MITRE eCTF 2024:
Medical Infrastructure Supply Chain (MISC)
Design Documentation

Jacob White, Jihun Hwang (Jimmy), Jack Roscoe, and Jaxson Pahukula

Purdue University (b01lers)

March 11, 2024

1 Introduction

Competition Overview. Embedded Capture The Flag™ (eCTF) is a semester-long cybersecurity competition which allows students to develop secure designs for and attacks against embedded systems. MITRE's eCTF is split into two phases. In the first phase, teams design and implement a secure embedded system based on a set of functionality and security requirements. In the second phase, teams analyze and attack the designs of other competing teams, receiving "flags" of the form `ectf{...}` and being awarded points for doing so, with a number of additional points for being the first to successfully attack a team's design, having the best posters and documentation, etc.

The overarching theme for MITRE eCTF 2024 [MIT24] is Medical Infrastructure Supply Chains (MISC), where "medical device" components must be authenticated and validated and must keep attestation data confidential, all while attackers have physical access to the devices or even insider access to component manufacture and replacement processes.

Team Overview. We are a team of 17 undergraduate and graduate students, all collaborating on this competition in-person at Purdue University for the Spring 2024 semester. Our team consists of mostly members of the Purdue Capture the Flag (CTF) team `b01lers`, along with some of the faculty advisors' students. The team is co-advised by Professors Christina Garman (CS), Kazem Taram (CS), and Santiago Torres-Arias (ECE).

Board Overview. The competition uses Analog MAX78000FTHR evaluation boards for all embedded devices. The boards can represent a medical device's Components (e.g., an insulin pump) or an Application Processor (i.e., a microcontroller for all Components).

2 Hardware Security Features

In order to ensure secure operation even while attackers have physical access to the board, we have extensively explored hardware security protections and countermeasures to “harden” the embedded system against side-channel, injection, and other hardware attacks: This includes using cryptographically-secure random number generators (CSPRNGs) instead of hardware-based “true” random number generators (TRNGs) whenever possible, constant-time memory and cryptographic operations, clock glitching and other fault injection detection, memory layout randomization, and even power analysis obfuscations, where relevant.

To ensure that the necessary cryptographic secrets and other data stored on these boards remain tamper-proof and confidential, we store them in the 512 KB flash memory. Since this flash memory resides on-chip, this makes it near-impossible for attackers to probe the flash memory for its contents or overwrite it with its own values.

3 Threat Model

3.1 Devices

Component. A Component performs functionality that is necessary for a medical device to function as intended. The functionality of these Components are arbitrary and irrelevant to this competition, but might represent something like e.g. an insulin pump’s actuator.

Application Processor. The Application Processor (AP, for short) functions as a microcontroller for the Medical Device’s Components, with Host tools controlling it via UART.

Medical Device. An AP together with its Component(s). A MISC medical device must support the following functionality: (1) list all provisioned and connected Components; (2) replace a provisioned Component; (3) attest to confidential Component data; (4) boot with valid Components; and (5) send authenticated data between the AP and Components.

3.2 Users

Attacker. An Attacker is a malicious party who is attempting to violate the security properties of a medical device – install counterfeit Components, compromise the build process, leak confidential data, etc. The Attacker either has physical or supply chain access.

Manufacturer. The Manufacturer is a trusted party which generates global secrets, builds AP firmware, and provisions the AP with data which helps authenticate Components on the medical device. We emphasize that, unlike some real-world supply chain scenarios, it remains outside the competition’s threat model for an Attacker to compromise the global secrets.

Component Facility. The Component Facility uses secure global secrets generated by the Manufacturer to build arbitrary medical Component firmware for a medical device. In some scenarios, the Attestation PIN which allows authorized Users access to a Component's confidential Attestation Data (e.g. patient data) might be compromised by the Attacker.

Repair Technician. The Repair Technician is responsible for replacing "damaged" Components (i.e., not visible to AP) with newly provisioned ones. In some scenarios, the replacement token for validating the Component swap might be compromised by the Attacker.

Operator. The Operator (e.g., a patient or technician) runs the Manufacturer's provided host tools on a general-purpose PC, sending commands to the AP over a UART serial interface to operate the MISC medical device. This simulates real-world device controls.

3.3 Security Requirements

(SR1) The AP must boot only after it confirms the validity of all necessary Components.

(SR2) The Component must confirm the validity of the AP. Components must not boot before being commanded to do so by a valid AP.

(SR3) The Attestation PIN and Replacement Token must be kept secret to prevent attackers from accessing Attestation Data or booting with counterfeit Components.

(SR4) The Attestation Data must remain confidential at all times, and should only be revealed to a valid AP after the user supplies a correct Attestation PIN.

(SR5) The post-boot MISC communications must be secure against any tampering, forgery, and replay attacks; that is, message authenticity and integrity must be preserved.

For more information, refer to "Security Requirements" in the documentation [MIT24].

3.4 Attack Scenarios

(AS1) **Operational Device**: Attackers do not have the PIN or replacement token, but have physical access to a fully functional and valid system (i.e., AP and 2 Components).

(AS2) **Damaged Device**: Same as AS1, but attackers only have 1 working Component.

(AS3) **Supply Chain Poisoning**: Attackers obtain the Attestation PIN and Replace Token, and can load counterfeit firmware onto the board to extract data and boot.

(AS4) **Black Box**: Attackers reverse-engineer 1 Component to extract data and boot.

For more information, refer to the "Attack Phase Flags and Scenarios" section of the documentation [MIT24]. We used a variant of the MITRE ATT&CK framework to assess the most important exploits related to the above.

3.5 Deployments

Each team's medical devices are built and distributed to attacking teams in 3 "Deployments":

(D1) AP1 provisioned for Components A & B, AP2 provisioned for Components C & D, and Components A,B, & C;

(D2) Counterfeit Component X; and

(D3) AP3 provisioned for Components A & B, Component A, and Replace Token & Attestation PIN

We assume the MITRE Organizers' bootloader and flags are securely stored on the boards, and regardless (as per eCTF rules) will not be directly attacked by other teams [MIT24].

3.6 Attack Flags

In summary, the table below describes each flag that an attacker is trying to gain [MIT24]:

Flag Name	Deployment	Scenario	Flag Format
Operational Pin Extract	D1	(AS1, SR3)	`ectf{pinextract_*}`
Operational Pump Swap	D1	(AS1, SR5)	`ectf{pumpswap_*}`
Damaged Boot	D1	(AS2, SR1)	`ectf{damagedboot_*}`
Black Box Boot	D2	(AS4, SR2)	`ectf{blackboxboot_*}`
Black Box Extract	D2	(AS4, SR4)	`ectf{blaxkboxextract_*}`
Supply Chain Boot	D3	(AS3, SR1)	`ectf{supplychainboot_*}`
Supply Chain Extract	D3	(AS3, SR4)	`ectf{supplychainextract_*}`

4 Cryptographic Primitives

In this section, we outline the cryptographic primitives/schemes that will be used to provide data confidentiality, authenticity, and integrity to our protocols. When choosing appropriate cryptographic schemes,we ensured that all the cryptographic primitives, their implementations, and other dependencies were open-source and either audited or formally verified, were designed to be side-channel resistant, contained no memory access errors, contained no known vulnerabilities related to weak randomness, and were used together in secure ways.

In the rest of this section, we describe the cryptographic schemes which we use to implement our secure MISC design, as described in Section 5.

4.1 Random Number Generation

Randomness is the a fundamental building block of many cryptography schemes, allowing secrets to be generated in an unpredictable manner.

The ARM TrustZone True Random Number Generator (TRNG) [ARM] will be used as the primary source of hardware-level randomness. To increase the rate of software-level random number generation and provide cryptographic assurances of continued high-entropy output, the TRNG will be used to seed a cryptographically-secure pseudorandom number generator (CSPRNG) as intended by the official documentation [ARM].

We decided to use a ChaCha20-based CSPRNG for similar reasons as the underlying stream cipher (see below). We also tested the quality of our random generators—both the CSPRNG used by our protocols and the TRNG used to seed it—by using the NIST SP 800-22 and Dieharder test suites, evaluating its results to be free of statistical bias.

4.2 HMAC

A hash-based message authentication code (HMAC) is used to provide data integrity. As the name implies, HMAC relies on the security of cryptographic hash functions. Informally, it uses a symmetric key on the message to generate a *tag* τ, which can be verified re-computing the tag themselves and checking that the tag is the same.

Given a secret key k and message m, HMAC can be built from a hash function H as follows:

$$\mathsf{HMAC}_k(m) \coloneqq H(k_1 \| H(k_2 \| m))$$

where $k_1 \coloneqq k \oplus (\texttt{0x36} \cdots \texttt{0x36})$ and $k_2 \coloneqq k \oplus (\texttt{0x5c} \cdots \texttt{0x5c})$. For security purposes, we will be using SHA-256 for the hash function. More details about HMAC can be found in the RFC 2104 standards document [KBC97].

4.3 ChaCha20 Authenticated Encryption

Authenticated Encryption (AE) is a cryptographic primitive which can be used to provide both confidentiality and authenticity of messages. Informally, AE can be thought of as an encryption together with a MAC tag τ, which is *internally* included as part of the ciphertext c and carefully checked during decryption.

Given a shared secret key k, encrypting an authenticated message m into ciphertext c is described as follows, where n is a public random nonce ("number used once"):

$$\begin{aligned} c &\coloneqq \mathsf{Enc}_k(m, n) \\ \mathsf{Dec}_k(c, n) &\coloneqq m \text{ or } \mathsf{Error} \end{aligned}$$

We decided to use ChaCha20-Poly1305 as it is one of the most secure symmetric authenticated encryption schemes against various side-channel and RNG-based attacks, and is designed to work well on embedded systems. As message authenticity is crucial to the design of MISC, ChaCha20-Poly1305 will provide an additional layer of authenticity in our protocol where needed. Curious readers can reference the RFC 8439 standards document [NL18] for more details on its construction

4.4 Ed25519 Digital Signatures

Digital signatures, similar to Message Authentication Codes (MACs), are used to authenticate the integrity of data. Unlike MACs, however, they have two separate keys: a *private* signing key to sign some message m and produce a *signature*, and a *public* verification key to check that the message has not been tampered with since signing.

Given a public-private keypair $(\mathsf{pk}, \mathsf{sk})$, signing a message m is described as follows:

$$\sigma := \mathsf{Sign}_{\mathsf{sk}}(m)$$
$$\mathsf{Ver}_{\mathsf{pk}}(m, \sigma) \to \{0, 1\}$$

We decided to use Ed25519 (i.e., EdDSA over the Curve25519 Twisted Edwards elliptic curve) due to its robustness against various side-channel attacks and RNG-based attacks, and is also easily compatible with embedded systems. Curious readers can reference the RFC 8032 standards document [JL17] for more details on its construction.

4.5 bcrypt Password Hashing

Informally, a hash function $H : \{0,1\}^* \to \{0,1\}^\ell$ takes in an arbitrary-length output and compresses it into a fixed-length (ℓ-bit) output. We decided to use `bcrypt` since it is a very well-established password-hashing scheme which provides memory-hardness guarantees which remain within the functional specifications of the relevant protocol and the operational limitations of the board.

5 MISC Protocols

5.1 Build & Deployment

Global Secrets. We rely on a simplified version of Public Key Pinning [WSM$^+$23], where trust relies on directly "pinning" an identity/context to and checking the usage of keys (in our case, via trusted Manufacturer pre-loading PKI keypairs and firmware onto the board), instead of using trusted Certificate Authorities or neighbors to build a hierarchy/web of trust for PKI. As such, all secrets must be loaded onto the board at build time, and can only be used in the specific context/protocols it is intended to be used in.

For the sake of brevity, anything marked $*\mathsf{sk}$ is a secret key, anything marked $*\mathsf{pk}$ or $*\mathsf{vk}$ is a public key, and all cryptographic data must be securely stored in tamper-proof memory. See Appendix A and each protocol below for the meaning and usage of relevant notation.

Error Handling Defenses. Note that, for all protocols, whenever an attack is detected (e.g., PIN or authentication token is incorrect; decryption, HMACs, signature, or other identifier cannot be verified; hardware-level glitching or tampering is suspected), we delay for an additional 5 seconds before resetting back to a ready/waiting state. To ensure that this complies with the MITRE eCTF rules [MIT24], and for other reasons , each side of the communication protocol must timeout after the maximum protocol time has been reached.

The 5s timeout requirement on suspected attacks helps mitigate brute-force and hardware attacks to some degree. In addition, we ensure that all relevant protocol-related checks will fallthrough to a generic protocol error state. While such error handling cases are left out of the protocol design documentation for brevity, they are included in our implementation.

5.2 List Protocol

The List protocol lists all connected and provisioned Components on the Medical Device, and works as follows:

```
List()
print all ProvCID's
for Component I2C addr do
    get CID from the component
    print CID only
```

Functionality. To allow any device to initiate the List protocol, both the AP's and Component's firmware should be able to initiate (`LIST_HELLO`) and acknowledge (`LIST_ACK`) List requests. It must take at most 3 seconds to complete the List protocol.

Security. The List protocol is **public and non-authenticated**, and must NOT be considered valid in other protocols. It is not related to any security requirements (Section 3.3).

5.3 Attestation Protocol

The attestation protocol provides a method for an AP to request sensitive "attestation data" from a component (i.e. CID, location, date, customer) which helps identify the component. The caller provides a private Attestation PIN (abbr. APIN) and the ID of the Component to attest (abbr. CID) over UART. Attestation Data must have the following format:

```
C>0x{CID}\n
LOC>{Attestation location}\n
DATE>{Attestation date}\n
CUST>{Attestation customer}\n
Attest\n
```

Our attestation protocol can be outlined as follows:

$\mathsf{Attest}(\mathsf{APIN}', \mathsf{CID})$

Application Processor	(I2C channel)	**Component** i
$h \coloneqq H(\mathsf{APIN} \Vert \mathsf{asalt})$ **from memory**		
$h' \leftarrow H(\mathsf{APIN}' \Vert \mathsf{asalt})$		
Verify $h = h'$		
	$\xrightarrow{\texttt{ATTEST\_INIT}}$	
		$n \leftarrow\$\ \mathsf{CSPRNG}()$
	$\xleftarrow{n}$	
$m_{A1} \leftarrow \texttt{ATTEST\_REQ}$		
$\tau \leftarrow \mathsf{HMAC}_{\mathsf{hsk}}(m_{A1} \Vert \mathsf{CID} \Vert n')$		
	$\xrightarrow{m'_{A1}, \mathsf{CID}, n', \tau}$	
		Verify $\mathsf{HMAC}_{\mathsf{hsk}}(m'_{A1} \Vert \mathsf{CID}_i \Vert n') = \tau$ and $\mathsf{CID} = \mathsf{CID}_i$ and $n' = n$ and $m'_{A1} = \texttt{ATTEST\_REQ}$
		if all checks pass **then** $m_{A2} \leftarrow \texttt{ATTEST\_RESP}$ $c \coloneqq \mathsf{Enc}_{\mathsf{ask}}(\mathsf{ADATA})$ **from memory**
	$\xleftarrow{m'_{A2}, c}$	
Verify $m'_{A2} = \texttt{ATTEST\_RESP}$		
$\mathsf{ADATA} \leftarrow \mathsf{Dec}_{\mathsf{ask}}(c)$		
return ADATA		

Assumption 1. *We assume that the Attestation data (*ADATA*) should never change after the Component firmware is built and loaded onto the board, and that it will never need to change unless it's sent back to the Component Facility for modifications. As a consequence, any Component i does NOT need to store the* ask *key that was used to encrypt its* ADATA*.*

Functionality. Only the AP should be able to initiate (ATTEST_REQ) Attestation requests. Only Components should be able to listen to Attestation requests and respond (ATTEST_RESP) to the Attestation requests. Honest Components checking the request's CID ensures only the expected Component will respond with its Attestation data (ADATA). It must take at most 3 seconds to complete Attest, which is easily doable using only symmetric HMAC and encryption as well as careful choice of work factor for the APIN hash. However, if an attack is detected (here, PIN check failing with $H(\mathsf{APIN}' \,\|\, \mathsf{asalt}) \neq H(\mathsf{APIN} \,\|\, \mathsf{asalt})$ means malicious Operator, and HMAC check failing with wrong τ or n means malicious AP) we are allowed to delay for an additional 5 seconds. If we can assume that the Attestation data is static (see above), storing only the ciphertext of the Component's ADATA in its memory is sufficient to ensure that a valid AP with the Component i's attestation key ask can decrypt to ADATA.

Security. If the AP or Component detects that the other party provides incorrect input (i.e. bad APIN or HMAC tag; see above), we delay for 5 seconds to limit the impact of brute-force attacks on the APIN. We keep asalt private and hash the APIN to frustrate dictionary and side-channel attacks. Component checks for matching "challenge" nonce in HMAC to prevent impersonating AP . Authenticated encryption and public-key pinning also prevent impersonating APs or Components. Components do not store its attestation key ask .

5.4 Replace Protocol

The replace protocol reprograms the AP with a new Component, overwriting a previously provisioned Component. The caller provides a private Replacement Token RTok, the ID of the component to replace $\mathsf{CID}_{\mathsf{out}}$, and the ID of the new component $\mathsf{CID}_{\mathsf{in}}$. Our boot protocol can be outlined as follows:

```
$\mathsf{Replace}(\mathsf{RTok}, \mathsf{CID}_{\mathsf{in}}, \mathsf{CID}_{\mathsf{out}})$
Application Processor

if Verify $H(\mathsf{RTok}) = H(\mathsf{RTok}_{\mathsf{actual}})$
      and $\mathsf{CID}_{\mathsf{out}} \in \mathsf{ProvCID}$
      and $\mathsf{CID}_{\mathsf{in}} \notin \mathsf{ProvCID}$
 then
   $\mathsf{ProvCID}[\mathsf{CID}_{\mathsf{out}}] \leftarrow \mathsf{null}$
   $\mathsf{ProvCID}[\mathsf{CID}_{\mathsf{in}}] \leftarrow \mathsf{open}$
```

Functionality. Only the AP can run Replace. It must take at most 5 seconds to complete Replace, which is enforced by carefully tuning the password-hash $H(\mathsf{RTok})$. If an attack is detected (here, $H(\mathsf{RTok})$ check failing) we are allowed to delay for an additional 5 seconds. The liveness of the new Component $\mathsf{CID}_{\mathsf{in}}$ are not checked until a downstream protocol (i.e., Boot). The list of provisioned Component IDs ProvCID will be retained across power cycles.

Security. This protocol does not allow for communication between the AP and Components $\mathsf{CID}_{\mathsf{in}}$, $\mathsf{CID}_{\mathsf{out}}$, so authentication and liveness checks in the newly-provisioned Component $\mathsf{CID}_{\mathsf{in}}$ most occur in downstream protocols (i.e., Boot). If the AP detects an incorrect input (i.e. bad RTok; see above), we delay for 5 seconds to mitigate brute-force attacks.

5.5 Boot Protocol

The boot protocol first authenticates the AP and its two Components as valid and provisioned, before having each Component send back its encrypted boot message to indicate that it is ready to boot. The AP then prints the component's boot message as well as its own, and all parts of the MISC device then disable pre-boot functionality, only allowing post-boot messaging thereafter (see Section 5.6). Our boot protocol can be outlined as follows:

```
Boot()

Application Processor                 (I2C channel)            Component $i$

for $(\mathsf{CID}_i, \mathsf{idx}_i) \in \mathsf{ProvCID}$ do
    $m_{B1} \leftarrow \texttt{BOOT\_CHAL}$
    $(r_{B1}, n_{B1}) \leftarrow\$ \mathsf{CSPRNG}()$
    $c_{B1} \leftarrow \mathsf{Enc}_{\mathsf{csk}}(m_{B1} \| \mathsf{CID}_i \| r_{B1}, n_{B1})$

                              $\xrightarrow{c_{B1}, n_{B1}}$

                                                    $(m_{B1}, \mathsf{CID}'_i, r_{B1}) \leftarrow \mathsf{Dec}_{\mathsf{csk}}(c_{B1}, n_{B1})$
                                                    Verify $\mathsf{CID}'_i = \mathsf{CID}_i$
                                                    $m_{B2} \leftarrow \texttt{BOOT\_CRSP}$
                                                    $(r_{B2}, n_{B2}) \leftarrow\$ \mathsf{CSPRNG}()$
                                                    $M_{B2} \leftarrow m_{B2} \| \mathsf{CID}_i \| \mathsf{idx}_i \| r_{B1} + 1 \| r_{B2}$
                                                    $\sigma_i \leftarrow \mathsf{Sign}_{\mathsf{msk}_i}(M_{B2})$
                                                    $c_{B2} \leftarrow \mathsf{Enc}_{\mathsf{csk}}(M_{B2} \| \sigma_i, n_{B2})$

                              $\xleftarrow{c_{B2}, n_{B2}}$

    $(M'_{B2} \| \sigma_i) \leftarrow \mathsf{Dec}_{\mathsf{csk}}(c_{B2}, n_{B2})$
                              (continued...)
```

```
Boot()
                                   (continued...)
  if $\mathsf{idx}_i = \mathsf{open}$ then
    $\mathsf{idx}_i \coloneqq \mathsf{ProvCID}[\mathsf{CID}_i] \leftarrow \mathsf{idx}'_i$
  Verify $\mathsf{idx}_i \neq \mathsf{null}$
  $\mathsf{mvk}_i \leftarrow \overline{\mathsf{mvk}}[\mathsf{idx}_i]$
  Verify $\mathsf{Ver}_{\mathsf{mvk}_i}(M_{B2}, \sigma_i) = 1$
        and $m_{B2} = \texttt{BOOT\_CRSP}$
        and $\mathsf{CID}'_i = \mathsf{CID}_i$
        and $r'_{B1} = r_{B1} + 1$
if Both Components valid then
  for $(\mathsf{CID}_i, \mathsf{idx}_i) \in \mathsf{ProvCID}$ do
    $m_{B3} \leftarrow \texttt{BOOT\_RESP}$
    $n_{B3} \leftarrow\$ \mathsf{CSPRNG}()$
    $M_{B3} \leftarrow m_{B3} \| \mathsf{CID}_i \| r_{B2} + 1$
    $\sigma_{AP} \leftarrow \mathsf{Sign}_{\mathsf{msk}_{AP}}(M_{B3})$
    $c_{B3} \leftarrow \mathsf{Enc}_{\mathsf{csk}}(M_{B3} \| \sigma_{AP}, n_{B3})$

                              $\xrightarrow{c_{B3}, n_{B3}}$

                                            $(M'_{B3} \| \sigma_{AP}) \leftarrow \mathsf{Dec}_{\mathsf{csk}}(c_{B3}, n_{B3})$
                                            Verify $\mathsf{Ver}_{\mathsf{mvk}_{AP}}(M_{B3}, \sigma_{AP}) = 1$
                                                  and $m_{B3} = \texttt{BOOT\_RESP}$
                                                  and $\mathsf{CID}'_i = \mathsf{CID}_i$
                                                  and $r'_{B2} = r_{B2} + 1$
                                              $m_{B4} \leftarrow \texttt{COMP\_BOOT\_MSG}$
                                              $n_{B4} \leftarrow\$ \mathsf{CSPRNG}()$
                                              $M_{B4} \leftarrow m_{B4} \| \mathsf{CID}_i$
                                              $c_{B4} \leftarrow \mathsf{Enc}_{\mathsf{csk}}(M_{B4}, n_{B4})$

                              $\xleftarrow{c_{B4}, n_{B4}}$

    $M''_{B4} \leftarrow \mathsf{Dec}_{\mathsf{csk}}(c_{B4}, n_{B4})$
    Verify $\mathsf{CID}''_i = \mathsf{CID}_i$
    $\mathsf{print}(\texttt{COMP\_BOOT\_MSG})$
  $\mathsf{print}(\texttt{AP\_BOOT\_MSG})$
$\mathsf{disable\_preboot\_and\_boot}()$                      $\mathsf{disable\_preboot\_and\_boot}()$
```

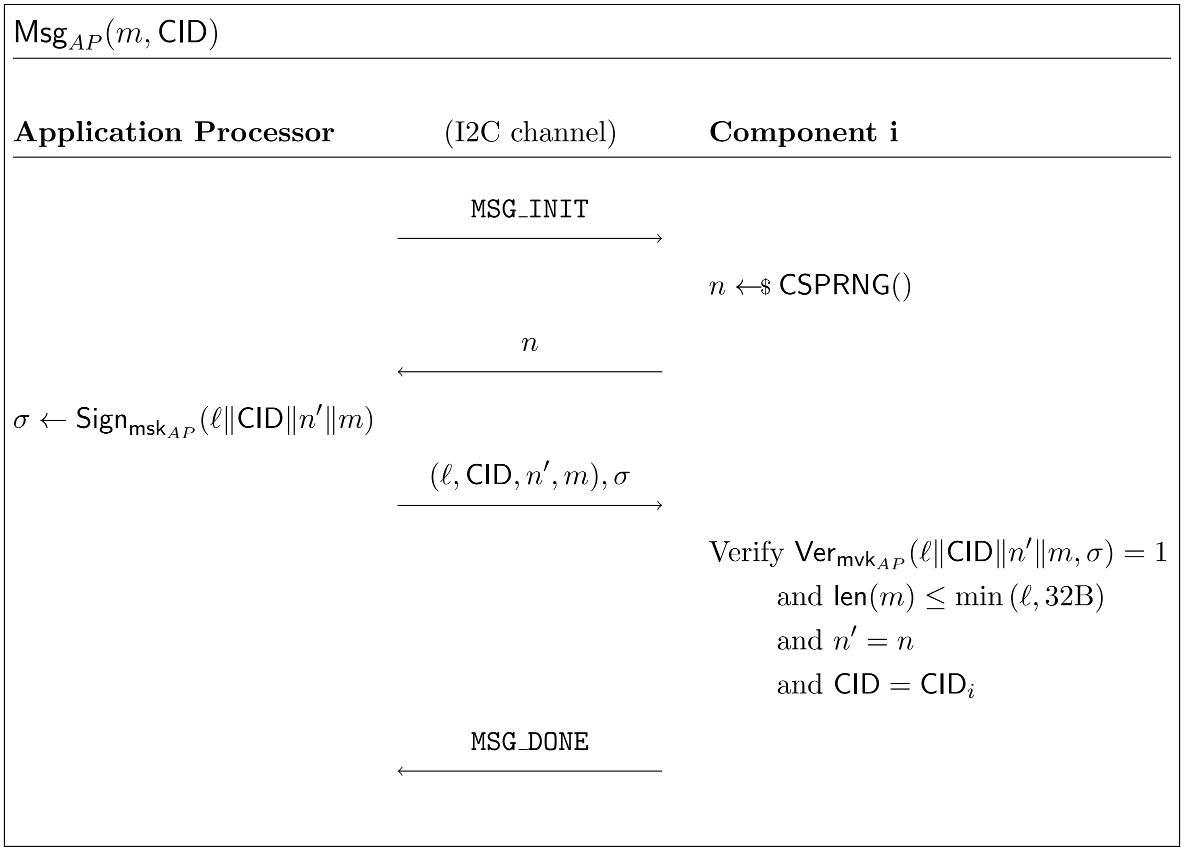
Functionality. The AP first checks to make sure both Components are live via a two-way combined Challenge-Response (CR) subroutine before booting, and the Component indicates to the AP that it is ready to boot by sending a boot message to the AP through the same CR subroutine. If a Component was replaced prior to boot (i.e., $\mathsf{ProvCID}[\mathsf{CID}_i] = \mathsf{open}$), its key-pin ID idx_i is received and persistently recorded by the AP for validation to function. Despite computationally expensive Sign and Ver checks and multiple rounds of I2C communication, we have ensured that it takes at most 3 seconds to complete Boot. See Section 5.6 for more details on the feasibility of public key-pinning signing keys.

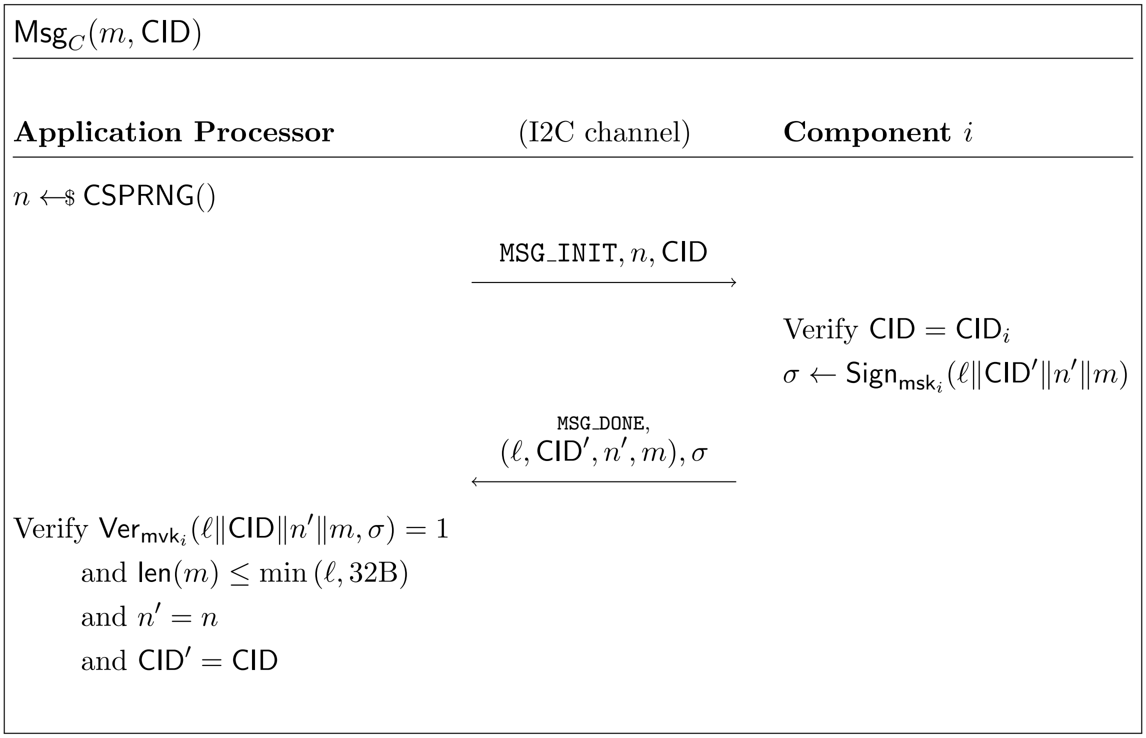
Security. Components are checked for liveness, uniqueness of $(\mathsf{CID}_i, \mathsf{idx}_i)$, and validity before the AP accepts the CR authentication checks. The CR subroutine provides a basic validity check via test-decryption, which secondarily serves to frustrate statistical attacks on the RNG . Additionally, digital signature checks in the CR subroutine also ensure that the AP and all components are identified and valid parts of the MISC device. Both the AP and Component can only boot after individually authenticating each other. Unique messages and challenges/nonces for each encryption help mitigate "oracle" and replay attacks. The random nonces r_{B1} and r_{B2} also serve as unique challenges and as session-binding. In case the Component's encrypted boot message contains the flag or other sensitive data, the received `COMP_BOOT_MSG` is encrypted when being sent back to the AP.

We also note that, even though in real boot protocols it is generally advisable to undergo some form of Attest protocol to verify the integrity of the device's components, this competition excluding RTok from the input of Boot, not providing any uniqueness guarantees on $H(\mathsf{ADATA})$ as a uniquely identifiable "fingerprint" of each individual Component, and not including any unique AP identifier makes it inadvisable to sanely or safely implement boot-attestation checks in the usual manner.

5.6 Messaging Protocol

After the MISC Medical Device (i.e., AP together with its Components) boots, the only valid functionality the device is allowed to perform is to send authenticated messages between the Application Processor and the Components. Our post-boot messaging protocol can be outlined as follows:


$\mathsf{Msg}_{AP}(m, \mathsf{CID})$
Application Processor
(I2C channel)
Component i
MSG_INIT
$n \leftarrow\!\!\$\ \mathsf{CSPRNG}()$
n
$\sigma \leftarrow \mathsf{Sign}_{\mathsf{msk}_{AP}}(\ell \| \mathsf{CID} \| n' \| m)$
$(\ell, \mathsf{CID}, n', m), \sigma$
Verify $\mathsf{Ver}_{\mathsf{mvk}_{AP}}(\ell \| \mathsf{CID} \| n' \| m, \sigma) = 1$
and $\mathsf{len}(m) \leq \min(\ell, 32\text{B})$
and $n' = n$
and $\mathsf{CID} = \mathsf{CID}_i$
MSG_DONE


$\mathsf{Msg}_{C}(m, \mathsf{CID})$
Application Processor
(I2C channel)
Component i
$n \leftarrow\!\!\$\ \mathsf{CSPRNG}()$
MSG_INIT, n, CID
Verify $\mathsf{CID} = \mathsf{CID}_i$
$\sigma \leftarrow \mathsf{Sign}_{\mathsf{msk}_i}(\ell \| \mathsf{CID}' \| n' \| m)$
MSG_DONE,
$(\ell, \mathsf{CID}', n', m), \sigma$
Verify $\mathsf{Ver}_{\mathsf{mvk}_i}(\ell \| \mathsf{CID} \| n' \| m, \sigma) = 1$
and $\mathsf{len}(m) \leq \min(\ell, 32\text{B})$
and $n' = n$
and $\mathsf{CID}' = \mathsf{CID}$

Functionality. Since the competition's rules in [MIT24] do not specify a need to support session-based handshakes or tear-down and since we can assume that the AP and Components have prior knowledge of each other (i.e., at manufacture time) and can agree on cryptographic algorithms out-of-band, we opt to avoid the complexities of TLS/SSL-based communication in favor of simpler tamper-proof protocols wherein only a single message is sent during each protocol instance. Since Boot is required to execute before Msg and because Boot verifies that both parties have valid signing/verification keypairs, it can be safely assumed that both parties know of the other's $\left\{\mathsf{mvk}_{AP}, \mathsf{mvk}_i \coloneqq \overline{\mathsf{mvk}}[\mathsf{ProvCID}[\mathsf{CID}_i]]\right\}$ to check against. We acknowledge that session-based protocols would be more efficient in terms of the number of rounds of I2C communication across multiple messages, but nevertheless our protocol still satisfies the requirement that it takes at most 1 second to complete each Msg.

We also acknowledge that storing all Components' public verification keys $\mathsf{mvk}_i \in \overline{\mathsf{mvk}}$ in flash memory is not the most scalable solution for many real-world scenarios (especially compared to, say, PKI distribution and validation via Certificate Authority). However, it may reasonably and theoretically scale to a non-trivial number of Components per deployment as long as we can assume that at least one of the following are true:

1. Signature keypairs $(\mathsf{msk}_i, \mathsf{mvk}_i)$ are unique to each Component Facility and Component *type* (e.g. 1 keypair for "MedCo" Insulin Pumps and 1 keypair for "Acme" Actuators, but same across all post-hoc manufactured "Acme" Actuators), with MISC Device Manufacturer sharing assigned key-pin identifier idx_i and Component Facility sharing mvk_i out-of-band prior to manufacture;
2. If unique keypairs per individual Component, the Manufacturer can pre-provision all such Components' identifying data onto each MISC Device's AP during assembly; or
3. Trusted Certificate Authorities and other online PKI web/chain-of-trust mechanisms are infeasible or non-scalable solutions to validate the identity associated with each verification key $\{\mathsf{mvk}_{AP}, \mathsf{mvk}_i\}$ compared to public key/certificate-pinning.

Security. We check to ensure that the message length ℓ of m does not exceed the speicfied maximum 32B and allocate sufficient memory to prevent potential buffer overflow attacks. We preserve integrity, authentication, and identification of the AP and Component through pre-established valid signing/verification keypairs (see above). Since Components are only manufactured with the pinned verification key mvk_{AP}, Components do not have access to the corresponding secret signing key msk_{AP} and due to technical limitations of the I2C communication firmware, Components are unable to directly message each other and must be mediated by a validated AP microcontroller. Unique "challenge" nonces for each signature help mitigate "oracle" and replay attacks, and without session-based messaging does not require stateful nonce list management (N.B. if both parties are malicious, security guarantees are meaningless here). Due to public key-pinning of the verification keys $\mathsf{mvk}_i \in \overline{\mathsf{mvk}}$ and the inclusion of CID_i in the corresponding signature of each message, attackers cannot impersonate as a valid Component i without the corresponding private signing key msk_i (and vice versa for AP). Signatures ensure that all messages are tamper-proof.

References

[ARM] ARM Software. TZ-TRNG: TrustZone True Number Generator. GitHub Repository. ARM website: https://www.arm.com/products/silicon-ip-security/random-number-generator.

[JL17] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017.

[KBC97] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[MIT24] MITRE Corporation. 2024 eCTF: The 2024 MITRE Engenuity Embedded Capture the Flag Competition, 2024. https://ectfmitre.gitlab.io/ectf-website/2024/index.html.

[NL18] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018.

[WSM$^+$23] Jeffery Walton, John Steven, Jim Manico, Kevin Wall, and Ricardo Iramar. Certificate and Public Key Pinning, March 2023. OWASP Foundation, LLC.

A Terminology and Notation

Abbreviations

AE Authenticated Encryption.

AES Advanced Encryption Standard.

AP Application Processor.

API Application Programming Interface.

AS Attack Scenario (aka Attack Phase in eCTF rules).

ASLR Address Space Layout Randomization.

C&C Command and Control.

CR Challenge-Response.

CSPRNG Cryptographically-Secure Pseudo-Random Number Generator.

eCTF Embedded Capture the Flag.

Ed25519 Edwards-curve Digital Signature Algorithm over Curve25519.

HMAC (Hash-based) Message Authentication Code.

HW Hardware.

I2C Inter-Integrated Circuit.

MISC Medical Infrastructure Supply Chain.

N/A Not Applicable.

NIST National Institute of Standards and Technology.

PIN Personal Identification Number.

PKI Public-Key Infrastructure.

RFC Request for Comments.

RNG Random Number Generator.

SHA Secure Hash Algorithms.

SR Security Requirement.

SSL Secure Socket Layer.

TLS Transpoprt Layer Security.

TRNG True Random Number Generator.

UART Universal Asynchronous Receiver-Transmitter.

List of Symbols

$\circ'$ Different or modified $\circ$.

$\overline{\circ}$ List of $\circ$.

$\leftarrow$ Assign equal to.

$\leftarrow\!\!\$$ Assign from random sample to.

$=$ Compare equal to.

$\neq$ Compare not equal to.

$:=$ Defined equal to.

$\{\ldots\}$ Set.

$\{\ldots\}^*$ Kleene star (≥ 0 repetitions in set).

$\|$ (Byte)string concatenation.

$\in$ (Is contained) in.

$\notin$ Not in.

$\oplus$ XOR.

APIN Attestation PIN.

ask Attestation Secret Key.

c Ciphertext.

CID Component IDentifier.

csk Challenge-response Secret Key.

H Cryptographic hash function.

hsk HMAC Secret Key.

idx Build Identifier (index to keylist).

k Cryptographic key.

ℓ Length (in Bytes).

m, M Message or plaintext.

msk Message Signing Key.

mvk Message Verification Key.

n Number used once (“nonce”).

pk Public key.

r Random number.

RTok Replace Token.

σ Signature.

sk Secret/signing key.

τ HMAC Tag.