MIT TechSec eCTF 2025 Design Document

Summary

Our defenses consist of three main components:

- Unique encryption keys for each timestamp in a given channel
- Key derivation tree allowing subscriptions to provide just enough information to derive keys for the intended subscription window
- Memory protection, disabling execution from rewritable regions of memory

For the Decoder, we use wolfCrypt for its SHA-2, AES-128-GCM, and Ed25519 functionality.

For the Encoder, we use the Python standard library for SHA-2 and the cryptography module for its AES-128-GCM and Ed25519 functionality.

MIT TechSec Team

Students

- Amy Chen
- David Choi
- Owen Conoly
- Sophie D'Halleweyn
- Brendan Halstead
- Alex He
- Fisher Jepsen
- Marvin Mao
- Spencer Pogorzelski
- Shih-Yu Wang
- Erik Xie

Advisor

- Mengjia Yan

Security Requirements

Here, we briefly justify our adherence to the various security requirements.

Security Requirement 1

An attacker should not be able to decode TV frames without a Decoder that has a valid, active subscription to that channel.

Our subscription update files provide the set of information needed to derive keys for the timestamp window associated with that subscription, and nothing more.

Security Requirement 2

The Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for.

Subscription update files are only accepted by the Decoder if they are encrypted using a symmetric key unique to that Decoder. Decode and Subscribe packets are also required to be signed with the Encoder's private key.

Security Requirement 3

The Decoder should only decode frames with strictly monotonically increasing timestamps.

The timestamp of the last successfully decoded frame is stored in Decoder RAM. If the timestamp of a frame sent as part of the Decode command is not greater than this stored timestamp, we return an Error instead of attempting to decode.

Cryptosystem Overview

We use AES-128-GCM for encrypting frames and subscriptions, and additionally use Ed25519 for signing Decode and Subscribe packets. Unique keys are used for each timestamp, and subscriptions use a key unique to the intended device.

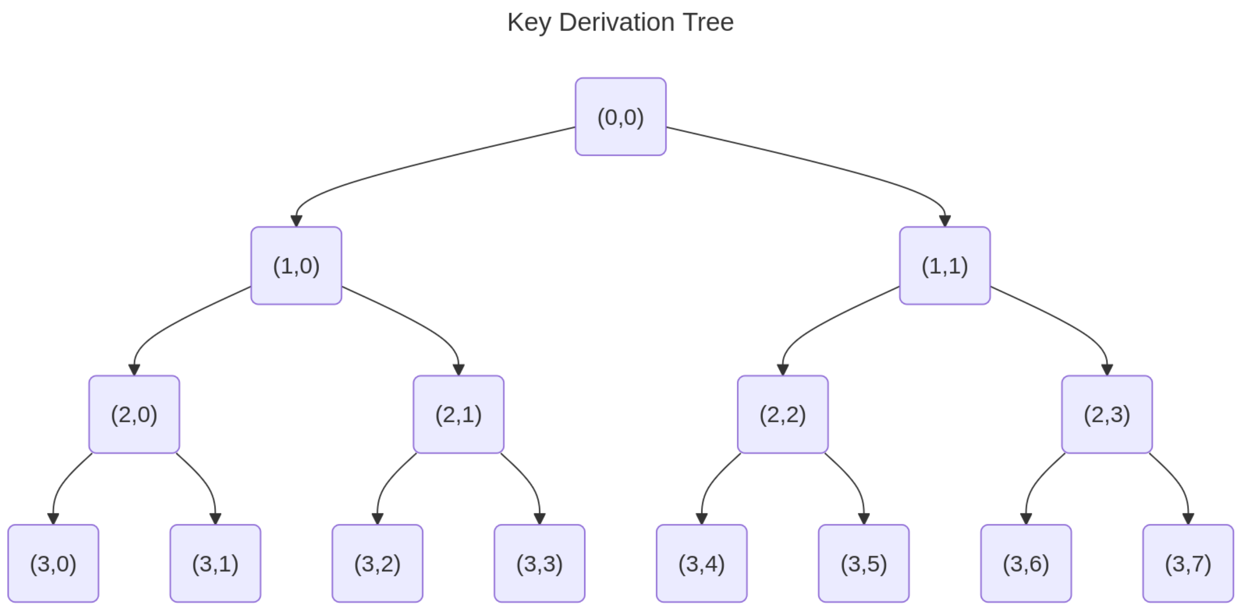
Definitions

- **H(m)**: SHA-2 digest of message *m*.
- **H_L(m)**: Upper 16 bytes of **H(m)**
- **H_R(m)**: Lower 16 bytes of **H(m)**
- **Root key**: Per-channel secret used to derive frame keys, **= randbytes(16)**
- **Frame key**: Per-channel per-timestamp frame encryption key, = **see details below**
- **Salt**: Per-deployment secret salt for deriving subscription keys, **= randbytes(16)**
- **Subscription key**: Per-device key for encrypting subscriptions, **= H_L(Salt || ID)**
- **Signing key**: Per-deployment Ed25519 priv. key for signing packets, **= randbytes(32)**

Frame Key Derivation Tree

Each channel of the satellite system has its own **root key**, which can be used to derive the unique **frame key** associated with a given timestamp for that channel. Frame keys are derived by descending our key derivation tree and applying **H_L** or **H_R** to the current node value if descending left or right in the tree respectively, until the desired leaf is reached.

With this scheme, Decoders can be provided subscriptions containing the minimal set of nodes needed to derive frame keys for their subscription window, *and nothing more*.



Above is the key derivation tree for a satellite system that only supports a 3 bit timestamp (and thus, only 8 potential timestamps). Each node is marked by a tuple of (Level, Index) and stores a 16-byte value. Nodes on the final level, i.e. a leaf, contain the **frame key** for the timestamp equal to that node's index.

Node (0, 0) contains the **root key**. The **frame key** for timestamp 0, stored in node (3, 0), is equal to **$H_L(H_L(H_L($root key$)))$**. The **frame key** for timestamp 1 is equal to **$H_R(H_L(H_L($root key$)))$.**

For a subscription valid for all timestamps, you can distribute just the node (0, 0). For a subscription valid just the last two timestamps 6 and 7, you can distribute node (2, 3).

Calculating the range of **frame keys** that a given node of level `l`, index `i`, in a tree of depth `d`, is able to derive is simple: $[t_{start}, t_{end}] = [i*2^{D-L}, (i+1)*2^{D-L}-1]$. This can be used to quickly determine whether to descend left or right to reach a desired **frame key**.

We argue the security of the frame key derivation by the assumption that SHA-2 is un-invertible, and that its output can be seen as random bytes. With this assumption, we need to know one of the ancestor nodes to know a certain frame key in a leaf node, and because the output of SHA-2 is random, we have no information about the right node if we have the left node's key, and vice versa.

Subscription Update Files

To ensure that one Decoder may not install updates intended for another Decoder, each genuine Decoder is provisioned with a unique **subscription key**. For a given decoder **ID**, the **subscription key** is derived at compile-time as $\mathbf{H_L}$**(Salt || ID).**

Knowledge of the subscription key for a given Decoder should not allow one to derive the subscription keys for other Decoders without inverting SHA-2.

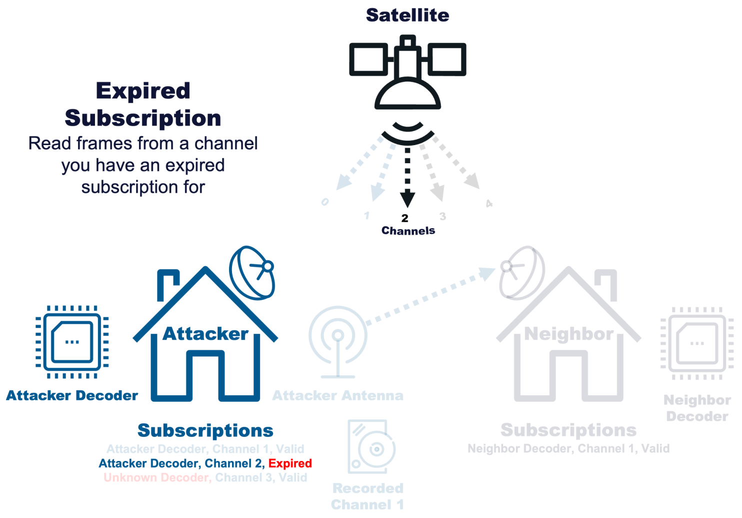
The encoder encrypts the subscription update file with the intended devices **subscription key** using AES-128-GCM. The packet is also signed with the Encoder's **signing key**. An invalid signature or failure to decrypt results in the subscription being rejected.

Dealing with Channel 0

Channel 0 has a unique requirement in that all Decoders, without any subscription update, must be able to decode frames sent on this channel. To satisfy this, Decoders are built with knowledge of the **root key** for Channel 0.

Attack Flags Security Analysis

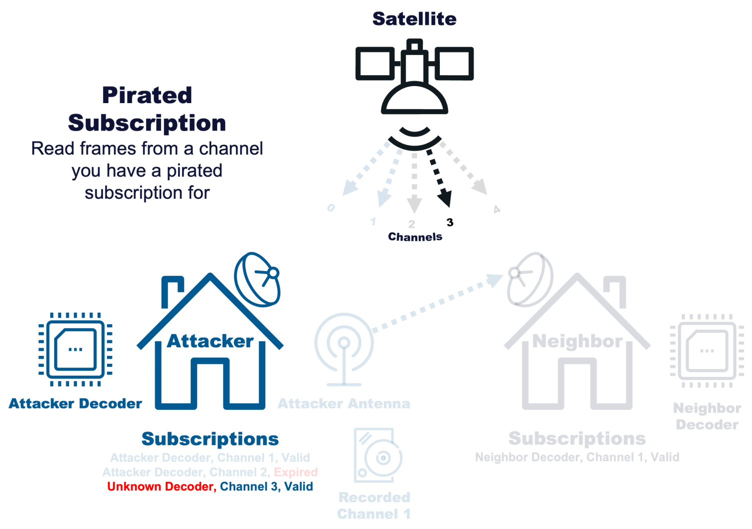
Flag 1: Expired Subscription


Satellite
Expired Subscription
Read frames from a channel you have an expired subscription for
0
1
2
Channels
3
4
Attacker Decoder
Attacker
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

Here, an attacker has a subscription update file meant for their device, for the timestamp range $[t_{start}, t_{end}]$. To get the flag, the attacker must decode frames with a timestamp $t > t_{end}$.

Thanks to our frame key derivation scheme, even with a full compromise of the Decoder, e.g. arbitrary code execution or dumping all flash or RAM, an attacker cannot produce the frame keys for timestamps $t > t_{end}$, and thus cannot decode the flag-containing frames.

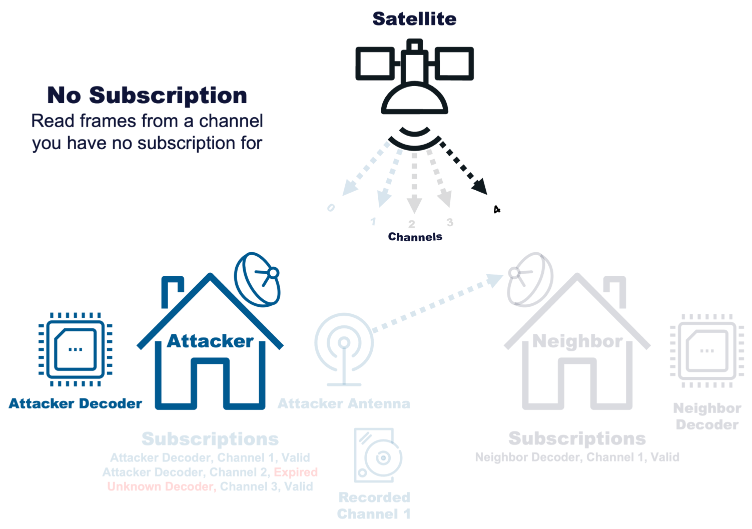
Flag 2: Pirated Subscription


Satellite
Pirated Subscription
Read frames from a channel you have a pirated subscription for
0
1
2
3
4
Channels
Attacker Decoder
Attacker
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

Here, an attacker has a subscription update file meant for *another* Decoder. To get the flag, the attacker must decode any frame on the channel this subscription is meant for.

Our subscription update files are encrypted using unique per-Decoder subscription keys. As the attacker does not have access to Decoder the subscription was made for, the attacker cannot obtain the needed subscription key directly through any hardware attacks or software attacks. If an attacker is able to leak a subscription key from another Decoder they have on hand, they would still need to invert SHA-2 in order to derive the subscription key needed for decoding frames on this channel.

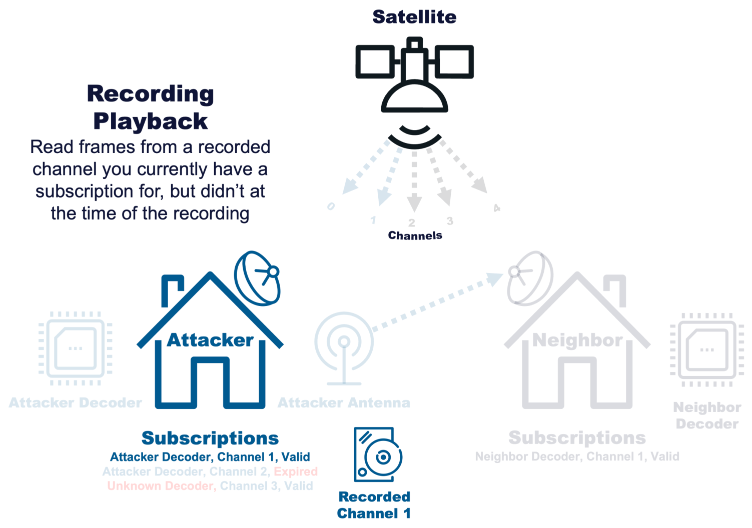
Flag 3: No Subscription


Satellite
No Subscription
Read frames from a channel you have no subscription for
0
1
2
3
4
Channels
Attacker
Attacker Decoder
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

Here, an attacker has a Decoder and nothing else. To get the flag, an attacker must decode any frame on a channel that they have no subscription for.

As channels have their own unique, randomly-generated root key, and as neither this root key nor any frame key derived from it is provided to an attacker, an attacker should not be able to decode any frame without breaking AES-128-GCM.

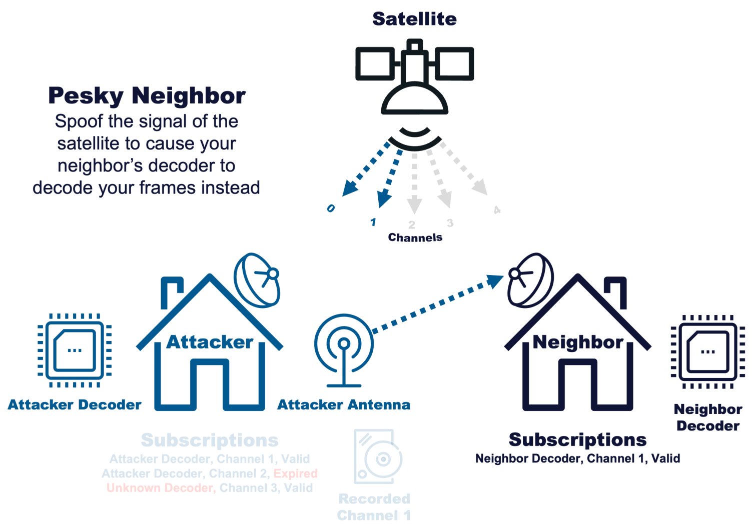
Flag 4: Recording Playback


Satellite
Recording Playback
Read frames from a recorded channel you currently have a subscription for, but didn't at the time of the recording
0
1
2
3
4
Channels
Attacker
Attacker Decoder
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

Here, an attacker has a subscription update file meant for their device, for the timestamp range [t_{start}, t_{end}]. To get the flag, the attacker must decode frames with a timestamp $t < t_{start}$.

Thanks to our frame key derivation scheme, even with a full compromise of the Decoder, e.g. arbitrary code execution or dumping all flash or RAM,, an attacker cannot produce the frame keys for timestamps $t < t_{start}$, and thus cannot decode the flag-containing frames.

Flag 5: Pesky Neighbor


Satellite
Pesky Neighbor
Spoof the signal of the satellite to cause your neighbor's decoder to decode your frames instead
0
1
2
3
4
Channels
Attacker
Attacker Decoder
Attacker Antenna
Neighbor
Neighbor Decoder
Subscriptions
Attacker Decoder, Channel 1, Valid
Attacker Decoder, Channel 2, Expired
Unknown Decoder, Channel 3, Valid
Recorded Channel 1
Subscriptions
Neighbor Decoder, Channel 1, Valid

Here, an attacker has a remote connection to a Decoder, the Neighbor, provisioned for our system. The Neighbor has a valid subscription to Channel 1. To get the flag, an attacker has to get the Neighbor to violate any of the security requirements.

For SR1, an attacker would have to get the Neighbor to decode a frame for a channel besides channels 0 and 1. An attacker is unable to generate their own rogue subscriptions for a device as, even if they could invert SHA-2 and derive the correct subscription key, they cannot produce the secret portion of the signing key.

For SR2, an attacker would have to be able to generate its own frames, for any channel, and get the Neighbor to decode one of those frames. Similar to the above, if an attacker is able to obtain the frame keys for Channel 1, they still cannot produce their own valid encoded frames without knowing the secret portion of the signing key.

For SR3, an attacker would have to get the Neighbor to decode frames out of order, i.e. decode frames without strictly monotonically increasing timestamps. Our enforcement mechanism for the increasing timestamps has been thoroughly tested.

Thus, to violate any security requirement, a software-based attack is required, and to frustrate this we use the ARM MPU to disable execution from RAM and make writing to flash more difficult.

Messaging Scheme

Packet structure

The general command packet structure is detailed below.

`Field name`	Length	Contents
Magic	**1**	**Message start byte, ‘%’**
Opcode	**1**	**Indicates the type of message (“D”, “S”, “L”, “A”, “E”, “G”)**
Length	**2**	**Length of the message body (variable)**
Body	**Variable**	**Opcode-dependent**

Response packets are dependent on the command.

List Command

Header

`Field name`	Length	Contents
Magic	**1**	**0x25 (“%”)**
Opcode	**1**	**0x4c (“L”)**
Length	**2**	**0x0000**

The list packet has no body. A list command with a non-zero length field will result in an Error message being sent back to the host.

List Response Body

`Field name`	Length	Contents
Num Channels	**4**	**Number of valid subscriptions**
Subscriptions	**Variable**	**An array of Subscription Entries of length `Num Channels`**

Subscription Entry

`Field name`	Length	Contents
Channel ID	**4**	**ID of the subscribed channel**
Start	**8**	**Start timestamp of subscription**
End	**8**	**End timestamp of subscription**

Subscribe Command

Header

Field name	Length	Contents
Magic	1	0x25 (“%”)
Opcode	1	0x4c (“S”)
Length	2	Variable

Body

Field name	Length	Contents
Nonce	12	Random nonce selected by Encoder
Auth Tag	16	AES-128-GCM authentication tag
Encrypted Update File	Variable	AES-128-GCM-encrypted Update File
Signature	64	Ed25519 Signature

- The authentication tag is computed over the Magic, Opcode, Length, Nonce fields.
- The signature is computed over the Magic, Opcode, Length, Nonce, Auth Tag, and Encrypted Update File fields.
- A failure to validate the signature or the authentication tag will result in an Error message being sent back to the host.
- A Subscribe packet with no body is returned on success, else an Error packet.

Update File

`Field name`	Length	Contents
Channel ID	4	Channel ID of the frame
Start	8	Start timestamp of subscription
End	8	End timestamp of subscription
Num Nodes	1	Number of nodes included in the update file
Nodes	Variable	An array of Nodes of length `Num Nodes`

Node

Field name	Length	Contents
Level	1	Level of the node
Index	8	Index of the node
Value	16	Value of the node

Decode Command

Header

Field name	Length	Contents
Magic	1	0x25 (“%”)
Opcode	1	0x44 (“D”)
Length	2	Variable

Body

Field name	Length	Contents
Channel	4	Channel ID of the frame
Timestamp	8	Timestamp of the frame
Nonce	12	Random nonce selected by Encoder
Auth Tag	16	AES-128-GCM authentication tag
Encrypted Frame	Variable	AES-128-GCM-encrypted Frame
Signature	64	Ed25519 Signature

- The authentication tag is computed over the Magic, Opcode, Length, Channel, Timestamp, Nonce fields.
- The signature is computed over the Magic, Opcode, Length, Channel, Timestamp, Nonce, Auth Tag, and Encrypted Frame fields.
- A failure to validate the signature or the authentication tag will result in an Error message being sent back to the host.

Decode Response Body

Field name	Length	Contents
Frame Data	Variable	Hopefully cute ASCII art