nccgroup

Matasano And ISEC Interns Summer 2014

Internet of Things Security

June 29, 2015 – Version 1.0

Prepared by

Brian Belleville (bbellevi@uci.edu)
Patrick Biernat (biernp@rpi.edu)
Adam Cotenoff (acotenoff@isecpartners.com)
Kevin Hock (kevin.hock@stonybrook.edu)
Tanner Prynn (tannerprynn+iot@gmail.com)
Sivaranjani Sankaralingam (sivarans@andrew.cmu.edu)
Terry Sun (terrynsun@gmail.com)
Daniel Mayer (daniel@matasano.com, Manager)

Abstract

The Internet of Things (IoT) is an emerging phenomenon where different kinds of devices that were previously not networked are being connected to networks. Examples include network connected thermostats, light bulbs, and door locks. These newly networked devices present additional attack surfaces, and due to the ad hoc nature of their implementations, many do not follow current security best practices. We assessed the security of several currently available IoT devices targeted at consumers. We considered all user-facing interfaces and all networking components to be in scope of our investigation, and evaluated the devices for common security vulnerabilities. All of the devices we investigated had numerous exploitable security flaws. We discuss in detail the vulnerabilities and the processes used to discover them.

Table of Contents

1 Introduction

Internet of Things (IoT) is a broad term encompassing a wide range of embedded devices that interface with or are controlled over the Internet. By combining the versatility and affordability of embedded microcontrollers with the convenience of Internet access, these devices aim to bring increased connectivity to our everyday lives. The emergence of the IoT market is relatively new, but its popularity has been expanding quickly and is expected to continue growing.

Typically, Internet of Things devices relies on low power, local wireless networks that allow a number of devices to communicate. Frequently, these devices are also connected to a hub, which acts as a bridge to the outside world. Users will generally only interact with a hub, often via a web interface, which translates and transmits their commands to each individual device.

Particularly interesting to us are home automation and security systems. These systems are meant to give users fine-grained control over their homes, regardless of where the users may be. This includes integrating and automating door locks, light bulbs, motion sensors, thermometers, cameras, and other similar devices. With such obvious implications for users' privacy and security, these systems are bound to become a magnet for malicious activity. In this paper, we take a closer look at how they operate, as well as their overall security.

What we found was concerning: although most devices attempt to protect their users, none of the devices we looked at did so successfully. We identified vulnerabilities which allow malicious actors to open door locks, disable motion sensors, lie about the status of various devices, and in the general case, use home automation systems to the detriment of their owners.

Organization of the Paper: In Section 2 we briefly introduce the protocols these IoT devices utilize. Section 3 presents a detailed overview of the systems we investigated. We describe the procedures used to examine each device in Section 4 and include a concise listing of the vulnerabilities we uncovered in Section 5. Finally, Section 6 identifies areas we were unable to research but believe would be worth investigating in the future.

2 Wireless Protocols Overview

In this section we provide a brief overview of the different wireless protocols that are commonly used for communication between devices in a low power wireless network. These protocols allow devices which are not connected directly to the Internet to join the Internet of Things.

2.1 IEEE 802.15.4

The 802.15.4 protocol is a standard published by the IEEE for low power, low data rate, wireless networks. It defines the physical layer (PHY) and medium access control sublayer (MAC) upon which many other networking standards are built. All of the following protocols discussed in this section are built upon the PHY and MAC layers specified by IEEE 802.15.4. This will only provide a brief overview of the features of 802.15.4 relevant to our research, see *IEEE Standard for Local and metropolitan area networks - Part 15.4* [IEE11] for the complete specification.

The IEEE 802.15.4 standard supports both star and peer to peer topologies. In the star topology, communication is established between devices and a central controller acting as the network coordinator. This topology is appropriate for systems needing a simple network such as home automation or PC peripherals. In the peer to peer topology there is still a node which acts as a network coordinator, but any two nodes are able to communicate with one another. This setup allows more complicated networks, such as mesh networks, to be established.

IEEE 802.15.4 specifies security services providing data confidentiality, data authenticity, and replay protection. This is controlled on a per packet basis, with each packet specifying what security level it was sent with. The different security levels allow the device to choose to enable encryption in combination with a message authentication code of varying lengths. If used, the message authentication code authenticates both the frame header, which is sent in the clear, and the payload which can be encrypted.

Replay protection is provided by a frame counter field in the auxiliary security header. This is a four byte value that is incremented with each message a device sends. The frame counter serves two purposes. When a device receives a packet it compares the frame counter of the packet to the most recently received frame counter for that source. If the received frame counter is less than or equal to the most recently received frame counter, the message should be ignored. This protects a device from responding to a replay of a packet. The frame counter is also used to construct the nonce used to initialize counter mode encryption. This ensures that the nonce is unique per message, and that if you attempt to circumvent replay protection by sending a packet with a new frame counter, but the same cipher text, the cipher text will not be properly decrypted, and the replay attack will fail.

2.1.1 6LoWPAN

IPv6 Over Low Power Wireless Personal Area Network (6LoWPAN) is a standard for compressing IPv6 packets within 802.15.4 data frames, and is specified in [Int11]. Its advantages include easy connectivity to other IP-based devices, the ability to use existing network infrastructure, and a widely used socket API [SB11].

2.1.2 ZigBee

ZigBee is a wireless networking specification built on IEEE 802.15.4, designed for low power and low date rate communication. It takes advantage of the network capabilities of IEEE 802.15.4 to transmit data through mesh networks that are often large and sparse, while adding many helpful features that make implementing a wireless network easy. ZigBee sees wide use in the automation market, and defines several application communication standards such as building automation, home automation, smart energy, and healthcare [Zig08].

ZigBee is a standard that many manufacturers take part in, so by design it is interoperable between devices from different manufacturers. It also adds its own security and application features which manufacturers can

choose to use depending on the market they are designing their products for, including encryption and authentication.

2.1.3 Z-Wave

Z-Wave is a proprietary, low power wireless communications protocol that targets the home automation market. The specifications, as well as the SDK, are not publically available. In addition, relatively little research exists about the protocol, despite the fact that it is widely used by the home automation market.

Like ZigBee, Z-Wave uses a mesh network topology to maximize reach and connectivity. By design, most battery operated end devices are only required to be active for short bursts of time to maximize uptime, while devices with permanent power sources act as relays. This model focuses on reliable, low bandwidth data delivery. Individual networks are distinguishable by 32-bit ''Home ID's,'' and devices are internally differentiable via 8-bit ''Node ID's.''

2.2 Common Tools

ZigBee In order to sniff ZigBee traffic, we used an ApiMote v4 board, which can both capture and inject packets. The board was built for use with KillerBee, an open source suite of tools built for ''exploring and exploiting the security of ZigBee and IEEE 802.15.4 networks'', and includes utilities that can sniff, decrypt, analyze, and replay traffic.

Z-Wave Overall, there is very little public research into Z-Wave. In an attempt to intercept Z-Wave traffic, we made use of a TI CC1110DK-MINI development kit. The Z-Force framework is a set of tools designed to run on this kit in order to investigate and attack Z-Wave networks. The development kit contains two chips capable of receiving and transmitting on Z-Wave frequencies. These chips support a range of modulation options which should enable a user to manipulate Z-Wave traffic given the correct settings.

3 Scope

Many of the devices we investigated shared a common paradigm. These systems have a hub that serves as the ''master'' device, extends control to the endpoint devices, and keeps track of the network's state. The hub maintains communication to local devices, as well as a remote server owned by the manufacturer. This allows users local and remote control over their home though a web interface or smartphone application. These easy to use systems make provide a compelling way for consumers to upgrade to a fully connected home.

In the following sections, we describe which systems and devices we assessed during the course of our research.

3.1 Lowes Iris

Researchers: Adam Cotenoff

Lowes Iris Home Management System is a collection of devices encompassing home automation, security, and monitoring. It can be controlled via mobile and web applications, allowing users to arm and disarm the alarm, change the temperature, or monitor their home energy usage. This kit can communicate over both ZigBee and ZWave, but during our research we focused on the ZigBee protocol.

The specific Iris kit we tested was the Iris Smart Kit which included:

- Iris Hub - the central hub for all Lowes Iris products
- Motion Detector - sets off alarm if someone walks by
- Contact Sensors - sets off the alarm if separated; used on doors and windows
- Range Extender - extends the range of the Lowes Iris network
- Keypad - arms and disarms the alarm
- Smart Plug - controls and monitors energy usage of anything plugged in
- Smart Thermostat - controls and monitors the temperature

During testing, we explored the Iris system at the hardware, software, and network level for vulnerabilities. We did not actively test the Lowes cloud infrastructure.

3.2 SmartThings

Researchers: Patrick Biernat, Kevin Hock, Tanner Prynn

SmartThings infrastructure connects consumer-owned wireless devices to the SmartThings cloud to allow easy management from the web or from mobile applications. We tested the security of the SmartThings devices including the SmartThings Hub. The hub acts similarly to a WiFi router for ZigBee and Z-Wave home automation devices by relaying messages back-and-forth to the Internet. During our testing we analyzed the hub at both the hardware and networking levels in order to find vulnerabilities, along with mobile applications on Android and iOS. We did not directly test the SmartThings cloud infrastructure.

In the course of our testing, we tested the following SmartThings-compatible devices:

- SmartThings Hub
- SmartThings Multi (Contact, Temperature, Humidity, Acceleration sensor)
- SmartThings Arduino Shield

- Kwikset 910 Z-Wave Door Lock

3.3 TCP Connected

Researchers: Brian Belleville, Terry Sun

The Connected by TCP lighting system is a system for wirelessly controlling home lighting. It is sold under the TCP brand, but appears to be designed and manufactured by Greenwave Systems.

The lighting system consists of a gateway, a physical wireless remote control, and up to 250 light bulbs. All components communicate through an IEEE 802.15.4 wireless network that is set up by the gateway. The gateway also provides the primary interface for users to access the system. Users can communicate with the gateway using either an Intranet web application or a mobile application. Additionally, the wireless remote control can broadcast messages to light bulbs in its vicinity. The web application, mobile application, and remote all allow the user to control the light bulbs in the network.

The mobile application has two modes of operation. If the user is connected to the same local network as the gateway, the mobile application will send requests directly to the gateway. If the user is not on the same local network, but has Internet access, the application will communicate with a web server owned by Greenwave Systems, which then relays the commands to the gateway. In order to enable communication through the public Internet, the user must first create an account and associate it with their specific gateway, which can only be done while connected to the same local network as the gateway.

The scope of our investigation included all functionality of the gateway, remote, and light bulbs. We examined the gateway's communication with GreenWave's remote server, but did not attempt to attack the server itself. We considered the mobile applications to be a minor part of the project, doing only a brief investigation with respect to its interaction with the rest of the system and basic security principles.

3.4 HAI MicroControl

Researcher: Sivaranjani Sankaralingam

Home Automation, Inc. (HAI) is a manufacturer of home automation devices which was acquired by Leviton in 2012. Leviton still manufactures some devices under the HAI brand. We received and tested a HAI MicroControl, which is a standalone ZigBee controller. To test the MicroControl's security, we connected a Kwikset ZigBee Lock. Our research on the HAI MicroControl focused on its ZigBee implementation. We did not test the HAI mobile applications that are designed by Leviton.

4.1 Lowes Iris

The Lowes Iris system can be used as a home automation, home security, and home monitoring system. This research focuses on the home security system, which includes a motion sensor, keypad, and contact sensors.

ZigBee Communication We started our investigation of the Iris system with the ZigBee protocol it uses to communicate between devices. We sniffed ZigBee traffic using the ZigTools framework [War], which provides utilities for sniffing and analyzing 802.15.4 traffic. Using this tool, we were able to identify and dissect the ZigBee network processes such as association, dissaociation, and key exchange. However, this framework does not include any tool to replay or inject ZigBee packets. For this, we utilized an ApiMote (see Section 2.2). The ApiMote runs the KillerBee software [LLC], which enabled us to craft our own ZigBee packets. ZigTools indicated that the Lowes Iris traffic specifically communicated over channel 25.

One of our main goals was to disable the security system, so we attempted to jam ZigBee communication on channel 25. When the motion sensor is activated and the alarm is set, the motion sensor operates much like a heartbeat, regularly sending packets indicating a normal state. When someone does walk past the motion sensor, it sends a data packet informing the hub, which triggers an alarm. We tried flooding the hub with packets indicating that no one had walked past the motion sensor, but we were unable to stop the alarm from triggering.

Iris Hub Hardware ZigBee encrypts packets and uses a message integrity code (MIC) to prevent tampering. The main issue we faced in replaying packets was that there appears to be an application layer mechanism which denies replayed packets. We wanted to get a better understanding of this, so we looked into dumping the firmware from the Iris Hub. We looked at the hub's circuit board and investigated the hub's ZigBee chip and flash memory chip. The Iris hub is based off of the Ember EM260 chipset. Unfortunately, this chip can only be debugged with an Ember specific debugging device. We attempted to utilize a Bus Pirate to interact with the memory chip's SPI pins, but we were unable to dump the memory.

Iris Web Application We also took a cursory look into the web application, and found it vulnerable to cross site request forgery (see Section 5.1.3). CSRF is a vulnerability that makes it possible to trick users into performing authenticated actions via a malicious request. By submitting our own request, we were able to disarm the alarm. We did not perform any active testing on the web application since it is hosted on Lowe's infrastructure.

Lowe's Mobile Applications Additionally, we looked into the mobile applications. Using idb, ''a tool to simplify some common tasks for iOS pentesting and research'' [May], we extracted all the plist files and saw that there was one that contained a list of internal server addresses (see Section 5.1.6). We also decompiled the Android APK and found similar infromation in an XML file. This file contained a list of what seemed like internal staging and development servers. We did not perform any testing on these servers, as it was out of the scope of this research.

4.2 SmartThings

SmartThings is a home automation company which was recently acquired by Samsung. Its system aims to make it easy for users to automatically control their homes, enable developers to build their own applications on top of the SmartThings platform, and to bring compatibility between its own devices and other manufacturer's devices. To that end, SmartThings has mobile applications, a web interface, and a hub which supports both Z-Wave and ZigBee.

4.2.1 Hardware

We began our investigation of SmartThings from the ground up, with the hardware. Understanding a device at the hardware level reveals the functions the device performs and allows an attacker to identify where they can intervene in order to impersonate or even take over the device. Before connecting our devices to the outside world, we inspected them in an offline environment, because we wanted to gather as much information as possible before connecting to the network. In the worst case, connecting to the network could trigger an automatic firmware update, patching security holes and making our job significantly more difficult.

Information Gathering and Device Identification We started with a teardown of the SmartThings Hub. A simple USB Microscope allowed us to inspect the chips up close, and identify their markings for later research. A crucial step in reverse engineering hardware is information gathering, but chip identification can be made very difficult on high-security devices. A motivated enough manufacturer dedicated to obscuring the chips they use means resorting to x-ray or decapping to glean information. SmartThings does not have any hardware obfuscation of this sort, and datasheets were readily available on their respective manufacturers' websites.

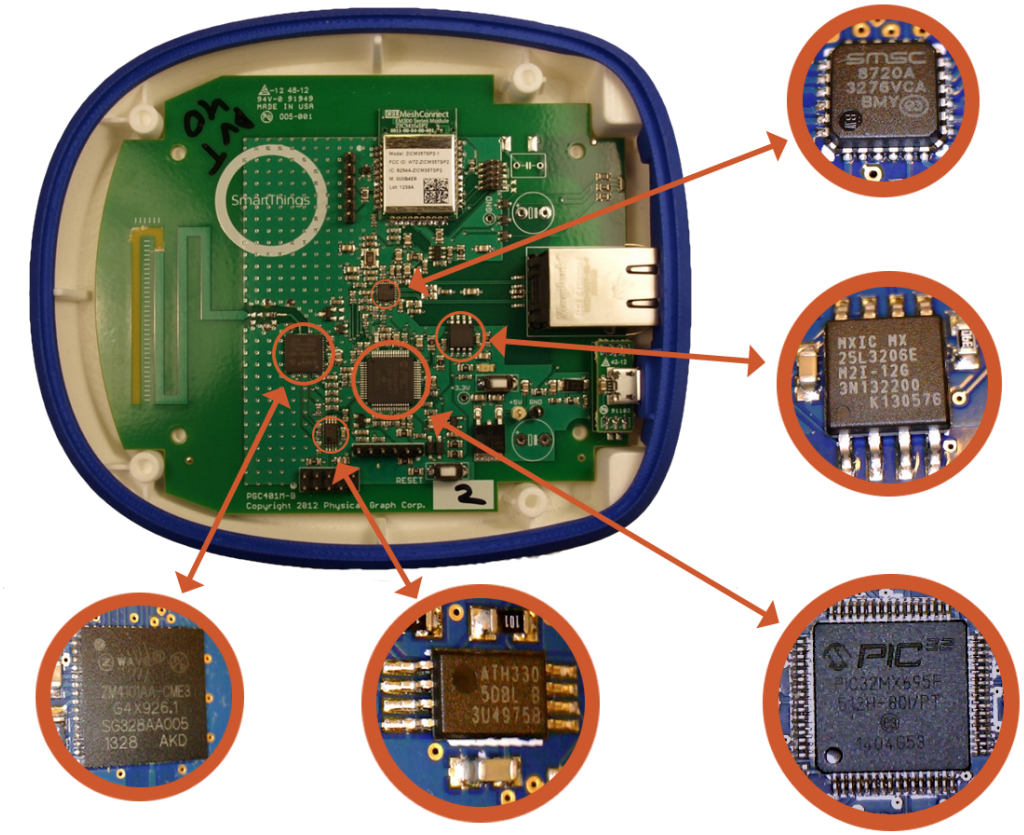


Figure 1: Clockwise from top right:
- SMSC Ethernet [SMS12]
- MXIC EEPROM [Int10]
- PIC32 Microcontroller [Tec13]
- Atmel EEPROM [Atm14]
- Z-Wave Transciever [Des12]

Hub image from SmartThings FCC documents [Gra].

The SmartThings devices all share an Ember EM357 module [Cel13], which runs its own firmware to communicate over ZigBee networks. The Hub has a full 32-bit PIC processor which coordinates its Ethernet, memory, ZigBee, and Z-Wave devices; the Multi has an accelerometer and humiture sensor but appears to offload its processing to the Ember chip. Both the Hub and Multi have groups of test points which fit standard headers - the FCC pictures even show these headers soldered in (see figure 2). In order to interact with these devices we soldered headers in to these test points.

The datasheets for the PIC and the Ember chips show that they support JTAG, a serial protocol that embedded devices often use for administration, testing, or recovery. Our first attempt at interacting with the Hub on a hardware level was using a JTAGulator, a device that connects to test points on a target board and automatically tries each possible JTAG pin configuration. If JTAGulation is successful, a valid JTAG configuration is found which could give a backdoor into the device or even allow us to dump its firmware. However, we were unable to find JTAG functionality on any of the test points we tried (labelled A, C, and D in figure 2).

Pin Tracing After the automatic identification of the JTAGulator proved unsuccessful, we proceeded by using a multimeter to manually trace the chips' pins to the test points on the board. The resulting traces are presented in Appendix B.1. Among other things, we discovered that the JTAG pins were connected to a seperate pad with a small custom header (labelled B in figure 2) rather than the standard test pins that we had soldered in.

Figure 2: The four test headers on the SmartThings Hub. Hub image from SmartThings FCC documents [Gra].

Logic Analysis Based on our research with the chips' datasheets, we discovered that there was a lot of ambiguity in which pins could be connected where and for what purpose. Some pins have up to four different functions depending on the state of the chip. Further, the chips are multipurpose and can be configured for different connection options: 1 or 2-wire serial, UART, JTAG, SPI, or even custom protocols. Additionally, many of these protocols have the option to use extra wires for more features, or faster communication; and have tweakable settings such as baud rate, endianness, check bits, etc. To comb through this mess we turned to logic analysis. A logic analyzer is like a cousin to an oscilloscope, and makes the process of debugging digital logic much simpler.

Using a logic analyzer (a Saleae Logic16) we realized that the two memory chips on the Hub both respond to the Serial Peripheral Interface (SPI) serial protocol. SPI is a well-known protocol which uses four wires: a clock wire, data in, data out, and chip select. The chip select wire is ued to make controlling multiple devices with one microcontroller cheap and efficient. The same clock and data lines are connected to every slave device, while a separate chip select line is run from the master to each slave. When talking to a specific device, that device's chip select line is pulled low and only it will listen or respond on the data lines. In this way, it takes three wires plus one additional wire for each slave device, so connecting and controlling a lot of chips is easy. Rather than needing special hardware to talk SPI, we used the built-in SPI capabilities of Arduino, the swiss-army knife of hardware hacking. These pins are labeled SS, MOSI, MISO, and SCK on our Arduino Mega ADK.

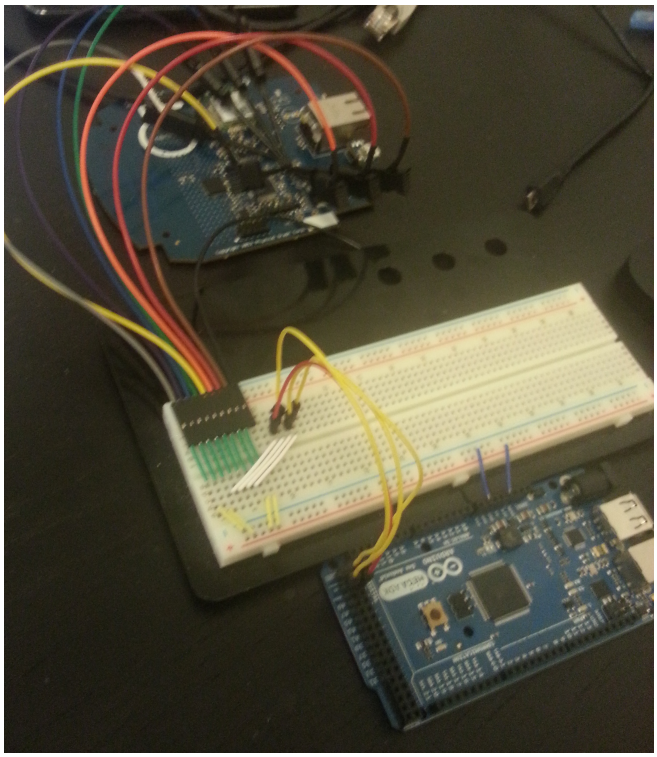


Figure 3: The Arduino wired to the SPI pins of the MXIC EEPROM.

Serial Protocols and Memory Retrieval From the traces and datasheets, we knew the correct wiring for the two memory chips. The manufacturers' documentation explains how a simple read memory command can be written over SPI, and the chips will respond with their memory contents. This process can be done in one of two ways: either the chip can be desoldered and removed from the board, or the board can be powered and commands issued to the chips while the board is running, with pins attached to the headers. The first method is potentially destructive, so we chose to use the second. There is a problem with interacting with the device while it is still attached to the board (known as *in-circuit*), however: the processor may attempt to access those chips at the same time, causing serious issues. Fortunately, as long as the hub is not connected to the network, it does not seem to use the chips, so we were able to read their memory. Unfortunately, it did not seem to have any useful information, being mostly filled with blank or unintelligible data. There were no locations in the memory dump that were readable by us as data or code.

The second header we looked at on the Hub is connected to the PIC32. Our suspicion was that this header was a PIC programming header. There are six pins in this header, two of which are connected to the PIC

through 40Ω resistors, which PIC specifies in its design documents. In an attempt to pull the firmware from the Hub, we purchased a PIC programmer, a small, $60 USB device which plugged directly into the headers we soldered onto the board. PIC's software recognized the processor immediately. However, attempting to read the firmware failed as the device has its code-protection bit set. Code protection means that the code running on the device is not readable externally - it is a feature designed to prevent copycats from making a clone of the device. This setting can't be turned off without completely wiping the device.

Ultimately, we came to a standstill on the hardware side. Despite digging deep into the workings of the SmartThings Hub, we came away without any large vulnerabilities. We discuss future areas of research in Section 6.2.1.

4.2.2 Networking

In order to gather as much information as possible about how the Hub uses its Ethernet connection, we set up a man-in-the-middle (MITM) between the Hub and the SmartThings servers. We created a disconnected network between our MITM machine and the Hub, and found that the Hub immediately attempts to connect to a SmartThings server over SSL. We set up our machine with DHCP and DNS servers, initially routing all requests to a local Apache server. The Hub will only communicate over SSL, so we created our own certificate authority and used it to sign a certificate for the SmartThings site. Those fake certificates were loaded into Apache, which was configured to accept any cipher suite, and the Hub accepted the SSL handshake. As it turns out, the Hub will accept any certificate, so it was not necessary to duplicate the SmartThings certificates - a self-signed certificate would have been accepted in the first place. Note that a secure way for accessing a known endpoint on an embedded device is to pin known-good certificates in the hardware, making man-in-the-middle attacks impossible.

We exploited the lack of certificate validation by writing a python script which acted as an intercepting proxy between the Hub and the server. Using this MITM script, we started gathering packets. We quickly discovered that SmartThings has its own protocol for communication between the Hub and the server. They communicate by sending hex data in raw SSL packets.

Protocol Reversing Some time spent dissecting these packets eventually allowed us to split the packets apart into recognizable pieces. Note that this packet format has changed in newer software revisions. We looked for pieces of the packets which were constant, predictable, or seemingly random. These packets all have a similar structure. They begin with a preamble which includes metadata such as packet length, command type, MAC address, and sequence number. Following this is an optional payload which contains command-specific data and a mandatory checksum, discussed in more depth in Appendix B.2. One thing to note is that there is no authentication in the requests - only the MAC Address is used to identify the sent packets. As long as an attacker knows the MAC Address of a Hub, they can impersonate it. This is not the case in the newer version of the Hub firmware, which prepends a header that appears to authenticate the Hub which is sending the packet. However, the server will still accept the older and less secure format. Details of both protocols are in Appendix B.2.

Hub Impersonation Due to the lack of authentication in the older protocol, we were able to impersonate both the Hub and the server. We also found that the both versions of the SmartThings protocol are vulnerable to replay attacks. The ability to impersonate a hub gives an attacker the ability to send fake notifications; for example, an attacker could cause a user to believe their door is locked when it is not. The impact of forged notifications is dependent on the attacker's knowledge of their target. Hubs are identified by the MAC address of their Ethernet chip, and the range of MAC addresses for the SmartThings Hub is `D0:52:A8:00:00:00` to `D0:52:A8:FF:FF:FF`. This range is slightly large to brute-force in a targeted attack, but in many cases, it is not

difficult to obtain the MAC address of the SmartThings Hub.

An additional caveat for this attack is that a valid packet payload needs to include the network ID of the specific device that would cause the notifications. Bruteforcing these is less feasible, as the numbers involved are quite large (32 bits). However, in our experience, these ID's tended to be relatively small, static, and not randomly generated. Combined with the lack of hub authentication, this should allow attackers to add fake devices to users' accounts, which may open the door to further attacks. Since testing this further would include interacting with the SmartThings infrastructure, such testing was not performed.

Server Impersonation In order to impersonate the server, an attacker will need to control the routing of the Hub's requests, or the DNS that the Hub asks to resolve. This attack vector is limited, because generally the SmartThings Hub will be attached to a trusted network (i.e. the user's router). However, given local network access, it is likely that an attacker will be able to control DNS, either via ARP spoofing or a vulnerability in the victim's router. In this scenario, an attacker could gain control over the router and change its DNS or routing settings to point at their own server. Oftentimes, home routers are poorly configured, running old, vulnerable firmware, or even accessible from a public IP address, so we consider this a practical attack. Once an attacker is able to exert control over the network, they will then be able to impersonate the server, due to the Hub's lack of certificate verification. At this point, the attacker has total control over the networked devices, such as the victim's door locks.

Firmware Dumping Once confident that we could intercept and parse packets, we looked at the firmware update process to see if we could pull the firmware while it was begin downloaded by the hub. Our hope was that we would be able to reconstruct the firmware from the packet log, as an alternative to reading it out from the hardware. We initiated a firmware update from the Android application, and allowed the server to send a firmware update command (command 16, Appendix B.2) to the hub. The hub responded with an acknowledgement packet (command 17) before the server sent the first firmware packet down. We stopped the firmware update process after allowing a few packets through to the hub. The firmware packets are split up into 128-byte chunks (command 18), and after the hub receives a chunk it will acknowledge it (command 02). So, by allowing the hub to connect and taking over after a firmware update command was sent, we could iterate through the firmware chunks until the entire firmware was downloaded. We then patched all the chunks back together to get a firmware blob.

The final stepping stone in the process was to inspect and reverse engineer this firmware. The first tools we turned to were Unix's `file` and `strings`, neither of which turned out to be helpful. The big gun of firmware analysis is `binwalk`, which again was unable to make inroads in our firmware file. It suggested that our file was encrypted, which was backed up by an entropy analysis of the file (figure 4).

A file with high entropy is likely to be either encrypted or compressed [/de13]. In an encrypted file, each bit is approximately random, which means the file will have uniformly high entropy. Compressed files, while having higher entropy then an uncompressed file, are often not uniform or have lower entropy then an encrypted file. Our firmware blob did not match a compressed file format and had very high uniform entropy, so we concluded that the Hub's firmware update was encrypted. In order to reverse the encryption, we would need to know the encryption algorithm as well as the encryption key. However, it is likely that both of these are stored on the PIC, inaccessible to us. The alternative is to allow the firmware to be decrypted by the Hub and then read it back, but it is not possible to externally access the PIC's memory. We concluded that we are currently unable to decrypt the Hub's firmware.

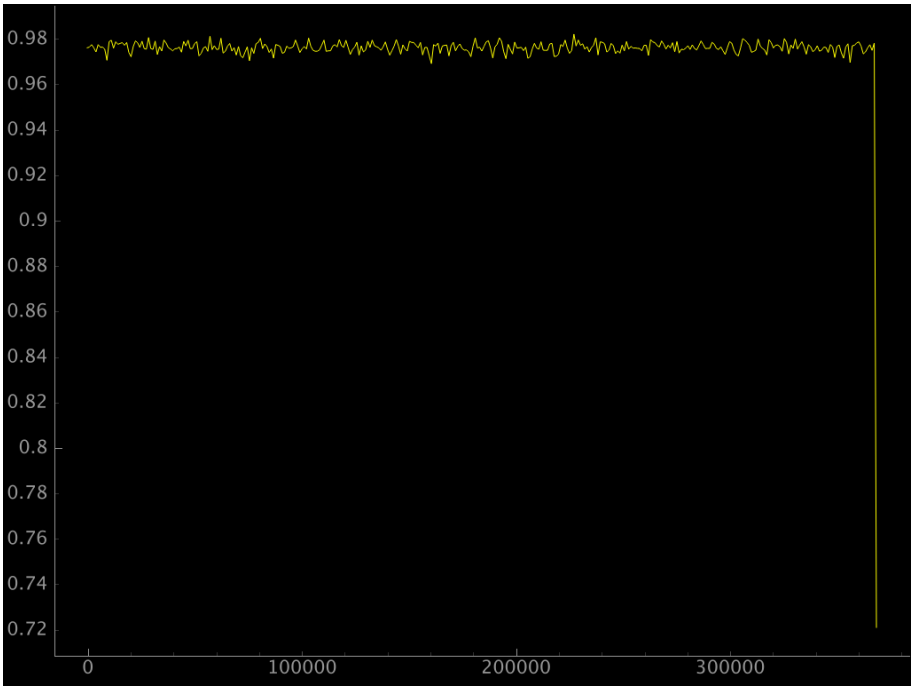


Figure 4: Output from `binwalk`'s entropy analysis shows the firmware to have very high entropy.

Z-Wave In addition to ZigBee, the SmartThings Hub can also communicate using Z-Wave. When trying to look for hardware to listen to Z-Wave, the Z-Force framework (see Section 2.2) looked promising. We began by flashing the Z-Force firmware onto a TI-Dev board, but unfortunately, we could not intercept any traffic. After getting in contact with the authors, we learned that the firmware was hardcoded for the EU (868 MHz) frequency. We managed to find settings which allowed us to view radio waves being sent from the Hub and Lock, but had trouble converting them into data we could manipulate.

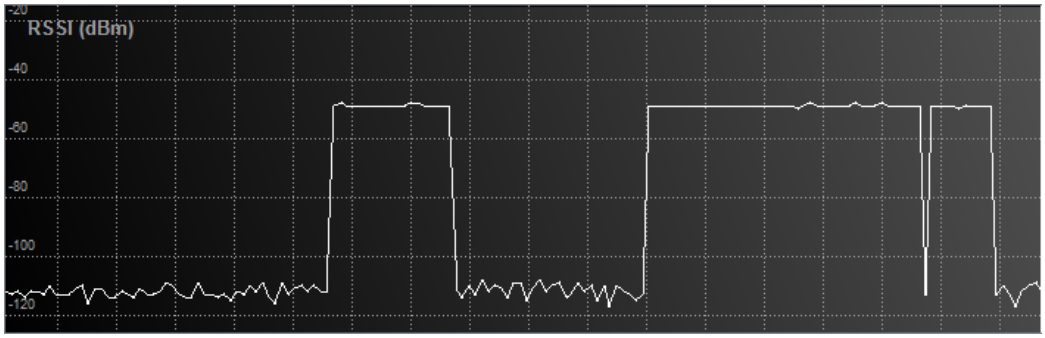


Figure 5: The ''unlock'' sequence sent from the Hub to the Kwikset Lock, using Z-Wave.

Telnet Server The last piece of wired network functionality exposed by the Hub is a Telnet server. Insecure telnet servers on embedded devices are very common. We started out by trying some default passwords, but didn't find one that worked. Next, we fired up THC-Hydra with a short password list, but the Hub only allows a single connection at a time and Hydra had some problems dealing with being disconnected. Finally, we wrote a small ruby script to run through a list of 1000 common passwords with no success. The main problem is that the Hub can take three to five seconds until it will accept a new password guess after a failed attempt, so brute forcing is unlikely to be feasible. However, a dictionary attack with a more targeted word list might

be successful.

4.2.3 Mobile Apps

We decided to take a closer look at SmartThings' Mobile apps, since they essentially contain virtual ''keys'' to a user's home. We were more interested in data storage than other classes of mobile vulnerabilities.

In iOS, requests are cached by default in an application's Cache.db file. The interesting thing we were looking for was the OAuth bearer token since it's the only thing needed to gain control over all associated devices. It also takes close to 50 years for the token to expire. If an attacker had physical access and jailbroke the phone, or a malicious app was installed and the phone was already jailbroken, then all of the users devices could be controlled.

On Android, our findings were slightly more fruitful. Aside from the bearer token, we found user credentials being stored in base64 encoded plaintext. For this to be dangerous, an attacker must circumvent Android's ''app sandboxing''. The difficulty of this varies per device, and depends on the amount of effort an attacker needs to go through in order to obtain root access. However, if a user has rooted their phone, an attacker may be able to remotely steal credentials via a malicious application.

Both apps expose users to an unnecessary amount of risk, which is especially high if the devices are rooted/jailbroken. It is safe to assume that given access to a user's phone, an attacker could gain access to their SmartThings Account.

4.2.4 Arduino Shield

The SmartThings Arduino Shield has the same Ember ZigBee chip [Cel13] as the Hub and the Multi, but attaches to an Arduino so that a user can program it, allowing a great amount of flexibility. Because of the ability for the user to program the shield, we believed it would allow us a greater insight into the way SmartThings uses the ZigBee protocol, which is a major attack surface.

When we initially started looking at the Shield, SmartThings had migrated their forums and all of the documentation had been lost. Fortunately, we were able to retrieve the documentation from archive.org, which had scraped the old forums, complete with a Dropbox link to the SmartThings library. But their library proved flaky, due to the unreliable SoftwareSerial code they used to communicate with the Ember chip. We again pulled out the logic analyzer to verify that this was the case - the Ember chip would write on the SoftwareSerial line and the Arduino simply would not recognize it.

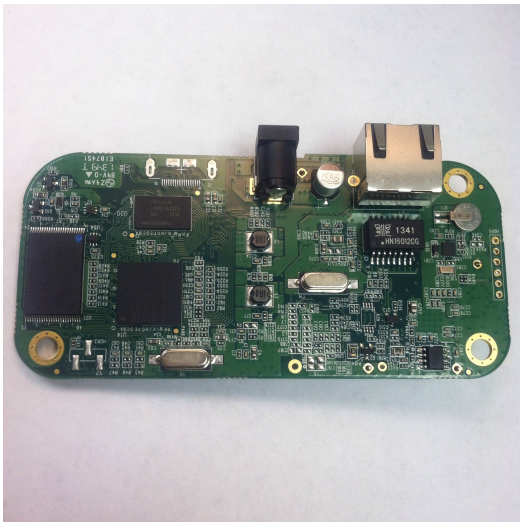
SoftwareSerial is used on the smaller Arduino boards because they only have a single serial port, which is used to communicate over USB to the host computer. Without the SoftwareSerial library, this port would have to communicate with either the shield or the computer, but not both. The Arduino Mega, on the other hand, has four serial ports, so it can use one to communicate with the computer and another to talk to the shield. We wrote a small Arduino program (Github) which simply passes through the computer and shield serial communication back and forth, giving us a terminal prompt on the Ember chip. This chip appears to be running firmware built by the Ember Application Framework [Cor08], a software builder provided by the manufacturer. Whenever the Shield recieves a message it prints out the details over its serial port, so with a bit of string parsing code we reproduced the functionality of the SmartThings library.

We took this opportunity to inspect the ZigBee communication between the Hub and Shield. This protocol proved significantly more difficult to parse than than the wired protocol discussed earlier. Wireshark stopped parsing it after the IEEE 802.15.4 header, and the packets do not match up with the ZigBee specification provided online. At least some part of the protocol appears to be encrypted, but we were thus far unable to identify the method. We did discover a small bug with messages sent to the Shield from the SmartThings

cloud: messages longer than 34 characters will cause the Hub to stop responding to further commands until it is restarted.

4.3 TCP Connected

The gateway has a few interesting hardware components. The main SoC is an Advance Micro APM80186-SKC600T, which is a PowerPC CPU with Ethernet and NAND controllers on board [lim]. The wireless stack is handled by an NXP JN5148, which includes an 802.15.4 transceiver and a 32-bit RISC processor [Sem]. The other main components are Hynix flash and DRAM chips. Pictures of the gateway's PCB are included below.



We examined the gateway, the mobile applications, and the wireless communications of the Connected by TCP lighting system. Each of these components will be discussed in the following sections.

4.3.1 Gateway Ethernet

Initially, we performed a port scan of the gateway using `nmap` and found ports 22, 80, 8888, and 9998 open.

We set up a man in the middle between the gateway and Greenwave's remote server by connecting the gateway to our computer with an Ethernet cable, then enabling tunneling from the computer's wireless connection. This forced all traffic from the gateway to go through our computer. Then, we used Wireshark to intercept its traffic.

We examined the web interface using Burp Suite, specifically using its proxy to intercept and replay requests (see Appendix C.1) for details.

Firmware Update With our man in the middle, we captured the gateway's firmware update request sent to the remote server. When we replayed this packet we received a link to a `rootfs.bin` file that we were able to download using HTTP. See Appendix C.2 for details. Using `binwalk` we extracted the firmware, a squashfs file system image we could explore on a local computer.

After we had the firmware, we extracted the `/etc/passwd` and `/etc/shadow` files. We used John the Ripper to launch a dictionary attack and recovered the root password "thinkgreen", within an hour. Section 4.3.2 contains more information on what we found in the gateway file system.

Remote Server Communication When a mobile application is connected to the system remotely, the app speaks to one of Greenwave Systems' servers, and the server relays messages to the gateway. This is possible because the gateway initiates a connection to the server on startup and keeps the connection open continuously. This allows the gateway to receive commands even though it is not visible to the public Internet. We forced the gateway to connect to our computer rather than Greenwave Systems' remote server using `dnsmasq` to spoof DNS entries to the gateway. By substituting our own IP address for GreenWave's web service at `tcp.greenwavereality.com` we could monitor the communications using Wireshark, as well as use Scapy to actively inject packets.

We found the communication between the server and the gateway is a binary protocol sent over TCP. A more detailed dissection of the protocol can be found in Appendix C.3. In addition to commands, the server and gateway maintained an active connection by periodically sending ping messages. The pings came in pairs, one sent from the gateway and one echoed from the server, and they all contained the same data.

We tried fuzzing the values in the commands and ping messages in different ways, such as by sending random data of varying lengths or extracting a valid command and randomizing any non-zero bytes. We also sent intentionally incorrect values as the ping reply. None of these produced a noticeable response, malformed messages were ignored, but the gateway maintained the connection.

4.3.2 Gateway Filesystem

The firmware contained the contents of the read-only filesystem loaded onto the gateway, but we were also able to examine the filesystem on the running gateway. The gateway had SSH running, and once we cracked the password from the shadow file we were able to connect as root. This allowed us to examine the files and the running processes. The gateway is running the Linux kernel with BusyBox utilities. The root filesystem containing the kernel and all applications is read-only, but there is also a writable filesystem for configuration files and persistent application storage.

We found a few interesting programs on the gateway filesystem: `6LoWPANd` is the program responsible for establishing the wireless network (more information about this program is in Appendix C.4), `holger` is the main application for coordinating the lighting system, and the executables in `/var/www/cgi-bin/` are the CGI applications responsible for the Intranet interface. We also loaded and executed our own programs by placing them in the writable storage. We used this capability to copy PowerPC versions of debugging utilities (such as `gdb` and `strace`) to the gateway and attach them to running processes to reverse engineer running applications directly on the gateway.

4.3.3 Wireless Communication

The gateway establishes a local wireless network using the 802.15.4 protocol. We used `zbdump` from https://-code.google.com/p/killerbee/KillerBee to capture this traffic, and Wireshark to analyze the resulting packet captures. We captured traffic while performing common tasks with the system such as turning lights on and off using both the gateway and the remote.

The program `6LoWPANd` is responsible for establishing the wireless network, including setting the encryption key. We found that the encryption key being used was set with a command line parameter to this daemon, and was a six-byte value, the MAC address of the gateway. This is then padded with zeros to a create 128-bit AES key (see Appendix C.4).

Once we had the encryption key, we were able to spoof commands to the lights. We captured and decrypted packets to turn the light bulbs on or off. We then reused the payloads to construct and encrypt our own packets. We incremented the frame counter field of the IEEE 802.15.4 packets to circumvent replay protection,

and encrypted and authenticated the packets with the encryption key we discovered (see Section 2.1 for more information on the security features provided by IEEE 802.15.4). The light bulbs responded to our packets and would turn on or off in response to the commands we sent.

4.3.4 Local Intranet Application

The gateway runs a web server which it uses to host an Intranet application that can be used to control the lighting system. Commands input to the web application result in POST requests to `/gwr/gop.php`. Each request includes XML-encoded data that comprises the command to the gateway. When a user first accesses the web application, a request is made sending a default username and password to the gateway.

This is done transparently to the user, and is not validated. We intercepted requests and changed the username and password to invalid values, and were still able to control the lighting system.

4.3.5 Mobile Applications

On the Android device, we used a wrapper around `tcpdump` to capture packets as we used the app. This was effective in sniffing unencrypted traffic between the phone and the gateway. When the phone is connected to the same network as the gateway, it issues HTTP POST requests directly to the gateway for `/gwr/gop.php`. The requests are identical to those generated by the Intranet application.

When the phone is not connected to the same local network as the gateway, commands go through Greenwave Systems' server. The application authenticates to the server by sending the username and hashed password of the user. Validation is performed on the credentials, and if incorrect credentials are sent, the user will not be able to control a lighting system.

To examine the app, we used a file browser to look at its stored configuration files. In addition, we pulled the apk from the phone and used Java Decompiler to decompile the class files that it contained. We analyzed how the app handles authentication but found no vulnerabilities.

There also is an iOS application, however due to time constraints we were unable to assess the iOS application.

4.3.6 Hardware

We disassembled various components of the hardware. For the gateway, we used a logic analyzer to listen on the various exposed headers to see if there was a serial port we could eavesdrop on. However, even though there were several headers and JTAG ports visible on the board, we were unable to get any recognizable data from the board. Section 3.3 has pictures of the gateway's PCB.

4.4 HAI MicroControl

The HAI MicroControl is a home automation device manufactured by Home Automation, Inc. The main component we tested on the HAI MicroControl was its ZigBee communication. We used an ApiMote V4 (see Section 2.2) for the sniffing, injection, and analysis of ZigBee packets. We also made use of a Freakduino Chibi, which runs the ZigTools framework to sniff ZigBee traffic.

4.4.1 ZigBee Association

To understand the process of association, we flooded the controller with a broadcast Beacon Request until we got a response with the Source address and the EPID of the controller. After inspecting the Beacon Response packet carefully, we found that the Association Permit field is set to false. This field is automatically set to false by the Coordinator (here it is the HAI Microcontrol) if the device is already associated. Initially, if no device was associated, this field is set to true. There is no means to change this manually unless the device is completely

disassociated and then re-added.

Transport Key Retrieval Next we tried to forge the association process. First, we sent a Beacon Request and waited for a response. We then sent a Leave Request originating from the Kwikset Lock, with the Remove Children and Rejoin fields under the Leave Request set to false. Once this packet is sent, the device is dissociated and does not communicate with the Controller anymore. Finally, we sent a Rejoin request back to the controller, impersonating the Kwikset Deadbolt, which successfully re-associates the device. We confirmed the addition of the device to the ZigBee network by observing a Device Announcement packet being broadcast across the network. Once the device joins, the transport key (used to encrypt ZigBee traffic over the air) is shared with all the devices in plaintext and can be sniffed out of the air.

Theoretically, just sending an association request followed by a data request until an association response is received should yield the transport key, but we were unable to forge the requests correctly as we never got a response back. After this failed, we tried to send a Permit Join request immediately after receiving a Beacon Response with the EPID of the controller. Since no device has been associated with the controller yet, we set the Association Permit field to True. Then, sending an Association Request yielded a response and a transport key.

Forced Device Association We successfully retrieved the transport key, but the device does not show up as associated in the HAI MicroControl. The MicroControl scans for exactly sixty seconds to facilitate the addition of new ZigBee devices into the network. We repeated the process of sending a Beacon Request to get the details of the controller, then sent a leave request and rejoin request in a denial-of-service (DoS) attack. We sent the DoS packets for sixty seconds until the requests are picked up. This allowed us to successfully impersonate the device to the MicroControl.

Device Takeover In reality, we would always have a device associated with the controller. In order for us to take control of that device, we would have to unpair the device with the coordinator by sending a leave request. This request does not need the short address of the device but just the MAC address, which makes the attack much easier, and allowed us to successfully add the device using a spoofed rejoin request.

This may not always work, because the Association Permit field is set to false and it is not possible to change it. Another thing to note is that when sending a rejoin request, the device is expected to provide a short address, which is actually assigned by the controller. An attacker would not know the hex address, but any address except `FFFF` and `0000` can be used. A Route Request is then broadcast across the network, and the next time a Rejoin or an Association request is sent, it is automatically routed to the address we requested. Thus, when there are multiple ZigBee devices across the network, we can request all those addresses and assign them to our device, thereby disconnecting them.

4.4.2 Kwikset Lock

We tried a number of attacks on the Kwikset Lock's ZigBee communication. We successfully decrypted the packets using the sniffed ZigBee Transport Key and found the Lock's lock and unlock commands (`0x01`-Lock and `0x00`-Unlock). In order to attack the lock, we needed to decrypt the packet, increment the sequence number, change the command, and then re-encrypt it using the Message Integrity Code (MIC) and the transport key. Unfortunately, we could not successfully re-encrypt the packet.

The second attack we tried was to wait until the sequence number wrapped and then replay the packet. This replay attack was unsuccessful for two reasons. First, we were unable to compute the MIC which the lock uses to ensure the message is valid. Second, we were uncertain about the limit of the sequence number. If we

were to blindly replay the packet, the MIC would be correct, but the sequence number would be rejected. Ultimately, we did not unearth vulnerabilities in the Kwikset Lock's ZigBee comunication.

4.5 Kwikset Lock

The Kwikset Z-Wave Lock is compatible with both the SmartThings platform and the HAI MicroControl. The analysis of those systems is in their respective sections, this section discusses the lock hardware independent of any hub.

The lock has two main components, the base, which interfaces directly with the dead bolt and keypad, and the Z-Wave module, which deals with wireless communication. We successfully dumped and analyzed the firmware running on the base, but were unable to communicate directly with the Z-Wave module. In order to program or otherwise interact with this chip, it is necessary to obtain a development kit. However, Sigma Designs requires anyone who wants a development kit to sign a non-disclosure agreement, which would have made it impossible for us to reveal any information we found.

4.5.1 Hardware

We initially targeted vulnerabilities in the lock itself in an attempt to completely bypass any home security system it happened to be attached to. It is important to note that if the lock was installed in a door, it would not be able to be disassembled, so vulnerabilities found in the hardware might lead to a dead end. We started by taking the lock apart to gain access to the board, which was a relatively simple process. By examining the board and its components we discovered that the microcontroller unit (MCU) was an MSP-430 variant. We also found some serial connections, which we soldered headers onto.

We were able to communicate with the MCU by connecting the board to a TI-Launchpad via Spy-Bi-Wire, then connecting the Launchpad to our laptop via USB, and finally interfacing with the chip via mspdebug. From here, we were able to dump the firmware and debug the code running on the MCU. We began reverse engineering the firmware by downloading the chip's data sheet and mapping out the functions referenced in the Interrupt Vector Table. Eventually we managed to map out where user PINs were stored, and how input was handled, but it became apparent that all processing of wireless commands happened in the RF module. For details about interfacing with the lock, as well as notes about the firmware, see Appendix E.1.

4.5.2 Z-Wave

Finally, we tested the Kwikset Lock's Z-Wave communication. While attempting to find an RF configuration that would allow us to intercept Z-Wave packets (see Section 6.2.4 for more details), we found a method of interfering with the lock's normal operation. Specifically, we are able to abuse clear channel assessment and use it as a vector for denial of service [CM]. This is somewhat similar to several ''jamming'' attacks against key fobs which were used in a series of luxury car thefts. However, rather than relying on a stronger signal to drown out other transmissions, this attack only requires that an attacker transmit continuously. For more information, see Appendix E.2.

5.1 Lowes Iris

5.1.1 Disclosure Timeline

- 2014-11-25: iSEC called Lowes indicating that there are security issues that need to be addressed.
- 2014-11-25: A customer support representative from Lowes asked iSEC to send the findings to them via email.
- 2014-11-25: iSEC sent an email to the representative containing the advisories.
- 2014-11-25: The representative responded stating that they forwarded the email to the appropriate department.
- 2014-12-18: iSEC followed up with the representative about the status of the advisories.
- 2014-12-19: Technical Director of Lowes Iris indicated that the concerns have been evaluated internally and no action needs to be taken.

5.1.2 Transport Key Sent in Plaintext

When a device associates with the Lowes Iris hub, the transport key is sent in plaintext from the hub to the device. This transport key is used by devices to encrypt and decrypt data to and from the hub. The key is also used to create message integrity codes (MICs) for frames sent and received. An attacker can sit outside of a house that uses the Lowes Iris System and sniff for ZigBee traffic. If an attacker can sniff ZigBee traffic, then the transport key can be sniffed out of the air. Key exchange should use a known, secure key exchange protocol.

5.1.3 No Cross Site Request Forgery Protection

There is no Cross Site Request Forgery (CSRF) protection on the web application. This can be used by an attacker to trick an authenticated user into disabling his/her alarm if it is set. The attacker can also put the Lowes Iris System in pairing mode and force the transport key to be sent at any time. The following vulnerability illustrates the impact of this.

5.1.4 Force Association to Hub to Receive Transport Key

Using the fact that the transport key is sent in plaintext (See Section 5.1.2) and that there is no CSRF protection on the web application (See Section 5.1.3), an attacker can force association between a compatible device and the Iris hub. The association process can be found in Appendix A.1.

An attacker can utilize a CSRF attack and trick a user into starting the association process. This will alert the Hub into sending out a permit join request. This permit join request allows any compatible device in range to be paired with the Hub for a certain duration of time. The Association Request and Data Request are not encrypted, nor do they use any message integrity code. This allows anyone to spoof and inject these packets from any compatible device. When these two frames are sent, the hub responds with an association response stating whether or not the association was successful or unsuccessful. If successful, the hub then sends the transport key to the device wanting to pair. Since the transport key is sent in plaintext, this key can be grabbed out of the air.

5.1.5 Alarm Stays Set After Dissociation

The disassociation of any device paired with the hub while the alarm is still set will not trigger the alarm. An attacker tricks an authenticated user into starting the dissociation process by exploiting the CSRF vulnerability. By sniffing the ZigBee traffic to see what devices are running based on their MAC addresses, an attacker can

then unpair all of the devices attached to the target Lowes Iris hub. By doing this, an attacker can disable any motion sensors or contact sensors. These would usually trigger the alarm and alert the owner of the house. However, since the attacker has unpaired these devices but the alarm is still set, the attacker can break into this house without triggering the alarm.

5.1.6 Internal Infrastructure Exposed in Mobile Applications

Internal infrastructure for the Lowes Iris Home Management System is exposed in both the iOS and the Android application. In the iOS application, there is a plist file called `servers.plist` which contains IP addresses and hostnames of what seems to be different VM and Xen servers for staging and development of this product. The same information can be found within the Android application in an XML file called `arrays.xml` This information contained in these files should not be public knowledge. An attacker can extract these files from either application and gain valuable knowledge about the Lowes Iris internal infrastructure.

5.2 SmartThings

5.2.1 Disclosure Timeline

- 2014-10-27: Reached out to vendor looking for security contact.
- 2014-10-27: Received contact email address and PGP public key.
- 2014-10-28: Disclosed vulnerabilities to vendor.
- 2014-10-29: Vendor acknowledges receipt and schedules phone discussion of vulnerabilities.
- 2014-01-02 Vendor indicates patches for some flaws have been rolled out.
- 2014-03-09 Vendor indicates final patches have been rolled out.

5.2.2 Server Impersonation

The SmartThings Hub does not perform certificate validation when using SSL to connect to SmartThings servers. This lack of validation means that the Hub does not know whether it is communicating with the SmartThings cloud, or an attacker. If an attacker is able to perform a DNS spoofing attack, they will be able to decrypt all packets sent by the Hub, or to send arbitrary commands. We successfully exploited this vulnerability by setting up a man-in-the-middle attack, and used it to open and close our door lock.

5.2.3 Protocol Downgrade

The SmartThings Cloud allows the connecting device to choose which protocol to use. An attacker can connect to the cloud using an outdated version of the protocol, sidestepping additional security features implemented in the newer version. Analysis of this protocol is present in Section 4.2.2, and the protocol's commands are described in Appendix B.2.

5.2.4 Hub Impersonation

The SmartThings cloud does not authenticate connections from the SmartThings Hub, and allows multiple connections from the same Hub simultaneously. Combined with the Protocol Downgrade vulnerability, this lets an attacker impersonate any SmartThings Hub, as long as they know its MAC Address. For more information, see Section 4.2.2.

5.2.5 Clear-Channel Assessment Attack

The Kwikset lock is susceptible to a Clear Channel Assessment attack [CM]. By broadcasting random data over Z-Wave, an attacker can prevent commands, including the ''lock'' command, from being sent. Furthermore, the attacker could impersonate the Hub, as in vulnerability Section 5.2.4, to make it appear as though the lock operation completed successfully. See Appendix E.2 for configuration details.

5.2.6 AWS Arbitrary File Manipulation

The SmartThings mobile applications use hardcoded Amazon Web Services (AWS) keys to upload and download images of users' homes. With these credentials, an attacker can list, upload, and download arbitrary files from the 'smartthings-custom-location-backgrounds' AWS bucket. This includes disclosure of images that customers have uploaded to SmartThings.

5.2.7 App Storage

The iOS application has cached HTTP requests that include an OAuth bearer token, while the Android application stores base64 encoded user credentials in local storage. If an attacker had physical access to a users phone, they could steal this information and use it to take control of a user's account. A malicious app on a jailbroken or rooted device would also be able to steal the user's information.

5.3 TCP Connected

5.3.1 Disclosure Timeline

- 2014-12-15: Reached out to vendor looking for security contact
- 2014-12-16: Received contact information
- 2014-12-31: Sent encrypted vulnerability details to contact.
- 2015-01-29: Contact confirmed receipt.
- 2015-02-13: Notified by contact that decryption was unsuccessful.
- 2015-02-13 - 2015-03-09 : Multiple attempts to exchange encryption keys unsuccessful.
- 2015-03-09 - Vulnerabilities disclosed successfully.

5.3.2 Intranet Application

No authentication is required to access the intranet application, so anyone who is connected to the same network as the gateway has the authority to change the state of the lighting system. The gateway does not validate that requests originate from the web application, so it is vulnerable to cross site request forgery (CSRF) attacks. However, a command must specify a bulb ID in order to affect its state, so an attacker needs a way to discover the bulb IDs of the user's system. These are 64-bit values so they are infeasible to guess through a brute force attack. We were not able to determine how the bulb IDs are assigned, but they do not appear to be random. The bulb IDs of two of the light bulbs we had differed in value by only 758, which is much less than would be expected for two random 64-bit integers. Since there appears to be some pattern to how the bulb IDs are assigned, an attacker may be able to determine a valid ID with high probability if they have access to other information about the system, allowing for a successful CSRF attack.

5.3.3 Mobile Application

Any mobile device connected to the same network as the gateway has full control over the lights. Additionally, while connected to the same network, a user is able to create a username and password that will allow them to control the gateway over the Internet. If an attacker is able to connect to the local network, they can register a username and password and then remotely control the lighting system when not connected to that network.

When connecting remotely through the mobile application, the password is hashed (`SHA256(username + MD5(password))`) before it is sent to the Greenwave Systems server. However the password is saved on the phone in plain text, accessible by anyone who has root privilege.

5.3.4 Firmware Update

No authentication is used in the process of a firmware update request from either end. The remote server does not authenticate the gateway, so an attacker can replay a request and receive a copy of the firmware, with the hard-coded `root` password. The gateway downloads firmware over HTTPS, but the certificate is not validated (see Section 5.3.5), and the firmware image is not signed.

While the update firmware request from the gateway contained a token, a MAC address, and the gateway's current version, none of these values are authenticated and we are able to receive the download link despite changing these values.

5.3.5 SSL Certificate Validation

During firmware download, the gateway does not validate the certificate of the remote server and will accept a self-signed certificate. This combined with unsigned firmware leaves the gateway vulnerable to downloading untrusted firmware. For example, an attacker could use DNS spoofing to pose as the remote server and provide a malicious firmware update.

5.3.6 Remote Server Impersonation

The binary protocol used to send remote commands from Greenwave Systems' server to the gateway is not encrypted, and there is no authentication built into the protocol. If an attacker manages to pose as the remote server to the gateway, they can send commands to control the lighting system. Additionally, the gateway simply ignores incorrect commands, it does not reset or close the connection, so an attacker has a very forgiving platform to reverse engineer and fuzz the protocol.

5.3.7 Open SSH Port, Weak Root Password

SSH is running on the gateway with a weak `root` password hard coded as part of the firmware. This allows anyone on the same network to obtain a `root` shell on the gateway. This gives an attacker complete control of the gateway. They can examine the file system and running processes, find the wireless encryption key, or load their own scripts and executables to writable storage and run code on the gateway.

The initialization scripts are all in the read-only filesystem, but some of them execute scripts in writable storage, so attacker could modify these scripts to execute their own malicious applications at startup. Firmware updates only change the read-only filesystem, but leave the writable storage unchanged, so a compromise will be persistent even across firmware updates. An attacker could also tamper with any part of the firmware by using `dd` to directly write to the underlying block device of the read-only file system.

5.3.8 Weak Encryption Key

The device is using the security features of 802.15.4 to encrypt and authenticate messages, but the key is so weak it is easy to bypass. The encryption key is set to the MAC address of the gateway, so there are only three bytes of potential randomness in the key. A MAC address is a 6-byte value, but the first three bytes are the organizationally unique identifier (OUI) of the manufacturer[IEE]. In this case it is the sequence `D4-A9-28`, which is the OUI of Greenwave Reality (now Greenwave Systems) [oui]. Since the search space is so small it is feasible to mount a brute force search to find the encryption key.

Once an attacker has discovered the encryption key they can observe all traffic on the wireless network and create their own packets to control the lighting system. It is also possible to mount a denial of service attack by broadcasting messages with large frame counters to a particular receiver. This will increment the internal replay counter of the receiver so that legitimate messages will subsequently be ignored [PKH+].

The previous attacks on the encryption required the attacker to first determine the encryption key, but because of how the frame counter is stored on the light bulbs it is possible to mount a replay attack without the key. Light bulbs keep the state of received frame counters in volatile memory. If the light bulb loses power, the knowledge of received frame counter values will be forgotten, and then a replay attack will succeed.

5.4 HAI MicroControl

5.4.1 Disclosure Timeline

- 2015-03-17: Request for security contact sent to Leviton.
- 2015-03-23: Follow-up request for security contact sent to Leviton.
- 2015-03-27: Call placed to Leviton requesting security contact.
- 2015-03-30: Response received from Leviton, details sent via fax.
- 2015-04-01: Acknowledgement received from Leviton.
- 2015-04-06: Follow-up phone call with Leviton. Leviton indicates disclosed vulnerabilities are inherent to Zigbee home automation and no action is required.

5.4.2 Static ZigBee Link Key

The MicroControl sets up its ZigBee network with a static Link Key across all ZigBee devices. The key is "ZigBeeAlliance09" encoded in hex, and is also used with its bytes in reverse order.

5.4.3 ZigBee Transport Key Disclosure

Forcing requests to the controller results in the current Transport Key being transmitted. The Transport Key is used to communicate between the MicroControl and its associated ZigBee devices. An attacker can perform this attack by sending a Beacon Request followed by a forged leave request, which results in the target device being disassociated from the controller. Next, a rejoin request is sent back to the controller, impersonating the target device. The device is successfully associates again, and the transport key is shared in plaintext over the network. See Section 4.4.1 for more details.

5.4.4 Device Impersonation

An attacker can impersonate any device and the MicroControl is not able to differentiate between the actual device and the attacker. When adding new devices to the network, the HAI MicroControl scans for sixty

seconds. By repeatedly sending leave and rejoin requests for sixty seconds, an attacker can force an association response and join the network with a new device. See Section 4.4.1 for more details.

5.4.5 Forced Short Address Assignment

An attacker can set the short address of an end device to any address he wants, without needing to know the current short address of the device. The major consequence is that an attacker can force the dissociation of devices from the network. When there are multiple ZigBee devices associated with the controller, the attacker can reassign a device to an in-use address, causing a collision. The device originally at the colliding address is then dissociated from the network.

6 Further Research

6.1 Lowes Iris

We were unable to figure out the application level security that the Lowes Iris devices were using since we could not properly dump the firmware. This is the most important item that should be researched moving forward. Additionally, It would be useful to figure how to interact with the ZigBee chip. Future research should include both dumping the contents from the flash memory chip and directly interfacing with the ZigBee chip.

6.2 SmartThings

6.2.1 Hardware

In the future, it will likely be useful to dump the memory chips at different points in time, in order to gather information about the Hub's state. Potentially, this could allow us to gather information about the wireless devices connected to it and the security of the wireless network. We also did not investigate the custom header (labelled B in figure 2) - a soldering iron and steady hands will enable a connection there. We were not able to communicate over header A either. Finally, retrieving the Hub's firmware in a readable state will allow a much deeper level of investigation into how it works. There is definitely still work to be done in order to gain a complete understanding of the SmartThings hardware, and many more SmartThings-compatible devices to test as well.

6.2.2 Wired Networking

The SmartThings infrastructure has two very large holes in its wired networking: its lack of SSL certificate verification and the fact that its protocol does not provide any authentication. The logical next step in attacking this protocol is to impersonate the SmartThings server and fuzz the messages sent to the Hub. Fuzzing is an automated way to gather information about which messages the Hub will accept, reject, or cause an error. A well-designed fuzzer can find a huge amount of bugs very quickly, and provide a wealth of new ways to attack the device.

6.2.3 ZigBee

While we succesfully sniffed ZigBee traffic between the SmartThings Hub, Arduino Shield, and Multi, we were not able to fully understand the packet format. The ZigBee packets appear correct through the IEEE 802.15.4 header, but the ZigBee header does not match the format we expected to see. Further, the data field in the ZigBee packets sent by the Ember chips appears to be encrypted, but we could not figure out the method. Once the packet format is successfully parsed, it will open up a new avenue for attacks on the SmartThings system.

6.2.4 Z-Wave

Unfortunately we were not able to intercept Z-Wave traffic. We attempted to use Z-Force (see Section 2.2), a Z-Wave packet interception and injection tool, but could not get it to work. We contacted one of the authors of the Z-Force framework who informed us that the firmware itself was hard-coded for an EU RF configuration. We attempted to figure out the correct RF configuration but were only partially succesful. Although we did not have time to fully explore this option, it is possible to fairly trivially modify the Z-Force firmware to use other RF settings. For more information about this, see Appendix F.1.

In order to communicate with the Z-Wave chip (a Sigma Designs ZW0301) inside of the Kwikset Lock we required a development kit from Sigma Designs, who requires all buyers to sign an NDA. Due to only having one of these chips, we did not attempt any attacks against the hardware itself since it was necessary to have a functioning chip to test other components. In the future, we would like to have the resources to fully test the Z-Wave functionality of the SmartThings system.

6.3 TCP Connected

The gateway coordinates the wireless network used to control the lighting system. We were able to observe the commands sent to change the state of the light bulbs. However further research is required to fully reverse engineer the protocol the gateway uses, including the process to add new devices to the network and exchange encryption keys.

6.4 Hai MicroControl

The majority of our research with the Hai MicroControl was focused on its ZigBee network. In order to gain more insight into ZigBee, it would be beneficial to fully understand the encryption algorithm and ZigBee's Message Integrity Code. It may then be possible for us to modify and re-encrypt the ZigBee packets correctly, allowing us to unlock the Kwikset Lock device over-the-air. Finally, we did not investigate the Internet functionality of the MicroControl, or the iPhone and Android applications, which is a huge surface for further research.

7 Conclusion

Internet-connected home automation systems are bringing us closer than ever to achieving the science fiction dream of a fully-automated home. That dream comes at a cost, however: as more technology is added into the home, more and more systems have to work together perfectly to keep that technology secure. We anticipate that the attacks laid out in this paper are only a milestone on a long road of vulnerabilities that will make up the history of the Internet-of-Things.

The systems we investigated provided ample opportunity for us to learn about the technologies that bridge embedded devices and the greater Internet. While it would be impossible to fully explore every aspect of every wireless home automation device on the market, none of the systems we looked at came away clean. For some manufacturers, security seems to be nothing more than an afterthought. Others have followed some standard secure practices, but lost their security in the implementation details. We know that further research will be done on these and other home automation devices, and we hope our research acts as a springboard in this ever-growing field.

As a long-term goal, it would be desirable to compile a best practices document for the design of IoT systems such that vendors have guidelines on how to prevent the introduction of flaws that actually impact users in practice.

References

[Atm14] Atmel. AT25128B and AT25256B Datasheet. http://www.atmel.com/Images/Atmel-8698-SEEPROM-AT25128B-256B-Datasheet.pdf, jul 2014. 10

[Cel13] Cel. MeshConnect EM357 Mini Modules Datasheet. http://www.cel.com/pdf/datasheets/MeshConnect_EM357_Mini_Modules_DS.pdf, oct 2013. 11, 16

[CM] Bo Chen and Vallipuram Muthukkumarasamy. Denial of Service Attacks Against 802.11 DCF. http://www98.griffith.edu.au/dspace/bitstream/handle/10072/12207/41331.pdf. 21, 24

[Cor08] Ember Corporation. EmberZNet Application Developer's Guide, nov 2008. 16

[/de13] /dev/ttyS0. Differentiate Encryption From Compression Using Math. http://www.devttys0.com/2013/06/differentiate-encryption-from-compression-using-math/, jun 2013. 14

[Des12] Sigma Designs. ZM4101 Z-Wave Integrated Wireless Module for Home Control, may 2012. 10

[Gra] Physical Graph. SmartThings FCC Documents. https://apps.fcc.gov/oetcf/eas/reports/ViewExhibitReport.cfm?mode=Exhibits&RequestTimeout=500&calledFromFrame=N&application_id=774759&fcc_id=R3Y-STH-ETH001. 10, 11

[IEE] IEEE Standards Association. *Guidelines for Use Organizationally Unique Identifier (OUI) and Company ID (CID)*. 26

[IEE11] IEEE Standards Association. *IEEE Standard for Local and metropolitan area networks Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*, sep 2011. 5

[Int10] Macronix International. MX25L3206E Datasheet, apr 2010. 10

[Int11] Internet Engineering Task Force (IETF). *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks*, sep 2011. 5

[KYT] Maxim Krasnyansky, Maksim Yevmenkin, and Florian Thiel. Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt. 40

[lim] Acal BFi limited. Energy-Efficient, Consumer SoC Embedded Processor. http://www.acalbfi.com/uk/p/0000001B23. 17

[LLC] River Loop Security LLC. Killerbee: Framework and tools for exploiting ZigBee and IEEE 802.15.4 networks. https://code.google.com/p/killerbee/. 9

[May] Daniel A. Mayer. idb - Github. https://github.com/dmayer/idb. 9

[NXP] NXP. *JN-AN-1110: JenNet-IP Border Router Solution Application Note*. 40

[oui] https://standards.ieee.org/develop/regauth/oui/oui.txt. 26

[PKH+] S. Park, K. Kim, W. Haddad, S. Chakrabarti, and J. Laganier. *IPv6 over Low Power WPAN Security Analysis*. Internet Engineering Task Force (IETF). 26

[SB11] Zach Shelby and Carsten Bormann. 6LoWPAN: The wireless embedded Internet - Part 1: Why 6LoWPAN? http://www.eetimes.com/document.asp?doc_id=1278794, may 2011. 5

[Sem] NXP Semiconductors. NXP JN5148. http://www.nxp.com/products/microcontrollers/wireless_microcontrollers/JN5148.html. 17

[SMS12] SMSC. LAN8720A/LAN8720Ai Datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/87

20a.pdf, aug 2012. 10

[Tec13] Microchip Technology. PIC32MX5XX/6XX/7XX Family Datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/61156H.pdf, 2013. 10

[War] Mike Warner. ZigTools: An Open Source 802.15.4 Framework. https://isecpartners.github.io//tools/2014/08/04/ZigTools-release.html. 9

[Zig08] ZigBee Alliance. *ZigBee Specification*, jan 2008. 5

A.1 Assocation Process

- Iris Hub ⇒ Permit Join Request ⇒ Broadcast
- Device ⇒ Association Request ⇒ Iris Hub
- Device ⇒ Data Request ⇒ Iris Hub
- Iris Hub ⇒ Association Response (Successful/Unsucessful) ⇒ Broadcast
- (if successful) Iris Hub ⇒ Transport Key ⇒ Device

B.1 SmartThings Hub Pin Traces

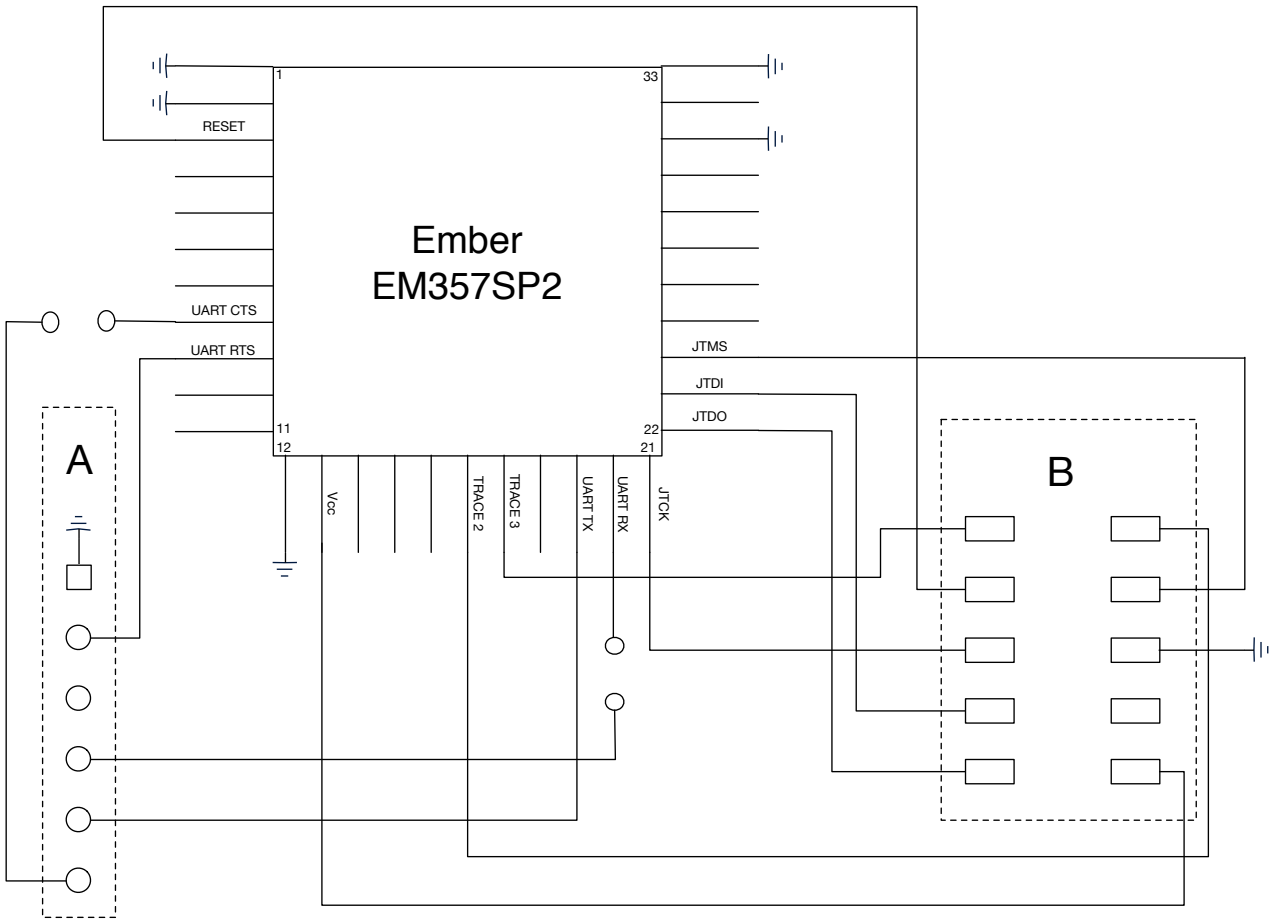


Figure 6: The Ember EM357 ZigBee Transciever. The rightmost set of rectangular pads is a custom header, while the left set of circular pads is a standard header. The gaps in the UART RX and CTS lines are unsoldered in production versions of the Hub.

B.2 SmartThings Hub Network Protocol

B.2.1 Version 1

Version 1 of the SmartThings Network Protocol is valid for Hub firmware version `000.010.00246`.

The protocol is a hex-encoded payload, wrapped in angle brackets and sent over SSL. The raw data looks like this (each line is a string of ASCII characters), and is terminated with a newline:

```
<1700D052A80043C4000180000002000A00F60000000000008A0A>
```

The packets are constructed as follows:

	LEN	CMD	MAC Addr.	Device	Z-Wave	Sequence	Data	CKSM
Size (bytes)	2	2	12	4	4	4	variable	4
Sample	`0D`	`02`	`D052A80043C4`	`0001`	`8000`	`0002`		`11CE`

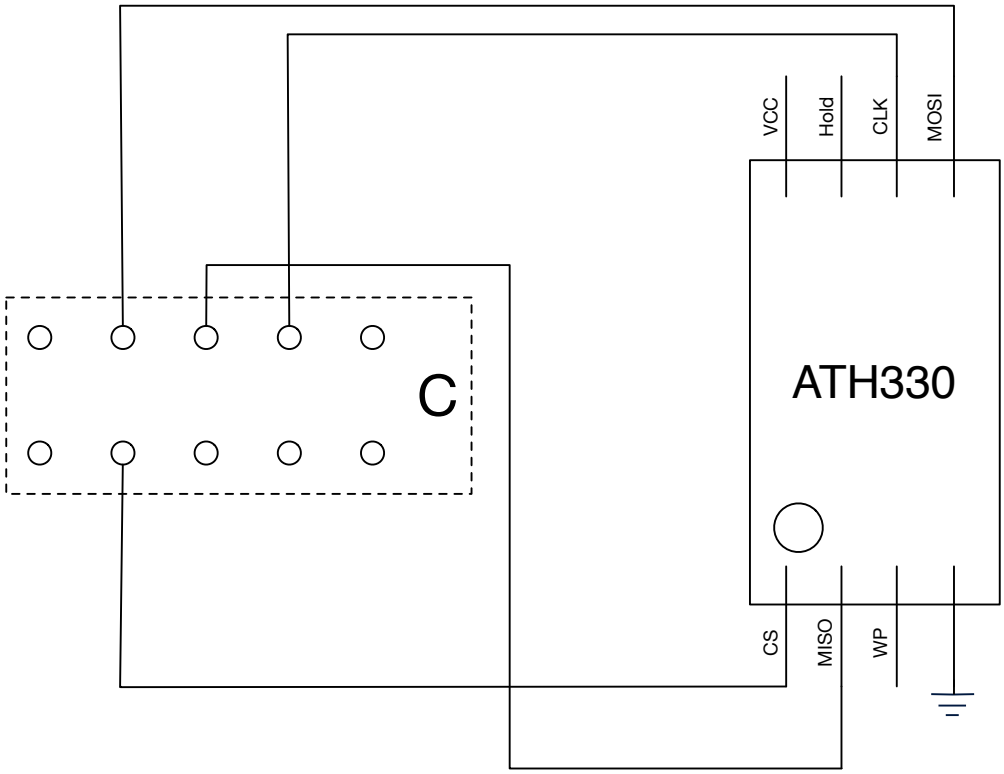


Figure 7: The Hub has two EEPROMs (memory chips) in addition to the memory built in to the PIC32 processor. The Atmel AT25256B EEPROM connects to a 5x2 set of standard pins and can be accessed using SPI.

1. LEN: The length of the packet, in hex
2. CMD: The command for this packet
3. MAC Addr.: The MAC Address of the Hub
4. Device: The source or target device on the ZigBee network
5. Z-Wave: The source or target device on the Z-Wave network
6. Sequence: The sequence number of this packet
7. Data: Variable length data field corresponding to the relevant command
8. CKSM: 16-bit CRC-16-CCITT checksum with IV `0xFFFF`

The following commands were observed during our testing:

Command	Name	Data Samples
`00`	**JOIN**	`00 0A 00F6 000000000000`
	Notes: Firmware Version `000.010.00246`	

01	**ACCEPT**	00000000
02	**ACK**	
04	**Keepalive**	
09	**Device - ?**	A97E 02FC010139000000 A97E 0101040139000100000010019 89EB 0101040138000100000 0
0A	**Multi - ?**	A97E 020102
0B	**Multi - Active** Notes: <2 byte network id>:<6 byte MAC>:<2 byte device id>:<1 byte data>	A97E D052A800453E 0003 80
0C	**Multi - Contact** Notes: <2 byte network id>:<3 byte status>	A97E 003100 E777 003000
0E	**Status** Notes: Sent after join/accept when device has been restarted or offline for a long time; Website says: ''stack status: 90''	90
14	**?**	0104000001010140008 9EB000000000B010A00
15	**Ember Command**	6E6574776F726B20706A6F696E203930 7A646F2073696D706C652030784139374520307830 31 7A646F2073696D706C652030784139374520307830 32

Notes: Ascii Decode
`zdo simple 0xA97E 0xNN`
`network pjoin 90`
`raw 0x0 { 00 00 0a 0a 68 65 6c 6c 6f }` (Shield: hello)
`send 0x89EB 1 01`

---

16 **Firmware Update (PUSH)** `0005A105`

---

17 **Firmware Update (ACCEPT)** `00`

---

18 **Firmware Update (CHUNK)**
Notes: `<2 byte chunk number>:<128 bytes hex data encoded as ascii>`

---

29 **Z-Wave Status** `FBCECE3101 00000..0000`
Notes: Z-Wave ID `FBCECE31 01`, sent after `0E` when device has been restarted or offline for a long time

---

40 **Multi - RSSI + LQI** `A97E 69FF`
Notes: `<1 byte RSSI>:<1 bytes LQI>`

---

41 **Multi - State**
`E777 0000010A1C5EFF`
`E777 000001061C64FF`
`A97E 010000FE1C64FF`

---

43 **Multi - ThreeAxis**
`A97E FFF8021F035B67FF`
`A97E FEC9035C01A068FF`
`A97E FCE1028A003366FF`
Notes: `<6 byte x,y,z>:<1 byte RSSI>:<1 byte LQI>`

B.2.2 Version 2

"Version 1" Style packets are prepended by 16 byte headers of the following form:

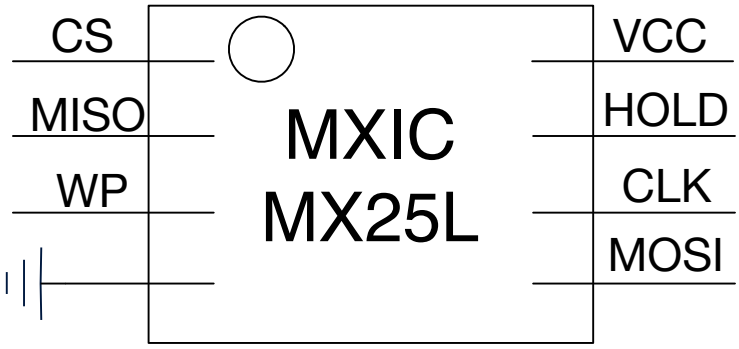


Figure 8: The MXIC MX25L3206E EEPROM does not connect to a test header, but the pins are spaced widely enough that a set of pin grabbers will hook on them. It can also be accessed using SPI.

PID	LEN	DIR	DEL	OVF	PKTN	CNST	????	DEL	TYP	TYPN
5E:A1:00	43	00	00	00	df	01	0b:de:41:fc	00	02	8c

1. PID: Protocol Identifier. Always equal to 5e:a1:00
2. LEN: Packet Length: Length of entire packet in base16.
3. DIR: Direction Byte: 0x00 when packet originates from Hub. 0x10 from server.
4. DEL: Delimeter: Appears to be a Null byte delimeter.
5. OVF: Overflow #: Number of times PKT has overflowed.
6. PKTN: Packet #: Global Packet Count (Of all Packet Types)
7. CNST: Constant: Appears to be a constant value.
8. ????: Mystery Bytes. Seem to be used for authentication. Can be replayed/reused.
9. TYP: Type: API or Binary? (Unconfirmed.)
10. TYPN: Packet count of packets of This Type.

Additionally, this version of the protocol includes some raw binary packets which take the place of certain Version 1 actions. They are prepended by a similar packet header, but do not always include the full 16 byte version.

Shown below is a "Hello" Packet from the hub:

PID	LEN	DIR	DEL	OVF	PKTN	CNST	????	MAC	DEL	FRM
5E:A1:00	1f	00	00	00	01	00	8c:d9:ff:4d:00:00:00:00:01	d0:52:a8:00:40:e1	00:01	00:0b:02:5b:04

1. ???: Mystery Bytes. Appear to be used for authentication.
2. MAC: Mac Address of the Hub
3. FRM: The firmware version of the connecting Hub.

We were not able to determine exactly how the four identification bytes are generated. They do, however, change with the packet number. There are 12 unique values which seem to loop for the first "Hello" packet. If it turns out that it is possible to automatically generate these values, then one could perform all "Version 1" attacks with "Version 2".

Packet Number	Bytes
01	8c:d9:ff:4d
02	72:d4:5c:ce
03	0f:29:93:f8
04	9b:50:cc:29
05	5c:63:8d:db
06	55:3c:92:0e
07	64:5c:90:3b
08	5b:d3:0c:ab
09	92:ba:c2:d3
0a	d3:41:8d:cd
0b	67:2e:ad:21
00	25:32:42:55

C.1 Local Intranet Application

Upon first connection, an email and password are sent to the gateway with the command `GWRLogin`; however, this combination is by default `admin/admin`, and is not validated by the server. In response, a token (fixed at `0123456789`) is sent back to the browser. This token is included in later requests, but again is not validated. Thus, there is no authentication required when communicating directly with the gateway.

C.2 Gateway Firmware Update

The request goes to `update.greenwavereality.com/roxy/update.php`.

Sample request body: MAC=D4:A9:28:01:F9:D0 & secret=177929cb196dd55e2818f8c9ea61fc & project=Apollo & current_version=2.0.39

Download link: `http://update.greenwavereality.com/roxy/download/1386066602/rootfs.bin`

C.3 Remote Server

The format of commands from the server to the gateway is:

`fe000000c81f000000090301f9d00051et45101`. Respectively, the emphasized sections indicate:

- The type of message (8 indicates power toggling and 9 indicates dim)
- A 12-bit subsection of a bulb ID (0x451 above)
- The command parameter (0x01 or 0x00 for power; a hexadecimal value from 0x0 to 0x64 for dim)

In addition to commands, the server and gateway maintained an active connection by periodically sending ping messages which all contained the same data: (`fe000000641f00000000`).

C.4 6LoWPANd

The function of 6LoWPANd is to direct network packets to the wireless chip. It does this by establishing a local TUN network interface [KYT] to allow it to receive packets from other processes running on the SOC targeting the wireless network. Communication to the wireless chip is performed over a serial port. After initializing the TUN interface and the wireless chip, messages are relayed from one to another. All of this is then transparent to applications that use the wireless communication, which do so by sending and receiving messages through sockets opened on the TUN interface, using the standard operating system APIs.

Initially we disassembled the 6LoWPANd binary do determine how the key is set, but the source code for 6LoWPANd is actually available from NXP [NXP].

The command line option 'k' sets the encryption key. We found that the command line options are parsed with `getopt_long`, and the case for k calls `inet_pton` on the input string to parse it as an IPv6 address. The 128 bit ''address'' is then used directly as the encryption key. So a value specified like this: `::D4:A9:28:01:F9:D0` will result in the key being set to the 128-bit value represented by the hex string:

`0000000000D400A90028000100F900D0`.

The configuration files for the gateway are located at `/etc/init.d/`. The daemon that handles the wireless communication is started by the script `/etc/init.d/6lowpand`, which sets the encryption key to a value found in the file `/media/config/jennic.config`. This config file sets the key used by the gateway to encrypt

and authenticate 802.15.4 to the MAC address of the gateway. The configurations are set by executing `/media/config/jennic.config` as a shell script, so an attacker could modify this file to start a malicious process at startup.

D.1 Appendix

This is an appendix of the different sequences of ZigBee Protocol packets that were being observed during the different processes:

D.1.1 Device Pairing with the Controller

1. A device announcement is made, indicating that the Kwikset Device has joined the network.
2. The Transport Key is shared amongst devices in the network.
3. An Active endpoint request is sent and the response includes the active endpoint list details and a status indicating success if a successful endpoint assignment happens.
4. A Simple descriptor request is made for each endpoint listed above which gives information about a particular endpoint device (e.g., Kwikset). The response includes an identifier for the device within 1-240 and it also shows the status of association (e.g., Successful).
5. Next, a Bind Request is sent which binds the coordinator and the device. A response is received indicating successful or unsuccessful binding.
6. A Configure reporting request is then sent and the response received back states the success, if the reporting is configured.
7. When the lock is associated with the coordinator, it is automatically Locked and shows that status. As the Next step, a set user descriptor request is sent wanting to set the Desc as 'LOCK'. A response is received stating confirmation of the user descriptor setting.

D.1.2 The Association Process

The sequence of what happens when a device gets associated with the coordinator.

1. A Beacon Request is broadcast which searches for available ZigBee coordinators.
2. A Beacon Response is received from the coordinators with their EPID.
3. An Association Request is sent, followed by a data request .
4. An Association response is received which includes the PAN ID and the short address of the device.

D.1.3 The UNLOCK/LOCK packet frame

Packet Structure of the UNLOCK/LOCK command frame. It contains four layers:

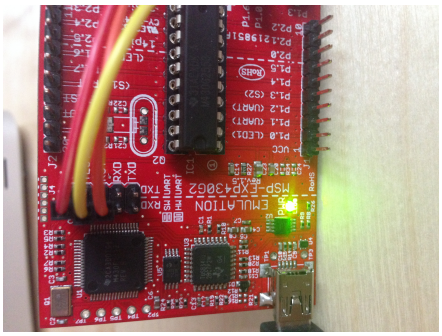
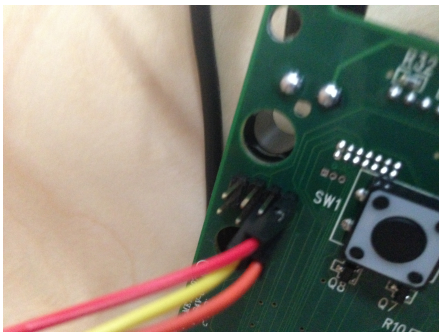
1. IEEE 802.15.4 layer with the Sequence Number, destination PAN and source and destination addresses.
2. The ZigBee Network Layer data contains data like source and destination addresses, the extended source which is the HAI MicroControl here. The ZigBee Security header contains information like the Key Sequence number and the nonce (MIC) if the extended nonce field is set to true.
3. The Application support Layer contains the Profile (0x0104) – Home Automation and the Cluster of the device in use.
4. The ZigBee cluster Library frame contains the actual lock (0x01) or unlock (0x00) command.

There are two commands exchanged in the process. The first is sent from the controller to the device and the second in the opposite direction. The second command includes one additional field, "Data" which has the actual lock or unlock command.

D.1.4 Route request

While testing for the Short Address Vulnerability as described in the previous section, the coordinator broadcasts a route request which contains a route request command frame in the Network Layer which mentions the Route ID and the destination short address along with the path cost. Using this command, the coordinator automatically resets the address of the device.

E.1 Firmware

Hold the board such that the black button is on top, and connect the pins as follows:

1. Vcc to Top Right
2. RST to Bottom Right
3. TEST directly left of RST.

Once you have the device connected to your computer, run `mspdebug rf2500` at a terminal to interface with the MCU.

There are a number of interesting addresses we identified. These should give you a good starting point if you want to investigate things further.

Interesting Functions:

- 0xC530: Activated on Key-Press
- 0xC51A: Optical Module
- 0xA002: Start Motor Function (Referenced by several functions that actually seem to check User-Input)

Interesting Memory Addresses:

- 0x1048: User PIN 1
- 0x1050: User PIN 2
- 0x054E: User Input Stored Here, reset to FF on timer.

E.2 Clear Channel Assessment Attack

These settings allowed us to prevent the Z-Wave enabled lock from properly communicating with the Hub. They can be easily set inside of SmartRF studio.

1. Base Frequency: 908.42 MHz
2. Xtal Frequency: 26.0 MHz
3. Modulation Format: GFSK
4. Channel Number: 0 (N/A)
5. Data rate: 9.6 kBaud
6. Deviation: 20 kHz
7. Channel Spacing: 200 Khz
8. RX filter BW: 58.035714 kHz
9. Manchester Enable: True

Place the device into Continuous TX mode and select "Modulated".

F.1 Z-Force

Within the Z-Force firmware, one function initializes most of the RF parameters. This function begins at offset 0x1E8. Setting parameters is as simple as modifying the binary in the locations where it is about to move values into external memory. Unfortunately, not all of the register settings are set, so one must patch in a routine which sets the remaining register settings and restores the system state. We believe that patching in the correct settings would allow one to use the Z-Force framework on US frequencies.

The raw binary must then be converted into Intel HEX format before it can be flashed onto the TI Dev-board. We used this method several times and had varying results, unfortunately, none of them produced output that looked correct. This process is automatable, but we did not have time to fully explore it.