[logo]

NUAV@NUSec DESIGN DOCUMENTATION

Version 0.1
Draft

Cameron Kennedy
Dennis Giese
Erik Uhlmann

www.nusec.us

Advised by: Prof. Guevara Noubir
(Northeastern University)


German Engineering

Made in USA

Overview

System design idea

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

Protection of flags

In many places keys are encrypted and payload data is signed. The only place any of these layers of encryption are unwrapped are on the SCEWL controller. The structure of these keys can be found in the Data Structures section.

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 a 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 timer, which is synchronized among all devices. Messages are only accepted if they have a valid counter and the timestamp is in a specific window. For additional security, we conduct a 3-Way handshake for direct messages, where the payload will not be accepted until the confirmation of the presence of the sender.

Data Structures

Features

- **System supports up to 256 provisioned devices, with up to 16 of them being in use at the same time**

Implementation

Keys and Configuration

- Factory
    - Dev_credentials (**SCEWL_ID, Dev_Key**): unique device ID and secret string to serialize and identify devices
    - **SSS_Dev_DB:** list of all created devices with **SCEWL_ID** and **Dev_Key,** used by the SSS to authenticate devices
- SSS
    - **master_key:** symmetric DES key which is distributed to all registered devices and used for encryption of a packet header
    - SED_credentials (**SED_ID, SED_pubKey, SED_privKey**): asymmetric key material for registered devices, will be reassigned after de-registration
    - SED_DB (SED_credentials, counter, assigned **SCEWL_ID**, dereg_password, timeout): Pool of 24 generated SED_credentials (16 in use, 8 spare), with last know counter and information about the assigned device.
- Unregistered device
    - Dev_credentials(**SCEWL_ID, Dev_Key**): unique device ID and secret string to serialize and identify devices, programmed in the factory and stored in flash memory
- Registered device/SED
    - Dev_credentials(**SCEWL_ID, Dev_Key**): unique device ID and secret string to serialize and identify devices, programmed in the factory and stored in flash memory
    - **master_key:** symmetric DES key which is distributed to all registered devices and used for encryption of a packet header
    - SED_credentials (**SED_ID, SED_pubKey, SED_privKey, current_counter**): asymmetric key material for registered devices and current message counter
    - SED_pub_DB (**SED_ID, SED_pubKey, SCEWL_ID, counter):** Array of the public information of the SED_DB, provisioned by the SSS at registration. The

SCEWL_ID can be updated thru 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

Random notes

Provisioning procedures

Create_deployment:
- Generate Dev_Key (e.g. secret string) and Dev_ID for each device
- Store Dev_ID:Dev_Key in SSS_Dev_DB
- Generate symmetric key Master_key
- Generate 16 (or 24) Keypairs in SED_DB: SED_ID
- Assign counter value to each SED_ID
- Assign empty timeout for each SED_ID
- Format SED_DB: SED_ID, SED_pKey, SED_sKey, Dev_ID, counter, timeout

add_sed:
- Retrieve Dev_Credentials by Dev_ID from sss:/secrets/
- Mark Dev_Credentials as checked out
- Build controller containing
    - Dev_Credentials

remove_sed:
- Remove Dev_Credentials from sss:/secrets/

Runtime procedures

Scewl_init:

- Idk

Scewl_register:

- Challenge/Response between Dev and SSS
    - SSS<-Dev: chal(n), “-1”
    - SSS->Dev: y, hmac(n, y, “foo”)
    - SSS<-Dev: y, Dev_ID, hmac(n,y, Dev_Key)
- (SSS checks if device is already checked out and exists in DB)
- (SSS checks out a free SED_ID, and assigns Dev_ID+dereg_password to it)
- SSS->Dev: Provisioning package
    - [(SED_ID, SED_pKey, SED_sKey, counter), master_key, time, [array of keypairs-1(SED_pKey, SED_ID, counter)], dereg_password]
- Dev starts timer
- Dev locks SSS interface with dereg_password

scewl_send (for SED -> SED messages):

- Verify that the sender ID is matching what we have stored in the firmware. this is a mitigation against compromised CPU code
- Look up the target device in the table stored from registration
- Prepend an inner header to the message
    - Sending and target device ID (the real ones)
    - 32 bit replay counter
- Sign message with local signing key
- Perform key exchange using local key exchange key and the target device’s key (cache result so next time we have it already)
- Use the resulting key to perform authenticated encryption of the signed message. Increment the nonces to ensure they do not get reused -- eg use 20 bytes of entropy from SSS + 4 byte counter
- Send the message, consisting of bus header, nonce, mac, and encrypted(signed(inner-header || msg)). Use an underlying broadcast message every time to obfuscate the real destination of each message (unless it’s FAA channel)
- When idling, send random bytes that look like real encrypted messages. Profile the length of real messages sent and received to make it look convincing

scewl_brdcst (for SED -> SED messages):

- run scewl_send with every device in the table
- (could be slow, this may need to be revised)

scewl_recv (for SED -> SED messages):

- Use the sender ID to look up the public key exchange key and perform key exchange (with cache)
- Make sure the message is either directed to us, or a broadcast message
- With the resulting key, decrypt the encrypted payload. If monocypher signals an error, drop the message
- Verify the signature on the decrypted message. Failure -> drop
- Verify the src on the inner header matches the outer header. Verify the replay counter on the inner header and drop any replayed messages
- Forward the message to the CPU

Scewl_deregister:

- Challenge/Response between Dev and SSS
    1. SSS<-Dev: chal(n), Dev_ID
    2. SSS->Dev: y, hmac(n, y, dereg_password)
    3. SSS<-Dev: y, Dev_ID, hmac(n,y, Dev_Key)
- SSS<-Dev: [array of keypairs-1(SED_pKey, SED_ID, seen counters)]
- (Dev nukes keys, unlocks SSS interface)
- (SSS marks Dev_ID as deregistered)
- (SSS checks in SED_ID [clears Dev_ID], marks it as free, sets timeout for the key)
- (SSS updates SED_DB with counters)

Registration of device:

1. Challenge/Response between Dev and SSS
    1. SSS<-Dev: chal(n), “-1”
    2. SSS->Dev: y, hmac(n, y, “foo”)
    3. SSS<-Dev: y, Dev_ID, hmac(n,y, Dev_Key)
2. (SSS checks if device is already checked out and exists in DB)
3. (SSS checks out a free SED_ID, and assigns Dev_ID+dereg_password to it)
4. SSS->Dev: Provisioning package [(SED_ID, SED_pKey, SED_sKey, counter), master_key, time, [array of keypairs-1(SED_pKey, SED_ID, counter)], dereg_password]
5. Dev starts timer
6. Dev locks SSS interface with dereg_password

Deregistration of device:

1. Challenge/Response between Dev and SSS
    4. SSS<-Dev: chal(n), Dev_ID
    5. SSS->Dev: y, hmac(n, y, dereg_password)
    6. SSS<-Dev: y, Dev_ID, hmac(n,y, Dev_Key)
2. SSS<-Dev: [array of keypairs-1(SED_pKey, SED_ID, seen counters)]
3. (Dev nukes keys, unlocks SSS interface)
4. (SSS marks Dev_ID as deregistered)
5. (SSS checks in SED_ID [clears Dev_ID], marks it as free, sets timeout for the key)
6. (SSS updates SED_DB with counters)

Removal of SED:

1. Delete Dev_ID from SSS_Dev_DB
2. Remove assigned SED_ID entry (e.g regenerate key pair, as we have spares)

Direct Message (From A to B):

Preq: A knows the SED_ID of B from Broadcasts

1. Encryption of Payload with symmetric key payload_key + timestamp
2. Create header: [counter, timestamp, SED_ID[A], Dev_ID, len, message type, payload_key]
3. Enc+Sign header
4. Encrypt with master_key: [SED_ID[A], message type, encrypted inner header]
5. Send packet: [[encrypted external header][encrypted payload]]
6. B receives packet, decrypts external and internal header, stores payload in buffer
    1. Checks:
        1. Counter for SED_ID[A] not smaller than expected
        2. Timestamp inside the window (e.g. 60 sec)
        3. len
7. Handshake

. B sends ACK message to A (for freshness and replay protection, counter +1)

a. A sends ACK message to B (with updated counter and counter+1)

8. A+B update counter for each SED_IDs
9. B decrypts payload (AEAD) and passes it to the CPU

This Page Intentionally Left Blank