Final Design Document
MITRE eCTF

IITM

January 2025

Contents

1 Introduction

This year, our team has changed our approach to security as compared to that of our past years of competing. We employ various various hashing algorithms and cryptographic schemes to guarantee security and confidentiality. Crucially, we assume that an attacker has **complete access** to the flash memory of the decoder in our security model.

2 Build

We have written our codebase in Rust, due to the additional security guarantees that it provides. We have used the Hardware Abstraction Library developed by UIUC, in order to interface with the board.

We have modified the Docker image, to allow it to generate Rust binaries. We use a `build.rs` to generate a `secrets.rs` file, which is used to import global secrets into our Rust code.

2.1 Build Instructions

The building process is identical to that of the reference design.

1. Building the docker image `docker build -t build-decoder ./decoder`
2. Building the decoder

```
# Build a Decoder in a Docker container:
#   '--rm' Delete the container when finished (remove when
#          debugging to see output after completion)
#   '-v ./decoder/:/decoder' Mount the local directory
#          'decoder/' to '/decoder' in the container
#   '-v ./global.secrets:/global.secrets' Mount the file
#          'global.secrets' to '/global.secrets' in the
#          container
#   '-v ./deadbeef_build:/out' Mount the local directory
#          'deadbeef.build/' to '/out' in the container
#   '-e DECODER_ID=0xdeadbeef' Define the Decoder ID of the
#          Decoder
#   'build-decoder' Specify the tag of the Docker image to use

docker run --rm -v ./decoder/:/decoder              \
           -v ./global.secrets:/global.secrets:ro   \
           -v ./deadbeef_build:/out                 \
           -e DECODER_ID=0xdeadbeef build-decoder
```

2.2 Global secrets

For each channel (numbered i) in our deployment, we generate a **master key** (denoted as K_i) for that channel. We also generate a **subscription key** K_S, common to all channels. These keys are **known only to the encoder**. This is crucial in ensuring that a completely compromised decoder would still not violate our security

requirements. These keys are not used directly, and are instead used to derive further keys, using secure hashing algorithms. Further, we also generate a public/private key pair for digitally signing the encoded frames. The private key is known only to the encoder, while the public key is stored in the decoder.

2.3 Local secrets

For a decoder numbered d, we store the decoder-specific key K_d in the decoder. This key is generated from the subscription key K_S, as follows:

$$K_d \equiv \mathsf{Hash}(K_S \| d) \tag{1}$$

Since decoder d only knows its own key K_d and not K_S, we can be guaranteed that a decoder cannot derive the key $K_{d'}$ of another decoder d' – because doing this would require breaking the Hash function. We choose an appropriate Hash, which is not malleable for fixed-length inputs. Additionally, the encoder, which knows the value of K_S, can trivially derive any decoder key.

3 Functional Requirements

3.1 Encoder

As required by the competition rules, the encoder is a `pip`-installable Python module, that takes a frame, timestamps, channel ID, and global secrets as input, and outputs an encrypted frame. Each encrypted frame is broadcast to all decoders.

We use a novel segment-tree-based cryptographic key-derivation scheme based on one-way functions, in order to encrypt our frames. Each frame is encrypted using a timestamp-dependent **frame key**, which is derived from the master key for the frame's channel. This scheme is described in detail in subsection 5.2. Our scheme allows for efficiently communicating a large range of frame keys to the decoder, by repeatedly deriving keys in a tree structure.

The structure of packets created by our encoder, is outlined in subsection 5.3. We use a public key signature scheme in order to ensure the integrity of frames, making sure to include all the relevant details of a frame in the signature.

3.2 Decoder

Our decoder is programmed using Rust. It decodes the subscription binary – a binary containing a valid subscription to a channel, including a start and end timestamp, and the channel number.

Subscriptions are encrypted and authenticated using keys derived from the decoder key K_d, which is generated separately for each decoder, using the subscription key K_S. (See subsection 2.3.)

Frames are decrypted by the decoder, using frame keys. Frame keys are timestamp-dependent. Keys for the subscribed duration are contained in `subscription.bin`.

4 Security Requirements

4.1 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.

4.1.1 Validity

This is achieved by encrypting and digitally signing the `subscription.bin` file. Encryption ensures the data remains confidential, and the digital signature verifies the authenticity and integrity of the subscription.

4.1.2 Subscription status

The keys required to decode specific frames are derived using our segment-tree based key derivation algorithm. Only the frame keys for the active subscription range are provided in the subscription file, and nothing more. This approach ensures that frames outside the subscription period cannot be decoded, as our scheme uses one-way functions to derive frame keys. Detailed implementation is provided in subsection 5.2.

4.1.3 Correct channel

The subscription file contains the channel ID, which the decoder uses to verify access permissions before allowing frame decoding. Encryption and signature protect the integrity and authenticity of this information.

Furthermore, the keys provided in the subscription file need to be derived from the master key of the correct channel, in order for decryption of frame data to work correctly in the first place. If keys for the wrong channel are used, we receive garbage data on decryption.

4.2 Security Requirement 2

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

To validate the authenticity of TV frames, a digital signature is appended to each packet. This mechanism is further explained in the *Signature section in Packet Structure*.

4.3 Security Requirement 3

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

The decoder maintains a counter, to ensure that the timestamps of incoming frames are strictly monotonically increasing, thereby preserving the correct sequence of decoded frames.

This is the only security requirement in our design which is not cryptographically guaranteed. In order to achieve this, we apply appropriate measures to protect against fault injection attacks.

4.4 Additional Security Checks

The decoder sets a *safe bit* in the flash memory whenever it detects unexpected behavior, and resets the decoder. Whenever the decoder boots, if the safe bit is set, it puts itself in a lockdown time loop. This is intended to prevent brute-force attacks.

5 Cryptography

5.1 Encryption schemes used

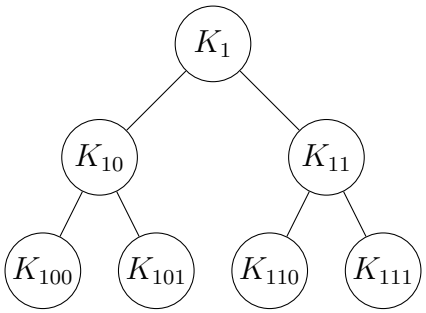
We use AES-256 in the CBC mode as our block cipher algorithm and ECDSA for our signature scheme.

5.2 Segment-tree key-derivation

We have come up with a novel key-derivation scheme in order to help us achieve the security requirements for this competition, which we describe briefly here. A longer discussion can be found here.

The TV frame broadcast can be thought of as a very long string of frames S, which we wish to encrypt. We then wish to generate a "subscription" for a substring of S, which is a piece of information D, which allows a recipient to decrypt that substring of S.

We achieve this by creating a complete binary tree, for each channel, as follows:



For each channel, a tree is created. The root node is set to the root key for the channel. For every node, the child nodes are derived using a one-way hash function, using different salts for the left hand and right hand derivation. For instance, the node K_{10} is derived from the K_1 by

$$H(K_1 \| L),$$

and the node K_{11} is derived from the same key as

$$H(K_0 \| R)$$

.

where L and R are two different publicly available strings, used as salts. We have employed SHA3-256 as the hashing function H.

Now, the leaf nodes of this tree are matched one-to-one with the frames of the broadcast. We use a tree of depth 64, so that we have 2^{64} leaf nodes. Thus, each leaf corresponds to one unique timestamp.

Now, to send a subscription for timestamps from i to j, we select a "cover" for the corresponding leaf nodes from i to j. The "cover" is a set of nodes, such that the leaves from i to j fall inside the subtrees of the nodes chosen, while leaves outside the range from i to j do not fall within the subtrees. E.g., if the cover contains nodes a, b, c, whose subtrees are T_a, T_b, T_c, then for all leaves from i to j, the leaf falls in $T_a \cup T_b \cup T_c$, but every other leaf does not. The process of finding such a cover is identical to that of point-updating elements in a Segment Tree, and it is possible to choose a set of at most $2 \cdot \log(j - i + 1)$ nodes for a given interval.

Once given a cover, the frame keys, corresponding to the leaf nodes can be derived using iterated hashing, from the appropriate node, whose subtree that the leaf falls under. A more detailed explanation can be found in the accompanying document.

5.3 Packet Structure

Every frame packet is comprised of 5 parts as listed below.

1. **Timestamp:** This is used to ensure only monotonically increasing frames are decoded. We also utilize the timestamp in the derivation of the decryption keys for the current frame, as described in the segment tree. This is sent in plain-text.
2. **Channel ID:** The ID of the channel on which the frame is to be broadcasted. This is sent in plain-text.
3. **IV:** Initialization Vector, to be used by AES for decryption.
4. **Data:** Frame data, fully encrypted with AES.
5. **HMAC:** Hash of the generated packet, used to ensure packets received by the decoder are authentic.
6. **Signature:** The channel ID, timestamp and encrypted data are signed using a public key signature scheme. This is verified at the receiver.

6 Additional security measures

6.1 Brute-force protection

On encountering suspicious behaviour, the board enters a lockdown state. This is achieved by writing a "safe bit" to the flash memory, which is read on boot. If this bit is set, the board sleeps for some time before the flash is cleared, and operations are resumed. This helps prevent bruteforce attacks.

6.2 Fault-injection protection

Before critical sections of the code are executed, the board performs random operations for a random amount of time. These random numbers are seeded using the on-board TRNG. This makes it very difficult to time fault injection attacks on the board.

6.3 Timing attack protection

We use constant-time comparison functions for verifying HMACs on the board. This prevents timing attacks on bypassing HMACs.

6.4 Side-channel analysis protection

Random values can never be decrypted by the board, as we always use authentication and encryption together. HMACs need to be bypassed before decryption is done. Thus, CPA attacks on AES cannot be performed. Further, we use SHA-3 as our hashing scheme, which is resistant to side-channel attacks as of date.

Team Members

1. Arivoli Ramamoorthy (EE23B008)
2. Abhinav I.S. (EE23B002)
3. Sanjeev Subrahmaniyan (EE23B102)
4. Madhav Tadepalli (EE23B040)
5. Nithin Ken Maran (EE23B050)
6. Kevin Kinsey (EP23B027)
7. Athish Pranav D. (CS21B011)
8. Md. Isfarul Haque (CS22B010)
9. Mohitabinav M. (ME21B117)
10. Nitin G. (EE22BB041)