MITRE eCTF 2024
Team Spartans

Michigan State University

Udbhav Saxena, Felipe Marques Allevato, Charles Selipsky, Riley Cook, Aashish Harishchandre, Aditya Chaudhari, Fatima Saad, Samay Achar, Krishna Patel, Ramisa Anjum, Radhe Patel

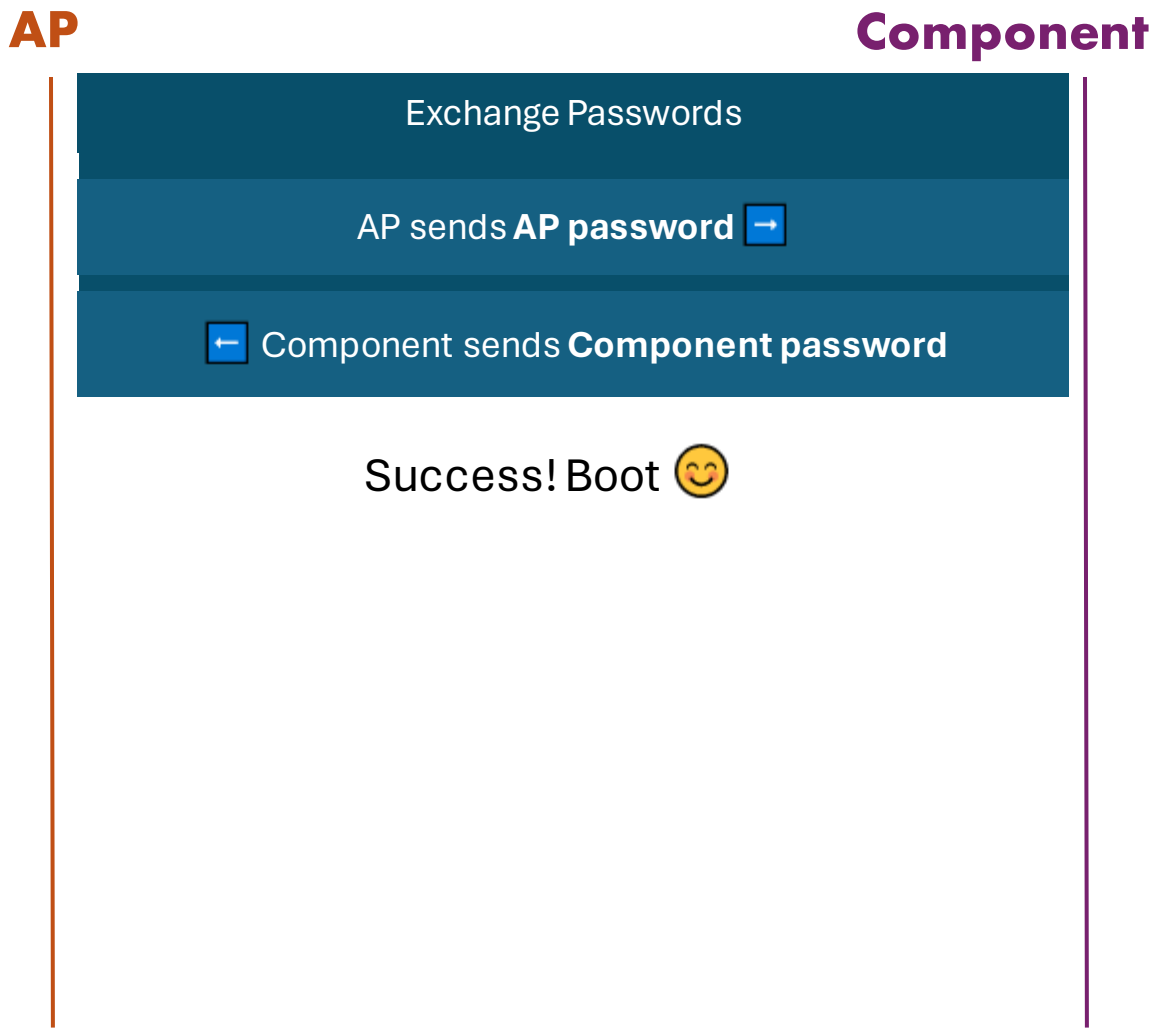
Building Up Our Design, One Step at a Time

Goal: Create a secure medical system composed of an **Application Processor (AP)** and **Components** that boots only when all devices are genuine and from the manufacturer.

After booting, create a secure channel for communication between the AP and components, ensuring the **integrity** and **authenticity** of messages.

Idea 1

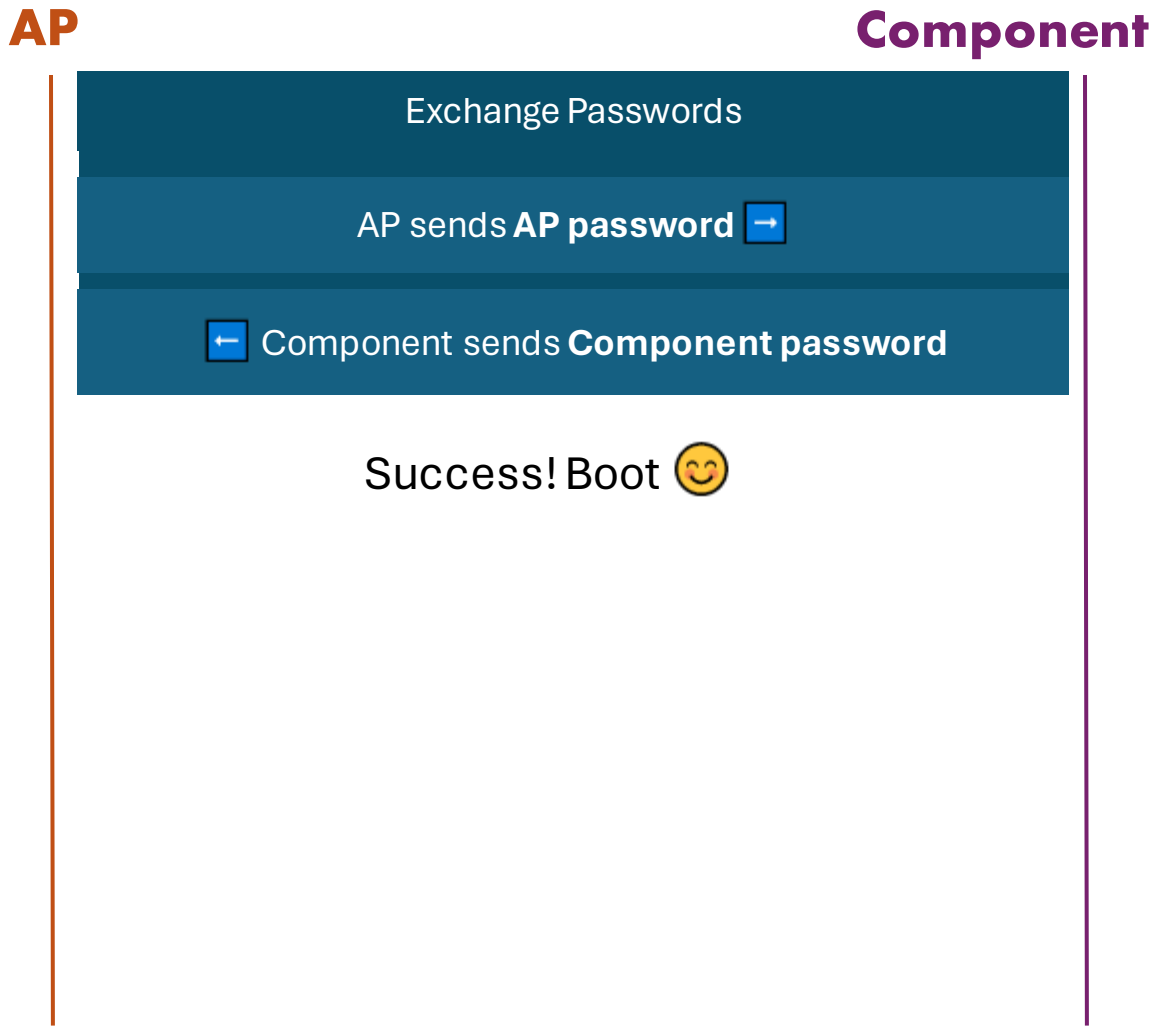
Let's use a **password** on either side, one for the AP and one for the Component. On a boot command, the AP and component simply share their password to verify each other and then boot.



Idea 1

Problem: A malicious third party can initiate a boot command with a fake component, make the AP share the password, then reuse it as a fake AP to make a component boot.

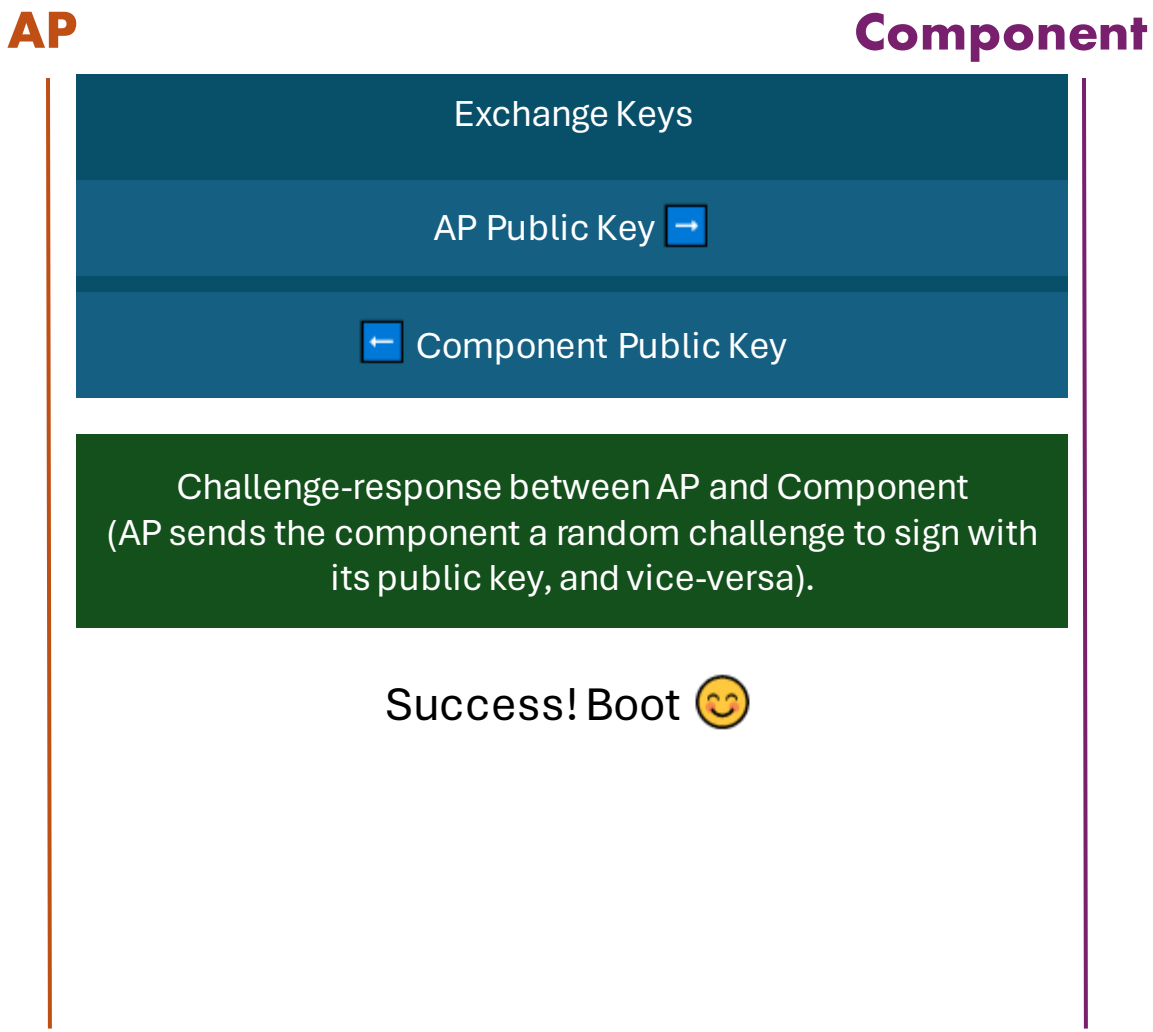
This was an attack we performed against a team that used this design.



Idea 2

Use **public key cryptography** along with **challenge-response**. Every AP and Component gets a keypair made up of a **Public Key** and a **Private Key**.

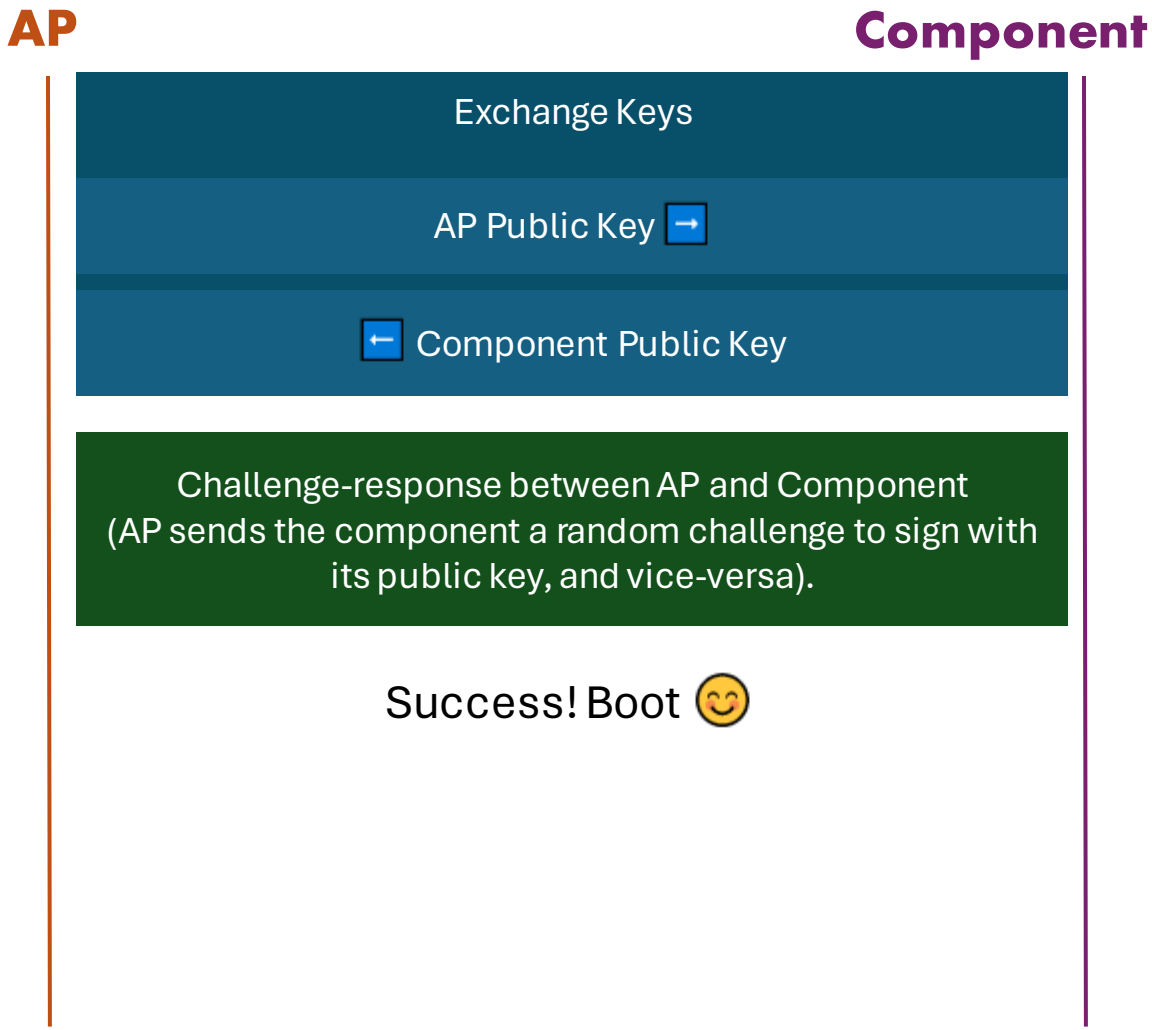
They exchange their Public Keys, and then sign a randomly generated challenge from the other party with their private key. Since the challenge is randomly generated each time, this makes sure that the other end owns the key since responses from older exchanges cannot be reused.



Idea 2

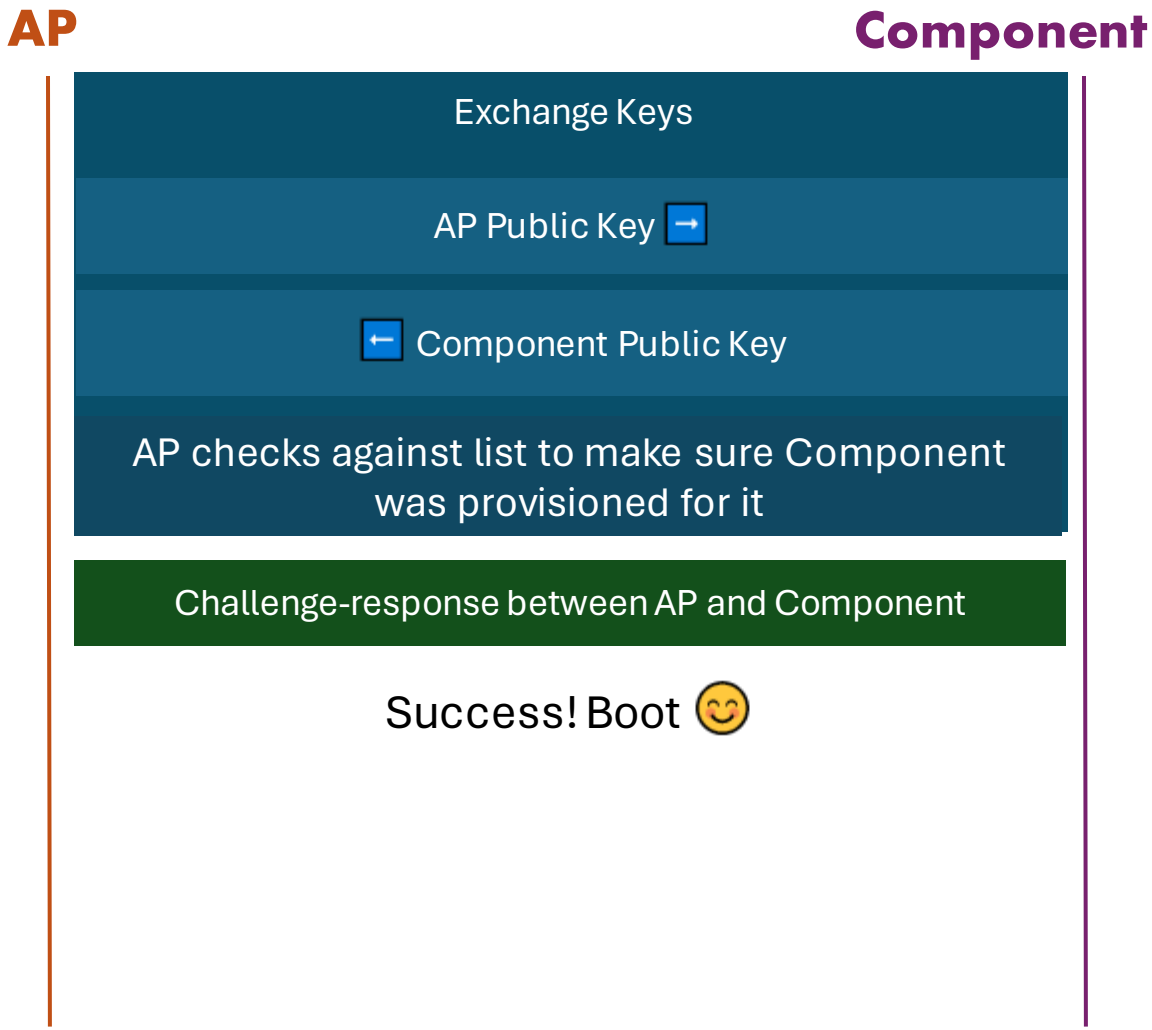
Problem: This design doesn't make sure that the APs and Components are actually from the manufacturer! Any device participating in the handshake, as long as it presents a random keypair, will cause a successful boot.

This was an attack we successfully executed against a team, and the only step required was to flash a board with their source code as-is and boot the MISC.



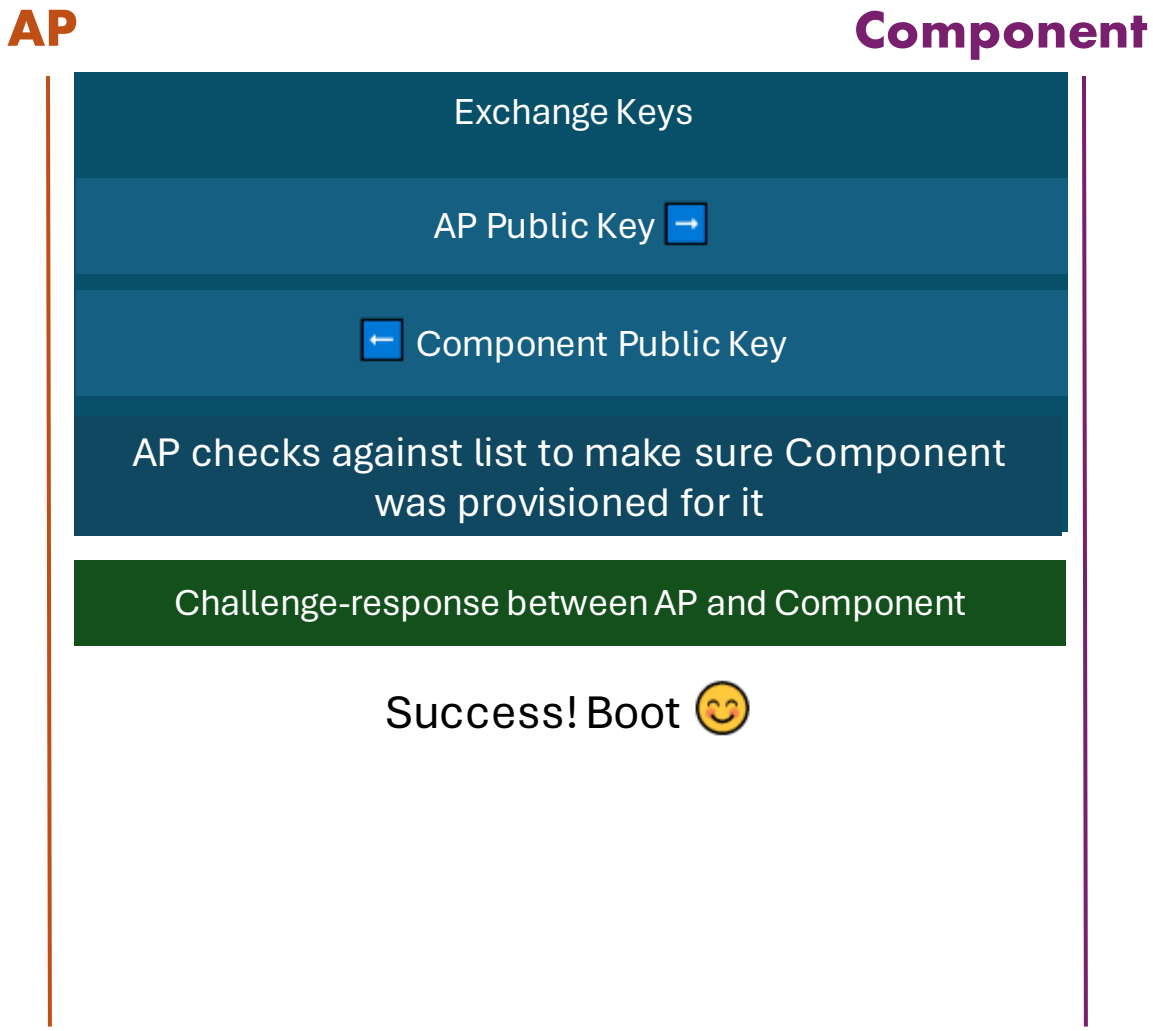
Idea 3

Have the AP store the public keys of all provisioned components when built, so that on a boot attempt it can check the provided key against the list and make sure that it is legitimate.



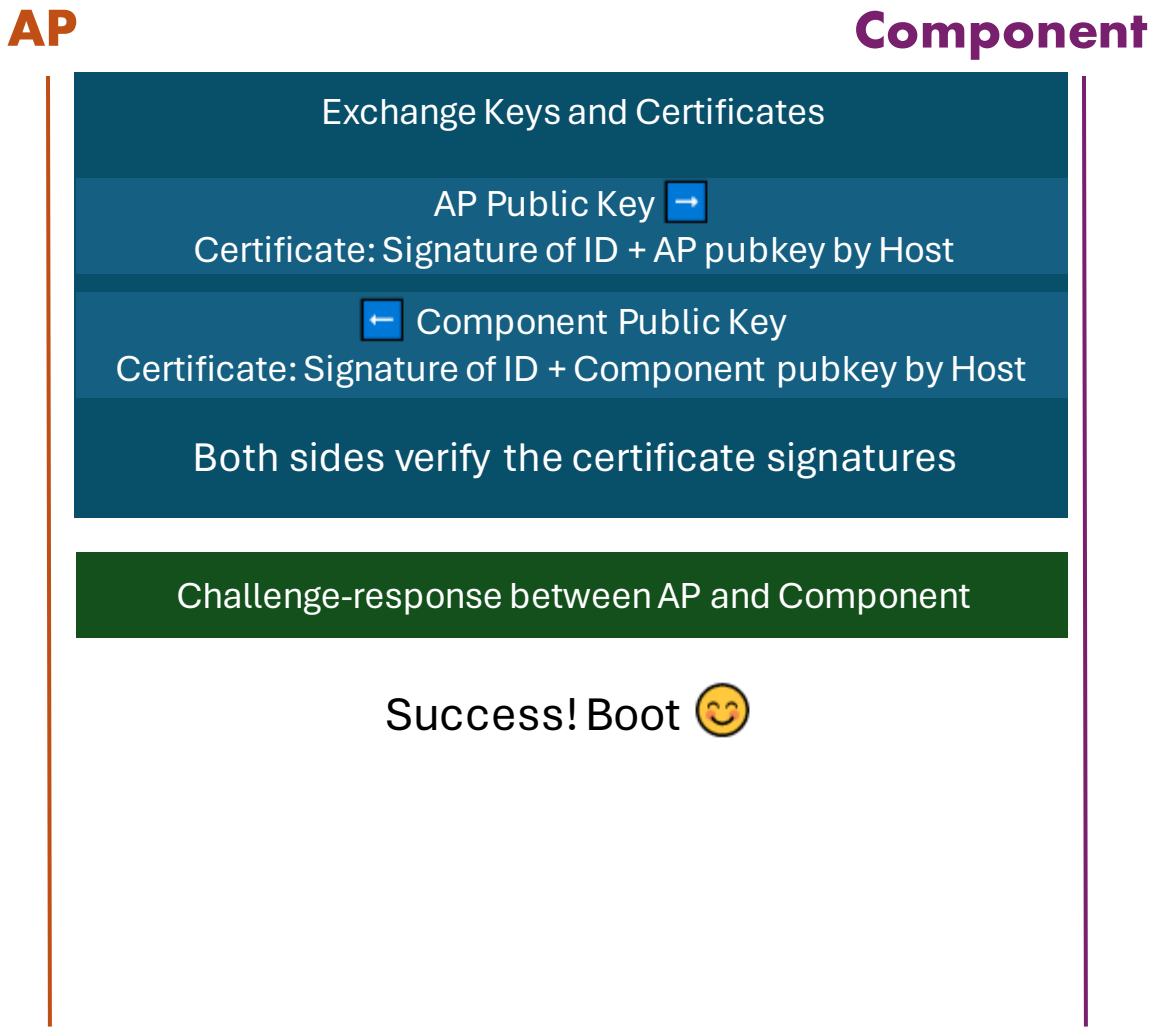
Idea 3

Problem: This design is not possible with the infrastructure we have in the competition! A MISC can be built in any order, and an application processor can have an arbitrary number of components built for it even after the AP has already been built. Thus, it is not possible to know all the possible component keys in advance when building the AP.



Idea 4

Use **certificates**. We can give the manufacturer (or the **host system**) its own keypair stored in the host secrets. Any AP or Component that is built will get its own keypair, but it will also get an additional packet of data which would be a combination of the device's identifier + the device's public key cryptographically signed by the host's secret key, which we call a **certificate**. Then, we also give all devices the host's public key so that they can verify these signatures before booting.



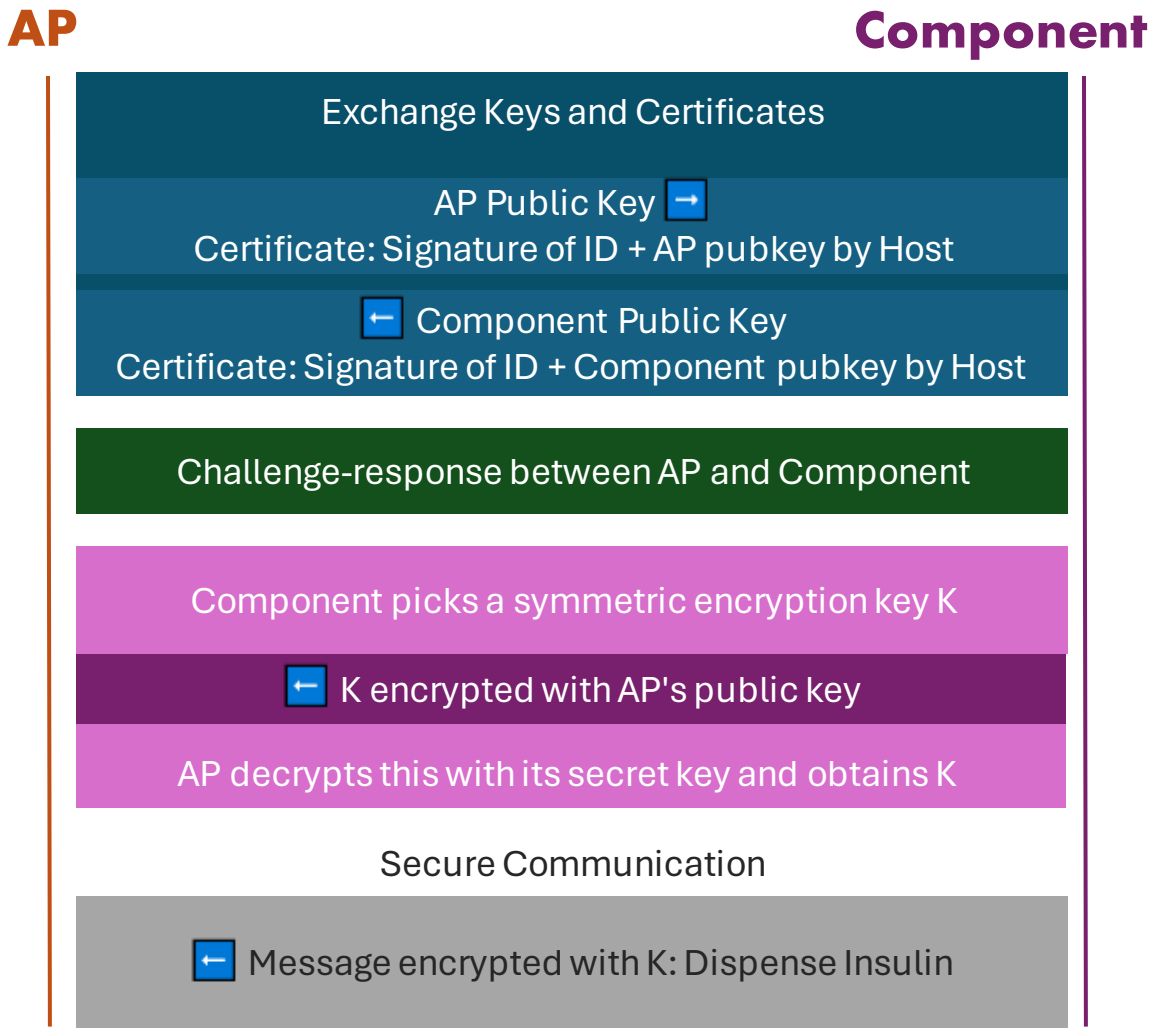
New Problem: Secure Communications

After booting, an AP needs to establish a secure channel of communication with all components where both sides can ensure **integrity** and **authenticity** of messages. Public key cryptography becomes inefficient for continuously sharing large messages, so it is preferred to do this using symmetric encryption. We can use **authenticated encryption** (ChaCha20-Poly1305 in our case) to be able to detect if message packets are tampered by a third party, in addition to providing confidentiality.

This now presents a problem of sharing a symmetric key between the devices.

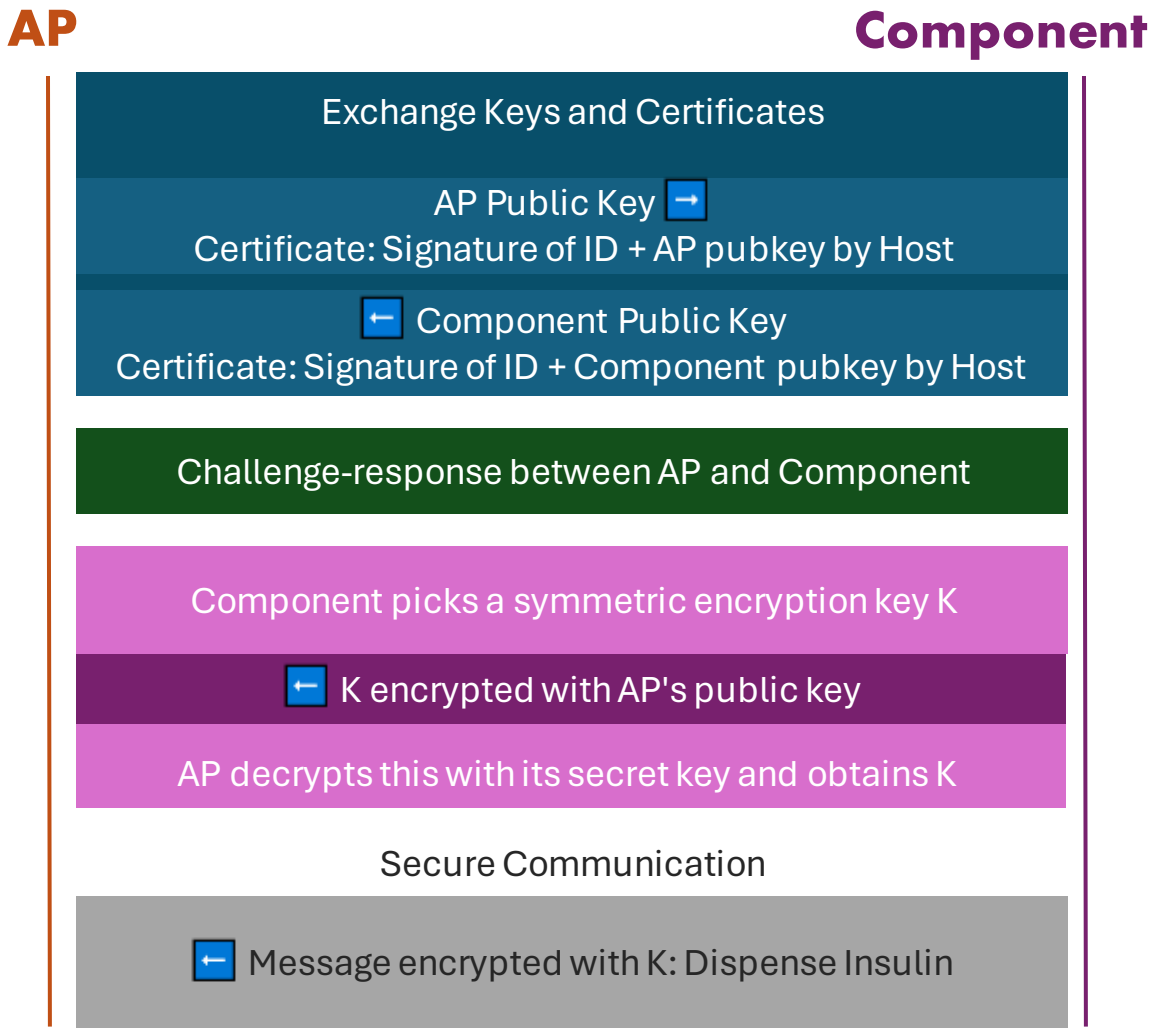
Idea 5

Since we've already verified the public keys between devices, one side (e.g., component) can simply choose a random encryption key, encrypt it using the other party's (AP's) public key, and send it across. This ensures that only the AP could decrypt this with its secret key, and then continue to use it for symmetric encryption.



Idea 5

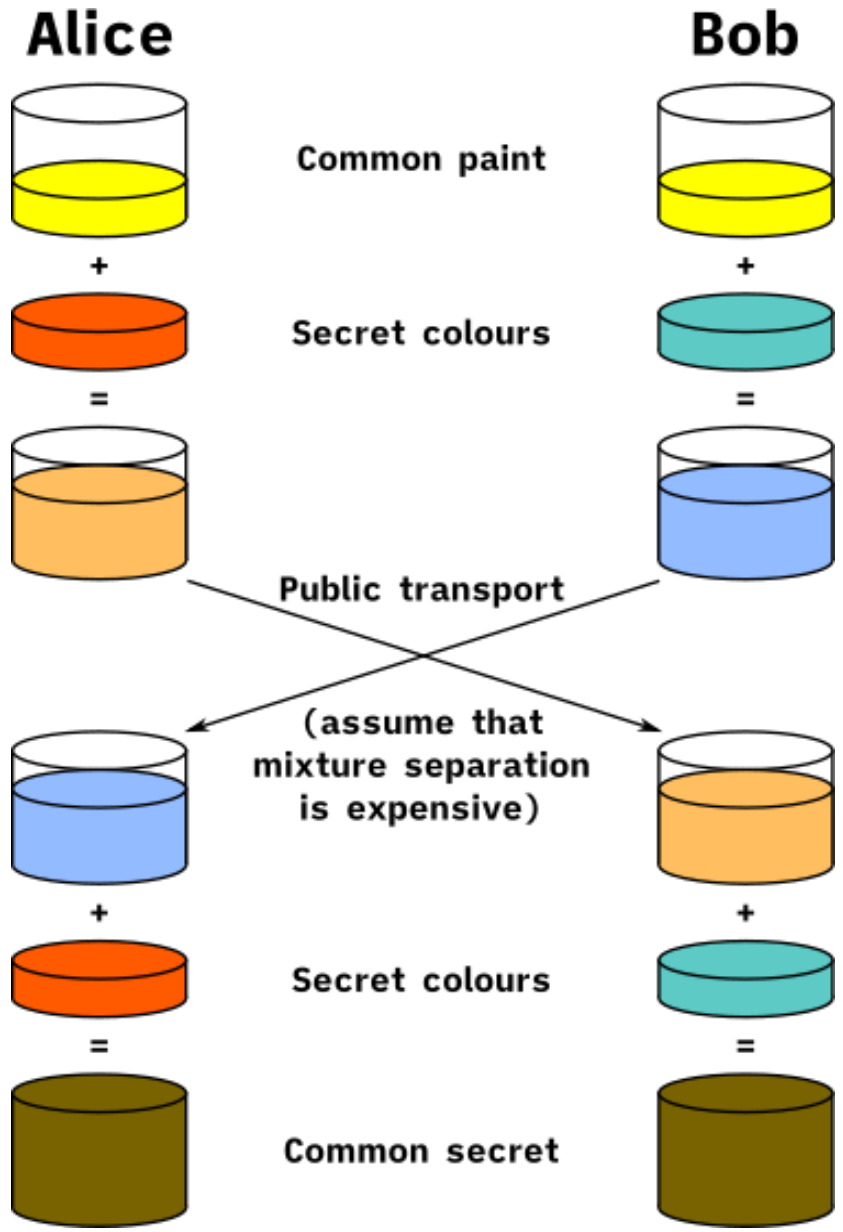
Problem: Hinges the security of all communications on the component's private key remaining a secret. An attacker could collect and store all communications between the AP and component. If there is a compromise of the AP's keys in the future, the attacker can go back and decrypt the packet sent by the component.



Idea 6

Use a **Diffie-Hellman Key Exchange** with ephemeral keys. In our design, both the AP and Component generate pairs of random **Curve25519** keys during the handshake. They then exchange the public points and combine them such that they both arrive at the same result, and no outside party observing this exchange can figure out the same key. This shared result is then used as the secret key for symmetric encryption, and the keys are then discarded after the session end (which is why they are called ephemeral).

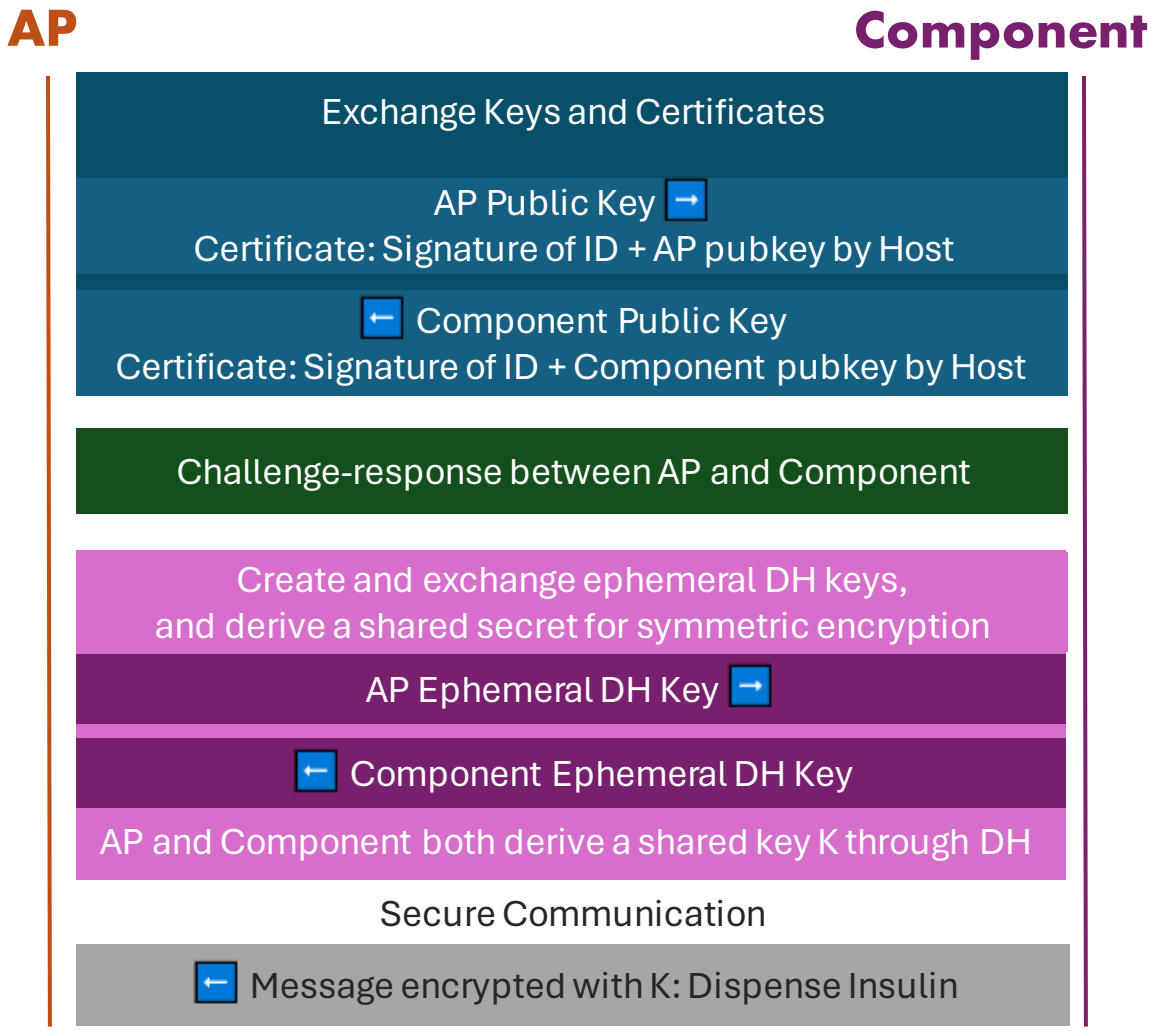
This offers perfect **forward-secrecy** of communications - which means that all communications are secure against a future compromise of either party's keys.



Idea 6

Use a **Diffie-Hellman Key Exchange** with ephemeral keys. In our design, both the AP and Component generate pairs of random **Curve25519** keys during the handshake. They then exchange the public points and combine them such that they both arrive at the same result, and no outside party observing this exchange can figure out the same key. This shared result is then used as the secret key for symmetric encryption, and the keys are then discarded after the session end (which is why they are called ephemeral).

This offers perfect **forward-secrecy** of communications - which means that all communications are secure against a future compromise of either party's keys.



An Unaddressed Flaw

We've ensured that messages between boards are encrypted and cannot be modified using authenticated encryption, but there is still a vulnerability that could lead to malfunction.

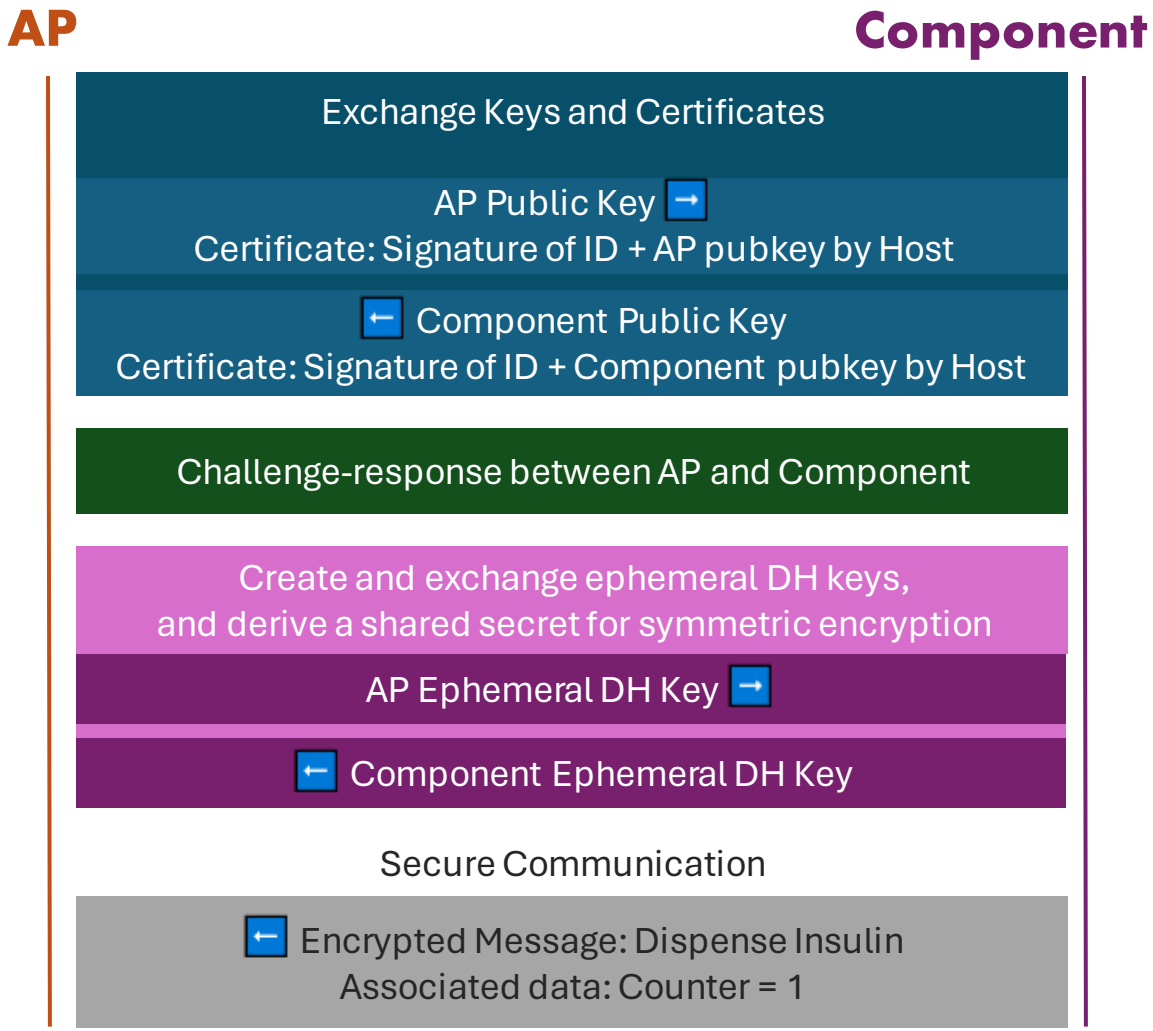
An attacker could mess with the functionality of an insulin pump component by receiving packets sent from an AP post-boot and resending it multiple times, causing it to dispense a dangerous amount of insulin. These packets would be valid and encrypted with the proper keys, but the component would have no way to know that they were only meant to be received once instead of multiple times!

This was an attack against a few teams that we unfortunately did not capture in time, as it required building a MITM board that could act as both an I2C controller and peripheral ☹

Idea 7

To prevent this, we can encrypt a **counter** with each message and keep track of it on the receiving end. The counter increments for every message sent, and we make sure to reject any packets that are below the latest value of the counter.

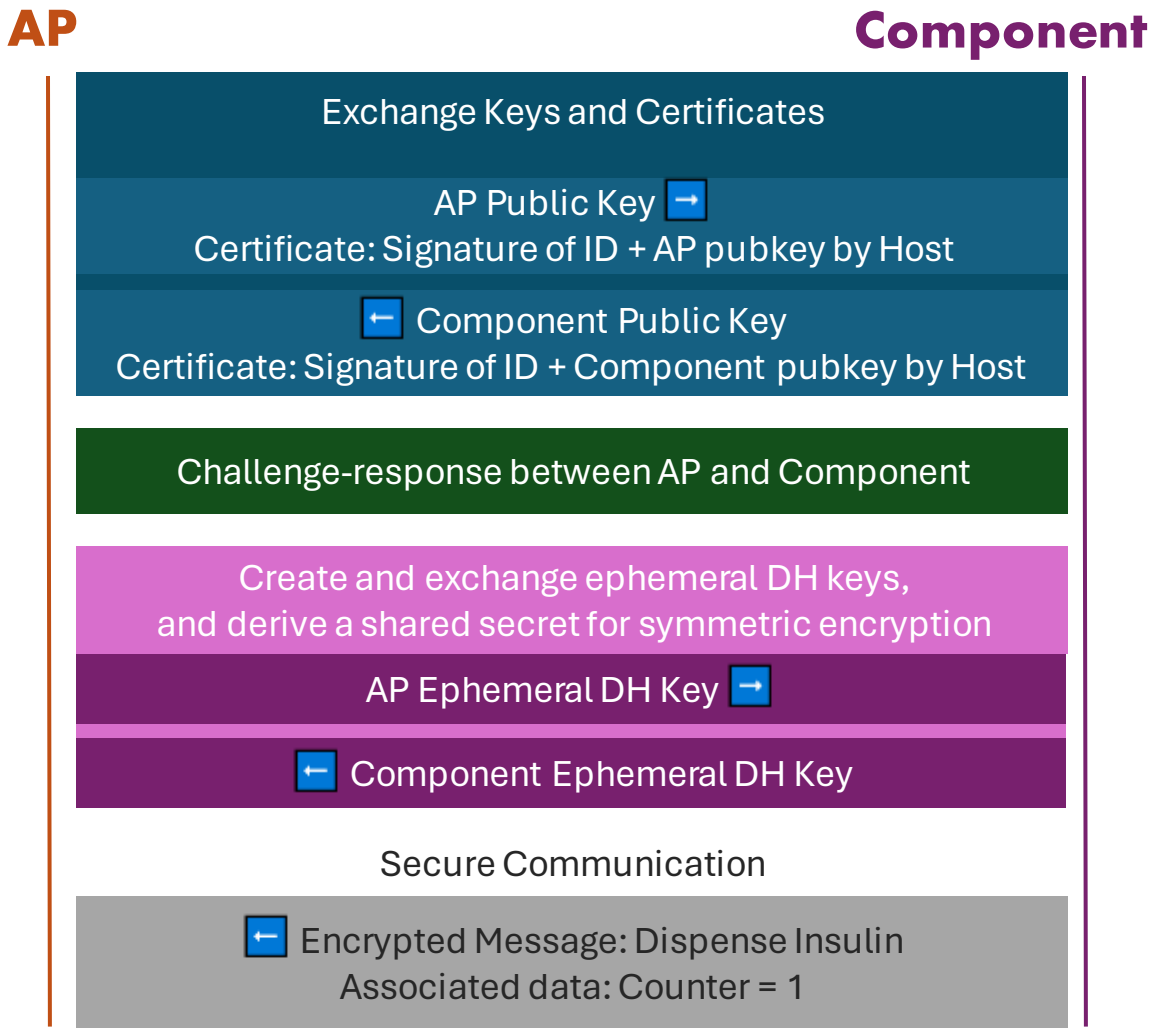
We can do this by adding **associated data** to the authenticated encryption (ChaCha20-Poly1305 in our case), using the counter value as additional data that is attached to every packet. If a device receives a packet that has an old or repeated value of counter, it refuses to accept it. This protects us against replay attacks.



Final design!

This is a simplified version of the cryptography in our final design.

In the implementation, some parts were condensed to make the handshake more efficient, such as using the ephemeral Diffie-Hellman keys as the unique nonces to be signed for the challenge-response, and sending them as part of the initial message including the public key and certificate.



Performance

MichState	75	88	1338	160	91	89	89

Leaves a lot to be desired but we successfully defended one flag!

Aiming to defend (and attack) more next year :)

the one singular unexploited flag hanging on for life

Thank You!

Sources

Diffie-Hellman Visualization

https://upload.wikimedia.org/wikipedia/commons/thumb/4/46/Diffie-Hellman_Key_Exchange.svg/375px-Diffie-Hellman_Key_Exchange.svg.png