practice

DOI:10.1145/3511664

Article development led by acmqueue
queue.acm.org

A deleted private key, a looming deadline, and a last chance to patch a new static root of trust into the bootloader.

BY PHIL VACHON

The Keys to the Kingdom

IT WAS A warm, late summer afternoon when a long-time client called. They needed someone to help them out of a jam—and fast. This client builds embedded devices found in offices around the world. Their latest creation had all the right security features, the best possible given their hardware constraints. These devices are driven by firmware running on a microcontroller that delivers robust wireless communications; I knew this was my kind of job.

The client's engineers had attempted to build the right security features into their firmware: a bootloader that normally can't be updated, a static root of trust contained therein, and cryptographically signed binary firmware updates. They had even hardened their bootloader to defend against anyone attacking the firmware update protocol directly—countermeasures against people like me. The client was proud of how secure the design was; after all, most consumer products at the time had barely figured out firmware readout protections.

An unlucky fat-fingering precipitated the current crisis: The client had accidentally deleted the private key needed to sign new firmware updates. They had some exciting new features to ship, along with the usual host of reliability improvements. Their customers were growing impatient, but my client had to stall when asked for a release date. How could they come up with a meaningful date? They had lost the ability to sign a new firmware release.

Given the sheer number of these devices in the field, as well as the cost per unit, a replacement program was the option of absolute last resort. The financial loss would be huge, and orchestrating such a program would be incredibly daunting. A reverse-engineering attempt was the final "Hail Mary" effort before a recall would be necessary.

The task seemed straightforward: Find a way to patch a new static root of trust into the bootloader (a philosophical question: Is it that *static*?), thereby enabling the client to sign firmware updates with a new key. Since the bootloader was hardened (so my client claimed), direct attacks on the firmware updater were out of the question.

The device in question has multiple serial ports that expose a variety of complex protocols once the client's application starts. The main microcontroller in the design included an off-the-shelf Arm Cortex-M3 core, with built-in flash and on-board RAM. This microcontroller had no hardware root of trust, so the bootloader provided all of the firmware security features. No-

IMAGE BY CKA/SHUTTERSTOCK

tably, there was nothing to ensure the bootloader wasn't modified.

The client did not have the exact source code for the firmware shipped on the device, since the entire release was lost during the fat-fingering. Worse, I could not easily use a production device to test this (it *could* be done, but would have added a lot of time to the engagement—time the client could not afford), and the client went to great lengths to protect the device against firmware read-out, even disabling the Joint Test Action Group (JTAG) debug port. To make things even more challenging, release firmware images would not emit debugging log messages. One fact worked in my favor, however: The firmware was built on top of open source components. Both the bootloader and the realtime operating system were used by many such projects.

Since time was of the essence, and static analysis of the production binary would take a lot of time, I needed to find a quick path to code execution. All I had was the compiled release version of the firmware and a new version of the source code the client wanted to install on the deployed devices in the field. The new code was a point from which to hunt for flaws in the communications protocol handling code; the client claimed this code had not changed substantially. I was used to working with less.

UARTs Know No Bounds

One universal asynchronous receiver/transmitter (UART), or serial port, exposed a framed command protocol. The handling logic worked as follows:

- As bytes arrive, the interrupt handler read out of a FIFO (first-in, first-out) hardware queue into a buffer until a FRAME END byte was received. An array of eight of these buffers was allocated as one contiguous array. There was no check to make sure the input message did not exceed the length of one of these buffers.
- After receiving a FRAME END byte, the interrupt handler pushed a pointer to the filled buffer, as well as the length of the payload (as an unsigned 32-bit integer) into the protocol handler thread's work queue. This signaled to the thread that data was ready to process.

To make things even more challenging, release firmware images would not emit debugging log messages.

- The protocol handler thread popped the new work item from its queue and copied the received bytes into a 1,024-byte work buffer. The number of bytes to copy was also not checked against the work buffer size.

There were a couple of interesting flaws. First, the interrupt handler code did not have logic to check that bytes were not being written past the end of its active buffer. Second, the protocol handler thread blindly accepted a payload length from the input buffer, copying whatever it was told, without checking. This bug could be used to copy a malicious message over other data adjacent to the work buffer.

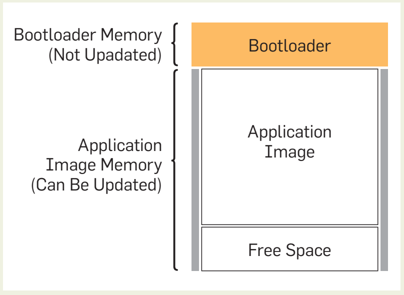
The next step was to load the released firmware image onto the standard development kit sold by the microcontroller's vendor. I needed to figure out a way to load my own code. Some quick experimentation showed that sending an excessively long message—10 KB of the value `0x4f`—caused the device to seize up. Success! But why did it crash?

The Jump Off `0x4f4f4f4f`

Having some version of the source code in hand and another similar version of the firmware running on a development kit meant I could start debugging. A quick check showed the device threw a bus fault when it failed. This happens on a Cortex-M CPU when trying to read, write, or attempt to execute instructions from an invalid address. The device's status registers indicated an invalid instruction fetch occurred at `0x4f4f4f4f`. This was good news for me, as I could now take control of the program counter.

The Cortex-M family of microcontrollers is designed to be easy to target with ordinary C code, requiring minimal assembly-language glue. Interrupt service routines (ISRs) are normal C functions that are called directly by hardware. When handling an interrupt request, logic built into the CPU core prepares a stack frame on the running process's stack. That frame stores the interrupted process's registers, as well as information about the CPU state and the instruction pointer value at the time of the interrupt. The CPU then switches to a separate interrupt mode stack pointer, and invokes

Figure 1. The use of Flash memory in the device.


Bootloader Memory (Not Upadated)
Bootloader
Application Image Memory (Can Be Updated)
Application Image
Free Space

the ISR. If you can overwrite the contents of the saved process context—especially the instruction pointer that was saved on interrupt entry—then you can tell the CPU to return to some different code later.

Now, all I needed to do was replace my buffer full of `0x4f` bytes with the address of some code I wanted to run. The next time the process woke up, the CPU would jump right to the address it read from this saved context.

Bootloader Controls are Essential

Controlling code execution is one problem, but I also needed to store the code that rewrites the root of trust somewhere. Many microcontrollers offer up to 1MB of flash memory, a luxurious amount of storage if you're an embedded systems developer. RAM is a more precious commodity—a few tens of KB in many cases. All code on this device can be run straight from flash, leaving RAM for CPU state and data structures. This means, however, that there is not enough memory to hold a complete program image when performing a firmware update.

Flash memory was split into two partitions on this device: bootloader memory and application image memory. Figure 1 depicts the use of flash memory. (Note that the firmware updater is built into the bootloader and cannot update itself.) Figure 2 shows the application image structure from a legitimate application image update. Figure 3 shows the application image's security features, plus some added code and data.

The updater built into the bootloader erases the entire application image memory and writes a new image in its place. Once the full image is ready in flash, the bootloader checks the signature of the application image, including the contents of the header, verifying its authenticity. If the signature check were to fail, the bootloader would immediately erase the data just downloaded. Modifying the firmware image was out of the question, because the signature check would fail. What else could I do?

The updater was simple. As long as you kept feeding it data, it would write the incoming data to flash.

Reviewing the code of the open source bootloader that the client had used showed a bug that could be of use: The signature check was performed only on the code region specified in the header. As long as the original header, code, and signature were unmodified, the bootloader would boot the image. A quick test proved this to be the case. An image with extra data appended booted successfully, with the extra data being ignored. Since all flash memory on this device is executable, I could simply jump to extra code appended to a valid update image.

So much for all that bootloader hardening …

Figure 2. Application image structure.

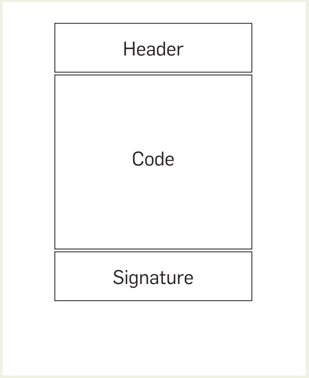
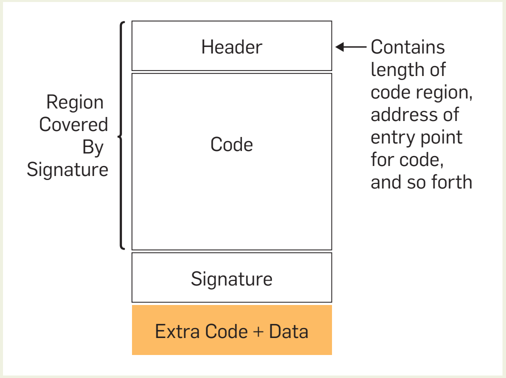


Figure 3. The application image's security features.


Region Covered By Signature
Header
Contains length of code region, address of entry point for code, and so forth
Code
Signature
Extra Code + Data

Rewriting the Bootloader

The last step was to write a payload that would "enhance" the bootloader to validate application image signatures using a new public key from the client. My payload was simple: Erase the original public key from flash and write the new key in its place. On subsequent reboots, the bootloader would accept new firmware images signed with the new key—one the client now keeps in a couple of *safe* places.

Victory!

My client's "Hail Mary" effort paid off. They soon shipped a firmware release with new features and fixes. Their clients were none the wiser that the original signing keys had been lost and that the new firmware image had been installed by taking advantage of bugs I found while reverse-engineering the device. The new firmware release also included fixes for these bugs.

Would you believe me if I told you this job was not unique? I have had this very situation play out for at least three different clients, all of whom were in the same jam. After delivering my "fix," I always follow up by advising my clients how to store and manage firmware signing keys. Since these are the keys to their devices, they deserve to be treated with respect—both for device lifecycle and security reasons.

Many commodity microcontrollers today offer a static root of trust, built into a boot ROM in silicon. In this case, pulling off such a hack would be a lot more difficult, making it even more important to protect the keys. Also, had the client's design used the memory-protection features offered by the Cortex-M series, this job would have been even more challenging.

Fortunately, while I have helped many clients with this problem, none has asked for this type of work more than once. Lesson learned?

Phil Vachon is a security architecture and engineering manager with Bloomberg's Office of the CTO, where he and his team work on projects related to identity, authentication, and the application of data science to operational security challenges. Previously, he co-founded a startup focused on high-speed packet capture and analysis. He has also developed high-frequency trading systems, designed and implemented firmware for identity and security infrastructure devices, built synthetic aperture radar data-processing tools, and worked on data-plane traffic engineering for carrier routers.