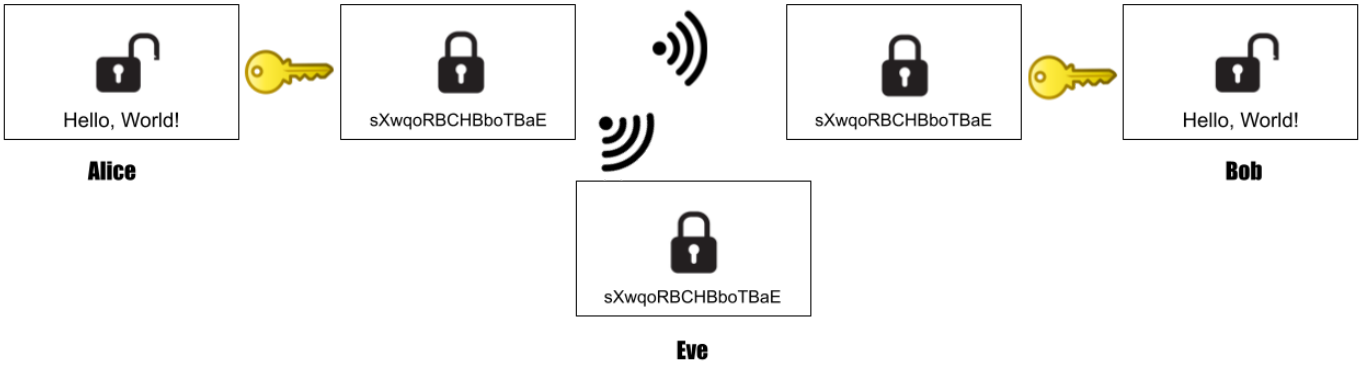
Defensive Write-Up: Hybrid Encryption with Elliptic Curve Cryptography

One important requirement for secure communications is message *confidentiality*, which ensures that only the intended target may read the message contents. When the threat of interception is present, *encryption* must be used to protect message confidentiality as packets are sent over the radio.

The most efficient form of encryption for large chunks of data is *symmetric* encryption, in which a single pre-shared key is used both for encryption and decryption. Like a physical lock on a package, the sender may lock the message with the secret key and send it over an untrusted medium. The recipient may unlock the message with a copy of the same key. Unless attackers discover the key, they will be unable to snoop on message contents.



Some teams decided to use a single deployment-wide encryption key for all messages. This is insecure because it creates many paths for attackers to discover the key. Instead, we decided to randomly generate a unique key for every single message, which improved our security. However, this also presented a problem: how would devices securely share this symmetric key without attackers discovering it?

This is where asymmetric cryptography comes in. Instead of using the same key for locking and unlocking, each party has a public and a private key. All devices are given all the public keys ahead of time by a “certificate authority”; in this case, the SCEWL Security Server. The private keys are kept secret, held by only the owning device.

When a device needs to send a message, it derives a shared secret from its own private key and the target’s public key. The target comes to this same value by combining *its* private key and the sender’s public key, hence the phrase “shared secret”. Using a method known as Elliptic Curve Diffie-Hellman, the definition of public keys implies that this value matches:

$$A_{priv} * B_{pub} = A_{priv} * B_{priv} * G = B_{priv} * A_{priv} * G = B_{priv} * A_{pub}$$

Once the shared secret is established, it is used to encrypt the symmetric key for each message, ensuring that only the intended recipient may decrypt the key and recover the text. This combination of symmetric and asymmetric cryptography is known as “hybrid” encryption.

During the attack phase, no teams successfully captured the “Package Recovery” flag which we designed this defensive measure to protect. This indicates that our asymmetric cryptography successfully protected the confidentiality of symmetric encryption keys, which in turn protected the confidentiality of messages so that attackers who intercepted encrypted packets were unable to decrypt them and view the contents. In the future, we would improve side-channel resistance so that keys are not leaked through power, timing, or other stats.

Attack Write-Up: Unkeyed Hashing in an Adversarial Environment

One design we encountered used a cryptographic hash function to ensure message integrity. When sending an encrypted message, a device calculates its SHA-256 sum and includes this value in the packet. When the message is received, the SHA-256 calculation is performed again by the recipient, and its output is compared to the provided hash. If the message has become corrupted in transmission, the two values will not match, and the recipient can discard the corrupted packet. This method successfully protects against incidental corruption during transmission.

However, during the Embedded Capture-The-Flag Competition, devices were placed in an *adversarial* environment. Attackers played the role of "man in the middle" with the ability to intercept and maliciously modify packets during transmission. To prevent devices from encountering an error and entering recovery mode, the communications controller must reject any packets which have been modified during transmission. Therefore, our goal as attackers was to convince the controller to accept a maliciously modified packet.

Packets are accepted as long as the hash calculation on the encrypted message matches the supplied hash. If the attackers can perform the hash calculation on their own, they can determine the correct value to send to the communications controller in order to convince it to accept the packet as valid. The attackers shall attach this calculated hash to their encrypted message and send the whole thing to the communications controller. Indeed, an attacker may easily calculate the SHA-256 hash of arbitrary values, including a maliciously constructed value.

The following code snippet demonstrates the strategy which a man-in-the-middle attacker could use to exploit this vulnerability:

```
          packet = input()
ciphertext, tag = parse_packet(packet)
   malicious_ct = evil_modify(ciphertext)
        new_tag = calc_hash(malicious_ct)
        payload = form_packet(malicious_ct, new_tag)
          print(payload)
```

By implementing this attack in our man-in-the-middle script, we successfully captured a recovery mode flag, demonstrating that a corrupted message was accepted as valid by the communications controller and was passed on to the CPU.

We conclude that simply computing the cryptographic hash of encrypted messages is not enough to ensure message integrity. Instead, a proper Message Authentication Code, or MAC, should be used. This can be constructed by taking a secret value which the attacker does not know as input. Then, the critical weakness that "attackers can perform the calculation on their own" is remedied. Attackers do not have the requisite information to forge a valid message tag.

One such construction is known as HMAC. Essentially a keyed hashing operation, this method relies on a cryptographic hash function such as SHA-256 and is easy to implement on any device which already provides SHA-256 functionality. The two authorized parties must first establish a shared secret HMAC key, which can be accomplished in a variety of ways. As long as the key is kept secret from the attackers, they will be unable to produce a valid message tag to accompany maliciously modified messages, so message integrity can be successfully enforced.