Smart Nest Thermostat: A Smart Spy in Your Home

Grant Hernandez[1], Orlando Arias[1], Daniel Buentello[2], and Yier Jin[1]

[1]Security in Silicon Laboratory, University of Central Florida
[2]Independent Researcher
yier.jin@eecs.ucf.edu

Abstract

The Nest Thermostat is a smart home automation device that aims to learn a user's heating and cooling habits to help optimize scheduling and power usage. With its debut in 2011, Nest has proven to be such a success that Google spent $3.2B to acquire the company. However, the complexity of the infrastructure in the Nest Thermostat provides a breeding ground for security vulnerabilities similar to those found in other computer systems. To mitigate this issue, Nest signs firmware updates sent to the device, but the hardware infrastructure lacks proper protection, allowing attackers to install malicious software into the unit. Through a USB connection, we demonstrate how the firmware verification done by the Nest software stack can be bypassed, providing the means to completely alter the behavior of the unit. The compromised Nest Thermostat will then act as a beachhead to attack other nodes within the local network. Also, any information stored within the unit is now available to the attacker, who no longer has to have physical access to the device. Finally, we present a solution to smart device architects and manufacturers aiding the development and deployment of a secure hardware platform.

1 Introduction

The concept of Internet of Things (IoT) and wearable devices has been widely accepted in the last few years with an increasing amount of smart devices being designed, fabricated, and deployed. It is estimated that there will be more than 50 billion network connected devices by 2020, the majority of which will be IoT and wearable devices[1]. The once science fiction scenes that showed our refrigerators ordering us milk and our washing machines messaging us when laundry needs to be done are now reality.

The convenience provided by networked smart devices also breeds security and privacy concerns. Nest founder Tony Fadell claimed in an interview, "We have bank-level security, we encrypt updates, and we have an internal hacker team testing the security ... [the Nest Thermostat] will never take off if people don't trust it." However, a deep look into the current IoT and wearable device design flow revealed to us that most of the current security considerations, if any, are put on the application and network level. That is, designers often treat IoT and wearable devices as standard networked devices and try to apply the security protections developed for regular, everyday use computing devices. It is rare to find any work done beyond firmware authentication or encryption. Most IoT and wearable devices collect usage information and other data and send it to a service provider, leading to privacy concerns. Full disclosure of the what is collected is rare, and anything that is actually published is often hidden in the legalese that is the privacy policies and terms of services of the unit.

[1]http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

In the rest of the paper, we will introduce our work identifying a security vulnerability in the Nest Thermostat, targeting the hardware infrastructure and the hardware-software boundary. We will demonstrate that attackers who understand the hardware can change the boot process of the device in order to upload malicious firmware, effectively bypassing the firmware update verification done by the software. From a positive angle, however, we argue that this same vulnerability offers legitimate users a way to defend themselves against the collection of data thus protecting their privacy and to extend the functionality of the device.

2 The Nest Thermostat

The Nest Thermostat is a smart device designed to control a central air conditioning unit based on heuristics and learned behavior. Coupled with a WiFi module, the Nest Thermostat is able connect to the user's home or office network and interface with the Nest Cloud, thereby allowing for remote control of the unit. It also exhibits a ZigBee module for communication with other Nest devices, but has remained dormant for firmware versions up to the now current 4.2.x series.

The Nest Thermostat runs a Linux kernel, coupled with some GNU userland tools, Busybox, other miscellaneous utilities supporting a proprietary stack by Nest Labs. To remain GPL compliant, the modified source code used within the device has been published and is available for download from Nest Lab's Open Source Compliance page at `https://nest.com/legal/compliance`, with the notable exception of the C library. A toolchain to build these sources is not provided either.

2.1 User Privacy

The Nest Thermostat will collect usage statistics of the device and environmental data and thus "learn" the user's behavior. This is stored within the unit and also uploaded to the Nest Cloud once the thermostat connects to a network. Not only usage statistics are uploaded, but also system logs and Nest software logs, which contains information such as the user's Zip Code, device settings, HVAC settings, and wiring configuration. Forensic analysis of the device also yields that the Nest Thermostat has code to prompt the user for information about their place of residence or office. Reports indicate that Nest plans to share this information with energy providers in order to generate energy more efficiently.

2.2 Architecture Overview

As a device itself, the Nest Thermostat is divided into two components, a backplate which directly interfaces with the air conditioning unit and a front panel with a screen, a button, a rotary dial and a motion sensor. The operating system runs on the front plate.

The backplate contains a STMicroelectronics low power ARM Cortex-M3 microcontroller with 128KiB of flash storage and 16KiB of RAM, coupled with a few driver circuits and an SHT20 temperature and humidity sensor. The backplate communicates with the front plate using a UART.

The front panel offers a Texas Instruments (TI) Sitara AM3703 microprocessor, 64MiB of SDRAM, 2Gibit (256MiB) of ECC NAND flash, a ZigBee module and a WiFi module supporting 802.11 b/g/n. The board for this device also offers a Texas Instruments TPS65921B power management module with HS USB capabilities. This part of the Nest has been the target of our research so far, as it contains the most hardware and handles all user data and input.

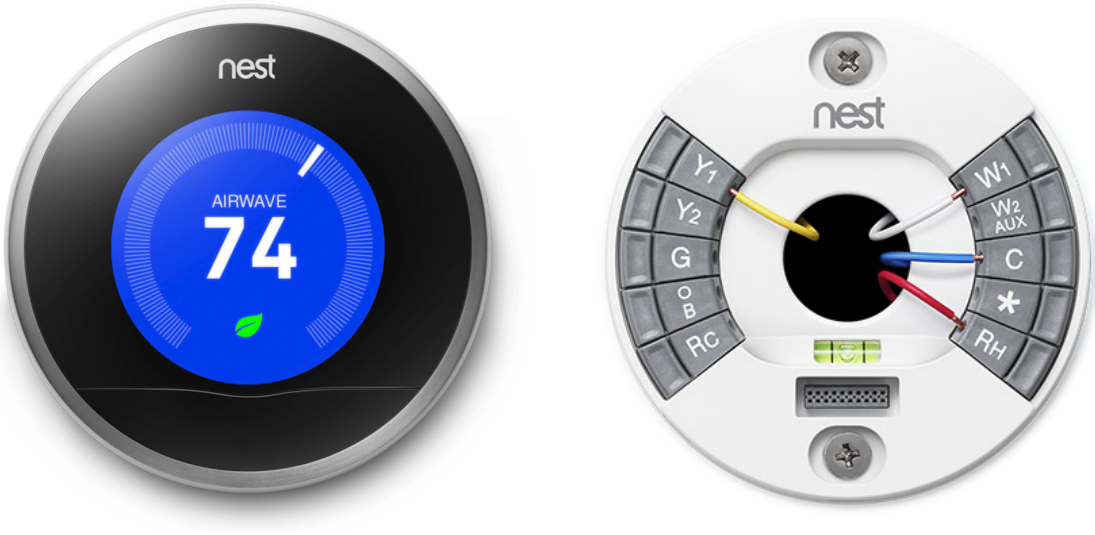

Figure 1: Front and back plates of a Nest Thermostat (credit: Nest)

2.3 Boot Process

Upon normal power on conditions, the Sitara AM3703 starts to execute the code in its internal ROM. This code initializes the most basic peripherals, including the General Purpose Memory Controller (GPMC). It then looks for the first stage bootloader, `x-loader`, and places it into SRAM. Once this operation finishes, the ROM code jumps into `x-loader`, which proceeds to initialize other peripherals and SDRAM. Afterwards, it copies the second stage bootloader, `u-boot`, into SDRAM and proceeds to execute it. At this point, `u-boot` proceeds to initialize the remaining subsystems and executes the `uImage` in NAND with the configured environment. The system finishes booting from NAND as initialization scripts are executed, services are run, culminating with the loading of the Nest Thermostat proprietary software stack. Figure 2 shows the normal boot process of the device.

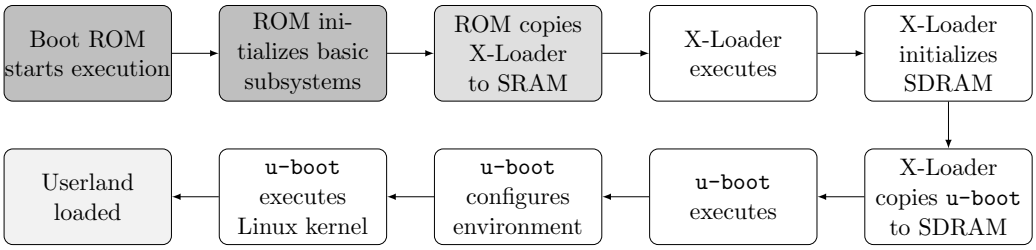

Figure 2: Standard Nest Thermostat boot process

3 The AM3703 - A Closer Look

The TI AM3703 microprocessor is composed of a 32 Channel DMA controller, a dual-output three-layer display processor, High Speed USB controller with USB OTG capabilities, an emulation module for debugging, a General Purpose Memory Controller (GPMC) to handle NAND/NOR flash, an SDRAM memory scheduler and controller, an 112KiB on-chip ROM which contains boot code, a 64KiB on-chip SRAM all connected by a Level 3 (L3) interconnect which runs at 200MHz.

sys_boot[5:0]	First	Second	Third	Fourth	Fifth
001101	XIP	USB	UART3	MMC1	
001110	XIPwait	DOC	USB	UART3	MMC1
001111	NAND	USB	UART3	MMC1	
101101	USB	UART3	MMC1	XIP	
101110	USB	UART3	MMC1	XIPwait	DOC
101111	USB	UART3	MMC1	NAND	

Table 1: Selected boot configurations.

The ARM core within the MPU subsystem uses a 256KiB cache to reach the L3 interconnect. Furthermore, a Level 4 interconnect adds the peripheral module to the memory map. This peripheral module handles the GPIO, UARTs, high speed multimaster I^2C bus, memory card controller, memory stick pro controller, watchdog timer, general purpose timers and other miscellaneous subsystems.

The ARM subchip integrates an ARM Cortex-A8 core, with Version 7 of the instruction set architecture, providing standard ARM instructions and Thumb-2 mode, the JazelleX Java accelerator and media extensions. It also integrates an ARM NEON core SIMD coprocessor. It connects to a 32KiB/32KiB instruction/data caches which proceeds to interface with a 256KiB 8-way associative cache supporting parity and ECC. The core also provides integrated trace and debug features. A simplified memory map of the device is shown in Figure 3.

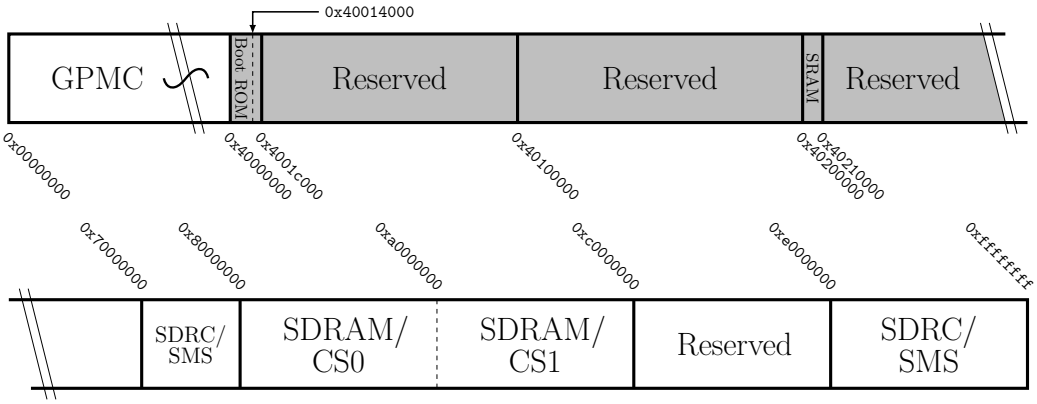

Figure 3: Simplified memory map (shaded areas are internal to the AM3703)

3.1 Device initialization

Power connections, clock and reset signals must be properly initialized before the AM3703 boots. The device boot configuration is given by six external pins, `sys_boot[5:0]`. After power-on reset, the value on these pins are latched into the `CONTROL.CONTROL_STATUS` register. Table 1 describes the boot selection process for select configurations.

After performing basic initialization tasks, the on-chip ROM may jump into a connected execute in place (XIP) memory, if the `sys_boot` pins are configured as such. This boot mode is executed as a blind jump to the external addressable memory as soon as it is available. Otherwise, the ROM constructs a boot device list to be searched for boot images and stores it in the first location of available scratchpad memory. The construction of this list depends on whether or not the device is

booting from a power-on reset state. If the device is booting from a power-on reset, the boot configuration is read directly from the `sys_boot` pins and latched into the `CONTROL.CONTROL_STATUS` register. Otherwise, the ROM will look in the scratchpad area of SRAM for a valid boot configuration, if it finds one, it will utilize it, otherwise it will build one from *permanent devices* as configured in the `sys_boot` pins.

Closer scrutiny of this process yields that under some circumstances, it is possible to boot the processor from a peripheral device, such as UART or USB. This is the method we are using to insert our code into the Nest Thermostat long before the firmware verification can take action.

4 Attack Vector

A global reset of the device can be triggered by pressing its button for about 10 seconds. Among other things, this causes the `sys_boot5` pin to go high, triggering peripheral booting. Coincidentally, the `sys_boot5` pin is directly exposed in an unpopulated header within the main circuit board, which can be utilized to directly trigger the USB booting behavior. Since the ROM does no cryptographic checks of the code being loaded, it freely executes this code, allowing total control of the device. A few limitations to this process must be observed. There is a strict timing window in which the ROM will be listening for any incoming program data, the initial payload must be `x-loader`, which is copied to SRAM and must initialize all remaining subsystems. Subsequent payloads must be able to fit and execute in SDRAM.

4.1 Initial attack

Our initial attack consisted of sending `x-loader` to the device by means of USB booting along with a custom `u-boot` image and a `ramdisk` with our final payload. Our `u-boot` was configured to boot the on-board kernel, utilizing the `ramdisk` as an initial root filesystem enabling us to automate our backdooring of Nest's root filesystem. The payload mounted the device's filesystem and using an already existing `netcat` binary we created a way to obtain a shell by modifying the boot scripts of the Nest Thermostat. This was demonstrated in our video which can be viewed at https://www.youtube.com/watch?v=7AnvTgAKa-g.

Using this backdoor, we were able to start exploring the filesystem of the device, attempting to reconstruct some "missing" items from userland, namely, the version of the C library used and any information on the toolchain used to build it. This information became available once the C library on the system was found and explored. Once the toolchain was rebuilt, we were able to run our own software on the device. Loading the software into the device was done using one of three ways: either using `netcat`, a custom `ramdisk` that executed the aforementioned attack, or the fact that the Nest Thermostat will identify and be treated as a USB mass storage device once fully booted and connected to a PC.

4.2 Refining a Backdoor

With a toolchain at hand, we cross-compiled `dropbear`, an SSH server, and installed it on the device. A user account was added to the unit by performing the required modifications in the `/etc/passwd`, `/etc/shadow` and `/etc/groups` files. On modifying the initialization scripts, we noticed that the Nest Thermostat attempts to start a secure shell server on its own, the binaries for which is not present within the filesystem. Further forensic analysis on this path yielded a pair of RSA and DSA host keys that were generated at some point. Checking multiple units showed that these keys are unique.

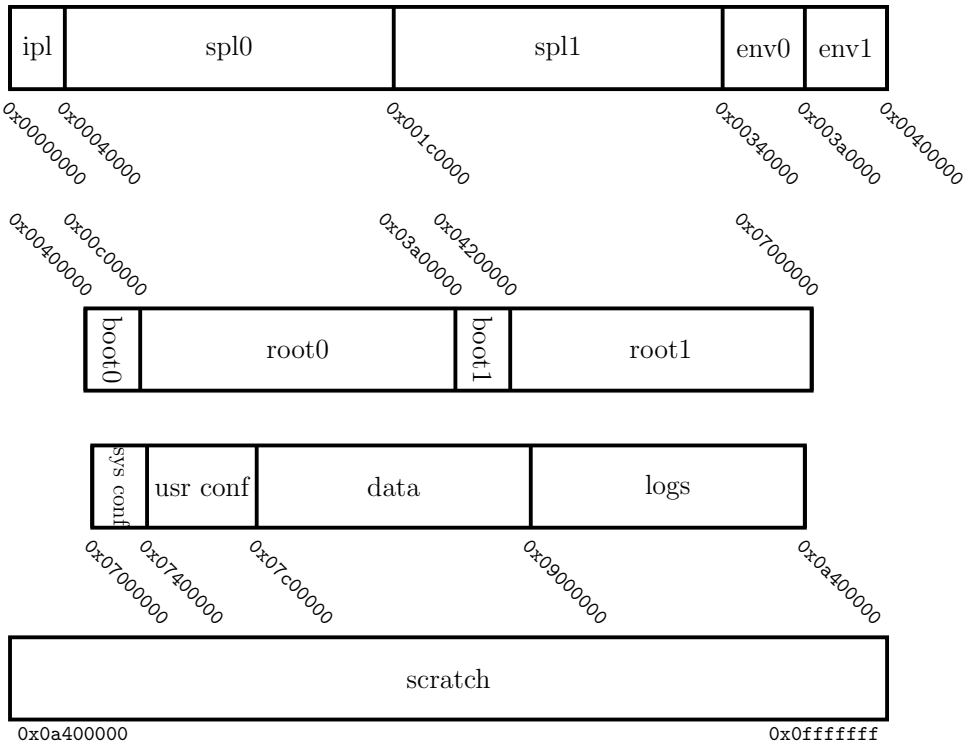

Figure 4: NAND layout

Having a secure shell server running within the device provided easy access to it within the local network. The issue at hand is that these networks are usually behind a NAT firewall, which does not allow for remote access without user intervention. We proceeded to develop a proof of concept client software running within the Nest Thermostat and installing it using the backdoor we had found. This client software would attempt to connect to a remote server and relay a message, proving the possibility of establishing the Nest Thermostat as part of a large botnet.

4.3 Modifying the Linux Kernel

Further analysis within the device yielded the presence of the compressed kernel configuration file within the `/proc` filesystem. To further study the device's operation, our research team has started to develop a custom Linux kernel based on the published sources with debugging features added. To avoid having to upload a kernel every single time we turned to `u-boot` for information on a possible location to permanently store it. The layout of NAND is shown in figure 4.

Patches had to be added to the kernel tree to allow polling in the OMAP serial drivers, allowing us to connect the enabled `kgdb` interface in the device and trap certain behavior. We have permanently written this kernel to the unit under the `boot1` section of NAND. We then use a custom `u-boot` to select which kernel to boot from upon powering on the unit.

Periodic updates to the Nest Thermostat firmware are provided by Nest Labs. At the time of writing, the latest firmware version available is 4.2.3. Since the Nest updater runs as root, it has complete access to the device's filesystem, meaning it can attempt to delete the backdoor client presented in the previous section. By using a custom kernel, we can insert our own modifications which protect our software from ever being modified. Controlling the kernel space means that we can control userland; that is, we are able to deploy a rootkit into the device if we so desire.

5 Consequences

A Nest Thermostat, as demonstrated, may easily be compromised during transport, deployment, or by an attacker having access to it on a non-secure location. As demonstrated above, it can then become a client on a botnet. Persistent rootkit installation is possible using our `ramdisk` method and a customized Linux kernel written into the unit. The customized Linux kernel would be used to hide the botnet software, which may remotely control the thermostat, transforming it into a beachhead for a remote attacker.

Furthermore, the very fact that the compromised Nest Thermostat sits in the network can be used to introduce rogue services. If a network uses DHCP internally, the Nest Thermostat can be made to act as a DHCP server. When a client node tries to resolve an IP address and DNS server using this protocol, a response can be given to have the client node use our own DNS server, thus having control of what addresses get resolved and to what IP. The Nest could also spoof ARP packets to masquerade as the router, allowing the capture of a targeted computer's network traffic.

The Nest Thermostat backdoor is exacerbated by the fact that local network credentials are stored in the device and become accessible to our client software. The extraction of these credentials from the device imply that we could deploy other rogue devices into the local network, further shaping local traffic and scanning for exploitable vulnerabilities other devices in the network may have. Attackers can pivot from the Nest Thermostat to other devices on the network. Suddenly, what was once a learning thermostat has been transformed into a spy that can not only report on the routines of the inhabitants of a certain home or office, but also on their cyber activities and provide a backdoor to their local network which could go unnoticed.

However, viewing this issue with a positive light, the aforementioned backdoor also provides legitimate users with an approach prevent Nest from collecting their personal data without their consent and to extend the functionality of the device itself. This actually opens a controversial discussion whether we should hack our own purchased devices in order to protect our own privacy and to add features manufacturers do not include.

6 A Possible Solution - Chain of Trust

The following is a proposed solution to the inherent problem of being able to compromise the Nest Thermostat.

In order to establish a proper chain of trust, the device must authenticate the code that is being run within it from the start. This means that the microprocessor must authenticate the first stage bootloader, in this case `x-loader`, which in terms must authenticate the code it attempts to run. Further, any code that is run in the unit must be authenticated by its launcher. The processor used in the Nest Thermostat does not support this feature, which Texas Instruments terms *secure boot*. That same supplier, however, provides custom cores with secure boot as an option, but they are not an "off the shelf" solution. Minimum order quantities must be met and there are also setup fees, which may not be desired for a startup with limited capital and time to market.

Secure boot authenticates the first stage bootloader with a key embedded into the main processing unit. Embedding keys onto a device's die is not unheard of, as many consumer devices, such as gaming consoles and modern UEFI-based secure boot personal computers do as such. It is imperative, however, that these keys remain in a secure location, outside the reach of the external software stack of the device if symmetric algorithms are used. If asymmetric verification algorithms are used, this last requirement is not as important.

A question can be raised here on how to protect userland. There are two main approaches to

this matter. One is to load and execute only signed binaries. This requires the kernel to have a custom loader that verifies these binaries before they are executed. Another approach is to encrypt the filesystem, this way, an attacker will not be able to externally modify it without first decrypting it. If this last approach was to be taken, decryption of the filesystem must be performed live within the microprocessor. Since the processor must be customized to support secure boot, an engine to perform on-the-go encryption and decryption of the filesystem can be used to hasten the process.

Further, it becomes necessary to protect the microprocessor-SDRAM channel, as it can be attacked as well. Marking pages as executable (nx flags) and performing on-the-fly encryption for pages becomes a necessity. Again, this encryption must be performed within the microprocessor and must be transparent to the software so a hardware based approach similar to the above can be used.

7 Conclusion

After a detailed analysis of the hardware infrastructure of the Nest Thermostat, we identified a backdoor associated to the boot process, which, as we demonstrated, can be leveraged by attackers to install malicious firmware. Since the attack happens before the on-board userland is loaded, the firmware verification employed is unable to detect and stop the intrusion. The resulting payload can potentially allow attackers to shape local network traffic from a remote location, further compromising other nodes.

Beyond the Nest Thermostat, we believe that most of the current IoT and wearable devices suffer from similar issues, lacking proper hardware protection to avoid similar attacks. As future work, we will further analyze the WiFis and ZigBee modules to check whether a backdoor exists in these communication channels, allowing for a remote attack.