Defense Writeup

One feature that we included in our design was key rotation, utilizing a different key for each transmitted message. We adopted this as a design goal (as opposed to using fixed keys) for two main reasons. First, a properly designed key rotation mechanism provides forward secrecy and future secrecy, ensuring that the compromise of one key does not result in the compromise of past messages or future messages. Secondly, frequent key changes provide protection against side-channel attacks, which record a number of operations using a key (e.g. timing info or power usage), and use those traces to extract the key.

Our implementation of key rotation used HKDF (Hash-based Key Derivation Function), which uses HMAC internally, which in turn uses a one-way hash function internally (for which we selected SHA256). Unlike other ad-hoc key rotation mechanisms, HKDF was designed to require fewer assumptions about the properties of the underlying hash function. (While HKDF uses an extract-then-expand approach, we assumed that the random secret provided by the SSS had enough entropy to render the extraction phase unnecessary.) The next key would then be computed as an HMAC keyed with the SSS secret over the concatenation of the previous key, an info field, and the sequence number of the message. The utilization of an additional secret in the key derivation process provides future secrecy, while a naive approach such as hashing the old key to get a new key only provides forward secrecy. The sequence number helps provide a guarantee that an input changes for every key rotation, and the info field provides context separation to derive different keys from the same initial input.

In order to implement the scheme, we had to decide on an appropriate value for the information field. In doing this, we encountered limitations with the amount of memory available on the SED controller. For the info field, we chose to use the target ID so that communications with each SED would have different keys. Because we had to store the previous key in memory, we had to dedicate a large amount of memory to an array that would store a key for each other SED that may be communicated with. This led us not to include the source ID, as that would be a quadratically growing memory cost of previous keys to keep track of, most of which would not even be active at once (because there could only be up to 16 SEDs active at a time, even though up to 256 SEDs would be ready to deploy at any time).

However, our choice of using only the target ID had the problem (which we did not realize until after we had moved into the attack phase) that all SEDs would derive the exact same sequence of broadcast keys, since broadcast messages utilize a special, constant target ID. We speculate that our design may have been broken this way, through a broadcast key recovered from one SED (e.g. through side-channel analysis) that was then used to decrypt other broadcast messages with the same sequence number or to forge broadcast messages to other SEDs that did not receive the original broadcast.

An alternative key derivation scheme we could have used would involve adding the source ID to the info field. With the assumption of a strong hash function, we would have the same guarantees of side-channel resistance, as well as forward and future secrecy, without the weakness of different SEDs deriving the same broadcast keys independently. The original paper

defining HKDF chose to use a feedback mode in order to ensure that successive hash function inputs were vastly different, as a heuristic protection against potential hash function weaknesses. However, as an attacker would have to capture an extremely large number of keys in order to mount a cryptanalytic attack on the key derivation function, this is a risk that may be acceptable. If we assume that the SHA256 hash function underlying HMAC is sufficiently strong, we can avoid the memory requirements of using a feedback mode by removing the previous key from the key derivation function call.

Attack Writeup

For a messaging protocol to be truly secure, it must provide both confidentiality and integrity to messages, which includes not only the message content itself but also all other associated metadata. We describe related tampering attacks and replay attacks that worked on multiple designs due to insufficiencies in their integrity checks.

One integrity failure resulted from a message MAC that could be recomputed by external parties. These designs used an unkeyed hash over the final encrypted message. We corrupted the message by overwriting bytes 32-35 with the value 1 (with the offset used to skip past an added header) and recomputed the attached hash, triggering recovery-mode in any of the SEDs receiving the tampered messages.

A different team used a custom home-made hash function to provide integrity. They hashed the unencrypted message instead of the encrypted message, which could have provided sufficient integrity protection given a strong hash function. However, their hash function had significant weaknesses, including an insufficient length and a multiplication-by-input step that would zero out the entire hash value if the message ended with enough null bytes. This was routinely triggered by the messages being transmitted, as they were zero-padded to a multiple of the block length before being encrypted. To corrupt the messages, we zeroed out the first 4 bytes of the message, which did not affect the hash due to the zero-padding weakness, triggering recovery-mode in SEDs receiving the tampered messages.

A different type of integrity failure resulted from a failure to protect message metadata with the message hash, which allowed us to use a replay attack to obtain no-fly-zone flags. The first time a SED received a message from a new SED, it would unconditionally accept the message and record the contained sequence number for replay prevention. When a deconflict message was sent, we could capture it and replay it to the same target, swapping out source IDs for ones it hadn’t seen before. Because the source ID was not protected by the hash, no integrity violation was detected by message processing. We sent enough messages to cause the SED to fly above the altitude ceiling, triggering the no-fly-zone behavior.

For all of the above flaws, there is a simple fix. Using a unified AEAD (authenticated encryption with associated data) algorithm such as AES-GCM avoids issues with weak hash functions and guarantees integrity of both the encrypted data and the associated data. Including the message headers in the associated data ensures that they cannot be tampered with.