NCC Group Whitepaper

Microcontroller Readback Protection: Bypasses and Defenses

February 20, 2020

Author

Sultan Qasim Khan

Abstract

Microcontrollers commonly include features to prevent the readout of sensitive information in internal storage. Such features are commonly referred to as *readback protection* or *readout protection*. This paper describes common readback protection implementation flaws, discusses techniques that can be used to defeat readback protection, and provides guidance to implement effective readback protection.

Table of Contents

Introduction

Modern microcontrollers containing internal flash memory commonly provide a mechanism to prevent the reading of this storage through the debugging interfaces. This defense mechanism is known as *readback protection*, or sometimes simply *read protection*. Readback protection is an important defensive feature because a microcontroller's internal flash may happen to store data with confidentiality requirements that should not be readable by unauthorized parties. Examples of sensitive data include the vendor's proprietary firmware, DRM keys, or even user secrets. Developers may wish to restrict attackers from being able to extract data from microcontrollers, even when the attackers have physical access to the device.

Many different mechanisms are used by microcontroller vendors to provide data readback protection, including:

- Disabling debugger access completely
- Preventing debuggers from reading memory-mapped flash addresses while otherwise leaving the debugging capability intact
- Preventing flash reads by any internal bus master other than the CPU instruction fetch mechanism
- Disabling flash reads within bootloaders
- Encrypting the contents of flash

Protection mechanisms such as these are often coupled with flash write-protect mechanisms to provide basic integrity guarantees, and help prevent unauthorized modifications to code and data.

When implemented and used correctly, readback protection mechanisms can prevent the casual dumping of microcontroller flash contents using off-the-shelf debuggers. With sufficient effort and resources, all microcontroller data protection mechanisms can be defeated by invasive methods (involving decapsulation: removal of packaging to attack the silicon directly). Many can also be broken through non-invasive fault-injection attacks (such as power or clock glitching). Furthermore, many microcontroller data protection mechanisms and bootloaders have implementation or logical flaws that allow circumvention without any sophisticated attacks at all. Some developers even attempt to implement readback protection on microcontrollers that inherently do not support it, either by misunderstanding silicon vendor intent and documentation, and/or not planning for readback protection in the component selection process.

This paper discusses common weaknesses in microcontroller data protection mechanisms, and provides recommendations to improve them. It is targeted at both security professionals and developers of products that need to protect secrets in non-volatile microcontroller memory. Specific examples of microcontroller data protection mechanisms that are susceptible to publicly documented attacks are discussed in detail. While attempts are made to quantify the costs of circumvention, this should be considered a point-in-time, as tools and techniques continue to evolve and reduce these costs. It should also be noted that while initial attack discovery and development can be a time consuming and/or costly process, the cost and required effort of repeating a previously developed attack is generally much lower.

Sections 3-6 of this paper describe common classes of vulnerabilities that defeat readback protection:

- Debug Pins Reconfigured for Alternate Functions on the following page
- Partially Restricted Hardware Debug Interfaces on page 6
- Independently Erasable Memory Blocks on page 9
- Bootloader Interfaces on page 10

Sections 7 and 8 describe advanced attack techniques than can bypass readback protection:

- Fault Injection on page 12
- Invasive Attacks on page 16

Sections 9 and 10 provide guidance for implementers to protect data stored in their microcontrollers:

- Protecting External Flash Contents on page 18
- Implementing Secure Bootloaders on page 19

Debug Pins Reconfigured for Alternate Functions

Microcontrollers commonly provide a mechanism to reconfigure hardware debug interface pins for alternate functions. JTAG[1] and Serial Wire Debug (SWD)[2] pins are often multiplexed with GPIO (General Purpose Input/Output) interfaces, ADCs (Analog to Digital Convertors), timer inputs, and a multitude of other alternate functions. Although pin multiplexing and remapping functionality allows debug pins to be allocated for alternative functions, it is usually not intended to be a security mechanism.

NCC Group frequently sees embedded systems remapping debug interface pins at runtime under the mistaken belief that it provides security against debugging. This is not an adequate defense, as the default state of the hardware debug interface pins is usually to expose the debug interface, with alternate functions being available only after software has reconfigured them at runtime.

Attackers wishing to make use of debug interfaces that are repurposed at runtime can often do so by holding the processor in the reset state, or by attaching a debugger early during boot before the software has had an opportunity to reconfigure the pins. Resetting a processor that disables the hardware debug interface at runtime will automatically reactivate the debug interface. An attacker can then place a breakpoint early in execution, such as at the reset vector, and then de-assert the reset signal. The processor would start to run, then immediately halt before the debug interface is disabled. Once the processor is stopped on a breakpoint while debug interfaces are accessible, debug interfaces can be used to access RAM and flash unless disabled through some other means.

As an example, STM32F1 series microcontrollers[3] do not implement any debug disable or code readout protection mechanisms. STMicroelectronics does not advertise these components as having any such security features. However, these microcontrollers do support disabling or re-purposing debug interface pins at runtime by writing to the `SWJ_CFG` bits of the `AFIO_MAPR` register. The excerpt below from the STM32F1 series reference manual RM0008 describes the `SWJ_CFG` bits.

9.4.2 AF remap and debug I/O configuration register (AFIO_MAPR)

Address offset: 0x04

Reset value: 0x0000 0000

Memory map and bit definitions for low-, medium- high- and XL-density devices:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					SWJ_CFG[2:0]			Reserved			ADC2_ETRGREG_REMAP	ADC2_ETRGINJ_REMAP	ADC1_ETRGREG_REMAP	ADC1_ETRGINJ_REMAP	TIM5CH4_IREMAP
					w	w	w				rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PD01_REMAP	CAN_REMAP[1:0]		TIM4_REMAP	TIM3_REMAP[1:0]		TIM2_REMAP[1:0]		TIM1_REMAP[1:0]		USART3_REMAP[1:0]		USART2_REMAP	USART1_REMAP	I2C1_REMAP	SPI1_REMAP
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:27 Reserved

Bits 26:24 **SWJ_CFG[2:0]:** Serial wire JTAG configuration

These bits are write-only (when read, the value is undefined). They are used to configure the SWJ and trace alternate function I/Os. The SWJ (Serial Wire JTAG) supports JTAG or SWD access to the Cortex® debug port. The default state after reset is SWJ ON without trace. This allows JTAG or SW mode to be enabled by sending a specific sequence on the JTMS / JTCK pin.

000: Full SWJ (JTAG-DP + SW-DP): Reset State

001: Full SWJ (JTAG-DP + SW-DP) but without NJTRST

010: JTAG-DP Disabled and SW-DP Enabled

100: JTAG-DP Disabled and SW-DP Disabled

Other combinations: no effect

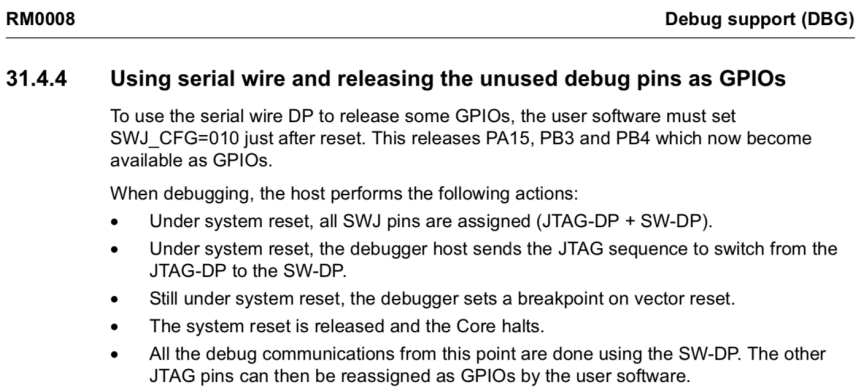
Figure 1: Documentation for SWJ_CFG bits in the STM32F1 series AFIO_MAPR register

[1] JTAG

[2] Serial Wire Debug

[3] STM32F1 Series

ST provides a hardware abstraction layer (HAL) macro called `__HAL_AFIO_REMAP_SWJ_DISABLE()` that writes to these bits to allow re-purposing debug interface pins for other signals. While ST does not intend for this function to be used for security, the documentation for this macro does not clearly indicate that it should not be relied upon as a security control and that it can be easily bypassed. Misguided developers may use this macro mistakenly believing that it securely disables debug to protect data stored within the microcontroller. As shown in the excerpt below, ST's own reference manual describes the trick of placing a breakpoint while in reset to work around runtime remapping of the debug interface.



RM0008 Debug support (DBG)

31.4.4 Using serial wire and releasing the unused debug pins as GPIOs

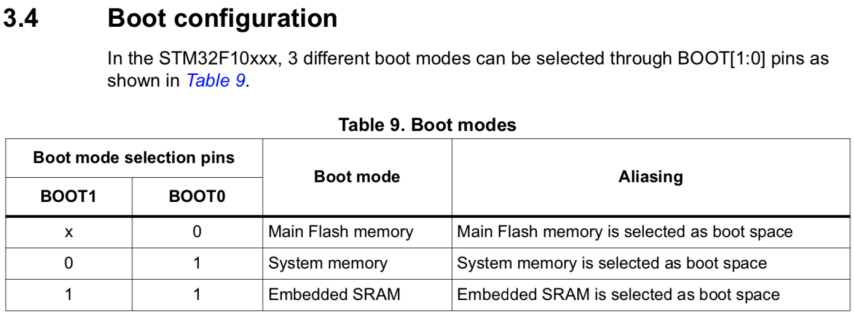
To use the serial wire DP to release some GPIOs, the user software must set SWJ_CFG=010 just after reset. This releases PA15, PB3 and PB4 which now become available as GPIOs.

When debugging, the host performs the following actions:

- Under system reset, all SWJ pins are assigned (JTAG-DP + SW-DP).
- Under system reset, the debugger host sends the JTAG sequence to switch from the JTAG-DP to the SW-DP.
- Still under system reset, the debugger sets a breakpoint on vector reset.
- The system reset is released and the Core halts.
- All the debug communications from this point are done using the SW-DP. The other JTAG pins can then be reassigned as GPIOs by the user software.

Figure 2: ST documentation describing how to debug a device with remapped debug pins

For many microcontrollers, boot mode pins can also be used to avoid running code that remaps debug interface signals. For example, a blogger named "Mark C."[4] has noted that rebooting an STM32F103 family microcontroller while holding the `BOOT0` pin high avoids running code on internal flash that remaps debug interface pins. Mark used this property to extract data from an STM32F103 based device that attempted to prevent flash readback through debug pin reassignment using the `__HAL_AFIO_REMAP_SWJ_DISABLE()` macro. The excerpt below from the STM32F1 series reference manual describes the boot mode pin functionality.



3.4 Boot configuration

In the STM32F10xxx, 3 different boot modes can be selected through BOOT[1:0] pins as shown in *Table 9*.

Table 9. Boot modes

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as boot space
0	1	System memory	System memory is selected as boot space
1	1	Embedded SRAM	Embedded SRAM is selected as boot space

Figure 3: Documentation describing boot mode pins on the STM32F1 series

In general, reassigning debug pins to other functions at runtime should not be considered an adequate protection against debug or data extraction. When readback protection is desired, the ability to completely disable debug interfaces in hardware should be a security requirement in the component selection process.

[4] How to bypass Debug Disabling and CRP on STM32F103

Many microcontrollers provide mechanisms that restrict debugger access to specific memory ranges and registers without completely disabling debug. Such mechanisms are usually intended to allow a limited degree of testing and debugging on production hardware without defeating readback protection or secure boot. However, these partial debug restrictions often leave logical holes that can be used to circumvent protections. NCC Group recommends disabling hardware debug interfaces completely to avoid the exposure of potential logical holes. If debugging functionality is required on production units, it should enforce strong authentication (as described in Implementing Secure Bootloaders on page 19) to prevent abuse.

The system bus masters in simple microcontrollers typically include CPU instruction fetch, CPU data fetch, Direct Memory Access (DMA) controllers, and processor debug. The debug mechanism usually also has access to CPU registers, and can manually halt and single-step the CPU through individual instructions. Most simple microcontrollers do not have a Memory Management Unit (MMU) to provide address translation (as used for virtual memory). Instead, software usually operates directly on physical addresses. Many, even simple, microcontrollers have a Memory Protection Unit (MPU) that can impose simple read/write/execute permissions on different address ranges. Because the MPU is typically implemented within the CPU core, these protections usually only apply to the CPU-initiated bus transactions and not to other bus masters such as the system memory debug Test Access Port (TAP) and DMA, which can allow the bypass of these MPU restrictions. The diagram below illustrates the architecture of a typical microcontroller, most closely resembling an ARM Cortex-M based device.

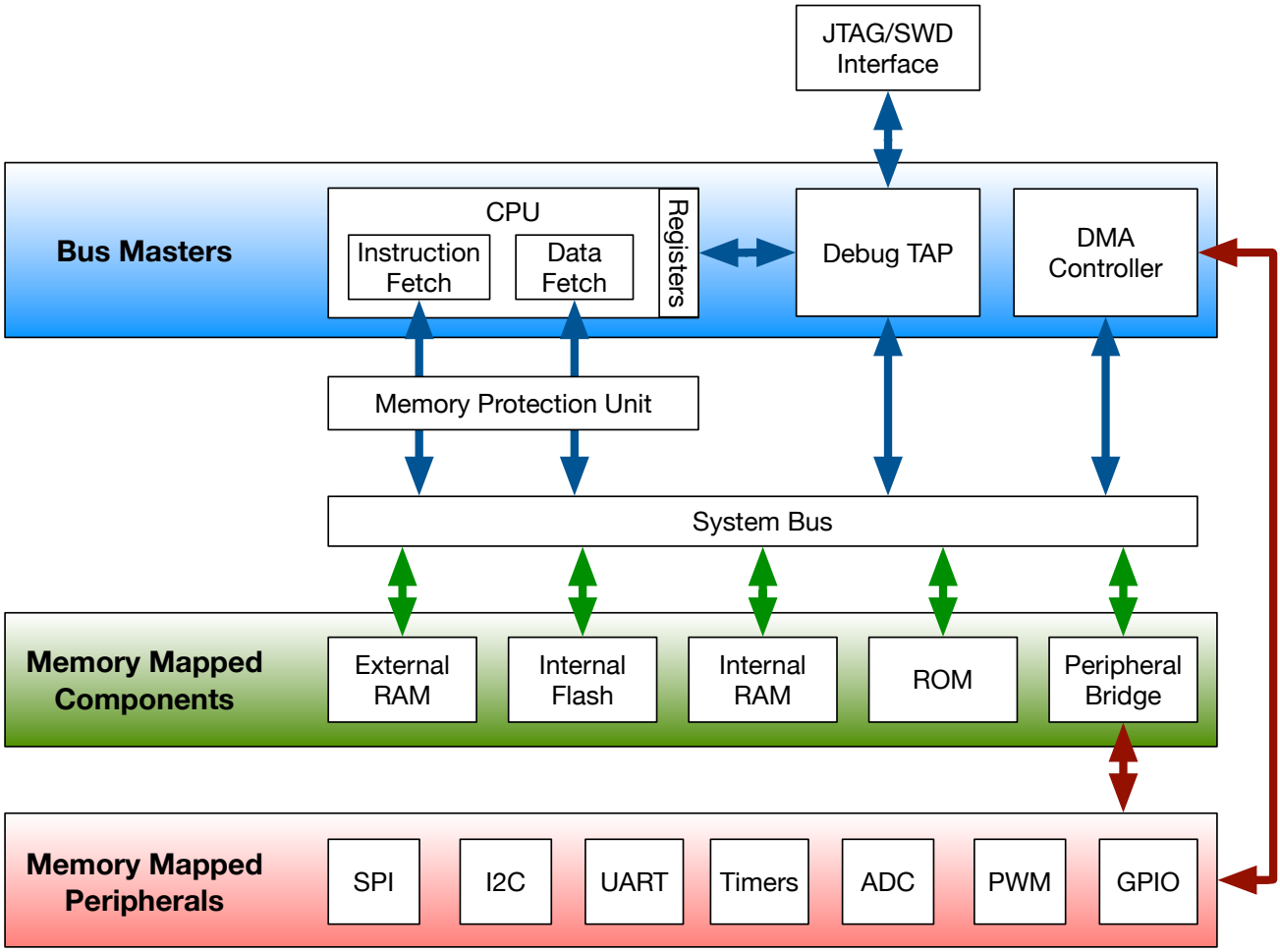


Figure 4: Architecture of a Typical Microcontroller

Within this architecture, there are many ways a debugger can be prevented from accessing protected regions of flash:

- **Disable JTAG/SWD altogether:** The most reliable way to prevent the abuse of debugging interfaces such as JTAG and SWD (Serial Wire Debug) is to disable them altogether. This includes disabling JTAG Boundary Scan and vendor-specific Design For Testing (DFT) features. Debug interfaces should be disabled using fuses or write-once bits internal to the microcontroller. Relying on easily manipulated external strap pins is generally insufficient.
- **Disable processor debug:** Many microcontrollers do not provide a mechanism to fully disable JTAG and SWD. Instead, they offer mechanisms to disable processor debug access ports such as the ARM AHB-AP that are accessible over interfaces like JTAG and SWD, while leaving intact other functionality such as JTAG Boundary Scan and sometimes vendor DFT features.
- **Block debugger access to certain memory regions and registers:** The debug disable mechanisms of some microcontrollers leave processor debug access ports active, and merely limit the capabilities of the debug access ports. Restrictions may include hardware blacklists of memory address ranges, and preventing the viewing or modification of select CPU registers.
- **Restrict flash access to specific bus masters:** Some microcontrollers protect flash contents by preventing flash reads by debug or CPU data fetch bus masters. Flash reads by the CPU instruction fetch mechanism cannot be disabled if code is executed in-place from flash. Some implementations also block flash reads by the DMA controller.

Restrictions less thorough than complete disablement of JTAG/SWD can leave logical holes to access sensitive data. Mechanisms that disable the CPU debug access port but leave other debug functionality intact may allow access to external flash and RAM via JTAG Boundary Scan, and possibly access to internal systems through vendor-specific DFT interfaces. Mechanisms that restrict debugger flash reads sometimes do not restrict RAM reads, exposing secrets in RAM copied from flash. Mechanisms that block flash and RAM access by debuggers may still allow processor register access, allowing the leaking of secrets by executing code. It may also be possible to use peripheral register access and Direct Memory Access (DMA) to bypass restrictions and expose secrets. Furthermore, hardware race conditions may prevent restrictions from working as intended. Partial restrictions on debug interfaces are often leaky abstractions.

An example of such a leaky abstraction is the PALL (Protect all) mechanism on the Nordic nRF51 series of microcontrollers for wireless applications.[5] This mechanism prevents debuggers from accessing flash and RAM address ranges. However, it still allows processor register access and single stepping of the processor. This weakness has been used by Kris Brosch[6] to dump the full flash contents of an nRF51 series microcontroller. He located a load instruction in code flash that reads data at the address contained in one register, and copied it to another register. He then repeatedly single stepped over this one load instruction, while manipulating the read address stored in the index register. By performing a load instruction with every address in flash, he was able to read the entire contents of flash memory. This flaw has been addressed by Nordic Semiconductor in the new nRF52 series, where setting the PALL bit now also blocks access to CPU registers.

Similar attacks exist against *execute-only memory* based readback protections, such as the Proprietary Code ReadOut Protection[7] (PCROP) on many STM32 family microcontrollers. Execute-only memory mechanisms allow marking certain memory regions as readable only by the CPU instruction fetch mechanism, and no other bus masters. Execute-only memory restrictions prevent the read of protected regions by the CPU data path and the debugger interface. However, it is still possible to recover instructions in execute-only memory regions by single stepping and observing changes to the system state. Single stepping can either be performed with a debugger, or by code executing on the device using interrupts. Code recovery using such techniques has been described and demonstrated by Schink and Obermaier.[8] For execute-only memory to be effective in preventing instruction recovery, it would need to be combined with the disablement of debugging interfaces, as well as preventing execution of untrusted code on the device. However, this would defeat the purpose of execute-only memory, which is normally to allow protecting specifc regions of code (such as proprietary third-party libraries), while otherwise allowing the execution and debugging of untrusted code.

[5] nRF51822
[6] Firmware dumping technique for an ARM Cortex-M0 SoC
[7] PCROP is different from and complementary to the RDP (ReaDout Protection) feature of STM32 microcontrollers.
[8] Taking a Look into Execute-Only Memory

Hardware race conditions in data protection mechanisms have also been found and described in public research. For example, the popular STM32F0 series microcontrollers[9] have a race condition allowing flash reads over SWD when in *Readout Protection Level 1* (partially restricted debug). Johannes Obermaier identified a flaw[10] where restrictions on debugger access require two system bus (ARM AHB) cycles to take effect after the start of the first debugger bus access. When system bus load is low, this race condition allows leaking one word of data from an arbitrary memory address. This race condition can be used repeatedly to extract all data from flash memory. Upgrading to *Readout Protection Level 2* (which disables debug completely) prevents this race condition from being exploited.

It may take an experienced researcher weeks or months to identify a weakness and develop an exploit. However, once an exploit is developed, attacks against vulnerable partially restricted debug interfaces can generally be performed in a few seconds or minutes using debugging equipment under $100 USD, such as the Black Magic Probe.[11]

[9] STM32F0 Series
[10] Shedding too much Light on a Microcontroller's Firmware Protection
[11] Black Magic Probe Wiki

Independently Erasable Memory Blocks

Some microcontrollers segment their memories into multiple regions or blocks, where the protection on separate regions can be configured independently. This is a useful feature for separating things like bootloaders, firmware, and data. Such separation allows better adherence to the principle of least privilege.[12] However, some microcontrollers allow independently erasing and removing protections from individual blocks without erasing all other blocks. The ability to independently remove protections from specific blocks without wiping all other blocks is a major flaw that allows trivially bypassing data protection. Independently erasable memory blocks allow replacing specific firmware components with versions that allow data exfiltration, while leaving data in all other blocks intact.

For example, older models of the Microchip PIC18F series are known to have this flaw. PIC18F microcontrollers from the K22 series[13] and older have multiple code flash blocks that can be erased independently. They have a boot block, two or more main code blocks, and separate blocks for data storage. Each block can be erased and unprotected individually without wiping the rest. To exploit this situation, an attacker can wipe the boot block and load their own code there, while leaving main code storage and data storage untouched. The attacker's code in the boot block can then read all other memory and exfiltrate it. To read the original boot block contents, an attacker can take a second untouched MCU, replace the main code blocks with their own code, have the original boot block jump into their code, and then have their own code read out the contents of the boot block. This technique was used by Milosch Meriac[14] to read out the protected code of a PIC18F452 inside an RFID card reader. It should be noted that the technique to read the boot block is contingent on the original boot block not securely verifying the integrity of the other code flash blocks.

Microchip fixed this flaw in the PIC18F K40 series[15] and newer. The newer models from the PIC18F K40 series and up do not allow independently erasing and clearing protections from individual flash blocks. However, the vulnerable PIC18F K22 series and older are still promoted, sold, and commonly used. Some older models from the Microchip PIC16F series (such as the PIC16F630) are also subject to a similar form of this issue, where code flash can be erased and unprotected independently of data EEPROM.

When flash erase mechanisms are implemented in a manner vulnerable to the methods described above, the required time and cost to conduct an attack are minimal. Once developed, such flash readback attacks can be conducted with only a few minutes of time and typically under $100 of debugging equipment.

[12] Principle of least privilege
[13] PIC18F K22 Series Datasheet
[14] Heart of Darkness - exploring the uncharted backwaters of HID iCLASSTM security
[15] PIC18F K40 Family

To support firmware updates on devices, OEMs often make use of a bootloader; a small module of code that executes on boot before the rest of the firmware. In addition to boot validation and firmware update functionality, embedded bootloaders may include debugging functionality including memory read/write access, hardware tests, and operations to verify programming is successful (such as hashing regions of flash). For security reasons, privileged bootloader interfaces should be protected by a strong authentication method to restrict dangerous functionality to authorized individuals, such as manufacturing, and Return Merchandise Authorization (RMA) operations for failure analysis and repair. Unfortunately, many embedded bootloaders contain inadequate authentication schemes, logical flaws, or side-channel leaks that can be used to defeat memory readback protections.

Authentication Issues

To minimize attack surface, any debug or diagnostic functionality not intended for use by normal end users should require authentication. Many bootloaders lack any form of authentication altogether, and provide privileged functionality that can trivially defeat readout protection, such as arbitrary flash or RAM read/write. Another common but problematic form of authentication is the use of static passwords to activate debug or diagnostic functionality. Such static passwords can leak, or may be present in update binaries. Several embedded bootloaders have authentication mechanisms with side channel information leaks that allow accelerated brute forcing of credentials, as discussed later in the *Side-Channel Leaks* subsection below. Some bootloader authentication mechanisms may have logic errors that allow bypassing authentication altogether. The proper way to authenticate access to bootloader debug functionality is with an asymmetric challenge/response scheme, such as the one described in Implementing Secure Bootloaders on page 19.

Logic Errors

Incorrect or incomplete logical checks in bootloaders can allow bypassing protections against flash read and write. One common class of bootloader logic errors is the incomplete or incorrect application of address blacklists. Many embedded bootloaders provide functionality to read and/or write memory contents at a specified address. To protect against malicious users, bootloaders often make use of blacklists that prevent access to sensitive regions. NCC Group has observed many cases of embedded bootloader address blacklists being incomplete or incorrect. Common blacklist issues include:

- Failing to account for memory addresses being aliased to another region
- Reusing blacklists from a different or older product
- Blacklists failing to prevent access to peripherals that can operate as bus masters and directly access sensitive regions (DMA for example)

One publicly documented case of an incomplete blacklist is in the bootloader of the Ledger Nano S[16] cryptocurrency hardware wallet. Nedospasov, Datko, and Roth presented in 2018 a technique to defeat Secure Boot checks on the hardware wallet by writing to an aliased address omitted by a bootloader blacklist check.[17] The wallet bootloader stored a magic value of `0xF00DBABE` in flash when the integrity of firmware had been verified, and cleared this magic value when firmware was modified. While blacklist checks in the bootloader prevented direct writes to the main address for this magic value (`0x08003000`), the bootloader failed to prevent modification to the alias of this location at address `0x00003000`. This allowed marking arbitrary firmware flashed to the device as valid, allowing arbitrary firmware to be executed. The execution of arbitrary firmware on a microcontroller would also inherently allow the reading and exfiltration of any secrets in internal flash.

A similar case of an insufficient address blacklisting allowing bypass of readback protection on the NXP LPC1343. Temeiza and Oswald discovered that memory write address restrictions in the LPC1343 boot ROM do not prevent writes to the bootloader stack in CRP1 (Code Read Protection level 1).[18] In CRP1, the LPC1343 boot ROM disallows writes below address `0x10000300`, but the blacklist omits the stack area that grows backwards from `0x10002000`. By writing to the bootloader stack using the insufficiently restricted write command, it was possible to use return oriented

[16] Ledger Nano S
[17] WALLET.FAIL
[18] Breaking Bootloaders on the Cheap

programming (ROP) to exploit the bootloader and read arbitrary flash contents on a CRP1 protected device.

Another example of a bootloader logic flaw seen repeatedly by NCC Group involves functionality to compute a hash of a flash region for verification purposes. Rather than allowing unrestricted read operations (a security concern), bootloaders frequently support hashing segments of microcontroller flash to ensure correct firmware has been loaded. However, if the bootloader allows the user to control the address being hashed or the number of bytes being hashed, then memory contents could be recreated using hashes of memory at user controlled offsets or sizes. For example, if individual bytes can be hashed, a simple lookup table can be used to recreate the original data. If the offset or byte count being hashed is controllable, an iterative approach can efficiently determine the next byte's value by computing the 256 possible hashes corresponding to the 256 possible next byte values. An instance of this vulnerability was publicly disclosed as CVE-2016-8462.[19]

To prevent data exfiltration using flash hashing functionality, bootloaders should ensure the start address and length of regions being hashed are fixed (generally to partition boundaries), and large enough to prevent mapping of hash results to flash contents. Also consider restricting the hashing functionality to partitions that need to be hashed during the manufacturing or firmware update process.

Since bootloader logic errors can be exploited through purely software attacks, they can be exploited repeatedly and often rapidly for very little cost to the attacker. However, as with all attacks, initial vulnerability discovery and exploit development may take days or months.

Side-Channel Leaks

Some embedded bootloaders protect sensitive functionality with a password or other authentication mechanism. However, many such protected bootloaders do not protect against timing and power side-channel leakage. Many embedded bootloaders do not perform time-invariant memory comparisons for password validation. This leaks information about the number of correct characters in the password, and makes brute-force attacks much faster. Furthermore, bootloaders that perform a flash erase or write operation after determining that the entered password is incorrect will show a large power side-channel signature filling flash memory charge pumps prior to the erase/write operation. The power signature warns an observant attacker that an erase or write is about to happen, and allows the attacker to cut power before data is actually changed.

Timing and power side-channel leakage has been used to attack the Bootstrap Loader (BSL) on certain models of Texas Instruments MSP430 microcontrollers. Research by Travis Goodspeed[20] found that versions 1.30 and 2.12 of the MSP430 BSL do not perform timing-safe password comparisons. The MSP430 BSL is designed to trigger a mass erase when an incorrect password is entered. Travis was able to perform a brute-force attack against BSL passwords by detecting the current surge filling charge pumps before an erase operation, and cutting power before the actual erasure. He accelerated his password brute-forcing using timing information leaked from the non constant time password comparison on vulnerable versions of the bootloader. Since the BSL for the affected MSP430 models was in ROM, this issue could not be patched in the code, and disabling the BSL altogether was the only viable option.

To avoid this class of vulnerability, time and power invariant comparison functions should be used (other side-channels may require further mitigations). Furthermore, a secure persistent attempt counter should be incremented pre-emptively prior to authentication operations to avoid the power-cutting technique described above. For more details on implementing secure bootloaders, see Implementing Secure Bootloaders on page 19.

The exploitation of side-channel information leakage from a bootloader is generally slower, less reliable, and requires more equipment than exploiting logical flaws. Nevertheless, once a side-channel attack against a bootloader has been developed, it can usually be conducted in a few hours using equipment under $1000 USD, such as an oscilloscope or the ChipWhisperer.[21]

[19] Disclosure for CVE-2016-8462
[20] Practical Attacks against the MSP430 BSL
[21] ChipWhisperer

Digital electronic circuits are designed to function correctly only at defined operating points, with specific voltage, clock frequencies, and temperatures, which are often visualized as a so-called *shmoo plot*.[22] Pushing the operating conditions outside this range can result in various kinds of failures. Near the boundary between good and bad operating characteristics, a region of partial failures can sometimes be discovered and exploited. Careful control by the attacker over the operating characteristics can yield useful results.

Figure 5: Example Shmoo plot of two variables

Supply voltages and clock signals for a microcontroller can be manipulated to induce faults such as incorrect evaluation of CPU instructions, skipped CPU instructions, or corrupt reads from flash. Such attacks are known as voltage and clock fault injection or glitching. Voltage and clock glitching attacks are considered non-invasive since they do not require decapsulation of microcontroller packages. Voltage glitching involves briefly lowering the supply voltage for a processor (to just above the brown out voltage) during the execution of select instructions. Clock glitching involves inducing faults by momentarily altering processor clock timing during the execution of select instructions, violating the setup and hold time requirements of internal logic.

Other forms of fault injection attacks can be performed via electromagnetic radiation and lasers. Electromagnetic (EM) fault injection can be performed using equipment such as the ChipShouter[23] which generates a short-duration high-intensity EM pulse that can induce currents within the internal circuitry of the chip. Laser optical fault injection is normally semi-invasive in that it generally requires decapsulation of the chip package, though no silicon tampering or probing is needed. Instead, short-duration laser (normally infrared or UV) pulses are used to excite specific areas of the chip (typically though the transparent backside) and induce targeted faults.

[22] Shmoo Plot
[23] ChipSHOUTER

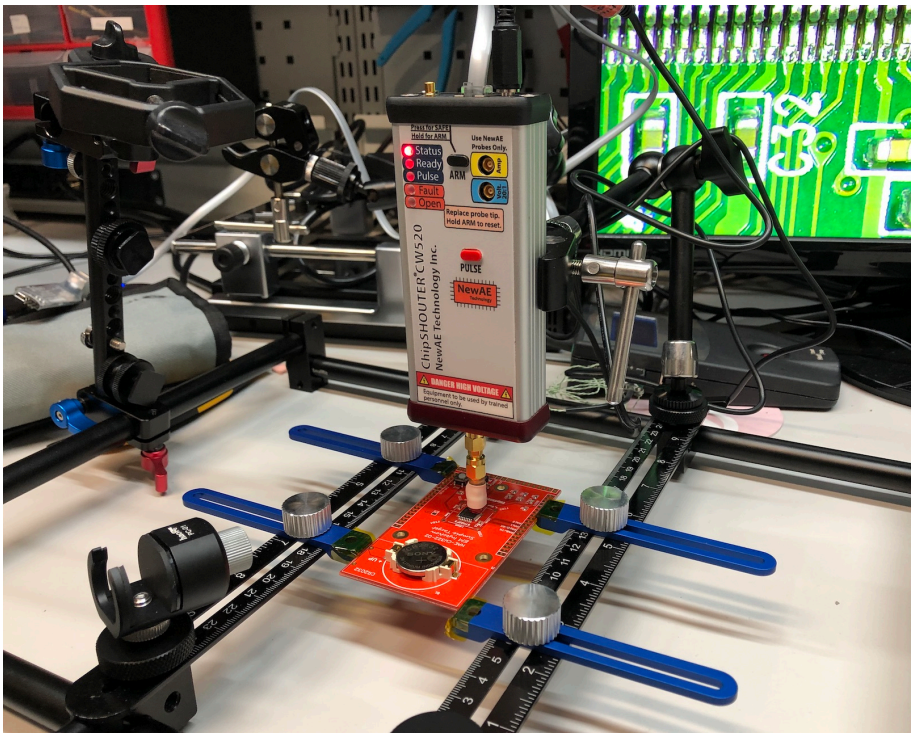


Figure 6: ChipShouter in use in an NCC Group hardware security lab

The execution of CPU instructions requires multiple stages, including fetch (loading instructions from memory), execution, instruction decoding (interpreting and preparing fetched instructions for execution the ALU), and writing back execution results to memory. This is a simplified view, and modern CPUs have pipelines that can include more than 20 stages, though typically microcontroller pipelines have 2-10 stages. Glitching of the fetch stage typically results in the fetch buffer not being updated, resulting in skipped instructions and double execution of the previous instruction.[24] Glitching of the execution stages typically result in incorrect results for the instruction; depending on the nature of the CPU and the glitch parameters, this may result in various constant value results, or sometimes random unpredictable results.

To improve CPU performance, most processors use pipelining to perform the stages in parallel for sequential instructions. Glitching a pipelined processor will often impact multiple stages simultaneously, such as fetch and execution. Nevertheless, it is often possible to cleanly skip single instructions even on a pipelined CPU. For example, on a simple two stage pipeline (fetch and execute), glitching the fetch of a target instruction may also corrupt the execution of the preceding instruction; however, the previously corrupt execution could be repeated correctly on the next clock cycle since the fetch buffer was not updated. The diagram[25] below illustrates an example of a five stage pipeline, consisting of instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write-back of results to a register (WB). Note that all these stages occur in parallel, and fault injection attempts on such a CPU could affect all of the stages concurrently.

[24] On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms
[25] Five stage pipeline diagram by "Sandstorm de", licensed under CC-BY-SA-4.0

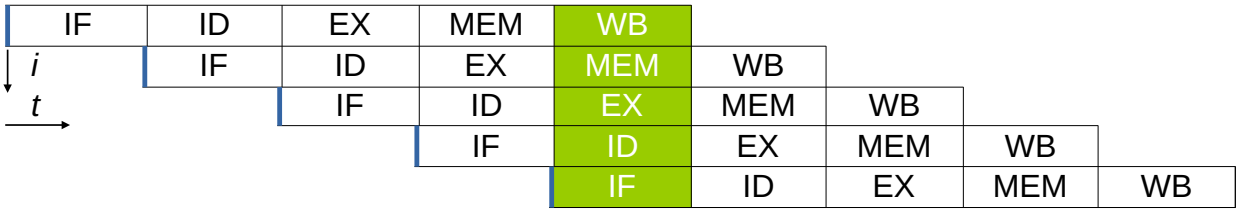


Figure 7: Illustration of a five stage CPU pipeline

Fault injection attacks are often used to bypass logical checks in software by causing incorrect memory loads or incorrect comparison operations. Bootloaders often read security flags stored in flash or fuses, and react differently depending on the value read. For example, the boot ROM on the NXP LPC1343 restricts memory read operations when it detects that *code read protection* (CRP) is enabled. Voltage glitching has been used by Chris Gerlinsky[26] to defeat CRP checks in the LPC1343 ROM bootloader, allowing a debugger to read out the microcontroller's firmware. This glitching attack caused momentary data corruption during a read operation which retrieved the CRP flag from the microcontroller's internal flash memory, and due to the fail-open behavior of the bootloader, it was coerced into believing that code read protection was disabled.

Another example of using voltage glitching to defeat read protection was demonstrated by Nedospasov, Datko, and Roth against a STM32F205 microcontroller from a Trezor cryptocurrency wallet.[27] By dropping supply voltage for 6 microseconds during option byte reading in the boot ROM, they were able to downgrade the microcontroller readout protection from RDP Level 2 (debug disabled) to RDP Level 1 (RAM and register access allowed). Downgrading to RDP Level 1 allowed extracting secrets from RAM, including the BIP 39 Bitcoin private key seed phrase.

Voltage glitching has also been used to defeat secure boot, extract secret keys, and re-enable JTAG on security-focused microcontrollers with ARMv8-M TrustZone support. General purpose microcontrollers with TrustZone support rarely include the fault injection countermeasures commonly seen in smart card microcontrollers. At 36C3, Thomas Roth demonstrated[28] an attack performing voltage glitching on the Microchip SAM L11 boot ROM code that configures the Implementation Defined Attribution Unit (IDAU) responsible for determining which memory ranges are associated with the secure world. By glitching during the IDAU setup, he was able to make flash and RAM ranges normally restricted to the secure world (TrustZone) accessible to the non-secure world including non-secure JTAG debug. He also demonstrated similar attacks glitching the Security Arbitration Unit (SAU) initialization on the Nuvoton NuMirco M2351, which is marketed as having resistance to fault injection.

To protect against fault injection attacks, high-security processors often include internal clock sources to prevent external clock manipulations, and supply voltage monitoring that resets the processor when voltage glitches are detected. Varied and redundant software checks can also be used to make the timing of glitching more difficult. It should be noted that more generic supply voltage monitoring systems are designed primarily for detecting gradual brownouts due to discharged batteries or being unplugged, rather than fast targeted glitch pulses of a few nanoseconds. Such brownout detection circuitry often fails to protect against targeted glitching attacks.

Characterizing a processor and identifying parameters for reliable fault injection attacks can take days or months depending on the number of parameters, and the range and granularity of variation. These parameters can include delay timing, glitch duration, voltage, and temperature. For optical and electromagnetic fault injection, parameters also include the location on the chip to apply the glitch and the intensity of the burst. The search space for delay timing

[26] Breaking Code Read Protection on the NXP LPC-family Microcontrollers
[27] WALLET.FAIL
[28] TrustZone-M(eh): Breaking ARMv8-M's security

can be particularly large, as modern microcontrollers often run over 100 MHz, and even a few millisecond window can include over a million instructions that can be targeted. The search space for glitch delay timing can sometimes be narrowed through power analysis, by identifying the power signature of operations the attacker wishes to affect.

Identifying a suitable trigger condition in order to exploit the desired system operation can also be a challenge. When glitching constant time code executed automatically following reset (such as boot ROMs and firmware initialization), glitch delay timing can be triggered following CPU reset. However, to glitch variable time operations or operations that are performed in response to a runtime input or event, a more complex trigger needs to be identified for precise delay timing. Such triggers can be based on power analysis (detecting spikes or patterns in current draw), monitoring input signals, or other measurements.

Once suitable triggers and glitch parameters have been identified, most fault injection attacks can be conducted and repeated in a matter of minutes using cost effective equipment, such as the ChipWhisperer[29]($300 USD), ChipShouter ($3000 USD), or a custom FPGA-based design. More rudimentary voltage glitching equipment can built with a microcontroller and MOSFET for under $5, as demonstrated by Thomas Roth at 36C3.[30]

[29] ChipWhisperer
[30] TrustZone-M(eh): Breaking ARMv8-M's security

Invasive attacks involve decapsulating chip packages using corrosive chemicals or laser ablation and then accessing the silicon directly using micro-probes, sometimes requiring edits with a Focused Ion Beam (FIB). The image below[31] shows a Texas Instruments MSP430 microcontroller decapsulated to expose the silicon die, in a manner similar to what would be performed in an invasive silicon attack.

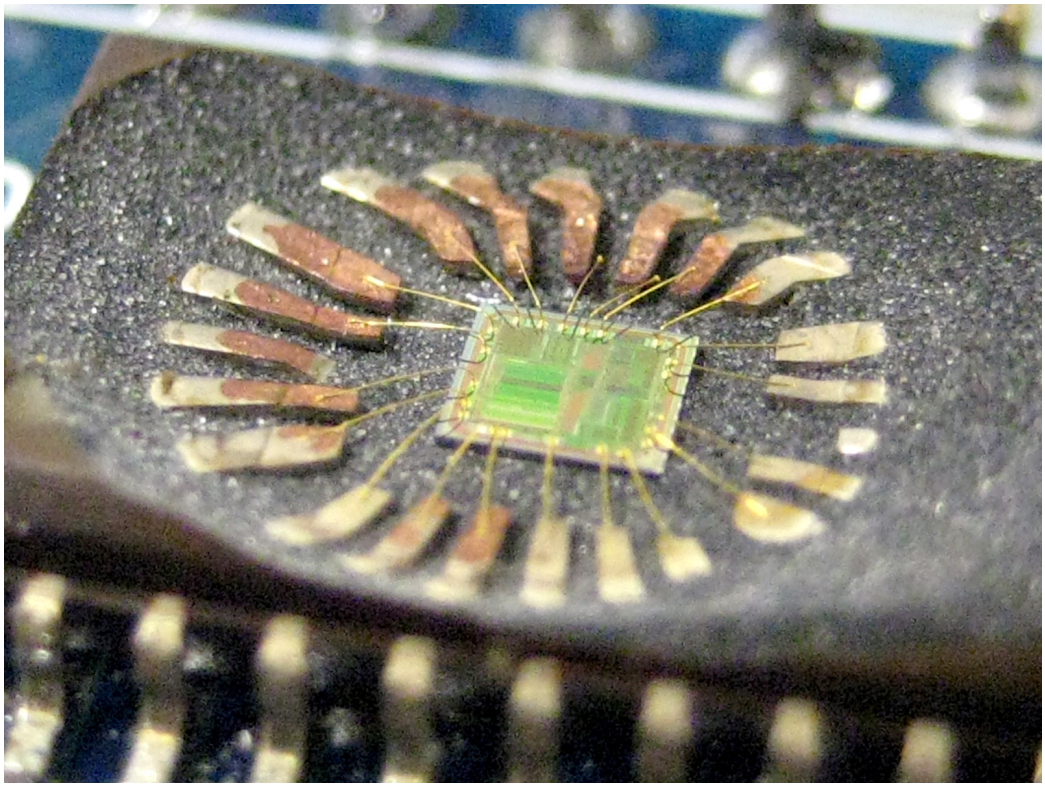


Figure 8: Decapsulated TI MSP430F1101A Microcontroller

Invasive attacks are nearly impossible to prevent definitively, though purpose-built security processors tend to be more resistant to invasive attacks than commodity microcontrollers. Common defensive technologies used by hardened processors include:

- Scrambling of internal bus signal ordering to confuse micro-probing attempts
- Encryption of internal flash and RAM
- Tamper reactive silicon mesh layers covering important components
- Internal clock generation to prevent clock glitching
- Voltage monitoring with glitch detection

With sufficient time, effort, equipment, and funding, invasive attacks can be used to extract secrets from any hardware. Depending on complexity, invasive silicon attacks have development costs ranging from a thousand to a million dollars. Invasive attacks require costly and sophisticated equipment, including a Scanning Electron Microscope (SEM), Focused Ion Beam (FIB), a clean room, highly reactive chemicals, and safety equipment for appropriate handling and fume containment. The skills to effectively make use of such equipment are also highly specialized. It should be noted that once an invasive attack has been developed and practiced, it can generally be executed repeatedly with just one or two days of lab time, at a cost much lower than the initial attack development. The development of an attack represents a non-recurring engineering (NRE) cost. In general, system designers seek to maximize the attacker's NRE

[31] Photo by Travis Goodspeed, licensed under CC BY 2.0

while minimizing added cost in designs.

Once silicon is exposed, there are several types of attacks possible. Fuses and ROM can be read using an optical microscope. Micro-probing allows reading of data and other sensitive signals that are transmitted on the internal buses of a microcontroller. Focused Ion Beams (FIBs) can be used to modify internal circuitry such as bypassing security defenses or reconnecting blown fuses. Flash memory cells storing critical values (such as lock bits) can be cleared in a targeted manner using ultraviolet (UV) light. This has been demonstrated by Andrew "bunnie" Huang to defeat PIC18F1320 readback protection,[32] and by Obermeier and Tatschner to downgrade STM32F0 readout protection from Level 2 to Level 1.[33] UV and IR (infrared) light can also be used in precisely timed bursts to interfere with the execution of instructions; this technique is known as optical fault injection, which is discussed further in Fault Injection on page 12. Defending against an invasive attacker with such capabilities is very difficult.

There are several companies that offer code and secret extraction services. Such services are frequently offered by companies in countries with weaker intellectual property protections, such as Russia and China. Some offer invasive data extraction from many common microcontrollers for under $2000.[34] Using uncommon microcontrollers may provide some security through obscurity since an attack would likely need to be developed before it can be executed, however this should not be relied upon in the long term. The readback protection on most general purpose microcontrollers should be considered to be breakable for a cost of at most, a few tens of thousands of dollars.

[32] Selective Erasure of PIC MCU Flash by Andrew "bunnie" Huang
[33] Shedding too much Light on a Microcontroller's Firmware Protection
[34] Russian Semi Research

Protecting External Flash Contents

Some microcontrollers do not contain internal flash memory for storing firmware, and instead execute code from external flash. Since external flash can be read trivially by a physically proximate attacker, the only ways to protect secrets stored in external flash are through the use of encryption or obfuscation. Software obfuscation of flash contents is inherently defeatable through reverse engineering of the software, and it is not recommended in this paper. Encryption of flash contents, the main alternative, requires hardware support.

Although many microcontrollers support external flash encryption, this section will reference the popular Espressif ESP32 series of chips as an example to highlight these common design weaknesses.

One requirement of preventing trivial flash decryption is that the microcontroller needs to prevent the execution of unauthorized code. Ideally, this should be achieved by encrypting external code flash using an authenticated encryption mode that can detect when blocks have been modified, reordered, or taken in pieces from other binaries encrypted with the same key. Unfortunately, it is non-trivial to properly implement authenticated encryption with random access to blocks in a hardware accelerator. The ESP32, like many others, does not support true authenticated encryption of external flash.

The ESP32 series microcontrollers support encrypting flash in either Electronic Code Book (ECB) mode, or in a mode where the AES encryption key is "tweaked" by the block number using an XOR operation.[35] The flash encryption key is stored in internal eFuses. For both ECB and tweaked key modes, it is possible through brute force to randomly generate an encrypted block that decrypts to valid attacker-desired instructions. ECB mode (with `FLASH_CRYPT_CONFIG` = `0`) should be avoided, because it provides no protection against swapping one encrypted block with another block taken from the same image or another image. With ECB mode, it may be possible to defeat security checks in firmware by replacing encrypted blocks of flash memory containing security critical instructions with valid alternate instructions from another location. ECB mode is also generally considered insecure, as it reveals the presence of repeated blocks in the plain text.

Enabling the ESP32 key tweak feature (where each block is encrypted by a unique key) prevents reordering blocks of firmware. However, this does not prevent replacing an encrypted block with a known block at the same offset from another valid firmware image. Furthermore, this still does not prevent brute force attacks that generate random encrypted blocks that decrypt to valid attacker-desired instructions.

At the time of writing, there have been no publicly documented attacks against the ESP32 making use of these cryptographic weaknesses. However, the malleability of non-authenticated block cipher modes is well understood by the security community. Developers should be aware of the cryptographic limitations of ESP32 firmware encryption before relying on it to protect secrets.

It should also be noted that cryptographic integrity verification (secure boot) in a bootloader does not fully protect against against flash content modification. An attacker attempting to decrypt external flash by running custom code could perform a "Time Of Check to Time Of Use" (TOCTOU) attack[36] by attaching a multiplexer to the flash interface. The multiplexer would present the legitimate signed flash contents to the bootloader for the purposes of boot image validation, then switch to a modified flash at runtime.

[35] Setting Setting `FLASH_CRYPT_CONFIG`
[36] Secure Boot & Flash Encryption

Implementing Secure Bootloaders

A bootloader is the very first piece of code to run on a system, and is typically responsible for some minimal hardware configuration, loading and validation of subsequent firmware modules, and normally contains some basic USB or serial communications for updating firmware. The first stage bootloader is often located in ROM within the microcontroller or SoC and forms a root of trust. Second (and later) stage bootloaders are located in flash memory. Many attacks that defeat microcontroller read protections make use of insecure bootloader functionality. This includes unauthenticated or weakly authenticated memory read/write functionality, and inadequate verification of code loaded onto the device. When data stored on microcontrollers needs to be protected against readout or modification, it is essential to have a secure bootloader.

The goal of this section is not only to describe common bootloader flaws, but to also provide high level guidance for product implementers on how to securely configure or implement a bootloader to protect code integrity and secret storage.

ROM Based Bootloaders

It is vitally important to have an immutable root-of-trust that cannot be replaced by an external attacker. ROM based bootloaders fit this requirement nicely, but paradoxically, ROM based bootloaders are a common source of security issues in microcontrollers. Since they are fixed in silicon and not easily customizable, ROM based bootloaders usually try to satisfy all different use cases for a component, and thus include functionality unnecessary for many use cases, increasing the attack surface. When vulnerabilities are found in ROM bootloaders, they require costly hardware changes to fix, and so rarely see updates.

Bootloaders for low-cost microcontrollers are frequently under-scrutinized. They are usually proprietary with source code unavailable for review, and are frequently not even provided in binary form (aside from being part of the silicon ROM). Furthermore, many silicon vendors make information about security-relevant bootloader functionality only available to specific customers under Non-Disclosure Agreements (NDAs).

The quality and functionality of ROM based bootloaders should be evaluated when selecting microcontrollers for a design. A secure embedded bootloader should fully support secure boot, require authentication for privileged functionality such as debug and serial communication, and allow disabling unnecessary functionality to minimize the attack surface. Ideally, all bootloaders, including proprietary ROM based ones, should also have undergone independent security review.

Secure Boot

The most important security feature for an embedded bootloader is Secure Boot - the cryptographic integrity and authenticity verification of code at boot time. Unfortunately, many low cost microcontrollers do not implement secure boot in their ROM based bootloaders, or implement it incompletely. A robust secure boot implementation should verify asymmetric cryptographic signatures at boot time for all code and configuration to be used. There are a number of common ways embedded bootloaders fail to properly implement secure boot, including:

- **Hash Instead of Signature:** Some embedded bootloaders accept images that only include a hash digest of the protected data rather than a cryptographic signature, without the device knowing a known-good hash digest against which to compare. While verification of a hash provides protection against accidental data corruption, an intentional attacker could change the provided hash at the same time they modify the data being hashed. An example of this flaw can be seen on some Amlogic S905 devices.[37]
- **Weak Hashing Algorithm:** Some embedded bootloaders verify data using outdated cryptographically weak hash functions (such as MD5 or SHA1), or non-cryptographic checksum functions (such as CRC32). When a weak hash function is used, an attacker may be able to construct a malicious software image with a hash matching the legitimate original.
- **Symmetric Signatures:** Some embedded bootloaders expect images to be signed using symmetric mechanisms

[37] Amlogic S905 SoC: bypassing the (not so) Secure Boot to dump the BootROM

such as CMAC[38] or GMAC.[39] The problem with symmetric signatures is that the verifying the signature needs the same key that was used to produced the signature. An attacker reverse engineering the bootloader and hardware may be able to recover the signing key from the device, and subsequently produce valid signatures for unauthorized binaries.

- **Encryption Instead of Authentication:** Some embedded bootloaders treat firmware encryption as a substitute to firmware authentication, which it is not. Many encryption schemes are malleable, where modification to the cipher text allows related modification of the plain text. Furthermore, the symmetric key used to encrypt the firmware could be recovered through reverse engineering of the hardware and bootloader. Firmware encryption can be a useful layer to protect intellectual property for a short time, but it cannot be considered a substitute for secure boot. It should be noted that microcontrollers that execute in-place from external flash are often dependent on firmware encryption as their only code integrity protection, and this can be problematic, as discussed in Protecting External Flash Contents on page 18.
- **Install-Time Verification Instead of Boot-Time:** Some embedded bootloaders verify new code at install-time but not at boot-time. For devices that run code contained in external flash memory, this is a major issue because an attacker with physical access could manually modify the code in flash. The risk of install-time verification without boot time verification is lower on devices that only keep code in the internal flash memory, since control of the microcontroller is generally required to modify internal flash contents. However, it may be possible for malware that exploits microcontroller firmware to gain persistence through the modification of internal flash contents.[40] While install-time software integrity checks are not a substitute for boot-time checks from a security standpoint, error reporting from install-time checks can improve user experience and reliability, and so generally both are recommended.
- **Bootloader Not Write Protected:** The root of trust for Secure Boot must be immutable. If the first stage bootloader or the public keys used by the first stage bootloader for image validation are stored in a flash region that is not write protected, then an attacker may be able to overwrite the bootloader or the public keys, defeating Secure Boot. ROM based bootloaders are inherently more resistant to modification, but key storage must also be immutable. OTP Fuses are a common immutable storage mechanism used for public keys.

Attack Surface Minimization

The simpler a bootloader is, the fewer ways there are to attack it. Many embedded bootloaders, particularly ROM based ones, incorporate a large amount of debug and test functionality to satisfy all possible OEM use cases. Examples of such debug and test functionality include memory read/write primitives, functionality to view and edit system configuration, functionality to compute hash digests of flash segments, and hardware functional test code. Some test and debug functionality is inherently dangerous, such as raw memory read/write primitives. Other functionality, such as hardware functional test code, may appear benign but could contain flaws in input validation, or interesting side-channel data leakages.

Secure bootloaders should allow disabling all functionality that is not needed for the product being developed. Ideally, unnecessary functionality should be removed at compile time, with unnecessary code not included in the final binary. For ROM based bootloaders that cannot be customized, functionality disablement at runtime should be based on one-time-programmable (OTP) fuse status.

Authenticated Access

For most products, it is reasonable to expect the user to authenticate before accessing any sensitive or personally identifiable information (PII). Such user authentication is typically implemented on user accessible interfaces as a matter of normal product functional requirements. Authentication is equally important on interfaces that provide privileged functionality that is not intended for users, but instead may be intended only for OEM use (such as debug, firmware loading, factory, and repair operations). This helps defend against attacks on user devices where the OEM is considered a threat (so called *insider attacks*). Because this level of authentication is most often product specific, it is

[38] One-key MAC
[39] Counter Mode
[40] Code injection attacks on harvard-architecture devices

unlikely to be supported by vendor-supplied internal ROM bootloaders, which is another strong reason for disabling such interfaces in ROM based bootloaders.

For some products, the threat model may further include the users themselves. This is the case where the device has additional stakeholders beyond the end user, such as when the user is not the owner (device lost/stolen, rental situations, enterprise devices, shared devices, etc), when media content providers or subsidizing entities are involved (SIM lock and other DRM use cases), or when supply chain attacks or *buyer's remorse*[41] are considered. In such cases it may further be necessary for additional authentication steps to be implemented on these privileged interfaces.

To minimize the risk of exposing sensitive keys or passwords, there should be no symmetric authentication keys or passwords shared among individual devices; and in fact, a growing number of jurisdictions are enacting or considering legislation against this bad practice.[42] Instead, device-unique keys or passwords must be used for authentication. The authentication protocol should implement brute force and replay countermeasures. To limit brute force attempts, there should be a securely stored persistent failed attempt counter that is incremented before an authentication attempt, and zeroed after authentication succeeds. A preemptive increment is required to avoid a timing attack as described in Bootloader Interfaces on page 10. Authentication logic should ensure that the current failed attempt counter is below a threshold before attempting authentication. Internal, difficult to access or modify persistent storage would be an ideal location for storing an attempt counter.

An effective mechanism to authenticate access would be through a challenge and response scheme such as the one described below:

1. The embedded device would generate a random numeric challenge nonce and present it to the user attempting to authenticate. Ideally, the nonce should be at least 128 bits, though a nonce of at least 32 bits can be sufficient when nonce generation attempts are limited. The challenge should be generated by a cryptographically secure pseudo-random number generator (CSPRNG) seeded by a hardware true random number generator (TRNG). The purpose of the challenge is to prevent replay attacks.
2. The user would relay the challenge along with the device serial number into an online service that they have logged into, and the online back-end would compute an HMAC of the challenge using a device-unique symmetric key. This HMAC would act as a one time access token. The back-end would have either a master key used to generate device-unique symmetric keys, or access to a database of unique symmetric keys each mapped to a specific device serial number. All access token generation attempts by a user would be logged by the online back-end. If manual entry of the token is required, and authentication attempts are limited by the embedded device, the token (HMAC) can safely be truncated to 32 bits for ease of entry (but no smaller to avoid easy guessing).
3. The user would provide the one-time access token to the embedded device. Prior to checking the access token, the embedded device will verify that the authentication attempt counter is below the attempt limit, and block access if it is not. Next, if the counter is below the limit, the device will pre-increment the counter before validating the token.
4. The embedded device would compute the expected token (HMAC), and compare it to the received value from the user in a time-invariant manner. If the supplied token matches the internally computed one, the authentication attempt counter would be zeroed, and the embedded device would allow privileged access.

[41] Reddit report of maintaining access to a returned security camera
[42] California, Oregon lead the way on smart home security

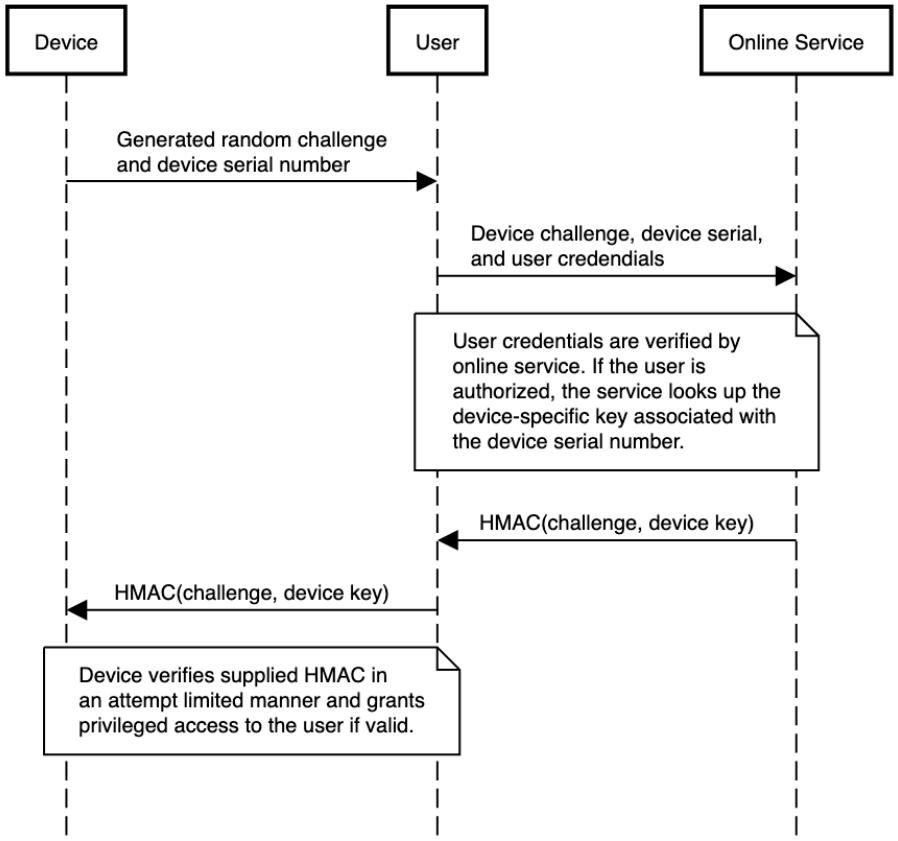


Figure 9: Sequence Diagram Illustrating the Authentication Process

The code below provides an example of a simple time-invariant memory comparison function that can be used to compare authentication tokens or passwords. It should be noted that the C standard cannot guarantee constant timing, but in practice, on current compilers, this should compile to be constant time for a given data length regardless of the buffer contents being compared.

```
bool time_invariant_memeq(void *a, void *b, size_t size)
{
    uint8_t diff = 0;
    uint8_t *ac = (uint8_t *)a;
    uint8_t *bc = (uint8_t *)b;
    for (size_t i = 0; i < size; i++)
        diff |= bc[i] - ac[i];
    return (diff == 0) ? true : false;
}
```

A challenge-response scheme could also be implemented using asymmetric cryptography, with a signing (private) key stored on the back-end, and a verifying key (public) key stored on the embedded device. The benefit of asymmetric signatures is that the public key on the device does not need to be kept secret, and could be safely shared across multiple devices. To prevent abuse, asymmetric access tokens should still be bound to a specific device serial number. Unlike symmetric signatures, asymmetric signatures tend to be large and are non-truncatable. On devices that support

automated token entry, such as over a network connection or loading a certificate file from a USB interface, this is a non-issue. However, on devices with limited IO capabilities, asymmetric signatures may be cumbersome for manual entry.

Implementing Custom Bootloaders

If there are no suitable microcontrollers with a secure ROM bootloader available, or if a microcontroller without a secure ROM bootloader has already been selected, it may be sufficient to implement a suitable Secure Boot solution by placing a custom bootloader in a write protected region of flash. A flash-based bootloader can be used as a substitute for a ROM-based bootloader, provided that it can be suitably protected from abuse; a combination of fully disabling debug interfaces and write protecting the bootloader flash segment is generally sufficient. When implementing a custom bootloader, the microcontroller vendor's ROM bootloader should be configured to ONLY load the flash-based bootloader, and any external interfaces for the ROM bootloader (such as serial or USB) should be disabled. This helps prevent the custom bootloader functionality from being bypassed by the ROM.

A custom bootloader should adhere to the principles described above: it should implement secure boot, keep attack surface to a minimum, and require secure authentication for any debug functionality that is provided. It should be noted that the security of a system is only as strong as its weakest link. A secure bootloader would not secure a microcontroller that provides no mechanism to disable debug functionality.

FRAM vs. Flash

While the majority of microcontrollers use internal NOR flash memory, there are also microcontrollers available containing internal Ferroelectric Random Access Memory (FRAM).[43] Internal FRAM provides several security benefits over internal flash memory that can aid the development of secure bootloaders and impede unauthorized readout. Unlike flash memory, FRAM permits low power, fast, and localized writes, and has very high write endurance. FRAM also has a write power signature that is identical to reads, because every FRAM access is both a read and a write. When data is not being modified, the value written is the same value that was read.

These properties are helpful when implementing bootloader authentication mechanisms, since they allow efficient non-volatile attempt counters without a prominent power signature. FRAM also enables fast mass erase operations that avoid the externally detectable current spike associated with flash memory charge pumps. The lack of write current spikes makes it more difficult to implement a power analysis based trigger for reset or glitching on flash writes.

In addition to fast low-power write support, FRAM provides improved resistance to invasive attacks compared to conventional flash memory. Sergei Skorobogatov has noted that FRAM *"offers very good security protection against invasive and non-invasive attacks because its state cannot be observed optically or detected with probes."*[44] An invasive attacker can observe data read from FRAM on internal buses, but FRAM contents cannot be easily read in-place with probes. Also, unlike flash and EEPROM, the state of FRAM cannot be altered by shining light on it. This prevents a common invasive attack against flash based microcontrollers where security bits in internal flash are cleared by targeted UV light, which may re-enable hardware debug ports.

[43] Ferroelectric RAM
[44] Semi-invasive attacks –A new approach to hardware security analysis

Conclusions

Ultimately, all microcontroller data protections can be defeated for a price. No microcontrollers can be considered fully safe from invasive attacks, but high-security microcontrollers can be selected to make invasive attacks more difficult. With appropriate equipment and skills, most ordinary microcontroller data protections can be defeated through invasive attacks with a budget of $25,000 USD or less.

However, many microcontroller protections can be defeated more easily without resorting to invasive attacks. Debug restrictions that do not fully disable debug interfaces often leave gaps that can be used to work around restrictions. ROM based password protected bootloaders often have side-channel information leakage, and brute force protections are often poorly designed or implemented. Even when bootloaders are logically correct, fault injection attacks may be used to defeat logical checks. To minimize the risk of data exposure through simple, low-cost attacks, NCC Group recommends the following best practices:

- Define security requirements early in the component selection process, and evaluate the quality of readback protection implementations as part of component selection
- Avoid storing high value non-revocable shared secrets on microcontrollers whenever possible, and use purpose built hardened secure microcontrollers when high value secret storage is necessary
- Disable processor debug functionality completely whenever possible
- Disable or require authentication for any bootloader development and test functionality that could risk exposing secrets
- Enforce secure boot and/or write protect flash to prevent the execution of unauthorized code
- Pre-increment (rather than post-increment) authentication attempt counters in bootloaders before checking supplied authentication tokens
- Address whitelists are preferable to blacklists when restricting regions that can be written to or read from in a bootloader
- Use timing-invariant comparison functions when verifying bootloader passwords or authentication tokens

To avoid repeating common mistakes that weaken readback protection:

- Do not consider re-purposing debug pins at run-time to be a security measure
- Do not allow the loading and execution of untrusted code without a full wipe of all secrets stored on the device
- Do not allow unauthenticated access to RAM, flash, or peripheral registers via external bootloader interfaces
- Do not treat firmware encryption as a substitute for firmware authentication

An important theme in these recommendations is that the security of readback protection should be considered throughout the development process, rather than being an afterthought. Impacts on readback protection security should be considered during component selection and during the development of all features to avoid the risk of features inherently breaking readback protection. While it is impossible to prevent microcontroller flash readout by a sufficiently capable and well funded adversary, it is possible to select components and design firmware that impede simple and low-cost attacks. Developers should aim to make the cost of extracting secrets from a microcontroller comparable to or greater than the value of the secrets themselves.

Acknowledgements

I would like to thank my colleagues Rob Wood, Jeremy Boone, Jon Szymaniak, and Jennifer Fernick for their extensive review and feedback that greatly contributed to the clarity and depth of this paper.