done with a firmware image found in the field. Use of these symbols is akin to looking in the back of a textbook; they are good for proving a point, but with few exceptions the field reverse-engineer does not have the luxury of referencing them.

M4S, as it will be called for short, is a hastily written Perl script which accesses an SQLite3 database. Queries are made using SQL with a few pre-processing macros, such as interpretation of hexadecimal addresses. For example, one might select the last five function names and entry addresses by "`select enhex(address), name from funcs order by address desc limit 5;`".

Macros allow for queries without parameters to easily be called; for example, "`.ivt`" will print the Interrupt Vector Table (IVT). Scripts are similar to macros, except that their names are delimited by slashes (/) rather than dots (.) and that they exist as files on disk rather than entries within a macros table.

3. SCHEMATIC DIAGRAMS

Reverse engineering the schematic diagram of a wireless sensor is necessary, but the connections of that schematic are not entirely random. For example, the MSP430F1611, which is commonly found in wireless sensors, has only two hardware serial ports.[2] In the Telos B, the SPI bus is connected to the first of these (USART0) and the FTDI USB/serial chip to the second (USART1).

This is true for clock pins, hardware-assisted I/O ports, JTAG pins, analog I/O, and similar special-purpose pins. Only general purpose I/O pins (GPIO) cannot be found by this method, but for most chip packaging these can be traced using syringes and a continuity tester.

4. ACCESSING FIRMWARE

In order to reverse engineer a wireless sensor, it is necessary to have access to the firmware. There are many ways to accomplish this, each of varying difficulty.

First, the firmware may often be obtained from a device by a JTAG adapter, as many devices do not blow the protection fuses. Fuses are either EEPROM cells, as in the case of the PIC and AVR chips, which may cleared with the rest of flash memory, or they are true fuses in the sense that they physically and irreparably blow, as in the case of the MSP430. Even then, the fuse may be physically reset with either a micro-electronic probe station or more sophisticated hardware.

When invasive or semi-invasive methods of firmware extraction are impractical, attacks on software can be used instead.

[1] http://msp430static.sf.net/

[2] The exception to this rule is the MSP430 serial bootstrap loader (BSL), which uses a software serial port.

To appear at Black Hat Federal, 2009,
February, 2009, Arlington, VA

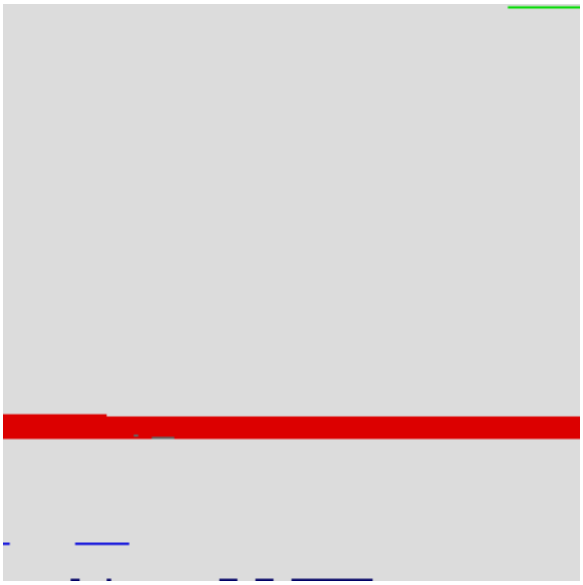

Figure 1: Memory Map of the TinyOS Blink Application

For example, there exists a timing vulnerability[4][3] in the password comparison routine of many versions of the serial bootstrap loader of the MSP430. By exploiting this vulnerability, it is possible to extract firmware even from a locked device. Use of stack overflow exploits, to be presented in this context in Section 11, has also been made by the author to extract locked firmware.

5. AUTOMATED ANALYSIS

5.1 Mapping Memory

One of the simplest forms of scripted analysis, shown in Figure 1, is a memory map. In this example, the low byte of the address is the horizontal position while the high byte is the vertical position. Origin (0x0000) is the bottom-left corner, 0xFF00 the top-left, and 0xFFFF the top-right.

By sight, one can recognize critical features of the application. At 0xFFE0, near the top-right, is the Interrupt Vector Table (IVT). In the bottom two rows (0x0000 to 0x0200), which are the Special Function Registers (SFR), one can see the I/O ports, clocks, and similar functions which are configured by the image.

The red region beginning at 0x4000, the $4 \times 16 = 64^{th}$ line, is the actual code of the application. From the IVT, listed in Figure 1, it can be seen that the IVT's RESET vector, at 0xFFFE, points to the bottom of flash memory. GCC for the MSP430 seems to always have that behavior, in that the RESET handler always points to the lowest address in flash memory. Another indication that this image came from GCC is that unused interrupts all point to the same address; other compilers typically leave those addresses as 0xFFFF, the default state for flash memory.

5.2 Graphs

A callgraph, such as is depicted in Figure 2, consists–in graph theory terms–of functions as vertices and function calls as edges. The sixteen entries of the IVT, from Table 1, are also seen here, in the bottom right. The pattern of many IVT entries passing to a single handler, which then calls a single leaf function, will become familiar to those reverse engineering MSP430 firmware compiled by GCC.

ffe0	403a	__ctors_end
ffe2	403a	__ctors_end
ffe4	403a	__ctors_end
ffe6	403a	__ctors_end
ffe8	403a	__ctors_end
ffea	40b4	sig_TIMERA1_VECTOR
ffec	4068	sig_TIMERA0_VECTOR
ffee	403a	__ctors_end
fff0	403a	__ctors_end
fff2	403a	__ctors_end
fff4	403a	__ctors_end
fff6	403a	__ctors_end
fff8	43b6	sig_TIMERB1_VECTOR
fffa	40fa	sig_TIMERB0_VECTOR
fffc	403a	__ctors_end
fffe	4000	_reset_vector__

Table 1: Interrupt Vector Table for Blink

6. MICROCONTROLLERS

6.1 Harvard and Von Neumann

Microcontroller architectures are considerably more varied than those of personal computers. Register counts vary, as do the purposes of registers. Unlike desktop processors, which are all Von Neumann for practical purposes, many microcontrollers are Harvard architecture, having one memory for data and another for instructions.

Von Neumann machines have a single address range within which are both executable code and non-executable data. This model will be familiar to those without embedded experience. A pointer is a pointer, and its result may be fetched as either code or data. Embedded examples include most implementations of ARM, PowerPC, and the MSP430.

By contrast, Harvard machines have two address spaces: one for code, and another for data. Separate memories provide the intentional advantage of much simpler hardware design, as both a word of code and a word of data may be fetched at the same time. As will be discussed later, this also complicated stack overflows, as Harvard machines are unable to execute data memory. Examples include the PIC, 8051, and AVR microcontrollers.

6.2 Memory Mapped I/O, IVT

When reverse engineering a unix application, it is common practice to identify system calls to the kernel as a means of finding functions of interest. For example, it might be worthwhile to trap all calls to read() and write() as a means of spying on the traffic of a closed-source application. In embedded systems such as wireless sensors, there is rarely a kernel to handle I/O; rather, an application will itself directly peek and poke I/O ports at memory-mapped addresses.

Suppose that we are looking for the serial byte transmit function of a Telos B device to its CC2420 radio. As that

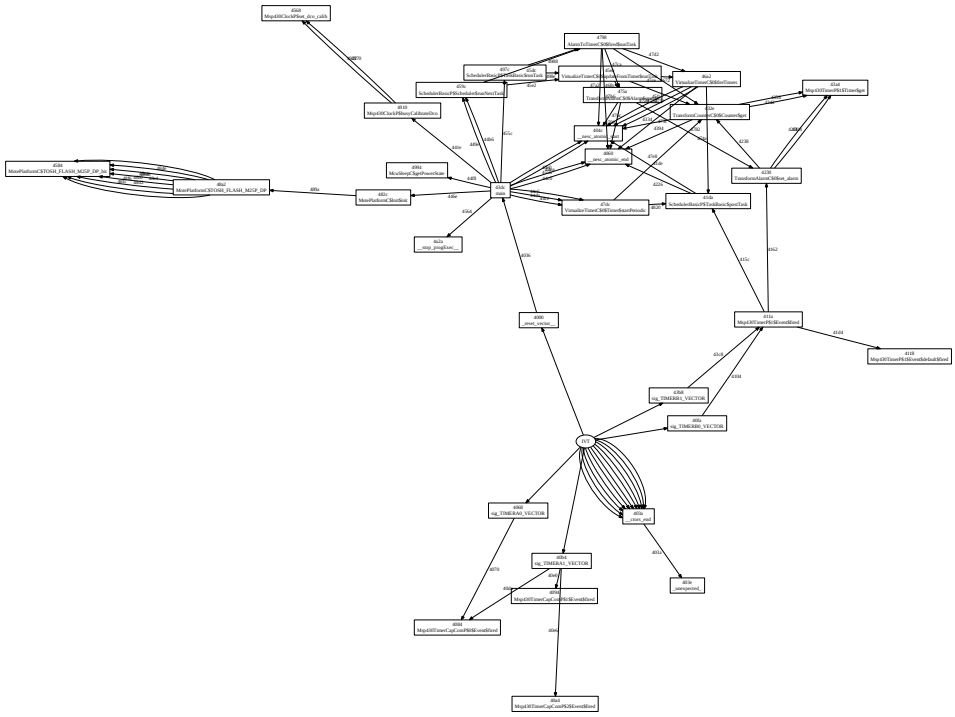

Figure 2: Call Graph for Blink

U0CTL	0x0070	Control
U0TCTL	0x0071	Transmit Control
U0RCTL	0x0072	Receive Control
U0MCTL	0x0073	Modulation Control
U0BR0	0x0074	Baud Rate 0
U0BR1	0x0075	Baud Rate 1
U0RXBUF	0x0076	Receive Buffer
U0TXBUF	0x0077	Transmit Buffer

Table 2: USART0 Special Function Registers

sensor uses the MSP430F1611, we can use the header files of any popular MSP430 compiler to identify the memory-mapped address to which bytes are written for transmission. By referencing the schematic[3] or by following circuit traces, it can be seen that the first serial port, USART0, connects to the CC2420 radio. Table 2 provides a list of USART0's Special Function Registers (SFR) which I/O functions will reference.

The SQL query "`select name from funcs where asm like '%0077%';`" returns two addresses with the TinyOS 2.0 BaseStation application, and the shorter of these two functions is the SPI write function, 'Msp430SpiNoDmaP SpiByte write'. [4]

This function's disassembled code is presented in Table 3. Note that the parameter, which was passed through r15, is copied into local variable r11, which is then copied into &0x0077, U0TXBUF. For practice, optimize the function by hand to eliminate usage of r11.

```
477e  push r11
4780  mov.b r15, r11
4782  call #16460
4786  mov.b r11, &0x0077
478a  call #16480
478e  mov.b &0x0002,r15
4792  and.b #64, r15
4796  jz $+6
4798  mov #1, r15
479a  jmp $+4
479c  clr r15
479e  and.b #-1, r15
47a0  cmp.b #0, r15
47a2  jz $-20
47a4  and.b #-65, &0x0002
47aa  call #18356
47ae  and.b #-1, r15
47b0  pop r11
47b2  ret
```

Table 3: Msp430SpiNoDmaP 0 SpiByte write

7. 16-BIT EMBEDDED OPERATING SYSTEMS

An embedded operating system is a very different piece of software from a desktop operating system. In the 16-bit world, these operating systems rarely if ever support preemptive multitasking or kernel/user separation. The OS is linked into the user application, and only a single user application will exist on a given device. Further, as everything is statically linked, a reverse engineer will never see symbols within a firmware image, only within pre-distribution executables such as those produced directly by a compiler.

Popular open source operating systems for wireless sensors include Contiki and TinyOS, but it is not uncommon for a vendor of WSN hardware to write his own operating system from scratch.

8. INLINING

It is common for embedded operating systems to optimize memory usage and execution speed by automatically inlining functions where appropriate. In such operating systems, a function will often be quite long, with many leaf functions contained within it. Further, these long functions will often require more registers than are available as scratch registers, leading to opening clauses of many PUSH instructions and ending clauses of many POP instructions.

9. NETWORKING STACKS

As has been implied by the previous sections, one can expect to find the networking stack mixed with the handling routines in the firmware of a wireless sensor node. Lacking multiprocessing, memory protection, and even memory management, there is little reason not to have the radio chip's driver simply call a function to handle an incoming packet. Similarly, there is little reason not to have a function directly call the radio driver to send a packet.

The exception to this can be profitable, and it involves delayed processing of a packet. Suppose, for example, that instead of directly calling the handler for a packet, the driver posts an event for later handling. When that event is executed, it might read the packet and then forward the packet to another event, which is also delayed. Doing this keeps the stack depth to a minimum and it can make power management more effective for low-packet-rate applications.

In the case of this chained approach, it might be possible to start a race condition. By replaying a valid administrative packet, followed by an invalid one, the attacker might cause the former to be validated and the latter to be executed.

In either case, it is important to identify how packets are being handled as, in the case of event-posting, there will might be no direct call path from the radio driver to the function handler in question.

10. SIMULATION

A good simulator is quite handy for testing a theory or proving a point. For the MSP430 platform, your author is rather font of MSP430simu, part of the MSPGCC project. For a detailed demonstration of modifying a simulator, see [6], which describes the modification of MSP430simu to produce LaTeX slides of an MSP430 stack overflow for [5].

[3]http://www.tinyos.net/scoop/special/hardware/

[4]An unwelcome side effect of the TinyOS macro parser, NesC, is that functions often have horribly complex names delimited by the dollar sign ($). For purposes of legibility, this delimiter will be presented as a space in prose and a double-underscore (__) in code within this paper.

11. STACK OVERFLOWS

11.1 Von Neumann

Stack overflow exploits of Von Neumann wireless sensors are similar to those of an older personal computer, in that the stack is executable. The primary difficulty is in finding vulnerable code, as will be discussed in Section 14, and in fitting an exploit into an 802.15.4 packet, which has a hard limit of 128 bytes.

The first exploit of a wireless sensor node was written by the author in the summer of 2007. Descriptions can be found in [5] and [2].

Bypassing this restriction is possible by recognizing making use of unused portions of RAM. Usually, global variables are placed at the bottom of RAM and a stack grows downward from the top of RAM. In the case of the MSP430, code injected to this region is immediately executable. Once a flash memory driver has been written to this region, it becomes possible to copy small blocks to flash memory for a larger executable. This can then contain a driver for external flash, slowly loading a larger image to external memory. Finally, when everything is ready, the internal firmware image can be moved to external flash, and an image from external flash loaded. If such an image were to broadcast its own injection routines, it would be a self-propagating worm.

Further, as an infected sensor would have access to the keys of its prior life, it could infect by the transmission of encrypted packets. In this way, the infection could spread along a web of trust, even in the case of perfect, unbreakable cryptography.

11.2 Harvard

Infecting a Harvard sensor is considerably more complicated in that injected code may not be run directly. Instead, preexisting code must be called, and the purpose of that code is often quite different from the intended behavior of an exploit.

While non-executable stack overflows were performed by Gu in [7], the first execution of foreign code was performed by Francillon in [1] by use of a technique known as return-oriented programming. In this technique, the tail of one function is called, which then returns to the tail of another function. Multiple tails, or meta-gadgets, can then be assembled into a larger gadget.

For example, in collaboration with Francillon, the author once found it necessary to place an arbitrary value, such as 0xDEAD, into r11 of an MSP430[5], after which the machine was to return to the entry point of anther function, 0xBEEF. Performing an overflow that leaves 0x47B0, 0xDEAD, 0xBEEF on the stack at the moment of a return will pop 0xDEAD into r11, then return to 0xBEEF if the contents of Table 3 is in memory.

By chaining many such gadgets together, Francillon was able to construct a meta-gadget that copies information from an arbitrary address in data memory to an arbitrary address in code memory. Once there, the code can be executed directly, making possible such things as self-propagating sensor worms.

[5]The MSP430 is not a Harvard machine, but all examples of this paper stick to that platform for the sake of consistency.

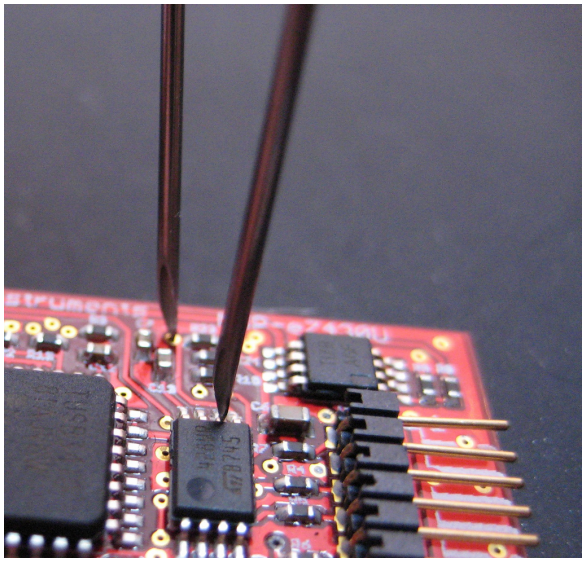

Figure 3: I^{2}C Bus Tap

12. PATCHING

As linkers lay out code in a regular fashion, often growing upward from a lower boundary, the position of code within an executable is predictable. This linking strategy leaves the upper regions of flash memory, with the exception of the IVT, empty on the MSP430. As no elegant tool yet exists for injecting code into such firmware, it can be manually patched into this upper region.

A nice feature of the popular Intel Hex format for microcontroller code is that each line of ASCII text is a data record, concluding with ":00000001FF". By grepping this line out of a file as it is prepended to another, a script can easily merge two images, adding a patch to an image.

13. BUS SNIFFING

Also worth noting is that the busses used on these devices are unprotected and vulnerable to sniffing and injection. Traffic may even be diverted by a bus extender, one that repeats only those messages which it wishes to let through. Figure 3 shows an I^{2}C bus being tapped in this manner.

14. FUZZING

Fuzz testing of a wireless sensor proposes a unique difficulty, in that the simulators for many of these platforms are lacking in sufficient accuracy to run an entire image. In the absence of such a simulator, how is the reverse engineer to identify and trace the cause of a crash?

In the case of blind fuzzing, many chips revert GPIO pins to a high-impedance input state and, while TinyOS does not, many commercial wireless sensors print debugging information over a serial port when booting. Either of these effects can be used to identify which packet caused a crash, so long as packets are sent slowly and the device does not lock up.

For cases in which the firmware is known by the attacker, the author has found it profitable to patch the image to contain a handler function which preserves the most recently received packet. The fuzzing then includes with a high frequency the address of that handler.

15. CONCLUSION

Several techniques have been presented for the reverse engineering and exploitation of 16-bit wireless embedded systems. It is suggested that the reader follow the citations of this paper to investigate any individual topic in greater detail.

16. REFERENCES

[1] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS 2008*.

[2] T. Goodspeed. MSP430 buffer overflow exploit for wireless sensor nodes, August 2007.

[3] T. Goodspeed. Practical attacks against the MSP430 BSL. 25C3, December 2008.

[4] T. Goodspeed. A side-channel timing attack of the MSP430 BSL. Black Hat USA, August 2008.

[5] T. Goodspeed. Stack overflow exploits for MSP430 wireless sensors over 802.15.4. Texas Instruments Developer Conference, February 2008.

[6] T. Goodspeed. Tracing with MSP430simu, LaTeX, and PowerPoint, January 2008.

[7] Q. Gu and R. Noorani. Towards self-propagate mal-packets in sensor networks. In *Wisec 2008*.