Secure Firmware Distribution for Automotive Control

We have designed and implemented a bootloader and a set of supporting tools for secure firmware distribution for automotive control. Our aim is to send firmware updates to our device so that we can add more features and fix any security problems after shipping.

Threat Model

1. Attacker tries to read firmware secrets from the update binary or directly from device.
2. Attacker tries to modify firmware update to insert malicious code.
3. Attacker tries to install old versions of firmware that have known vulnerabilities.
4. Attacker tries to read firmware secrets from a leaked recording of a technician's readback.

Security Goals

Confidentiality

Firmware should be protected during update and while it is running on the device to ensure no one can extract sensitive information or proprietary algorithms. Additionally, a recording of a technician's readback communication should not expose firmware secrets.

Integrity

Bootloader should be able to verify that firmware is from a legitimate source and was not altered during transit.

Authentication

Only protected firmware images that are created in secure environment should be accepted by the bootloader. Only authorized technicians (in a factory setting) should be able to read the firmware from the device.

Achieving security goals

The firmware is signed and encrypted before it is sent to the bootloader and bootloader writes the firmware only if the signature verifies. It is also encrypted by the bootloader when the readback tool tries to read the firmware. Integrity and authentication are achieved with the help of signatures.

Writing firmware

Due to memory restrictions, firmware is sent in page-sized chunks. Each chunk contains version number and is signed and encrypted. After all the firmware pages, a measurement is sent to the bootloader. This measurement is a hash chain of all the pages. We are using HMAC-SHA1 to achieve keyed authentication and AES-CBC-256 to encrypt the firmware.

Booting firmware

Before giving control to the firmware, bootloader reads the firmware from memory and tries to verify the measurement. It gives control to the firmware only if the measurement is correct. All local variables, including key material, are wiped from the verification function’s stack before the application runs.

Readback

Readback is achieved with the help of a challenge-response protocol which starts with the readback host tool sending a request to initiate the protocol. On receiving a request, the bootloader sends an encrypted nonce as a challenge to the readback tool. If the technician has access to the right key, he/she needs to write the key to the secret_configure_output.txt. Readback tool then decrypts the nonce and creates a request which contains the unencrypted nonce. The request also contains the start address and number of bytes to be read along with a random IVf. Presence of correct nonce in the request authenticates the technician to the bootloader. Request is also signed and encrypted by the readback tool and verified by bootloader before sending the firmware pages.
Hash of IVf acts as initialization vector to encrypt the first page sent by bootloader. The next page is encrypted using hash of IV of previous page. We designed the IV to be sent by the host tool due to lack of entropy on the device. Encrypted pages do not have IVs sent along with them, so plaintext IVs are not present in the recording. Both readback tool and bootloader derive the next IV by hashing the previous IV.

Source of Entropy

We do not have a good source of entropy on the device. Hence, we generate a pseudo random number with the help of time and random seed. The random numbers generated using this pseudo random number generator act as nonces in readback protocol. We also use these random numbers to produce random delays between critical branch instructions to make glitching attacks difficult.

Secret vault for keys and metadata

Keys are stored in the secret vault section which is in bootloader address space. Keys are erased from the stack every time after they are used. The secret vault also contains other metadata such as current version of firmware, its size and if debug firmware (version 0) has been written to the device. The measurement or hash chain of current firmware is also stored in the secret vault.

Side channel prevention

We have added instructions to AES CBC implementation to perform operations on an inverse copy of each data block to defend against side channel attacks for key extraction.

Protection against glitching

At each critical comparison branch, we perform three checks to increase the difficulty of bypassing a check by glitching. Additionally, we add random delays to make timing estimations difficult.

Memory Map

section	size	addr
.data	0x86	0x800100
.bootloader_libs	0x1742	0x0
.bootloader_funcs	0xfc0	0x1800
.text	0x1b12	0x1e000
.secret_vault	0x140	0x1fd00
.bss	0x10	0x800186
.comment	0x11	0x0
.note.gnu.avr.deviceinfo	0x40	0x0

Custom sections:

1. Bootloader_libs: This section holds the cryptographic libraries
2. Bootloader_funcs: This section holds a few functions of bootloader such as boot_firmware and readback.
3. Secret_vault: This section contains metadata of firmware such as version number and size along with symmetric key used for encryption and a random seed.

Host Tools

Build Tool

This tool generates secret key and a seed and builds the bootloader. The key and seed are baked into the source code of bootloader before building it. The secrets are saved to the file secret_build_output.txt. The key is later used to encrypt and sign firmware and the seed is later used to generate random numbers. Avrdude can be used to write bootloader onto the device. Note that lock.hex must be provided to avrdude.

Configure Tool

We use the configure tool to copy the contents of secret_build_output.txt to secret_configure_output.txt.

Bundle and Protect Tool

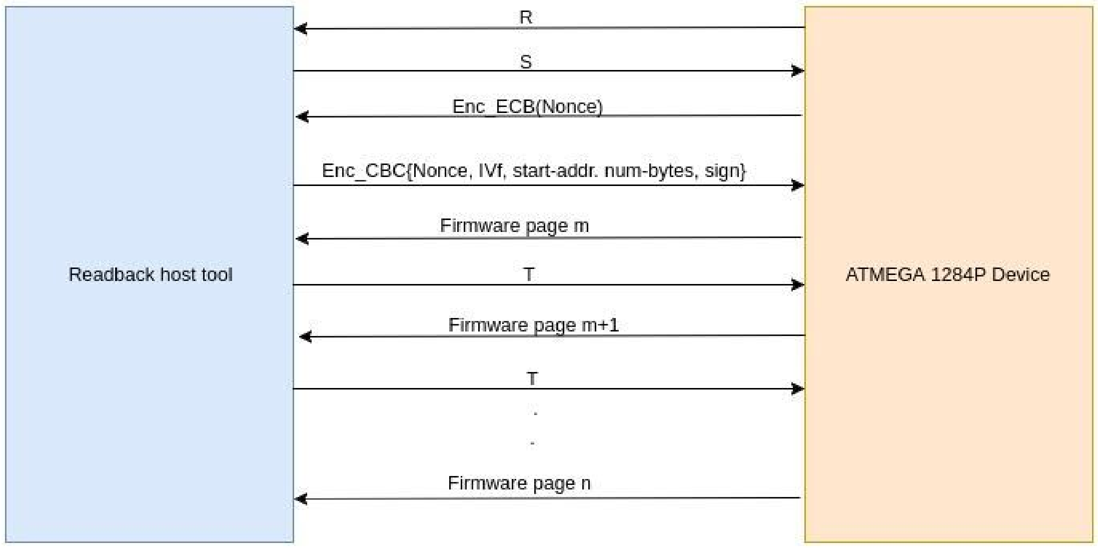
This tool creates chunks of firmware and appends version number to each of them. It then signs and encrypts each of them. It calculates measurement/hash chain of the firmware and signs and encrypts it, then appends it at the end of the protected firmware. This measurement is later sent to the bootloader to verify integrity of firmware. Protect tool also adds a random seed value to the firmware which is later used as a source of randomness.

Update Tool

This tool reads the protected firmware and sends it to bootloader. It sends the measurement after sending the firmware.

Readback Tool:

The host tool and the readback functionality in bootloader perform a series of secure handshake steps before any firmware data is displayed to the technician. The diagram above gives a brief overview of the steps involved in the handshake. IVf is the initialization vector sent to the device which uses it as IV while encrypting firmware pages.



Readback Protocol

Protocol details

Bootloader

1. In the readback mode, the bootloader writes a 'R' on UART1, signifying that it is in readback mode.
2. The bootloader after receiving a 'S' from the host tool, generates a random nonce using the nonce generation logic in PRNG.c. This random nonce in encrypted in AES-ECB mode and sent to the host tool.
3. The bootloader receives the validation block from the host tool, it decrypts the block and then by using the start address and number of bytes sends the block of desired data first encrypted in AES-CBC mode, then sent page-by-page.

Readback tool

When the host tool receives an 'R' from bootloader over UART1, it sends a 'S' on UART1.

1. On receipt of 'S', Bootloader encrypts a random nonce in ECB mode, sends to host.
2. The encrypted nonce is read by the host tool. This nonce is then appended to the request and sent to bootloader encrypted in AES CBC mode. We call this chunk of encrypted message as Validation Block and its individual elements are given below:
   IV_R = IV(16 bytes) used by AES-CBC on host side
   IV_F = IV(16 bytes) used by AES-CBC on bootloader side
   HMAC = HMAC-SHA1 hash

IV_R || (((nonce || IV_F || START_ADDR || SIZE) || HMAC) || PAD)) Enc_{IVR}

3. The encrypted block of data is received page-by-page. The desired number of bytes are kept and the remaining bytes are stripped off. (Bootloader sends bytes that are page aligned, so from the last page all the extra bytes are stripped off before printing the output)

Nonce generation logic

Nonces are generated as follows:

```
HMAC-SHA1(timer value || Bootloader seed || firmware seed)
```

Bootloader seed: Seed acquired from the secure bootloader section.(SECURE_VAULT)
Firmware seed: Seed acquired from the metadata of a valid firmware update

These three seeds are concatenated and a hash(HMAC-SHA1) is computed. The truncated value of this hash is used as the random nonce.

Important Note on Compiling Firmware

The firmware will not be loaded at 0x0000 since our design places cryptographic libraries at that location. A linker flag with the correct start address must be used when compiling any firmware that is to run on our device:

CLINKER = -Wl,--section-start=.text=0x2f00

The correct address is specified by the constant APP_START_ADDR in bootloader.c