NUAV@NUSec DESIGN DOCUMENTATION

[Due to this year’s constraints, our documentation logo is virtual and can be emulated in QEMU rather than being present here][1]

Version 0.99+0.01

Cameron Kennedy
Dennis Giese
Erik Uhlmann

Advised by: Prof. Guevara Noubir
(Northeastern University)

[1] Same applies to our EULA

German Engineering

Made in USA

Overview

System design idea

The first line of defense for our system design was our Terms of Service and EULA, where we sue any customer who dares hack our design or gain control of UAVs. We spent most of our time and budget discussing the EULA and deciding on our team name and emoji. Unfortunately, we were informed that the EULA was not actually enforceable or legally binding, so we quickly tried to implement some technical security measures in the remaining time.

The core SCEWL Bus system implements cryptographic protection of messages over the air, offering resilience against message forgery, tampering, and replay. Additionally, software protections lock down the system in the face of software exploits and make it more challenging to perform side-channel analysis.

In order to make it easier to manage cryptographic key material for many devices, we utilize a system of floating keys, which are dynamically distributed to booting devices by the SCEWL Security Server. This enables the design to save a lot of RAM which can be used for other security protections. Devices sign and encrypt messages which can only be decrypted by the intended target devices by using the target device keys to encrypt the key material for the message contents. Additionally, every message uses a random key for its contents, and we obfuscate the other encryption and signing operations with randomized fake operations to make SCA difficult, which is also protected by rate-limiting of send and receive operations within the allowed timing constraints. We additionally made modifications to the monocypher crypto library to add randomized delays to all crypto operations used by our code. Two levels of replay counters in the messages protect against replay attacks, and are constantly synchronized across the whole swarm using periodic internal broadcasts, as well as by the SSS when a device registers and deregisters in the live deployment.

Runtime entropy for any random numbers needed by our system is provided by a PRNG seeded with secure random bytes sent in the SSS registration flow, combined with entropy embedded into each controller binary. The entropy is periodically mixed by combining it with the system tick counter, in particular the tick counter values during I/O operations, ensuring that it becomes increasingly unpredictable even if the initial state is somehow leaked. Our PRNG implementation additionally includes randomized fake PRNG operations to thwart side-channel analysis on the real internal PRNG state.

Finally, we have enabled MPU-powered protection of memory regions against unauthorized reads, writes, and code execution, as well as stack canaries and MPU no-permissions zones for protection against buffer overflows. Additionally, the runtime system emits no diagnostic information of any kind which could be picked up by attackers, even in the case of errors.

Security goals

Confidentiality

Messages should be only decryptable and readable to the intended recipients. We will use both symmetric and asymmetric cryptography to achieve this goal.

Integrity

Messages should be protected against changes. We use Authenticated Encryption with Associated Data (AEAD) and signatures to prevent attackers of modification of messages.

Authentication

Messages in the deployment should be only accepted if they origin from authorized devices. This will be ensured by maintaining a list of authorized public keys and device identities. All messages need to have a signed message header, which can be verified by the receiving device.

Replay Protection

Attackers could try to replay existing messages again into the system. This will be prevented due to our implementation of freshness verification. Devices will know the current status of the message counter of a particular device and can reject replayed messages. In addition, we will use time-stamps and will reject messages which fall outside a timeframe.

Defense-in-Depth

To defend against buffer overflows, we enable the Memory Protection Units (MPU) functionality and use a new revision of our famous *Protectonator™.* These mechanisms prevent attackers from using any vulnerability in our code. The SSS was implemented in Racket[2], according to the Design Recipe[3] which provides useful abstractions for safe, efficient I/O with devices[4].
To protect against side channel attacks, we developed multiple defenses. For example, we obfuscate the swarm key, so that it is more difficult to observe. Our crypto operations are executed with both, real and fake keys. The order of the operations are randomized. In addition we add random delays before, inside and after crypto operations. The receiving and sending operations are rate-limited and increase the required time for side-channel attacks. To hide real transmissions, the swarm transmits fake messages.

[2] https://racket-lang.org/
[3] https://htdp.org/2020-8-1/Book/part_preface.html#(part._sec~3asystematic-design)
[4] Since the Ubuntu repos were 8 versions behind, we decided to compile the current Racket 8.0 release from source in our Dockerfiles. Be warned that this can make the first-time `create_deployment` step of our system take a bit of time

Protection of flags

UAV ID Recovery

“Read a broadcasted SCEWL Transmission”
When the device is provisioned it gets device specific keys. We are not using any long-term, systemwide used message encryption keys. All communications, including broadcasts are encrypted end-to-end. Therefore, a device which is not registered and correctly provisioned, will not receive a communication. Key material on a particular device will be protected by anti-side-channel measurements.

Package Recovery

“Read a targeted SCEWL Transmission”
Same as for broadcasted messages, targeted messages are encrypted for one particular party with its public key. An attacker requires the private key to decrypt that message. This key is only stored on the SED and the SSS.

Recovery Mode

“Modify the content of a SCEWL Transmission”
All messages are encrypted for one particular party and are signed by the sender. In addition, the payload is encrypted using a message key and AEAD encryption. This ensures the integrity of the payload. Broken messages cannot be decrypted and will not be forwarded to the CPU.

Drop Package

“Have full control of the plaintext contents of a SCEWL Transmission”
Similar as for the Recovery Mode, the integrity of the message is protected. In order to be able to send messages, the attacker would need the private key of a device and would need the public key of the target device. This information should not be available. Additionally, due to the encryption of the header with a master key, the attacker cannot observe device IDs of packets.

No-Fly Zone

“Take control of the UAV”
Being able to inject traffic requires again the presence of key material. A already sent message cannot be replayed, as devices keep track of a message counter. This counter gets updated every time a device receives a message or gets registered at the base. Additionally, we use a instance counter, so that newly deployed drones will not accept old messages. A timer is synchronized among all devices. Messages are only accepted if they have a valid counter and instance id.

Features

Simple and scalable system

System supports up to 256 provisioned devices, with up to 24 of them being in use at the same time. A message of 16 kBytes to 24 devices takes only 143 µs[5]. The overhead is minimal. By using the same operations for directed and broadcasted messages, the codebase is smaller, the attack surface is reduced, and the software is more efficient.

Device Anonymity

SED IDs and SCEWL IDs are not assigned permanently. Instead the SED IDs are used from a pool every time a Device gets registered. In addition, the keys are rotated.
Due to the design, the protocol does not reveal the source and destination of a particular communication. Also, fake messages are transmitted in between real messages.

Proven and standard cryptography

Our design uses well proven modern cryptographic methods. We use Monocypher[6], which is a small and efficient library, that provides all the functionality we need. The design uses the following methods: X25519 and XChaCha20 for encryption, Ed25519 for signatures and Blake2b for challenge-response authentication. Additionally, a random number generator is used to ensure secure cryptographic operations. It is seeded with external entropy. To protect key material, it is stored only in volatile memory. A leakage of binaries does not break any security guarantees.

Side-channel protections

Our system was designed with side-channel attacks in mind. For this we implemented multiple counter measurements against side-channel attacks:

1. All cryptographic operations contain random delays with random XOR operations. This prevents the alignment of traces in side-channel attacks.
2. Secret keys are blinded before use
3. Cryptographic operations are run multiple times with one real and multiple fake keys. The order of operations is randomized and fake computations are not used. In traces these operations are not immediately distinguishable from each other.

[5] However, the CPU interface is still rate-limited by the controller according to about 50% of the allowed timing requirements
[6] https://monocypher.org/

Buffer overflow protection

The used TI chip supports Memory Protection Units (MPU)[7], which is implemented by QEMU[8]. We configure and use this feature to protect our memory against buffer overflows attacks and unexpected behaviour. In addition, we implemented an additional level of memory protection, which will prevent any buffer overflows.

[7] https://www.ti.com/lit/ds/symlink/lm3s6965.pdf

[8] During implementation of this security measurement we accidentally marked the .text section as no-execute and then were wondering why the controller exploded

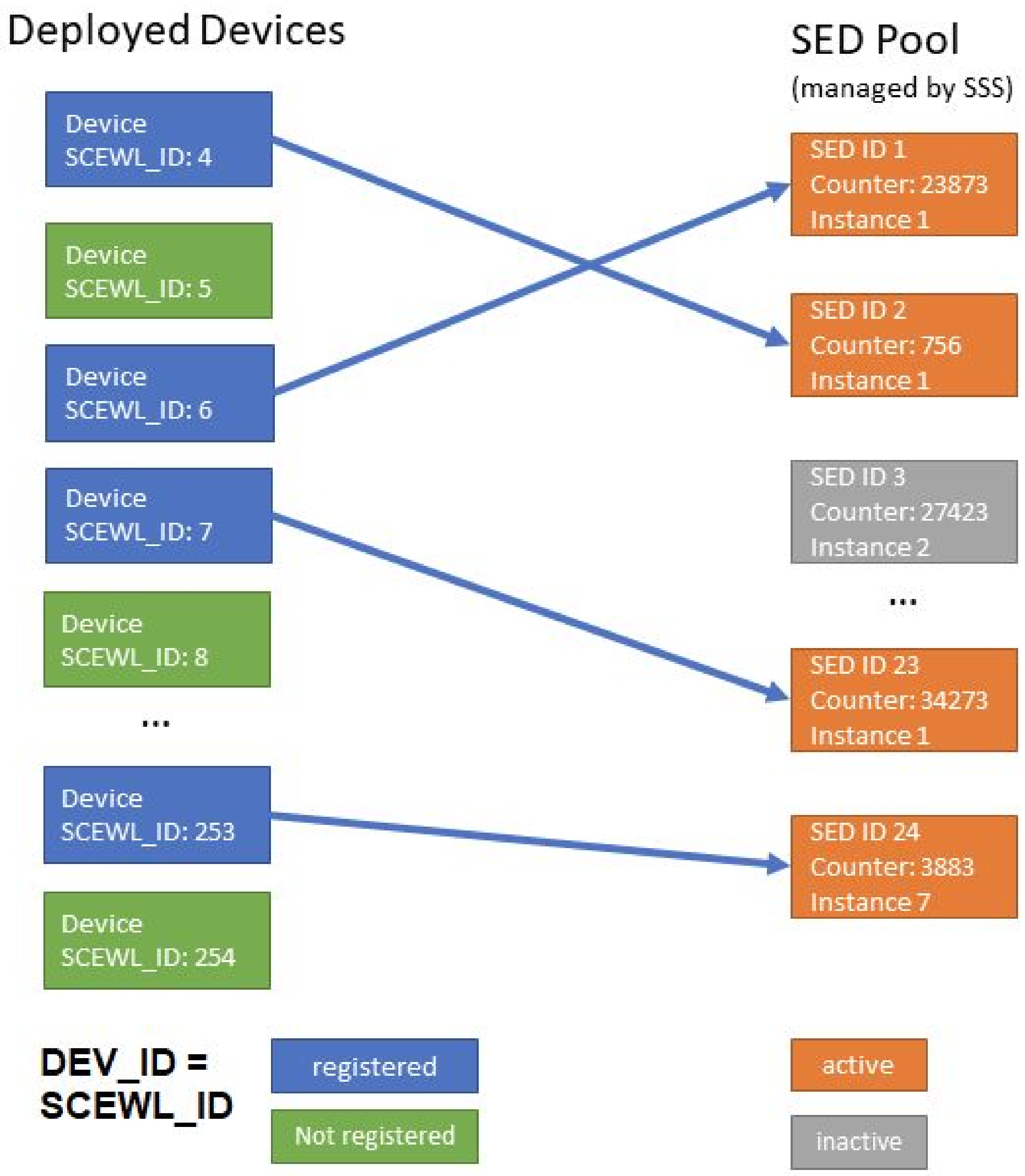
Implementation

Keys and Configuration

***DEV_ID** and **SCEWL_ID** are the same and are interchangeable.*

- Factory
    - Dev_credentials (**DEV_ID, Dev_Key**): unique device ID and secret string to serialize and identify devices
    - **Dev_DB:** list of all created devices with **DEV_ID** and **Dev_Key,** used by the SSS to authenticate devices
- SSS
    - **swarm_key:** symmetric encryption key which is distributed to all registered devices and used for encryption of a packet header
    - SED_credentials (**SED_ID, SED_xchgKey, SED_signKey**): asymmetric key material for registered devices, will be reassigned after de-registration
    - SED_DB (SED_credentials, counter, assigned **DEV_ID**, dereg_password, timeout, instance_counter): Pool of 24 generated SED_credentials (16 in use, 8 spare), with last known counter and information about the assigned device.
- Unregistered device
    - Dev_credentials(**DEV_ID, Dev_Key**): unique ID and secret string to serialize and identify devices, programmed in the factory and stored in flash memory
- Registered device/SED
    - Dev_credentials(**DEV_ID, Dev_Key**): unique device ID and secret string to serialize and identify devices, programmed in the factory and stored in flash memory
    - **swarm_key:** symmetric encryption key which is distributed to all registered devices and used for encryption of a packet header
    - SED_credentials (**SED_ID, SED_xchgKey, SED_signKey, current_counter, instance_counter**): asymmetric key material for registered devices and current message counter
        - **xchgKey :** x25519 keypair
        - **signKey :** ed25519 keypair
    - SED_pub_DB (**SED_ID, SED_xchgPubkey, SED_signPubkey, DEV_ID, counter, instance_counter):** Array of the public information of the SED_DB, provisioned by the SSS at registration. The SCEWL_ID can be updated through broadcasts. Counter keeps track of seen message IDs and is used for message freshness.
    - **random_seed**: seed for entropy, provisioned by the SSS at registration
    - **time**: time provisioned by the SSS
    - **dereg_password**: SED-specific password that gets set by the SSS and locks the wired interface

Relation DEV_ID, SCEWL_ID and SED_ID



Provisioning procedures

Create Deployment (Create_deployment)

(1a_create_sss.Dockerfile) → SSS Docker is build here

1. Install Racket compiler
2. Get random seeds from Random.org

Add Device to deployment (Add_sed)

(2b_create_sed_secrets.Dockerfile) → reuses SSS Docker

1. Get random seeds from Random.org
2. Generate Dev_Key (16 Byte secret string) for DEV_ID
3. Store DEV_ID:Dev_Key in SSS Database (SQLite)

(2c_build_controller.Dockerfile)

1. Get random seeds from Random.org
2. Build controller containing
    a. DEV_ID:Dev_Key
    b. Entropy
3. Delete unused secrets

Remove Device from deployment (remove_sed)

- Remove DEV_ID from SSS Database

Runtime procedures

Launch SSS (launch_sss_d)

Structs

<u>Swarm-key</u>

32 bytes random data

<u>DEV_DB</u>

- List of up to 256 Devices
    - DEV_ID: Device ID programmed at factory
    - Dev_Key: Secret key for Device

<u>SED_DB</u>

- Array with 24 Entries
    - SED ID
    - Xchg-key: X25519 keypair for encryption
    - Sign-key: Ed25519 keypair for signatures
    - DEV_ID: assigned device SCEWL_ID
    - Counter: Last known counter value for the particular SED
    - Dereg-time: deregistration timeout
    - Dereg-password: deregistration password
    - Inst_counter: Last known instance for the particular SED

Flow

1. Generate symmetric key swarm-key
2. Generate 24 SED entries in SED_DB

Registration of SED device (Scewl_register)

Used methods

- Blake2b hash function

Required data

- SSS_Password: secret for a deployment
- DEV ID: id between 1-256 identifying a device (aka SCEWL_ID)
- DEV_KEY: secret key for a specific DEV ID

Structs

<u>sss_handshake_out1</u>

- nonce_n : random nonce provided by the unprovisioned device
- id: device id, for unprovisioned devices = -1

<u>sss_handshake_in</u>

- nonce_y : random nonce provided by the SSS
- auth : authentication message, blake2b(SSS_password, nonce_n, nonce_y)

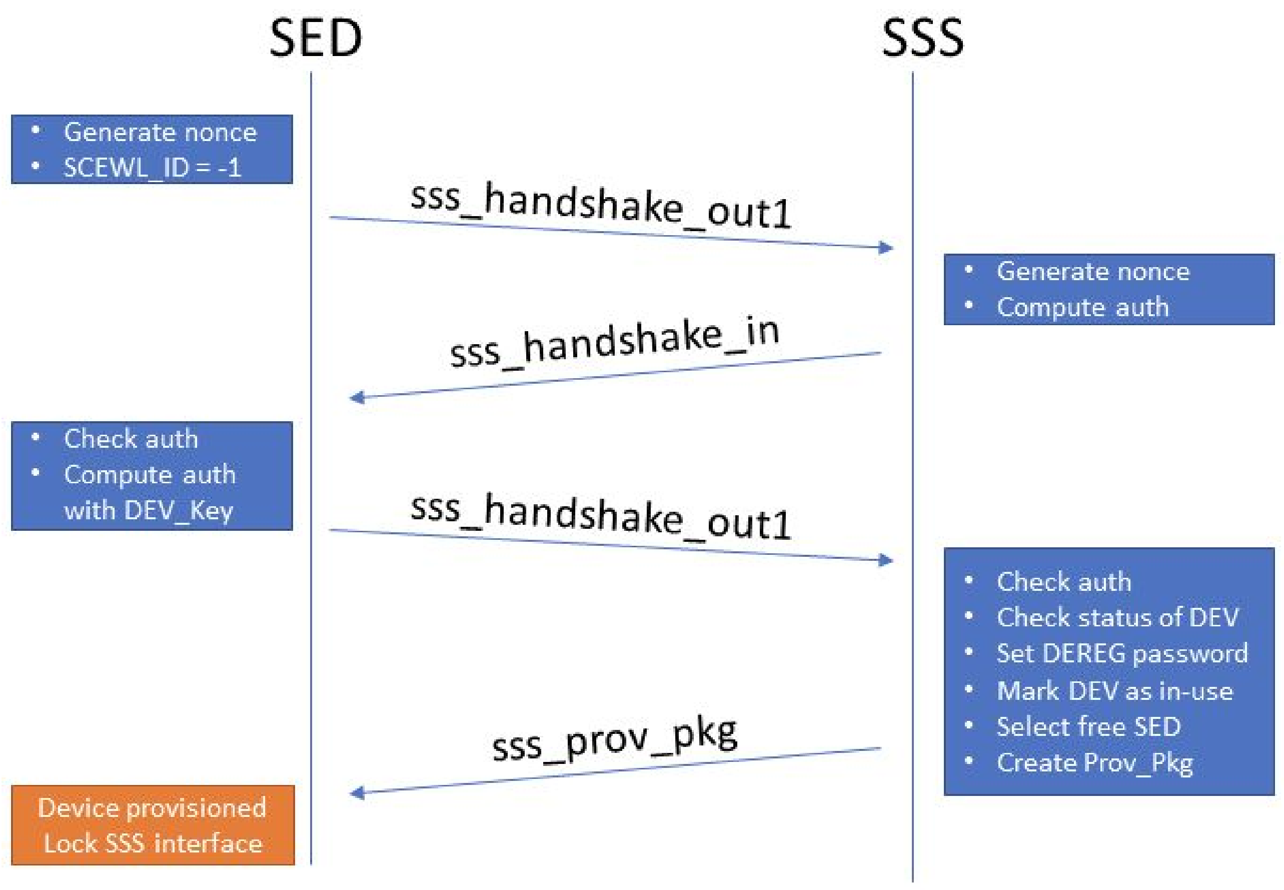
<u>sss_handshake_out2</u>

- nonce_y: random nonce provided by the SSS
- dev_id: Device ID generated at the factory (id between 1-256)
- auth: authentication message, blake2b(DEV_KEY, nonce_n, nonce_y)

<u>sss_prov_pkg</u>

- SED: ID (1-24)
- xchg_sec: Secret portion of the device encryption key
- sign_sec: Secret portion of the device signature key
- swarm_key: Secret for encryption of the header
- Dev_pub: Array with information about all 24 SED
    - Xchg_pub :Public portion of the device encryption key
    - sign_pub: Public portion of the device signature key
    - Scewl_id: id between 1-256
    - Counter: Last known counter value for the particular SED
    - Inst_counter: Last known instance for the particular SED
- runtime_entropy: initial entropy
- Dereg_password: password for the deregistration
- Unix_seconds: current unix time

Registration protocol flow



De-Registration of SED device (Scewl_deregister)

Used methods

- Blake2b hash function

Required data

- DEV ID: id between 1-256 identifying a device
- DEV_KEY: secret key for a specific DEV ID
- SED ID: assigned SCEWL ID to SED
- DEREG password: deregistration password for a deployed SCEWL ID

Structs

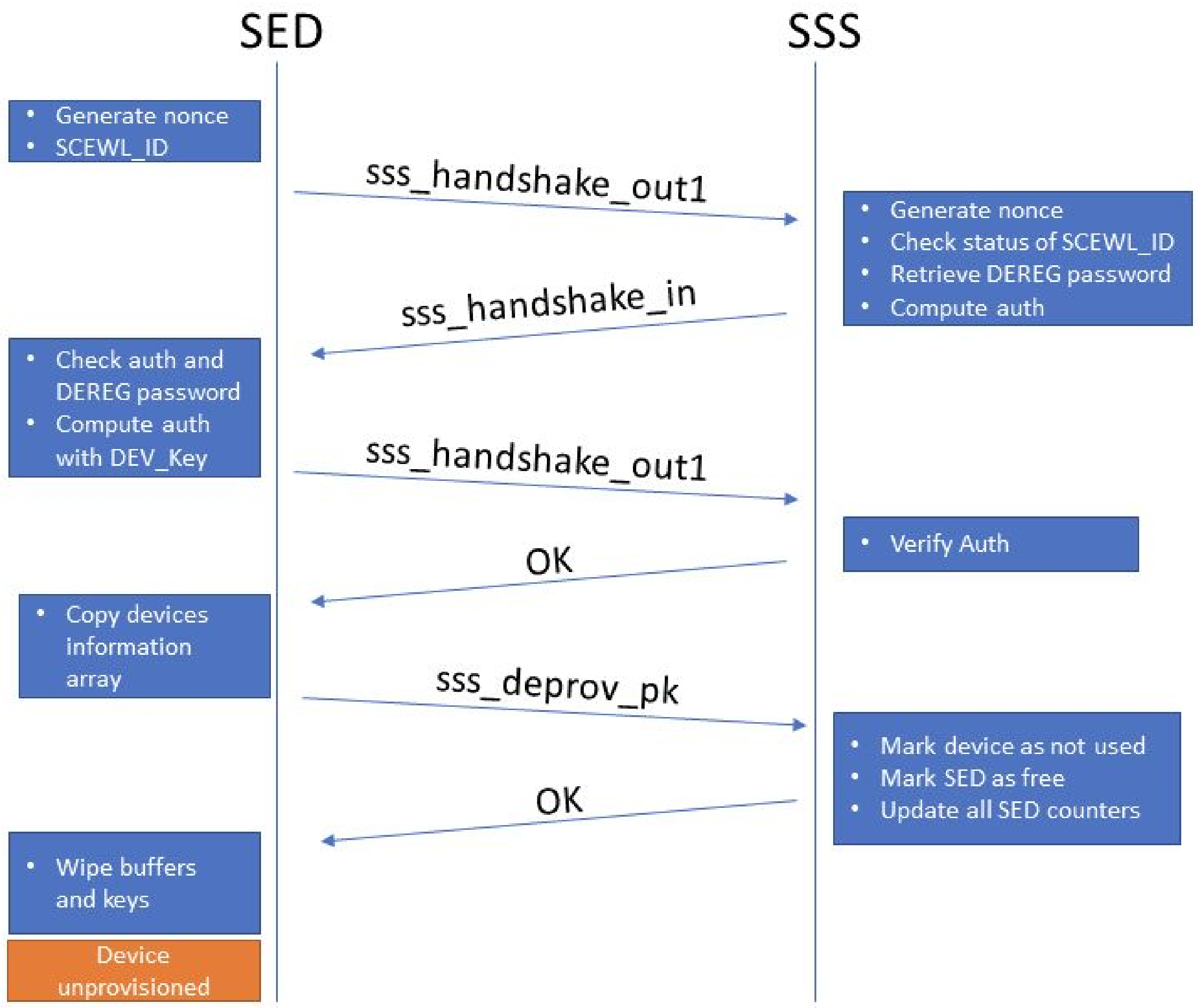
<u>sss_handshake_out1</u>
- nonce_n : random nonce provided by the unprovisioned device
- id: SCEWL ID

<u>sss_handshake_in</u>
- nonce_y : random nonce provided by the SSS
- auth : authentication message, blake2b(DEREG password, nonce_n, nonce_y)

<u>sss_handshake_out2</u>
- nonce_y: random nonce provided by the SSS
- dev_id: Device ID generated at the factory
- auth: authentication message, blake2b(DEV_KEY, nonce_n, nonce_y)

<u>Sss_deprov_pk</u>
- Dev_pub: Array with information about all 24 SED
    - Xchg_pub :Public portion of the device encryption key
    - sign_pub: Public portion of the device signature key
    - Scewl_id: (id between 1-256)
    - Counter: Last known counter value for the particular SED
    - Inst_counter: Last known instance for the particular SED

Registration protocol flow



Sending Direct Message (secure_send)

Direct messages are technically broadcasts, which only contain one valid header slot. SED devices that are not recipients have only random data in the header and wont be able to decrypt it. Other than that, there are no differences between direct and broadcasted messages.

Used methods

- X25519
- Ed25519
- XChaCha20

Required data

- SED ID: assigned ID to the target
- TGT SCEWL ID: (id between 1-256)
- TGT Instance counter: current instance of target
- Counter : current message counter

Structs

<u>Message keys</u>

- <u>Msg_key:</u> Random 32byte key for message encryption
- <u>Fake msg_key:</u> Random 32byte key used to fake key in headers

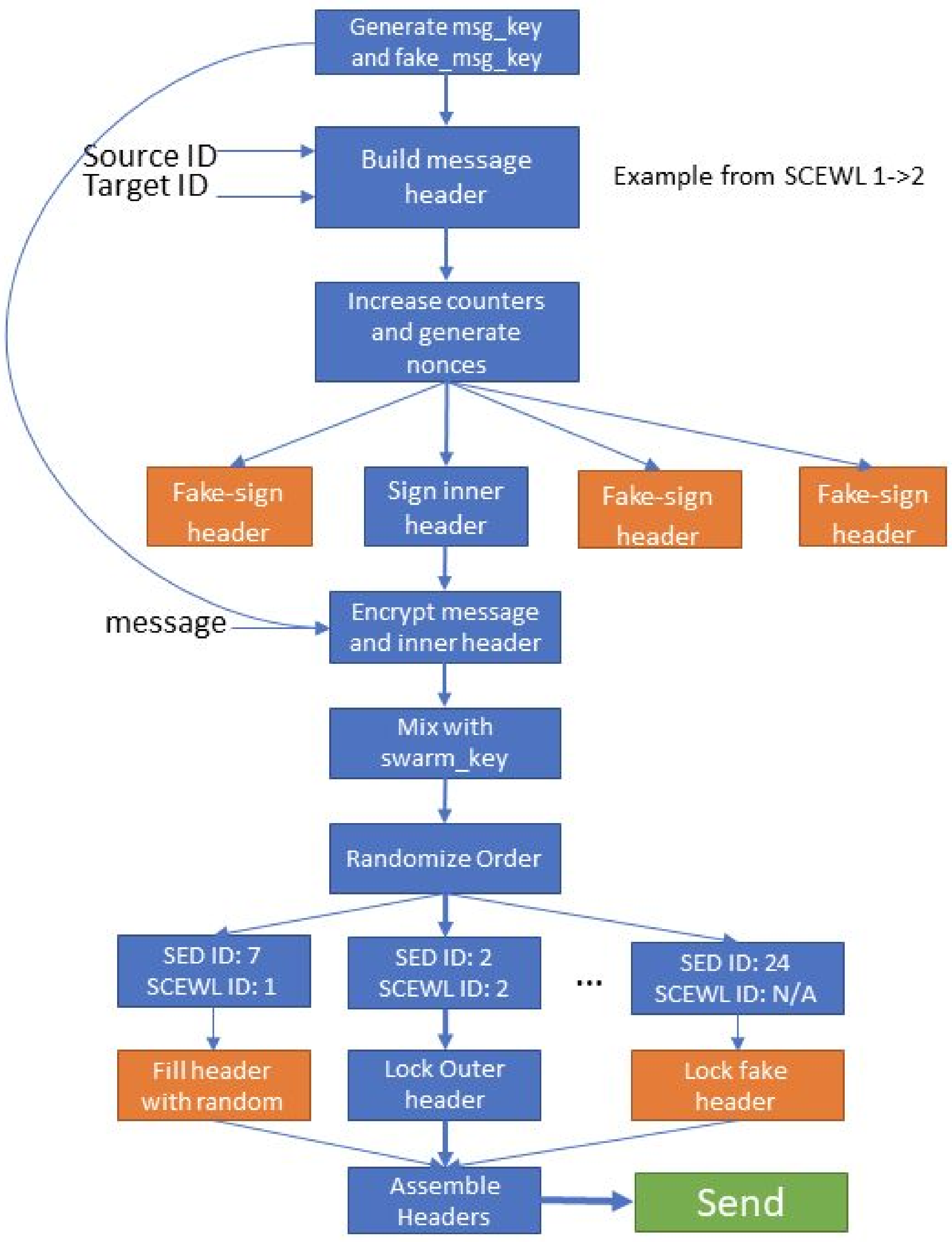
<u>Inner header (secure_header)</u>

- real_src: SCEWL ID of source
- real_tgt: SCEWL ID of target/ or broadcast
- sh_type: message type
- counter: message counter
- Src_inst_counter: instance counters
- Dst_inst_counter: instance counters

<u>Outer header (locked_msg)</u>

- Key_material: 24x encrypted headers per each SED in deployment
  - Nonce
  - Mac
  - message_key: message key
- nonce
- Mac
- Sig: signature
- secure_header
- Msg: encrypted message

Sending direct message protocol flow



Sending Broadcast Message (secure_send)

Broadcasts are packets where all SED headers are locked correctly. The individual SED can decrypt "their" header. Other than that, there are no differences between direct and broadcasted messages.

Used methods

- X25519
- Ed25519
- XChaCha20

Required data

- SED ID: assigned ID to the target
- TGT SCEWL ID: set to broadcast
- TGT Instance counter: current instance of target
- Counter : current message counter

Structs

Message keys

- Msg_key: Random 32byte key for message encryption
- Fake msg_key: Random 32byte key used to fake key in headers

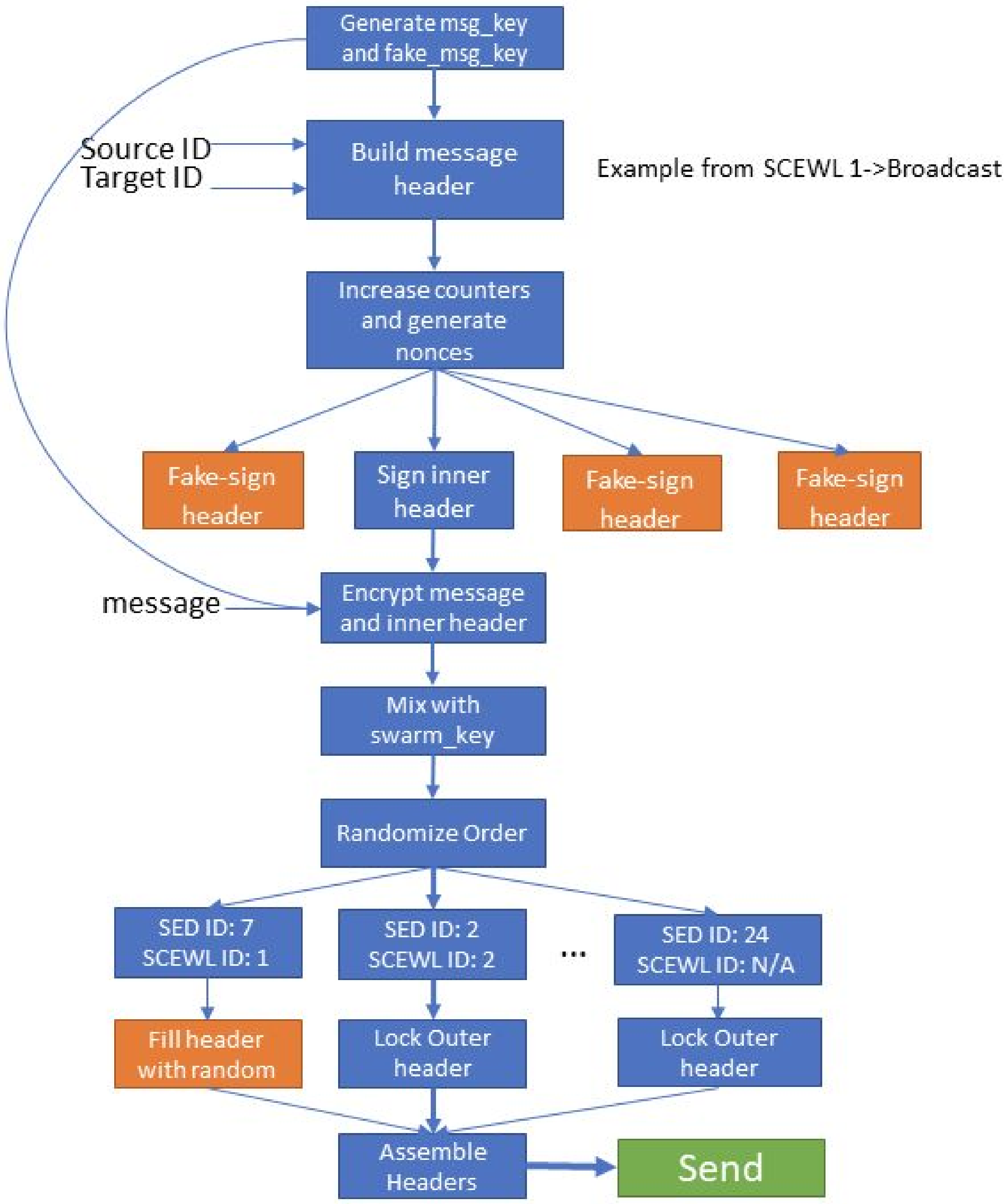
Inner header (secure_header)

- real_src: SCEWL ID of source
- real_tgt: SCEWL ID of target/ or broadcast
- sh_type: message type
- counter: message counter
- Src_inst_counter: instance counters
- Dst_inst_counter: instance counters

Outer header (locked_msg)

- Key_material: 24x encrypted headers per each SED in deployment
    - Nonce
    - Mac
    - message_key: message key
- nonce
- Mac
- Sig: signature
- secure_header
- Msg: encrypted message

Sending broadcast message protocol flow



Receiving Message (secure_recv)

From the perspective of the receiving device, there are no differences between direct and broadcasted messages.

Used methods

- X25519
- Ed25519
- XChaCha20

Structs

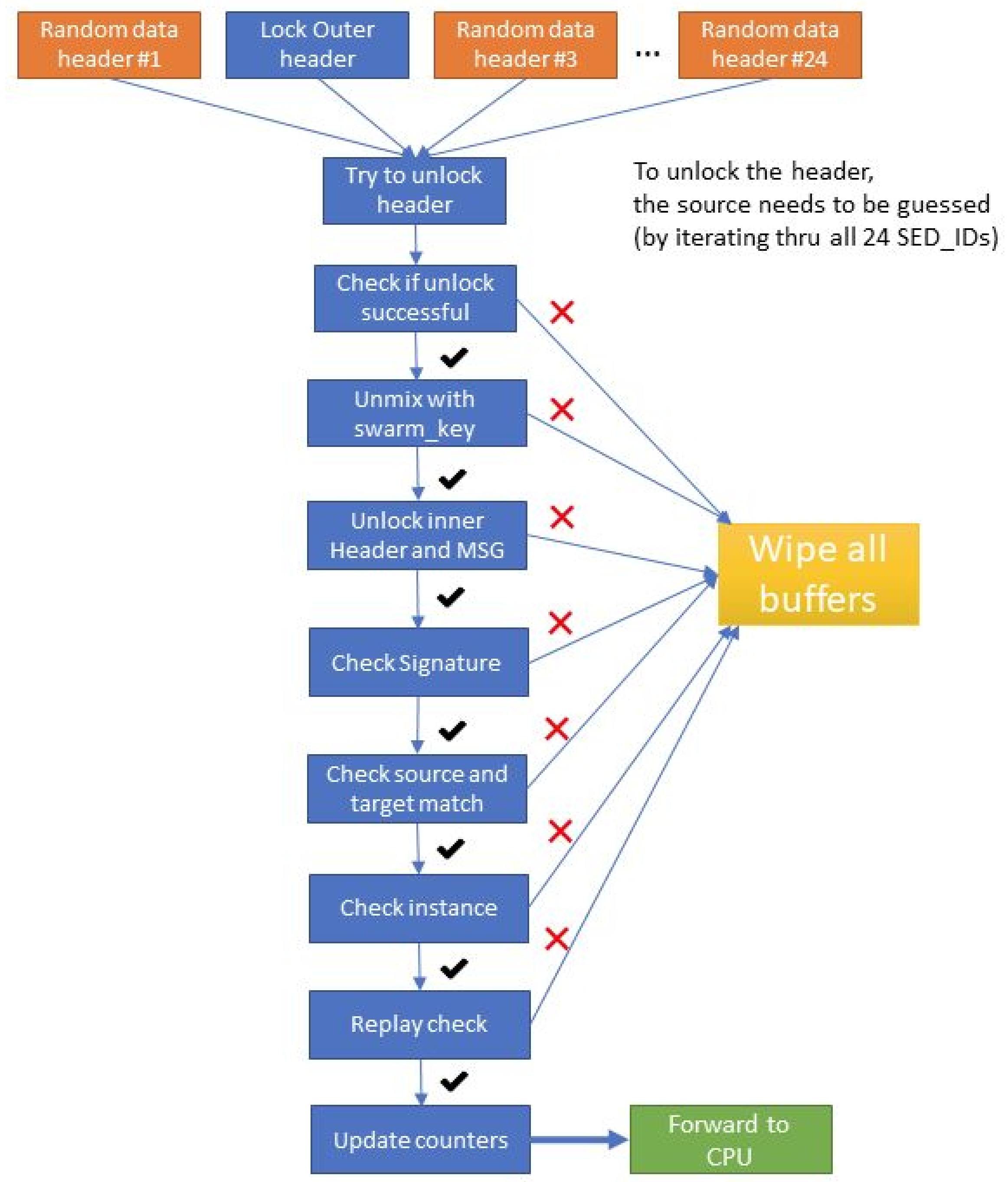
<u>Outer header (locked_msg)</u>

- Key_material: 24x encrypted headers per each SED in deployment
  - Nonce
  - Mac
  - message_key: message key
- nonce
- Mac
- Sig: signature
- secure_header
- Msg: encrypted message

<u>Inner header (secure_header)</u>

- real_src: SCEWL ID of source
- real_tgt: SCEWL ID of target/ or broadcast
- sh_type: message type
- counter: message counter
- Src_inst_counter: instance counters
- Dst_inst_counter: instance counters

Receiving message protocol flow



This Page Intentionally Left Blank