# Low-level Design Vulnerabilities in Wireless Control Systems Hardware

Travis Goodspeed
University of Tennessee, Knoxville
Jefferson City, TN
travis@utk.edu

Darren R. Highfill and Bradley A. Singletary
EnerNex Corporation
Knoxville, TN
{darren, brad}@enernex.com

**Abstract:** Control systems elements like Advanced Metering Infrastructure (AMI) networks fully field wireless sensors and controls outside a utility's physical security perimeter, placing them at a high risk of compromise. System attackers have every opportunity to damage, sniff, spoof, or tamper communications hardware platforms for malicious, hobbyist, or incidental reasons. This paper demonstrates the relevance of common control systems communications hardware vulnerabilities that lead to direct control systems compromise. The paper describes several enabling vulnerabilities exploitable by an attacker, the design principles that causing them to arise, the economic and electronic design constraints that restrict their defense, and ideas for vulnerability avoidance. Topics include design induced vulnerabilities such as the extraction and modification of communications device firmware, man-in-the-middle attacks between chips of a communications devices, circumvention of protection measures, bus snooping, and other attacks. Specific examples are identified in this report, ranked by attack feasibility. Each attack was investigated against actual IEEE 802.15.4 radio architectures.

**Keywords:** Wireless Security, IEEE 802.15.4, Embedded Systems, Design Vulnerability, Hardware Vulnerability, Firmware Vulnerability, Hardware Attack

## 1 Introduction

Vulnerabilities of wireless devices concern more than merely the wireless channel. When a crucial element of a control system sits within physical reach of an attacker, as happens in AMI and other systems, the attacker may physically observe and modify the device. In this paper, we demonstrate a few specific attacks against embedded systems in technical detail. Products are vulnerable to such attacks due to architectural limitations (as in the case of bus snooping), programmer carelessness (as in the case of buffer overflows), and unintended analog properties of digital microchips (as in the case of voltage glitching). It is the primary intention of the authors to present a few of these attacks in detail

sufficient for a proof of concept implementation. Secondly, the impact of these vulnerabilities is discussed.

When properly engineered, a given piece of critical infrastructure is not fragile; system component resilience, redundancy, monitoring, and other hardening techniques reduce the risks of failure and compromise for a critical system. In many deployed control systems, individual components of a critical infrastructure's control system are too often fragile with respect to the overall system, and the weakest components are particularly vulnerable to attack. Physical security, information confidentiality, information integrity, information availability and other direct risk mitigating control types are therefore applied to protect control systems component assets. The concept of system to component vulnerability is recursive if we treat the components themselves as systems, opening up the device's own physical design for mitigation. If the physical environmental and physical security constraints expected for a system can not be met for a component, can we expect to retain information security at the system level? The answer depends on the security context in which a control system component is deployed, the means and the methods by which it interacts with other components.

When control systems and their components have been fielded or pushed outside a security mitigated entity perimeter, but are counted on for a measurable level of functionality, they become subject to direct catastrophic exploit of vulnerabilities in individual hardware and software components. This problem is enhanced by the fact that interconnecting networks are required to link control system assets to each other and to points of command and control. Asset owners usually field control systems with redundancy so they can afford to lose some number of components or interconnects and still operate properly. Wireless network topologies, like mesh networks, are seeing wide adoption for field device networks since they offer low cost path redundancy should a link be blocked or fail. For some control systems, components are also made redundant. Redundancy and in general reliability engineering do not offer much security beyond basic availability concerns. In a field system, should a component be lost to a skilled attacker, it puts access to the control network and likely the control application at direct risk, endangering other field assets. It is the case that greater care is required to mitigate the threat on the component level, despite the lack of field perimeters.

This paper focuses in on a specific enabler of control systems interconnect: wireless components and their wireless related subcomponents. Wireless media technology is experiencing a significant adoption in fielded control systems. Wireless enables remote access, remote control, condition monitoring, and other tasks common to field systems and control infrastructure in ways that wired interconnects cannot. In the electric utility market sector, wireless offers significant market acceleration and cost reduction for topics like in-substation monitoring and control, and advanced metering infrastructure (AMI). From a security standpoint, field wireless interconnect infrastructure is only differentiated from field wired interconnect infrastructure by the fact that wired media links are constructed along a specific spatial path while wireless media links currently has much broader areas of spatial propagation of the signal, even for directional signals. While this difference is a general concern for wireless, in field systems, a lack of strong physical perimeters makes these differences irrelevant.

The paper examines how a threat agent with wireless hardware engineering skills would see the components and sub-components of wireless enabled control systems as lacking any security mitigation due to the commonality of vulnerabilities across the wireless component and subcomponent markets. Attacks studied are always developed from the ground up, as threat agents do not typically operate within any particular design guideline. Vulnerability is demonstrated regardless of the component crypto-system. This paper illuminates several of the many common means of exploiting hardware, and explores how such vulnerabilities occur. We review how the minute details matter, but are often overlooked when development efforts are resource, budget, and time constrained. While attacks were evaluated for specific platforms and architectures, the authors believe multiple wireless vendors have significant risk associated with categories considered.

In this paper components manufactured by Texas Instruments dominate the examples because of the authors' familiarity with them, and their broad use for IEEE 802.15.4 wireless sensor designs. This paper does not demonstrate any particular vulnerability of these parts as compared to their competition, and no recommendation is made to avoid the part numbers cited within these examples. Further, it is likely the vulnerabilities discussed are not unique to a specific vendor.

## 2 Embedded System Architecture Security Impact

Standard wireless embedded implementation technologies such as IEEE 802.15.4 are generally designed to serve specific market needs. Therefore, the market offers components that translate such standards to mass producible designs. Embedded wireless technologies typically, but not always, have relatively low power consumption, component cost, computational power requirements, design cost, and implementation cost. Commodity variants of components that implement wireless technology generally have higher individual reliability than custom designs, and a ready and willing engineer talent pool to integrate them. Almost all such components are designed to leverage or integrate with existing mass production components and subcomponents such as microcontrollers, RAM chips, ROM chips, and others. All of the above is highly desirable. As with all such technologies that have the potential to achieve economy of scale in design and implementation, vulnerability generally follows or surpasses all cost optimizations and design trade-offs unless specifically mitigated. Such optimizations and economies of scale can serve to broaden the impact of overlooked security flaws, turning their advantage into a weakness.

This paper does not attempt to cover all potential aspects for such wireless technology implementations, much less the entire range of implementation issues for a single technology. We present security vulnerabilities for typical components found in specific IEEE 802.15.4 implementations; and abstract them to help translate real-world tactical security vulnerabilities as recognizable design classes requiring consideration for mitigation. This paper does not educate the reader in the many nuances of RF design. For RF design and implementation issues, see individual standards such as [7] and engineering references including, but not limited to, [8, 9].

We present an abstraction of monolithic vulnerable aspects of a typical commodity IEEE 802.15.4 platform, the Telos-B development kit. While the Telos-B is a basic user-programmable development kit, its architecture is close enough to most typical applications to be considered general. This abstraction is intended to give the reader a repeatable context as a starting point when looking at other platform architectures.

Figure 1 is a logical drawing of the Telos-B 802.15.4 implementation, indicating major subsystems found in the design. Each subsystem represents a starting point for vulnerability analysis for the design. Figure 2 is an annotated image of the Telos-B front side and Figure 3 is of the reverse. In the annotated figures we see an analog front end (antenna and filter only), a TI CC2420 802.15.4 radio, its oscillator, an MSP430 host MCU, its oscillator, a serial EEPROM and the power supply.

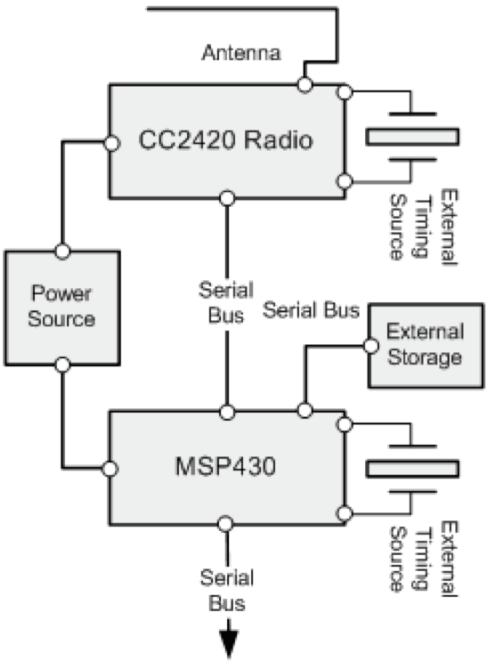


*Figure 1 – Telos-B Node Logical View*

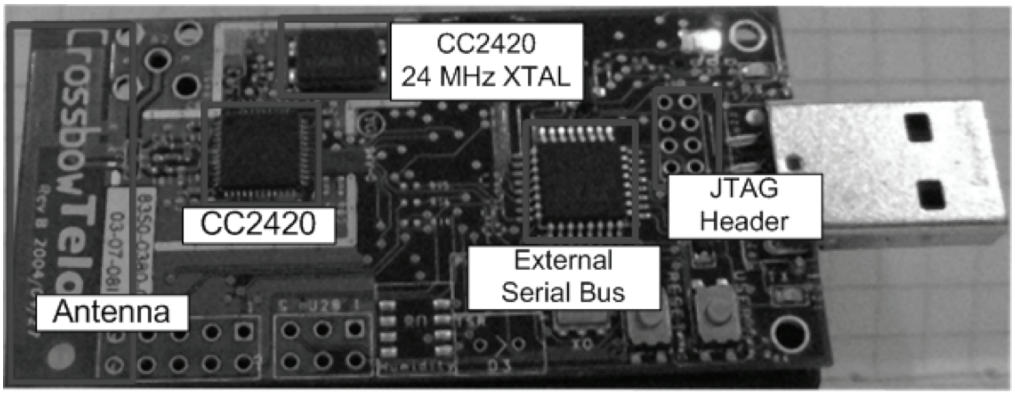


*Figure 2 – Telos-B Node Annotated Physical View [Front]*

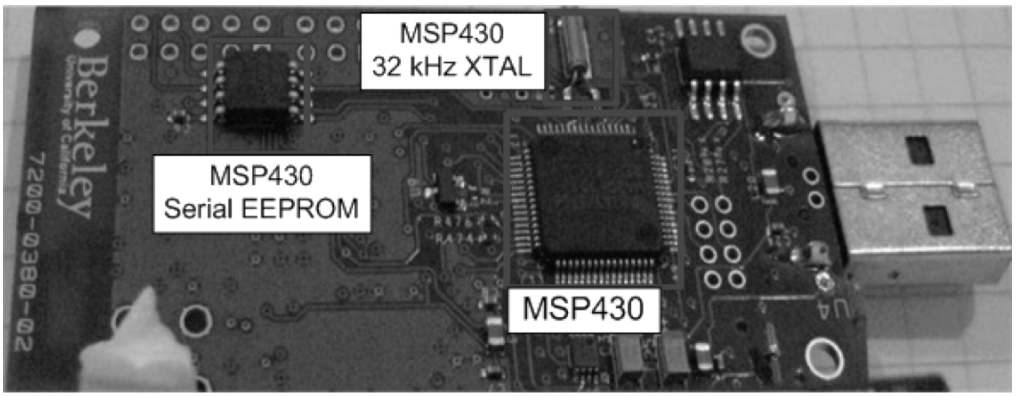


*Figure 3 – Telos-B Node Annotated Physical View [Reverse]*

Looking at Figures 1 to 3 from the standpoint of a designer, we observe various subsystems and design implications. In embedded RF applications such as wireless sensor based capital condition monitoring, the application software is housed within a low power application host processor such the pictured MSP430. The RF physical, media access, link layer, and sometimes network layers will be offloaded onto an RF component such as the pictured CC2420. Breaking up the design lets designers implement the RF portion of the application with the best possible RF module for the lowest time to market while targeting host applications to the optimal host processor. Most standalone communications modules will be linked to their host processor by a trivial board-level serial bus such as SPI or I2C. In some designs the host processor also contains the RF stack implementation, eliminating the board-level serial bus. Components such as microcontrollers or host processors rarely fully implement the analog portion of an RF module.

Antennae, inbound and outbound amplifiers, RF switches, and various filters are generally integrated separately as their requirements vary widely across potential

applications. Due to their application orientation, host processors will have external timing means. In general external oscillators reduce processor chip cost and allow the designer to scale the system to the cheapest clock source meeting application requirements. Though typically not used, many 802.15.4 RF modules have a means to slave a host's clock to the RF module to further reduce design cost. Power is supplied to the devices as required by the module, though often platform power requirements are aligned to reduce component count and subsequent cost.

This simplistic overview of 802.15.4 RF logical and physical hardware can be viewed as a system which when treated as a whole may be viewed as having integrity, availability, and confidentiality up to the design of the standards and applications it implements. PHY, MAC, LINK, and higher layer network security may be considered nominal. If we look at the Figures 1 to 3 again as an attacker, though, a different story unfolds. The specific implementation architecture of the host or RF module processor signals multiple potential software attack vectors against the network implementation. The specific component interconnection conveys where hardware level snooping and man in the middle attacks are possible. Environmental design requirements on the components illustrate where device malfunction might be exploited to bypass implementation security mechanisms. Storage and memory architecture identifies likely targets for direct key and firmware extraction.

Figure 4 shows a portion of an embedded wireless design broken up by major component and augmented with minor components perceived to house a majority of platform hardware vulnerabilities possible, some of which are explored in this paper. We have generalized the prior logical diagram to typify vulnerability of the majority of platforms available on the market. Instance platforms will be seen to modify this model by combining components, adding minor components, and even combining the major ones. Modifications that combine components do not necessarily reduce the vulnerability expressed in the diagram. The Zigbee-capable 802.15.4 chipsets listed in Table 1 were used as data points to support this general architecture. We chose to assess Zigbee oriented chipsets as they are built on IEEE 802.15.4, Zigbee specifies a network layer, and such chips usually have part of the overall security mechanisms built in for developer convenience. Providing a network layer is a likely requirement for larger wireless control systems with credible impact if compromised.

While the options for direct attack against a commodity 802.15.4 hardware implementation are nearly limitless, we can easily imagine that hardware oriented attacks could involve some of the following assuming a larger motive of arbitrary control over a portion of a wireless control network's behavior.

- Confidentiality
    - Snooping Bus Traffic
    - Extracting Firmware for Vulnerability Analysis
    - Extracting Stored Information
    - Snooping Side Channels

- Integrity
    - Tampering Bus Traffic
    - Replacing Hardware Components
    - Modifying Existing Components
    - Bypassing Hardware Components
    - Disrupting or Distorting Normal Hardware Operation
    - Bypassing Software Components
- Availability
    - Jamming or Shrouding
    - Alert/Condition Flooding
    - Run Battery Down

Note that the specific effect of an attack can often be created by multiple means. For example, a bus may be snooped by subverting any endpoint on the bus, or perhaps creating a new endpoint. The unfortunate reality is that the bus and means of configuration are the enabler for multiple attacks. Means for securing the bus do exist, but device design and material costs increase significantly.

Components expressed in Figure 4 include:

**RF Front End:** The set of fully analog components required to guide waves to the PHY layer processor.

> Key Subcomponents: Antenna(s), antenna switches, low noise/power amplifiers, baluns
>
> Key External Dependencies: Power supply, RF/EM environment, temperature environment
>
> Potentially Vulnerable to: Jamming, Shrouding, Disruption, Distortion, Spoofing, Replay, Tampering, other manipulation of higher layer behavior through direct physical attack.
>
> Discussion: The RF front end is difficult to defend, and somewhat out of scope for this paper. Generally, defense at this layer is better found in the form of a coordinated external wireless intrusion detection system, higher layer modulation, and possibly multiple coordinated antennas.

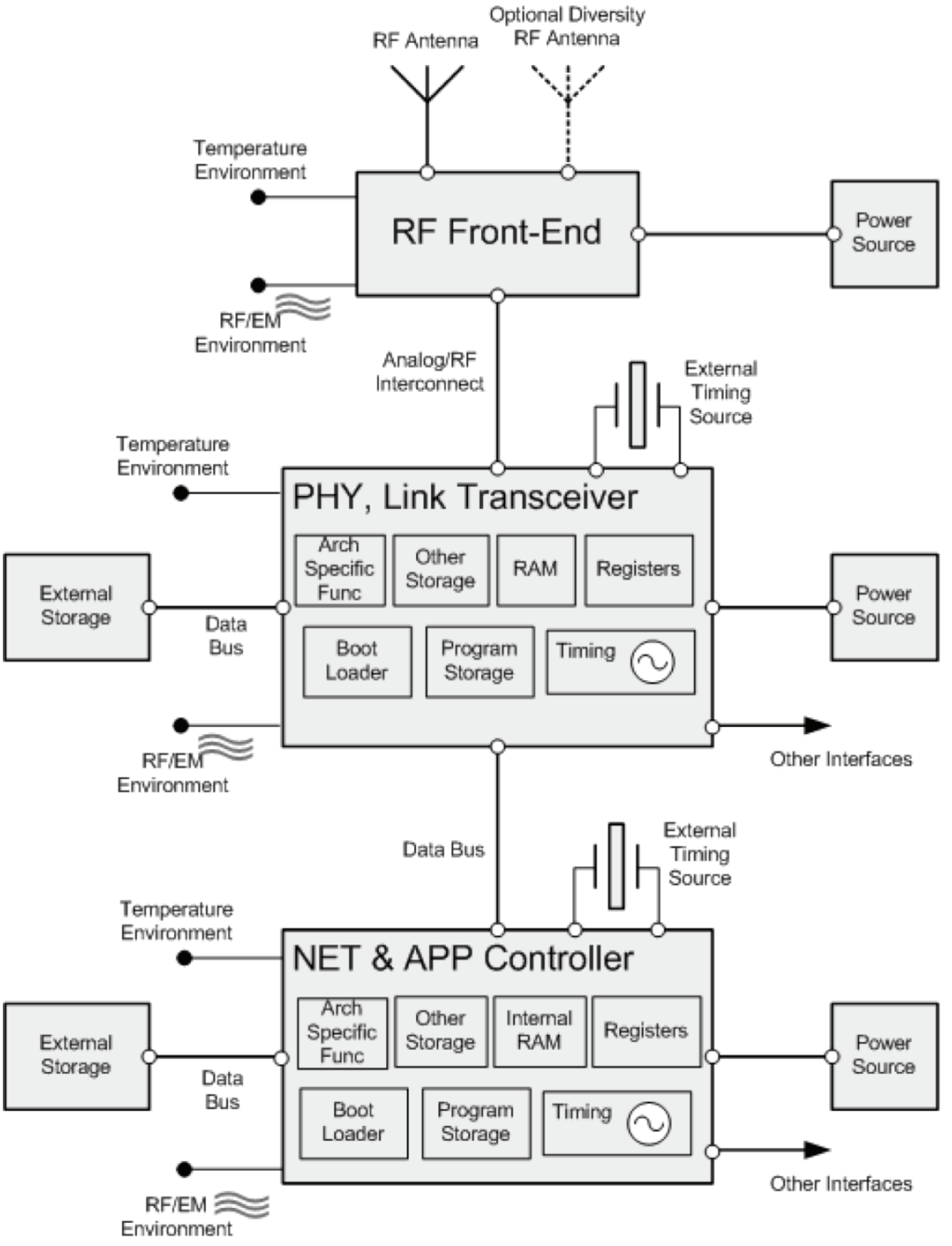


*Figure 4 – Generalization of Embedded Wireless Architecture Attack Vectors*

**PHY, Link Transceiver:** The subcomponent that deals with PHY, MAC, and LINK layer issues. Potentially executes link layer cryptography algorithms.

> Key Subcomponents: Registers, RAM, other storage, boot loader, internal program storage, internal timing source, and architecture specific functionality
>
> Key External Dependencies: RF Front End, NET & App Controller, data bus to NET & App Controller, external timing source, power supply, RF/EM environment, temperature environment
>
> Potentially Vulnerable to: DoS, Disruption, Distortion, Spoofing, Snooping, live code injection, serial Bus tampering, reconfiguration, firmware analysis, firmware tampering, snooping side channels, environmental tampering, etc.
>
> Discussion: The PHY, Link Transceiver is likely vulnerable to any attacks that a microcontroller would be vulnerable to.

**NET & APP Controller:** The subcomponent that primarily focuses on executing any higher layer network functionality. This is generally an independent microprocessor or microcontroller that may also run the application.

> Key Subcomponents: Registers, RAM, other storage, boot loader, internal program storage, internal timing source, and architecture specific functionality
>
> Key External Dependencies: PHY, Link Transceiver, external buses, data bus to the PHY, Link Transceiver, external timing source, power supply, RF/EM environment, temperature environment, external storage
>
> Potentially Vulnerable to: DoS, Disruption, Distortion, Spoofing, Snooping, live code injection, serial Bus Tampering, reconfiguration, flash/RAM snooping, flash/RAM tampering, firmware analysis, firmware tampering, snooping side channels, environmental tampering, tampering of external flash, etc.
>
> Discussion: The NET & APP Controller is likely vulnerable to to direct attacks against its integrity, whether executing live code, or dormant. The NET & APP controller flash and RAM likely contain cryptographic material that is subject to disclosure. Further configuration parameters for the PHY, Link Transceiver are often stored or loaded by this subcomponent.

| Mfr. | Chipset | Storage | TX/RX | Antenna Diversity | Integration Bus(s) | Integrated Micro-controller | Host Processor Required |
|---|---|---|---|---|---|---|---|
| Ember | EM250 | 128 kB Flash 5 kB RAM | 5 dBm / -100 dBm | NO | UART / SPI / I2C | XAP2B | NO |
| Ember | EM260 | 1024 B Flash | 4.5 dBm / -99 dBm | NO | UART / SPI / I2C | XAP2B | YES |
| Atmel | AT86RF230 | 128 B SRAM | 3 dBM / -101 dBm | NO | SPI | Custom / Undisclosed | YES |
| Atmel | AT86RF231 | 128 B SRAM | 3 dBM / -101 dBm | YES | SPI | Custom / Undisclosed | YES |
| TI | CC2420 | 368 B RAM | 0 dBm / -95 dBm | NO | SPI | Custom / Undisclosed | YES |
| TI | CC2430 / CC2431 | 32/64/128 kB Flash 8 kB RAM | 0 dBm / -92 dBm | NO | SPI | 8051 | NO |
| TI | CC2480A1 | 128 kB Flash 8 kB SRAM | 0 dBm / -92 dBm | NO | SPI | 8051 | NO |
| TI | CC2520 | 768 B RAM MCU configured | 5 dBm -98 dBm | NO | SPI | Custom / Undisclosed | YES |
| Freescale | MC13202 | ? RAM ? ROM | 3 dBm / -92 dBm | Partial TX \|\| RX | SPI | Custom / Undisclosed | YES |
| Freescale | MC1321x | 4 kB RAM 60 kB Flash | 0 dBm / -92 dBm | Partial TX \|\| RX | SPI | HCS08-A | NO |
| Freescale | MC1322x | 96 kB RAM 128 kB Flash 80 kB ROM | 4 dBm / -95 dBm | NO | UART / SPI / I2C | ARM7 | NO |

*Table 1 – Zigbee Capable RF Chips*

The following sections illustrate how vulnerabilities on RF hardware architectures might unfold in architecture oriented software and hardware attacks. We begin by looking at the simplest common vulnerabilities and work our way to more complicated vulnerabilities. We only present a few attacks for any vulnerability class.

# 3 Software Attacks

Certain software vulnerabilities may be considered hardware architecture vulnerabilities in the case where the attack is enabled by security weakness in the platform architecture, and not necessarily a direct fault of the developer's adherence to security oriented implementation. It is likely that without rigorous vulnerability testing and platform assessment, common hardware architecture induced software vulnerabilities may be located by an attacker and exploited. The attacker has many options to identify attack avenues for embedded RF subcomponents.

It is possible if not likely that firmware may be extracted from the device through one means or another, even if the firmware was not specifically designed to be modified or extracted from the device. It may even be possible to place modified firmware back onto the device. Firmware is generally feasible to reverse engineer for common vulnerabilities. The attacker can directly test any unsecured API's, pins, buses, or other means of giving data to the processor for typical architectural vulnerabilities. It is also the case that even if such vulnerabilities do not seem to exist or seem secure, that their enabling conditions can possibly be induced by environmental or direct tampering of the hardware on an architecture by architecture basis. Tools and documentation generally exist to support development, maintenance, and debugging at this level. The end result of software based attacks is generally a low cost, large scale compromise of wireless network integrity and confidentiality that may lead to commodity compromise of other platforms over the air.

## 3.1 Stack Overflows

Buffer overflows of one kind or another are a popular method among attackers for inserting executable code into a running process on a victim workstation or server. Similarly, executable code may be injected into control systems hardware, particularly--but not exclusively--those that rely upon Von Neumann architecture devices. As most of this hardware lacks an operating system with user/kernel mode separation, injected code often has full privileges on the victim machine. It could erase all of flash memory, broadcast erroneous sensor results, perform dangerous motor actions, reveal cryptographic key material, and broadcast any radio message. Note that taking control of a wireless node via over the air exploit can result in a node that can be used to further propagate the malicious code to other nodes!

In this section a functional stack overflow exploit for the MSP430-based Telos B wireless sensor node will be described, leaving [4] to serve as a more detailed tutorial on the subject. In review, the MSP430 is a 16-bit Von Neumann architecture with many RISC properties. The MSP430 serves as the host processor for the sensor node. MSP430 instructions and int datatype are both 16-bit and even aligned. Calling conventions differ by compiler, but for simple functions, a return pointer is pushed to the stack followed by local variables. For these examples, all parameters will be passed by registers rather than on the stack. Also recall that the call-stack grows downward in C based implementations, with each pushed item residing at a lower address than that which preceded it.

Suppose, for example, that int machlang[] is an array of MSP430 machine code 16-bit integers. That is, executable code begins at memory label &machlang and machlang is the entry point of that code. The traditional direct method in C of executing code stored at a properly aligned random access memory offset is via a function pointer:

```
//Declare a function pointer.
int (*machfn)() = NULL;
//Set it to machlang.
machfn= (int (*)()) machlang;
//Call that function.
machfn();
```

However, there exist less direct ways of doing this. Consider how the following code manipulates the program counter.

```
void machret(){
 //oldpc will be initialized to its own address/
 int *oldpc=&oldpc;
 //Increment oldpc by 2, such that it points to the
 //stored PC value.
 oldpc++;
 //Overwrite the old PC value with the address of code
 //to be executed.
 *oldpc=machlang;

 //Code execution returns to machlang, not to the
 //calling function.
 return;
}
```

In the code above, machret() replaces a return value stored on the stack with a new address within memory. Nothing in the standard C language prevents this from compiling and working in practice. A stack overflow attack works similarly to this, with this exception that, on the victim application author's part, the stack overflow is accidental, rather than intentional. Such a victim function might look like the following, in which radio_msg_t is a C struct containing the contents of a received radio packet.

```
//Handle a command present within a received message.
void handlemsg(radio_msg_t* rcm){

 //Copy the incomming command to a stack string.
 char cmd[6];
 strcpy(cmd,rcm->cmd);

 //Set LEDs as is appropriate.
 if(!strcmp(cmd,"RED"))
 setleds(1);
 if(!strcmp(cmd,"GREEN"))
 setleds(2);
 if(!strcmp(cmd,"BLUE"))
 setleds(4);

 return;
}
```

The handlemsg() function behaves as expected for commands `RED', `GREEN', `BLUE', each of which sets the apropriate LED by the setleds() function. Unexpected commands such as `green' and `GRUN' will have no visible effect, as neither matches strcmp()'s case-sensitive comparisons.

It must be remembered, however, that a string in C is nothing more than an array of characters terminated by zero. "RED" within memory is {'R','E','D',0}, and it is the defined behavior of strcpy() to copy until the terminating zero. In the case of strcpy(a,b), the memory after a will be overwritten if b is the longer of the two strings.

In the handlemsg() example above, the stack frame began with a return pointer. After the return pointer was pushed cmd's six-character buffer. The pointer is held within a scratch register, and is not pushed to the stack as it might be on a machine, such as the x86, with fewer registers. Thus the stack has the following contents, relative to an origin address O.

```
O     0x???? return pointer
O-6=cmd {'r','e','d',0,?,?} character buffer
```

Supposing that the string `Goodspeed' where written to the stack, it would become:

```
O-2   {'d',0}
O     {'e','e'}overwritten stack pointer
O-6=cmd {'G','o','o','d','s','p'}character buffer
```

As 'e'=0x65, the program counter would be set to the illegal value 0x6565 upon a return. (The value is illegal because it is odd-aligned.) By more carefully choosing the relevant value, code may jump to within the string itself. For example, if the string of the incoming packet were to be stored at 0x0200, the string in the code below would redirect code execution into itself.

```
{
//begin with six bytes of nonsense, to fill the
//character buffer
0xDEAD, 0xBEEF, 0xCAFE,
//follow that with the new address
0x020A,
//then fill with a zero byte and pad to even alignment
0x0000,
//follow that with executable code
0xE3B2,0x0011,
0x3FFD
}
```

In this case, the code inverts the first eight I/O pins within an infinite loop. This pattern is quite distinctive on a scope, and blinking it through LED's on P0 is an effective way of ensuring that the code is executing and that the attack was successful. Once this behavior has been verified, the attacker may replace the payload with any executable code he pleases, so long as it is sufficiently small. Larger code may be executed by using multiple fragment payloads, each of which copies a short fragment to an unused region of memory, executing that region once it has been fully populated.

It is significantly more difficult to perform an injection on a Harvard-architecture chip, such as the Atmel AVR, as the program counter must be overwritten with an address in code memory and any incoming packet will be stored in data memory. To circumvent this, executable instructions which are already present within code memory must be called with new parameters to copy data memory into code memory, at which point it can be executed. This was first accomplished for wireless sensors by Francillon and Castelluccia in [10] who constructed their exploit by an ingenious use of a modified CPU simulator.

## 3.2 Reverse Engineering

For this section, we make the reasonable assumption that firmware may be extracted from the device. Once code has been extracted from a device, it is in the form of machine language. Machine language is easily translated into assembly code, but such code contains neither comments nor--in the typical case of the smaller embedded systems, symbolic name labels. While the code is rather difficult to read without experience, the reading of code presents little challenge to a skilled reverse engineer,

particularly if that code was generated by a C compiler with no intention of obfuscation. Tools such as IDA Pro, the Interactive DisAssembler, provide excellent methods of analysis and navigation for a wide variety of common computer architectures. For more obscure architectures, it is useful to write a plugin for an existing tool or write a new analysis program from scratch. This section presents a few simple methods of analysis which the primary author implemented in a quick Perl script, one that lead to the MSP430static tool for reverse engineering MSP430 firmware. More detail on this subject is to be found in [14], which are early notes from the first revision of the tool.

Function isolation is the process of dividing a large program into its constituent functions. From the assumptions that the compiler is GCC and that standard C programming practices have been followed, three key observations can be made which are true for most, but not quite all, functions.

1. Every function that is called by another begins at an address which is the target of a CALL, and every interrupt handler begins at an address within the interrupt vector table (IVT) at the end of memory.
2. Every function ends with RET while every interrupt handler ends with RETI.
3. While a function may contain a RET statement before its end, such non-final RET statements always precede the greatest target of a relative jump.

1, 2 and 3 imply that

4. For each IVT entry and for each target of a CALL statement, one may isolate the function by the following algorithm:

```
Let X be 0.
For each instruction, in sequential ascending order
beginning at the entry address, where PC is the
instruction's address,
 If the instruction is a relative jump of any form,
 let X be the target address.
 If the instruction is RET or RETI,
 and if X<=PC,
 break.
```

Once functions have been isolated from one another, it is often trivial to identify which functions interact with particular peripherals. For example, the most common methods of setting bits on Port 1 of an MSP430 all make use of the P1OUT variable. P1OUT is a byte of memory mapped I/O, and anything written to it is written to the output pins of Port 1. In point of fact, its address is 0x21. For an example of machine code that writes to a port, see the last few bytes of crafted payload code on the previous page. All addresses as a source or destination less than 0x200 imply access to a peripheral of some kind.

A call graph is also valuable in attempting to understand the contents of a microcontroller's firmware. A call graph is a graph in which each vertex represents a function, while each edge represents a function call. As each function is deciphered, it

can be given a name which replaces its address in the graph. In this manner, a reverse engineer may identify functions by relation. For example, he might recognize the I/O functions get_byte() and put_byte() by looking for P1OUT and P1IN. Then, seeing that only one function calls put_byte(), he might quickly identify the put_string() function.

Closely related to the call graph, and nearly as useful, is the control flow graph. This graph consists of instructions as vertices, with edges representing paths from one instruction to another. For most instructions, this edge will point to the following instruction. Exceptions include branching instructions such as calls, returns, and jumps. By weighting edges with the execution delay of each source vertex, a graph can be made which can visually demonstrate the timing vulnerable described elsewhere in this document, as well as in [1].

Another diagram of interest is the memory map, which is not a graph in the graph-theory sense. MSP430static generates a memory map by the following rules. This map allows one to distinctly see a few select features, such as the bootloader of the TI EZ430 firmware as described in [15]. The horizontal axis presents the less signficant byte of the address, while the vertical axis presents the more significant byte. 0x0000 is the bottom left corner; 0x00FF is the bottom right; and 0xFFFF is the upper right.

1. An instance of 0xFFFF is black, as it is likely uninitialized flash memory.
2. An instance of 0x0000 is white, as it is likely initialized RAM memory.
3. An address referenced by a peek or a poke is drawn in blue, as it is likely a global variable of some sort.
4. The Interrupt Vector Table, ranging from 0xFFE0 to 0xFFFF, is drawn in green.
5. Any word of memory not defined by the preceding rules is drawn in red, as it is either code or data.

Note: this paper is published in black and white. The blue, green and red appear as shades of gray.

In the case of the EZ430U firmware, shown in Figure 5, the bulk of code is found as a large red block near the bottom. Beneath that code is RAM, I/O, and masked ROM. The thin red band near the top of memory is a bootloader, which allows for firmware updates over the device's USB port. MSP430static was used to determine the exact properties required of the lower firmware image, allowing a new application to be linked into the same bootloader. In the case of a control-system device, similar techniques might be used to patch the firmware of a device to add a backdoor.

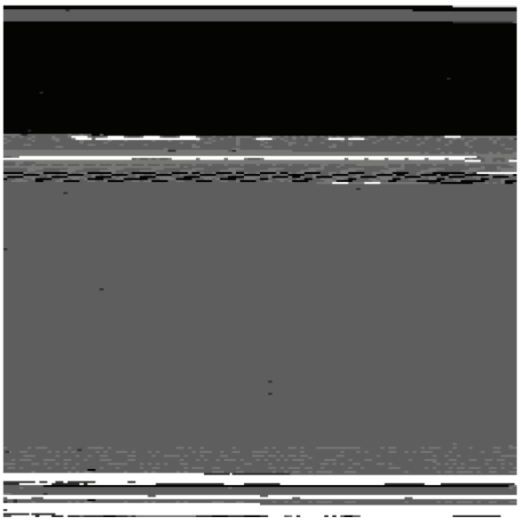

*Figure 5 – EZ430U Memory Map*

While the first attempt at MSP430static was self-contained in Perl, the second attempt used SQLite3 as a database engine. Tables are populated with code. Next tables are extracted for peeks, pokes, function calls, functions, and other such things. It is worth noting that the limited memory of the MSP430 makes it practical to do this in pure SQL, without any sophisticated optimization. The author has found little or no noticeable delay when running interactive queries. The only action which consumes considerable time, on the order of thirty seconds, is the initial import of a very large application.

Reverse engineering of firmware, of course, depends upon the availability of that firmware. This is obtained from firmware updates that a vendor releases, from unprotected memory chips on a board, or by bypassing the security of a protected or internal memory device. Commercial services exist which will extract the firmware of a microcontroller for fees ranging from a few hundred to a few thousand dollars. The techniques for extracting firmware additionally include environmental integrity attacks such as timing analysis attacks, voltage and clock glitching, and invasive attacks. Invasive attacks are beyond the scope of this paper.

## 3.3 Timing Attacks

Environmental parameters for hardware are generally assumed to remain within normative design constraints. Timing vulnerabilities present themselves in embedded systems as a result of code which would be perfectly acceptable, and perfectly secure, in a workstation or server application. A perfect example of this is a timing attack against the MSP430's serial bootstrap loader (BSL) password comparison function. Here we shall present only the technical aspects of a single, vulnerable version, leaving [1] to cover the subject in more detail.

Bootstrap loaders are a valuable target in the hacking of embedded systems because they are shared among many chips within a family, are often contained within unpatchable masked ROM, and rarely support being disabled. The security of any embedded system, particularly a wireless one with units beyond the fence-line, often depends upon the sanctity of its memory. In addition to programming a chip, bootstrap loaders also allow for the contents of memory to be read, usually with only a password or bit of EEPROM to defend against malicious access. The following example describes a method of bypassing the password protection memory of early-model MSP430 microcontrollers.

Most members of the MSP430 family of microcontrollers from Texas Instruments contain a serial bootstrap loader, or BSL, which allows a chip to be reprogrammed after the JTAG fuse has been blown. The official documentation for this feature is SLAA089 [3]. This sequence is initiated by pulsing the TEST or TCK line before raising the !RST pin to a logical 1 as is shown in Figure 6. Once entered, the BSL application acts similarly to a debugging monitor, allowing a client to read and write memory, jump to a specified address, and perform other useful functions.

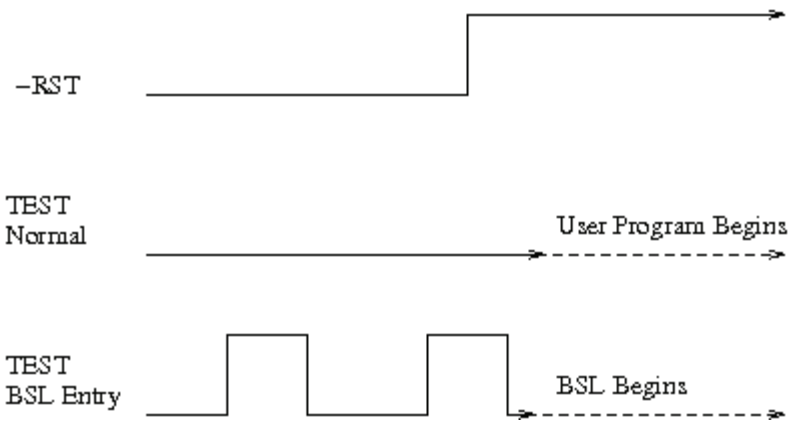


*Figure 6 – Logic Timing for BSL Entry*

To prevent unauthorized access to firmware, the BSL is protected by a 256 bit password, which is also the interrupt vector table of sixteen 16-bit pointers to interrupt handlers. Becher attempted in [2] to determine the minimum keyspace, which is to say the number of bits out of 256 which are truly unpredictable in a rather small program. Beginning with 256 bits in an array of sixteen 16-bit interrupt handler function pointers, their reasoning follows:

1. As all MSP430 instructions must be even-aligned, every valid interrupt handler must have a least-significant bit of 0. Therefore, the space is reduced to 16*15=240 bits.

2. As the reset vector always points to the beginning of flash memory, the space is further reduced to 15*15=225 bits.
3. All unused interrupts point to the same address. As a worst case would have at least four distinct handlers, the keyspace will be reduced to no less than 4*15=60 bits by this observation.
4. As code is placed in a contiguous region of memory, a smaller program would have less are in which to place interrupt handlers. Supposing the program uses only 2^11=2kb of flash memory, the key domain is reduced to only 4*10=40 bits.

Becher concluded that while the key domain was smaller than it might be, a brute force attach would take 128 years. Later work in [5] shortened this estimate to 32 years and includes a previously unpublished defensive tool by Becher, one that randomizes IVT handler addresses to increase the IVT's entropy.

This would seem to imply that code is secure on this platform, but that is unfortunately not the case. Many versions of the bootstrap loader contain a bug in the password comparison routine. A loop reads each of the password bytes individually, setting a bit in a register if a byte is incorrect. The code in C might look vaguely like the following:

```
for(i=0;i<32;i++){
 if(IVT[i]!=getbyte())
 access|=ACCESSDENIED;
}
```

The vulnerability here, which was fixed by TI and then subsequently re-introduced, is that each iteration of the loop takes longer to execute--2 clock cycles longer to be precise, when the byte being compared is unequal. Timing here is controlled only by two factors: (1) delays introduced when the receiver waits for a broadcast from the attacker, and (2) the correctness of the password. In a standard transaction, the receiver will wait for the start bit of each serial packet from the transmitter. This delay drowns out anything that might be measured by 2, as the receiver synchronizes its timing to the transmitter with each byte. By transmitting the stop bit early, however, it is possible for a malicious participant to prevent (1) such that (2) is responsible for all variances in timing.

Such an attack was implemented by the first author in the form of a handheld device. Figure 7 shows serial traffic transmitted from the MSP430 BSL in three discrete packets. The first packet, from a prior operation, is not relevant to this discussion. The second is an ACK, acknowledging the attackers timing and telling him to transmit. Between the second and third, a password is presented and compared. The third byte is a NAK, telling the attacker that the password is incorrect. The attacking device, dubbed the BSLCracker, measures the difference in timing to determine the accuracy of any given password attempt, homing in to the correct password in fewer than a thousand attempts.

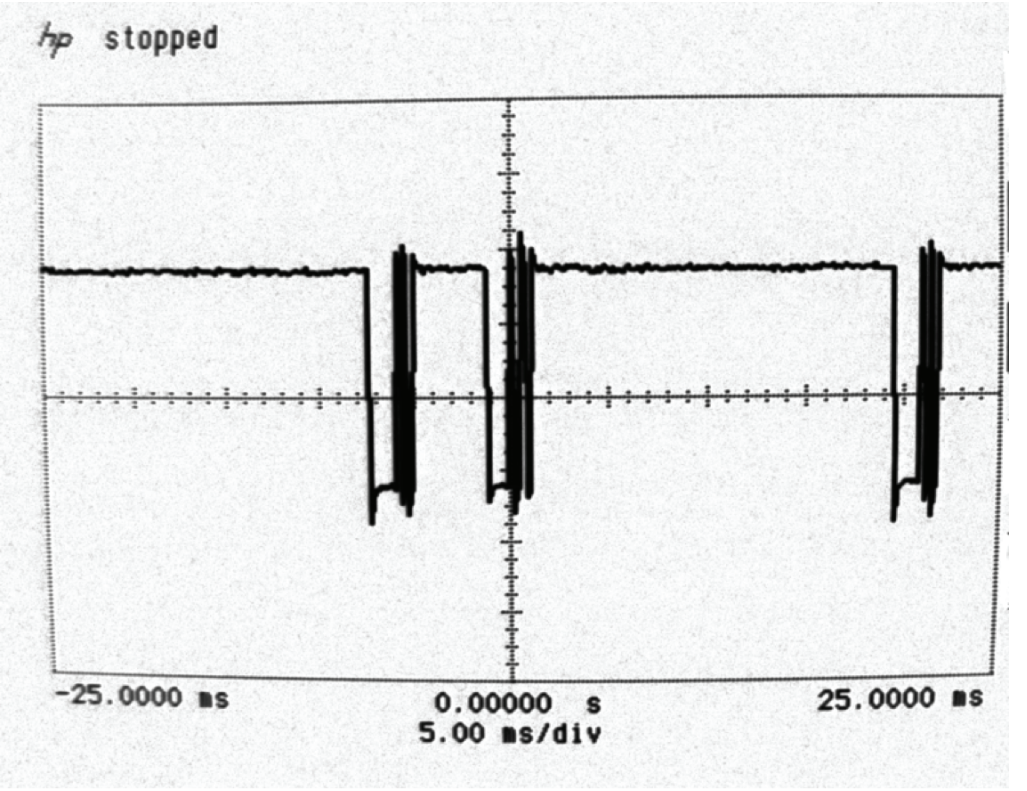


*Figure 7 – Scope View of BSL Password Timing*

Attacks of this sort impact the confidentiality of memory. Should a vulnerable bootstrap loader be used, an attacker with but a few moments of physical access can extract code, cryptographic keys, and similar information. He may also patch firmware and continue execution where it left off, such that downtime for the attack might not be distinguished from downtime due to radio interference.

It should not, however, be assumed that bootloaders are the only code vulnerable to such an attack. Nearly any password comparison whose timing depends upon the correctness of a challenge will be vulnerable when that timing is exposed, or can be exposed, to an attacker. Just as password comparisons are limited by many networked computers to a small, finite number of attempts, embedded systems ought to do the same. This is exactly what some recent versions of the MSP430 BSL do, erasing all of flash memory after a failed password attempt [3].

# 4 Hardware Attacks

## 4.1 Bus Sniffing

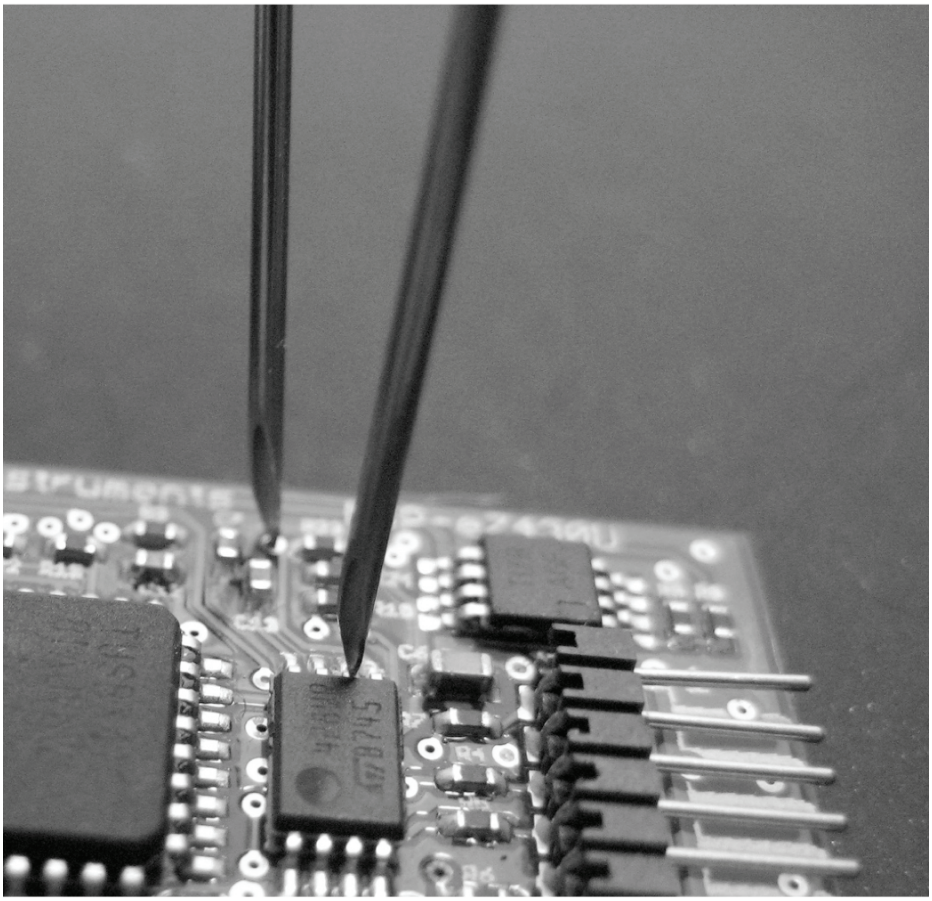

*Figure 8 – Ad-hoc 12 C Sniffing*

It is common for embedded systems to use a peripheral device--such as a radio or EEPROM--which is distinct from the central microcontroller, the two being connected by SPI or another type of serial bus. While this is a convenient and industry-standardized method of connecting chips, it is utterly inappropriate for a device which must be beyond a fence-line. Figure 8 depicts the author using two syringes and a protocol adapter to extract the firmware of an I2C EEPROM. As the EEPROM contains executable code, the author was able to replace its firmware with that of a compatible board, making this device function with Linux whereas it previously could not.

Not all ambitions are so noble, and embedded developers would be wise to ask themselves what an attacker might gain by bus snooping. Although few would be so naive as to use external boot memory in a secure device, many use the same bus to connect a radio chip to a microcontroller. Radio chip manufacturers, in the rush to be 802.15.4 compliant, have implemented AES128 internally in hardware so as to alleviate the costs of development on the part of their customers. Implementing cryptography in hardware, which would be claimed as a security feature on any datasheet, has introduced a vulnerability: traffic between the microcontroller and the radio is left unencrypted!

Thus, to sniff packets on a network, an attacker need only stick a few needles into an authorized participant. By listening as the keys are loaded, he can get a copy to place in his own radio. By injecting new packets on the insecure bus, he is able to broadcast with proper cryptography on an otherwise secure channel.

Unified chips, such as the Ember EM250, Freescale MC1322X, and TI 2480A1, alleviate this somewhat by keeping the bus within a single chip, rather than exposing the bus on a PCB. On the other hand, many designs may promote their application data up across external communications buses, not necessarily disclosing specific RF link key material, but potentially disclosing other application and application level key material. Notably, the Ember chip has hardware sniffing support with designated pins that may likely be enabled by default. In fact, design documentation recommends enabling sniffer modes and pins by default for rapid debug and analysis, giving detailed register settings required to do so.

## 4.2 Firmware Replacement

It is obvious that firmware of the CPU might be extracted and replaced, and considerable attention is given to preventing this on some platforms. However, what is much less commonly considered is bus attacks may be performed without needles by replacing the firmware of another chip which resides on the same bus. A USB to serial chip, such as the one on the Telos-B module, might have its firmware replaced, causing it to function normally except for a periodic packet selection, injection or omission. Many chips, even those which are not advertised as microcontrollers, contain an internal 8051 core to simplify design, leading to additional commodity compromise.

## 4.3 Denial of Service

Denial of service attacks are not terribly difficult to implement against the 802.15.4 hardware. Most 802.15.4 radios have test modes in which standards performance may be violated to verify basic radio and circuit properties. Even still, the standards do not need to be violated to effect coherent denial of service attacks. One which is particularly interesting involves the use of the unbuffered serial transmit mode of the CC2420, as described in [11], to constantly broadcast data. Alternatively, transmitter test modes of the chip can generate either an unmodulated carrier or a pseudo-random modulated spectrum. While such methods are often sufficient to jam an ongoing broadcast, such an action is rarely necessary as the CCA (Clear Channel Assessment) feature of 802.15.4 will prevent most devices from broadcasting in spite of even weak interference.

An interesting extension of this is an attack by Stajano, Cvrcek, and Lewis in [12]. By using the unbuffered serial receive mode of a 2420 and multiple receivers, they are able to read the header of an incoming packet before jamming the remainder of the frame. In this way, they can selectively jam only a single participant or type of request. This style of jamming, called fast following, and others are thoroughly investigated alongside defensive strategy in [13].

## 4.4 Instruction Execution Glitching

Suppose that a computer program has been thoroughly audited by hand, as well as by abstract interpretation or similar formal methods. There exists in this code no behavioral vulnerability, and there is no execution path through the code which might benefit an attacker. Such code can still be exploited by an attacker with hardware access using a technique known as "glitching". Glitching takes advantage of the misbehavior that a microcontroller exhibits when given either an irregular clock or an unstable power source. A glitcher, such as the commercial device shown in Figure 9 intentionally either drops voltage beneath the minimum required by the target device or introduces momentary noise into the clock at a precisely calculated moment in the execution of a program. Effects vary by architecture, but include the skipping of instruction phases and the misreading of memory.

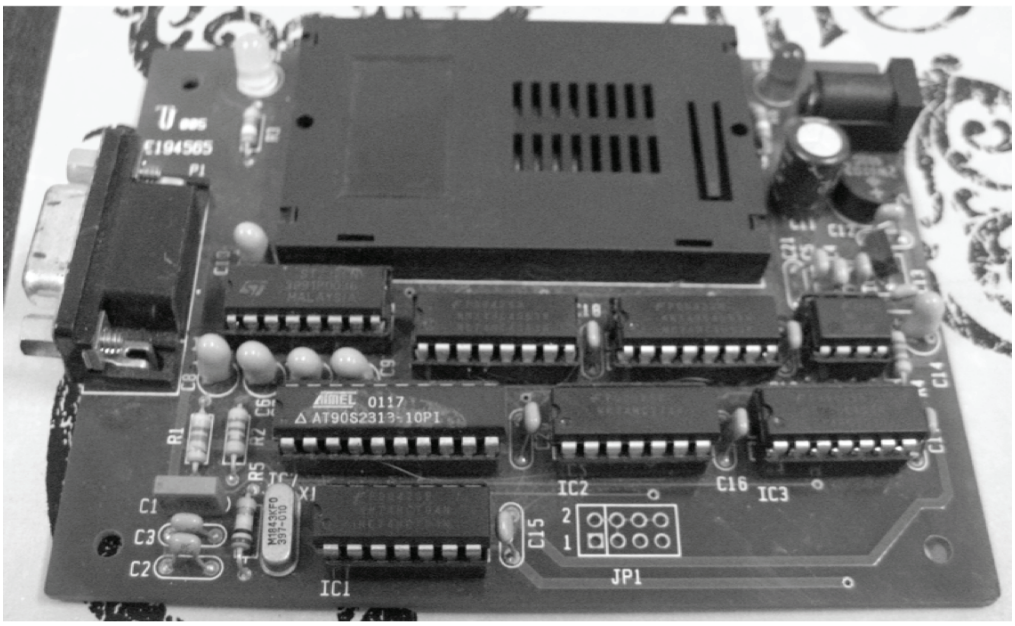


*Figure 9 – Instruction Glitching Hardware*

This timing glitcher uses 74HC4053 bi-directional analog MUX gates to switch either the clock or the power from standard values to malicious ones. Voltage may be dropped to an arbitrary analog value set by a PWM-fed ADC, and the clock can be switched between cycles to jump ahead of its stable rate. These actions, requiring lower latency

than can be attained from a PC, are performed by an onboard Atmel microcontroller. Firmware is fed to the microcontroller over the serial port, and a unique firmware image is used to attack each individual victim device.

In operation, the glitcher causes a momentary fault during the execution of a critical instruction. For example, the code in the figure below disables the serial bootstrap loader by entering an infinite loop if 0xAA55 is found in flash memory at the address 0xFFBE. This infinite loop continues until a watchdog timer resets the chip, exiting the BSL. By inducing a fault in the jump instruction, an attacker can slip out of the while() loop, continuing to the rest of the bootstrap loader as if it hadn't been disabled. By glitching just this one instruction, an attack with knowledge of a device's firmware will have gained access to any keys or individually identifying information that reside within nonvolatile memory.

```
cmp 0xAA55, &0xFFBE
jz $+0
```

Glitching began as a method of attacking smart cards, and many smart cards designs now defend against such attacks by use of onboard clocks and decoupling capacitors, to defend against clock and power glitching respectively. Smartcard firmware vendors have recently begun to implement defenses against these attacks. Microcontrollers, by contrast, have acquired internal oscillators only to simplify component placement, and it is not yet common to find them with internal decoupling capacitors. Their bootloaders implement few if any defenses against glitching, leaving them open to attack.

# 5 Conclusion

In this paper we have presented a few practical attacks against wireless control system elements which lie outside of any fence line, within physical reach of an attacker. We have demonstrated methods by which an attacker might (1) inject executable code, (2) reverse engineer firmware, (3) guess a password by timing, (4) sniff and inject frames on an internal bus, (5) replace the firmware of a secondary microcontroller, (6) jam the radio, and (7) forcibly enter a disabled bootloader of the average 802.15.4 node. In each case, devices are commonly vulnerable because they use components which are not intended to defend against attack. Components in each case have been designed to be low-power, low-cost, easy to construct, and easy to debug. Such designs have many advantages, both in terms of business and engineering, but they leave open security holes which may also require consideration for their potential impact on engineering and especially business.

The argument can be made that, in many applications, these vulnerabilities are irrelevant. These attack vectors could certainly be ignored, for example, by the maker of a low-volume children's toy. This is not the case, however, in equipment used for applications with broad impact such as for pay-television and advance electric metering infrastructure (AMI). The satellite television industry has found itself subject to increasing sophisticated attacks involving hardware attacks upon its radio receivers, and it is to be

expected that utility metering and other embedded control systems will be subject to similar attack. Most of the attacks on satellite television to date, however, have had the relatively simple motivation of theft. While theft is a very real concern for electric utilities, the power industry also carries the additional burden of being considered critical infrastructure.

The potential impact of a meter compromise on AMI presents a particularly weighty set of concerns. If an attacker were able to compromise a meter they could explore potential for extending the attack beyond the individual meter, especially if they had gained access to cryptographic material. By definition, AMI establishes a two-way communication system between the meter and the electric utility. The communications backhaul to the utility will include aggregation/distribution points affecting hundreds, if not thousands of meters. Depending on the system architecture this backhaul may involve the meter being directly meshed to a network of thousands of other meters. Ultimately the backhaul will lead to the utility offices where the system as a whole communicates with and controls potentially millions of meters.

The functions this network will perform include (but may not be limited to) usage/billing information, firmware upgrades, pricing/emergency signals designed to impact electric system load, and potentially the remote operation of a disconnect/service switch in the meter. At the same time, the physical location of these meters also means that (1) potential attackers will be in close proximity or direct physical access to the target for indefinite periods of time, and (2) deployment of these meters for a large utility can span multiple years and involve hundreds of millions of dollars.

The design of the bulk electric system is such that large and sudden shifts in load can cause destabilization of the grid with a tendency toward a cascading effect, as was the case in the August 14th, 2003 Northeast Blackout. If an attacker were able to simultaneously affect load at a large number of meters, the system could be at risk of a large-scale blackout with no easy remedy to regain confidence in load control. While such a scenario is a significant extrapolation from the tangible evidence presented in this paper, the lesson bears consideration: the security of a system depends upon the security of its components. Cyber attacks are about leveraging a small vulnerability into something much more significant. We must protect the small if we wish to enjoy the large.

---

**About the Authors** – Travis Goodspeed is an independent embedded systems consultant from Knoxville, Tennessee and a frequent speaker at information security conferences. He enjoys prototyping his MSP430 password cracker, reverse engineering, design optimization of 1970's-style television games, and electronic belt buckles.

Darren Highfill is the Utility Security Practice Lead for EnerNex Corporation. Darren is the Chair of the AMI-SEC Task Force and UtiliSec Working Group under the UCA International Users Group, leads the government/private-sector collaborative AMI Security Acceleration Project, serves as the Principle Investigator and Program Director for the DOE-funded Lemnos Interoperable Security Project, and is responsible for all aspects of security for Consumers Energy AMI project. Darren has also managed EnerNex's support of the Tennessee Valley Authority for several years and is one of the system architects for the PowerWAN – TVA's wide-

area IP-communications network. Darren is a Certified Information Systems Security Professional (CISSP).

Brad Singletary is a consultant with EnerNex Corporation, operating out of Knoxville, TN since May of 2007 and has consulted for Consumers Energy since approximately August of 2007 on topics spanning HAN, Security, and GIS issues for AMI. Brad consults for Consumers Energy as a HAN Architect designing and evaluating home area network (HAN) technology capability for the energy-managed home of Michigan's future. Brad holds a M.S. in Computer Science from Georgia Institute of Technology and a B.S. in Computer Science from East Tennessee State University. Additionally, Brad is a Certified Information Systems Security Professional (CISSP).

## References

[1] Goodspeed, A Side-Channel Timing Attack of the MSP430 BSL, Black Hat USA 2008

[2] Becher et all, Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks, SPC 2006

[3] Texas Instruments, SLAA089 -- Features of the MSP430 Bootstrap Loader

[4] Goodspeed, MSP430 Buffer Overflow Exploit for Wireless Sensor Nodes, Toorcon 2007

[5] Goodspeed, MSP430 BSL Passwords: Brute Force Estimates and Defenses

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

[7] IEEE 802.15.4-2003

[8] Farahani, S. (2008). Zigbee Wireless Networks and Transceivers. Newnes.

[9] Proakis, J. and Salehi M. (2007). Digital Communications, 5th. ed. McGraw-Hill

[10] Francillon and Castelluccia, Code Injection Attacks on Harvard-Architecture Devices, CCS 2008

[11] Texas Instruments, CC2420 Datasheet

[12] Stajano, F., Cvrcek, D., and Lewis, M. (2008). Steel, cast iron, and concrete: Security engineering for real world wireless sensor networks. In S.M. Bellovin et al. (Eds.): ACNS 2008, LNCS 5037, pp. 460–478, Springer-verlag.

[13] Wood, A. D., Stankovic, J. A., and Zhou, G. DEEJAM: Defeating energy-efficient jamming in IEEE 802.15.4-based Wireless Networks

[14] Goodspeed, Static Analysis of MSP430 Firmware in Perl

[15] Goodspeed, Repurposing the TI EZ430U, Part 3