Rusty Replays

Or: How I learned to stop worrying and love Rust and GPTMs

1.1 Defensive section

The C language, while mature and well-supported, uses unsafe paradigms by default, and many of the cryptographic softwares which support C for use in embedded systems are archaic, not ergonomic, or have known side-channel attacks. We expected (and confirmed) that many teams would misuse C implementations of cryptographic software and would simply have subtle errors which would allow for exploitation.

To mitigate these expected problems, we implemented our controller in Rust and minimised the use of inherently unsafe code while maximising reliance on well-reviewed crates. The embedded Rust community has developed a large codebase for safe development of embedded software. Similarly, the cryptographic Rust community has developed a large codebase of well-reviewed (sometimes the industry standard) implementations of cryptographic algorithms; all of which are compatible with embedded Rust and many of which have side-channel mitigations built in. This, along with enforcement of correct usage guaranteed by the Rust compiler, reduced our likelihood of misusing these libraries.

While we certainly weren't invulnerable to all attacks, we were not compromised due to misuse of cryptographic libraries, separating us from many other teams. Unfortunately, we had a major, yet subtle, issue introduced due to a failure to match our specification of our counter-based replay prevention. Testing using Rust's built-in test configuration would have discovered this issue and allowed us to fix it before releasing our code to the attackers. Additionally, use of tools such as Galois' Crucible would allow us to formally check for logic errors with little extra configuration, as it supports Rust code out of the box. In short, with more work on formal specification of our solution—and with some of the features Rust provides for us—we could have greatly improved our defensive stance in this competition.

1.2 Offensive section

A relatively concise replay attack proved effective for most designs. Given that all UAVs start at the same altitude, we can choose a UAV for which all location broadcasts will be blocked and saved inside a data structure in our man-in-the-middle script (typically the first UAV launched). While saving the broadcasts from the target, our script also drops any other broadcasts sent out from other UAVs to keep all drones at the initial altitude of 100. After collecting 10-15 location broadcasts from the target UAV, we send all of them along the network, causing all other UAVs currently flying at the initial altitude of 100 to reply with deconflict messages.

Because we send multiple broadcasts (previously undelivered) from the target UAV, other receiving UAVs continue to respond, as if the target UAV is still at the initial altitude. If the team implemented a simple message counter, there is no message timeout, and replayed broadcasts are accepted. This resulted in multiple deconflict responses, forcing the target UAV above its flight ceiling.

Many teams implemented simple broadcast and direct message counters, which work by tracking the number of messages received from a given UAV, then comparing this value to a given counter value appended to a message. This approach is suitable for preventing replay of any single message, but is ineffective at preventing attackers from suppressing messages to be used against the network at a later time.

Furthermore, it is difficult to insulate a network of embedded systems from this attack given that no operating system exists for each network node to track elapsed time. Without the use of a real-time clock, it may be prohibitively difficult to protect against this attack for a team with limited time and resources.

A proposed solution would integrate some means for any network node (in this case, UAV) to evaluate time elapsed during message receipt. Such an approach might be to implement a system counter on the SSS, whose value is distributed to each SED which then increments this at constant intervals across the network. By using the device's General-Purpose Timers, detailed on page 334 of the chip specification, one can use the real-time clock present in the hardware to synchronise time between devices as initialised by the SSS at registration time. Then, checking a range around this time as deemed appropriate by experimentation, one would be able to verify that a message was sent at a reasonably recent time and thus mitigate time-based replay attacks.