Part 1: Security Feature

The secure SCEWL system was required to protect messages being sent between devices in both unicast and broadcast styles. To accomplish this, encryption keys are necessary for enforcing the confidentiality, integrity, and authenticity properties of the messages. Confidentiality indicates that the message contents cannot be viewed/retrieved by an adversary that does not have the encryption key; integrity indicates that the message contents cannot be modified by an attacker and still be valid during decryption; authenticity indicates that the message was sent from the person who *claims* they sent it. Since up to 16 devices need to be running and able to talk to each other at any time, and since new devices can join and leave the network at any time, our system utilized a key distribution scheme. This scheme allowed all devices to communicate securely with the properties above, even when all the devices may not know of each others' existence at start up.

The scheme uses a combination of symmetric and public keys to achieve secure communication. A symmetric key is used for both encryption and decryption, with the same general algorithm used for both functions. A public key scheme assigns a private/public key pair to each participant, where the public key is made available to everyone else, and the private key is only known by the participant. Data encryption is performed by encrypting the message with the **receiver's** public key, so that only the receiver is able to decrypt the message with their **private key**. Additionally, the sender can **sign** the message using their own **private** key since only they can perform that operation. The receiver can then **verify** the signature using the sender's **public key**.

Our system uses the SCEWL Security Server (SSS), which is a root of trust in our design, to create a public/private key pair for all devices joining the network. As an extra step, the SSS signs a certificate using a combination of the new device's keys and the SSS's own keys. When this new device (device A), communicates with another device (device B), device B asks for the certificate and public key of device A. Device B then verifies that the certificate has in fact been authorized by the SSS, and was made specifically for the keys that were assigned to device A. Device B can verify the SSS signature on the certificate because all of the devices know the SSS public key.

The certificate is necessary because it is impossible for the SSS to predict which devices will run on the network in order to tell a new device the public keys of all other devices. The certificate is critical for device B to receive a message from device A, when device A may have joined later than device B. With the SSS signature on the device A certificate, device B is able to verify that device A is in fact a legitimate device that was created by the SSS. The full cryptographic protocol that uses these keys for communication is called Datagram Transport-Layer Security (DTLS); for more information refer to [1].

For the broadcast protocol, it would be unfeasible for one device to send a unicast message to every device because public key cryptography can take non-trivial amounts of time. Therefore, a **symmetric key** algorithm is used for broadcasting, where every device that joins the network is given the broadcast key. Since all devices have the key, they can all decrypt the same message. To prevent attackers from collecting significant amounts of ciphertext from the same key to perform cryptanalytic attacks, the broadcast key is refreshed using a pseudo-random function (PRF). The broadcast protocol uses a modified version of the Secure Real-time Transport Protocol (SRTP), but preserves the key derivation PRF defined in [2].

One aspect of the key scheme that could be improved is to calculate, and not approximate, the appropriate broadcast key update rate to not only guarantee protection against cryptanalysis, but also against side-channel attacks. For this challenge, the key was refreshed every 17 frames (a max of 17KB).

References:

[1]https://www.pixelstech.net/article/1459585203-Introduction-to-DTLS%28Datagram-Transport-Layer-Security%29 ; DTLS overview
[2] https://tools.ietf.org/html/rfc3711#appendix-B.3 ; SRTP key derivation function

Part 2: Attack Analysis

Hash: the output of a hash function.
Hash Function: a function that maps arbitrary length inputs onto fixed length outputs. Ideally this mapping would be random, so that similar inputs do not create similar outputs. They can be used to ensure data integrity by checking the hash of received data against a hash that is known to be correct.
Collision: A collision occurs when two distinct inputs to a hash function result in the same hash. A good hash function should make predicting two inputs that collide prohibitively difficult.

We unsuccessfully attacked another team's implementation by exploiting a flaw in their protection against packet modification. Their packets contain an encrypted data segment followed by a hash of the unencrypted data. This is not an ideal arrangement, since the unprotected hash technically leaks information about the unencrypted data to the attacker. This would be merely a theoretical concern if this team had used a standard hash function, but they decided to invent their own. The following is a python implementation of their hash:

```
def get_hash(data):
  h = 0
  for i in range(len(data)//4):
     h = h + data[i*4 + 0]
     h = h ^ data[i*4 + 1]
     h = h - data[i*4 + 2]
     h = h * data[i*4 + 3]
     h = int(format(h, 'b')[-32:], 2) #field is 32 bits as implemented
  return h
```

This hash function has a number of non-ideal properties, including that the first three steps only directly touch the final quarter of the calculated hash, and changing later bits in your data causes fewer differences in the final hash than earlier bits. However, these could be true in a hash that is still useful for our purposes. However, there is one glaring oversight in this design: The final step is multiplying the calculated value by a byte pulled from the data, meaning *the hash output will always be zero if the final byte of data is zero.* This leads to a 1/256 chance of collision on random input data. The actual input data is non-random, but the team's implementation appears to zero-pad data at the end to ensure the length is a multiple of 4 bytes (so the hash function will work correctly) which makes the collision more likely.

The actual attack would work as follows: find a packet with a hash field of zero, and change random bits in the payload. Eventually, decrypting that payload will result in corrupted data that ends with 0x00, which will pass the hash comparison and be treated as valid data. This corrupted data would be passed on to the CPU where it would fail the checksum, triggering the "Recovery Mode" flag. We were unsuccessful in capturing this flag because we ran out of time.

One solution the team could implement to avoid this issue is using a pre-existing hash function that does not have these design flaws.